Microsoft

Программируем Windows® Phone 7

Чарльз Петзольд

ОПУБЛИКОВАНО

Microsoft Press

A Division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2011 by Microsoft Corporation

Все права защищены. Ни одна часть данной книги не может быть воспроизведена или использована в какой-либо форме или каким-либо образом без предварительного письменного разрешения издателя.

Контрольный номер библиотеки конгресса (LCCN): 2010939982

ISBN: 978-0-7356-4335-2

Книги издательства Microsoft Press доступны по каналам оптовых и розничных продаж по всему миру. Для получения дополнительной информации о переводных изданиях обратитесь в местное отделение Корпорации Майкрософт или свяжитесь непосредственно с международным отделом Microsoft Press International по факсу (425) 936-7329. Комментарии по данной книге можно оставить по адресу http://www.microsoft.com/learning/booksurvey.

Microsoft и торговые марки, перечисленные в документе http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EMUS.aspx, являются торговыми марками группы компаний Майкрософт. Все остальные торговые марки являются собственностью соответствующих владельцев.

Все приводимые здесь в качестве примера компании, организации, продукты, доменные имена, адреса электронной почты, логотипы, персоналии, адреса и фамилии являются вымышленными и не имеют никакой связи с реальными компаниями, организациями, продуктами, доменными именами, адресами электронной почты, логотипами, персоналиями, адресами или событиями.

Данная книга представляет мнения и взгляды ее автора. Сведения, содержащиеся в книге, предоставляются без каких-либо выраженных, установленных или подразумеваемых гарантий. Ни авторы, ни корпорация Майкрософт, ни дистрибьюторы и распространители не несут никакой ответственности за любой урон, предполагаемый или нанесенный данной книгой прямо или косвенно.

Cover: Tom Draper Design

Содержание

СОДЕРЖАНИЕ **3**

ВВЕДЕНИЕ **9**

Организация........ 9
Требуемые навыки 10
Системные требования 10
Использование эмулятора телефона 10
Примеры кода 11
В последний момент........ 11
Благодарности 11
Список опечаток и поддержка книги 12
Нам интересно ваше мнение 12
Оставайтесь «на связи» 13

ЗДРАВСТВУЙ, WINDOWS PHONE 7 .... **15**

Разработка для Windows Phone 7 15
Аппаратные средства........ 17
Датчики и сервисы 19
File | New | Project........ 20
Первое приложение для телефона на Silverlight 20
Стандартные файлы Silverlight 23
Цветовые темы 29
Пункты и пикселы........ 30
XAP – это ZIP 32
Приложение для телефона на XNA 32

ОРИЕНТАЦИЯ **39**

Silverlight и динамическая компоновка 39
События изменения ориентации экрана........ 45
Ориентация в приложении на XNA 46
Простые часы (*очень* простые часы) 49

ОСНОВЫ РАБОТЫ С СЕНСОРНЫМ ВВОДОМ **54**

Обработка простого касания в XNA........ 54
Обработка жестов в XNA........ 57
События простого касания в Silverlight........ 59
События Manipulation 62
Маршрутизированные события 65
Странное поведение? 66

РАСТРОВЫЕ ИЗОБРАЖЕНИЯ ИЛИ ТЕКСТУРЫ **67**

Создание текстуры на XNA 68
Элемент *Image* в Silverlight 69
Изображения из Интернета........ 71
Image и ImageSource........ 73
Загрузка хранящихся локально растровых изображений из кода........ 75
Захват изображения с камеры 76

Библиотека фотографий телефона 79

Датчики и службы 83

Акселерометр 83
Простой уровень нивелира 88
Географические координаты 92
Использование картографического сервиса 95

Вопросы архитектуры приложений 102

Реализация простейшей навигации 102
Передача данных на страницы 108
Совместное использование данных страницами 109
Хранение данных вне экземпляров 114
Идеал многозадачности 116
Переключение задач в телефоне 116
Состояние страницы 118
Изолированное хранилище 121
Захоронение и параметры для приложений на XNA 124
Тестирование и эксперименты 129

Мощь и слабость XAML 131

TextBlock в коде 132
Наследование свойств 134
Синтаксис свойство-элемент 135
Цвета и кисти 136
Содержимое и свойства содержимого 142
Коллекция ресурсов 144
Совместное использование кистей 145
x:Key и *x:Name* 148
Введение в стили 149
Наследование стилей 150
Темы 151
Градиент 152

Элементы и свойства 154

Основные фигуры 154
Трансформации 155
Анимация со скоростью видео 162
Обработка событий манипуляций 164
Элемент *Border* 165
Свойства и строковые элементы *TextBlock* 168
Изображения более подробно 170
Воспроизведение фильмов 173
Режимы прозрачности 173
Мозаичные кисти, не создающие мозаики 175

Вопросы компоновки 177

Grid с одной ячейкой 178
Элемент *StackPanel* 179
Конкатенация текста с помощью *StackPanel* 182
Вложенные панели 184
Видимость и компоновка 185
Два примера использования *ScrollViewer* 187

Механизм компоновки 192
Панель, взгляд изнутри 194
Клон *Grid* с одной ячейкой 195
Пользовательский вертикальный *StackPanel* 198
Старомодный *Canvas* 200
Canvas и *ZIndex* 205
Canvas и сенсорный ввод 206
Grid всемогущий 207

ПАНЕЛЬ ПРИЛОЖЕНИЯ И ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 210

Значки *ApplicationBar* 210
Jot и параметры приложения 217
Jot и сенсорный ввод 221
Jot и *ApplicationBar* 223
Элементы и элементы управления 227
RangeBase и *Slider* 229
Простой *Button* 235
Концепция свойства *Content* 238
Стили темы и приоритетность 241
Иерархия класса Button 242
Реализация секундомера 244
Кнопки и стили 253
TextBox и ввод с клавиатуры 254

СВОЙСТВА-ЗАВИСИМОСТИ 263

Описание проблемы 263
В чем отличие свойств-зависимостей 266
Наследование от *UserControl* 275
Новый тип переключателя 283
Панели и свойства 288
Присоединенные свойства 293

ПРИВЯЗКА ДАННЫХ 298

Источник и цель 298
Цель и режим 300
Конвертеры привязок 302
Относительный источник 307
Источник «this» 307
Механизмы уведомления 310
Простой сервер привязки 311
Задание *DataContext* 316
Простые решения 321
Конвертеры со свойствами 325
Передача и прием 328
Обновления привязок *TextBox* 333

ВЕКТОРНАЯ ГРАФИКА 344

Библиотека *Shapes* 344
Canvas и *Grid* 345
Перекрытие и *ZIndex* 347
Полилинии и произвольные кривые 348
Наконечники, соединения и пунктир 353
Многоугольник и заливка 358

Свойство *Stretch* ... 361
Динамические многоугольники ... 361
Элемент *Path* ... 364
Геометрические элементы и трансформации ... 369
Группировка геометрических элементов ... 373
Универсальный *PathGeometry* ... 374
Класс *ArcSegment* ... 376
Кривые Безье ... 382
Синтаксис разметки контура ... 390
Как создавалась данная глава ... 394

РАСТРОВАЯ ГРАФИКА ... 400

Иерархия класса Bitmap ... 400
WriteableBitmap и *UIElement* ... 402
Работа с пикселами ... 408
Векторная графика в растровой матрице ... 412
Изображения и захоронение ... 416
Сохранение в библиотеку изображений ... 424
Приложение расширений для обработки фотографий ... 430

АНИМАЦИИ ... 439

Сравнение анимации, основанной на кадрах, и анимации, использующей временную шкалу ... 439
Цели анимации ... 442
Щелчок и разворот ... 443
Некоторые вариации ... 446
Анимации, описанные в XAML ... 450
Поучительная история ... 452
Анимация по ключевым кадрам ... 458
Триггер по событию Loaded ... 461
Анимация присоединенных свойств (или нет) ... 468
Сплайны и ключевые кадры ... 472
Проблема прыгающего мяча ... 479
Функции сглаживания ... 483
Анимация трансформации перспективы ... 488
Анимации и приоритетность свойств ... 493

ДВА ШАБЛОНА ... 497

ContentControl и *DataTemplate* ... 497
Анализ дерева визуальных элементов ... 501
Основы *ControlTemplate* ... 505
Диспетчер визуальных состояний ... 514
Совместное и повторное использование стилей шаблонов ... 522
Библиотека пользовательских элементов управления ... 525
Вариации на тему *Slider* ... 530
Такой необходимый *Thumb* ... 538
Пользовательские элементы управления ... 541

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ СПИСКАМИ ... 547

Элементы управления списками и деревья визуальных элементов ... 548
Настройка представления элементов ... 554
Выбор в *ListBox* ... 557
Привязка к *ItemsSource* ... 561
Базы данных и бизнес-объекты ... 566

Эти замечательные шаблоны данных 580
Сортировка 582
Замена панели 587
Построение гистограммы при помощи *DataTemplate* 589
Картотека 595

СВОДНОЕ ПРЕДСТАВЛЕНИЕ И ПАНОРАМА 605

Сходства и отличия 605
Сортировка коллекции музыкальных произведений по композитору 615
Подключение XNA 618
Музыкальные классы XNA: MediaLibrary 620
Вывод альбомов на экран 625
Музыкальные классы XNA: *MediaPlayer* 629

ПРИНЦИПЫ ДВИЖЕНИЯ 636

Простейший подход 636
Краткий обзор векторов 638
Перемещение спрайтов с помощью векторов 642
Параметрические уравнения 644
Возможности, обеспечиваемые функцией преобразования 647
Изменение размера текста 648
Два приложения, реализующие вращение текста 651

ТЕКСТУРЫ И СПРАЙТЫ 657

Разновидности метода *Draw* 657
Еще одно приложение «Здравствуй, Мир»? 659
Перемещение по заданной траектории 663
Перемещение вдоль полилинии 666
Эллиптическая траектория 670
Обобщенное решение для перемещения по кривой 673

ДИНАМИЧЕСКИЕ ТЕКСТУРЫ 677

Целевой объект прорисовки 677
Сохранение содержимого целевого объекта прорисовки 685
Отрисовка линий 688
Работа с пикселами 695
Геометрия рисования прямых линий 698
Изменение существующих изображений 708

ОТ ЖЕСТОВ К ТРАНСФОРМАЦИЯМ 712

Жесты и свойства 712
Масштабирование и вращение 715
Трансформации матриц 722
Жест *Pinch* 726
Скольжение и инерция 732
Множество Мандельброта 734
Панорамирование и масштабирование 744
Компоненты игры 749
Аффинные и неаффинные преобразования 753

ИСПОЛЬЗОВАНИЕ СЕНСОРНОГО ВВОДА В ИГРОВЫХ ПРИЛОЖЕНИЯХ 761

Еще больше компонентов игры 761
Холст PhingerPaint 765

Небольшой обзор SpinPaint 774
Код SpinPaint 776
Процесс рисования 780
PhreeCell и колода карт 784
Игровое поле 785
Play и Replay 792

ПРИМЕНЕНИЕ АКСЕЛЕРОМЕТРА В ИГРОВЫХ ПРИЛОЖЕНИЯХ 804

Трехмерные векторы 804
Улучшенная визуализация «пузырька» 807
Графическое представление 814
Следуй за катящимся шаром 821
Лабиринт 831

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 842

ОБ АВТОРЕ 907

Введение

Данная книга является подарком от группы разработки Windows Phone 7 в Майкрософт всему сообществу разработчиков, и я испытываю чувство гордости из-за своей причастности к этому. В книге я продемонстрирую основы написания приложений для Windows Phone 7 с использованием языка программирования C# и инфраструктур Silverlight и двухмерного XNA.

Да, «*Программируем Windows Phone 7*» свободно доступна в Интернете для скачивания, но для тех читателей, которые остаются верны бумажным изданиям – одним из них являюсь и я – предлагается данная книга. Она разделена на два тома, «*Программируем Windows Phone 7 в Microsoft Silverlight*» и «*Программируем Windows Phone 7 в Microsoft XNA Framework*», оба снабжены полным предметным указателем**.**

Те деньги, которые вы сэкономили, загрузив данную книгу бесплатно, пожалуйста, потратьте на другие книги. Несмотря на обилие информации в Интернете, книги по-прежнему остаются лучшим способом изучения методик разработки, предоставляя последовательное и единообразное повествование. Каждый проданный экземпляр вызывает слезы радости у автора, так пускай текут полноводные реки слез авторского счастья!

В частности, я рекомендую приобрести книги, *дополняющие* материалы данной книги. Например, здесь я практически не касался Веб-сервисов, и это является серьезным недостатком, потому что Веб-сервисы будут приобретать все большее значение в приложениях Windows Phone 7. Мой обзор XNA ограничивается двухмерной графикой. Я надеюсь добавить в следующее издание этой книги несколько глав, посвященных 3D графике, но не собираюсь погружаться во все тонкости разработки игровых приложений для Xbox. И я также не рассматриваю здесь никаких других средств разработки, кроме Visual Studio (даже не обсуждаю Expression Blend).

Издательство Microsoft Press готовит к выпуску несколько дополнительных книг по Windows Phone 7. «*Windows Phone 7 Silverlight Development Step by Step*» [1], авторами которой являются Энди Вигли (Andy Wigley) и Питер Фут (Peter Foot), предлагает подход, более ориентированный на инструментальные средства. Несмотря на то что книга Майкла Стро (Michael Stroh) «*Windows Phone 7 Plain & Simple*» [2] – это больше руководство по *использованию* телефона, а не по разработке приложений для него, я уверен, разработчики найдут почерпнут из нее некоторое понимание и идеи.

Более того, до меня дошли слухи, что мой старый приятель Дуг Болинг (Doug Boling) трудится над книгой по разработке корпоративных приложений для Windows Phone 7. Это должен быть шедевр, поэтому не забудьте приобрести его для своей библиотеки.

Организация

Данная книга разделена на три части. Первая часть обсуждает основные концепции программирования для Windows Phone 7, используя для примеров приложения как на Silverlight, так и на XNA. Преимущественное большинство разработчиков для Windows Phone 7 выберут одну из платформ, но, я думаю, важно, чтобы все разработчики имели хотя бы *базовое* представление об альтернативном варианте.

Вторая часть книги полностью посвящена Silverlight, и третья – двухмерному XNA. Для удобства читателей все главы построены на знаниях, полученных в предыдущих главах, и

---

[1] Разработка для Windows Phone 7 на Silverlight,шаг за шагом *(прим. переводчика).*

[2] Windows Phone 7, легко и просто *(прим. переводчика).*

образуют последовательное обучающее повествование. Поэтому данную книгу надо читать последовательно.

Требуемые навыки

Эта книга предполагает, что читатель знаком с основными принципами .NET-разработки и имеет достаточный опыт работы с языком программирования C#. Тем, кто пока не имеет таких навыков, полезно будет прочитать мою книгу «*.NET Book Zero: What the C or C++ Programmer Needs to Know about C# and the .NET Framework*»[1], которая свободно доступна в сети на моем сайте по адресу www.charlespetzold.com/dotnet.

Системные требования

Чтобы использовать данную книгу надлежащим образом, необходимо загрузить и установить инструменты разработки Windows Phone Developer Tools, которые включают Visual Studio 2010 Express for Windows Phone, XNA Game Studio 4.0 и экранный эмулятор Windows Phone, используемый для тестирования приложений, если реальное устройство недоступно. Все самые последние сведения и загрузки предоставляются по адресу *http://developer.windowsphone.com*.

Эти инструменты можно установить поверх Visual Studio 2010, расширяя возможности Visual Studio 2010 для разработки приложений для телефона. Я использовал именно такую конфигурацию.

Эмулятор телефона предоставляет широкие возможности, но в некоторый момент приложения должны быть развернуты на реальном устройстве, работающем под управлением Windows Phone 7. Зарегистрировавшись по адресу *http://developer.windowsphone.com* как разработчик приложений для телефона, вы получите возможность разблокировать свой телефон, т.е. сможете развертывать на нем свои приложения из Visual Studio.

Приложения, приводимые в данной книге в качестве примеров, я тестировал на телефоне LG GW910. Для информации, последней установленной версией была сборка под номером 7.0.7003.0.

Использование эмулятора телефона

Windows Phone 7 поддерживает мультисенсорный ввод, и обработка мультисенсорного ввода является важной частью разработки приложения для телефона. При использовании эмулятора Windows Phone Emulator перемещения и щелчки кнопки мыши могут моделировать сенсорный ввод, но только от одного касания. По-настоящему протестировать мультисенсорный ввод на эмуляторе позволяет лишь монитор с поддержкой мультисенсорного ввода, работающий под управлением Windows 7.

В отсутствие такого монитора мультисенсорный ввод можно смоделировать с помощью нескольких мышей. По адресу *http://multitouchvista.codeplex.com* предлагается необходимая для этого загрузка и ссылка на документ *http://michaelsync.net/2010/04/06/step-by-step-tutorial-installing-multi-touch-simulator-for-silverlight-phone-7*, в котором предоставляются все инструкции.

---

[1] «Самая первая книга по .NET: что должен знать разработчик на C или C++ о C# и .NET Framework» (*прим. переводчика*).

Устройства, работающие под управлением Windows Phone 7, также имеют встроенный акселерометр, который может быть *очень* сложно смоделировать на эмуляторе. Пер Бломквист (Per Blomqvist), научный редактора данной книги, на сайте http://accelkit.codeplex.com нашел приложение, в котором датчик акселерометра моделируется с помощью Веб-камеры и набора инструментов ARToolkit, и получаемые данные передаются в эмулятор Windows Phone 7 через сервер TCP/HTTP. Никто из нас не опробовал это приложение в действии, но звучит довольно заманчиво.

Примеры кода

Для иллюстрации концепций программирования на Silverlight и XNA в данной книге приведено около 190 законченных приложений. Многие из них невелики и просты, но есть и большие и довольно интересные.

Некоторые разработчики предпочитают изучать среды разработки путем воссоздания проектов в Visual Studio, вручную копируя приведенный на страницах книги код. Другие изучают код и выполняют поставляемые готовые приложения, чтобы увидеть результат. Если вы относитесь ко второй категории, весь исходный код в виде ZIP-файла можно загрузить с моего Веб-сайта www.charlespetzold.com/phone или из блога Microsoft Press по адресу http://blogs.msdn.com/b/microsoft_press/.

Если некоторые фрагменты кода будут полезны в ваших собственных программных проектах, используйте его без всяких ограничений, либо один к одному, либо внося необходимые изменения. Именно для этого я и привожу примеры.

В последний момент

Когда книга уже близилась к завершению, в свет вышла первая версия набора инструментов Silverlight for Windows Phone Toolkit, включающая некоторые дополнительные элементы и элементы управления. Она доступна для загрузки по адресу http://silverlight.codeplex.com. Очень часто эти наборы инструментов Silverlight включают предварительные версии элементов и элементов управления, которые впоследствии входят в состав выпускаемых версий Silverlight. К сожалению, я не смог включить обсуждение содержимого этого набора инструментов в соответствующие главы данной книги.

Иногда Visual Studio не может выполнить сборку или развертывание приложений на XNA. При возникновении такой проблемы на стандартной панели инструментов в выпадающем списке Solution Platforms (Платформы решения) вместо «Any CPU» (Любой ЦП) выберите «Windows Phone». Или из меню Build (Сборка) вызовите Configuration Manager (Диспетчер конфигураций) и в выпадающем списке Solution Platform (Платформа решения) вместо «Any CPU» выберите «Windows Phone».

На странице моего сайта по адресу www.charlespetzold.com/phone будет представлена информация по этому книге и, возможно, некоторые сведения по следующему изданию. Также я надеюсь активно вести блок по программированию для Windows Phone 7.

Благодарности

Своим существованием эта книга обязана Дейву Эдсону (Dave Edson) – моему старому другу еще с начала 90-х, со времен *Microsoft Systems Journal* – которому и пришла в голову замечательная мысль о том, что я мог бы написать обучающее руководство по Windows Phone 7. Дейв организовал для меня тренинг по данной тематике в Майкрософт в

декабре 2009, и меня это зацепило. Тодд Брикс (Todd Brix) дал добро на написание книги, а Ананд Ийер (Anand Iyer) уладил все вопросы с Microsoft Press.

Бен Райан (Ben Ryan) стартовал этот проект в Microsoft Press, и Девону Масгрейву (Devon Musgrave) досталась незавидная роль по доведению моего кода и прозы до уровня, достойного печатного издания. (Мы все давно знакомы. Вы найдете имена Бена и Девона на странице о соблюдении авторского права пятого издания книги *Programming Windows*, опубликованного в далеком 1998 году.)

Моим научным редактором был педантичный Пер Бломквист. Он, несомненно, протестировал весь код и файлов примеров, и листингов, приведенных в книге, и в процессе этого выявил несколько допущенных мною, откровенно говоря, позорных ошибок.

Дейв Эдсон также редактировал главы и выполнял роль посредника между мной и группой Windows Phone в Майкрософт при решении возникающих технических проблем и вопросов. Для первой «черновой» редакции Аарон Стебнер (Aaron Stebner) подготовил основное руководство; Майкл Клачер (Michael Klucher) редактировал главы; Кирти Дешпанд (Kirti Deshpande), Чарли Кайндел (Charlie Kindel), Кейси Макги (Casey McGee) и Шон Остер (Shawn Oster) давали очень ценные рекомендации и советы. Большое спасибо Бонни Лехенбаеру (Bonnie Lehenbauer) за редактирование одной из глав.

Я также в долгу перед Шоном Харгривсом (Shawn Hargreaves) за редактирование XNA-кода и перед Йочаем Кириати (Yochay Kiriaty) и Ричардом Бейли (Richard Bailey) за предоставление полной информации по захоронению.

Моя жена, Дирда Синнотт, была образцом спокойствия все эти последние месяцы, когда ей приходилось терпеть мгновенные перемены настроения, безумные выкрики в монитор компьютера и твердое убеждение в том, что написание книги освобождает от выполнения даже самых основных работ по дому.

Но, увы! Я не вправе обвинять кого-либо в ошибках или других недостатках данной книги. Все они исключительно на моей совести.

Чарльз Петзольд
Нью-Йорк и Роско, NY
22 октября, 2010

Список опечаток и поддержка книги

Мы приложили максимум усилий, чтобы обеспечить безошибочность содержимого книги и сопутствующих материалов. В случае обнаружения ошибки просим сообщить о ней в отдел Microsoft Press Book Support по электронной почте (*mspinput@microsoft.com*). (Пожалуйста, обратите внимание, что по этому адресу не предоставляется поддержка программных продуктов Майкрософт.)

Нам интересно ваше мнение

Для Microsoft Press главным приоритетом является удовлетворенность читателей, поэтому мы очень ценим ваши отзывы. Свои мнения и замечания по книге можно высказать по адресу:

http://www.microsoft.com/learning/booksurvey

Это короткий опрос, и мы внимательно читаем *все* комментарии и предложения. Заранее благодарны за ваш вклад.

Оставайтесь «на связи»

Давайте вести непрерывную дискуссию! Мы на Twitter: *http://twitter.com/MicrosoftPress*

Часть I

Основы

Глава 1
Здравствуй, Windows Phone 7

В определенный момент становится очевидным, что используемые подходы не соответствуют вашим ожиданиям. Возможно, пришла пора просто избавиться от груза прошлого, сделать глубокий вдох и подойти к решению задачи по-новому, используя свежие идеи. В компьютерной индустрии мы называем это «перезагрузка».

Перезагрузка на рынке мобильных телефонов инициирована новым подходом Майкрософт. Прекрасный внешний вид, необыкновенные шрифты и новые принципы компоновки Microsoft Windows Phone 7 делают его не просто символом разрыва с прошлым (Windows Mobile), но и выгодно отличают от других смартфонов, представленных на рынке в настоящее время. Устройства, работающие под управлением Windows Phone 7, будут выпускаться несколькими производителями и предлагаться рядом операторов мобильной связи.

Поддержка двух популярных в настоящее время платформ разработки, Silverlight и XNA, гарантирует, что в Windows Phone 7 найдется много интересного и для разработчиков.

Silverlight – браузерное развитие Windows Presentation Foundation (WPF) – уже обеспечил Веб-разработчиков беспрецедентными возможностями разработки сложных пользовательских интерфейсов, предоставляя традиционные элементы управления, высококачественный текст, векторную графику, мультимедиа, анимацию и привязку данных, которые могут выполняться во множестве браузеров и на разных платформах. Windows Phone 7 расширяет использование Silverlight на мобильные устройства.

XNA (три буквы, обозначающие «XNA не аббревиатура») – это игровая платформа Майкрософт, поддерживающая основанную на спрайтах 2D графику и 3D графику с традиционной архитектурой игрового цикла. Несмотря на то, что главным предназначением XNA является написание игр для консоли Xbox 360, разработчики могут создавать на XNA программы и для ПК, и для стильного аудиоплеера Майкрософт Zune HD.

И Silverlight, и XNA могут использоваться в качестве платформы приложения для Windows Phone 7, выбор за разработчиками. И вот здесь возникает, как говорится, «проблема выбора».

Разработка для Windows Phone 7

Все программы для Windows Phone 7 создаются с использованием управляемого кода .NET. Предлагаемые в данной книге примеры написаны на C#, но приложения для Windows Phone 7 могут создаваться и на Visual Basic .NET. Свободно доступный для загрузки Microsoft Visual Studio 2010 Express for Windows Phone включает XNA Game Studio 4.0 и экранный эмулятор телефона, а также интегрируется с Visual Studio 2010. Визуальные элементы и анимация для приложений Silverlight могут создаваться в Microsoft Expression Blend.

Платформы Silverlight и XNA для Windows Phone 7 имеют ряд общих библиотек, т.е. некоторые библиотеки XNA могут использоваться в программе Silverlight и наоборот. Но нельзя создавать программу, сочетающую в себе визуальные элементы обеих платформ. Вероятно, это будет возможным в будущем, но не сейчас. Если в вашей голове созрела идея на миллион долларов, не спешите создавать проект в Visual Studio, решите сначала, на какой платформе она будет реализовываться: на Silverlight или XNA.

Как правило, Silverlight используется для программ, которые можно классифицировать как приложения или утилиты. Эти программы являются сочетанием разметки и кода. Для

описания разметки – главным образом, компоновки элементов управления и панелей пользовательского интерфейса – используется Расширяемый язык разметки приложений (Extensible Application Markup Language, XAML1). В файлах выделенного кода могут реализовываться операции по инициализации и некоторая логика, но основным их назначением является обработка событий элементов управления. Silverlight позволяет реализовать в Windows Phone стиль Насыщенных Интернет-приложений (Rich Internet Applications, RIA), включая мультимедиа и Веб. Для Windows Phone создана версия Silverlight 3, в которую не вошли некоторые возможности, не подходящие для телефона, но они компенсированы рядом дополнений.

Главное назначение XNA – создание высокопроизводительных игр. Для 2D игр спрайты и подложки описываются с помощью растровых изображений; для 3D игр создаются трехмерные модели. Действие игры, включающее перемещение графических объектов по экрану и запрос пользовательского ввода, обрабатывается встроенным игровым циклом XNA.

Удобно провести границы и принять, что Silverlight используется для приложений, а XNA – для игр, но это не должно накладывать ограничения. Вне всяких сомнений, Silverlight может применяться для реализации игр, и традиционные приложения могут создаваться на XNA, хотя подчас это будет сопряжено со значительными трудностями.

В частности, Silverlight идеально подходит для игр с небольшими требованиями по графике, либо использующих векторную, а не растровую графику, либо темп которых определяется реакцией пользователя, а не таймером. Программы такого типа, как Тетрис, прекрасно могли бы быть реализованы на Silverlight. А вот XNA довольно сложно распространить на области применения Silverlight. Реализация окна списка на XNA может быть занятным для некоторых разработчиков, но для основной массы программистов это было бы пыткой.

В первых нескольких главах данной книги Silverlight и XNA рассматриваются параллельно, затем каждой платформе посвящается отдельная часть. Я подозреваю, что некоторые разработчики займутся исключительно Silverlight или XNA и даже не будут утруждать себя изучением другой среды. Надеюсь, что это явление не примет массового характера. Положительный момент в том, что Silverlight и XNA настолько не похожи, что между ними можно переключаться туда и обратно без всякой путаницы!

Майкрософт позиционируют Silverlight как клиентскую часть или «лицо» облака. Таким образом, сервисы облака и Windows Azure Platform формируют важную составляющую разработки приложений для Windows Phone 7. Windows Phone полностью «готов к работе с облаком». Программы учитывают местонахождение пользователя и имеют доступ к картам и другим данным посредством Bing и Windows Live. Один из предлагаемых сервисов облака – Xbox Live – обеспечивает возможность участия программ XNA в многопользовательских сетевых играх и также доступа к ним приложений Silverlight.

Программы, создаваемые вами для Windows Phone 7, будут продаваться и устанавливаться через сайт партнерских решений для Windows Phone (Windows Phone Marketplace). Этот сайт обеспечивает регистрацию и гарантирует, что программы отвечают минимальному набору требований надежности, эффективности и нормам этики.

Я охарактеризовал Windows Phone 7 как символ разрыва с прошлым. Если проводить сравнение с предыдущими версиями Windows Mobile, это, несомненно, так. Но поддержка Silverlight, XNA и C# не разрывает связей с прошлым, а является сбалансированным сочетанием преемственности и нововведений. Несмотря на свой младенческий возраст, Silverlight и XNA уже зарекомендовали себя как мощные и популярные платформы. Многие

---

[1] Читается «замл».

опытные разработчики уже работают с одной из этих инфраструктур (вероятно, не так многие с обеими, но это лишь пока) и демонстрируют свое воодушевление, публикуя огромное количество информации в сети и организуя множество сообществ. C# стал любимым языком многих разработчиков (и моим в том числе). Он обеспечивает возможность совместного использования библиотек Silverlight и XNA приложениями, а также программами для других .NET-сред.

Аппаратные средства

Разработчики, имеющие опыт создания приложений для устройств, поддерживающих Windows Mobile, найдут существенные отличия в стратегии Майкрософт для Windows Phone 7. Она отличается крайней предупредительностью в описании аппаратных средств, которые часто называют «железом».

Первоначальные версии устройств Windows Phone 7 будут иметь экран одного размера. (В будущем ожидается использование еще одного размера экрана.) Присутствие многих аппаратных возможностей гарантируется на каждом устройстве.

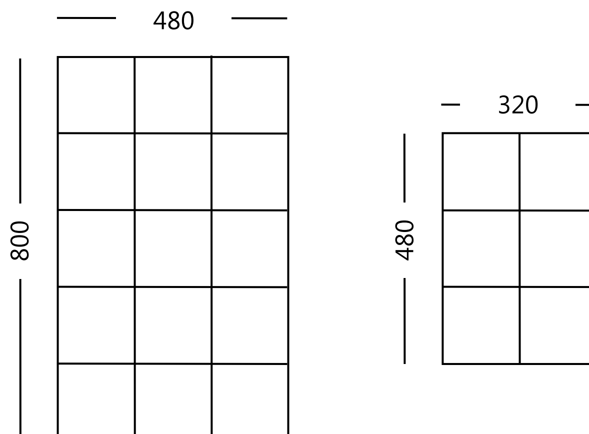
На фронтальной части телефона имеется дисплей, поддерживающий мультисенсорный ввод, и три кнопки, как правило, располагающиеся под дисплеем. Наименование этих кнопок в порядке слева направо:

- **Back (Назад)** Программы могут использовать эту кнопку для навигации, во многом аналогично кнопке Back в Веб-браузере. При использовании со стартовой страницы программы эта кнопка приводит к завершению программы.
- **Start (Пуск)** Эта кнопка обеспечивает переход пользователя к стартовому экрану телефона; в противном случае этот экран недоступен программам, выполняющимся на телефоне.
- **Search (Поиск)** Операционная система использует эту кнопку для запуска поиска.

Первоначальные версии устройств Windows Phone 7 имеют экран размером 480 × 800 пикселов. В будущем ожидается также применение экранов размером 320 × 480 пикселов. Никаких других вариантов для Windows Phone 7 не планируется, поэтому очевидно, что эти два размера экрана играют очень важную роль в разработке телефона.

Теоретически, лучше всего создавать программы, которые самостоятельно адаптируются к любому размеру экрана. Но это не всегда возможно, в частности, для игр. Вероятнее всего, приложения будут ориентироваться на эти два размера экрана, что будет реализовано в виде различных фрагментов кода и разных XAML-файлов для компоновки, которая зависит от размеров экрана.

Как правило, я буду ссылаться на эти два размера, как на «большой» и «маленький» экран. Наибольший общий знаменатель горизонтального и вертикального размеров обоих экранов – 160, т.е. их можно изобразить с помощью квадратов со стороной 160 пикселов:



Я показываю экраны в портретном режиме, потому что обычно именно так ориентированы экраны смартфонов. Для сравнения, экран оригинального Zune – 240 × 320 пикселов; Zune HD – 272 × 480.

Конечно, телефон можно перевернуть горизонтально и получить экран в альбомном режиме. Некоторые программы могут требовать определенной ориентации экрана телефона, другие будут более гибкими.

Разработчик полностью контролирует, в какой мере создаваемое им приложение поддерживает ориентацию экрана. По умолчанию приложения Silverlight отображаются в портретном режиме, но можно сделать так, чтобы они самостоятельно приспосабливались к изменениям ориентации экрана. Специально для целей определения изменения ориентации предусмотрены новые события, а некоторые изменения обрабатываются автоматически. Для сравнения, разработчики игр обычно задают конкретную ориентацию экрана пользователя. Программы на XNA используют альбомный режим по умолчанию, но это свойство легко переопределить.

В портретном режиме маленький экран равен половине старого VGA-экрана (т.е. 640 × 480). В альбомном режиме большой экран имеет размеры, соответствующие так называемому WVGA («wide VGA1»). В альбомном режиме соотношение ширины и высоты маленького экрана составляет 3:2 или 1,5; для большого экрана это соотношение 5:3 или 1,66.... Ни одно из этих соотношений не совпадает с пропорциями телевизионного экрана, которые для телевизоров стандартного разрешения составляют 4:3 или 1,33… и для телевизоров с высоким разрешением – 16:9 или 1,77…. Соотношение высоты и ширины экрана Zune HD – 16:9.

Как во многих современных телефонах и Zune HD, для экранов телефонов Windows Phone 7, скорее всего, будет использоваться технология ОСИД («органический светоизлучающий диод»)2, хотя это не является требованием к оборудованию. Экраны ОСИД отличаются от плоских экранов, используемых в прошлом, тем, что их энергопотребление пропорционально излучаемому свету. Например, ОСИД-экран потребляет менее половины энергии, необходимой для жидкокристаллического (ЖК) монитора того же размера, но это только в режиме, когда экран преимущественно темный. Для полностью светлого экрана ОСИД потребляет более чем в три раза больше энергии, необходимой для ЖК.

Продолжительность автономной работы имеет первостепенную важность для мобильных устройств, поэтому эта характеристика ОСИД-дисплеев предполагает применение

[1] Широкий VGA (*прим. переводчика*).
[2] Organic light emitting diode, OLED

преимущественно черных фонов с редкими графическими элементами и контрастных шрифтов. Независимо от этого пользователи Windows Phone 7 могут выбирать между двумя основными цветовыми темами: светлый текст на темном фоне или темный текст на светлом фоне.

Пользовательский ввод для программ Windows Phone 7 будет осуществляться посредством мультисенсорного ввода. Экраны поддерживают технологию емкостного касания. Это означает, что они отвечают только на прикосновение человека, но не реагируют на касание стилусом или другие формы давления. touch Экраны устройств Windows Phone 7 должны распознавать одновременное касание как минимум в четырех точках.

Аппаратные клавиатуры необязательны. Необходимо предполагать, что дизайн телефонов может быть разным, поэтому при использовании клавиатуры экран может быть либо в портретном, либо в альбомном режиме. Программа на Silverlight, использующая ввод с клавиатуры, *должна* реагировать на изменения ориентации экрана, чтобы пользователь мог просматривать экран и использовать клавиатуру, не задаваясь вопросом, что за идиот разрабатывал приложение. Также предоставляется экранная клавиатура, которую в кругах Windows-пользователей называют Soft Input Panel (Панель функционального ввода) или SIP. В приложениях на XNA также реализовывается работа с аппаратной клавиатурой и SIP.

Датчики и сервисы

Устройство Windows Phone 7 должно иметь ряд других аппаратных возможностей – иногда называемых датчиками – и предоставлять некоторые программные сервисы, возможно, с аппаратной поддержкой. Рассмотрим те из них, которые наиболее интересны разработчикам:

- **Wi-Fi** Для доступа к Интернету телефон снабжен Wi-Fi в дополнение к доступу к данным по технологиям 3G (3G data access), предоставляемому поставщиком мобильной связи. В программное обеспечение, установленное на телефоне, включена версия Internet Explorer.

- **Камера** Телефон снабжен камерой с разрешением не менее 5 мегапикселов и вспышкой. Программы могут вызывать ПО камеры для осуществления ввода с нее или регистрироваться как приложение расширений для обработки фотографий. В этом случае они будут отображаться в меню для получения доступа к сфотографированным изображениям, например, для обработки этих изображений определенным образом.

- **Акселерометр** измеряет ускорение, что является физической величиной, обозначающей изменение скорости. Если камера неподвижна, акселерометр реагирует на изменение гравитации. Программы могут получать трехмерный вектор, определяющий положение камеры относительно земли. Акселерометр также может выявлять резкие перемещения телефона.

- **Местоположение** По желанию пользователя телефон может применять множество стратегий определения своего географического местоположения. Телефон передает в аппаратное устройство GPS данные из Интернета или вышек сотовой связи. При его перемещении также могут предоставляться данные о направлении и скорости.

- **Вибрация** Программное управление вибрацией телефона.

- **FM-радио** Программный доступ к FM-радио.

- **Принудительные уведомления** Для обновления данных, предоставляемых некоторыми Веб-сервисами, телефону пришлось бы регулярно опрашивать эти сервисы. Это могло бы приводить к разрядке батареи и сокращению времени автономной работы.

Для решения этой проблемы был создан сервис принудительных уведомлений. Он берет на себя задачу по опросу всех необходимых сервисов и передает на телефон уведомления только при обновлении данных.

File | New | Project

Я буду предполагать, что у читателя установлена Visual Studio 2010 Express for Windows Phone, либо самостоятельная версия, либо как дополнение к Visual Studio 2010. Для удобства я буду называть эту среду разработки просто «Visual Studio».

Традиционная программа «Здравствуй, мир!», отображающая лишь коротенькую фразу, может казаться глупой для неразработчиков. Но разработчики уже знают, что такое приложение служит, по крайней мере, двум полезным целям. Во-первых, это способ проверить, насколько просто (или нелепо сложно) вывести на экран простую текстовую строку. Во-вторых, это возможность для разработчиков пройти процесс создания, компиляции и выполнения программы без особых проблем. При разработке программ для мобильных устройств этот процесс немного усложняется, потому что написание и компиляция программы выполняется на ПК, а развертываться и выполняться она будет на реальном телефоне или, как минимум, на его эмуляторе.

В данной главе представлены программы для отображения фразы «Hello, Windows Phone 7!», созданные на Microsoft Silverlight и Microsoft XNA.

Чтобы просто сделать эти программы немного более интересными, я ставлю задачу вывести текст в центре экрана. В приложении на Silverlight будут использоваться цвета фона и текста, выбранные пользователем в разделе Themes (Темы) окна Settings (Настройки) телефона. Приложение на XNA будет выводить белый текст на темном фоне, чтобы обеспечить меньшее энергопотребление ОСИД.

Если вы готовы создать свою первую программу для Windows Phone 7, пора запустить Visual Studio, в меню File (Файл) выбрать New (Новый) и затем Project (Проект).

Первое приложение для телефона на Silverlight

В диалоговом окне New Project (Новый проект) слева под Installed Templates (Установленные шаблоны) выберите Visual C# и затем Silverlight for Windows Phone. На средней панели выберите Windows Phone Application (Приложение Windows Phone). Выберите, где будет размещаться проект, и введите имя проекта, SilverlightHelloPhone.

Когда проект создан, на экране появится изображение телефона с большим экраном (размером 480 × 800 пикселов) в портретном режиме. Это конструктор. Создать приложение, можно просто перетягивая элементы управления из панели инструментов на рабочую область, но я хочу показать, как самостоятельно писать код и создавать разметку.

Для данного проекта SilverlightHelloPhone автоматически было создано несколько файлов. Их можно увидеть под именем проекта в Solution Explorer (Обозреватель решений) справа. В папке Properties (Свойства) располагаются три файла, на которые при создании простого примера приложения на Silverlight для телефона можно не обращать внимания. Эти файлы имеют значение только при создании реального приложения.

Тем не мене, если открыть файл WMAppManifest.xml, сверху в теге App можно увидеть атрибут:

```
Title="SilverlightHelloPhone"
```

Это просто имя выбранного проекта. Вставим пробелы, чтобы сделать его более удобным для восприятия:

```
Title="Silverlight Hello Phone"
```

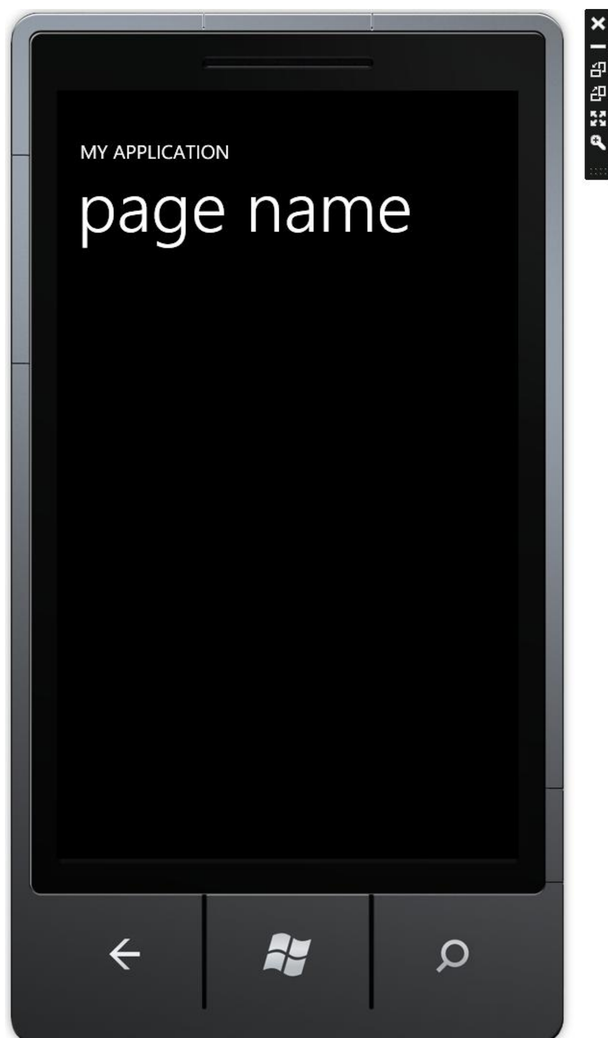
Это имя используется телефоном и эмулятором телефона для отображения программы в списке установленных приложений, представляемом пользователю. Самые любознательные могут также отредактировать файлы ApplicationIcon.png и Background.png, используемые телефоном для создания графического символа программы. Файл SplashScreenImage.jpg отображается при запуске программы.

На стандартной панели инструментов Visual Studio под меню программы можно будет увидеть выпадающий список, отображающий, скорее всего, «Windows Phone 7 Emulator» (Эмулятор Windows Phone 7) или «Windows Phone 7 Device» (Устройство Windows Phone 7). Это средство для развертывания создаваемой программы либо на эмуляторе, либо на реальном телефоне, подключенном к компьютеру разработки через USB.

Чтобы убедиться в том, что все работает нормально, выберите Windows Phone 7 Emulator и нажмите F5 (или выберите Start Debugging (Начать отладку) в меню Debug (Отладка)). Будет быстро выполнена сборка программы, и в строке состояния появится сообщение «Connecting to Windows Phone 7 Emulator…» (Выполняется подключение к эмулятору Windows Phone 7…). Загрузка эмулятора, если он используется в текущем сеансе впервые, может занять некоторое время. Если оставить эмулятор выполняться между циклами редактирования/сборки/выполнения, Visual Studio не придется вновь выполнять подключение к нему.

Вскоре на экране появится эмулятор телефона. Поскольку наша простенькая ничего не делающая программа на Silverlight развернута на эмуляторе, она будет выполнена после заставки[1]. В телефоне можно будет увидеть практически ту же картинку, что мы видели в конструкторе.

[1] После заставки и до завершения разворачивания приложения, некоторое время будет отображаться стартовый экран телефона с блоком Internet Explorer на нем (*прим. научного редактора*).



Эмулятор телефона имеет небольшое всплывающее меню, которое появляется в верхнем правом углу при перемещении указателя мыши в эту область. Это меню позволяет изменять ориентацию или размер эмулятора. По умолчанию размер отображаемого эмулятора составляет 50% фактического размера, размер рисунка на этой странице примерно такой же. При отображении эмулятора в 100% величину, он выглядит огромным. Сразу же возникнут вопросы: «Как телефон такого размера поместится в моем кармане?»

От масштаба отображения зависит плотность пикселей. Разрешение экрана компьютера обычно примерно 100 пикселей на дюйм. (По умолчанию Windows предполагает, что разрешающая способность экрана – 96 точек на дюйм.) Экран реального устройства Windows Phone 7 имеет более чем в 2,5 раза большую разрешающую способность. При отображении эмулятора в 100% величину размер всех точек экрана телефона будет составлять примерно 250% их реального размера.

Чтобы прервать выполнение программы и вернуться к ее редактированию, в Visual Studio нажмите Shift-F5 или выберите Stop Debugging (Остановить отладку) в меню Debug, либо щелкните кнопку Back эмулятора.

Не закрывайте сам эмулятор, т.е. не нажимайте кнопку X вверху всплывающего меню эмулятора! Если эмулятор останется открытым, последующие развертывания будут выполняться намного быстрее.

Пока эмулятор выполняется, он сохраняет все развертываемые в нем приложения. Если щелкнуть стрелку в верхнем правом углу стартового экрана, будет выведен список всех

приложений. В нем можно будет найти и наше приложение, обозначенное строкой «Silverlight Hello Phone». Его можно запустить из этого списка снова. Если эмулятор закрыть, приложение исчезнет из списка.

Имеющееся устройство, работающее под управлением Windows Phone 7, необходимо зарегистрировать на странице партнерских решений портала Windows Phone 7 (*http://developer.windowsphone.com*). После подтверждения регистрации подключите телефон к ПК и запустите настольное ПО Zune. Чтобы разблокировать телефон для разработки, запустите приложение Windows Phone Developer Registration (Регистрация разработчика для Windows Phone) и введите в нем свой Windows Live ID. После этого можете развертывать приложения из Visual Studio на телефон.

Стандартные файлы Silverlight

Загрузив проект в Visual Studio, заглянем в Solution Explorer. Там мы найдем две пары ключевых файлов: App.xaml и App.xaml.cs, MainPage.xaml и MainPage.xaml.cs. Файлы App.xaml и MainPage.xaml – это файлы XAML, тогда как App.xaml.cs и MainPage.xaml.cs – это файлы C#. Такой своеобразный принцип именования файлов подразумевает, что два файла C#-кода являются файлами выделенного кода, связанными с двумя XAML-файлами. Они содержат код для поддержки разметки. Это основная концепция Silverlight.

Сделаем небольшой обзор этих четырех файлов. В файле App.xaml.cs можно найти описание пространства имен, которое совпадает с именем проекта, и класс *App*, производный от Silverlight-класса *Application* (Приложение). Приведем фрагмент этого файла, чтобы продемонстрировать его общую структуру:

Проект Silverlight: SilverlightHelloPhone Файл: App.xaml.cs (фрагмент)

```
namespace SilverlightHelloPhone
{
    public partial class App : Application
    {
        public App()
        {
            …
            InitializeComponent();
            …
        }
        …
    }
}
```

Во всех приложениях на Silverlight имеется класс *App*, производный от *Application*. Этот класс осуществляет все рутинные операции по общей инициализации приложения, запуску и завершению выполнения. Можно заметить, что этот класс определен как *partial* (частичный). Это говорит о том, что проект, вероятно, должен включать еще один C#-файл с дополнительными членами класса *App*. Но где этот файл?

В проекте также есть файл App.xaml, общая структура которого такова:

Проект Silverlight: SilverlightHelloPhone Файл: App.xaml (фрагмент)

```
<Application
    x:Class="SilverlightHelloPhone.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
```

```
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone">
    …
</Application>
```

Данный файл выглядит как XML, или, точнее, это XAML-файл, являющийся важной составляющей концепции Silverlight. В частности, разработчики часто используют файл App.xaml для хранения *ресурсов*, используемых приложением. Эти ресурсы могут включать цветовые схемы, градиентные кисти, стили и т.д.

Корневым элементом является *Application* – класс Silverlight, от которого наследуется класс *App*. Корневой элемент включает описания четырех пространств имен XML. Два из них являются общими для всех приложений Silverlight, два – используются только в приложениях для телефонов.

Первым объявляется пространство имен «xmlns», стандартное для Silverlight. Оно используется компилятором для определения местоположения и идентификации классов Silverlight, таких же как сам класс *Application*. Как и большинство описаний пространств имен XML, этот URI не указывает ни на что конкретное, это просто URI, принадлежащий Майкрософт и определенный для этой цели.

Второе пространство имен XML связано с самим XAML и позволяет ссылаться в файле на некоторые элементы и атрибуты, являющиеся, скорее, частью XAML, а не Silverlight. По общепринятым правилам это пространство имен ассоциируется с префиксом «x» (что означает «XAML»).

К таким атрибутам, поддерживаемым XAML и обозначаемым префиксом «x», относится Class (Класс), который часто называют «x class». В этом конкретном XAML-файле *x:Class* присвоено имя *SilverlightHelloPhone.App*. Это означает, что класс *App* в пространстве имен .NET *SilverlightHelloPhone* наследуется от Silverlight-класса *Application*, корневого элемента. Это описание того же класса, что мы видели в файле App.xaml.cs, но с использованием совершенно другого синтаксиса.

Файлы App.xaml.cs и App.xaml описывают две части одного и того же класса App. Во время компиляции Visual Studio проводит синтаксический разбор App.xaml и формирует еще один файла кода App.g.cs. «g» означает «generated» или «автоматически сформированный». Этот файл можно найти в подпапке \obj\Debug проекта. App.g.cs также является частичным описанием класса *App* и включает метод *InitializeComponent* (Инициализировать компонент), который вызывается из конструктора в файле App.xaml.cs.

Файлы App.xaml и App.xaml.cs можно редактировать, но не стоит тратить время на App.g.cs. Этот файл создается повторно при каждой сборке проекта.

При запуске программы класс *App* создает объект типа *PhoneApplicationFrame* (Рамка приложения для телефона) и присваивает этот объект собственному свойству *RootVisual*. Это рамка шириной 480 пикселов и высотой 800 пикселов, которая занимает весь экран телефона. После этого объект *PhoneApplicationFrame* ведет себя подобно Веб-браузеру и переходит к объекту *MainPage* (Главная страница).

MainPage – второй основной класс любого приложения на Silverlight, он описывается во второй паре файлов: MainPage.xaml и MainPage.xaml.cs. В небольших приложениях на Silverlight больше всего внимания разработчики уделяют именно этим двум файлам.

Если не брать во внимание длинный список директив *using*, файл MainPage.xaml.cs очень прост:

Проект Silverlight: SilverlightHelloPhone   Файл: MainPage.xaml.cs (фрагмент)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;

namespace SilverlightHelloPhone
{
    public partial class MainPage : PhoneApplicationPage
    {
        // Конструктор
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

Директивы *using* для пространств имен, начинающихся со слов *System.Windows*, предназначены для классов Silverlight; иногда их необходимо дополнить еще некоторыми директивами using. Пространство имен *Microsoft.Phone.Controls* включает расширения Silverlight для телефона, в том числе и класс *PhoneApplicationPage* (Страница приложения для телефона).

Опять же, перед нами еще одно *частичное* (partial) описание класса. В данном случае описывается класс *MainPage*, производный от Silverlight-класса *PhoneApplicationPage*. Этот класс определяет визуальные элементы, которые пользователь видит на экране при выполнении программы *SilverlightHelloPhone*.

Вторая половина класса *MainPage* описывается в файле MainPage.xaml. Привожу практически полный файл, он был лишь немного переформатирован соответственно печатной странице. Также из него изъят раздел в конце, который закомментирован, но являет собой довольно устрашающий фрагмент разметки:

Проект Silverlight: SilverlightHelloPhone   Файл: MainPage.xaml (практически полностью)

```
<phone:PhoneApplicationPage
    x:Class="SilverlightHelloPhone.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <!--LayoutRoot - это основной контейнер, в котором располагается все содержимое
страницы-->
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
```

```
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!--TitlePanel включает имя приложения и заголовок страницы-->
        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
                       Style="{StaticResource PhoneTextNormalStyle}"/>
            <TextBlock x:Name="PageTitle" Text="page name" Margin="9,-7,0,0"
                       Style="{StaticResource PhoneTextTitle1Style}"/>
        </StackPanel>

        <!--ContentPanel - место для размещения дополнительного содержимого-->
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

Первые четыре описания пространств имен XML аналогичны приведенным в файле App.xaml. Как и в файле App.xaml, атрибут *x:Class* также находится в корневом элементе. Здесь он показывает, что класс *MainPage* в пространстве имен *SilverlightHelloPhone* наследуется от Silverlight-класса *PhoneApplicationPage*. Этот класс *PhoneApplicationPage* требует собственного описания пространства имен XML, поскольку не является частью стандартного Silverlight.

Описания пространств имен, ассоциированные с префиксами «d» («designer1») и «mc» («markup compatibility2»), предназначены для программ визуального редактирования XAML, таких как Expression Blend и дизайнер в самой Visual Studio. Атрибуты *DesignerWidth* (Ширина дизайнера) и *DesignerHeight* (Высота дизайнера) игнорируются во время компиляции.

При компиляции программы автоматически формируется файл MainPage.g.cs (его можно найти в подпапке \obj\Debug), содержащий еще одну часть описания класса *MainPage*. Она включает метод *InitializeComponent*, который вызывается из конструктора в файле MainPage.xaml.cs.

Теоретически, файлы App.g.cs и MainPage.g.cs формируются автоматически в процессе сборки исключительно в целях внутреннего использования компилятором, и разработчики могут абсолютно не обращать на них никакого внимания. Однако порой, когда изобилующая ошибками программа формирует исключение, на экране появляется один из этих файлов. Анализ этих файлов до того, как они загадочным образом появляются перед вашими глазами, может помочь в понимании проблемы. Тем не менее, не пытайтесь редактировать их, чтобы устранить проблему! Она, скорее всего, кроется в соответствующем файле XAML.

В корневом элементе MainPage.xaml можно найти настройки *FontFamily* (Семейство шрифтов), *FontSize* (Размер шрифта) и *Foreground* (Цвет шрифта), применяемые к странице в целом. Элемент *StaticResource* (Статический ресурс) и данный синтаксис будут рассмотрены в главе 7.

Тело файла MainPage.xaml включает несколько вложенных элементов *Grid* (Сетка), *StackPanel* (Стек-панель) и *TextBlock* (Блок текста).

Обратите внимание на то, что я использую слово *элемент*. В разработке на Silverlight оно имеет два значения. Это термин XML, используемый для обозначения элементов, ограниченных начальными и конечными тегами. Но также элементами в Silverlight называют визуальные объекты. И, кстати, слово *элемент* появляется в именах двух Silverlight-классов.

---

[1] Дизайнер (*прим. переводчика*).

[2] Совместимость разметки (*прим. переводчика*).

Многие используемые в Silverlight классы являются частью этой важной иерархии классов:

Object
  DependencyObject (абстрактный)
    UIElement (абстрактный)
      FrameworkElement (абстрактный)

Не только *UIElement* (Элемент пользовательского интерфейса), а и многие другие классы Silverlight наследуются от *DependencyObject* (Объект с поддержкой зависимостей). Но *UIElement* отличается тем, что является классом, который может отображаться на экране как визуальный объект и принимать пользовательский ввод. (В Silverlight все визуальные объекты могут принимать пользовательский ввод.) Традиционно ввод осуществляется с клавиатуры и мыши. На телефоне он выполняется преимущественно посредством сенсорного ввода.

Единственный класс, наследуемый от *UIElement – FrameworkElement* (Элемент структуры). Разделение этих двух классов в Windows Presentation Foundation сложилось исторически. В WPF разработчики могут создавать собственные уникальные структуры, наследуясь от *UIElement*. В Silverlight это невозможно, поэтому данное разделение и имеет такое большое значение.

Одним из классов, наследуемых от *FrameworkElement*, является *Control* (Элемент управления) (в контексте разработки графических пользовательских интерфейсов этот термин употребляется чаще, чем *элемент*). Некоторые объекты, которые в других программных средах обычно называют *элементами управления*, в Silverlight более корректно именовать *элементами*. К производным от Control относятся кнопки и ползунки, которые обсуждаются в главе 10.

Еще один класс, наследуемый от *FrameworkElement – Panel* (Панель). Он является родительским классом элементов *Grid* и *StackPanel*, которые мы видели в файле MainPage.xaml. Панели – это элементы, в которых могут размещаться и компоноваться определенным образом множество дочерних элементов. Более подробно панели будут рассмотрены в главе 9.

Также от *FrameworkElement* наследуется класс *TextBlock* (Блок текста) – элемент, который чаще всего используется для отображения блоков текста размером до абзаца. Два элемента *TextBlock* в файле MainPage.xaml обеспечивают отображение двух фрагментов текста заголовка программы Silverlight.

PhoneApplicationPage, *Grid*, *StackPanel* и *TextBlock* – все это классы Silverlight. В разметке они становятся элементами XML. Свойства этих классов становятся атрибутами XML.

Вложенные элементы в MainPage.xaml, как говорится, формируют *дерево визуальных элементов*. В Silverlight-приложении для Windows Phone 7 дерево визуальных элементов всегда начинается с объекта типа *PhoneApplicationFrame*, который занимает всю поверхность отображения телефона. В программе на Silverlight для Windows Phone 7 всегда есть только один единственный экземпляр *PhoneApplicationFrame*, в просторечии называемый *рамкой*.

И, в противоположность этому, в программе может быть множество экземпляров *PhoneApplicationPage*, которые обычно называют просто *страницами*. В любой отдельно взятый момент времени в рамке размещается всего одна страница, но имеется возможность перехода к другим страницам. По умолчанию страница не занимает всю поверхность отображения рамки, оставляя место для панели задач (которую также называют строкой состояния) вверху экрана телефона.

В нашем простом приложении всего одна страница, которая названа *MainPage*, соответственно. Эта *MainPage* включает *Grid*, в котором располагается *StackPanel* с парой элементов *TextBlock*, и еще один *Grid*. Все они образуют иерархическое дерево. Визуальное дерево, создаваемое Visual Studio для программы на Silverlight, выглядит следующим образом:

PhoneApplicationFrame
  PhoneApplicationPage
    Grid под названием «LayoutRoot»
      StackPanel под названием «TitlePanel»
        TextBlock под названием «ApplicationTitle»
        TextBlock под названием «PageTitle»
      Grid под названием «ContentPanel»

Нашей первоначальной целью было создать приложение на Silverlight, выводящее некоторый текст в центре экрана. Но поскольку у нас имеется пара заголовков, немного подкорректируем формулировку задачи: цель состоит в отображении в центре страницы текста, не являющегося ее заголовком. Область страницы для отображения содержимого представлена элементом *Grid*, описанным в конце файла после комментария «ContentPanel – место для размещения дополнительного содержимого». Этот *Grid* имеет имя «ContentPanel» (Панель для содержимого). Я так и буду его называть, панель или сетка для содержимого. Область экрана, соответствующую этому *Grid*, не включая заголовки, обычно именуют областью содержимого.

В ContentGrid можно вставить новый *TextBlock*:

Проект Silverlight: SilverlightHelloPhone Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Text (Текст), *HorizontalAlignment* (Выравнивание по горизонтали) и *VerticalAlignment* (Выравнивание по вертикали) – все это свойства класса *TextBlock*. Свойство *Text* типа *string* (строка). Свойства *HorizontalAlignment* и *VerticalAlignment* типа перечисление *HorizontalAlignment* и *VerticalAlignment*, соответственно. Для ссылки на тип перечисления в XAML необходимо указать только имя члена.

При редактировании MainPage.xaml можно также немого подкорректировать элементы *TextBlock*, чтобы они не выглядели так стандартно. Меняем

```
<TextBlock … Text="MY APPLICATION" … />
```

на

```
<TextBlock … Text="SILVERLIGHT HELLO PHONE" … />
```

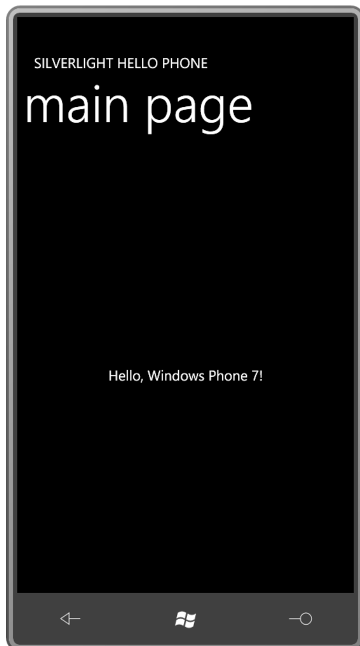
и

```
<TextBlock … Text="page title" … />
```

на:

```
<TextBlock … Text="main page" … />
```

В одностраничном приложении на Silverlight особого смысла в заголовке нет. Поэтому второй *TextBlock* можно удалить. Внесенные в данный XAML-файл изменения будут отражены в конструкторе. Теперь эту программу можно откомпилировать и запустить:



Размер данного снимка экрана и большинства снимков экрана в данной книге примерно соответствует реальному размеру телефона. Хромированная окантовка еще более усиливает ощущение того, что это реальный телефон или его эмулятор (surrounded by some simple ”chrome” that symbolizes either the actual phone or the phone emulator).

Несмотря на всю свою простоту, данное приложение демонстрирует некоторые основополагающие принципы разработки на Silverlight, включая динамическую компоновку. Файл XAML определяет компоновку элементов в дереве визуальных элементов. Каждый элемент может самостоятельно динамически менять свое местоположение и размеры. В зависимости от значений свойств *HorizontalAlignment* и *VerticalAlignment* элемент может размещаться в центре другого элемента или (о чем несложно догадаться) вдоль одного из краев экрана, или в одном из углов. *TextBlock* – это лишь один из целого ряда элементов, доступных для использования в программе на Silverlight. К ним относятся растровые изображения, фильмы и такие привычные элементы управления, как кнопки, ползунки и списки.

Цветовые темы

На стартовом экране телефона или эмулятора телефона щелкните или коснитесь правой стрелки в верхнем правом углу, перейдите к странице Settings (Настройки) и выберите Theme (Тема). Тема Windows Phone 7 включает цвет фона (Background) и контрастный цвет (Accent). Можно выбрать темную (светлый текст на темном фоне, которая использовалась до этого) или светлую (темный текст на светлом фоне) цветовую тему. Выберем светлую тему и снова запустим SilverlightHelloPhone. Почувствуем некоторое удовлетворение от того, что цвета темы применяются автоматически:

Несмотря на то что эти цвета применяются автоматически, приложение не ограничено только ими. Если требуется изменить цвет отображаемого текста, можно задать атрибут *Foreground* в теге *TextBlock*, например:

```
Foreground="Red"
```

Его можно поместить в любое место тега. При вводе этого атрибута вам будет предложен выпадающий список доступных цветов. Silverlight поддерживает 140 именованных цветов, которые поддерживаются многими браузерами, и один дополнительный, 141-ый цвет – *Transparent* (Прозрачный).

В реальных программах все применяемые цвета должны тестироваться с доступными темами, чтобы в один прекрасный момент текст не пропал загадочным образом или не стал трудночитаемым.

Пункты и пикселы

Еще одно свойство *TextBlock*, которое можно без труда изменить – *FontSize*:

```
FontSize="36"
```

Но что именно означает эта цифра?

Единицами измерения в Silverlight являются пикселы, и *FontSize* не исключение. Задавая 36, мы получаем шрифт, высота которого от самого верха надстрочного элемента до самого низа подстрочного элемента составляет примерно 36 пикселов.

Но со шрифтами все не так просто. Фактическая высока результирующего *TextBlock* будет составлять 48 пикселов, что примерно на 33% выше, чем подразумевается значением *FontSize*. Благодаря этому дополнительному промежутку (который называют *межстрочным интервалом*) строки текста не «наползают» друг на друга.

Традиционно размеры шрифтов выражаются в *пунктах*. В обычной типографии пункт примерно равен 1/72 дюйма, но в цифровой типографии за пункт часто принимается строго 1/72 дюйма. Шрифт размером 72 пункта равен примерно дюйму. (Я говорю «примерно», потому что размер пункта отражает высоту группировки строк, и, на самом деле, только

дизайнер шрифта определяет то, насколько большими должны быть символы шрифта размером 72 пункта.)

Как преобразовывать пикселы в пункты и обратно? Очевидно, что это можно сделать только для конкретного устройства вывода. На принтере с разрешением 600 точек на дюйм (dots-per-inch, DPI), например, 72-й шрифт будет 600 пикселов высотой.

Обычно используемые сегодня мониторы, как правило, имеют разрешение в районе 100 DPI. Например, возьмем монитор в 21", обеспечивающий 1600 пикселов по горизонтали и 1200 пикселов по вертикали. Это составляет 2000 пикселов по диагонали. Разделим на 21" и получим примерно 95 DPI.

По умолчанию Microsoft Windows предполагает разрешение экрана равным 96 DPI. При таком предположении соотношения размеров шрифтов и количества пикселов описываются следующими выражениями:

пункты = ¾ × пикселы

пункты = 4/3 × пикселы

Данное отношение применимо только к обычным мониторам, но людям так нравятся эти формулы преобразования, что они используют их и в разработке для Windows Phone 7.

Итак, задавая свойству *FontSize* значение

```
FontSize="36"
```

можно утверждать, что задан шрифт размером 27 пунктов.

Чтобы получить размер в пикселах, увеличьте заданный в пунктах размер на 33%. Именно это значение задается свойству *FontSize* элемента *TextBlock*. Высота результирующего *TextBlock* будет еще на 33% больше *FontSize*.

Вопрос размера шрифта становится более острым при переходе к экранам с большим разрешением, которые используются на устройствах Windows Phone 7. Экран размером 480 × 800 имеет по диагонали 933 пиксела. Экран телефона, который я использовал для выполнения примеров для данной книги, равен примерно 3½", что обеспечивает плотность пикселов около 264 DPI. (Разрешение экрана обычно выражается числом, кратным 24.) Грубо говоря, это разрешение почти в два с половиной раза больше, чем разрешение обычных мониторов.

Это не означает, что все размеры шрифтов, используемые для обычных экранов, должны обязательно быть увеличены в два с половиной раза для телефона. Чем выше разрешение экрана телефона – и меньше расстояние просмотра, что свойственно телефонам – тем меньшие шрифты могут применяться.

Для Веб-браузера в Silverlight по умолчанию применяется *FontSize*, равный 11 пикселам, что соответствует размеру шрифта 8,25 пункта. Это достаточно для монитора настольного компьютера, но несколько маловато для телефона. Поэтому Silverlight для Windows Phone определяет коллекцию типовых размеров шрифтов. (Подробнее эти вопросы рассматриваются в главе 7.) Стандартный файл MainPage.xaml включает следующий атрибут в корневом элементе:

```
FontSize="{StaticResource PhoneFontSizeNormal}"
```

FontSize наследуется через дерево визуальных элементов и применяется ко всем элементам *TextBlock*, для которых не задано собственное свойство *FontSize*. Его значение – 20 пикселов, что почти в два раза больше стандартного в Silverlight *FontSize*, применяемого для настольных приложений. Согласно стандартным формулам *FontSize*, равный 20 пикселам,

соответствует 15 пунктам. Но размер фактически отображаемого на экране телефона шрифта составляет примерно 2/5 того размера, какой имел бы шрифт в 15 пунктов в печатном варианте.

Фактическая высота *TextBlock*, отображающего текст такого размера, примерно на 33% больше *FontSize*. В данном случае это около 27 пикселов.

XAP – это ZIP

В каталоге \bin\Debug проекта, созданного Visual Studio для SilverlightHelloPhone, можно увидеть файл SilverlightHelloPhone.xap. Это так называемый XAP-файл (произносится «зап»). Именно этот файл развертывается на телефоне или эмуляторе телефона.

XAP-файл – это пакет файлов, упакованных в очень популярном формате сжатия ZIP. (Если требуется найти в толпе разработчиков на Silverlight, просто прокричите: «XAP – это ZIP».) Чтобы заглянуть внутрь файла, переименуйте SilverlightHelloPhone.xap в SilverlightHelloPhone.zip. Там вы обнаружите несколько файлов растровых изображений, являющихся частью проекта, XML-файл, XAML-файл и файл SilverlightHelloPhone.dll, который является скомпилированным двоичным файлом (кодом) приложения.

Все ресурсы, необходимые приложению, можно сделать частью проекта Visual Studio и добавить в этот XAP-файл. Приложение будет выполнять доступ к этим файлам во время выполнения. Основные концепции рассмотрим в главе 4.

Приложение для телефона на XNA

Далее у нас по плану приложение на XNA, отображающее небольшое приветствие в центре экрана. Тогда как в приложениях на Silverlight текст обычно превалирует, в видеоиграх его встретишь не часто. В играх роль текста сведена к описанию правил или отображению счета. Поэтому сама концепция приложения «здравствуй, мир» не вполне вписывается в общую идеологию программирования на XNA.

В XNA даже нет встроенных шрифтов. И приложение на XNA, выполняющееся на телефоне, не может использовать те же встроенные шрифты телефона, что и программы на Silverlight, как это можно было бы предположить. Silverlight применяет векторные шрифты TrueType, а XNA ничего не знает о таких экзотических концепциях. Для XNA все, включая шрифты, является растровыми изображениями.

Если в приложении на XNA требуется использовать определенный шрифт, он должен быть встроен в исполняемый файл как коллекция растровых изображений для каждого символа. XNA Game Studio (которая интегрирована в Visual Studio) очень упрощает сам процесс встраивания шрифта, но тут возникают некоторые серьезные правовые проблемы. Вы можете легально распространять приложение на XNA, использующее тот или иной шрифт, только при условии, если имеете право на распространение этого встроенного шрифта, а это невозможно для большинства шрифтов, поставляемых с самой Windows или приложениями Windows.

Чтобы помочь в решении этого правового затруднения, Майкрософт предоставляет лицензию на использование шрифтов Ascender Corporation именно в целях их применения в приложениях на XNA. Вот эти шрифты:

Kootenay	Lindsey
Miramonte	Pescadero
Miramonte Bold	**Pescadero Bold**

Pericles	Segoe UI Mono
Pericles Light	**Segoe UI Mono Bold**

Обратите внимание, что в шрифте Pericles в качестве строчных букв используются уменьшенные заглавные, поэтому, вероятно, он подойдет только для заголовков.

В Visual Studio в меню File выберите New и Project. В левой части диалогового окна выберите Visual C# и XNA Game Studio 4.0. В середине выберите Windows Phone Game (4.0). Задайте месторасположение и имя проекта, XnaHelloPhone.

Visual Studio создает два проекта, один для логики приложения и другой для его содержимого. Приложения на XNA обычно включают большой объем содержимого, которым преимущественно являются растровые изображения и трехмерные модели, но также и шрифты. Чтобы добавить шрифт в это приложение, щелкните правой кнопкой мыши проект, созданный для содержимого (он обозначен «XnaHelloPhoneContent (Content)»), и во всплывающем меню выберите Add (Добавить) и New Item (Новый элемент). Выберите Sprite Font, оставьте имя файла как есть, SpriteFont1.spritefont, и щелкните Add.

Слово «спрайт» («sprite» в переводе на русский означает «эльф») широко распространено в игровых приложениях и обычно обозначает небольшое растровое изображение, которое может очень быстро перемещаться (так же как эльфы, живущие в волшебном лесу). В XNA даже шрифты являются спрайтами.

SpriteFont1.spritefont появится в списке файлов каталога Content, и вы можете редактировать изобилующий комментариями XML-файл, описывающий шрифт.

Проект XNA: XnaHelloPhone Файл: SpriteFont1.spritefont (полностью за исключением комментариев)

```
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">
    <FontName>Segoe UI Mono</FontName>
    <Size>14</Size>
    <Spacing>0</Spacing>
    <UseKerning>true</UseKerning>
    <Style>Regular</Style>
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Между тегами *FontName* указан шрифт Segoe UI Mono, но его можно заменить любым другим шрифтом, из приведенных ранее. Если желаете использовать Pericles Light, укажите его полное имя, но для Miramonte Bold, Pescadero Bold или Segoe UI Mono Bold необходимо написать просто Miramonte, Pescadero или Segoe UI Mono и ввести Bold (Полужирный) между тегами Style (Стиль). Bold может использоваться и для других шрифтов, но для них он будет синтезирован, тогда как для Miramonte, Pescadero или Segoe UI Mono будет использоваться специально разработанный полужирный шрифт.

Теги *Size* (Размер) обозначают размер шрифта в пунктах. В XNA, как и в Silverlight, координаты и размеры задаются в пикселах, но в XNA за основу при преобразовании между пикселами и пунктами взято разрешение 96 DPI. Для XNA-приложения 14 шрифт равен 18-

2/3 пикселам. Это очень близко к шрифту размером 15 пунктов или с *FontSize* равным 20 пикселов в Silverlight для Windows Phone.

В разделе *CharacterRegions* (Диапазоны символов) файла указываются диапазоны шестнадцатеричных кодировок Unicode. По умолчанию используется диапазон от 0x32 до 0x126, что включает все обычные символы набора символов ASCII.

SpriteFont1.spritefont не является достаточно описательным именем. Я бы назвал файл так, чтобы было понятно, о каком шрифте идет речь. Если сохраняются стандартные настройки шрифта, можно переименовать файл в Segoe14.spritefont. Если взглянуть на свойства этого файла – щелкните правой кнопкой мыши имя файла и выберите Properties – можно увидеть, что значением Asset Name (Имя ресурса) является имя файла без расширения: Segoe14. Значение Asset Name используется для ссылки на шрифт в приложении. Если хотите запутать себя, можете изменить Asset Name и задать ему значение, отличное от имени файла.

Изначально проект XNAHelloPhone включает два C#-файла: Program.cs и Game1.cs. Первый очень простой и, как выясняется, не имеет отношения к играм для Windows Phone 7! Директива препроцессора активирует класс *Program* (Программа), только если определен символ WINDOWS или XBOX. При компиляции программ для Windows Phone вместо них задается символ WINDOWS_PHONE.

Чаще всего при создании небольших игр основная часть времени уходит на файл Game1.cs. Класс *Game1* наследуется от *Game* (Игра). Первоначально в нем определены два поля: *graphics* (графические элементы) и *spriteBatch* (Пакет спрайтов). К этим двум полям я хочу добавить еще три:

Проект XNA: XnaHelloPhone Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
namespace XnaHelloPhone
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        string text = "Hello, Windows Phone 7!";
        SpriteFont segoe14;
        Vector2 textPosition;
        …
    }
}
```

В этих трех новых полях просто указан текст для отображения, используемый для этого шрифт и месторасположения текста на экране. Координаты задаются в пикселах относительно верхнего левого угла экрана. Структура *Vector2* имеет два поля: *X* и *Y*, типа *float* (число с плавающей точкой). В целях обеспечения лучшей производительности в XNA все значения с плавающей точкой берутся с одинарной точностью. (В Silverlight – с двойной точностью.) Структура *Vector2* часто используется для задания точек, размеров и даже векторов в двухмерном пространстве.

При запуске игры на телефоне, создается экземпляр класса *Game1* и выполняется конструктор *Game1*. Рассмотрим стандартный код:

Проект XNA: XnaHelloPhone Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
```

```
    Content.RootDirectory = "Content";

    // По умолчанию частота кадров для Windows Phone составляет 30 кадр/с.
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

Первое выражение обеспечивает инициализацию поля *graphics*. Во втором выражении для *Game* определяется свойство *Content* (Содержимое) типа *ContentManager* (Диспетчер содержимого), и *RootDirectory* (Корневой каталог) является свойством этого класса. Значение «Content» этого свойства соответствует папке Content, в которой хранится шрифт Segoe размером 14 пунктов. В третьем выражении задается время игрового цикла программы, что управляет частотой обновления изображения. Экраны устройств Windows Phone 7 обновляются с частотой 30 кадров в секунду.

Когда экземпляр *Game1* создан, вызывается его метод *Run* (Выполнить), и базовый класс *Game* инициирует процесс запуска игры. Один из первых шагов – вызов метода *Initialize* (Инициализировать), который может быть перегружен в производных от *Game* классах. XNA Game Studio автоматически формирует скелетный метод, в который я не буду ничего добавлять:

Проект XNA: XnaHelloPhone   Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    base.Initialize();
}
```

В методе *Initialize* шрифт или любое другое содержимое не должно загружаться. Это происходит несколько позже, когда базовый класс вызывает метод *LoadContent* (Загрузить содержимое).

Проект XNA: XnaHelloPhone   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    Vector2 textSize = segoe14.MeasureString(text);
    Viewport viewport = this.GraphicsDevice.Viewport;

    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);

}
```

Первое выражение данного метода формируется автоматически. Вскоре мы увидим, как этот объект *spriteBatch* используется для вывода спрайтов на экран.

Все остальные выражения были добавлены мной. Как можно заметить, перед всеми именами свойств, таких как *Content* и *GraphicsDevice* (Графическое устройство), я поставил ключевое слово *this*, чтобы напомнить себе, что это свойства, а не статические классы. Как уже говорилось, свойство *Content* типа *ContentManager*. Универсальный метод *Load* (Загрузить) обеспечивает загрузку содержимого в приложение, в данном случае, это содержимое типа *SpriteFont*. Имя, указанное в кавычках, соответствует Asset Name, указанному в свойствах содержимого. Это выражение обеспечивает сохранение результата в поле *segoe14* типа *SpriteFont*.

В XNA спрайты (включая текстовые строки) обычно позиционируются через задание координат в пикселах верхнего левого угла спрайта относительно верхнего левого угла экрана. Для расчета этих координат необходимо знать и размер экрана, и размер текста при отображении его конкретным шрифтом.

У класса *SpriteFont* есть чрезвычайно удобный метод *MeasureString* (Измерить строку), возвращающий объект *Vector2* с размером конкретной текстовой строки в пикселах. (Для шрифта Segoe UI Mono размером 14 пунктов, высота которого эквивалентна 18-2/3 пикселам, метод *MeasureString* возвратит высоту 28 пикселов.)

Как правило, для получения размера экрана в приложении на XNA используется свойство *Viewport* (Окно просмотра) класса *GraphicsDevice*. Оно доступно через свойство *GraphicsDevice* класса *Game* и предоставляет свойства *Width* (Ширина) и *Height* (Высота).

После этого довольно просто вычислить *textPosition* (Положение текста) – координаты точки относительно верхнего левого угла окна просмотра, в которой будет располагаться верхний левый угол текстовой строки.

На этом этап инициализации программы завершается, и начинается фактическое действие. Приложение входит в *игровой цикл*. Синхронно с обновлением экрана, которое происходит с частотой 30 кадров в секунду, в приложении вызываются два метода: *Update* (Обновить) и за ним *Draw* (Рисовать). Снова и снова: *Update*, *Draw*, *Update*, *Draw*, *Update*, *Draw*…. (На самом деле, все несколько сложнее; методу *Update* необходимо 1/30 секунды для выполнения, но мы обсудим вопросы хронометража более подробно в одной из следующих глав.)

Метод *Draw* обеспечивает отрисовку образов на экране. И это *все*, что он может делать. Все подготовительные вычисления для отрисовки должны осуществляться в методе *Update*. Метод *Update* подготавливает программу к выполнению метода *Draw*. Очень часто приложения на XNA реализуют перемещение спрайтов по экрану на основании пользовательского ввода. Для телефона пользовательский ввод осуществляется преимущественно посредством сенсорного ввода. Вся обработка пользовательского ввода также должна происходить во время выполнения метода *Update*. Пример этому рассмотрим в главе 3.

Методы *Update* и *Draw* должны быть написаны так, чтобы они выполнялись максимально быстро. Полагаю, это само собой разумеется. Однако здесь имеются также некоторые очень важные моменты, которые могут быть не так очевидны.

Следует избегать включения в методы *Update* и *Draw* кода, выполняющего рутинные операции по распределению памяти из локальной кучи приложения. В определенный момент времени сборщик мусора .NET захочет вернуть часть этой памяти, и пока он будет выполнять свою работу, игра может немного притормаживать. В главах, посвященных разработке на XNA, будут представлены способы избежать распределения памяти из кучи.

Скорее всего, в методах *Draw* не будет возникать никаких проблем. Обычно все неприятности кроются в методе *Update*. Избегайте применения выражений *new* для классов. Это всегда приводит к распределению памяти. Однако нет ничего страшного в создании экземпляров структур, потому что эти экземпляры хранятся в стеке, а не в куче. (XNA использует структуры, а не классы, для многих типов объектов, необходимых в *Update*.) Но распределение памяти из кучи также может происходить и без явных выражений *new*. Например, конкатенация двух строк приводит к созданию новой строки в куче. Для выполнения каких-либо операций со строками в *Update* следует использовать *StringBuilder* (Построитель строк). Очень удобно, что XNA предоставляет для отображения текста методы, использующие объекты *StringBuilder*.

Но в нашем приложении XnaHelloPhone метод *Update* абсолютно тривиальный. Отображаемый текст зафиксирован в одной единственной точке. Все необходимые вычисления уже выполнены в методе *LoadContent*. Поэтому оставляем метод *Update* без изменений, просто в том виде, в каком он был изначально создан XNA Game Studio:

Проект XNA: XnaHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    base.Update(gameTime);
}
```

В формируемом по умолчанию коде для проверки события нажатия кнопки Back использует статический класс *GamePad* (Игровой планшет). Это событие является сигналом к выходу из игры.

И, наконец, метод *Draw*. Его автоматически созданная версия просто закрашивает фон голубым цветом:

Проект XNA: XnaHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}
```

Васильковый цвет (CornflowerBlue) приобрел культовый статус в сообществе разработчиков на XNA. При работе над программой на XNA увидеть голубой экран очень утешительно, поскольку это означает, что программа, по крайней мере, дошла до метода *Draw*. Но в целях энергосбережения при использовании ОСИД-экранов желательно применять более темные фоны. Я нашел компромиссный вариант и сделал фон темно-синим. Как и Silverlight, XNA поддерживает 140 цветов, которые уже считаются стандартными. Выводимый текст будет белого цвета:

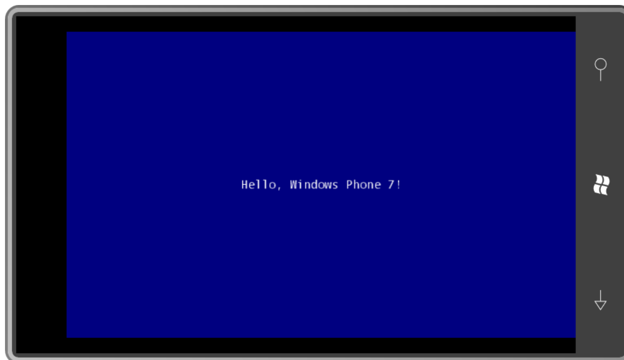
Проект XNA: XnaHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, text, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Спрайты выводятся на экран пакетами в составе объекта *SpriteBatch*, который был создан во время вызова метода *LoadContent*. Между вызовами *Begin* (Начало) и *End* (Конец) может осуществляться множество вызовов метода *DrawString* (Отрисовать строку) для отрисовки текста и *Draw* для отрисовки растровых изображений. Вызываться могут только эти методы. В данном конкретном вызове *DrawString* указан шрифт, выводимый текст, местоположение

верхнего левого угла текста относительно верхнего левого угла экрана и цвет. И вот, что мы получаем:

Вот это любопытно! По умолчанию программы на Silverlight отображаются в портретном режиме, а программы на XNA – в альбомном. Давайте повернем телефон или эмулятор:



Намного лучше!

Но тут возникает вопрос: всегда ли приложения на Silverlight выполняются в портретном режиме, а приложения на XNA – в альбомном?

Глава 2
Ориентация

По умолчанию, программы на Silverlight для Windows Phone 7 выполняются в портретном режиме, а программы на XNA – в альбомном. В данной главе рассматривается, как изменить это поведение по умолчанию, и изучаются другие вопросы, касающиеся размеров экрана, размеров элементов и событий.

Silverlight и динамическая компоновка

Если запустить приложение SilverlightHelloPhone из предыдущей главы и повернуть телефон или эмулятор на бок, обнаружится, что изображение не меняет расположения в зависимости от ориентации экрана. Это легко исправить. В корневом теге *PhoneApplicationPage* файла MainPage.xaml замените значение атрибута

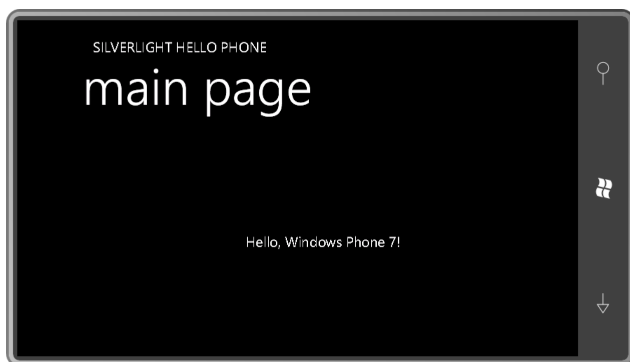
```
SupportedOrientations="Portrait"
```

на:

```
SupportedOrientations="PortraitOrLandscape"
```

SupportedOrientations (Поддерживаемые ориентации) – это свойство *PhoneApplicationPage*. В качестве его значения может быть задан один из элементов перечисления *SupportedPageOrientation* (Поддерживаемые ориентации страницы): *Portrait*, *Landscape* или *PortraitOrLandscape*.

Выполняем компиляцию еще раз. Теперь при повороте телефона или эмулятора на бок соответствующим образом разворачивается и содержимое страницы:



Свойство *SupportedOrientations* в случае необходимости также позволяет обеспечить только *альбомный* режим отображения содержимого.

Такое изменение ориентации изображения является превосходной демонстрацией возможностей динамической компоновки в Silverlight. Все элементы изменили местоположение, и некоторые из них даже изменили размер. Silverlight берет начало в WPF и настольных технологиях, поэтому исторически в него были заложены возможности реагировать на изменения размеров и пропорций окна, которые прекрасно переносятся в приложения для телефонов.

Двумя самыми важными свойствами при работе с динамической компоновкой являются *HorizontalAlignment* и *VerticalAlignment*. В предыдущей главе использовать эти свойства в

приложении на Silverlight для расположения текста по центру было, безусловно, проще, чем выполнять вычисления на основании размеров экрана и текста, как это требовалось в XNA.

С другой стороны, если сейчас вам будет поставлена задача разместить ряд строк текста, скорее всего, вы подсчитаете, что сделать это в XNA проще, чем в Silverlight.

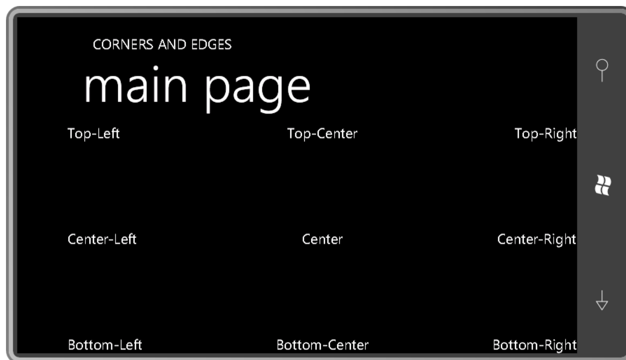
Уверяю вас, что в Silverlight тоже имеются средства для организации элементов. Исключительно для этой цели существует отдельная категория элементов под названием *панели*. Элементы можно позиционировать даже с использованием заданных в пикселах координат, если так вам удобнее. Но полностью и во всех подробностях панели рассматриваются только в главе 9.

А пока попытаемся разместить множество элементов в сетке для содержимого. Обычно *Grid* организует свое содержимое в ячейки, идентифицируемые строками и столбцами, но данная программа помещает девять элементов *TextBlock* в *Grid* с одной ячейкой для демонстрации использования свойств *HorizontalAlignment* и *VerticalAlignment* в девяти различных сочетаниях:

Проект Silverlight: SilverlightCornersAndEdges Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Top-Left"
               VerticalAlignment="Top"
               HorizontalAlignment="Left" />

    <TextBlock Text="Top-Center"
               VerticalAlignment="Top"
               HorizontalAlignment="Center" />

    <TextBlock Text="Top-Right"
               VerticalAlignment="Top"
               HorizontalAlignment="Right" />

    <TextBlock Text="Center-Left"
               VerticalAlignment="Center"
               HorizontalAlignment="Left" />

    <TextBlock Text="Center"
               VerticalAlignment="Center"
               HorizontalAlignment="Center" />

    <TextBlock Text="Center-Right"
               VerticalAlignment="Center"
               HorizontalAlignment="Right" />

    <TextBlock Text="Bottom-Left"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Left" />

    <TextBlock Text="Bottom-Center"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Center"  />

    <TextBlock Text="Bottom-Right"
               VerticalAlignment="Bottom"
               HorizontalAlignment="Right" />
</Grid>
```

Как и во многих более простых приложениях на Silverlight этой книги, я задал свойству *SupportedOrientations* объекта *MainPage* значение *PortraitOrLandscape*. И вот, как это выглядит, если развернуть телефон или эмулятор:



Кажется, что на экране представлены все возможные сочетания. На самом деле здесь *не* отображены *настройки по умолчанию* для свойств *HorizontalAlignment* и *VerticalAlignment*. По умолчанию в качестве значений этих свойств применяются элементы перечисления *Stretch* (Растянуть). При использовании значений по умолчанию *TextBlock* расположится в верхнем левом углу, так как если бы были заданы значения *Top* (Сверху) и *Left* (Слева). Но не так очевидно будет то, что *TextBlock* занимает всю площадь *Grid*. *TextBlock* имеет прозрачный фон (и изменить его никак нельзя), поэтому заметить разницу не так просто, но я продемонстрирую это в следующей главе.

Несомненно, свойства *HorizontalAlignment* и *VerticalAlignment* играют важную роль в системе компоновки в Silverlight. Это же можно сказать о свойстве *Margin* (Поле). Добавим *Margin* в первый *TextBlock* этого приложения:

```
<TextBlock Text="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           Margin="100" />
```

Теперь между *TextBlock* и левым и верхним краями клиентской области имеется отступ в 100 пикселов. Свойство *Margin* типа *Thickness* (Толщина) – это структура с четырьмя свойствами: *Left*, *Top*, *Right* (Справа) и *Bottom* (Снизу). Если в XAML задать всего одно число, это значение будет использоваться для всех четырех сторон. Также можно задать два значения:

```
Margin="100 200"
```

Первое из них применяется к правому и левому полям, второе – к нижнему и верхнему. Если задано четыре значения

```
Margin="100 200 50 300"
```

они применяются в порядке: левое, верхнее, правое и нижнее. Внимание: если поля слишком велики, текст может быть полностью или частично перекрыт ими. Silverlight сохраняет поля даже ценой усечения элементов.

Если обоим свойствам, *HorizontalAlignment* и *VerticalAlignment*, задать значение *Center* (Центр), и задать для *Margin* четыре разных числа, визуально текст уже не будет находиться в центре области содержимого. Silverlight выполняет центрирование на основании размера элемента, включая поля.

TextBlock также имеет свойство *Padding* (Отступ):

```
<TextBlock Text="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           Padding="100 200" />
```

Padding также типа *Thickness*, и при использовании с *TextBlock* визуально невозможно отличить *Padding* от *Margin*. Но это, несомненно, разные вещи: *Margin* – это пространство вне

TextBlock, *Padding* – пространство внутри *TextBlock*, не занятое текстом. При использовании *TextBlock* для обработки событий касания (как будет показано в следующей главе), он будет реагировать на касания в области *Padding*, тогда как касания в области *Margin* будут игнорироваться.

Свойство *Margin* определяется в классе *FrameworkElement*. В реальных приложениях на Silverlight практически всем элементам задается *Margin*, отличное от нуля. Это предотвращает наложение элементов друг на друга. Свойство *Padding* используется реже. Оно задается только для *TextBlock*, *Border* (Рамка) и *Control*.

Margin может использоваться для позиционирования множества элементов в *Grid* с одной ячейкой. Обычно так не делают, для этого есть намного более удобные способы, но это возможно. Я приведу такой пример в главе 5.

Важно понимать, чего мы *не* делам. Мы не задаем явно *Width* и *Height* элемента *TextBlock*, как это происходит в некоторых устаревших программных средах:

```
<TextBlock Text="Top-Left"
           VerticalAlignment="Top"
           HorizontalAlignment="Left"
           Width="100"
           Height="50" />
```

Мы угадываем размер *TextBlock*, не располагая той информацией об элементе, какой располагает сам *TextBlock*. В некоторых случаях *Width* и *Height* необходимо задавать, но не здесь.

Свойства *Width* и *Height* типа *double*. Значениями по умолчанию для них являются специальные значения с плавающей точкой Not a Number[1] или NaN. Если требуется получить *фактическую* ширину и высоту элемента, когда он отображается на экране, необходимо обратиться к свойствам *ActualWidth* (Фактическая ширина) и *ActualHeight* (Фактическая высота). (Но, внимание, эти свойства будут иметь ненулевые значения, только после того, как элемент выведен на экран.)

Для получения сведений, касающихся размеров элементов, предоставляются некоторые полезные события. Событие *Loaded* (Загружен) формируется, когда визуальные элементы впервые выведены на экран. Событие *SizeChanged* (Размер изменен) поддерживается элементами для оповещения об изменении ими размера. Событие *LayoutUpdated* (Компоновка обновлена) используется для оповещения об изменении компоновки, как это происходит при изменении ориентации.

Проект SilverlightWhatSize демонстрирует применение метода *SizeChanged* для отображения размеров нескольких элементов на стандартной странице. Необходимость в таких точных значениях размеров возникает не часто, но порой они могут представлять интерес.

Событие может быть ассоциировано с обработчиком события прямо в XAML, но сам обработчик события должен быть реализован в коде. При вводе имени события в XAML (например, *SizeChanged*) Visual Studio предложит создать обработчик события. Это и было сделано мною с событием *SizeChanged* сетки для содержимого:

Проект Silverlight: SilverlightWhatSize   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      SizeChanged="ContentPanel_SizeChanged">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
```

[1] Не число (*прим. переводчика*).

```
                VerticalAlignment="Center" />
</Grid>
```

Свойству *Name* (Имя) элемента *TextBlock* присвоено значение «txtblk». Свойство *Name* играет очень особую роль в Silverlight. Если сейчас мы выполним компиляцию приложения и заглянем в файл MainPage.g.cs – файл кода, автоматически формируемый компилятором на основании файла MainPage.xaml – то увидим в классе *MainPage* ряд полей, среди которых будет и поле *txtblk* типа *TextBlock*:

```
internal System.Windows.Controls.TextBlock txtblk;
```

Также мы заметим, что данное значение этому полю задается программно в методе *InitializeComponent*:

```
this.txtblk = ((System.Windows.Controls.TextBlock)(this.FindName("txtblk")));
```

Это означает, что в любой момент после вызова метода *InitializeComponent* конструктором MainPage.xaml.cs любой код класса *MainPage* может ссылаться на этот элемент *TextBlock* в файле XAML, используя переменную *txtblk*, которая хранится как поле класса.

В файле MainPage.xaml обратите внимание, что некоторым элементам имена присваиваются с использованием синтаксиса *x:Name*, а не *Name*. В XAML эти два атрибута практически эквивалентны. Только *Name* применяется исключительно для элементов (т.е. экземпляров классов, производных от *FrameworkElement*, потому что именно в нем описано свойство *Name*), а *x:Name* годится для всего.

Это означает, что в коде класса MainPage в файле MainPage.xaml.cs имеется поле *ContentPanel*, предусмотренное для ссылки на стандартный *Grid* из MainPage.xaml, и то же самое для остальных элементов MainPage.xaml.

Присваивание имен элементам – один из двух основных способов взаимодействия кода и XAML. Второй способ – обработка в коде событий, которые формируются элементами, описанными в XAML. Рассмотрим обработчик события *SizeChanged* сетки для содержимого, который Visual Studio формирует автоматически:

Проект Silverlight: SilverlightWhatSize   Файл: MainPage.xaml.cs (фрагмент)

```
private void ContentPanel_SizeChanged(object sender, SizeChangedEventArgs e)
{

}
```

Мне не нравятся обработчики, создаваемые Visual Studio. Как правило, я удаляю ключевое слово *private*, переименовываю обработчики событий так, чтобы их имена начинались со слова *On*, и убираю подчеркивания. Этот обработчик я назвал бы *OnContentPanelSizeChanged* (При изменении размера панели для содержимого). Также обычно я заменяю аргументы событий *e* на *args*.

Но для данного приложения я оставлю все как есть. Первым в этот метод передается аргумент *sender* (отправитель). Это элемент, сформировавший событие, которым в данном случае является *Grid* под именем *ContentPanel*. Второй аргумент включает данные, касающиеся конкретного события.

Я добавил тело этого метода, в котором свойству *Text* элемента *txtblk* просто присваивается более длинная строка, состоящая из нескольких строк:

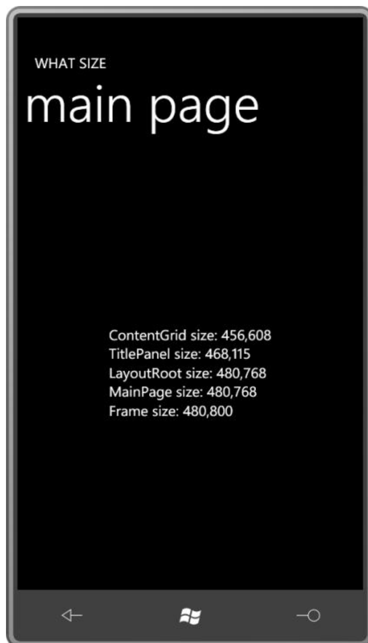
Проект Silverlight: SilverlightWhatSize Файл: MainPage.xaml.cs (фрагмент)

```
private void ContentPanel_SizeChanged(object sender, SizeChangedEventArgs e)
{
    txtblk.Text = String.Format("ContentPanel size: {0}\n" +
                                "TitlePanel size: {1}\n" +
                                "LayoutRoot size: {2}\n" +
                                "MainPage size: {3}\n" +
                                "Frame size: {4}",
                                e.NewSize,
                                new Size(TitlePanel.ActualWidth,
TitlePanel.ActualHeight),
                                new Size(LayoutRoot.ActualWidth,
LayoutRoot.ActualHeight),
                                new Size(this.ActualWidth, this.ActualHeight),
                                Application.Current.RootVisual.RenderSize);
}
```

Эти пять элементов типа *Size*, который является структурой со свойствами *Width* и *Height*. Размер самого *ContentPanel* доступен через свойство *NewSize* (Новый размер) аргументов события. Для следующих трех элементов я использовал свойства *ActualWidth* и *ActualHeight*.

Обратите внимание на последний элемент. Статическое свойство *Application.Current* возвращает объект *Application*, ассоциированный с текущим процессом. Это объект *App*, созданный программой. Он имеет свойство *RootVisual* (Корневой визуальный элемент), которое ссылается на рамку, но определено типа *UIElement*. Свойства *ActualWidth* и *ActualHeight* описаны *FrameworkElement*; это класс, наследуемый от *UIElement*. Вместо приведения я решил использовать свойство типа *Size*, описываемое классом *UIElement*.

Первое событие *SizeChanged* возникает при создании страницы и расстановке ее элементов, т.е. когда сетка для содержимого меняет свой размер от 0 до некоторого конечного значения:



Размер *MainPage* на 32 пиксела меньше размера рамки, как раз достаточно для размещения панели задач вверху экрана. Чтобы панель задач не отображалась во время выполнения приложения (и, таким образом, приложение имело в своем распоряжении весь экран целиком), замените значение атрибута корневого элемента MainPage.xaml:

```
shell:SystemTray.IsVisible="True"
```

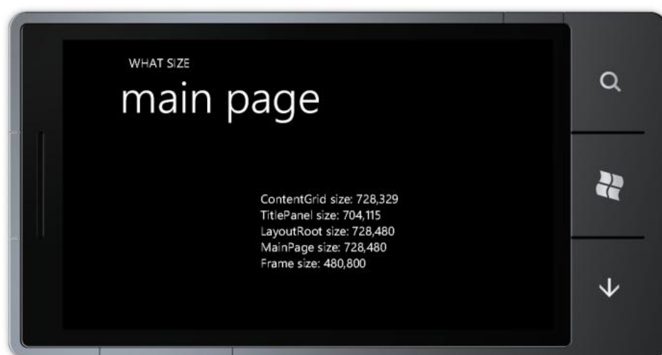
на

```
shell:SystemTray.IsVisible="False"
```

Синтаксис этого атрибута может казаться несколько странным. *SystemTray* (Панель задач) – это класс из пространства имен *Microsoft.Phone.Shell*, и *IsVisible* (Является видимым) – свойство этого класса. Класс и свойство появляются вместе, потому что это свойство особого типа, называемое *присоединенным свойством* (*attached property*). Более подробно присоединенные свойства будут рассмотрены в главе 9.

Самый верхний *Grid* под именем *LayoutRoot* имеет тот же размер, что и *MainPage*. Размер *TitlePanel* (Панель заголовков) (включает два заголовка) по вертикали и размер *ContentPanel* по вертикали в сумме не соответствуют размеру *LayoutRoot* по вертикали, поскольку *TitlePanel* имеет по вертикали поле в 45 пикселов (17 пиксела сверху и 28 пикселов снизу).

Последующие события *SizeChanged* возникают, когда какой-то элемент дерева визуальных элементов обусловливает изменение размеров, или когда меняется ориентация телефона:



Обратите внимание, что рамка не меняет ориентацию. В альбомном режиме панель задач занимает 72 пиксела ширины *MainPage*.

События изменения ориентации экрана

Во многих приложениях на Silverlight, приводимых в данной книге, свойству *SupportedOrientations* будет задано значение *PortraitOrLandscape*, я постараюсь создавать не зависящие от ориентации экрана приложения. Для приложений на Silverlight, которые принимают текстовый ввод, критически важно, чтобы ориентация приложения соответствовала расположению клавиатуры (если таковая имеется). А то, как будет располагаться клавиатура, предвидеть невозможно.

Очевидно, что обработка изменений ориентации экрана выходит далеко за рамки задания свойства *SupportedOrientations*! В некоторых случаях требуется менять компоновку из кода класса страницы. Для реализации особой обработки в *PhoneApplicationFrame* и *PhoneApplicationPage* предусмотрены события *OrientationChanged* (Ориентация изменена). *PhoneApplicationPage* дополняет это событие удобным и защищенным методом *OnOrientationChanged* (При изменении ориентации), доступным для переопределения.

В классе *MainPage* в проекте SilverlightOrientationDisplay показано, как переопределяется *OnOrientationChanged*, но он используется лишь для вывода на экран текущей ориентации. Сетка для содержимого в этом проекте включает простой *TextBlock*:

Проект Silverlight: SilverlightOrientationDisplay Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Рассмотрим полный файл выделенного кода. Конструктор при инициализации *TextBlock* присваивает его свойству *Text* текущее значение свойства *Orientation* (Ориентация), которое является элементом перечисления *PageOrientation* (Ориентация страницы):

Проект Silverlight: SilverlightOrientationDisplay Файл: MainPage.xaml.cs

```
using System.Windows.Controls;
using Microsoft.Phone.Controls;

namespace SilverlightOrientationDisplay
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
            txtblk.Text = Orientation.ToString();
        }

        protected override void OnOrientationChanged(OrientationChangedEventArgs
args)
        {
            txtblk.Text = args.Orientation.ToString();
            base.OnOrientationChanged(args);
        }
    }
}
```

Метод *OnOrientationChanged* получает новое значение из аргументов события.

Ориентация в приложении на XNA

По умолчанию в приложениях на XNA для Windows Phone используется альбомная ориентация, возможно, для обеспечения совместимости с другими экранами, используемыми для игр. Поддерживается альбомная ориентация во всех направлениях, так что при переворачивании устройства как на левый, так и на правый бок, ориентация изображения на экране будет меняться соответственно. Если вы предпочитаете игры с портретным расположением, изменить эту настройку не составляет труда. Добавим в конструктор класса *Game1* приложения XnaHelloPhone следующие выражения:

```
graphics.PreferredBackBufferWidth = 320;
graphics.PreferredBackBufferHeight = 480;
```

Задний буфер – это область, в которой XNA создает графические элементы, выводимые на экран методом *Draw*. И размером, и пропорциями этого буфера можно управлять. Поскольку заданная здесь ширина буфера меньше его высоты, XNA предполагает, что изображение требуется выводить в портретном режиме:

Посмотрите на это! Соотношение размеров заднего буфера отличаются от пропорций экрана устройства Windows Phone 7, поэтому изображение выводится с черными полосами вверху и внизу экрана! Текст имеет тот же размер в пикселах, но выглядит больше, потому что разрешение экрана уменьшилось.

Даже пусть вы не являетесь большим поклонником такой зернистости изображения, вызывающей ностальгические воспоминания, но подумайте о применении меньшего заднего буфера, если игре не требуется такое высокое разрешение, какое предоставляет экран телефона. Это обеспечит повышение производительности и снизит энергопотребление. Размер заднего буфера может быть любым в диапазоне от 240 × 240 до 480 × 800 (для портретного режима) или 800 × 480 (для альбомного). XNA использует соотношение размеров для определения используемого режима отображения.

Задание необходимого заднего буфера является замечательным способом ориентировать приложение на экран определенного размера в коде, но также обеспечивает возможности применения устройств с другими размерами экранов, которые могут появиться в будущем.

По умолчанию размер заднего буфера равен 800 × 480, но его фактический размер на экране несколько меньше, поскольку требуется обеспечить место для панели задач. Чтобы избавиться от панели задач (и досадить пользователям, которые всегда хотят знать, который сейчас час), в конструкторе *Game1* можно задать:

```
graphics.IsFullScreen = true;
```

Кроме того, можно сделать так, чтобы игры на XNA реагировали на изменения ориентации экрана, но для этого, конечно же, придется немного изменить их структуру. Самый простой тип реструктуризации для обеспечения учета изменения ориентации продемонстрирован в проекте XnaOrientableHelloPhone. Его поля теперь включают переменную *textSize* (Размер текста):

Проект XNA: XnaOrientableHelloPhone Файл: Game1.cs (фрагмент демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
```

```
    string text = "Hello, Windows Phone 7!";
    SpriteFont segoe14;
    Vector2 textSize;
    Vector2 textPosition;
    …
}
```

Конструктор *Game1* включает выражение, определяющее свойство *SupportedOrientations* поля *graphics*:

Проект XNA: XnaOrientableHelloPhone Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Делаем возможным отображение и в портретном режиме
    graphics.SupportedOrientations = DisplayOrientation.Portrait |
                                     DisplayOrientation.LandscapeLeft |
                                     DisplayOrientation.LandscapeRight;

    // По умолчанию для Windows Phone частота кадров составляет 30 кадр/с.
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

Применяя *SupportedOrientation*, можно ограничить поддерживаемые телефоном режимы отображения. Это выражение, обеспечивающее поддержку и портретного, и альбомного режима отображения, выглядит простым, но здесь имеются некоторые побочные эффекты. При изменении ориентации происходит сброс графического устройства (что приводит к формированию некоторых событий), и размеры заднего буфера изменяются. Можно подписаться на событие *OrientationChanged* класса *GameWindow* (Игровое окно) (которое доступно через свойство *Window* (Окно)) либо проверять свойство *CurrentOrientation* (Текущая ориентация) объекта *GameWindow*.

Я выбрал несколько иной подход. Рассмотрим новый метод *LoadContent*, который, как можно будет заметить, принимает размер текста и сохраняет его как поле, но не получает размеров окна просмотра.

Проект XNA: XnaOrientableHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    textSize = segoe14.MeasureString(text);
}
```

Параметры окна просмотра можно получить в ходе выполнения метода *Update*, поскольку размеры окна просмотра отражают ориентацию экрана.

Проект XNA: XnaOrientableHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечивает возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
```

```
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);
    base.Update(gameTime);
}
```

Какой бы ни была текущая ориентация, метод *Update* вычисляет положение текста. Метод *Draw* аналогичен тем, что были представлены ранее.

Проект XNA: XnaOrientableHelloPhone Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, text, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Теперь телефон или эмулятор можно переворачивать в разных направлениях, и изображение на экране будет менять ориентацию соответственно.

Если требуется получить размер экрана телефона независимо от заднего буфера или ориентации (но с учетом панели задач), сделать это можно с помощью свойства *ClientBounds* (Границы клиентской области) класса *GameWindow*, обратиться к которому можно из свойства *Window* класса *Game*:

```
Rectangle clientBounds = this.Window.ClientBounds;
```

Простые часы (*очень* простые часы)

До сих пор в данной главе были рассмотрены два события Silverlight, *SizeChanged* и *OrientationChanged*, но использовались они по-разному. Событие *SizeChanged* я ассоциировал с обработчиком события в XAML, а для события *OrientationChanged* я переопределил эквивалентный метод *OnOrientationChanged*.

Конечно, обработчики этих событий могут быть определены полностью в коде. Одним очень удобным для приложений на Silverlight классом является *DispatcherTimer* (Таймер-диспетчер), который периодически обращается к приложению посредством события *Tick* (Тик) и побуждает его выполнить какую-то работу. Например, таймер используется для приложения, моделирующего часы.

Сетка для содержимого проекта SilverlightSimpleClock включает только расположенный по центру *TextBlock*:

Проект Silverlight: SilverlightSimpleClock Файл: MainPage.xaml (фрагмент)

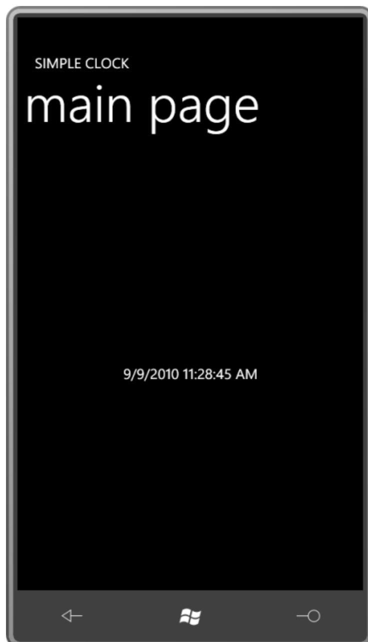
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Рассмотрим файл выделенного кода полностью. Обратите внимание на директиву *using* для пространства имен *System.Windows.Threading*, которое не используется по умолчанию. Это пространство имен, в котором находится *DispatcherTimer*:

Проект Silverlight: SilverlightSimpleClock Файл: MainPage.xaml.cs

```
using System;
using System.Windows.Threading;
using Microsoft.Phone.Controls;

namespace SilverlightSimpleClock
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();

            DispatcherTimer tmr = new DispatcherTimer();
            tmr.Interval = TimeSpan.FromSeconds(1);
            tmr.Tick += OnTimerTick;
            tmr.Start();
        }

        void OnTimerTick(object sender, EventArgs args)
        {
            txtblk.Text = DateTime.Now.ToString();
        }
    }
}
```

Конструктор инициализирует *DispatcherTimer*, указывая ему вызывать *OnTimerTick* (По тику таймера) раз в секунду. Обработчик этого события просто преобразует текущее время в строку, чтобы присвоить ее как значение *TextBlock*.



Несмотря на то, что *DispatcherTimer* определен в пространстве имен *System.Windows.Threading*, метод *OnTimerTick* вызывается в одном потоке со всем приложением. В противном случае, приложение не имело бы прямого доступа к *TextBlock*. Элементы Silverlight и связанные с ними объекты не являются потокобезопасными и будут

препятствовать доступу к ним из других потоков. Процедура доступа к элементам Silverlight из второстепенных потоков выполнения рассматривается в главе 5.

Часы – это еще одно приложение на Silverlight в данной главе, в котором свойство *Text* элемента *TextBlock* меняется динамически во время выполнения. Новое значение выводится, как по мановению волшебной палочки, без всякой дополнительной работы. Все это очень отличается от более ранних сред работы с графикой, в которых использовались Windows API или MFC. В них приложение выполняет отрисовку «по требованию», т.е. когда область окна становится недействительной и требует перерисовки, или когда приложение намеренно объявляет область недействительной, чтобы вызвать принудительную перерисовку.

Зачастую кажется, что приложение на Silverlight вообще ничего не отрисовывает! По сути своей Silverlight – это слой визуальной компоновки, который работает в заданном графическом режиме и организует все визуальные элементы в единую композицию. Элементы, такие как *TextBlock*, существуют как действительные сущности этого слоя композиции. В некоторый момент *TextBlock* формирует собственное визуальное представление (и выполняет повторную отрисовку при изменении одного из свойств, таких как *Text*), но все, что он отрисовывает, сохраняется вместе с визуализированным выводом всех остальных элементов дерева визуальных элементов.

Для сравнения, приложение на XNA выполняет отрисовку для каждого нового кадра экрана. Это концептуально отличается от более ранних сред разработки для Windows, так же как и от Silverlight. Это очень мощная возможность, но, я уверен, всем прекрасно известно, чем чревата такая мощь.

Иногда экран приложения XNA статичен, программе нет необходимости обновлять его с каждым кадром. Для сохранения энергии метод *Update* может вызывать метод *SuppressDraw* (Отменить отрисовку) класса *Game*, чтобы воспрепятствовать вызову соответствующего метода *Draw*. Метод *Update* по-прежнему будет вызываться 30 раз в секунду, потому что он должен выполнять проверку пользовательского ввода, но если код в *Update* вызывает *SuppressDraw*, *Draw* не будет выполняться в этом игровом цикле. Если *Update* не вызывает *SuppressDraw*, *Draw* выполняется.

Программе на XNA, моделирующей часы, не нужен таймер, потому что таймер уже встроен в обычный игровой цикл. Для создаваемых здесь часов мы не предполагаем отображения миллисекунд, т.е. экран должен обновляться лишь каждую секунду. Применим метод *SuppressDraw*, который будет предотвращать лишние вызовы *Draw*.

Рассмотрим поля XnaSimpleClock:

Проект XNA: XnaSimpleClock Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Viewport viewport;
    Vector2 textPosition;
    StringBuilder text = new StringBuilder();
    DateTime lastDateTime;
    …
}
```

Обратите внимание, что вместо того, чтобы определять поле *text* типа *string*, я задал *StringBuilder*. При создании в методе *Update* новых строк для отображения во время *Draw*

(как будет делать данное приложение), следует использовать *StringBuilder*. Это позволяет избежать распределений кучи, которые возникают в случае применения обычного типа *string*. Наше приложение будет лишь создавать новую строку каждую секунду, поэтому нам, на самом деле, нет особой необходимости применять здесь именно *StringBuilder*, но сделаем это в качестве тренировки. Чтобы работать со *StringBuilder*, необходимо добавить директиву для пространства имен *System.Text*.

Также обратите внимание на поле *lastDateTime* (Текущие дата и время). Оно используется в методе *Update* для определения необходимости обновления отображаемого времени.

Метод *LoadContent* принимает шрифт и окно просмотра экрана:

Проект XNA: XnaSimpleClock Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    viewport = this.GraphicsDevice.Viewport;
}
```

Логика сравнения значений *DateTime* (Дата и время) для определения, изменилось ли время с момента последнего вызова метода *Update*, несколько запутанная, потому что объекты *DateTime*, полученные в ходе двух последовательных вызовов *Update* будут разными *всегда*. Отличаться в них будут значения поля *Millisecond* (Миллисекунды). Поэтому новое значение *DateTime* вычисляется на основании текущего времени, полученного посредством *DateTime.Now*, но за вычетом миллисекунд:

Проект XNA: XnaSimpleClock Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Получаем DateTime без миллисекунд
    DateTime dateTime = DateTime.Now;
    dateTime = dateTime - new TimeSpan(0, 0, 0, 0, dateTime.Millisecond);

    if (dateTime != lastDateTime)
    {
        text.Remove(0, text.Length);
        text.Append(dateTime);
        Vector2 textSize = segoe14.MeasureString(text);
        textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                                   (viewport.Height - textSize.Y) / 2);
        lastDateTime = dateTime;
    }
    else
    {
        SuppressDraw();
    }

    base.Update(gameTime);
}
```

Здесь все просто. Если время изменилось, вычисляются новые значения *text*, *textSize* и *textPosition*. Поскольку *text* – это *StringBuilder*, а не *string*, старое содержимое удаляется и

сохраняется новое. В методе *MeasureString* класса *SpriteFont* есть перегрузка для *StringBuilder*, поэтому вызов выглядит точно так же.

Если время не изменилось, вызывается *SuppressDraw*. В результате *Draw* вызывается лишь раз в секунду.

DrawString также имеет перегрузку для *StringBuilder*:

Проект XNA: XnaSimpleClock Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, text, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

И вот результат:

Могут возникнуть некоторые сложности с использованием *SuppressDraw* во время первого запуска программы. Но применение этого метода является одной из основных методик в XNA для сокращения энергопотребления приложений.

Глава 3
Основы работы с сенсорным вводом

Даже для опытных разработчиков на Silverlight и XNA Windows Phone 7 предлагает возможность, которая, скорее всего, окажется новой и необычной. Экран телефона чувствителен к прикосновению и существенно отличается от старых сенсорных экранов, в основном повторяющих ввод с мыши, или экранов планшетных устройств, которые могут распознавать рукописный ввод.

Мультисенсорный экран устройства Windows Phone 7 может распознавать одновременное касание как минимум в четырех точках. Именно обработка взаимодействия этих одновременных касаний делает задачу реализации мультисенсорного ввода такой сложной для разработчиков. Но для данной главы я припас несколько менее амбициозную цель. Я просто хочу познакомить читателей с сенсорными интерфейсами в контексте примеров приложений, которые могут реагировать на простые касания.

Тестирование критически важного кода реализации мультисенсорного ввода необходимо выполнять на реальном устройстве, работающем под управлением Windows Phone 7. Между тем эмулятор телефона будет реагировать на действия мыши и преобразовывать их в сенсорный ввод. Чтобы использовать сенсорный ввод непосредственно на эмуляторе, его необходимо запустить под управлением Windows 7 на устройстве с поддерживающим мультисенсорный ввод экраном.

Приведенные в данной главе программы во многом схожи с простыми приложениями «Hello, Windows Phone 7!» из первой главы. Единственное отличие в том, что при касании текста пальцем, он будет случайным образом менять свой цвет, а при касании вне области текста, он будет опять возвращаться к исходному белому цвету (или любому другому цвету, какой будет применен к тексту при запуске программы).

В программе на Silverlight сенсорный ввод реализован через события. В приложении на XNA сенсорный ввод передается через статический класс, опрашиваемый в ходе выполнения метода *Update*. Одно из основных назначений XNA-метода *Update* – проверка состояния сенсорного вводи и внесение изменений, которые отображаются на экране во время выполнения метода *Draw*.

Обработка простого касания в XNA

В XNA устройства мультисенсорного ввода называют *сенсорной панелью*. Для обработки такого ввода используются методы статического класса *TouchPanel* (Сенсорная панель). Имеется также возможность обработки жестов, но пока начнем с более простыми данными касания.

Можно (но это не является обязательным) получать сведения о самом устройстве мультисенсорного ввода через вызов статического метода *TouchPanel.GetCapabilities*. Объект *TouchPanelCapabilities* (Возможности сенсорной панели), возвращаемый этим методом, имеет два свойства:

- *IsConnected* (Подключен) имеет значение *true*, если сенсорная панель доступна. Для телефона его значение всегда *true*.

- *MaximumTouchCount* (Максимальное число касаний) возвращает количество точек касания, как минимум 4 для телефона.

  Для большинства задач достаточно использовать один из двух статических методов *TouchPanel*. Для получения ввода от простого касания при каждом вызове *Update* после запуска программы, скорее всего, будет вызываться этот метод:

  ```
  TouchCollection touchLocations = TouchPanel.GetState();
  ```

  TouchCollection (Коллекция касаний) – это коллекция, включающая нуль или более объектов *TouchLocation* (Место касания). *TouchLocation* имеет три свойства:

- *State* (Состояние), его значениями являются элементы перечисления *TouchLocationState* (Состояние места касания): *Pressed* (Нажат), *Moved* (Перемещен), *Released* (Высвобожден).
- *Position* (Местоположение) – это *Vector2*, обозначающий положение пальца относительно верхнего левого угла окна просмотра.
- *Id* – целое число, идентифицирующее отдельное касание от состояния *Pressed* до *Released*, то есть в течение всего времени касания.

Если ни один палец не касается экрана, коллекция *TouchCollection* пуста. Когда палец впервые касается экрана, в *TouchCollection* появляется объект, свойство *State* которого имеет значение *Pressed*. Последующие вызовы *TouchPanel.GetState* покажут, что значение *State* объекта *TouchLocation* равно *Moved*, даже если фактически палец никуда не перемещался. Когда палец будет убран с экрана, свойство *State* объекта *TouchLocation* примет значение *Released*. Последующие вызовы *TouchPanel.GetState* продемонстрируют, что коллекция *TouchCollection* опять пуста.

Единственное исключение, если палец быстро «постукивает» по экрану – т.е. поднимается и опускается на экран с частотой примерно 1/30 секунды – свойство *State* объекта *TouchLocation* от значения *Pressed* сразу перейдет к значению *Released*, минуя состояния *Moved*.

Я описал касание всего одним пальцем. Как правило, экрана будут касаться множество пальцев; и опускаться, перемещаться и покидать экран они будут независимо друг от друга. Для отслеживания отдельного касания используется свойство *Id* (Идентификатор). Для одного отдельно взятого касания *Id* будет неизменным от состояния *Pressed*, на протяжении всех перемещений (значения *Moved*) и до состояния *Released*.

Части при работе с простым касанием используется объект *Dictionary* (Словарь) с ключами, созданными на основании свойства *Id*, для извлечения данных конкретного касания.

TouchLocation также имеет очень удобный метод *TryGetPreviousLocation* (Попытаться получить предыдущее местоположение), который вызывается следующим образом:

```
TouchLocation previousTouchLocation;
bool success = touchLocation.TryGetPreviousLocation(out previousTouchLocation);
```

Вызов этого метода практически всегда происходит, когда *touchLocation.State* имеет значение *Moved*, для получения предыдущего местоположения и вычисления разницы. Если значение *touchLocation.State* равно *Pressed*, *TryGetPreviousLocation* возвратит *false*, и значением *previousTouchLocation.State* будет элемент перечисления *TouchLocationState.Invalid*. Такие же результаты будут получены в случае вызова этого метода для *TouchLocation*, который был возвращен *TryGetPreviousLocation*.

Предлагаемое здесь мною приложение меняет цвет текста при касании пользователем экрана, т.е. обработка *TouchPanel.GetStates* будет относительно простой. Логика приложения

будет проверять только объекты *TouchLocation*, значение свойства *State* которых равно *Pressed*.

Назовем этот проект XnaTouchHello. Как и во всех рассматриваемых до этого проектах на XNA, нам понадобится шрифт, который я немного увеличил, чтобы обеспечить более удобную для касания мишень. Потребуется еще несколько дополнительных полей:

Проект XNA: XnaTouchHello Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Random rand = new Random();
    string text = "Hello, Windows Phone 7!";
    SpriteFont segoe36;
    Vector2 textSize;
    Vector2 textPosition;
    Color textColor = Color.White;
    …

}
```

Метод *LoadContent* аналогичен используемому ранее, за исключением того, что *textSize* сохраняется как поле. Это обеспечит возможности доступа к нему при последующих вычислениях:

Проект XNA: XnaTouchHello Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    segoe36 = this.Content.Load<SpriteFont>("Segoe36");
    textSize = segoe36.MeasureString(text);
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);
}
```

Как это свойственно приложениям на XNA, «действие» происходит преимущественно в методе *Update*. Этот метод вызывает *TouchPanel.GetStates* и затем поэлементно обходит коллекцию объектов *TouchLocation*, выбирая те из них, значение *State* которых равно *Pressed*.

Проект XNA: XnaTouchHello Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    TouchCollection touchLocations = TouchPanel.GetState();

    foreach (TouchLocation touchLocation in touchLocations)
    {
        if (touchLocation.State == TouchLocationState.Pressed)
        {
            Vector2 touchPosition = touchLocation.Position;
```

```
            if (touchPosition.X >= textPosition.X &&
                touchPosition.X < textPosition.X + textSize.X &&
                touchPosition.Y >= textPosition.Y &&
                touchPosition.Y < textPosition.Y + textSize.Y)
            {
                textColor = new Color((byte)rand.Next(256),
                                      (byte)rand.Next(256),
                                      (byte)rand.Next(256));
            }
            else
            {
                textColor = Color.White;
            }
        }
    }

    base.Update(gameTime);
}
```

Если *Position* оказывается где-нибудь внутри области, занимаемой текстовой строкой, полю *textColor* (Цвет текста) присваивается случайное значение RGB для цвета с помощью одного из конструкторов структуры *Color* (Цвет). В противном случае *textColor* присваивается значение *Color.White*.

Метод *Draw* практически аналогичен используемому в предыдущих примерах, только цвет текста теперь стал переменным:

Проект XNA: XnaTouchHello Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    this.GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe36, text, textPosition, textColor);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Единственная проблема в том, что касание не так строго детерминировано, как этого хотелось бы. Даже при касании одним пальцем контактов с экраном может быть несколько. В некоторых случаях один и тот же цикл *foreach* в методе *Update* может задавать *textColor* несколько раз!

Обработка жестов в XNA

Класс *TouchPanel* также включает возможности распознавания жестов, что демонстрирует проект XnaTapHello. В данном проекте используются те же поля, что и в XnaTouchHello, но несколько отличается метод *LoadContent*:

Проект XNA: XnaTapHello Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    segoe36 = this.Content.Load<SpriteFont>("Segoe36");
    textSize = segoe36.MeasureString(text);
```

```
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                               (viewport.Height - textSize.Y) / 2);

    TouchPanel.EnabledGestures = GestureType.Tap;
}
```

Обратите внимание на последнее выражение. *GestureType* (Тип жеста) – это перечисление, элементами которого являются *Tap* (Касание), *DoubleTap* (Двойное касание), *Flick* (Скольжение), *Hold* (Удержание), *Pinch*[1] (Сведение), *PinchComplete* (Сведение завершенно), *FreeDrag* (Произвольное перетягивание), *HorizontalDrag* (Перетягивание по горизонтали), *VerticalDrag* (Перетягивание по вертикали) и *DragComplete* (Перетягивание завершено). Эти элементы определены как битовые флаги, таким образом, они могут комбинироваться с помощью побитового C#-оператора OR.

Метод *Update* совсем другой.

Проект XNA: XnaTapHello Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gestureSample = TouchPanel.ReadGesture();

        if (gestureSample.GestureType == GestureType.Tap)
        {
            Vector2 touchPosition = gestureSample.Position;

            if (touchPosition.X >= textPosition.X &&
                touchPosition.X < textPosition.X + textSize.X &&
                touchPosition.Y >= textPosition.Y &&
                touchPosition.Y < textPosition.Y + textSize.Y)
            {
                textColor = new Color((byte)rand.Next(256),
                                      (byte)rand.Next(256),
                                      (byte)rand.Next(256));
            }
            else
            {
                textColor = Color.White;
            }
        }
    }

    base.Update(gameTime);
}
```

Несмотря на то, что данное приложение рассматривает только один тип жеста, код довольно универсален. Если жест доступен, метод *TouchPanel.ReadGesture* (Прочитать жест) возвращает его как объект типа *GestureSample* (Пример жеста). Кроме применяемых здесь *GestureType* и *Position*, у этого объекта имеется еще свойство *Delta* (Приращение), обеспечивающее данные о перемещении в виде объекта *Vector2*. Для некоторых жестов (таких как *Pinch*)

---

[1] Обычно используется для операции Zoom (Масштабирование), путем сведения или разведение пальцев. (*прим. научного редактора*).

GestureSample также предоставляет состояние второй точки касания через свойства *Position2* и *Delta2*.

Метод *Draw* аналогичен используемому в предыдущем случае, но поведение данного приложения будет несколько иным. В предыдущей программе текст меняет цвет, когда палец касается экрана; во втором изменение цвета происходит, когда палец убирается с экрана. Средству распознавания жестов необходимо дождаться завершения жеста, чтобы определить его тип.

События простого касания в Silverlight

Как и XNA, Silverlight поддерживает два разных программных интерфейса для работы с мультисенсорным вводом, которые можно категоризировать как интерфейс обработки простого и интерфейс обработки сложного касания. Интерфейс обработки простого касания построен на событии *Touch.FrameReported*, которое очень похоже на XNA-класс *TouchPanel*. Отличается оно лишь тем, что это событие, и оно не включает обработку жестов.

Интерфейс обработки сложного касания включает три события, определяемые классом *UIElement*: *ManipulationStarted* (Обработка началась), *ManipulationDelta* (Приращение в ходе обработки) и *ManipulationCompleted* (Обработка завершилась). События *Manipulation*, как их обобщенно называют, консолидируют взаимодействие множества касаний в движение и коэффициенты масштабирования.

Ядром интерфейса обработки простого касания в Silverlight является класс *TouchPoint* (Точка касания), экземпляр которого представляет отдельное касание экрана. *TouchPoint* имеет четыре свойства только для чтения:

- *Action* (Действие) типа *TouchAction* (Действие касания) – перечисление с элементами *Down* (Вниз), *Move* (Перемещение) и *Up* (Вверх).
- *Position* типа *Point* (Точка), значение которого определяется относительно верхнего левого угла конкретного элемента. Будем называть этот элемент *опорным*.
- *Size* типа *Size*. Это свойство должно представлять область касания (и, следовательно, давление, создаваемое пальцем, в некотором роде), но эмулятор Windows Phone 7 не возвращает полезных значений.
- *TouchDevice* типа *TouchDevice*.

Объект *TouchDevice* имеет два свойства только для чтения:

- *Id* типа *int*, используется для идентификации касаний. Каждое отдельное касание ассоциировано с уникальным *Id* на протяжении всех событий, начиная от *Down* и до *Up*.
- *DirectlyOver* (Непосредственно над) типа *UIElement* – самый верхний элемент, расположенный прямо под пальцем.

Как видите, Silverlight-объекты *TouchPoint* и *TouchDevice* предоставляют преимущественно те же сведения, что и XNA-объект *TouchLocation*. Свойство *DirectlyOver* объекта *TouchDevice* часто очень полезно для определения, какого элемента пользователь касается.

Для использования интерфейса обработки простого касания необходимо установить обработчик статического события *Touch.FrameReported*:

```
Touch.FrameReported += OnTouchFrameReported;
```

Метод *OnTouchFrameReported* выглядит следующим образом:

```
void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
{
    …
}
```

Этот обработчик события принимает все события касания в ходе выполнения приложения. Объект *TouchFrameEventArgs* (Аргументы события касания рамки) имеет свойство *TimeStamp* (Отметка времени) типа *int* и три метода:

- *GetTouchPoints(refElement)* (Получить точки касания) возвращает *TouchPointCollection* (Коллекция точек касания)
- *GetPrimaryTouchPoint(refElement)* (Получить основную точку касания) возвращает один *TouchPoint*
- *SuspendMousePromotionUntilTouchUp()* (Приостановить перемещения мыши до завершения касания)

В общем случае вызывается метод *GetTouchPoints* и в него передается опорный элемент. Значения свойств *Position* объектов *TouchPoint* в возвращенной коллекции определяются относительно этого элемента. Если передать *null* в *GetTouchPoints*, значения свойств *Position* будут установлены относительно верхнего левого угла окна просмотра приложения.

Между опорным элементом и элементом *DirectlyOver* нет никакой связи. Событие всегда обрабатывает все касания приложения в целом. Вызов *GetTouchPoints* или *GetPrimaryTouchPoints* для конкретного элемента *не* означает, что будут обрабатываться касания только этого элемента, это означает лишь то, что значение свойства *Position* будет определяться относительно этого элемента. (Поэтому координаты *Position* вполне могут быть отрицательными, если место касания находится слева или над опорным элементом.) Элемент *DirectlyOver* определяет элемент, находящийся непосредственно под пальцем.

Для разговора о втором и третьем методах необходимо сделать небольшое вступление. Событие *Touch.FrameReported* появилось в Silverlight для настольных приложений, в которых для логики обработки событий мыши существующих элементов управления удобно автоматически использовать сенсорный ввод. По этой причине события касания приравнены к событиям мыши.

Но это распространяется только на «первую» точку касания, т.е. на действия пальца, коснувшегося экрана первым, когда ни один другой палец его не касается. Если вы не хотите, чтобы действия этого касания трактовались как события мыши, обработчик события должен начинаться так:

```
void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
{
    TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);

    if (primaryTouchPoint != null && primaryTouchPoint.Action == TouchAction.Down)
    {
        args.SuspendMousePromotionUntilTouchUp();
    }
    …
}
```

Метод *SuspendMousePromotionUntilTouchUp* может вызываться только в момент первого касания первым пальцем, когда все остальные пальцы еще не коснулись экрана.

Для Windows Phone 7 такая логика представляет некоторые проблемы. Как сказано выше, по сути, происходит отключение обработки событий мыши для всего приложения. Если приложение для телефона включает элементы управления Silverlight, изначально

написанные для ввода с помощью мыши и не обновленные для приема сенсорного ввода, эти элементы управления фактически будут деактивированы.

Конечно, можно проверять свойство *DirectlyOver* и делать селективную приостановку обработки событий мыши. Но в телефоне не должно быть элементов, обрабатывающих ввод с помощью мыши, кроме тех которые не обрабатывают сенсорный ввод! Поэтому, вероятно, больше смысла будет в том, чтобы *никогда* не приостанавливать обработку событий мыши.

Я оставляю решение за вами и за устаревшими элементами управления, обрабатывающими ввод посредством мыши. Для приложения, которое я хочу написать, важна только первоначальная точка касания в момент, когда ее *TouchAction* имеет значение *Down*, поэтому я могу использовать ту же самую логику.

В проекте SilverlightTouchHello *TextBlock* описывается в XAML-файле:

Проект Silverlight: SilverlightTouchHello Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               Padding="0 34"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Обратите внимание на значение *Padding*. Я знаю, что свойство *FontSize* отображаемого здесь текста равно 20 пикселам, это обеспечивает *TextBlock* высотой около 27 пикселов. Также мне известно о рекомендации, что мишень касания не должна быть меньше 9 миллиметров. Если разрешение экрана телефона равно 264 DPI, 9 миллиметров – это 94 пиксел. (9 миллиметров разделить на 25,4 миллиметра/дюйм и умножить на 264 пикселов/дюйм.) *TextBlock* не хватает 67 пиксела. Поэтому я задаю значение *Padding*, которое добавляет по 34 пиксела сверху и снизу (но не по бокам).

Я применил здесь *Padding*, а не *Margin*, потому что *Padding* – это область *внутри TextBlock*. Таким образом, *TextBlock* фактически становится больше, чем предполагает размер текста. *Margin* – это область *вне TextBlock*. Она не является частью *TextBlock* и касания ее не учитываются как касания *TextBlock*.

Рассмотрим файл выделенного кода полностью. Конструктор *MainPage* определяет обработчик событий *Touch.FrameReported*.

Проект Silverlight: SilverlightTouchHello Файл: MainPage.xaml.cs

```
using System;
using System.Windows.Input;
using System.Windows.Media;
using Microsoft.Phone.Controls;

namespace SilverlightTouchHello
{
    public partial class MainPage : PhoneApplicationPage
    {
        Random rand = new Random();
        Brush originalBrush;

        public MainPage()
        {
            InitializeComponent();
            originalBrush = txtblk.Foreground;
```

```
            Touch.FrameReported += OnTouchFrameReported;
        }

        void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
        {
            TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);

            if (primaryTouchPoint != null && primaryTouchPoint.Action ==
TouchAction.Down)
            {
                if (primaryTouchPoint.TouchDevice.DirectlyOver == txtblk)
                {
                    txtblk.Foreground = new SolidColorBrush(
                            Color.FromArgb(255, (byte)rand.Next(256),
                                                (byte)rand.Next(256),
                                                (byte)rand.Next(256)));
                }
                else
                {
                    txtblk.Foreground = originalBrush;
                }
            }
        }
    }
}
```

Этот обработчик событий обрабатывает только первые точки касания, для которых *Action* имеет значение *Down*. Если свойство *DirectlyOver* возвращает элемент *txtblk*, создается случайный цвет. В отличие от XNA структура *Color* в Silverlight не имеет конструктора для задания цвета как комбинации значений красного, зеленого и синего, но в ней есть статический метод *FromArgb*, который создает объект *Color* на основании значений альфа, красного, зеленого и синего каналов, где альфа – это прозрачность. Для получения непрозрачного цвета, задайте альфа-каналу значение 255. Это очевидно не во всех файлах XAML, но свойство *Foreground* типа *Brush* (Кисть) – это абстрактный класс, от которого наследуется *SolidColorBrush* (Одноцветная кисть).

Если *DirectlyOver* не *txtblk*, цвет текста не меняется на белый. В случае если бы пользователь выбрал цветовую тему с черным текстом на белом фоне, это привело бы к тому, что текст исчез бы с экрана. Вместо этого свойству *Foreground* присваивается изначально заданный для *TextBlock* цвет. Он определяется в конструкторе.

События Manipulation

Интерфейс обработки сложного касания в Silverlight включает три события: *ManipulationStarted*, *ManipulationDelta* и *ManipulationCompleted*. Это события не занимаются отдельными касаниями, они консолидируют действия множества касаний в операции преобразования и масштабирования. Также они аккумулируют сведения о скорости, поэтому могут использоваться для реализации инерции, несмотря на то что не поддерживают ее напрямую.

События *Manipulation* будут рассмотрены более подробно далее в данной книге. В этой главе я применю *ManipulationStarted* просто для выявления контакта пальца с экраном, но не буду заниматься обработкой последующих действий.

Тогда как *Touch.FrameReported* обеспечивал данные касания для всего приложения, события *Manipulation* делают это для отдельных элементов. Таким образом, в SilverlightTapHello1 обработчик события *ManipulationStarted* может быть закреплен за *TextBlock*:

Проект Silverlight: SilverlightTapHello1 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!"
              Padding="0 34"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

Файл MainPage.xaml.cs включает такой обработчик события:

Проект Silverlight: SilverlightTapHello1 Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();

    public MainPage()
    {
        InitializeComponent();
    }
    void OnTextBlockManipulationStarted(object sender,
                                    ManipulationStartedEventArgs args)
    {
        TextBlock txtblk = sender as TextBlock;

        Color clr = Color.FromArgb(255, (byte)rand.Next(256),
                                        (byte)rand.Next(256),
                                        (byte)rand.Next(256));

        txtblk.Foreground = new SolidColorBrush(clr);

       args.Complete();
    }
}
```

Обработчик события получает элемент, формирующий это событие, из аргумента *sender*. Им всегда будет *TextBlock*. Сведения о TextBlock также доступны из свойства *args.OriginalSource* и свойства *args.ManipulationContainer*.

Обратите внимание на вызов метода *Complete* (Завершить) аргументов события в конце. Он не является обязательным, но эффективен для уведомления системы о том, что в дальнейшем события *Manipulation* для обработки этого касания не нужны.

Данная программа некорректна. Если выполнить ее, можно заметить, что она работает только частично. Прикосновение к *TextBlock* меняет цвет текста случайным образом. Но если коснуться вне *TextBlock*, цвет *не* возвращается к исходному белому. Поскольку это событие было задано для *TextBlock*, обработчик события вызывается, только когда пользователь касается *TextBlock*. Программа не обрабатывает никаких других событий *Manipulation*.

Приложение, удовлетворяющее моим исходным техническим условиям, должно обрабатывать *все* события касания страницы. Обработчик события *ManipulationStarted* должен быть определен для *MainPage*, а не только для *TextBlock*.

Хотя, безусловно, этот вариант возможен, но существует более простой способ. Класс *UIElement* определяет все события *Manipulation*. Но класс *Control* (от которого наследуется *MainPage*) дополняет эти события защищенными виртуальными методами. Можно не задавать обработчик события *ManipulationStarted* в *MainPage*, а просто перегрузить виртуальный метод *OnManipulationStarted* (Когда обработка началась).

Такой подход реализован в проекте SilverlightTapHello2. Файл XAML не обрабатывает никакие события, но задает имя для *TextBlock*, которое может использоваться в коде:

Проект Silverlight: SilverlightTapHello2 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
              Padding="0 34"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Класс *MainPage* перегружает метод *OnManipulationStarted*:

Проект Silverlight: SilverlightTapHello2 Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    Brush originalBrush;

    public MainPage()
    {
        InitializeComponent();
        originalBrush = txtblk.Foreground;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        if (args.OriginalSource == txtblk)
        {
            txtblk.Foreground = new SolidColorBrush(
                        Color.FromArgb(255, (byte)rand.Next(256),
                                            (byte)rand.Next(256),
                                            (byte)rand.Next(256)));
        }
        else
        {
            txtblk.Foreground = originalBrush;
        }

       args.Complete();
        base.OnManipulationStarted(args);
    }
}
```

В *ManipulationStartedEventArgs* (Аргументы события обработка началась) свойство *OriginalSource* (Первоначальный источник) указывает на источник события, т.е. на расположенный поверх всех остальных элемент, которого касается пользователь. Если значение этого свойства равно *txtblk*, метод создает случайный цвет для свойства *Foreground*. Если нет, свойство *Foreground* возвращается к исходному цвету.

В этом методе *OnManiulationStarted* обрабатываются события *MainPage*, но свойство *OriginalSource* показывает, что фактически событие произошло в элементе, расположенном ниже в визуальном дереве. Это так называемая возможность *обработки маршрутизированных событий* в Silverlight.

Маршрутизированные события

В разработке для Microsoft Windows ввод посредством клавиатуры и мыши всегда поступает в конкретные элементы управления. Ввод с клавиатуры всегда направляется в элемент управления, имеющий фокус ввода. Ввод с мыши всегда направляется в активный элемент управления, расположенный непосредственно под указателем мыши, поверх всех остальных элементов. Так же обрабатывается ввод со стилуса и касание. Но иногда такая логика неудобна. В некоторых случаях нижележащий элемент управления нуждается в пользовательском вводе больше, чем элемент, расположенный непосредственно под указателем устройства ввода.

Для большей гибкости Silverlight реализует систему *обработки маршрутизированных событий*. Большинство событий пользовательского ввода, включая три события *Manipulation*, формируются соответственно парадигме, применяемой в Windows: они формируются расположенным поверх остальных активным элементом, к которому прикасается пользователь. Но если этот элемент не обрабатывает данное событие, оно передается в его родительский элемент, и так далее вверх по визуальному дереву, заканчивая элементом *PhoneApplicationFrame*. Любой элемент в этой цепочке может захватывать этот ввод и каким-то образом его обрабатывать, и также останавливать его дальнейшее распространение вверх по дереву.

Вот почему можно перегрузить метод *OnManipulationStarted* в *MainPage* и принимать события *Manipulation* для *TextBlock*. По умолчанию *TextBlock* не обрабатывает эти события.

Аргумент события *ManipulationStarted* – *ManipulationStartedEventArgs*. Он наследуется от *RoutedEventArgs* (Аргументы маршрутизированного события). Именно *RoutedEventArgs* определяет свойство *OriginalSource*, обозначающее элемент, который сформировал событие.

Но это предлагает другой подход, сочетающий в себе две методики, представленные в SilverlightTapHello1 и SilverlightTapHello2. Рассмотрим XAML-файл проекта SilverlightTapHello3:

Проект Silverlight: SilverlightTapHello3 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               Padding="0 34"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

Имя (*Name*) *TextBlock* аналогично используемому в первой программе. В ней обработчик события *ManipulationStarted* задан для *TextBlock*. И обработчик события, и перегруженный *OnManipulationStarted* находятся в файле выделенного кода:

Проект Silverlight: SilverlightTapHello3 Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    Brush originalBrush;

    public MainPage()
    {
        InitializeComponent();
        originalBrush = txtblk.Foreground;
```

```
    }

    void OnTextBlockManipulationStarted(object sender,
                                        ManipulationStartedEventArgs args)
    {
        txtblk.Foreground = new SolidColorBrush(
                    Color.FromArgb(255, (byte)rand.Next(256),
                                        (byte)rand.Next(256),
                                        (byte)rand.Next(256)));
        args.Complete();
        args.Handled = true;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        txtblk.Foreground = originalBrush;

        args.Complete();
        base.OnManipulationStarted(args);
    }
}
```

Логика разнесена на два метода, что делает всю обработку намного более элегантной, на мой взгляд. Метод *OnTextBlockManipulationStarted* принимает события, только когда происходит касание *TextBlock*. Событие *OnManipulationStarted* отвечает за все события *MainPage*.

На первый взгляд может показаться, что здесь ошибка. После вызова *OnTextBlockManipulationStarted* событие продолжает свое «путешествие» вверх по визуальному дереву, и *OnManipulationStarted* возвращает цвет к исходному белому. Но на самом деле происходит не это. Правильность выполнения всей логики обеспечивает выражение в конце обработчика *OnTextBlockManipulationStarted* для *TextBlock*:

```
args.Handled = true;
```

Это выражение говорит о том, что событие уже обработано и *не* должно передаваться далее вверх по визуальному дереву. Удалите это выражение – и *TextBlock* никогда не изменит своего исходного цвета, по крайней мере, не на так долго, чтобы это можно было заметить.

Странное поведение?

Теперь попробуем следующее. Во многих рассмотренных до сих пор приложениях на Silverlight центрирование *TextBlock* в рамках сетки для содержимого обеспечивалось следующими двумя атрибутами:

```
HorizontalAlignment="Center"
VerticalAlignment="Center"
```

Удалите их из SilverlightTapHello3, выполните повторную компиляцию и запустите приложение. Текст будет выведен в верхнем левом углу *Grid*. Но теперь, если коснуться *любой точки* в области под *TextBlock*, текст будет менять свой цвет случайным образом. Возвращение к исходному белому цвету обеспечивается только касанием области заголовка над текстом.

По умолчанию свойствам *HorizontalAlignment* и *VerticalAlignment* присваивается значение *Stretch*. Фактически *TextBlock* заполняет весь *Grid*. Вы не можете видеть этого, но пальцы не лгут.

Как будет показано далее, для других элементов – отображающих растровые изображения, например – этот эффект растяжения намного более очевиден.

Глава 4
Растровые изображения или текстуры

Еще одним объектом, который применяется в приложениях на Silverlight и XNA так же широко, как и текст, является *растровое изображение*. Ранее растровое изображение определялось как двумерный массив битов, соответствующих пикселам графического устройства.

В Silverlight растровое изображение иногда называют просто *изображением*, но это, главным образом, наследие Windows Presentation Foundation, где *изображениями* называют и растровые изображения, и векторные рисунки. И в WPF, и в Silverlight элемент *Image* обеспечивает вывод на экран растровых изображений, но сам не является растровым изображением.

В XNA растровое изображение типа *Texture2D* и, следовательно, часто называется *текстурой*, но этот термин преимущественно относится к 3D-графике, где растровые изображения используются для заполнения поверхностей 3D-объектов. В 2D-графике на XNA растровые изображения часто используются как спрайты.

Растровые изображения также используются в качестве значков приложений в телефоне. При создании нового проекта на XNA или Silverlight в Visual Studio для различных целей создается три растровых изображения.

Форматом растровых изображений в Windows является BMP, но в последние годы он утратил свою былую популярность, поскольку большее распространение получили форматы со сжатием. В настоящее время самыми широко используемыми форматами являются:

- JPEG (Joint Photography Experts Group)
- PNG (Portable Network Graphics)
- GIF (Graphics Interchange File)

XNA поддерживает все три (и многие другие). Silverlight поддерживает только JPEG и PNG. (Разработчики на Silverlight не всегда помнят об этом и часто удивляются, почему это приложение на Silverlight отказывается выводить на экран GIF или BMP.)

Применяемые форматами PNG и GIF алгоритмы сжатия не приводят к потере данных. Исходное растровое изображение может быть восстановлено со 100% точностью. Поэтому эти алгоритмы часто называют алгоритмами сжатия «без потерь».

JPEG реализует алгоритм «с потерями», потому что отбрасывает визуальные данные, которые не воспринимаются человеческим глазом. Такой тип сжатия хорош для реалистичных изображений, таких как фотографии, но менее подходит для растровых изображений, производных от текста или векторных рисунков, таких как архитектурные чертежи или мультфильмы.

И Silverlight, и XNA обеспечивает возможность интерактивной или алгоритмической обработки растровых изображений на уровне пикселов для формирования или изменения имеющихся растровых изображений. Эта тема будет рассмотрена в главе 14 (для Silverlight) и главе 21 (для XNA). Здесь же мы больше сосредоточимся на методах получения растровых

изображений из различных источников, включая само приложение, Веб, встроенную камеру телефона и библиотеку фотографий телефона.

Создание текстуры на XNA

Разработка 2D-графики в XNA – практически исключительно процесс перемещения спрайтов по экрану. Поэтому можно ожидать, что загрузка и отрисовка растровых изображений в приложении на XNA не составляет никакого труда, и это действительно так.

Назовем первый проект XnaLocalBitmap, потому что это растровое изображение будет храниться как часть содержимого приложения. Чтобы добавить новое растровое изображение в проект содержимого приложения, щелкните правой кнопкой мыши имя проекта XnaLocalBitmapContent, выберите Add, затем New Item и Bitmap File (Файл растрового изображения). Растровое изображение можно создать прямо в Visual Studio.

Или оно может быть создано во внешней программе, как я сделал в данном случае. В качестве такой внешней программы удобно использовать Windows Paint. Для данного примера я создал следующее растровое изображение 320 пиксела шириной и 160 пикселов высотой:

Я сохранил его в файле Hello.png.

Чтобы добавить этот файл и сделать его содержимым приложения, щелкните правой кнопкой мыши проект XnaLocalBitmapContent в Visual Studio, выберите Add и Existing Item (Существующий элемент). После этого перейдите к файлу. Когда файл появится в проекте, его можно щелкнуть правой кнопкой мыши и просмотреть Properties. Вы увидите, что его свойство Asset Name имеет значение «Hello».

Наша задача – вывести растровое изображение посередине экрана. В файле Game1.cs описываем поле для хранения *Texture2D* и еще одно поле для координат:

Проект XNA: XnaLocalBitmap Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D helloTexture;
    Vector2 position;
    …
}
```

Оба поля задаются во время выполнения метода *LoadContent*. Для загрузки *Texture2D* используем тот же универсальный метод, который применялся для загрузки *SpriteFont*. Класс

Texture2D имеет свойства *Width* и *Height*, определяющие размеры растрового изображения в пикселах. Как и в приложении с выводом текста посередине экрана из главы 1, поле *position* обозначает координаты верхнего левого угла растрового изображения на экране в пикселах:

Проект XNA: XnaLocalBitmap Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    helloTexture = this.Content.Load<Texture2D>("Hello");
    Viewport viewport = this.GraphicsDevice.Viewport;
    position = new Vector2((viewport.Width - helloTexture.Width) / 2,
                           (viewport.Height - helloTexture.Height) / 2);
}
```

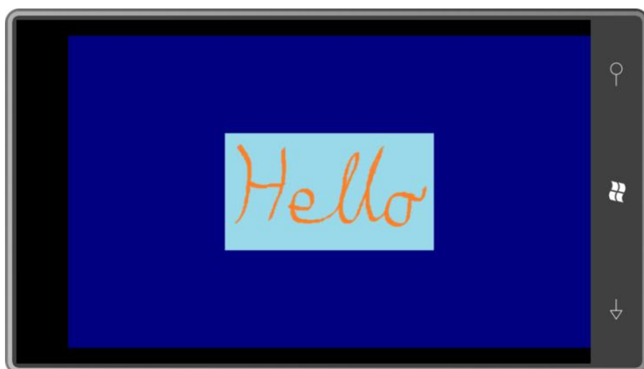
Класс *SpriteBatch* имеет семь методов *Draw* для формирования визуального представления растровых изображений. Этот, безусловно, самый простой:

Проект XNA: XnaLocalBitmap Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(helloTexture, position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Последний аргумент *Draw* – это цвет, который может использоваться для смягчения существующих цветов растрового изображения. Используйте *Color.White*, если хотите, чтобы цвета растрового изображения отображались без искажения.

И вот, что мы получили:



Элемент *Image* в Silverlight

Аналогичное приложение на Silverlight даже еще проще. Создадим проект SilverlightLocalBitmap. Сначала создадим в проекте папку для хранения растрового изображения. Это не обязательно, но делает проект более аккуратным. Обычно разработчики называют эту папку Images (Рисунки), Media (Мультимедиа) или Assets (Ресурсы) в зависимости от типов файлов, для которых она предназначается. Щелкните

правой кнопкой мыши имя проекта, выберите Add и затем New Folder (Новая папка). Назовем ее Images. Затем щелкаем правой кнопкой мыши имя папки и выбираем Add и Existing Item. Переходим к файлу Hello.png. (Если вы самостоятельно создали другой растровый рисунок, не забывайте, что Silverlight поддерживает только файлы в формате JPEG и PNG.)

Из кнопки Add выберите Add или Add as Link (Добавить как ссылку). Если выбрана опция Add, файл будет физически скопирован в подпапку проекта. При выборе Add as Link в проекте будет сохранена только ссылка на файл, но файл изображения все равно будет скопирован в исполняемый файл.

Заключительный шаг – щелкаем правой кнопкой мыши имя файла растрового изображения и открываем Properties. Убеждаемся, что Build Action (Действие при сборке) имеет значение Resource (Ресурс). Можно изменить значение Build Action на Content, но давайте пока оставим его неизменным, и остановимся на различиях несколько позже.

В Silverlight элемент *Image* используется для отображения растровых изображений точно так же, как элемент *TextBlock* используется для отображения текста. Задайте в качестве значения свойства *Source* (Источник) элемента *Image* путь к файлу растрового изображения в проекте:

Проект Silverlight: SilverlightLocalBitmap Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Source="Images/Hello.png" />
</Grid>
```

Изображение несколько отличается от обеспечиваемого приложением на XNA, и дело не только в заголовках. По умолчанию элемент *Image* расширяет или растягивает растровое изображение максимально, как только возможно, для заполнения контейнера (сетки для содержимого), сохраняя при этом его пропорции. Особенно это заметно, если задать атрибуту *SupportedOrientations* начального тега *PhoneApplicationPage* значение *PortraitOrLandscape* и повернуть телефон на бок:

Если требуется вывести растровое изображение в его натуральную величину, задайте свойству *Stretch* элемента Image значение *None*:

```
<Image Source="Images/Hello.png"
       Stretch="None" />
```

Больше вариантов будет рассмотрено в главе 8.

Изображения из Интернета

Одна из самых замечательных возможностей, предоставляемых элементом *Image* – возможность задания URL в качестве значения свойства *Source*, как в данном проекте на Silverlight:

Проект Silverlight: SilverlightWebBitmap Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Source="http://www.charlespetzold.com/Media/HelloWP7.jpg" />
</Grid>
```

И вот что получается:

Несомненно, это довольно просто, и извлечение изображений из Интернета, вместо того чтобы сохранять их в приложении, конечно же, намного сокращает размер исполняемого файла. Но нет никакой гарантии, что приложение, выполняющееся под управлением Windows Phone 7, будет иметь подключение к Интернету, не говоря уже о других проблемах, связанных с загрузкой файлов. Элемент *Image* имеет два события, *ImageOpened* (Изображение открыто) и *ImageFailed* (Ошибка загрузки изображения), которые могут использоваться для определения успешности или сбоя загрузки.

В приложениях для Windows Phone 7, использующих большое количество растровых изображений, необходимо все тщательно продумать. Растровые изображения могут встраиваться в исполняемый файл и быть гарантированно доступными, или можно сэкономить память телефона и загружать изображения только по мере необходимости.

В XNA загрузить растровое изображение из Интернета не так просто, но .NET-класс *WebClient* (Веб-клиент) делает все относительно безболезненно. С этим классом несколько проще работать, чем с общепринятым альтернативным вариантом (*HttpWebRequest* (Веб-запрос HTTP) и *HttpWebResponse* (Веб-ответ HTTP)), и часто он является предпочтительным выбором для загрузки единичных элементов.

WebClient может использоваться для загрузки либо строк (как правило, XML-файлов), либо бинарных объектов. Фактическая передача выполняется асинхронно. Когда файл передан,

WebClient вызывает метод, чтобы обозначить успешное завершение загрузки или ее сбой. Этот вызов метода происходит в потоке логики выполнения приложения, так что вы пользуетесь преимуществами асинхронной передачи данных без явной обработки вторичных потоков.

Чтобы использовать *WebClient* в приложении на XNA, понадобиться добавить ссылку на библиотеку System.Net. Для этого в Solution Explorer под именем проекта щелкните правой кнопкой мыши References (Ссылки) и выберите Add Reference (Добавить ссылку). В таблице .NET выберите System.Net. (приложения на Silverlight получают ссылку на System.Net автоматически.)

Файл Game1.cs проекта XnaWebBitmap также требует директивы *using* для пространства имен *System.Net*. Данное приложение описывает те же поля, что и предыдущая программа:

Проект XNA: XnaWebBitmap   Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D helloTexture;
    Vector2 position;
    …
}
```

Метод *LoadContent* создает экземпляр *WebClient*, задает метод обратного вызова и затем инициирует передачу:

Проект XNA: XnaWebBitmap   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    WebClient webClient = new WebClient();
    webClient.OpenReadCompleted += OnWebClientOpenReadCompleted;
    webClient.OpenReadAsync(new
Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg"));
}
```

Метод *OnWebClientOpenReadCompleted* (При завершении операции открытия-чтения Веб-клиентом) вызывается, когда файла полностью загружен. Возникнет желание проверить, не была ли отменена загрузка, и не возникло ли ошибки. Если все в порядке, свойство *Result* (Результат) аргументов события будет типа *Stream* (Поток). Этот *Stream* можно использовать со статическим методом *Texture2D.FromStream* (Из потока) для создания объекта *Texture2D*:

Проект XNA: XnaWebBitmap   Файл: Game1.cs (фрагмент)

```
void OnWebClientOpenReadCompleted(object sender, OpenReadCompletedEventArgs args)
{
    if (!args.Cancelled && args.Error == null)
    {
        helloTexture = Texture2D.FromStream(this.GraphicsDevice, args.Result);
        Viewport viewport = this.GraphicsDevice.Viewport;
        position = new Vector2((viewport.Width - helloTexture.Width) / 2,
                               (viewport.Height - helloTexture.Height) / 2);
    }
}
```

Метод *Texture2D.FromStream* поддерживает форматы JPEG, PNG и GIF.

По умолчанию значение свойства *AllowReadStreamBuffering* (Разрешить буферизацию потока чтения) объекта *WebClient* равно *true*. Это означает, что событие *OpenReadCompleted* (Открытие-чтение завершено) будет формироваться, только когда весь файл будет полностью загружен. Объект *Stream*, доступный в свойстве *Result*, фактически является потоком в памяти, однако, это экземпляр внутреннего класса библиотек .NET, а не самого *MemoryStream* (Поток в памяти).

Если задать *AllowReadStreamBuffering* значение *false*, свойство *Result* будет сетевым потоком. Класс *Texture2D* не позволит выполнять чтение из этого потока в основном потоке выполнения программы.

Как правило, метод *LoadContent* производного от *Game* класса вызывается до первого вызова метода *Update* или *Draw*, но важно помнить, что между вызовами *LoadContent* и *OnWebClientOpenReadCompleted* пройдет масса времени. В течение этого времени выполняется асинхронное чтение, но класс *Game1* продолжает выполняться, как обычно, вызывая *Update* и *Draw*. Поэтому попытка доступа к объекту *Texture2D* должна выполняться, только когда точно известно, что он действителен:

Проект XNA: XnaWebBitmap Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    if (helloTexture != null)
    {
        spriteBatch.Begin();
        spriteBatch.Draw(helloTexture, position, Color.White);
        spriteBatch.End();
    }

    base.Draw(gameTime);
}
```

В реальном приложении желательно также предусмотреть некоторое уведомление пользователя в случае, если растровое изображение не может быть загружено.

Image и ImageSource

Несомненно, *WebClient* может использоваться в приложении на *Silverlight*, но при работе с растровыми изображениями в этом, в общем, нет необходимости, потому что классы, предусмотренные для работы с растровыми изображениями, уже реализуют асинхронную загрузку.

Однако более тщательное рассмотрение элемента *Image* приводит в замешательство. Элемент *Image* не является растровым изображением, он просто обеспечивает отображение растрового изображения. Во всех примерах ранее в качестве значения свойства *Source* объекта *Image* задавался путь к файлу или URL:

```
<Image Source="Images/Hello.png" />
<Image Source="http://www.charlespetzold.com/Media/HelloWP7.jpg" />
```

Можно предположить, что это свойство *Source* типа *string*. Извините, но вовсе нет! На самом деле здесь представлен синтаксис XAML, скрывающий довольно многое. Свойство *Source* фактически типа *ImageSource* (Источник изображения). Это абстрактный класс, от которого

наследуется *BitmapSource* (Источник растрового изображения). А это еще один абстрактный класс, описывающий метод *SetSource* (Задать источник), который позволяет загружать растровое изображение из объекта *Stream*.

От класса *BitmapSource* наследуется *BitmapImage* (Растровое изображение), который поддерживает конструктор, принимающий объект *Uri* и также включающий свойство *UriSource* (Источник URI) типа *Uri*. Проект SilverlightTapToDownload1 имитирует приложение, загружающее растровое изображение, URL которого известен только во время выполнения. XAML включает элемент *Image* без растрового изображения:

Проект Silverlight: SilverlightTapToDownload1 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img" />
</Grid>
```

BitmapImage требует подключения пространства имен *System.Windows.Media.Imaging* посредством директивы *using*. Когда *MainPage* регистрирует касание, он создает *BitmapImage* из объекта *Uri* и задает его в качестве значения свойства *Source* элемента *Image*:

Проект Silverlight: SilverlightTapToDownload1 Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    Uri uri = new Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg");
    BitmapImage bmp = new BitmapImage(uri);
    img.Source = bmp;

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Не забудьте коснуться экрана, чтобы начать загрузку!

Класс *BitmapImage* описывает события *ImageOpened* и *ImageFailed* (которые дублируются элементом *Image*) и также включает событие *DownloadProgess* (Процесс загрузки).

Если есть желание, *WebClient* можно явно использовать в приложении на Silverlight, как показано в следующем проекте. Файл SilverlightTapToDownload2.xaml аналогичен SilverlightTapToDownload1.xaml. Файл выделенного кода использует *WebClient* практически так же, как это было в предыдущем приложении на XNA:

Проект Silverlight: SilverlightTapToDownload2 Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    WebClient webClient = new WebClient();
    webClient.OpenReadCompleted += OnWebClientOpenReadCompleted;
    webClient.OpenReadAsync(new
Uri("http://www.charlespetzold.com/Media/HelloWP7.jpg"));

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}

void OnWebClientOpenReadCompleted(object sender, OpenReadCompletedEventArgs args)
```

```
{
    if (!args.Cancelled && args.Error == null)
    {
        BitmapImage bmp = new BitmapImage();
        bmp.SetSource(args.Result);
        img.Source = bmp;
    }
}
```

Обратите внимание на использование *SetSource* для создания растрового изображения из объекта *Stream*.

Загрузка хранящихся локально растровых изображений из кода

В приложении на Silverlight растровое изображение, добавляемое в проект как ресурс, встраивается в исполняемый файл. Обращаться к локальному растровому изображению напрямую из XAML настолько просто, что лишь очень немногие опытные разработчики на Silverlight смогут без подготовки рассказать, как можно сделать это в коде. А мы рассмотрим для этого проект SilverlightTapToLoad.

Как и остальные приложения на Silverlight в данной главе, проект SilverlightTapToLoad включает сетку для содержимого с элементом *Image*. Растровое изображение Hello.png хранится в папке Images, и его свойство Build Action имеет значение Resource.

Для использования класса *BitmapImage* в файле MainPage.xaml.cs должна присутствовать директива *using* для пространства имен *System.Windows.Media.Imaging*. Еще одна директива *using* для *System.Windows.Resources* требуется для работы с классом *StreamResourceInfo* (Данные потокового ресурса).

Когда происходит касание экрана, обработчик события выполняет доступ к ресурсу с помощью статического метода *GetResourceStream*, описанного классом *Application*:

Проект Silverlight: SilverlightTapToLoad   Файл: MainPage.xaml.cs

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    Uri uri = new Uri("/SilverlightTapToLoad;component/Images/Hello.png",
UriKind.Relative);
    StreamResourceInfo resourceInfo = Application.GetResourceStream(uri);
    BitmapImage bmp = new BitmapImage();
    bmp.SetSource(resourceInfo.Stream);
    img.Source = bmp;

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Посмотрите, насколько сложен этот URL! Он начинается с названия приложения, потом следует точка с запятой, потом слово «component», и за ним уже имя папки и файла. Если для файла Hello.png изменить значение Build Action с Resource на Content, синтаксис существенно упроститься:

```
Uri uri = new Uri("Images/Hello.png", UriKind.Relative);
```

В чем разница?

В подпапке *Bin/Debug* проекта Visual Studio можно увидеть файл нашего приложения SilverlightTapToLoad.xap. Если изменить расширение этого файла на ZIP, мы получим возможность заглянуть внутрь. Основным содержимым этого файла будет SilverlightTapToLoad.dll, скомпилированный двоичный файл.

В обоих случаях очевидно, что растровое изображение хранится где-то в файле XAP. Разница в том, что:

- Когда Build Action растрового изображения задано значение Resource, оно хранится в файле SilverlightTapToLoad.dll вместе со скомпилированным приложением.
- Когда Build Action растрового изображения задано значение Content, оно хранится вне файла SilverlightTapToLoad.dll, но в файле XAP. Если переименовать XAP-файл в ZIP-файл, в нем можно увидеть папку *Images* и файл.

Какой из вариантов лучше?

В документе Майкрософт, озаглавленном «Creating High Performance Silverlight Applications for Windows Phone»[1], для Build Action ресурсов приложения рекомендуется задавать значение Content, а не Resource, что позволит сократить размер двоичного файла и время загрузки приложения при запуске. Однако если эти ресурсы располагаются в библиотеке Silverlight, используемой приложением, лучше включить их в двоичный файл, задав для Build Action значение Resource.

Если программа использует большое количество изображений, и нет желания помещать их все в XAP-файл, но есть опасения по поводу загрузки из Интернета, почему бы не скомбинировать оба способа? Включите в XAP-файл изображения с низким разрешением (или высокой степенью сжатия), но версии этих же изображений более высокого качества загружайте асинхронно в процессе выполнения приложения.

Захват изображения с камеры

Кроме встраивания растровых изображений в приложение или их загрузки через Веб, Windows Phone 7 позволяет также использовать изображения со встроенной камеры.

Приложение не может управлять самой камерой. Из соображений безопасности нельзя допускать, чтобы программа просто произвольно делала снимок или могла «увидеть», что находится в области видимости камеры. Поэтому приложение, по сути, вызывает стандартную утилиту работы с камерой, пользователь выбирает вид и делает снимок, и изображение передается в приложение.

Используемые для этого классы находятся в пространстве имен *Microsoft.Phone.Tasks*. В этом пространстве имен имеется несколько классов-задач, называемых Задачи выбора (*choosers*) и Задачи выполнения (*launchers*). Концептуально, эти классы очень похожи, но задачи выбора возвращают данные в приложение, а задачи выполнения нет.

Класс *CameraCaptureTask* (Задача по захвату изображения с камеры) наследуется от универсального класса *ChooserBase* (Базовая задача выбора), который определяет событие *Completed* (Завершен) и метод *Show* (Показать). Приложение описывает обработчик события *Completed* и делает вызов *Show*. Когда вызывается обработчик события *Completed*, аргумент события *PhotoResult* (Полученная фотография) содержит объект *Stream* для фотографии. С этого момента мы уже знаем, что делать.

---

[1] Создание высокопроизводительных приложений для Windows Phone на Silverlight (*прим. переводчика*).

Как и предыдущие приложения данной главы, SilverlightTapToShoot имеет файл MainPage.xaml, включающий сетку для содержимого с элементом *Image*. Рассмотрим файл выделенного кода полностью:

Проект Silverlight: SilverlightTapToShoot Файл: MainPage.xaml.cs

```
using System.Windows.Input;
using System.Windows.Media.Imaging;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Tasks;

namespace SilverlightTapToShoot
{
    public partial class MainPage : PhoneApplicationPage
    {
        CameraCaptureTask camera = new CameraCaptureTask();;

        public MainPage()
        {
            InitializeComponent();

            camera.Completed += OnCameraCaptureTaskCompleted;
        }

        protected override void OnManipulationStarted(ManipulationStartedEventArgs
args)
        {
            camera.Show();

            args.Complete();
            args.Handled = true;
            base.OnManipulationStarted(args);
        }

        void OnCameraCaptureTaskCompleted(object sender, PhotoResult args)
        {
            if (args.TaskResult == TaskResult.OK)
            {
                BitmapImage bmp = new BitmapImage();
                bmp.SetSource(args.ChosenPhoto);
                img.Source = bmp;
            }
        }
    }
}
```

Это приложение можно выполнить на эмуляторе телефона. Если коснуться экрана эмулятора, вызов *Show* инициирует задачу по работе с камерой, и выполняется переход на страницу, которая напоминает реальную камеру. «Сделать снимок» можно, коснувшись значка в верхнем правом углу экрана. (Условное «фото» – это просто большой белый квадрат с небольшим черным квадратом возле одной из кромок.) После этого необходимо щелкнуть кнопу Accept (Принять).

Также это приложение, безусловно, может быть выполнено на самом телефоне; при этом телефон не должен быть подключен к ПК с выполняющимся настольным ПО Zune. С помощью Visual Studio разверните приложение на телефон и, прежде чем тестировать его, закройте ПО Zune.

Чтобы выполнить в Visual Studio отладку использующего камеру приложения, которое развернуто на телефоне, можно применить небольшую программу, выполняющуюся в режиме командной строки, под названием WPDTPTConnect32.exe или WPDTPTConnect64.exe (в зависимости от того, ведется ли разработка в 32- или 64-разрядной системе). Эти

программы являются альтернативой ПО Zune, и благодаря им отладчик Visual Studio может управлять приложением, хотя оно и выполняется на телефоне. Прежде чем использовать эти программы, закройте ПО Zune.

В любом случае, по нажатию кнопки Accept камера закрывается и выполняется метод *OnCameraCaptureTaskCompleted* (После завершения задачи по захвату изображения с камеры). Он создает объект *BitmapImage*, берет значение входящего потока из *args.ChoosenPhoto* и присваивает объект *BitmapImage* элементу *Image*, который будет выводить фотографию на экран.

Процесс в целом кажется довольно простым. Концептуально программа вызывает процесс камеры и затем по его завершении возобновляет выполнение.

Но документация по Windows Phone 7, с которой я ознакомился, предупреждает, что все не совсем так. Происходит еще кое-что, что не так очевидно на первый взгляд и что может лишить присутствия духа.

После того как вызван метод *Show* объекта *CameraCaptureTask*, приложение SilverlightTapToShoot завершается. (Но не сразу. Метод *OnManipulationStarted* может вернуть значения в приложение, и формируется еще пара событий, но после этого приложение абсолютно точно завершается.)

Выполняется утилита работы с камерой. Когда она завершается, происходит повторный запуск SilverlightTapToShoot, создается новый экземпляр приложения. Выполнение начинается с самого начала, вызывается конструктор *MainPage*, который связывает событие *Completed* класса *CameraCaptureTask* с методом *OnCameraCaptureTaskCompleted*, после чего вызывается этот метод.

По этим причинам документация рекомендует при использовании задачи выбора или выполнения, такой как *CameraCaptureTask*, определять объект как поле и включать обработчик события *Completed* в конструктор приложения. Кроме того, обработчик должен располагаться в конструкторе максимально близко к концу, поскольку он будет вызываться при повторном запуске программы.

Такое завершение и повторное выполнение приложения свойственно разработке для Windows Phone 7 и получило название *захоронение* (*tombstoning*). Когда приложение завершается, потому что начинается обработка задач работы с камерой, операционная система телефона сохраняет значительный объем сведений, чтобы запустить приложение снова после завершения работы с камерой. Тем не менее, удерживаемых данных не достаточно для полного восстановления приложения в состояние, в котором оно было до «захоронения». За это отвечает разработчик.

Использование задач запуска или выбора – это лишь один из примеров, когда захоронение имеет место. Также оно происходит, когда пользователь покидает приложение, нажимая кнопку Start телефона. Через некоторое время пользователь может вернуться в приложение, нажав кнопку Back, и программе придется повторно запуститься из состояния захоронения. Также захоронение выполняется, когда отсутствие деятельности на телефоне приводит к его блокировке.

Если приложение выполняется, и пользователь нажимает кнопку Back, происходит *не* захоронение, а обычное завершение приложения.

При захоронении, очевидно, требуется сохранять данные состояния приложения, чтобы его можно было восстановить при повторном запуске. Несомненно, Windows Phone 7 предлагает специальные возможности для этого, что будет рассмотрено в главе 6.

Несмотря на все вышесказанное, в последних версиях операционной системы Windows Phone 7, включая ту, которую я использую при работе над данной книгой, применение *CameraCaptureTask* не обеспечивает захоронения. Но никогда не помешает подготовиться заранее.

Библиотека фотографий телефона

Фотографии, сделанные при помощи телефона, и фотографии, загружаемые в телефон при синхронизации с ПК, формируют библиотеку фотографий телефона. Любое приложение, выполняющееся на телефоне, может работать с этой библиотекой двумя способами:

- Для приложения класс *PhotoChooserTask* (Задача по выбору фотографии) во многом аналогичен *CameraCaptureTask*, только он обеспечивает доступ к библиотеке фотографий и предоставляет пользователю возможность выбрать одну фотографию, которая затем возвращается в приложение.
- В пространстве имен XNA *Microsoft.Xna.Framework.Media* имеется класс *MediaLibrary* (Библиотека мультимедиа) и связанные классы, с помощью которых приложение может получать коллекции всех фотографий, хранящихся в библиотеке фотографий, и представлять их пользователю.

Рассмотрим эти два подхода на примере двух приложений. Просто для разнообразия (и чтобы продемонстрировать использование XNA-классов в приложении на Silverlight) первый подход реализуем на XNA и второй – на Silverlight.

Обе эти программы можно выполнять на эмуляторе телефона. Эмулятор включает небольшую коллекцию фотографий, которая предусмотрена специально для целей тестирования. Однако в случае тестирования приложений на телефоне он должен быть отключен от ПК, или ПО Zune должно быть закрыто, поскольку оно не допустит одновременного доступа еще одного приложения к библиотеке фотографий телефона. Закрыв Zune, запускайте WPDTPTConnect32.exe или WPDTPTConnect64.exe, которое позволяет Visual Studio проводить отладку приложения, выполняющегося на телефоне.

В приложении XnaTapToBrowse необходимо подключить пространство имен *Microsoft.Phone.Tasks* посредством директивы *using*. Это позволит создать объект *PhotoChooserTask* и другие поля:

Проект Silverlight: XnaTapToBrowse Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D texture;
    PhotoChooserTask photoChooser = new PhotoChooserTask();
    …
}
```

Соответственно рекомендациям документации этот класс прикрепляет событие *Completed* к конструктору:

Проект Silverlight: XnaTapToBrowse Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
```

```
    Content.RootDirectory = "Content";

    // По умолчанию для Windows Phone частота кадров составляет 30 кадр/с.
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    TouchPanel.EnabledGestures = GestureType.Tap;
    photoChooser.Completed += OnPhotoChooserCompleted;
}
```

Как обычно, метод *Update* проводит проверку на наличие пользовательского ввода. В случае касания он инициирует событие *Show* объекта *PhotoChooserTask*:

Проект Silverlight: XnaTapToBrowse Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
        if (TouchPanel.ReadGesture().GestureType == GestureType.Tap)
            photoChooser.Show();

    base.Update(gameTime);
}

void OnPhotoChooserCompleted(object sender, PhotoResult args)
{
    if (args.TaskResult == TaskResult.OK)
        texture = Texture2D.FromStream(this.GraphicsDevice, args.ChosenPhoto);
}
```

После этого обработчик события *Completed* из потока, доступного из свойства *ChosenPhoto* (Выбранная фотография), создает *Texture2D*. Пока этот объект недоступен, перегруженный метод *Draw* не делает попыток сформировать его визуальное представление:

Проект Silverlight: XnaTapToBrowse Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    if (texture != null)
    {
        spriteBatch.Begin();
        spriteBatch.Draw(texture, this.GraphicsDevice.Viewport.Bounds, Color.White);
        spriteBatch.End();
    }

    base.Draw(gameTime);
}
```

В данном случае метод *Draw* в *SpriteBatch* немного изменен. Во втором аргументе вместо координат *Texture2D* возвращается прямоугольник, размер которого равен размеру окна просмотра. Это обусловливает изменение размеров фотографии, что весьма вероятно приведет к ее искажению, поскольку пропорции исходного изображения не учитываются. Безусловно, это решаемые проблемы, но они требуют более сложного кода.

В приложение SilverlightAccessLibrary необходимо включить ссылку на библиотеку Microsoft.Xna.Framework DLL. При этом, скорее всего, появится предупреждение о включении библиотеки XNA в приложение на Silverlight. Ничего страшного! Область содержимого файла MainPage.xaml включает *Grid* с *Image* и *TextBlock*:

Проект Silverlight: SilverlightAccessLibrary Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img" />

    <TextBlock Name="txtblk"
               TextWrapping="Wrap"
               TextAlignment="Center"
               VerticalAlignment="Bottom" />
</Grid>
```

Данное приложение не будет представлять пользователю всю библиотеку фотографий (реализовать эту задачу с помощью лишь рассмотренных нами до сих пор неполнофункциональных элементов компоновки Silverlight довольно сложно). В данном случае одна фотография будет выбираться случайным образом, и другая – по касанию пользователем экрана:

Проект Silverlight: SilverlightAccessLibrary Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    MediaLibrary mediaLib = new MediaLibrary();
    Random rand = new Random();

    public MainPage()
    {
        InitializeComponent();
        GetRandomPicture();
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        GetRandomPicture();

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void GetRandomPicture()
    {
        PictureCollection pictures = mediaLib.Pictures;

        if (pictures.Count > 0)
        {
            int index = rand.Next(pictures.Count);
            Picture pic = pictures[index];

            BitmapImage bmp = new BitmapImage();
            bmp.SetSource(pic.GetImage());
            img.Source = bmp;

            txtblk.Text = String.Format("{0}\n{1}\n{2}",
                                        pic.Name, pic.Album.Name, pic.Date);
        }
    }
}
```

Экземпляр XNA-класса *MediaLibrary* создан как поле. Метод *GetRandomPicture* (Возвратить случайный рисунок) получает объект *PictureCollection* (Коллекция рисунков) из класса *MediaLibrary* и случайным образом выбирает одно изображение. Метод *GetImage* (Возвратить изображение) объекта *Picture* (Рисунок) возвращает поток и данные *Name*, *Album* (Альбом) и *Data* (Сведения), отображаемые приложением в *TextBlock*.

Приложения для Windows Phone 7 может также сохранять растровое изображение в библиотеку. Все сохраняемые растровые изображения размещаются в специальном альбоме под названием Saved Pictures (Сохраненные рисунки). Как это реализовывается, будет рассмотрено в главах 14 и 22.

Глава 5
Датчики и службы

Данная глава посвящена двум возможностям сбора сведений об окружающем мире в Windows Phone 7. С согласия пользователя служба определения местоположения предоставляет приложению данные о местоположении телефона в традиционных географических координатах, долгота и широта, тогда как акселерометр сообщает приложению направление вниз.

Ни акселерометр, ни служба определения местоположения не будут корректно работать в открытом космосе. Это их роднит.

Акселерометр и служба определения местоположения, как будто, довольно просты, поэтому в данной главе также рассматриваются вопросы, касающиеся вторичных потоков выполнения, которые обрабатывают асинхронные операции и выполняют доступ к Веб-сервисам.

Акселерометр

Устройства Windows Phone имеют акселерометр – небольшое аппаратное устройство, по сути, измеряющее силу, которая, согласно элементарной физике, пропорциональна ускорению. Когда телефон неподвижен, акселерометр регистрирует силу притяжения, т.е. позволяет определить положение телефона относительно земли.

Моделирование уровня нивелира является основным применением акселерометра. Кроме того, акселерометр может также обеспечивать исходные данные для интерактивной анимации. Например, можно смоделировать управление курьерским мотоциклом по улицам мегаполиса, поворачивая телефон влево или вправо, как будто рулем.

Акселерометр также реагирует на неожиданные перемещения, такие как встряхивание или толчки, что позволяет моделировать игру в кости или некоторые другие виды действий с элементом случайности. Изобретение различных применений акселерометра является одной из множества задач разработки приложений для телефонов.

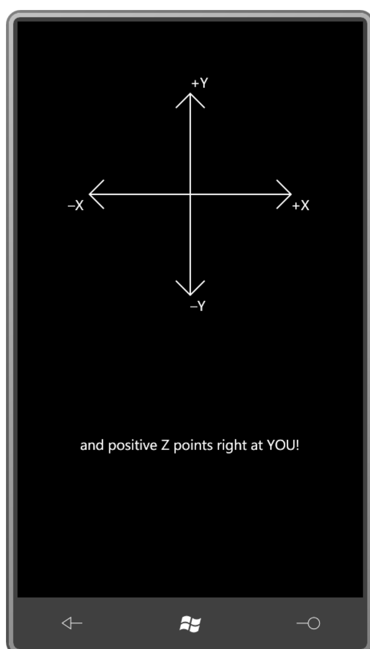
Выходные данные акселерометра удобно представить в виде трехмерного вектора. Обычно векторы записываются полужирным шрифтом, поэтому вектор ускорения может быть обозначен так: ***(x, y, z)***. В XNA есть тип трехмерный вектор, в Silverlight нет.

Тогда как три координаты (*x, y, z*) точно определяют точку в пространстве, вектор ***(x, y, z)*** обозначает направление и величину. Очевидно, что точка и вектор взаимосвязаны. Направление вектора ***(x, y, z)*** – это луч из точки (0, 0, 0) в точку (*x, y, z*). Но вектор ***(x, y, z)*** это совсем не отрезок, соединяющий (0, 0, 0) и (*x, y, z*). Это направление этого отрезка.

Для вычисления модуля вектора ***(x, y, z)*** используется трехмерная форма теоремы Пифагора:

$$\text{Модуль} = \sqrt{x^2 + y^2 + z^2}$$

Для работы с акселерометром ассоциируем с телефоном трехмерную систему координат. Не важно, как ориентирован телефон, но положительное направление оси Y будет проходить вдоль максимального размера телефона снизу (где располагаются кнопки) вверх, и положительное направление оси X – слева направо. Ось Z направлена перпендикулярно телефону, прямо на пользователя.



Это традиционная трехмерная система координат, такая же используется при создании 3D-графики на XNA. Ее называют *правой* системой координат. Расположите указательный палец правой руки вдоль положительного направления X, средний палец – вдоль положительного направления Y, и большой палец будет указывать на положительное направление Z.

Эта координатная система фиксирована относительно телефона независимо от его положения в пространстве и независимо от ориентации изображения выполняемых приложений. Кстати, как можно догадаться, акселерометр является основным компонентом, обеспечивающим возможность изменения ориентации изображения приложений Windows Phone 7.

Когда телефон неподвижен, вектор акселерометра указывает на землю. Модуль равен 1, т.е. 1*g*[1], что соответствует силе притяжения на поверхности земли. Когда телефон расположен вертикально, вектор ускорения – **(0, –1, 0)**, т.е. направлен прямо вниз.

Поверните телефон на 90° против часовой стрелки (так называемое левостороннее альбомное расположение), и вектор ускорения станет равным **(–1, 0, 0)**; переверните телефон «вверх ногами» – вектор равен **(0, 1, 0)**; еще один поворот на 90° против часовой стрелки обеспечивает правостороннее альбомное расположение и вектор ускорения равный **(1, 0, 0)**. Положите телефон на стол экраном вверх, и вектор ускорения будет **(0, 0, –1)**. (Эмулятор Windows Phone 7 всегда возвращает это значение.)

Конечно, вектор ускорения *редко* будет принимать такие точные значения, даже его модуль не будет точно равен 1. Для неподвижного телефона значения модуля могут колебаться в пределах нескольких процентов для разных ориентаций. На Луне величина вектора ускорения Windows Phone 7 будет в районе 0,17, но мобильная связь неустойчивая.

Я описал значения вектора ускорения для неподвижного устройства. Вектор ускорения может менять направление (и модуль может быть больше или меньше), когда телефон перемещается с разной скоростью. Например, если тряхнуть телефон влево, вектор ускорения будет указывать вправо, но только пока устройство набирает скорость. Как только скорость стабилизируется, вектор ускорения опять будет отражать только силу притяжения.

---

[1] g - ускорение свободного падения, ~9,8 м/с^2 (*прим. научного редактора*).

Когда телефон, движущийся влево, начнет замедляться, вектор ускорения ненадолго направится влево до остановки устройства.

При свободном падении модуль вектора ускорения, теоретически, должен стремиться к нулю.

Для работы с акселерометром используется библиотека Microsoft.Devices.Sensors и пространство имен *Microsoft.Devices.Sensors*. В файле WMAppManifest.xml необходимо указать:

```
<Capability Name="ID_CAP_SENSORS" />
```

Это задается по умолчанию.

В приложении создается экземпляр класса *Accelerometer* (Акселерометр), задается обработчик события *ReadingChanging* (Изменение показаний прибора) и вызывается метод *Start*.

И тут начинаются фокусы. Рассмотрим проект SilverlightAccelerometer.project, который просто обеспечивает выведение на экран текущих показаний. Центрированный *TextBlock* описывается в файле XAML:

Проект Silverlight: SilverlightAccelerometer Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Это приложение будет обеспечивать вывод на экран вектора акселерометра на протяжении всего времени его выполнения, поэтому создание класса *Accelerometer* и вызов метода *Start* происходит в конструкторе:

Проект Silverlight: SilverlightAccelerometer Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();

    Accelerometer acc = new Accelerometer();
    acc.ReadingChanged += OnAccelerometerReadingChanged;

    try
    {
        acc.Start();
    }
    catch (Exception exc)
    {
        txtblk.Text = exc.Message;
    }
}
```

Документация предупреждает, что вызов *Start* может привести к формированию исключения, поэтому приложение защищает себя от этого. Класс *Accelerometer* также поддерживает методы *Stop* (Остановить) и *Dispose* (Удалить), но в данной программе они не используются. Для обеспечения сведения о доступности акселерометра и его состоянии предоставляется свойство *State*.

Событие *ReadingChanged* обеспечивается классом аргументов события *AccelerometerReadingEventArgs*. Этот объект имеет свойства *X*, *Y* и *Z* типа *double* и свойство *TimeStamp* типа *DateTimeOffset* (Смещение даты-времени). В приложении SilverlightAccelerometer задача обработчика события – форматировать эти данные в строку и задать ее как значение свойства *Text* объекта *TextBlock*.

Загвоздка здесь в том, что обработчик события (в данном случае, *OnAccelerometerReadingChanged* (При изменении показаний акселерометра)) вызывается в другом потоке выполнения. Это означает, что он должен обрабатываться особым образом.

Немного теории. Создание и доступ ко всем элементам и объектам пользовательского интерфейса в приложении на Silverlight выполняется в основном потоке выполнения, который часто называют *потоком пользовательского интерфейса* или *UI-потоком*. Эти объекты пользовательского интерфейса не являются потокобезопасными; для них не предусмотрен одновременный доступ из множества потоков. Поэтому Silverlight не допустит доступа к объекту пользовательского интерфейса из потока, не являющегося UI-потоком.

Это означает, что метод *OnAccelerometerReadingChanged* не может напрямую обратиться к элементу *TextBlock*, чтобы задать новое значение его свойству *Text*.

К счастью в пространстве имен *System.Windows.Threading* описан класс *Dispatcher* (Диспетчер), который обеспечивает решение этой проблемы. Через класс *Dispatcher* можно направлять задания из потока, не являющегося UI-потоком, в очередь, которая позднее будет обработана UI-потоком. Все это звучит довольно запутанно, но с точки зрения разработчика все довольно просто, потому что эти задания принимают форму простых вызовов методов.

Экземпляр *Dispatcher* уже доступен. Класс *DependencyObject* (Зависимость) описывает свойство *Dispatcher* типа *Dispatcher*, и от *DependencyObject* наследуется множество классов Silverlight. С экземплярами всех этих классов можно работать из потоков, не являющихся UI-потоками, потому что все они имеют свойства *Dispatcher*. Из любого производного от *DependencyObject* класса, созданного в UI-потоке, может использоваться любой объект *Dispatcher*. Они все одинаковые.

Класс *Dispatcher* определяет метод *CheckAccess* (Проверить возможность доступа), который возвращает *true*, если текущий поток имеет доступ к конкретному объекту пользовательского интерфейса. (Метод *CheckAccess* также дублируется самим *DependencyObject*.) Для случаев, когда доступ к объекту из текущего потока невозможен, *Dispatcher* предоставляет две версии метода *Invoke* (Вызвать), с помощью которых выполняется отправка заданий для UI-потока.

В проекте SilverlightAccelerometer реализована версия кода, доскональная с точки зрения синтаксиса, но несколько позже я покажу, как его сократить.

В развернутой версии требуется делегат и метод, соответствующий этому делегату. Делегат (и метод) не должен иметь возвращаемого значения, но у него должно быть такое количество аргументов, которого будет достаточно для выполнения задания. В данном случае это задание состоит в присвоении строки в качестве значения свойства *Text* объекта *TextBlock*:

Проект: SilverlightAccelerometer Файл: MainPage.xaml.cs (фрагмент)

```
delegate void SetTextBlockTextDelegate(TextBlock txtblk, string text);

void SetTextBlockText(TextBlock txtblk, string text)
{
    txtblk.Text = text;
}
```

OnAccelerometerReadingChanged отвечает за вызов *SetTextBlockText* (Задать текст блока текста). Он впервые использует *CheckAccess*, чтобы определить возможность вызова метода *SetTextBlockText* напрямую. Если это невозможно, обработчик вызывает метод *BeginInvoke* (Начать вызов). Его первым аргументом является экземпляр делегата с методом *SetTextBlockText*. За ним следуют все необходимые *SetTextBlockText* аргументы:

Проект: SilverlightAccelerometer Файл: MainPage.xaml.cs (фрагмент)

```
void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    string str = String.Format("X = {0:F2}\n" +
                               "Y = {1:F2}\n" +
                               "Z = {2:F2}\n\n" +
                               "Magnitude = {3:F2}\n\n" +
                               "{4}",
                               args.X, args.Y, args.Z,
                               Math.Sqrt(args.X * args.X + args.Y * args.Y +
                                                   args.Z * args.Z),
                               args.Timestamp);

    if (txtblk.CheckAccess())
    {
        SetTextBlockText(txtblk, str);
    }
    else
    {
        txtblk.Dispatcher.BeginInvoke(new
SetTextBlockTextDelegate(SetTextBlockText),
                                    txtblk, str);
    }
}
```

Это не самый плохой вариант, но необходимость перепрыгивать из потока в поток повлекла за собой применение дополнительного метода и делегата. А можно ли все сделать прямо в обработчике событий?

Да! У метода *BeginInvoke* есть перегруженный вариант, принимающий делегат *Action*, который определяет метод, не имеющий возвращаемого значения и аргументов. Можно создать анонимный метод прямо в вызове *BeginInvoke*. Полный код, следующий за созданием строкового объекта, выглядит следующим образом:

```
if (txtblk.CheckAccess())
{
    txtblk.Text = str;
}
else
{
    txtblk.Dispatcher.BeginInvoke(delegate()
    {
        txtblk.Text = str;
    });
}
```

Анонимный метод начинается с ключевого слова *delegate* (делегат), за которым следует тело метода, заключенное в фигурные скобки. Пустые круглые скобки после ключевого слова *delegate* не являются обязательными.

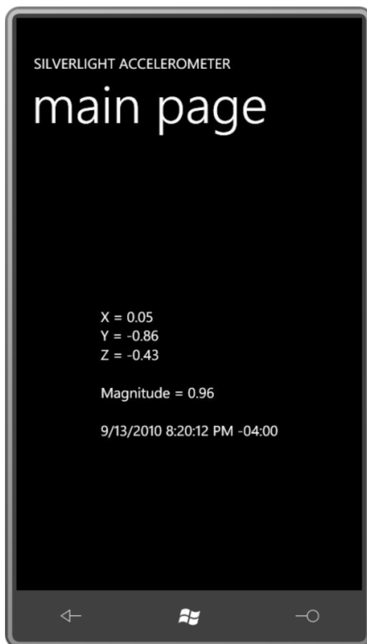
Анонимный метод также может быть описан с помощью лямбда-выражения:

```
if (txtblk.CheckAccess())
{
    txtblk.Text = str;
```

```
}
else
{
    txtblk.Dispatcher.BeginInvoke(() =>
    {
        txtblk.Text = str;
    });
}
```

Дублирующийся код, задающий *str* свойству *Text* объекта *TextBlock*, выглядит некрасиво (и нежелателен, если бы выражений было больше одного), но вызов *CheckAccess*, на самом деле, не нужен. Можно просто вызвать *BeginInvoke*, и ничего плохого не случится, если он вызывается из UI-потока.

В эмуляторе Windows Phone 7 нет акселерометра, поэтому он всегда возвращает значение (0, 0, –1), свидетельствующее о том, что телефон лежит на плоской поверхности. Выполнять это приложение имеет смысл только на реальном телефоне:

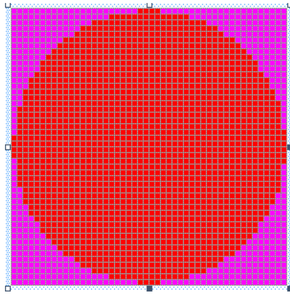


Полученные здесь значения свидетельствуют о том, что телефон располагается вертикально, но несколько наклонен назад, что является обычным положением при использовании телефона.

Простой уровень нивелира

В любой мастерской можно без труда найти нивелир или, как его еще называют, ватерпас или уровень. Небольшой пузырек всегда плавает на поверхности жидкости и позволяет визуально увидеть, как поверхность расположена относительно земли: параллельно, вертикально или под наклоном.

Проект XnaAccelerometer включает растровое изображение Bubble.bmp размером 48 × 48 пикселов, представляющее собой красный круг:



Благодаря пурпурному цвету, заполняющему углы изображения, при формировании визуального представления эти области становятся прозрачными.

Как и в приложении на Silverlight, в данном случае понадобиться подключить библиотеку Microsoft.Devices.Sensors и пространство имен *Microsoft.Devices.Sensors*.

Поля класса *Game1*, главным образом, предназначаются для переменных, обеспечивающих позиционирование растрового изображения на экране:

Проект XNA: XnaAccelerometer Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float BUBBLE_RADIUS_MAX = 25;
    const float BUBBLE_RADIUS_MIN = 12;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Vector2 screenCenter;
    float screenRadius;     // меньше, чем BUBBLE_RADIUS_MAX

    Texture2D bubbleTexture;
    Vector2 bubbleCenter;
    Vector2 bubblePosition;
    float bubbleScale;

    Vector3 accelerometerVector;
    object accelerometerVectorLock = new object();
     …

}
```

В конце этого фрагмента можно увидеть поле *accelerometerVector* (Вектор ускорения) типа *Vector3*. Обработчик события *OnAccelerometerReadingChanged* будет сохранять в это поле новое значение, и метод *Update* будет использовать это значение при вычислении местоположения растрового изображения.

Методы *OnAccelerometerReadingChanged* и *Update* выполняются в разных потоках: один – в задающем поле, другой – в читающем. Нет никакой проблемы в том, если поле задается или читается в одной команде на машинном языке. Это было бы так, если бы *Vector3* был классом, являющимся ссылочным типом, используемым преимущественно посредством указателей. Но *Vector3* – это структура (тип значения), состоящая из трех свойств типа *float*, каждое из которых занимает четыре байта, что составляет в сумме 12 байт или 96 бит. Задание или чтение этого поля *Vector3* связано с передачей такого объема данных.

Устройство, работающее под управлением Windows Phone 7, имеет как минимум 32-разрядный процессор ARM. Если взглянуть на набор команд ARM, вы не найдете среди них такую, которая обеспечивала бы передачу 12 байтов сразу. Это означает, что поток

акселерометра, сохраняющий новое значение *Vector3*, может быть прерван методом *Update* основного потока программы, извлекающим это значение. В *X*, *Y* и *Z* могут быть записаны значения из разных операций чтения.

Несмотря на то, что это не так уж смертельно для этого приложения, давайте будем делать все абсолютно безопасным и надежным. Для этого воспользуемся C#-выражением *lock* (блокировать), чтобы значение *Vector3* гарантированно записывалось и читалось двумя потоками без прерывания. В этом задача переменной *accelerometerVectorLock* (Блокировка вектора акселерометра).

Я решил создать объект *Accelerometer* и задать обработчик события в методе *Initialize*:

Проект XNA: XnaAccelerometer Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try
    {
        accelerometer.Start();
    }
    catch
    {
    }

    base.Initialize();
}

void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accelerometerVectorLock)
    {
        accelerometerVector = new Vector3((float)args.X, (float)args.Y,
(float)args.Z);
    }
}
```

Обратите внимание, что обработчик события использует выражение *lock* для задания поля *accelerometerVector*. Это предотвращает доступ к данному полю из метода *Update* в течение этого короткого промежутка времени.

Метод *LoadContent* загружает растровое изображение, используемое для моделирования нивелира, и инициализирует несколько переменных, используемых для его позиционирования:

Проект XNA: XnaAccelerometer Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Viewport viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);
    screenRadius = Math.Min(screenCenter.X, screenCenter.Y) - BUBBLE_RADIUS_MAX;

    bubbleTexture = this.Content.Load<Texture2D>("Bubble");
    bubbleCenter = new Vector2(bubbleTexture.Width / 2, bubbleTexture.Height / 2);
}
```

Если свойства *X* и *Y* акселерометра равны нулю, «пузырек» отображается в центре экрана. Вот зачем нужны *screenCenter* (Центр экрана) и *bubbleCenter* (Центр «пузырька»). Значение *screenRadius* (радиус экрана) – это расстояние от центра, когда модуль компонентов *X* и *Y* равен 1.

Метод *Update* выполняет защищенный доступ к полю *accelerometerVector* (Вектор акселерометра) и вычисляет *bubblePosition* (Положение «пузырька») на основании значений компонентов *X* и *Y*. Может показаться, что я перепутал компоненты *X* и *Y* при вычислении, но это лишь потому, что в XNA по умолчанию используется альбомный режим, т.е. координаты обратны координатам вектора ускорения. По умолчанию поддерживаются оба альбомных режима (левосторонний и правосторонний), поэтому вектор ускорения важно умножать на -1, когда телефон располагается в режиме *LandscapeRight* (Правостороннее альбомное расположение):

Проект XNA: XnaAccelerometer Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    Vector3 accVector;

    lock (accelerometerVectorLock)
    {
        accVector = accelerometerVector;
    }

    int sign = this.Window.CurrentOrientation ==
                                    DisplayOrientation.LandscapeLeft ? 1 : -1;

    bubblePosition = new Vector2(screenCenter.X + sign * screenRadius * accVector.Y,
                                 screenCenter.Y + sign * screenRadius *
accVector.X);
    float bubbleRadius = BUBBLE_RADIUS_MIN + (1 - accVector.Z) / 2 *
                            (BUBBLE_RADIUS_MAX - BUBBLE_RADIUS_MIN);
    bubbleScale = bubbleRadius / (bubbleTexture.Width / 2);

    base.Update(gameTime);
}
```

Кроме того, на основании компонента *Z* вектора вычисляется коэффициент *bubbleScale* (Масштаб «пузырька»). Идея в том, что «пузырек» увеличивается, когда телефон располагается экраном вверх, и уменьшается, если экран наклоняется вниз, как будто экран является одной из сторон настоящего прямоугольного резервуара с жидкостью, и размер «пузырька» отражает его удаление от поверхности.

Перегруженный *Draw* использует длинную версию метода *Draw* класса *SpriteBatch*.

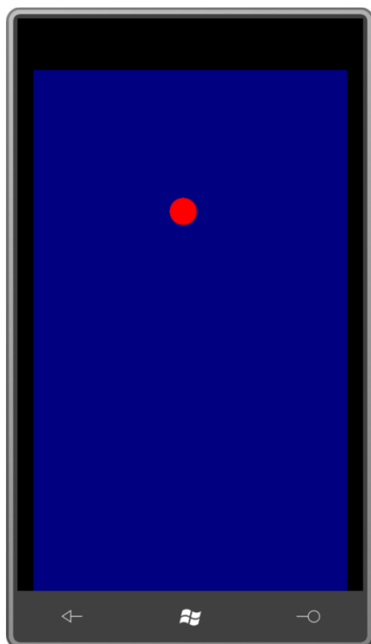
Проект XNA: XnaAccelerometer Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(bubbleTexture, bubblePosition, null, Color.White, 0,
                     bubbleCenter, bubbleScale, SpriteEffects.None, 0);
    spriteBatch.End();
```

```
    base.Draw(gameTime);
}
```

Обратите внимание на аргумент *bubbleScale*, который меняет размеры растрового изображения. Центр масштабирования задается предыдущим аргументом метода, *bubbleCenter*. Эта точка также совпадает со значением *bubblePosition* относительно экрана.

Такое приложение выглядит не особо впечатляюще, а выполнять его на эмуляторе вообще скучно. Данное изображение свидетельствует о том, что телефон располагается вертикально и слегка наклонен назад:



Акселерометр очень чувствителен и требует сглаживания данных. Это и другие связанные с акселерометром вопросы обсуждаются в главе 24

Географические координаты

С согласия пользователя приложение для Windows Phone 7 может принимать географические координаты телефона по методу Assisted-GPS или A-GPS.

Наиболее точный метод определения местоположения – по сигналам спутников Глобальной системы позиционирования (Global Positioning System, GPS). Но GPS может быть медленной. Эта система плохо работает в больших городах и в помещениях, ее применение энергоневыгодно, потому что приводит к большому расходу заряда батареи. Для энергосбережения и увеличения скорости, система A-GPS может определять местоположение по вышкам сотовой связи или сети. Эти методы намного более производительны и надежны, но менее точные.

Основным классом для определения местоположения является *GeoCoordinateWatcher* (Система отслеживания географических координат). Нам понадобится ссылка на сборку System.Device и директива *using* для пространства имен *System.Device.Location*. В файле WMAppManifest.xml должен присутствовать такой тег:

```
<Capability Name="ID_CAP_LOCATION" />
```

Конструктор *GeoCoordinateWatcher* принимает один из элементов перечисления *GeoPositionAccuracy* (Точность географических координат):

- *Default* (По умолчанию)
- *High* (Высокая)

После создания объекта *GeoCoordinateWatcher* необходимо задать обработчик события *PositionChanged* (Местоположение изменилось) и вызвать метод *Start*. Событие *PositionChanged* возвращает объект *GeoCoordinate* (Географические координаты), имеющий восемь свойств:

- *Latitude* (Широта), значение типа *double* в диапазоне от –90 до 90 градусов.
- *Longitude* (Долгота), значение типа *double* в диапазоне от –180 до 180 градусов.
- *Altitude* (Высота) типа *double*.
- *HorizontalAccuracy* (Точность в горизонтальном направлении) и *VerticalAccuracy* (Точность в вертикальном направлении) типа *double*.
- *Course* (Курс), значение типа *double* в диапазоне от 0 до 360 градусов.
- *Speed* (Скорость) типа *double*.
- *IsUnknown* (Неизвестно), булево значение, которое равно *true*, если *Latitude* или *Longitude* не является числом.

Если приложение не имеет разрешения на определение местоположения, свойства *Latitude* и *Longitude* будут иметь значение *Double.NaN*, и *IsUnknown* будет *true*.

Кроме того, в *GeoCoordinate* есть метод *GetDistanceTo* (Определить расстояние до), вычисляющий расстояние между двумя объектами *GeoCoordinate*.

Я остановлюсь на первых двух свойствах. Их называют *географическими координатами*. Они определяют точку на поверхности Земли. Широта – это угловое расстояние от экватора. Как правило, широта обозначается как угол от 0 до 90 градусов с указанием С или Ю (сервер или юг). Например, широта Нью-Йорка примерно 40°С. В объекте *GeoCoordinate* широты к северу от экватора соответствуют положительным значениям, и широты к югу от экватора – отрицательным. Северный Полюс соответствует 90°, и Южный Полюс – –90°.

Географические местоположения с одинаковой широтой образуют *линию широты* или *параллель*. Для заданной широты долгота – это угловое расстояние от нулевого меридиана, который проходит через Гринвичскую астрономическую обсерваторию, Англия. Долготу определяют как западную или восточную. Нью-Йорк располагается на 74° западной долготы, потому что он находится на запад от нулевого меридиана. В объекте *GeoCoordinate* положительные значения долготы соответствуют восточному полушарию, и отрицательные значения – западному. Значения 180 и –180 встречаются на линии перемены дат.

Пространство имен *System.Device.Location* включает классы, которые используют географические координаты для определения адресов (улиц и городов), но эта возможность не реализована в первой выпущенной версии Windows Phone 7.

Проект XnaLocation просто обеспечивает вывод на экран числовых значений.

Проект XNA: XnaLocation Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
```

```
    string text = "Obtaining location...";
    Viewport viewport;
    Vector2 textPosition;
    …
}
```

Как и в случае с акселерометром, *GeoCoordinateWatcher* создается и инициализируется в перегруженном *Initialize*. Обработчик события вызывается в том же потоке, поэтому для форматирования результатов в строку не надо делать ничего особенного:

Проект XNA: XnaLocation Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    GeoCoordinateWatcher geoWatcher = new GeoCoordinateWatcher();
    geoWatcher.PositionChanged += OnGeoWatcherPositionChanged;
    geoWatcher.Start();

    base.Initialize();
}

void OnGeoWatcherPositionChanged(object sender,
                                 GeoPositionChangedEventArgs<GeoCoordinate> args)
{
    text = String.Format("Latitude: {0:F3}\r\n" +
                         "Longitude: {1:F3}\r\n" +
                         "Altitude: {2}\r\n\r\n" +
                         "{3}",
                         args.Position.Location.Latitude,
                         args.Position.Location.Longitude,
                         args.Position.Location.Altitude,
                         args.Position.Timestamp);
}
```

Метод *LoadContent* просто получает шрифт и сохраняет *Viewport* для позиционирования текста впоследствии:

Проект XNA: XnaLocation Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    viewport = this.GraphicsDevice.Viewport;
}
```

Размер отображаемой строки может меняться в зависимости от значений, поэтому ее местоположение вычисляется на основании ее размера и значений *Viewport* в методе *Update*:

Проект XNA: XnaLocation Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    Vector2 textSize = segoe14.MeasureString(text);
    textPosition = new Vector2((viewport.Width - textSize.X) / 2,
```

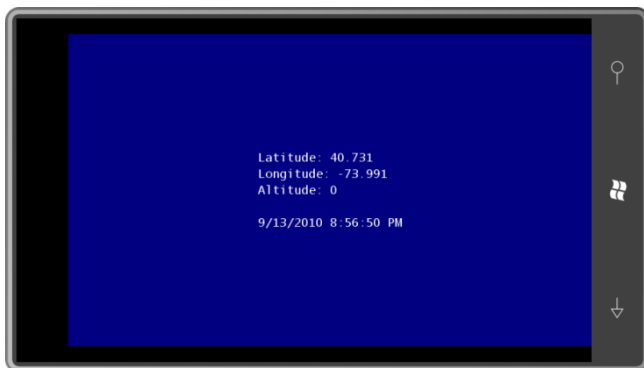
```
                                  (viewport.Height - textSize.Y) / 2);
    base.Update(gameTime);
}
```

Метод *Draw* обычный:

Проект XNA: XnaLocation Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(kootenay14, text, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

GeoCoordinateWatcher выполняется в течение всего выполнения приложения, поэтому он должен обновлять данные о местоположении в случае перемещения телефона. Вот где я живу:



Но на эмуляторе телефона приложение *GeoCoordinateWatcher* может не работать. При использовании с некоторыми бета-версиями инструментов разработки для Windows Phone 7 акселерометр всегда возвращает координаты Принстона, Нью-Джерси, наверное, как робкое напоминание о месте, где Алан Тьюринг получил свою первую ученую степень.

Использование картографического сервиса

Несомненно, для большинства людей, которых интересует их местоположение, предпочтительнее увидеть карту, а не множество числовых координат. Демонстрационное приложение на Silverlight службы определения местоположения выводит на экран карту, которая поступает в программу в виде растрового изображения.

В реальном приложении для телефона, скорее всего, использовался бы Bing Maps, в частности, элемент управления Silverlight Bing Maps, специально предназначенный для телефона. К сожалению, применение Bing Maps в приложении требует открытия учетной записи разработчика и получения ключа доступа и учетных данных. Все это делается бесплатно и просто, но данный вариант не очень удобен для программы, используемой в качестве примера в книге.

Поэтому я воспользуюсь альтернативным вариантом, для которого не нужны ключи или учетные данные. Такой альтернативой являются Microsoft Research Maps, которые можно

найти на сайте *msrmaps.com*. Материалы аэрофотосъемки предоставлены Службой геологии, геодезии и картографии США (United States Geological Survey, USGS). Microsoft Research Maps предоставляет эти снимки через Веб-сервис MSR Maps Service, который по старой памяти до сих пор называют TerraService.

Недостаток в том, что эти снимки несколько устаревшие, и сервис не вполне надежен.

MSR Maps Service – это сервис SOAP (Simple Object Access Protocol[1]), операции которого описаны в файле WSDL (Web Services Description Language[2]). На нижнем уровне все транзакции между приложением и Веб-сервисом осуществляются в виде XML-файлов. Но чтобы облегчить жизнь разработчика, обычно на базе WSDL-файла создается *прокси*. Он представляет собой коллекцию классов и структур, благодаря которым приложение может взаимодействовать с Веб-сервисом посредством вызовов методов и событий.

Этот прокси можно сформировать прямо в Visual Studio. Вот как я сделал это. Сначала я создал в Visual Studio проект Windows Phone 7 под названием SilverlightLocationMapper. В Solution Explorer я щелкнул правой кнопкой мыши имя проекта и выбрал Add Service Reference (Добавить ссылку на сервис). В поле Address (Адрес) я ввел URL WSDL-файла MSR Maps Service: *http://MSRMaps.com/TerraService2.asmx*.

(Можно было бы предположить, что URL должен быть *http://msrmaps.com/TerraService2.asmx?WSDL*, поскольку именно так обычно описываются ссылки на WSDL-файлы. Такой адрес работал бы только поначалу, поскольку является устаревшим.)

После того, как ввели URL в поле Address, нажмите Go. Visual Studio выполнит доступ к сайту и возвратит то, что найдет там. Это будет один сервис со старым именем TerraService.

После этого можно заменить универсальное имя ServiceReference1 в поле Namespace (Пространство имен). Я ввел имя MsrMapsService и нажал OK.

MsrMapsService появится под проектом в Solution Explorer. Если щелкнуть маленький значок Show All Files (Показать все файлы) вверху Solution Explorer, можно увидеть все созданные файлы. В частности, под MsrMapsService и Reference.svcmap вы увидите вложенный Reference.cs – большой файл (более 4000 строк) с пространством имен XnaLocationMapper.MsrMapsService, являющимся сочетанием исходного имени проекта и имени, выбранного нами для Веб-сервиса.

Этот файл Reference.cs включает все классы и структуры, необходимые для работы с Веб-сервисом и задокументированные на сайте *msrmaps.com*. Чтобы получить возможность работать с этими классами в приложении, добавим директиву *using*:

```
using SilverlightLocationMapper.MsrMapsService;
```

Также понадобиться ссылка на сборку System.Device и директивы *using* для пространств имен *System.Device.Location*, *System.IO* и *System.Windows.Media.Imaging*.

В файле MainPage.xaml я сохранил для свойства *SupportedOrientations* настройку по умолчанию, *Portrait*, удалил заголовок страницы, чтобы высвободить больше места, и переместил панель заголовка под сетку для содержимого, просто на случай, если вдруг что-то выйдет за ее рамки и заслонит заголовок. Перенос панели заголовка под сетку для содержимого в файле XAML гарантирует, что визуально она будет расположена сверху.

Рассмотрим сетку для содержимого:

---

[1] Простой протокол доступа к объектам (*прим. переводчика*).
[2] Язык описания Веб-сервисов (*прим. переводчика*).

Проект Silverlight: SilverlightLocationMapper Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="statusText"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               TextWrapping="Wrap" />

    <Image Source="Images/usgslogoFooter.png"
           Stretch="None"
           HorizontalAlignment="Right"
           VerticalAlignment="Bottom" />
</Grid>
```

TextBlock используется для отображения состояния и (возможных) ошибок; *Image* выводит логотип United States Geological Survey.

Растровые изображения карт будут вставляться между *TextBlock* и *Image*, таким образом, они будут заслонять *TextBlock*, но *Image* останется сверху.

В файле выделенного кода всего два поля: одно для *GeoCoordinateWatcher*, который обеспечивает данные о местоположении; и другое для прокси-класса, созданного при добавлении Веб-сервиса:

Проект Silverlight: SilverlightLocationMapper Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    GeoCoordinateWatcher geoWatcher = new GeoCoordinateWatcher();
    TerraServiceSoapClient proxy = new TerraServiceSoapClient();
    …
}
```

Прокси-класс используется путем вызова его методов, которые делают сетевые запросы. Все эти методы являются асинхронными. Для каждого вызываемого метода должен быть обеспечен обработчик события завершения выполнения метода, которое формируется при поступлении запрашиваемых данных в приложение.\

Событие завершение выполнения имеет несколько аргументов. Ими являются свойство *Cancelled* (Отменен) типа *bool*; свойство *Error* (Ошибка), которое равно *null*, если ошибки нет; и свойство *Result* (Результат), которое зависит от того, что запрашивалось.

Я захотел, чтобы процесс начинался после загрузки приложения и вывода его на экран, поэтому задал обработчик события *Loaded*. Это обработчик *Loaded* определяет обработчики для двух событий завершения, которые я запрошу от прокси, и также запускает *GeoCoordinateWatcher*:

Проект Silverlight: SilverlightLocationMapper Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();
    Loaded += OnMainPageLoaded;
}

void OnMainPageLoaded(object sender, RoutedEventArgs args)
{
    // Задаем обработчики событий для прокси TerraServiceSoapClient
    proxy.GetAreaFromPtCompleted += OnProxyGetAreaFromPtCompleted;
```

```
    proxy.GetTileCompleted += OnProxyGetTileCompleted;

    // Запускаем выполнение GeoCoordinateWatcher
    statusText.Text = "Obtaining geographic location...";
    geoWatcher.PositionChanged += OnGeoWatcherPositionChanged;
    geoWatcher.Start();
}
```

Когда координаты получены, вызывается следующий метод *OnGeoWatcherPositionChanged* (При изменении местоположения). Этот метод начинает выполнение с отключения *GeoCoordinateWatcher*. Программа не может постоянно обновлять экран, поэтому никакие дополнительные сведения о местоположении ей уже не нужны. Долгота и широта выводятся в *TextBlock* под названием *ApplicationTitle* (Заголовок приложения), отображаемом вверху экрана.

Проект Silverlight: SilverlightLocationMapper Файл: MainPage.xaml.cs (фрагмент)

```
void OnGeoWatcherPositionChanged(object sender,
                                 GeoPositionChangedEventArgs<GeoCoordinate> args)
{
    // Отключаем GeoWatcher
    geoWatcher.PositionChanged -= OnGeoWatcherPositionChanged;
    geoWatcher.Stop();

    // Координаты задаются как текст заголовка
    GeoCoordinate coord = args.Position.Location;
    ApplicationTitle.Text += ": " + String.Format("{0:F2}°{1} {2:F2}°{3}",
                                            Math.Abs(coord.Latitude),
                                            coord.Latitude > 0 ? 'N' : 'S',
                                            Math.Abs(coord.Longitude),
                                            coord.Longitude > 0 ? 'E' : 'W');
    // Запрашиваем прокси для AreaBoundingBox
    LonLatPt center = new LonLatPt();
    center.Lon = args.Position.Location.Longitude;
    center.Lat = args.Position.Location.Latitude;

    statusText.Text = "Accessing Microsoft Research Maps Service...";
    proxy.GetAreaFromPtAsync(center, 1, Scale.Scale16m,
(int)ContentPanel.ActualWidth,

(int)ContentPanel.ActualHeight);
}
```

Метод завершается после первого обращения к прокси. При вызове *GetAreaFromPtAsync* широта и долгота используются как точка центрирования. Также передаются некоторые другие данные. Второй аргумент – 1, чтобы получить вид со спутника, и 2, чтобы получить карту (как будет показано в конце данной главы). Третий аргумент – это желаемый масштаб изображения, элемент перечисления *Scale*. Выбранный в данном случае элемент означает, что каждый пиксел возвращенного растрового изображения эквивалентен 16 метрам.

Будьте внимательны, некоторые коэффициенты масштабирования (в частности, *Scale2m*, *Scale8m* и *Scale32m*) обусловливают возвращение файлов в формате GIF. Никогда не забывайте, что Silverlight не поддерживает GIF! Для остальных масштабных коэффициентов возвращаются файлы JPEGS.

Последними аргументами *GetAreaFromPtAsync* являются ширина и высота области, которую должна будет занимать карта.

Все возвращаемые MSR Maps Service растровые изображения квадратные со стороной 200 пикселов. Практически всегда для заполнения всей выделенной области потребуется

несколько таких изображений. Например, если последние два аргумента *GetAreaFromPtAsync* – 400 и 600, понадобится 6 растровых изображений.

Ну, на самом деле, нет. Для заполнения области высотой 400 и шириной 600 пикселов потребуется 12 растровых изображений, 3 по горизонтали и 4 по вертикали.

Загвоздка вот в чем. Эти растровые изображения не создаются специально по запросу приложения. Они уже существуют на сервере во всех возможных масштабах. Географические координаты, охватываемые этими растровыми изображениями, фиксированы. Наша задача покрыть определенную область экрана фрагментами карты, и эта область должна быть центрирована соответственно заданным координатам. Но существующие фрагменты не будут подходить точно, часть из них поместится в выделенной области, а некоторые неизбежно будут выходить за края.

В результате вызова *GetAreaFromPtAsync* (в следующем методе *OnProxyGetAreaFromPtCompleted*) возвращается объект типа *AreaBoundingBox* (Ограничивающее область окно). Это довольно сложная структура, которая, тем не менее, содержит все данные, необходимые для запроса всех требуемых элементов карты по отдельности и последующей их компоновки в сетке.

Проект Silverlight: SilverlightLocationMapper   Файл: MainPage.xaml.cs (фрагмент)

```
void OnProxyGetAreaFromPtCompleted(object sender, GetAreaFromPtCompletedEventArgs
args)
{
    if (args.Error != null)
    {
        statusText.Text = args.Error.Message;
        return;
    }

    statusText.Text = "Getting map tiles...";

    AreaBoundingBox box = args.Result;
    int xBeg = box.NorthWest.TileMeta.Id.X;
    int yBeg = box.NorthWest.TileMeta.Id.Y;
    int xEnd = box.NorthEast.TileMeta.Id.X;
    int yEnd = box.SouthWest.TileMeta.Id.Y;

    // Выбор всех необходимых фрагментов карты
    for (int x = xBeg; x <= xEnd; x++)
        for (int y = yBeg; y >= yEnd; y--)
        {
            // Создание объекта Image для отображения фрагмента карты
            Image img = new Image();
            img.Stretch = Stretch.None;
            img.HorizontalAlignment = HorizontalAlignment.Left;
            img.VerticalAlignment = VerticalAlignment.Top;
            img.Margin = new Thickness((x - xBeg) * 200 -
box.NorthWest.Offset.XOffset,
                                       (yBeg - y) * 200 -
box.NorthWest.Offset.YOffset,
                                       0, 0);

            // Вставка после TextBlock, но перед Image с логотипом
            ContentPanel.Children.Insert(1, img);

            // Определение ID фрагмента карты
            TileId tileId = box.NorthWest.TileMeta.Id;
            tileId.X = x;
            tileId.Y = y;

            // Вызов прокси для получения фрагмента карты (Обратите внимание, что
```

```
Image является пользовательским объектом)
            proxy.GetTileAsync(tileId, img);
        }
}
```

Не буду останавливаться на *AreaBoundingBox* подробно, потому что он достаточно хорошо описан на сайте *msrmaps.com*. Я нашел там очень полезные фрагменты кода, реализующие аналогичную логику, которые были написаны для Windows Forms (которые, как мне кажется, уже несколько устарели).

Обратите внимание, что в цикле для каждого фрагмента карты создается объект *Image*. Все эти объекты имеют одинаковые значения свойств *Stretch*, *HorizontalAlignment* и *VerticalAlignment*, но разные *Margin*. *Margin* определяет, как каждый фрагмент размещается в сетке для содержимого. Свойства *XOffset* (Смещение по X) и *YOffset* (Смещение по Y) показывают, на сколько фрагменты карты будут выступать сверху и слева. Сетка для содержимого не обрезает содержимого, поэтому эти фрагменты могут выступать за границу страницы приложения.

Заметьте также, что каждый объект *Image* передается как второй аргумент в метод *GetTileAsync* прокси-класса. Это аргумент *UserState* (Состояние пользователя). Прокси ничего не делает с этим аргументом, просто возвращает его как свойство *UserState* аргументов события завершения, что показано ниже:

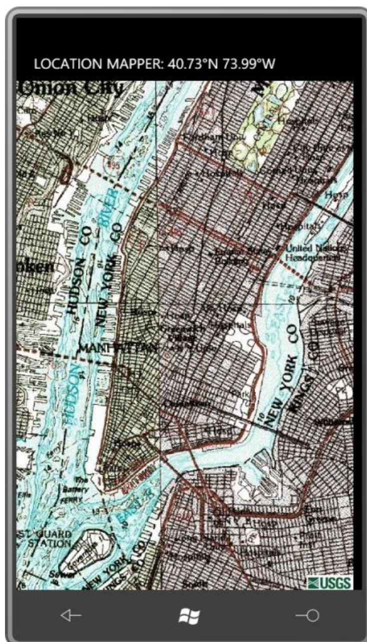
Проект Silverlight: SilverlightLocationManager   Файл: MainPage.xaml.cs (фрагмент)

```
void OnProxyGetTileCompleted(object sender, GetTileCompletedEventArgs args)
{
    if (args.Error != null)
    {
        return;
    }

    Image img = args.UserState as Image;
    BitmapImage bmp = new BitmapImage();
    bmp.SetSource(new MemoryStream(args.Result));
    img.Source = bmp;
}
```

Вот так метод связывает отдельный фрагмент растрового изображения с определенным элементом *Image*, уже находящимся в сетке для содержимого.

По собственному опыту я знаю, что в большинстве случаев приложение не получает всех запрашиваемых фрагментов. Если вам посчастливится и, кроме того, вы запустите это приложение, находясь где-то в окрестностях моего дома, экран будет выглядеть следующим образом:

Если изменить второй аргумент в вызове *proxy.GetAreaFromPtAsync* и задать 2 вместо 1, приложению будут возвращены изображения карты, а не снимки со спутника:



В этом есть некоторое очарование старины (и я поклонник живописи), но, боюсь, современные пользователи привыкли уже к чему-то более «продвинутому».

Глава 6
Вопросы архитектуры приложений

Приложение на Silverlight для Windows Phone 7 включает несколько стандартных классов:

- класс *App*, производный от *Application*;
- экземпляр класса *PhoneApplicationFrame*;
- и один или более классов, производных от *PhoneApplicationPage*.

Данная глава отчасти посвящена вот этим «и более» классам. Рассматриваемые до сих пор приложения включали всего один класс *MainPage*, наследуемый от *PhoneApplicationPage*. В более сложных приложениях возможно наличие нескольких страниц, по которым пользователь может перемещаться, во многом аналогично навигации по Веб-страницам.

Может показаться, что навигация - это лишь дополнительный аспект программирования на Silverlight, применимый только к написанию приложений на Silverlight и не имеющий отношения к XNA. Но навигация напрямую связана с очень важными для приложений Windows Phone 7 вопросами *захоронения* (*tombstoning*). Захоронение приложения Windows Phone 7 происходит, когда пользователь переходит к другому приложению через стартовый экран телефона. Это касается и приложений на XNA.

Реализация простейшей навигации

Проект SilverlightSimpleNavigation начинается как обычно, с класса *MainPage*. Также, как обычно, описываем два элемента *TextBlock* для заголовков:

Проект Silverlight: SilverlightSimpleNavigation Файл: MainPage.xaml (фрагмент)

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="SIMPLE NAVIGATION" … />
    <TextBlock x:Name="PageTitle" Text="main page" … />
</StackPanel>
```

Область содержимого MainPage.xaml включает только *TextBlock*, определяющий обработчик для своего события *ManipulationStarted*:

Проект Silverlight: SilverlightSimpleNavigation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Navigate to 2nd Page"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Padding="0 34"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

Обратите внимание на свойство *Text* этого *TextBlock*: «Navigate to 2nd page» (Перейти на вторую страницу). Файл выделенного кода включает обработчик *ManipulationStarted* и также перегрузку метода *OnManipulationStarted* для всей страницы:

Проект Silverlight: SilverlightSimpleNavigation Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();

    public MainPage()
    {
        InitializeComponent();
    }

    void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
    {
        this.NavigationService.Navigate(new Uri("/SecondPage.xaml",
UriKind.Relative));

        args.Complete();
        args.Handled = true;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentPanel.Background = new SolidColorBrush(
            Color.FromArgb(255, (byte)rand.Next(255),
                                (byte)rand.Next(255),
                                (byte)rand.Next(255)));

        base.OnManipulationStarted(args);
    }
}
```

При касании страницы в любом месте вне *TextBlock* фон *ContentPanel* меняет цвет случайным образом. Когда касание происходит в области *TextBlock*, обработчик выполняет доступ к свойству *NavigationService* (Служба навигации) страницы. Это объект типа *NavigationService*, включающий свойства, методы и события, связанные с навигацией, в том числе и самый важный метод *Navigate* (Перейти):

```
this.NavigationService.Navigate(new Uri("/SecondPage.xaml", UriKind.Relative));
```

Аргументом этого метода является объект типа *Uri*. Обратите внимание на косую черту перед SecondPage.xaml и использование *UriKind.Relative* для обозначения URI относительно страницы MainPage.xaml.

Создаем вторую страницу в проекте SilverlightSimpleNavigation, щелкнув правой кнопкой мыши имя проекта в обозревателе проекта в Visual Studio и выбрав в меню Add и New Item. В диалоговом окне Add New Item (Добавить новый элемент) выберем Windows Phone Portrait Page (Страница Windows Phone в портретном режиме) и назовем ее SecondPage.xaml (Вторая страница).

В результате этой операции создается не только SecondPage.xaml, но и файл выделенного кода SecondPage.xaml.cs. Эти два файла SecondPage практически ничем не отличаются от файлов MainPage, создаваемых Visual Studio в обычном порядке. Как и *MainPage*, *SecondPage* наследуется от *PhoneApplicationPage*.

Имя приложения в SecondPage.xaml будет тем же, что и в FirstPage.xaml, но заголовок страницы – «second page» (вторая страница):

Проект Silverlight: SilverlightSimpleNavigation Файл: SecondPage.xaml (фрагмент)

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="SIMPLE NAVIGATION" … />
    <TextBlock x:Name="PageTitle" Text="second page" … />
</StackPanel>
```

Область содержимого SecondPage.xaml практически аналогична MainPage.xaml, только в *TextBlock* отображается «Go Back to 1st Page» (Назад на первую страницу):

Проект Silverlight: SilverlightSimpleNavigation Файл: SecondPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Go Back to 1st Page"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Padding="0 34"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

Файл выделенного кода класса *SecondPage* также очень схож с классом *FirstPage*:

Проект Silverlight: SilverlightSimpleNavigation Файл: SecondPage.xaml.cs (фрагмент)

```
public partial class SecondPage : PhoneApplicationPage
{
    Random rand = new Random();

    public SecondPage()
    {
       InitializeComponent();
    }

    void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs 
args)
    {
        this.NavigationService.GoBack();

        args.Complete();
        args.Handled = true;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentPanel.Background = new SolidColorBrush(
            Color.FromArgb(255, (byte)rand.Next(255),
                                (byte)rand.Next(255),
                                (byte)rand.Next(255)));

        base.OnManipulationStarted(args);
    }
}
```

Опять же, при касании в любой области экрана вне *TextBlock* фон меняет цвет случайным образом. Когда касание происходит в рамках *TextBlock*, обработчик вызывает другой метод *NavigationService*:

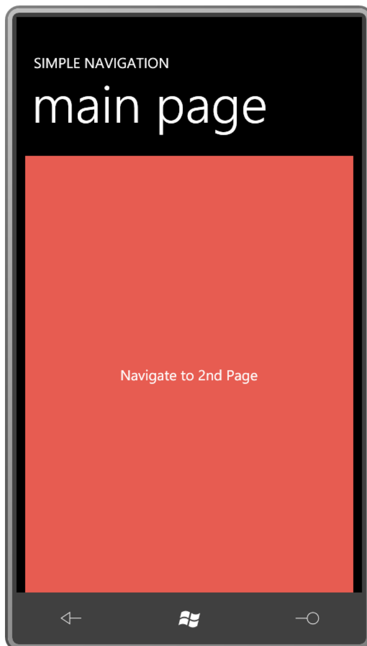
```
this.NavigationService.GoBack();
```

Этот вызов возвращает приложение на страницу, с которой был выполнен переход на SecondPage.xaml, в данном случае это MainPage.xaml. Еще раз посмотрим на вызов метода *Navigate* в MainPage.xaml.cs:

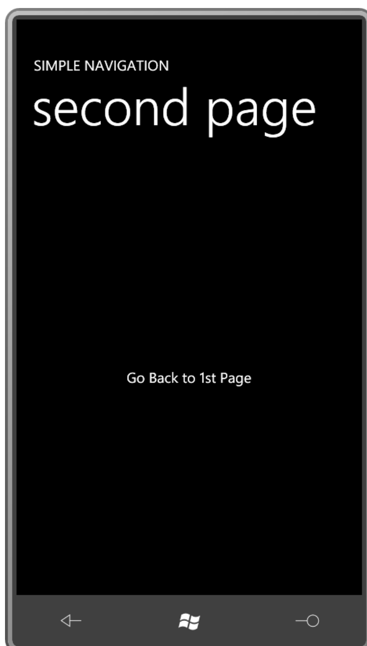
```
this.NavigationService.Navigate(new Uri("/SecondPage.xaml", UriKind.Relative));
```

В основе реализации навигации в приложении на Silverlight лежат XAML-файлы во многом аналогично тому, как в традиционной Веб-среде для этого используются HTML-файлы. Экземпляр класса *SecondPage* создается в фоновом режиме. Экземпляр *PhoneApplicationFrame* реализует в приложении практически все механизмы навигации, но открытый интерфейс *PhoneApplicationFrame* включает не экземпляры производных *PhoneApplicationPage*, а объекты *Uri* и XAML-файлы.

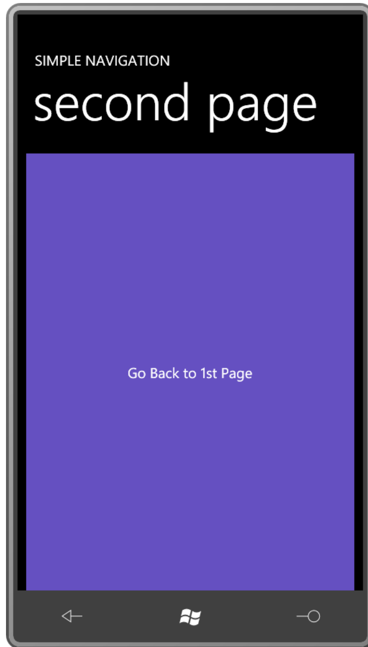
Запустим приложение. Выполнение начинается с главной страницы. Можно коснуться экрана, чтобы изменить его цвет:



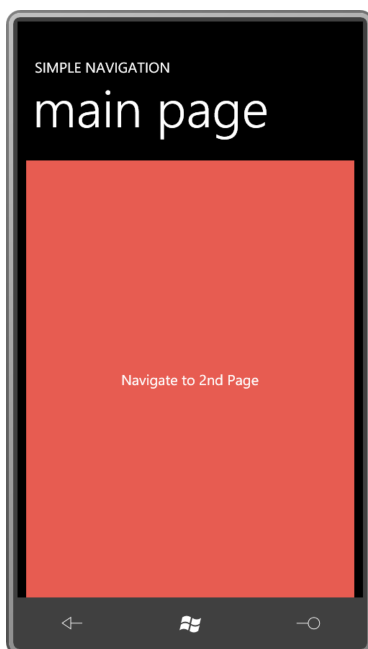
Теперь коснемся *TextBlock* с надписью «Navigate to 2nd Page». На экран выводится вторая страница:



Можете коснуться экрана, чтобы изменить его цвет:



Теперь коснемся *TextBlock* с надписью «Go Back to 1st Page». (Или можно нажать кнопку Back телефона.) Это возвратит нас на главную страницу. Ее цвет остался неизменным с момента, когда мы покинули ее:



Теперь снова коснемся *TextBlock*, чтобы перейти на вторую страницу:

Фон черный. Вторая страница *не* сохраняет цвет, заданный ей при предыдущем нашем посещении. Абсолютно очевидно, что это совершенно новый экземпляр класса *SecondPage*.

Система навигации в Silverlight для Windows Phone строится на концепции *стека* – структуры данных с доступом по принципу «last-in-first-out[1]». В данной книге страницу, вызвавшую метод *Navigate*, я буду называть *исходной* страницей, и страницу, на которую выполняется переход – страницей *перехода*. После вызова *Navigate* исходная страница помещается в стек, и новый экземпляр страницы перехода создается и отображается. После вызова *GoBack* (Вернуться) (или нажатия кнопки Back телефона), текущая страница закрывается, из стека извлекается страница, помещенная туда последней, и выводится на экран.

В приложении на Silverlight кнопка Back телефона выполняет ту же функцию, что и вызов метода *GoBack*. Исключением является случай, когда вы находитесь на начальной странице приложения. Тогда по нажатию кнопки Back приложение прекращает выполнение.

Попробуем заменить вызов *GoBack* в SecondPage.xaml.cs следующим:

```
this.NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
```

Это не то же самое, что вызов *GoBack*, и он не обеспечит возвращения к исходному экземпляру *MainPage*. В результате этого вызова *SecondPage* перейдет к *новому* экземпляру *MainPage*. Если продолжить нажимать *TextBlock* на каждой открывающейся странице, стек заполнится чередующимися экземплярами *MainPage* и *SecondPage*, каждый из которых может быть разного цвета. Чтобы закрыть все эти страницы и, в конце концов, завершить выполнение приложения, придется воспользоваться кнопкой Back телефона.

Navigate и *GoBack* – два основных метода *NavigationService*. Маловероятно, что в приложениях для телефона придется пользоваться чем-то сверх этого. Никогда нельзя забывать, что для телефона нет особого смысла реализовывать очень сложные схемы переходов в рамках приложения, а также необходимо обеспечивать пользователю некоторое средство отслеживания того, как он попал на текущую страницу и как вернуться назад.

---

[1] Последний вошел, первый вышел (*прим. переводчика*).

Но, наверное, основным применением вторичных страниц в приложении на Silverlight для телефона является использование их в качестве диалоговых окон. Если приложению требуется получить некоторые данные от пользователя, оно переходит на новую страницу для сбора этих сведений. Пользователь вводит необходимые данные и возвращается к основной странице. Такой сценарий будет продемонстрирован в главе 10.

Передача данных на страницы

Вероятность использования страниц в качестве диалоговых окон подымает два вопроса:

- Как реализовать передачу данных с исходной страницы на страницу перехода?
- Как обеспечить возвращение данных при возвращении на исходную страницу?

Любопытно что для первого случая реализована специальная возможность, а вот для второго нет. Рассмотрим эту возможность и затем обратимся к самым общим решениям второй проблемы.

Следующий проект называется SilverlightPassData. Он практически повторяет первый пример данной главы, за исключением того что при переходе с *MainPage* к *SecondPage* в *SecondPage* передается текущий цвет фона главной страницы, и *SecondPage* создается с этим исходным цветом.

Область содержимого MainPage.xaml такая же, как и в предыдущем приложении:

Проект Silverlight: SilverlightPassData Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Navigate to 2nd Page"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Padding="0 34"
               ManipulationStarted="OnTextBlockManipulationStarted" />
</Grid>
```

Не будем останавливаться на перегруженном методе *OnManipulationStarted*, он абсолютно аналогичен предыдущему примеру, а вот обработчик события *ManipulationStarted* для *TextBlock* немного доработан:

Проект Silverlight: SilverlightPassData Файл: MainPage.xaml.cs (фрагмент)

```
void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
{
    string destination = "/SecondPage.xaml";

    if (ContentPanel.Background is SolidColorBrush)
    {
        Color clr = (ContentPanel.Background as SolidColorBrush).Color;
        destination += String.Format("?Red={0}&Green={1}&Blue={2}",
                                     clr.R, clr.G, clr.B);
    }

    this.NavigationService.Navigate(new Uri(destination, UriKind.Relative));

    args.Complete();
    args.Handled = true;
}
```

Если для *ContentPanel* (Панель содержимого) в качестве *Background* (Фон) используется кисть *SolidColorBrush*, обработчик получает *Color* и трансформирует значения для красного, зеленого и синего каналов в строку, которая прикрепляется к имени страницы перехода. После этого URI приобретает примерно такой вид:

"/SecondPage.xaml?Red=244&Green=43&Blue=91"

Как видите, это обычная строка HTML-запроса.

Также в проекте SilverlightPassData имеется класс *SecondPage*, полностью аналогичный классу с таким же именем из первого проекта за исключением того, что в файл выделенного кода включен перегруженный метод *OnNavigatedTo* (При переходе к):

Проект Silverlight: SilverlightPassData Файл: SecondPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    IDictionary<string, string> parameters = this.NavigationContext.QueryString;

    if (parameters.ContainsKey("Red"))
    {
        byte R = Byte.Parse(parameters["Red"]);
        byte G = Byte.Parse(parameters["Green"]);
        byte B = Byte.Parse(parameters["Blue"]);

        ContentPanel.Background =
            new SolidColorBrush(Color.FromArgb(255, R, G, B));
    }

    base.OnNavigatedTo(args);
}
```

В класс *NavigationEventArgs* (Параметры события перехода) посредством директивы *using* понадобится включить пространство имен *System.Windows.Navigation*.

Метод *OnNavigatedTo* определяется классом *Page*, от которого наследуется *PhoneApplicationPage*. Этот метод вызывается сразу после создания страницы. В тот момент, когда вызывается *OnNavigatedTo*, конструктор страницы уже выполнен, конечно же, но практически больше ничего не произошло.

Класс страницы перехода может выполнять доступ к строкам запроса, используемым для вызова страницы, через свойство *NavigationContext* (Контекст перехода) типа *NavigationContext*. У класса *NavigationContext* только одно открытое свойство *QueryString* (Строка запроса), которое возвращает словарь, сохраненный в переменной *parameters* (параметры). В данном примере предполагается, что если присутствует строка запроса «Red», то должны существовать также «Blue» и «Green». Все строки передаются в метод *Byte.Parse*, в котором на их основании выполняется восстановление значения цвета.

Теперь при переходе с *MainPage* к *SecondPage* цвет фона остается неизменным. Однако при возвращении назад на *MainPage* этого не происходит. Такой встроенной возможности, как строка запроса, для передачи данных с одной страницы на другую нет.

Совместное использование данных страницами

Не стоит забывать, что все страницы приложения имеют удобный доступ к классу *App*, производному от *Application*. Статическое свойство *Application.Current* возвращает объект *Application*, связанный с приложением, и его можно просто привести к *App*. Это означает, что

в классе *App* можно хранить данные для совместного использования несколькими страницами приложения.

В классе *App* проекта SilverlightShareData опишем простое открытое свойство:

Проект Silverlight: SilverlightShareData Файл: App.xaml.cs (фрагмент)

```
public partial class App : Application
{
    // открытое свойство для хранения данных, совместно используемых страницами
    public Color? SharedColor { set; get; }

    …
}
```

В данном примере свойство *Color* описано не просто как *Color*, а как свойство, допускающее пустое значение. Это сделано для случаев, когда свойство *Background* панели содержимого (*ContentPanel*) не *SolidColorBrush*. Тогда свойство *Background* имеет значение *null*, и *Color* не должно в нем храниться. Если бы это свойство было типа *Color*, значение *Color* сохранялось бы в нем по умолчанию; значение *null* для *Color* соответствует прозрачному черному, что является неверным. Даже непрозрачный черный является неверным значением, если пользователь выбрал светлую цветовую схему.

Приложение практически ничем не отличается, только при касании *TextBlock* на *MainPage* обработчик сначала делает попытку сохранить цвет в новом свойстве класса *App* и уже после этого выполняет переход на *SecondPage*:

Проект Silverlight: SilverlightShareData Файл: MainPage.xaml.cs (фрагмент)

```
void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
{
    if (ContentPanel.Background is SolidColorBrush)
        (Application.Current as App).SharedColor =
                        (ContentPanel.Background as SolidColorBrush).Color;

    this.NavigationService.Navigate(new Uri("/SecondPage.xaml", UriKind.Relative));

    args.Complete();
    args.Handled = true;
}
```

После этого перегруженный *OnNavigatedTo* в *SecondPage* выполняет доступ к этому свойству:

Проект Silverlight: SilverlightShareData Файл: SecondPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    Color? sharedColor = (Application.Current as App).SharedColor;

    if (sharedColor != null)
        ContentPanel.Background =
                new SolidColorBrush(sharedColor.Value);

    base.OnNavigatedTo(args);
}
```

Аналогично при нажатии *TextBlock* на *SecondPage* обработчик сохраняет текущее значение цвета фона в класс *App* и только после этого вызывает *GoBack*:

Проект Silverlight: SilverlightShareData Файл: SecondPage.xaml.cs (фрагмент)

```
void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
{
    if (ContentPanel.Background is SolidColorBrush)
        (Application.Current as App).SharedColor =
                        (ContentPanel.Background as SolidColorBrush).Color;

    this.NavigationService.GoBack();

    args.Complete();
    args.Handled = true;
}
```

В классе *MainPage* также имеется перегруженный *OnNavigatedTo*, т.е. он тоже может извлекать сохраненный цвет и задавать его как фоновый цвет контейнера:

Проект Silverlight: SilverlightShareData Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    Color? sharedColor = (Application.Current as App).SharedColor;

    if (sharedColor != null)
        ContentPanel.Background =
                new SolidColorBrush(sharedColor.Value);

    base.OnNavigatedTo(args);
}
```

Теперь при переходе со страницы на страницу передается и цвет страницы.

Использовать класс *App* как хранилище совместно используемых данных настолько удобно, что эта методика получила широкое распространение. Тем не менее, следует рассмотреть более структурированные решения, в которых задействованы только страницы, участвующие в переходах, без привлечения какого-либо стороннего класса, каким является *App*.

Кроме виртуального метода *OnNavigatedTo*, класс *Page* также определяет метод *OnNavigatedFrom* (При переходе от), который на первый взгляд кажется намного менее полезным, ведь в любом случае страница знает, что с нее выполняется переход, потому что только что был вызван метод *Navigate* или *GoBack*.

Но и *OnNavigatedFrom*, и *OnNavigatedTo* включают параметры события типа *NavigationEventArgs*, который описывает два свойства: *Uri* типа *Uri* и *Content* типа *object*. Эти свойства всегда определяют страницу, к которой выполняется переход.

Например, *MainPage* вызывает *Navigate*, передавая в него аргумент «/SecondPage.xaml». При вызове метода *OnNavigatedFrom* в него передаются аргументы события, где свойство *Uri* указывает на «/SecondPage.xaml» и свойство*Content* типа *SecondPage*. Это вновь созданный экземпляр *SecondPage*, который будет выведен на экран после перехода, и это наиболее удобный способ получить этот экземпляр. Далее вызывается метод *OnNavigatedTo* *SecondPage* с теми же параметрами события: *Uri*, указывающим на «/SecondPage.xaml», и объектом *SecondPage*.

Аналогично, когда *SecondPage* вызывает метод *GoBack*, вызывается метод *OnNavigatedFrom*, и в него передаются параметры, включающие свойство *Uri*, которое указывает на «/MainPage.xaml», и свойство *Content* с экземпляром *MainPage*. После этого вызывается метод *OnNavigatedTo MainPage* с теми же параметрами события.

Это означает, что во время выполнения метода *OnNavigatedFrom* класс имеет возможность задать свойство или вызвать метод класса страницы перехода.

Рассмотрим пример под названием SilverlightInsertData. Этот проект включает две страницы, *MainPage* и *SecondPage*, и XAML-файлы, аналогичные тем, которые были рассмотрены ранее. В классе *MainPage* нет какой-либо логики для реализации изменения цвета случайным образом, для получения цвета он использует *SecondPage*. *SecondPage* можно рассматривать как диалоговое окно, возвращающее случайное значение цвета в *MainPage*.

Привожу файл выделенного кода для *MainPage* почти полностью:

Проект Silverlight: SilverlightInsertData Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    public Color? ReturnedColor { set; get; }

    void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
    {
        this.NavigationService.Navigate(new Uri("/SecondPage.xaml",
UriKind.Relative));

        args.Complete();
        args.Handled = true;
    }
    …
}
```

Обратите внимание на свойство *ReturnedColor* (Возвращенный цвет) типа *Color* необязательной определенности, точно так же как в классе *App* в предыдущем приложении.

Рассмотрим файл выделенного кода *SecondPage*:

Проект Silverlight: SilverlightInsertData Файл: SecondPage.xaml.cs (фрагмент)

```
public partial class SecondPage : PhoneApplicationPage
{
    Random rand = new Random();

    public SecondPage()
    {
        InitializeComponent();
    }

    void OnTextBlockManipulationStarted(object sender, ManipulationStartedEventArgs
args)
    {
        this.NavigationService.GoBack();

        args.Complete();
```

```
        args.Handled = true;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentPanel.Background = new SolidColorBrush(
            Color.FromArgb(255, (byte)rand.Next(255),
                                (byte)rand.Next(255),
                                (byte)rand.Next(255)));

        base.OnManipulationStarted(args);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        if (ContentPanel.Background is SolidColorBrush)
        {
            Color clr = (ContentPanel.Background as SolidColorBrush).Color;

            if (args.Content is MainPage)
                (args.Content as MainPage).ReturnedColor = clr;
        }

        base.OnNavigatedFrom(e);
    }
}
```

Как и в предыдущих приложениях, *SecondPage* при касании меняет цвет фона случайным образом и при касании *TextBlock* вызывает метод *GoBack*. Изменен перегруженный метод *OnNavigatedFrom*, который вызывается вскоре после вызова *GoBack* классом. Если доступно действительное значение *SolidColorBrush*, этот метод проверяет, выполняется ли переход к объекту типа *MainPage*. Если да, он сохраняет объект *Color* в свойстве *ReturnedColor* класса *MainPage*.

MainPage может извлечь значение этого свойства в своем перегруженном методе *OnNavigatedTo*:

Проект Silverlight: SilverlightInsertData Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    …
    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        if (ReturnedColor != null)
            ContentPanel.Background =
                    new SolidColorBrush(ReturnedColor.Value);

        base.OnNavigatedTo(args);
    }
}
```

По сути, *MainPage* вызывает *SecondPage* для получения значения *Color*, просто как обычное диалоговое окно. Но *SecondPage* всегда открывается с черным фоном (или белым, если выбрана светлая цветовая тема).

SecondPage не может инициализироваться из любого свойства *MainPage*, поскольку вызов *OnNavigatedTo*, получаемый *SecondPage*, не ссылается на исходную страницу. Для обеспечения симметричной функциональности в *SecondPage* необходимо было бы описать собственное открытое свойство *Color*, которое должно было бы быть инициализировано в перегруженном *OnNavigatedFrom* в *MainPage*.

Еще одним небольшим отличием в данном приложении является описание свойства *ReturnedColor* в *SecondPage*. При переходе *MainPage* к *SecondPage* в *MainPage* вызывается метод *OnNavigatedFrom*, сохраняющий экземпляр *SecondPage*, к которому выполняется переход, в одно из полей *MainPage*. По завершении выполнения *SecondPage* сохраняет значение *Color* в своем свойстве *ReturnedColor* и вызывает *GoBack*. После этого вызывается метод *OnNavigatedTo* в *MainPage*. *MainPage* может использовать сохраненный как поле экземпляр *SecondPage* для доступа к свойству *ReturnedColor*.

Такая схема кажется вполне жизнеспособной, но работает не всегда. Проблема в невозможности гарантировать, что переход с *MainPage* и возвращение к нему будет выполнен с одного и того же экземпляра *SecondPage*. Совсем скоро мы вернемся к этому вопросу и рассмотрим его более детально.

Хранение данных вне экземпляров

Каждый раз *MainPage* переходит к новому экземпляру *SecondPage*. Именно поэтому *SecondPage* каждый раз запускается в исходном состоянии. Потому что это всегда новый экземпляр.

Если требуется, чтобы *SecondPage* «запоминал», по крайней мере, заданный цвет, необходимо каким-то образом реализовать внешнее хранение этих данных. Для этого подойдет *MainPage*.

Также данные состояния *SecondPage* могут храниться в *изолированном хранилище*. Хранение в изолированном хранилище во многом аналогично хранению на обычном дисковом накопителе. Для доступа к нему используются классы пространства имен *System.IO.IsolatedStorage*. Любое приложение Windows Phone 7 имеет доступ к изолированному хранилищу, но только к тем файлам, которое оно само создало. Изолированное хранилище позволяет приложению сохранять данные между выполнениями и идеально подходит для хранения параметров приложения.

Примеры изолированного хранилища будут представлены в данной главе несколько позже.

Третье решение обеспечивает класс *PhoneApplicationService*, описанный в пространстве имен *Microsoft.Phone.Shell*. Экземпляр *PhoneApplicationService* создается в стандартном файле App.xaml:

```
<Application.ApplicationLifetimeObjects>
    <!--Обязательный объект, обрабатывающий события жизненного цикла приложения-->
    <shell:PhoneApplicationService
        Launching="Application_Launching" Closing="Application_Closing"
        Activated="Application_Activated" Deactivated="Application_Deactivated"/>
</Application.ApplicationLifetimeObjects>
```

За тегом *PhoneApplicationService* следуют четыре события, связанные с обработчиками. Примеры этих событий будут приведены в данной главе чуть ниже. Не создавайте новый *PhoneApplicationService*. Этот существующий экземпляр *PhoneApplicationService* можно получить посредством статического свойства *PhoneApplicationService.Current*.

PhoneApplicationService включает свойство *State*. Это словарь, который обеспечивает возможность сохранять и восстанавливать данные. Свойство *State* типа *IDictionary<string, object>*. Объекты сохраняются в словарь с помощью текстовых ключей. Эти данные хранятся только в ходе выполнения приложения, поэтому такой метод хранения не подходит для параметров приложения, которые должны сохраняться между запусками приложения. Данные, сохраняемые приложением только в ходе его выполнения, иногда называют «промежуточными» данными (transient data).

Любой объект, сохраняемый в словаре *State*, должен быть сериализуемым, т.е. таким объектом, который может быть преобразован в XML и вновь восстановлен из XML. Конструктор этого объекта должен быть открытым и без параметров. Все открытые свойства объекта должны быть либо сериализуемыми, либо таких типов, для которых существуют методы *Parse* для преобразования строк в объекты.

Не всегда очевидно, какие объекты являются сериализуемыми, а какие нет. Когда я только начинал экспериментировать, я попытался сохранить в словаре *State* объекты *SolidColorBrush*. Приложение сформировало исключение «Type 'System.Windows.Media.Transform' cannot be serialized»[1]. Мне потребовалось некоторое время, чтобы вспомнить, что у класса *Brush* есть свойство *Transform* (Преобразование) типа *Transform*, который является абстрактным классом. Вместо этого мне пришлось сериализовать *Color*.

Скорректируем предыдущее приложение так, чтобы *SecondPage* использовал свойство *State*. Проект SilverlightRetainData остается практически неизменным, только добавляется директива для пространства имен *Microsoft.Phone.Shell* и два перегруженных метода в *SecondPage*:

Проект Silverlight: SilverlightRetainData Файл: SecondPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    if (ContentPanel.Background is SolidColorBrush)
    {
        Color clr = (ContentPanel.Background as SolidColorBrush).Color;

        if (args.Content is MainPage)
            (args.Content as MainPage).ReturnedColor = clr;

        // Сохраняем значение цвета
        PhoneApplicationService.Current.State["Color"] = clr;
    }

    base.OnNavigatedFrom(args);
}

protected override void OnNavigatedTo(NavigationEventArgs args)
{
    // Извлекаем значение цвета
    if (PhoneApplicationService.Current.State.ContainsKey("Color"))
    {
        Color clr = (Color)PhoneApplicationService.Current.State["Color"];
        ContentPanel.Background = new SolidColorBrush(clr);
    }

    base.OnNavigatedTo(args);
}
```

Если время вызова метода *OnNavigatedFrom* имеется доступный действительный объект *Color*, он сохраняется в словаре *State* с ключом «Color»:

```
PhoneApplicationService.Current.State["Color"] = clr;
```

При выполнении перегруженного *OnNavigatedTo*, если ключ существует, значение *Color* загружается из словаря, и из *Color* формируется *SolidColorBrush*. Ключ не будет существовать, если приложение только что начало выполнение и переход на *SecondPage* выполняется впервые. Но при последующих переходах на *SecondPage* страница будет создаваться с использованием значения цвета, заданного при предыдущем ее посещении.

---

[1] Тип 'System.Windows.Media.Transform' не может быть сериализован (*прим. переводчика*).

При каждом выходе из приложения через нажатие кнопки Back при открытой главной странице словарь *State* уничтожается вместе со всем *PhoneApplicationService*. Словарь *State* подходит только для хранения промежуточных данных, используемых приложением во время выполнения. Если требуется сохранять данные между запусками приложения, используйте изолированное хранилище.

Теперь попробуем такой сценарий. Перейдем к *SecondPage*. Коснемся экрана, чтобы изменить его цвет. Теперь нажмем кнопку телефона Start. Мы покинули приложение SilverlightRetainData и вернулись на стартовый экран телефона, с которого можно переходить к любым другим приложениям. Но если вдруг мы решим нажать кнопку телефона Back и вернуться к приложению SilverlightRetainData и его *SecondPage*, цвет остался неизменным с момента, когда мы вышли из приложения.

Теперь вернемся на *MainPage*. Цвет главной страницы аналогичен цвету *SecondPage*. Находясь на *MainPage*, нажмем кнопку телефона Start и покинем приложение. Можем немного «побродить» по другим приложениям, но после этого нажимаем кнопку Back необходимое число раз, чтобы вернуться к SilverlightRetainData и *MainPage*.

Вот чудеса! Экран изменил цвет! Что же произошло?

Идеал многозадачности

Основной мечтой нескольких последних десятилетий было научить персональные компьютеры выполнять несколько задач одновременно. Но когда дело доходит до пользовательских интерфейсов, многозадачность становится еще более проблематичной. Резидентные программы (Terminate-and-Stay-Resident, TSR) MS-DOS и кооперативная многозадачность ранней Windows были лишь первыми робкими попытками в непрекращающейся до сих пор борьбе. Теоретически, переключать процессы легко. Но организовать совместное использование ресурсов – включая экран и целый ряд различных устройств ввода – очень сложно.

Тогда как среднестатистический пользователь, возможно, восхищается тем, с какой легкостью современная Windows может «жонглировать» множеством разных приложений одновременно, для нас, разработчиков, реализация многозадачности до сих пор является довольно сложной проблемой. Мы тщательно разрабатываем UI-потоки, чтобы обеспечить их мирное взаимодействие с потоками, не являющимися UI-потоками, всегда оставаясь настороже, ожидая скрытого вероломства асинхронных операций.

Каждый новый программный интерфейс приходится каким-то образом, часто довольно неуклюже, приспосабливать к идеям многозадачности. По мере того, как мы привыкаем к этому API, мы привыкаем и к этим неуклюжим приемам. В конце концов, нам даже начинает казаться, что эти приемы и являются правильным решением проблемы.

В Windows Phone 7 таким неуклюжим приемом является *захоронение*.

Переключение задач в телефоне

Мы хотим, чтобы наши телефоны были почти такими же, как компьютеры. Мы хотим иметь доступ к множеству приложений. Мы хотим запускать то или иное приложение по первой необходимости, и чтобы это приложение выполнялось максимально быстро и предоставляло доступ к неограниченным ресурсам. Но также нам бы хотелось, чтобы это приложение сосуществовало с другими выполняющимися на устройстве приложениями, поскольку мы хотим иметь возможность переключаться между множеством приложений.

Возможность переключения между множеством выполняющихся приложений на телефоне практически нецелесообразна. Для этого потребовалось бы определенного рода окно, показывающее все выполняющиеся в данный момент приложения, такое как панель задач Windows. Эта панель задач должна была бы постоянно оставаться на экране, отнимая столь ценное пространство у активных приложений, либо потребовалась бы специальная кнопка или команда для выведения на экран панели или списка задач.

Вместо этого Windows Phone 7 организовывает управление множеством активных приложений через реализацию стека. В некотором смысле этот стек приложения расширяет стек страницы на все Silverlight-приложение. Телефон можно рассматривать как устаревший Веб-браузер без вкладок и кнопки Forward (Вперед). Но в нем есть кнопка Back и кнопка Start, обеспечивающая переход к экрану запуска, с которого можно запустить новое приложение.

Предположим, вы запустили приложение Analyze (Анализ). Немного поработали и решили выйти из него. Нажимаете кнопку Back. Приложение Analyze завершается, и вы возвращаетесь к экрану запуска. Это самый простой сценарий.

Позднее вы решаете запустить Analyze снова. В ходе работы с Analyze возникает необходимость проверить что-то в Интернете. Вы нажимаете кнопку Start, чтобы вернуться к экрану запуска и выбрать Internet Explorer. Во время «путешествия» по Интернету вы вдруг вспоминаете, что давно ни во что не играли. Нажимаете кнопку Start и выбираете Backgammon (Нарды). Пока вы обдумываете преимущества конкретного хода, вы снова нажимаете кнопку Start и запускаете калькулятор. Через некоторое время вас начинает мучить совесть, из-за того что вы бездельничаете, поэтому вы нажимаете кнопку Start и запускаете Draft (Проект).

Draft – это многостраничное приложение на Silverlight. С главной страницы вы переходите к другим страницам.

Теперь начинаем нажимать кнопку Back. Последовательно закрываются все страницы, находящиеся в стеке приложения Draft. Draft завершается, и вы переходите к калькулятору. В нем до сих пор отображаются какие-то результаты проводимых вами вычислений. Закрываете калькулятор и переходите к Backgammon. Игра находится в том состоянии, в котором вы ее оставили. Backgammon закрывается, и вы возвращаетесь в Internet Explorer. Опять последовательно закрываете все открытые Веб-страницы, завершаете IE. Возвращаетесь в Analyze, закрываете Analyze и опять оказываетесь на странице запуска. Теперь стек пуст.

Такой тип навигации является хорошим компромиссом для небольших устройств. Также он соответствует характеру взаимодействия пользователя с Веб-браузером. Концепция стека очень проста: по нажатию кнопки Start текущее приложение помещается в стек, чтобы дать возможность выполняться новому приложению; по нажатию кнопки Back текущее приложение завершается, а из стека извлекается то, которое было отправлено туда перед этим, т.е. самое верхнее.

Однако ограниченные ресурсы телефона требуют того, чтобы находящиеся в стеке приложения потребляли как можно меньше ресурсов. Поэтому приложение, помещенное в стек, не продолжает работать. Оно даже не приостанавливается. Происходит нечто более жесткое. Процесс фактически завершается. И когда это приложение извлекается из стека, оно начинает выполнение с нуля.

В этом и есть суть захоронения. Приложение «убивают», но потом позволяют ему возродиться вновь.

Вероятно, вы видели в фильмах, что воскрешение умерших может приводить к жутким результатам. Практически всегда то мерзкое нечто, которое восстает из могилы, совсем не похоже на чистый и светлый образ, каким он был до ухода.

Фокус в том, чтобы заставить эксгумированное приложение выглядеть и вести себя так же, как это было в момент перед его захоронением. Это можно обеспечить только во взаимодействии с Windows Phone 7. Телефон предоставляет инструменты (события и место для размещения некоторых данных); задача разработчика – использовать эти инструменты для восстановления приложения в презентабельное состояние. В идеале, пользователь не должен даже догадываться, что это совершенно новый процесс.

Для некоторых приложений не требуется, чтобы эксгумация была на 100% успешной. Мы все имеем опыт работы с множеством Веб-страниц и знаем, что приемлемо и что нет. Например, предположим, что при просмотре большой Веб-страницы вы прокрутили ее вниз и перешли к другой странице. При возвращении на исходную страницу нет ничего страшного в том, если она будет отображаться в исходном состоянии, а не в том месте, куда вы перешли при прокручивании.

Но в то же время, если вы потратили 10 минут на заполнение огромной формы, то абсолютно точно *не* хотите опять увидеть пустую форму после того, как другая страница сообщит о какой-то мелкой ошибке, сделанной вами при заполнении.

Теперь определимся с терминологией, которой будем пользоваться в дальнейшем:

- Если приложение вызывается с экрана запуска, мы говорим, что оно *запущено*.
- Если приложение завершает свою работу в результате нажатия кнопки Back, оно *завершается*.
- Если приложение выполняется, и пользователь нажимает кнопку Start, говорят, что приложение *деактивировано*, хотя фактически оно «мертво». Это состояние захоронения.
- Когда пользователь возвращается к приложению, и оно выходит из состояния захоронения, говорят, что приложение *активировано*, даже несмотря на то, что на самом деле оно запускается с нуля.

Состояние страницы

Проект *SilverlightFlawedTombstoning* – это простое одностраничное приложение на Silverlight, которое отвечает на касания экрана изменением цвета фона *ContentGrid* случайным образом и выводит в заголовке страницы общее число касаний. Все самое интересное происходит в файле выделенного кода:

Проект Silverlight: SilverlightFlawedTombstoning Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    int numTaps = 0;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
```

```
    {
        ContentPanel.Background =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }
}
```

Небольшой метод *UpdatePageTitle* (Обновить заголовок страницы) вызывается из конструктора приложения (в результате чего на экран всегда выводится 0) и из перегруженного *OnManipulationStarted*.

Выполняем сборку приложения и развертываем его на телефоне или эмуляторе телефона нажатием клавиши F5 (или выбирая Start Debugging в меню Debug). Настраиваем Visual Studio так, чтобы видеть окно Output (Вывод). Когда приложение запущено, несколько раз касаемся экрана, чтобы изменить цвет и увеличить счет касаний. Теперь нажимаем кнопку телефона Start. В Visual Studio можно увидеть, что два потока приложения завершены и приложение завершено, но фактически для телефона оно деактивировано и захоронено.

Теперь нажимаем кнопку Back, чтобы вернуться в приложение. Появится пустой экран со словом «Resuming…» (Возобновление…) и в Visual Studio будет выведено окно Output с загружаемыми библиотеками. Это приложение возвращается к жизни.

Однако когда приложение вновь появляется на экране, цвет и количество касаний утрачены. Все что нажито непосильным трудом! Все пропало! Не годится приложению так воскресать после захоронения. Данные состояния должны сохраняться. (Теперь можно понять, почему описанный после рассмотрения приложения SilverlightInsertData подходит не для всех случаев. В той схеме предполагается сохранение экземпляра *SecondPage* при переходе *MainPage* на эту страницу. Но если пользователь с *SecondPage* переходит на стартовую страницу и затем возвращается, он переходит не в тот экземпляр, который был сохранен *FrontPage* (Титульная страница), а в совершенно новый.)

Перегруженные методы *OnNavigatedTo* и *OnNavigatedFrom* класса *Page*, от которого наследуется *PhoneApplicationPage*, предоставляют замечательную возможность сохранять и повторно загружать данные состояния страницы. Эти методы вызываются, когда страница выводится на экран в результате загрузки рамкой, и когда страница выгружается из рамки.

Эти методы особенно удобны для многостраничных приложений на Silverlight. Каждый раз, когда пользователь переходит к странице, создается новый экземпляр *PhoneApplicationPage*. Чтобы обеспечить нормальные переходы со страницы на страницу, необходимо сохранять и повторно загружать данные ее состояния. Перегрузка *OnNavigatedTo* и *OnNavigatedFrom* эффективно решает две проблемы.

Windows Phone 7 перекладывает большую часть ответственности за восстановление приложения после захоронения на само приложение, но, тем не менее, при активации загружает нужную страницу. Поэтому, вероятно, страничным приложениям на Silverlight, сохраняющим и восстанавливающим данные состояния в свойстве *State* класса *PhoneApplicationSerivce* в ходе выполнения методов *OnNavigatedTo* и *OnNavigatedFrom*, не понадобится специально обрабатывать захоронение. Операционная система телефона

сохраняет значение этого свойства *State* пока приложение деактивировано и захоронено, но удаляет его, если приложение действительно закрывается и завершается.

Файл выделенного кода SilverlightBetterTombstoning включает директиву *using* для *Microsoft.Phone.Shell* и использует словарь *State*. Рассмотрим класс полностью:

Проект Silverlight: SilverlightBetterTombstoning Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    int numTaps = 0;
    PhoneApplicationService appService = PhoneApplicationService.Current;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        ContentPanel.Background =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        appService.State["numTaps"] = numTaps;

        if (ContentPanel.Background is SolidColorBrush)
        {
            appService.State["backgroundColor"] =
                                    (ContentPanel.Background as SolidColorBrush).Color;
        }

        base.OnNavigatedFrom(args);
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Загружаем numTaps
        if (appService.State.ContainsKey("numTaps"))
        {
            numTaps = (int)appService.State["numTaps"];
            UpdatePageTitle(numTaps);
        }

        // Загружаем цвет фона
        object obj;

        if (appService.State.TryGetValue("backgroundColor", out obj))
            ContentPanel.Background = new SolidColorBrush((Color)obj);

        base.OnNavigatedTo(args);
```

```
    }
}
```

Обратите внимание, полю *appService* присваивается *PhoneApplicationService.Current*. Это делается просто для удобства доступа к свойству *State*, но можно использовать и длинную ссылку *PhoneApplicationService.Current.State*.

Сохранить элементы в словарь *State* проще, чем извлечь их оттуда. Такое выражение:

```
appService.State["numTaps"] = numTaps;
```

обеспечивает замещение существующего элемента, если ключ «numTaps» существует, или добавляет новый элемент, если ключа нет. Сохранить цвет фона несколько сложнее. По умолчанию свойство *Background* объекта *ContentPanel* имеет значение *null*, поэтому перед попыткой сохранить свойство *Color* выполняется проверка того, что это значение не *null*.

Для извлечения элементов из словаря этот синтаксис не подойдет. В случае несуществования ключей будет сформировано исключение. (И эти ключи *не* будут существовать при запуске приложения.) Метод *OnNavigatedTo* представляет два разных стандартных способа доступа к элементам. В первом случае проверяется наличие ключа в словаре; во втором используется метод *TryGetValue* (Попытаться получить значение), который возвращает *true*, если ключ существует.

В реальном приложении, вероятно, в качестве ключей будут использоваться *строковые* переменные, что позволит избежать нечаянного введения несовместимых значений. (Если вы печатаете безупречно, не волнуйтесь о том, что множество идентичных строк заполонят хранилище: строки изолируются, и идентичные строки консолидируются в одну.) Также, скорее всего, для выполнения этих операций вы решите создать стандартные процедуры.

Попробуйте выполнить это приложение так же, как предыдущее: нажмите F5, чтобы развернуть его на телефоне или эмуляторе телефона из Visual Studio. Несколько раз коснитесь экрана. Нажмите кнопку Start, как будто хотите запустить новое приложение. Visual Studio сообщит о завершении процесса. Теперь нажимайте кнопку Back. Приложение возобновляет выполнение с сохраненными настройками, так что «труп» выглядит просто, как новенький!

Обратите внимание, что настройки сохраняются, когда происходит захоронение приложения (т.е. когда пользователь покидает приложение по кнопке Start и затем возвращается), но не когда новый экземпляр запускается из списка запуска. Такое поведения является правильным. Операционная система удаляет словарь *State*, когда приложение действительно завершается. Словарь State предназначен только для хранения промежуточных данных и не годится для данных, используемых другими экземплярами приложения.

Если определенные данные должны использоваться всеми экземплярами приложения, необходимо реализовать *параметры приложения*. Вы тоже можете это сделать.

Изолированное хранилище

Для каждого приложения, установленного на устройстве Windows Phone 7, выделяется собственная постоянная область памяти на диске, которую называют *изолированным хранилищем*. Приложение работает с этой областью памяти с помощью классов пространства имен *System.IO.IsolatedStorage*. В изолированное хранилище могут помещаться и извлекаться целые файлы, и в приложении, завершающем данную главу, я покажу, как это делается. Сейчас же остановимся на специальном применении изолированного хранилища

для хранения параметров приложения. Для этой цели существует класс *IsolatedStorageSettings* (Параметры изолированного хранилища).

Параметры приложения применяются к приложению в целом, а не только к отдельной странице. Некоторые параметры приложения, возможно, могут применяться к множеству страниц. Поэтому подходящим местом для работы с этими параметрами является класс *App*.

Объект *PhoneApplicationService* (такой же объект *PhoneApplicationService* использовался для хранения промежуточных данных) создается в файле App.xaml и задает обработчики для четырех событий:

```
<shell:PhoneApplicationService Launching="Application_Launching"
                               Closing="Application_Closing"
                               Activated="Application_Activated"
                               Deactivated="Application_Deactivated"/>
```

Событие *Launching* (Запуск) формируется при первом запуске приложения с экрана запуска. Событие *Deactivated* (Деактивирован) происходит при захоронении приложения. И событие *Activated* (Активирован) возникает при воскрешении приложения после захоронения. Событие *Closing* (Закрывается) формируется, когда приложение на самом деле завершается, возможно, по нажатию кнопки Back пользователем.

Итак, когда приложение запускается, возникает либо событие *Launching*, либо *Activated* (но никогда оба) в зависимости от того, было ли оно запущено с экрана запуска или восстановлено после захоронения. По завершении приложения формируется событие *Deactivated* или *Closing* в зависимости от того, произошло ли захоронение приложения или его действительное завершение.

Приложение должно загружать параметры приложения во время события *Launching* и сохранять их в ответ на событие *Closing*. Это очевидно. Но также параметры приложения должны сохраняться во время события *Deactivated*, потому что приложение не знает, будет ли оно когда-нибудь впоследствии восстановлено. При воскрешении приложение должно загрузить параметры приложения во время события *Activated*, потому что в противном случае оно не будет знать об этих настройках.

Вывод: параметры приложения должны загружаться во время событий *Launching* и *Activated* и сохраняться во время *Deactivated* и *Closing*.

Для приложения SilverlightIsolatedStorage я решил, что количество касаний должно сохраняться как промежуточные данные, т.е. как часть состояния страницы. Но цвет фона должен быть параметром приложения и быть общим для всех экземпляров.

В App.xaml.cs я описал следующее открытое свойство:

Проект Silverlight: SilverlightIsolatedStorage Файл: App.xaml.cs (фрагмент)

```
public partial class App : Application
{
    // Параметры приложения
    public Brush BackgroundBrush { set; get; }
    …
}
```

По-видимому, это может быть одним из многих параметров приложения, доступных во всем приложении.

App.xaml.cs уже имеет пустые обработчики событий для всех событий *PhoneApplicationService*. Для каждого обработчика описываем тело, состоящее из единственного вызова метода:

Проект Silverlight: SilverlightIsolatedStorage Файл: App.xaml.cs (фрагмент)

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    LoadSettings();
}

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    LoadSettings();
}

private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    SaveSettings();
}

private void Application_Closing(object sender, ClosingEventArgs e)
{
    SaveSettings();
}
```

Рассмотрим методы *LoadSettings* (Загрузить параметры) и *SaveSettings* (Сохранить параметры). Оба метода работают с объектом *IsolatedStorageSettings*. Как и свойство *State* класса *PhoneApplicationService*, объект *IsolatedStorageSettings* является словарем. Один метод загружает (и другой сохраняет) свойство *Color* свойства *BackgroundBrush*, используя для этого код, подобный рассмотренному нами ранее:

Проект Silverlight: SilverlightIsolatedStorage Файл: App.xaml.cs (фрагмент)

```
void LoadSettings()
{
    IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

    Color clr;

    if (settings.TryGetValue<Color>("backgroundColor", out clr))
        BackgroundBrush = new SolidColorBrush(clr);
}

void SaveSettings()
{
    IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

    if (BackgroundBrush is SolidColorBrush)
    {
        settings["backgroundColor"] = (BackgroundBrush as SolidColorBrush).Color;
        settings.Save();
    }
}
```

И, наконец, рассмотрим новый файл MainPage.xaml.cs. Этот файл, и любой другой класс в приложении, может получить доступ к объекту *App*, приводя статическое свойство *Application.Current* к *App*. Конструктор *MainPage* получает значение свойства *BackgroundBrush* из класса *App*, и метод *OnManipulationStarted* задает это свойство *BackgroundBrush*.

Проект Silverlight: SilverlightIsolatedStorage Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
```

```
    int numTaps = 0;
    PhoneApplicationService appService = PhoneApplicationService.Current;

    public MainPage()
    {
        InitializeComponent();
        UpdatePageTitle(numTaps);

        // Выполняем доступ к классу App для получения параметров из изолированного
хранилища
        Brush brush = (Application.Current as App).BackgroundBrush;

        if (brush != null)
            ContentPanel.Background = brush;
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        SolidColorBrush brush =
            new SolidColorBrush(Color.FromArgb(255, (byte)rand.Next(256),
                                                    (byte)rand.Next(256),
                                                    (byte)rand.Next(256)));
        ContentPanel.Background = brush;

        // Выполняем доступ к классу App для сохранения параметров в изолированном
хранилище
        (Application.Current as App).BackgroundBrush = brush;

        UpdatePageTitle(++numTaps);

        args.Complete();
        base.OnManipulationStarted(args);
    }

    void UpdatePageTitle(int numTaps)
    {
        PageTitle.Text = String.Format("{0} taps total", numTaps);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        appService.State["numTaps"] = numTaps;

        base.OnNavigatedFrom(args);
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Загружаем numTaps
        if (appService.State.ContainsKey("numTaps"))
        {
            numTaps = (int)appService.State["numTaps"];
            UpdatePageTitle(numTaps);
        }
    }
}
```

Поскольку цвет фона был переведен из промежуточных данных страницы в параметры приложения, ссылки на него удалены из перегруженных методов *OnNavigatedFrom* и *OnNavigatedTo*.

Захоронение и параметры для приложений на XNA

Как правило, приложения на XNA не строятся вокруг страниц, как приложения на Silverlight. Однако если это требуется, безусловно, в рамках приложения на XNA тоже можно

реализовать собственную страничную структуру. Вспомним, что состояние кнопки телефона Back проверяется при каждом вызове стандартного метода *Update*. Эта логика может использоваться в целях навигации, а также для завершения приложения. Но эту задачу я оставляю для самостоятельной проработки.

Приложение на XNA может использовать тот же класс *PhoneApplicationService*, с которым работали приложения на Silverlight, для хранения промежуточных данных состояния во время захоронения. Также этот класс может применяться приложением на XNA для установки обработчиков четырех событий *PhoneApplicationService*: *Launching*, *Activated*, *Deactivated* и *Closing*. Для этого понадобиться указать ссылки и на библиотеку Microsoft.Phone (для самого *PhoneApplicationService*), и на System.Windows (для интерфейса *IApplicationService*, реализуемого *PhoneApplicationService*). В файле Game1.cs необходимо подключить пространство имен *Microsoft.Phone.Shell* посредством директивы *using*.

В конструкторе класса *Game1* с помощью статического свойства *PhoneApplicationService.Current* можно получить ассоциированный с приложением экземпляр *PhoneApplicationService*.

Также класс *Game* описывает пару удобных и полезных для обработки захоронения виртуальных методов: *OnActivated* (При активации) и *OnDeactivated* (При деактивации). Метод *OnActivated* вызывается при запуске и повторной активации, метод *OnDeactivated* вызывается при деактивации и завершении приложения, во многом аналогично виртуальным методам *OnNavigatedTo* и *OnNavigatedFrom* страницы на Silverlight.

В приложении XnaTombstoning, которое завершает данную главу, я попытался повторить функциональность и структуру приложения SilverlightIsolatedStorage. XnaTombstoning использует события *PhoneApplicationService* для сохранения и восстановления параметров приложения (*Color*) и переопределяет события *OnDeactivated* и *OnActivated* для сохранения промежуточных данных (количества касаний).

Но я пошел несколько дальше и реализовал более обобщенное решение для параметров приложения. Для проекта *XnaTombstoning* был создан выделенный класс *Settings*, использующий обобщенные возможности изолированного хранилища, которые обеспечивают работу с реальными файлами, а не просто с параметрами. Нам понадобится ссылка на библиотеку System.Xml.Serialization для этого класса, а также директивы *using* для пространств имен *System.IO*, *System.IO.IsolatedStorage* и *System.Xml.Serialization*.

Проект Silverlight: XnaTombstoning Файл: Settings.cs (фрагмент)

```
public class Settings
{
    const string filename = "settings.xml";

    // Параметры приложения
    public Color BackgroundColor { set; get; }

    public Settings()
    {
        BackgroundColor = Color.Navy;
    }

    public void Save()
    {
        IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication();
        IsolatedStorageFileStream stream = storage.CreateFile(filename);
        XmlSerializer xml = new XmlSerializer(GetType());
        xml.Serialize(stream, this);
        stream.Close();
```

```
        stream.Dispose();
    }

    public static Settings Load()
    {
        IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication();
        Settings settings;

        if (storage.FileExists(filename))
        {
            IsolatedStorageFileStream stream =
                    storage.OpenFile("settings.xml", FileMode.Open);
            XmlSerializer xml = new XmlSerializer(typeof(Settings));
            settings = xml.Deserialize(stream) as Settings;
            stream.Close();
            stream.Dispose();
        }
        else
        {
            settings = new Settings();
        }

        return settings;
    }
}
```

Основная идея здесь в том, что в ходе выполнения метода *Save* в изолированном хранилище сериализуется и сохраняется сам экземпляр класса *Settings*. Затем в методе *Load* выполняется его извлечение и десериализация. Обратите внимание, что метод *Load* статический и возвращает экземпляр класса *Settings*.

При сериализации класса *Settings* сериализуются все его открытые свойства. У этого класса всего одно открытое свойство *BackgroundColor* типа *Color*, но не составит никакого труда добавить в него больше свойств в ходе доработки и усложнения приложения.

В методе *Save* посредством статического метода *IsolatedStorageFile.GetUserStoreForApplication* получаем область изолированного хранилища, которая зарезервирована для данного приложения. Этот метод возвращает объект типа *IsolatedStorageFile* (Файл изолированного хранилища), но имя несколько не соответствует сути. Функциональность объекта *IsolatedStorageFile* больше соответствует *файловой системе*, а не *файлу*. Этот объект используется для хранения каталогов, создания и открытия файлов. Вызов *CreateFile* (Создать файл) возвращает *IsolatedStorageFileStream* (Файловый поток изолированного хранилища), который в данном примере используется с объектом *XmlSerializer* (Модуль сериализации XML) для сериализации и сохранения файла.

Метод *Load* несколько сложнее, поскольку существует вероятность того, что приложение выполняется впервые и файл settings.xml не существует. В этом случае *Load* создает новый экземпляр *Settings*.

Обратите внимание на конструктор, который инициализирует свойства значениями по умолчанию. В данном случае это касается только открытого свойства *BackgroundColor*. Если в какой-то момент добавить второе открытое свойство для другого параметра приложения, в конструкторе необходимо будет задать для него значение по умолчанию. Это новое свойство будет инициализировано в конструкторе при первом выполнении новой версии приложения, но метод *Load* извлечет файл, не имеющий этого свойства, таким образом, новая версия плавно интегрируется с предыдущей.

Еще одно замечание: данная схема подходит, только если свойства, представляющие параметры приложения, являются сериализуемыми. В более сложном приложении это

условие может не выполняться. Для несериализуемых объектов, которые, тем не менее, должны быть сохранены в изолированное хранилище, в этот файл можно включить свойство, но его описание необходимо обозначить атрибутом *[XmlIgnore]*. Благодаря этому данное свойство будет игнорироваться при сериализации, но для его обработки в методах *Save* и *Load* придется предусмотреть специальный код.

Остальной код проекта XnaTombstoning обрабатывает функциональность касания экрана и ответ на них в виде изменения цвета фона случайным образом, а также подсчет количества касаний. Цвет фона рассматривается как параметр приложения (что очевидно из его включения в класс *Settings*), и количество касаний является промежуточным параметром.

Рассмотрим фрагмент класса *Game1*, включающий поля, конструктор и события *PhoneApplicationService*:

Проект Silverlight: XnaTombstoning Файл: Game1.cs (фрагмент)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Settings settings;
    SpriteFont segoe14;
    Viewport viewport;
    Random rand = new Random();
    StringBuilder text = new StringBuilder();
    Vector2 position;
    int numTaps = 0;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для Windows Phone составляет 30 кадр/с.
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        TouchPanel.EnabledGestures = GestureType.Tap;

        PhoneApplicationService appService = PhoneApplicationService.Current;
        appService.Launching += OnAppServiceLaunching;
        appService.Activated += OnAppServiceActivated;
        appService.Deactivated += OnAppServiceDeactivated;
        appService.Closing += OnAppServiceClosing;
    }

    …

    void OnAppServiceLaunching(object sender, LaunchingEventArgs args)
    {
        settings = Settings.Load();
    }

    void OnAppServiceActivated(object sender, ActivatedEventArgs args)
    {
        settings = Settings.Load();
    }

    void OnAppServiceDeactivated(object sender, DeactivatedEventArgs args)
    {
        settings.Save();
    }

    void OnAppServiceClosing(object sender, ClosingEventArgs args)
```

```
    {
        settings.Save();
    }
}
```

Объект *Settings* под именем *settings* сохраняется как поле. Конструктор подключает обработчики для четырех событий класса *PhoneApplicationService*, и параметры приложения сохраняются и загружаются именно в обработчиках этих событий.

Перегруженный метод *LoadContent* не несет в себе никаких сюрпризов:

Проект Silverlight: XnaTombstoning Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    viewport = this.GraphicsDevice.Viewport;
}
```

Метод *Update* фиксирует касания, обновляет поле *numTaps* (Количество касаний), производит выбор нового цвета случайным образом и также подготавливает объект *StringBuilder* к отображению числа касаний:

Проект Silverlight: XnaTombstoning Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем выход из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
        if (TouchPanel.ReadGesture().GestureType == GestureType.Tap)
        {
            numTaps++;
            settings.BackgroundColor =  new Color((byte)rand.Next(255),
                                                  (byte)rand.Next(255),
                                                  (byte)rand.Next(255));
        }

    text.Remove(0, text.Length);
    text.AppendFormat("{0} taps total", numTaps);
    Vector2 textSize = segoe14.MeasureString(text.ToString());
    position = new Vector2((viewport.Width - textSize.X) / 2,
                           (viewport.Height - textSize.Y) / 2);

    base.Update(gameTime);
}
```

Обратите внимание, что новый цвет сохраняется не как поле, а как свойство *BackgroundColor* экземпляра *Settings*. Затем это свойство используется в перегруженном методе *Draw*:

Проект Silverlight: XnaTombstoning Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(settings.BackgroundColor);

    spriteBatch.Begin();
```

```
    spriteBatch.DrawString(segoe14, text, position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Промежуточное значение поля *numTaps* сохраняется и восстанавливается из словаря *State* объекта *PhoneApplicationService* в перегруженных методах *OnActivated* и *OnDeactivated*:

Проект Silverlight: XnaTombstoning Файл: Game1.cs (фрагмент)

```
protected override void OnActivated(object sender, EventArgs args)
{
    if (PhoneApplicationService.Current.State.ContainsKey("numTaps"))
        numTaps = (int)PhoneApplicationService.Current.State["numTaps"];

    base.OnActivated(sender, args);
}

protected override void OnDeactivated(object sender, EventArgs args)
{
    PhoneApplicationService.Current.State["numTaps"] = numTaps;
    base.OnDeactivated(sender, args);
}
```

Решение сохранять и восстанавливать параметры приложения в одном наборе обработчиков событий, а промежуточные параметры – в другом наборе перегруженных виртуальных методов, довольно произвольно. Метод *OnActivated* будет вызываться в приложении практически одновременно с формированием событий *Launching* и *Activated*; и метод *OnDeactivated* – одновременно с формированием событий *Deactivated* и *Closing*. Более концептуальное различие в том, что *OnActivated* и *OnDeactivated* ассоциированы с экземпляром *Game*, поэтому должны использоваться для свойств, связанных с игрой, а не для параметров приложения в целом.

Возможно, возникнет необходимость сохранять как промежуточный параметр несериализуемый объект. Однако поскольку он не является сериализуемым, для его хранения не может использоваться словарь *State* класса *PhoneApplicationService*, а понадобится изолированное хранилище. При этом необходимо гарантированно обеспечить, что этот объект не будет случайно извлечен и повторно использован при очередном выполнении приложения. На этот случай в словаре *State* предусмотрен флаг, указывающий на необходимость загрузки промежуточного объекта из изолированного хранилища.

Тестирование и эксперименты

Разработчики Майкрософт, которые занимаются созданием приложений для Windows Phone 7 намного дольше, чем многие из нас, говорят, что реализация захоронения является, пожалуй, одним из самых сложных аспектов разработки для телефона. Представленные мною в данной главе методики хороши как исходные приемы, но требования всех приложений немного отличаются. Несомненно, желательно реализовывать максимальный объем тестирования в собственных приложениях, всегда полезно точно знать, какие методы приложения и в каком порядке вызываются. Для этого очень пригодится метод *Debug.WriteLine* из пространства имен *System.Diagnostics*.

Часть II
Silverlight

Глава 7
Мощь и слабость XAML

Как мы видели, приложение на Silverlight – это, главным образом, смесь кода и XAML. В XAML обычно описывается компоновка визуальных элементов приложения и в коде выполняется обработка событий, включая все события пользовательского ввода и события, формируемые элементами управления в результате обработки событий пользовательского ввода.

Основной объем работ по созданию и инициализации объектов в XAML традиционно выполняется конструктором страницы или класса окна. Из-за этого может создаться впечатление, что XAML играет совершенно незначительную роль в приложении, но на самом деле это абсолютно не так. Как следует из названия, XAML полностью совместим с XML, т.е. его можно создавать как вручную, так и автоматически с помощью любых доступных инструментов.

XAML обычно занимается аспектами создания и инициализации объектов, но некоторые функции Silverlight выходят далеко за рамки просто инициализации. Одна из таких возможностей – привязка данных. Она заключается в установлении взаимосвязи между элементами управления или между элементами управления и данными, благодаря чему обеспечивается автоматическое обновление свойств без необходимости реализации обработчиков событий. Вся анимация также может быть описана в XAML.

Несмотря на то что XAML называют «декларативным языком», он, безусловно, не является полноценным языком программирования. XAML не позволяет выполнять арифметические действия и динамически создавать объекты.

Опытные разработчики, впервые сталкиваясь с XAML, подчас не приемлют его. Я испытал это на собственном опыте. Все, что мы ценим и любим в таких языках программирования, как C#: обязательные объявления, строгая типизация, контроль выхода за границы массива, возможности трассировки для отладки – теряет смысл, когда все сводится к текстовым строкам XML. Однако с годами я очень неплохо приспособился к XAML и даже наслаждаюсь той свободой, которую он обеспечивает при работе с визуальными элементами приложения. В частности, мне нравится, как наследственные отношения элементов управления воспроизведены в характерной для XML структуре родитель-потомок, а также возможность экспериментировать с XAML, даже просто в дизайнере Visual Studio.

Все задачи, реализуемые в Silverlight, можно распределить по трем категориям:

- То, что можно сделать либо в коде, либо в XAML
- То, что можно сделать только в коде (например, обработка событий и реализация методов)
- То, что можно сделать только в XAML (например, шаблоны)

И в коде, и в XAML могут создаваться экземпляры классов и структур и задаваться свойства этих объектов. Класс или структура, экземпляр которой создается в XAML, должна быть описана как (безусловно) открытая (public) и иметь конструктор без параметров. При создании экземпляра класса XAML не располагает никакими возможностями передачи каких-либо данных в его конструктор. В XAML определенное событие может быть ассоциировано с обработчиком события, но сам обработчик события должен быть реализован в коде. В XAML нельзя вызвать метод, потому что, опять же, нет никаких способов передачи аргументов.

Практически все приложение на Silverlight может быть реализовано в коде. Однако навигация по страницам строится на базе XAML-файлов классов, производных от *PhoneApplicationPage*. Есть еще одна важная задача, которая *должна* реализовываться в XAML: создание *шаблонов*. Существует два применения шаблонам: для визуального представления данных с использованием коллекций элементов и элементов управления, а также для переопределения внешнего вида элемента управления с сохранением его функциональности. В коде можно создавать альтернативы шаблонам, но не сами шаблоны.

У тех, кто уже немного поработал с Silverlight и накопил небольшой опыт разработки приложений на Silverlight, может возникнуть желание использовать программу проектирования, такую как Expression Blend, которая будет автоматически формировать XAML. Я хочу дать совет, как программист программисту: *научитесь писать XAML самостоятельно*. Или, как минимум, вы должны уметь читать автоматически сформированный XAML.

Одна из самых замечательных особенностей XAML в том, что он позволяет экспериментировать с ним в интерактивном режиме, и, экспериментируя с XAML, можно многое узнать о Silverlight. Существуют инструменты разработки, специально созданные для таких экспериментов с XAML. В этих приложениях используется статический метод *XamlReader.Load*, который может преобразовывать текст XAML в объект во время выполнения.

В главе 13 будет представлено приложение, с помощью которого можно экспериментировать с XAML прямо на телефоне, до тех пор можете делать это в дизайнере Visual Studio. Как правило, он быстро и точно реагирует на все изменения, вносимые в XAML. Однако в сложных случаях вам все-таки придется выполнить сборку и развертывание приложения, чтобы увидеть, что происходит.

TextBlock в коде

Прежде чем мы перейдем к экспериментам с XAML, я должен сделать еще одно важное предупреждение: осваивая XAML, не забудьте C#!

Вспомним XAML-версию *TextBlock* в *Grid* из главы 2:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Фактически, элементы в XAML (такие как *TextBlock*) – это классы. Атрибуты этих элементов (такие как *Text*, *HorizontalAlignment* и *VerticalAlignment*) – свойства класса. Давайте рассмотрим, насколько просто с помощью кода создать *TextBlock* и вставить *TextBlock* в *Grid*, описанный в XAML,.

В проекте TapForTextBlock (Касание для блока текста) реализована такая функциональность, что при каждом касании экрана в коде создается новый *TextBlock*. Файл MainPage.xaml включает TextBlock, центрированный по сетке для содержимого:

Проект Silverlight: TapForTextBlock Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Center"
```

```
                VerticalAlignment="Center" />
</Grid>
```

При каждом касании экрана в файле выделенного кода для *MainPage* создается дополнительный *TextBlock*. При этом для задания свойства *Margin* нового элемента *TextBlock*, чтобы разместить его случайным образом в сетке для содержимого, используются размеры существующего *TextBlock*:

Проект Silverlight: TapForTextBlock   Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();

    public MainPage()
    {
        InitializeComponent();
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        TextBlock newTextBlock = new TextBlock();
        newTextBlock.Text = "Hello, Windows Phone 7!";
        newTextBlock.HorizontalAlignment = HorizontalAlignment.Left;
        newTextBlock.VerticalAlignment = VerticalAlignment.Top;
        newTextBlock.Margin = new Thickness(
            (ContentPanel.ActualWidth - txtblk.ActualWidth) * rand.NextDouble(),
            (ContentPanel.ActualHeight - txtblk.ActualHeight) * rand.NextDouble(),
            0, 0);

        ContentPanel.Children.Add(newTextBlock);

        args.Complete();
        args.Handled = true;
        base.OnManipulationStarted(args);
    }
}
```

Нет необходимости строго придерживаться приведенной последовательности действий. Можно сначала добавить *TextBlock* в *ContentPanel* и только потом задавать свойства *TextBlock*.

Но это тот род вещей, которые просто невозможно реализовать в XAML. XAML не может отвечать на события, не может произвольным образом создавать новые экземпляры элементов, не может обращаться к классу *Random* (Случайный) и, безусловно, не может проводить вычисления.

Можно воспользоваться синтаксическим расширением, появившимся в C# 3.0, которое позволяет одновременно создать экземпляр класса и определить его свойства:

```
TextBlock newTextBlock = new TextBlock
{
    Text = "Hello, Windows Phone 7!",
    HorizontalAlignment = HorizontalAlignment.Left,
    VerticalAlignment = VerticalAlignment.Top,
    Margin = new Thickness(
            (ContentPanel.ActualWidth - txtblk.ActualWidth) * rand.NextDouble(),
            (ContentPanel.ActualHeight - txtblk.ActualHeight) * rand.NextDouble(),
            0, 0)
};

ContentPanel.Children.Add(newTextBlock);
```

Это делает код *немного* более похожим на XAML (за исключением вычислений и вызовов метода *rand.NextDouble*), но все равно можно заметить, что XAML обеспечивает более краткую запись. В коде в качестве значений свойств *HorizontalAlignment* и *VerticalAlignment* должны быть заданы члены перечислений *HorizontalAlignment* и *VerticalAlignment*, соответственно. В XAML требуется задать лишь имя члена.

Из XAML нельзя явно увидеть, что у *Grid* есть свойство *Children* (Потомки), что это свойство является коллекцией, и то что размещение *TextBlock* в *Grid* фактически добавляет *TextBlock* в коллекцию *Children*. В коде процесс добавления *TextBlock* в *Grid* должен быть более явным.

Наследование свойств

Чтобы поэкспериментировать с XAML, удобно создать проект специально для этого. Назовем этот проект XamlExperiment и разместим *TextBlock* в сетке для содержимого:

Проект Silverlight: XamlExperiment   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!" />
</Grid>
```

Текст отображается в верхнем левом углу клиентской области страницы. Сделаем его курсивом. Для этого зададим соответствующее значение свойству *FontStyle* нашего *TextBlock*:

```
<TextBlock Text="Hello, Windows Phone 7!"
           FontStyle="Italic" />
```

Или можно добавить атрибут *FontStyle* в тег *PhoneApplicationPage*:

```
<phone:PhoneApplicationPage … FontStyle="Italic" …
```

Этот атрибут *FontStyle* может располагаться в теге *PhoneApplicationPage* в любом месте. Обратите внимание, что если свойство задано в данном теге, его значение распространяется на *все* элементы *TextBlock* на странице. Это называется *наследованием свойств*. Значения определенных свойств – сюда входят лишь *Foreground* и свойства шрифтов *FontFamily*, *FontSize*, *FontStyle*, *FontWeight* (Насыщенность шрифта) и *FontStretch* (Ширина шрифта) – распространяются по всему дереву визуальных элементов. Именно благодаря этому значения свойств *FontFamily*, *FontSize* и *Foreground* (а теперь еще и *FontStyle*) для *TextBlock* могут задаваться в *PhoneApplicationPage*.

Наследование свойств можно проследить, начиная с объекта *PhoneApplicationPage*. *FontStyle* задается для *PhoneApplicationPage*, его значение наследуется самым внешним *Grid*, затем объектами, размещенными в *Grid*, и, наконец, *TextBlock*. Кажется, все замечательно, проблема лишь в том, что в *Grid* нет свойства *FontStyle*! Если задать *FontStyle* для элемента *Grid*, Visual Studio выдаст предупреждение. Наследование свойств – несколько более сложный вопрос, чем просто передача значений от родителя потомку, и это один из аспектов Silverlight, который тесно связан с ролью *свойств-зависимостей*, которые будут рассматриваться в главе 11.

Несмотря на то что свойству *FontStyle* задано значение *Italic* в теге *PhoneApplicationPage*, задаем *FontStyle* повторно прямо в *TextBlock*:

```
<TextBlock Text="Hello, Windows Phone 7!"
           FontStyle="Normal" />
```

Теперь текст в этом конкретном *TextBlock* будет отображаться обычным, не курсивным шрифтом. Как мы видим, значение *FontStyle*, заданное в *TextBlock* – которое называют

локальным значением или *локальным параметром* – имеет более высокий приоритет, чем унаследованное значение свойства. Если поразмыслить над этим логически, станет очевидным, что так оно и должно быть. При этом унаследованное значение и локальный параметр более приоритетны, чем значение по умолчанию. Эти отношения можно представить в виде простой схемы:

Локальные параметры имеют приоритет над

Унаследованными свойствами, которые являются более приоритетными, чем

Значения по умолчанию

Эта схема будет дополняться по мере того, как мы будем узнавать новые способы задания свойств.

Синтаксис свойство-элемент

Удалим все имеющиеся задания *FontStyle* и присвоим атрибутам *TextBlock* следующие значения:

```
<TextBlock Text="Hello, Windows Phone 7!"
           FontSize="36"
           Foreground="Red" />
```

Поскольку это XML, мы можем разделить тег *TextBlock* на открывающий и закрывающий теги, ничего не вставив между ними:

```
<TextBlock Text="Hello, Windows Phone 7!"
           FontSize="36"
           Foreground="Red">
</TextBlock>
```

Но можно сделать то, что на первый взгляд покажется странным. Удалим атрибут *FontSize* из открывающего тега и зададим его следующим образом:

```
<TextBlock Text="Hello, Windows Phone 7!"
           Foreground="Red">
    <TextBlock.FontSize>
        36
    </TextBlock.FontSize>
</TextBlock>
```

Теперь у *TextBlock* есть дочерний элемент *TextBlock.FontSize*, и его значение задано между тегами *TextBlock.FontSize*.

Это синтаксис *свойства-элемента* (*property-element*). Он играет весьма важную роль в XAML. Введение такого синтаксиса также способствует закреплению некоторой терминологии, объединяющей .NET и XML. Теперь этот один элемент *TextBlock* включает три типа идентификаторов:

- *TextBlock* – это *объектный элемент*, т.е. объект .NET, созданный на базе XML-элемента.
- *Text* и *Foreground* – это *свойства-атрибуты* (*property attributes*), т.е. .NET-свойства, заданные как XML-атрибуты.
- *FontSize* теперь задается *свойством-элементом*, т.е. .NET-свойством, выраженным как XML-элемент.

Впервые увидев синтаксис свойства-элемента, я подумал, не является ли это каким-то расширением XML. Конечно, нет. Точка – вполне допустимый символ для XML-тегов, поэтому с точки зрения вложенных XML-тегов все абсолютно законно. Вот то что они включают имя

класса и имя свойства – это специфика, которую распознают только синтаксические анализаторы XAML (как машинные, так и двуногие).

Однако есть одно ограничение: в теге свойства-элемента не может быть больше ничего:

```
<TextBlock Text="Hello, Windows Phone 7!"
           Foreground="Red">
    <!-- Недопустимый тег свойства-элемента! -->
    <TextBlock.FontSize здесь не может быть больше ничего!>
        36
    </TextBlock.FontSize>
</TextBlock>
```

Также нельзя создать для одного и того же свойства и свойство-атрибут, и свойство-элемент, как сделано в данном фрагменте:

```
<TextBlock Text="Hello, Windows Phone 7!"
           FontSize="36"
           Foreground="Red">
    <TextBlock.FontSize>
        36
    </TextBlock.FontSize>
</TextBlock>
```

Это неправильно, потому что свойство *FontSize* задано дважды.

Если вернуться к началу MainPage.xaml, можно увидеть еще одно свойство-элемент:

```
<Grid.RowDefinitions>
```

RowDefinitions (Описания строк) – это свойство *Grid*. В App.xaml мы найдем еще два:

```
<Application.Resources>
<Application.ApplicationLifetimeObjects>
```

И *Resources* (Ресурсы), и *ApplicationLIfeTimeObjects* (Объекты времени жизни приложения) являются свойствами *Application*.

Цвета и кисти

Вернем *TextBlock* в исходное состояние:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!" />
</Grid>
```

На экране отображается текст белого цвета (или черного, в зависимости от выбранной темы), потому что свойство *Foreground* задано в корневом элементе MainPage.xaml. Можно переопределить пользовательские настройки, задав свойство *Background* для *Grid* и *Foreground* для *TextBlock*:

```
<Grid x:Name="ContentPanel" Background="Blue" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

У *Grid* есть свойство *Background*, но нет свойства *Foreground*. У *TextBlock* есть свойство *Foreground*, но нет свойства *Background*. Свойство *Foreground* является наследуемым. Иногда может показаться, что наследуется и свойство *Background*, но это не так. По умолчанию *Background* имеет значение *null*, что делает фон прозрачным. Если фон элемента прозрачен, сквозь него виден фон его родительского элемента, это и создает впечатление того, что данное свойство наследуется.

Значения *null* и *Transparent* свойства *Background* создают одинаковый визуальный эффект, но обусловливают различную реакцию элемента на касание. *Grid*, свойству *Background* которого задано значение по умолчанию *null*, не распознает сенсорный ввод! Если требуется, чтобы у *Grid* не было собственного фона, но он отвечал на сенсорный ввод, свойству *Background* должно быть задано значение *Transparent*. Также можно сделать обратное: задать свойству *IsHitTestVisible* (Является видимым для касания) элемента с не-*null* фоном значение *false*, что обеспечит его безразличность к сенсорному вводу.

Цвета могут задаваться не только стандартными названиями, но также строкой, включающей одноразрядные шестнадцатеричные значения для красного, зеленого и синего в диапазоне от 00 до FF. Например:

```
Foreground="#FF0000"
```

Это тоже красный. Также можно задать *четыре* двузначных шестнадцатеричных числа, первое из которых является значением альфа-канала, определяющим прозрачность. Значение 00 – полностью прозрачный, FF – полностью непрозрачный, и все промежуточные значения определяют промежуточные степени прозрачности. Зададим такое значение:

```
Foreground="#80FF0000"
```

Текст будет несколько размытого пурпурного цвета из-за просвечивающегося синего фона.

Если поставить буквы *sc* перед знаком решетки, для задания компонентов красного, синего и зеленого можно использовать значения от 0 до 1:

```
Foreground="sc# 1 0 0"
```

Также перед этими тремя цифрами можно указать значение от 0 до 1 для альфа-канала.

Эти два метода числового задания цветов не эквивалентны, в чем можно убедиться, поместив эти два *TextBlock* в один *Grid*:

```
<Grid x:Name="ContentPanel" Background="Blue" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="RGB COLOR"
               HorizontalAlignment="Left"
               Foreground="#808080" />

    <TextBlock Text="scRGB COLOR"
               HorizontalAlignment="Right"
               Foreground="sc# 0.5 0.5 0.5" />
</Grid>
```

В обоих случаях задан серый цвет, но элемент справа выглядит намного светлее элемента слева.

Цвета, получаемые с помощью шестнадцатеричного формата, вероятно, наиболее привычны. Одноразрядные значения красного, зеленого и синего прямо пропорциональны напряжению, передаваемому на пикселы экрана. Яркость мониторов находится в нелинейной зависимости от напряжения, но человеческий глаз также имеет нелинейную чувствительность к яркости. Эти две «нелинейности» уравновешивают друг друга (почти), поэтому текст слева выглядит несколько темнее.

При использовании цветового пространства scRGB мы задаем значения в диапазоне от 0 до 1, которые пропорциональны яркости, но нелинейность восприятия яркости человеческим глазом создает эффект приглушенного цвета. Чтобы получить средненасыщенный серый с помощью scRGB, необходимо задавать значения, намного меньше 0,5, например:

```
Foreground="sc# 0.2 0.2 0.2"
```

Вернемся к одному *TextBlock* в *Grid*:

```
<Grid x:Name="ContentPanel" Background="Blue" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

Точно так же как мы делали это ранее со свойством *FontSize*, вынесем свойство *Foreground* как свойство-элемент:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        Red
    </TextBlock.Foreground>
</TextBlock>
```

Когда мы задаем свойство *Foreground* в XAML, для элемента в фоновом режиме создается *SolidColorBrush*. *SolidColorBrush* можно также создать явно в XAML:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        <SolidColorBrush Color="Red" />
    </TextBlock.Foreground>
</TextBlock>
```

Свойство *Color* тоже можно вынести как свойство-элемент:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                Red
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Можно пойти даже еще дальше:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                <Color>
                    <Color.A>
                        255
                    </Color.A>
                    <Color.R>
                        #FF
                    </Color.R>
                </Color>
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Обратите внимание, что свойство *A* структуры *Color* должно быть задано явно, потому что его значение по умолчанию 0, что означает прозрачный.

Использование свойств-элементов, возможно, не имеет особого смысла для простых цветов и *SolidColorBrush*. Но эта методика становится просто незаменимой, если требуется задать в XAML значение свойства, которое не может быть выражено простой текстовой строкой. Например, если необходимо использовать градиентную кисть, а не *SolidColorBrush*.

Начнем с простого одноцветного *TextBlock*, но при этом вынесем свойство *Background* контейнера для содержимого как свойство-элемент:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
```

```
        <SolidColorBrush Color="Blue" />
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

Удалим *SolidColorBrush* и заменим его на *LinearGradientBrush* (Линейная градиентная кисть):

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

У *LinearGradientBrush* есть свойство типа *GradientStops* (Точки градиента), поэтому добавим теги свойств-элементов для свойства *GradientStops*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

Свойство *GradientStops* типа *GradientStopCollection* (Коллекция точек градиента), поэтому вставим теги и для него:

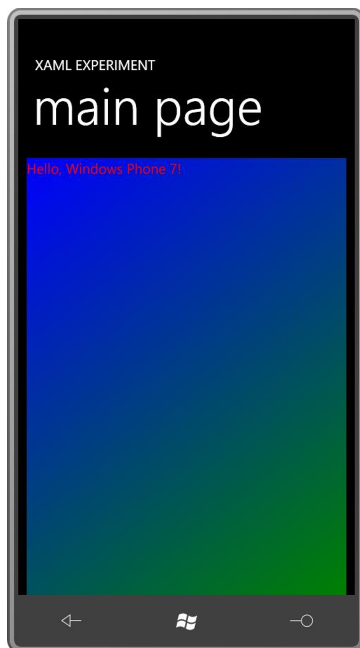
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

Теперь добавим здесь пару объектов *GradientStop*. У *GradientStop* есть свойства *Offset* (Смещение) и *Color*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
```

```
                Foreground="Red" />
</Grid>
```

Вот так с помощью свойств-элементов можно создать градиентную кисть в разметке. На экране телефона это выглядит следующим образом:



Значения *Offset* находятся в диапазоне от 0 до 1 и отсчитываются относительно элемента, закрашиваемого кистью. Может использоваться более двух смещений:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="0.5" Color="White" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

По сути, кисть «знает» размер области для закрашивания и подстраивает себя соответственно ей.

По умолчанию градиент распространяется из верхнего левого угла в нижний правый угол. Такое поведение обусловлено значениями по умолчанию свойств *StartPoint* (Начальная точка) и *EndPoint* (Конечная точка) объекта *LinearGradientBrush*. Как следует из их имен, значения этих свойств являются координатами, и отсчитываются они относительно верхнего левого угла закрашиваемого элемента. Значение по умолчанию для *StartPoint* – (0, 0), т.е. верхний левый угол; значение по умолчанию для *EndPoint* – (1, 1), т.е. нижний правый угол. Если изменить их и задать, например, (0, 0) и (0, 1), градиент будет распространяться сверху вниз:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
```

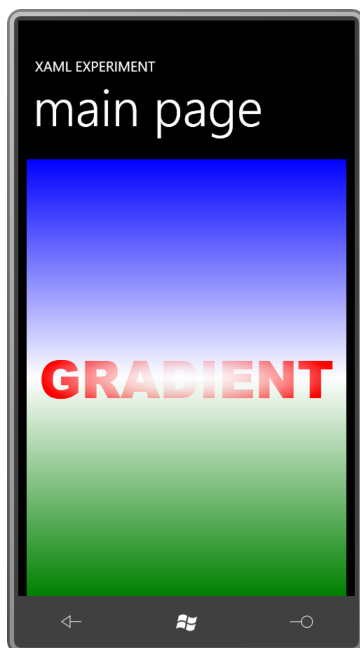
```
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="0.5" Color="White" />
                    <GradientStop Offset="1" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="Hello, Windows Phone 7!"
               Foreground="Red" />
</Grid>
```

Каждая точка задается просто двумя числами, разделенными пробелом или запятой. Есть также свойства, определяющие, что происходит вне области допустимых значений *Offset*, если диапазон задан значениями, отличными от 0 и 1.

LinearGradientBrush является производным от *GradientBrush*. От *GradientBrush* наследуется еще один класс, *RadialGradientBrush* (Радиальная градиентная кисть). Рассмотрим разметку для более крупного *TextBlock*, в качестве значения *Foreground* которого задан *RadialGradientBrush*:

```
<TextBlock Text="GRADIENT"
           FontFamily="Arial Black"
           FontSize="72"
           HorizontalAlignment="Center"
           VerticalAlignment="Center">
    <TextBlock.Foreground>
        <RadialGradientBrush>
            <RadialGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Transparent" />
                    <GradientStop Offset="1" Color="Red" />
                </GradientStopCollection>
            </RadialGradientBrush.GradientStops>
        </RadialGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

И вот как это сочетание выглядит на экране:



Содержимое и свойства содержимого

Всем известно, что XML может быть несколько «многословным», но тем не менее приведенный выше код разметки с градиентными кистями немного более «многословный», чем надо. Рассмотрим *RadialGradientBrush*, изначально заданный для *TextBlock*:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
            <GradientStopCollection>
                <GradientStop Offset="0" Color="Transparent" />
                <GradientStop Offset="1" Color="Red" />
            </GradientStopCollection>
        </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
</TextBlock.Foreground>
```

Во-первых, если в коллекции имеется хотя бы один элемент, теги самой коллекции можно убрать. Это означает, что теги *GradientStopCollection* могут быть опущены:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <RadialGradientBrush.GradientStops>
            <GradientStop Offset="0" Color="Transparent" />
            <GradientStop Offset="1" Color="Red" />
        </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
</TextBlock.Foreground>
```

Более того, многие используемые в XAML классы имеют атрибут *ContentProperty* (Свойство содержимого). Слово «атрибут» имеет разное значение в .NET и XML. Здесь мы говорим о .NET-атрибуте, под которым подразумеваются некоторые дополнительные сведения, ассоциированные с классом или членом этого класса. Если взглянуть на документацию класса *GradientBrush* (класса, от которого наследуются и *LinearGradientBrush*, и *RadialGradientBrush*), можно увидеть, что в его описание включен атрибут типа *ContentPropertyAttribute* (Атрибут свойства содержимого):

```
[ContentPropertyAttribute("GradientStops", true)]
public abstract class GradientBrush : Brush
```

Этот атрибут обозначает одно свойство этого класса, которое предполагается как содержимое этого класса, и для которого теги свойства-элемента не нужны. Для *GradientBrush* (и его потомков) таким свойством является *GradientStops*. Это означает, что теги *RadialGradientBrush.GradientStops* можно удалить из разметки:

```
<TextBlock.Foreground>
    <RadialGradientBrush>
        <GradientStop Offset="0" Color="Transparent" />
        <GradientStop Offset="1" Color="Red" />
    </RadialGradientBrush>
</TextBlock.Foreground>
```

Теперь не так многословно, но при этом вполне понятно: два объекта *GradientStop* являются содержимым класса *RadialGradientBrush*.

Ранее в данной главе я создал *TextBlock* в коде и добавил его в коллекцию *Children* объекта *Grid*. Но в XAML мы не видим никакой ссылки на эту коллекцию *Children*. Ссылки нет, потому что атрибут *ContentProperty* класса *Panel* (родительского класса *Grid*) определяет свойство *Children* как содержимое *Panel*:

```
[ContentPropertyAttribute("Children", true)]
public abstract class Panel : FrameworkElement
```

Чтобы сделать разметку более наглядной, в нее можно включить свойство-элемент для свойства *Children*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Children>
        <TextBlock Text="Hello, Windows Phone 7!" />
    </Grid.Children>
</Grid>
```

Аналогично *PhoneApplicationPage* наследуется от *UserControl* (Пользовательский элемент управления), у которого также есть атрибут *ContentProperty*:

```
[ContentPropertyAttribute("Content", true)]
public class UserControl : Control
```

Атрибут *ContentProperty* класса *UserControl* – это свойство *Content*. (Чтобы понять это предложение, его лучше увидеть, чем услышать!)

Предположим, требуется вставить в *Grid* два элемента *TextBlock* и в качестве *Background* для *Grid* использовать *LinearGradientBrush*. Для этого сначала в теги *Grid* можно поместить свойство-элемент *Background* и за ним добавить два элемента *TextBlock*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />

    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />
</Grid>
```

Также можно сначала задать два элемента *TextBlock* и после них вставить свойство-элемент *Background*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />

    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />

    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

Но вот если поместить свойство-элемент *Background* между двумя *TextBlock*, ничего не получится:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="TextBlock #1"
               HorizontalAlignment="Left" />

    <!-- Свойство-элемент не может располагаться здесь! -->
    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
```

```
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Text="TextBlock #2"
               HorizontalAlignment="Right" />
</Grid>
```

Абсолютно очевидные проблемы с этим синтаксисом выявляются, когда мы помещаем недостающие свойства-элементы для свойства *Children* нашего *Grid*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Children>
        <TextBlock Text="TextBlock #1"
                   HorizontalAlignment="Left" />
    </Grid.Children>

    <!-- Свойство-элемент не может располагаться здесь! -->
    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Offset="0" Color="LightCyan" />
            <GradientStop Offset="1" Color="LightPink" />
        </LinearGradientBrush>
    </Grid.Background>

    <Grid.Children>
        <TextBlock Text="TextBlock #2"
                   HorizontalAlignment="Right" />
    </Grid.Children>
</Grid>
```

Теперь очевидно, что свойство *Children* задано дважды, и, несомненно, это неверно.

Коллекция ресурсов

В некотором смысле, одна из основных задач разработки ПО для компьютеров – максимальный уход от повторений кода. (Или, по крайней мере, в той части, что касается кода, создаваемого людьми. Мы не против, если наши компьютеры повторяются, мы просто хотим, чтобы эти повторения были эффективными.) XAML может показаться особенно плодородной почвой для повторений, потому что это всего лишь разметка, а не настоящий язык программирования. Не составит труда представить ситуации, когда целый ряд элементов будут иметь одинаковые параметры (например, значения свойств *HorizontalAlignment*, *VerticalAlignment* или *Margin*). Безусловно, для таких случаев удобно иметь способ избежать повторяющейся разметки. Если когда-либо возникнет необходимость изменить одно из этих свойств, изменять его один раз и в одном месте намного удобнее, чем бесконечное множество раз.

К счастью, создателями XAML были разработчики, которые (как и все мы) предпочитают не вводить одно и то же снова и снова.

Самым универсальным решением, позволяющим избежать повторяющейся разметки, является применение *стиля* Silverlight. А предварительные настройки стилей – самый обобщенный механизм совместного использования стилей. Их называют *ресурсом*. Не стоить путать ресурсы, о которых пойдет речь здесь, с ресурсами, обсуждаемыми в главе 4 при рассмотрении встраивания изображений в приложение. Чтобы избежать возможной путаницы, я буду называть ресурсы в данной главе XAML-ресурсами, даже если они могут существовать также и в коде.

XAML-ресурсы – это всегда экземпляры определенного .NET-класса или структуры, либо существующего класса или структуры, либо пользовательского класса. Если в качестве XAML-

ресурса определен конкретный класс, создается только один экземпляр данного класса, который используется совместно всеми сторонами, нуждающимися в этом ресурсе.

Факт совместного использования ресурсов немедленно налагает ограничение на использование многих классов в качестве XAML-ресурсов. Например, один экземпляр *TextBlock* не может использоваться более одного раза, потому что у *TextBlock* должен быть уникальный родитель и уникальное местоположение в границах этого родителя. И то же самое можно сказать о любом другом элементе. Все производные от *UIElement* не будут выступать в роли ресурсов, потому что не могут быть использованы совместно.

А вот совместное использование кистей – довольно типичное явление. Это самый распространенный способ обеспечить характерное и единообразное визуальное представление приложения. Анимации тоже подходят для совместного использования, как и текстовые строки и числа. Их можно рассматривать как XAML-эквиваленты строковых или числовых констант в приложении на C#. Если необходимо изменить одну из них, можно просто внести изменения в ресурс, и не выискивать все случаи употребления этих величин по всему XAML.

Для обеспечения хранения ресурсов в классе *FrameworkElement* определено свойство *Resources* типа *ResourceDictionary* (Словарь ресурсов). Для любого элемента, производного от *FrameworkElement*, можно задать *Resources* как свойство-элемент. По соглашению его размещают сразу под открывающим тегом. Рассмотрим коллекцию для класса страницы, унаследованного от *PhoneApplicationPage*:

```
<phone:PhoneApplicationPage … >

    <phone:PhoneApplicationPage.Resources>
        …
    </phone:PhoneApplicationPage.Resources>

    …
</phone:PhoneApplicationPage>
```

Коллекцию ресурсов, ограниченную тегами *Resources*, иногда называют *разделом ресурсов*. Все элементы данного конкретного *PhoneApplicationPage* могут использовать перечисленные ресурсы.

Класс *Application* также определяет свойство *Resources*, и файл App.xaml, автоматически создаваемый Visual Studio в новом приложении на Silverlight, включает пустой раздел ресурсов:

```
<Application … >
    <Application.Resources>
    </Application.Resources>
    …
</Application>
```

Ресурсы, определенные в коллекции *Resources* для *FrameworkElement*, доступны только в рамках этого элемента и вложенных в него элементов. Ресурсы, определенные в классе *Application*, доступны всему приложению.

Совместное использование кистей

Предположим, страница включает несколько *TextBlock*, и для *Foreground* всех этих элементов требуется применить *LinearGradientBrush*. Идеальное условие для использования ресурса.

Первый шаг – определение *LinearGradientBrush* в разделе ресурсов XAML-файла. Если ресурс задается в производном от *FrameworkElement* элементе, он должен быть описан до его применения, и доступ к нему имеет только данный или вложенный в него элемент.

```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Key="brush">
        <GradientStop Offset="0" Color="Pink" />
        <GradientStop Offset="1" Color="SkyBlue" />
    </LinearGradientBrush>
</phone:PhoneApplicationPage.Resources>
```

Обратите внимание на атрибут *x:Key*. Каждый ресурс должен иметь имя ключа. Существует всего четыре ключевых слова, которые необходимо предварять «x». Мы уже видели три из них – *x:Class*, *x:Name* и *x:Null* – и *x:Key* четвертое.

Для реализации доступа к этому ресурсу предусмотрено несколько типов синтаксиса. Довольно развернутый способ – вынести свойство *Foreground* нашего *TextBlock* как свойство-элемент и задать в качестве его значения объект типа *StaticResource*, указывая имя ключа:

```
<TextBlock Text="Hello, Windows Phone 7!">
    <TextBlock.Foreground>
        <StaticResource ResourceKey="brush" />
    </TextBlock.Foreground>
</TextBlock>
```

Но есть и более краткий синтаксис, в котором используется так называемое *расширение разметки XAML*. Расширение разметки всегда заключается в фигурные скобки. Вот как выглядит расширение разметки для *StaticResource*:

```
<TextBlock Text="Hello, Windows Phone 7!"
           Foreground="{StaticResource brush}" />
```

Обратите внимание, что в рамках расширения разметки слово «brush» используется без кавычек. Использование кавычек в расширении разметки запрещено.

Скажем, требуется сделать общим параметр *Margin*. *Margin* типа *Thickness*, и в XAML он может быть задан одним, двумя или четырьмя числами. Вот пример ресурса *Thickness*:

```
<Thickness x:Key="margin">
    12 96
</Thickness>
```

Предположим, необходимо совместно использовать свойство *FontSize*. Это свойство типа *double*, и в этом случае нам понадобиться небольшая помощь. Структура *Double*, которая лежит в основе типа данных C# *double*, определена в пространстве имен *System*. Но описания пространств имен XML в обычном XAML-файле ссылаются только на классы Silverlight из пространств имен Silverlight. Нам требуется включить описание пространства имен XML для пространства имен *System* в корневой элемент страницы. Вот оно:

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

Это стандартный синтаксис для ассоциирования пространства имен XML и пространства имен .NET. Прежде всего зададим для пространства имен XML имя, созвучное с пространством имен .NET. В данном случае подойдет «system» (некоторые разработчики используют «sys» или просто «s»). За написанным через дефис «clr-namespace» следуют двоеточие и имя пространства имен .NET. Если нас интересуют объекты текущей сборки, больше ничего делать не надо. В противном случае, ставим точку с запятой, «assembly=» и указываем необходимую сборку; здесь используется стандартная mscorlib.lib (Microsoft Common Runtime Library[1]).

---

[1] Библиотека общеязыковой среды выполнения (*прим. переводчика*).

Теперь мы можем определять ресурс типа *double*:

```
<system:Double x:Key="fontsize">
    48
</system:Double>
```

Все эти три ресурса определяются в проекте ResourceSharing и используются в двух элементах *TextBlock*. Привожу здесь раздел ресурсов полностью:

Проект Silverlight: ResourceSharing   Файл: MainPage.xaml (фрагмент)

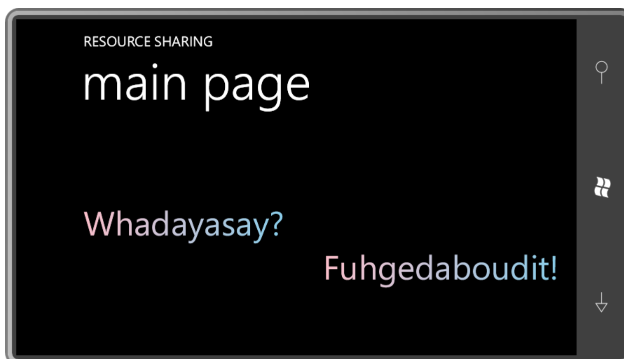
```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Key="brush">
        <GradientStop Offset="0" Color="Pink" />
        <GradientStop Offset="1" Color="SkyBlue" />
    </LinearGradientBrush>

    <Thickness x:Key="margin">
        12 96
    </Thickness>

    <system:Double x:Key="fontsize">
        48
    </system:Double>
</phone:PhoneApplicationPage.Resources>
```

Сетка для содержимого включает два элемента *TextBlock*:

Проект Silverlight: ResourceSharing   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Whadayasay?"
               Foreground="{StaticResource brush}"
               Margin="{StaticResource margin}"
               FontSize="{StaticResource fontsize}"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    <TextBlock Text="Fuhgedaboudit!"
               Foreground="{StaticResource brush}"
               Margin="{StaticResource margin}"
               FontSize="{StaticResource fontsize}"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom" />
</Grid>
```

Снимок экрана демонстрирует, как это выглядит в действии:



Свойство *Resources* – это словарь, поэтому имена ключей в разделе ресурсов должны быть уникальными. Но в разных коллекциях ресурсов имена ключей могут повторяться. Например, вставим следующую разметку прямо после открывающего тега сетки для содержимого:

```
<Grid.Resources>
    <Thickness x:Key="margin">96</Thickness>
</Grid.Resources>
```

Этот ресурс переопределит тот, который задан в *MainPage*. Поиск ресурсов производится по имени ключа снизу вверх по дереву визуальных элементов, после чего выполняется поиск в коллекции *Resources* класса *App*. Поэтому коллекция *Resources* в App.xaml – замечательное место для размещения ресурсов, используемых всем приложением.

Если этот фрагмент разметки поместить в *Grid* под именем «LayoutRoot» (Корневой элемент разметки), он будет также доступен и элементам *TextBlock*, потому что они являются дочерними элементами этого *Grid*. Но если поместить разметку в *StackPanel*, озаглавленный «TitlePanel» (и заменить *Grid* на *StackPanel*), разметка будет проигнорирована. Поиск ресурсов производится вверх по дереву визуальных элементов, а это будет лишь другая ветвь по отношению к элементам *TextBlock*.

Этот фрагмент разметки будет также проигнорирован, если будет помещен в сетку для содержимого *после* описания элементов *TextBlock*. Сейчас он недоступен, потому что располагается после ссылки на него.

x:Key и x:Name

Чтобы использовать XAML-ресурс из кода, можно просто указать в свойстве *Resources* имя необходимого ресурса как индекс. Например, в файле выделенного кода MainPage.xaml.cs данный код будет обеспечивать извлечение ресурса под именем «brush» из коллекции *Resources* элемента *MainPage*:

```
this.Resources["brush"]
```

После этого, вероятно, понадобится привести этот объект к соответствующему типу, в данном случае это либо *Brush*, либо *LinearGradientBrush*. Сборка коллекции *Resources* выполняется во время обработки XAML, поэтому этот ресурс не может использоваться до вызова метода *InitializeComponent* в конструкторе файла выделенного кода.

Точно так же можно ссылаться на ресурсы, определенные в других коллекциях *Resource* того же XAML-файла. Например, пусть имеется ресурс «margin», описанный в коллекции *Resources* сетки для содержимого. Доступ к нему можно организовать следующим образом:

```
ContentPanel.Resources["margin"]
```

Если в коллекции *Resources* элемента ресурс с таким именем не найден, поиск выполняется в коллекции *Resources* класса *App*. Если ресурс не найден и там, индексатор возвращает *null*.

Аспекты, связанные с наследием Silverlight 1.0, позволяют использовать для идентификации ресурсов *x:Name*, а не *x:Key*:

```
<phone:PhoneApplicationPage.Resources>
    <LinearGradientBrush x:Name="brush">
    …
</phone:PhoneApplicationPage.Resources>
```

Здесь есть одно большое преимущество: имя хранится как поле в автоматически формируемом файле кода, что позволяет ссылаться на ресурс в файле выделенного кода, как на любое другое поле:

```
txtblk.Foreground = brush;
```

Это самый удобный синтаксис для совместного использования ресурсов в XAML и коде. Но если для ресурса применяется *x:Name*, это имя должно быть уникальным в рамках XAML-файла.

Введение в стили

Одним из самых распространенных элементов в коллекции *Resources* является *Style*, который по сути является набором значений свойств для конкретного типа элементов. Кроме ключа, *Style* также требует задания *TargetType* (Целевой тип):

```
<Style x:Key="txtblkStyle"
       TargetType="TextBlock">
       …
</Style>
```

Между открывающим и закрывающим тегами размещается одно или более описаний *Setter*. У *Setter* два свойства: одно из них называется *Property* (Свойство), и в качестве его значения задается имя свойства; другое – *Value* (Значение). Приведем несколько примеров:

```
<Style x:Key="txtblkStyle"
       TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="Margin" Value="12 96" />
    <Setter Property="FontSize" Value="48" />
</Style>
```

Предположим, мы хотим также включить *Setter* для свойства *Foreground*, но его значением является *LinearGradientBrush*. Существует два способа сделать это. Если ранее был описан ресурс с ключом «brush» (как в проекте ResourceSharing), можно вставить ссылку на этот ресурс:

```
<Setter Property="Foreground" Value="{StaticResource brush}" />
```

Или можно прибегнуть к синтаксису свойства-элемента для свойства *Value* и встроить кисть прямо в описание *Style*. Вот как это сделано в коллекции *Resources* в проекте StyleSharing:

Проект Silverlight: StyleSharing Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="txtblkStyle"
           TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Margin" Value="12 96" />
        <Setter Property="FontSize" Value="48" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Pink" />
                    <GradientStop Offset="1" Color="SkyBlue" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Чтобы применить этот стиль к элементу типа *TextBlock*, зададим свойство *Style* (которое определено в *FrameworkElement*, так что доступно всем типам элементов):

Проект Silverlight: StyleSharing Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Whadayasay?"
               Style="{StaticResource txtblkStyle}"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    <TextBlock Text="Fuhgedaboudit!"
               Style="{StaticResource txtblkStyle}"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom" />
</Grid>
```

Изображение на экране ничем не отличается от вывода предыдущего приложения, что весьма поучительно. Таким образом, значения *HorizontalAlignment* и *VerticalAlignment* определены в *Style*, но переопределены локальными параметрами в двух элементах *TextBlock*. А значение *Foreground*, заданное в *Style*, переопределяет значение, наследуемое по дереву элементов.

Итак, теперь мы можем немного дополнить схему, начатую ранее в данной главе:

Локальные параметры имеют приоритет над

Настройками стилей, которые являются более приоритетными, чем

Унаследованные свойства, которые являются более приоритетными, чем

Значения по умолчанию

Наследование стилей

Стили могут дополнять или изменять другие стили посредством наследования. Зададим в качестве значения свойства *BasedOn* (Основан на) нашего *Style* определенный ранее *Style*. Вот так выглядит коллекция *Resources* проекта StyleInheritance:

Проект Silverlight: StyleInheritance Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="txtblkStyle"
           TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Margin" Value="12 96" />
        <Setter Property="FontSize" Value="48" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Pink" />
                    <GradientStop Offset="1" Color="SkyBlue" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>

    <Style x:Key="upperLeftStyle"
           TargetType="TextBlock"
           BasedOn="{StaticResource txtblkStyle}">
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="VerticalAlignment" Value="Top" />
    </Style>
```

```
    <Style x:Key="lowerRightStyle"
           TargetType="TextBlock"
           BasedOn="{StaticResource txtblkStyle}">
        <Setter Property="HorizontalAlignment" Value="Right" />
        <Setter Property="VerticalAlignment" Value="Bottom" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Два последних описания *Style* переопределяют свойства *HorizontalAlignment* и *VerticalAlignment*, заданные в первом стиле. Благодаря этому два элемента *TextBlock* могут использовать эти два разных стиля:

Проект Silverlight: StyleInheritance Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Whadayasay?"
               Style="{StaticResource upperLeftStyle}" />

    <TextBlock Text="Fuhgedaboudit!"
               Style="{StaticResource lowerRightStyle}" />
</Grid>
```

Неявные стили, введенные в Silverlight 4, не поддерживаются в Silverlight для Windows Phone.

Темы

Посредством расширения разметки *StaticResource* Windows Phone 7 предопределяет множество ресурсов, которые могут использоваться во всем приложении. Существуют встроенные цвета, кисти, имена шрифтов, размеры шрифтов, поля и стили текста. Некоторые из них описываются в корневом элементе MainPage.xaml, обеспечивая значения по умолчанию для всей страницы:

```
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
```

Все встроенные темы можно найти в разделе Themes (Темы) документации по Windows Phone 7. Данные ресурсы следует применять к кистям переднего плана и фона. Это обеспечит выполнение требований пользователя и при этом не допустит, чтобы текст случайно стал невидимым. Некоторые встроенные размеры шрифтов могут отличаться на телефонах с небольшим экраном, и эти отличия могут помочь портировать приложения, создаваемые для большого экрана, на новое устройство.

Что произойдет, если пользователь перейдет к странице Settings телефона и поменяет тему прямо во время выполнения приложения? При переходе к странице настроек, приложение будет захоронено, и когда оно будет повторно активировано, его выполнение начнется заново, следовательно, оно автоматически будет использовать новые цвета.

Выбираемая пользователем цветовая тема включает цвет фона и переднего плана (белый на черном фоне либо черный на белом фоне), а также контрастный цвет: magenta (пурпурный), purple (фиолетовый), teal (бирюзовый), lime (светло-зеленый), brown (коричневый), pink (розовый), orange (оранжевый), blue (голубой) (по умолчанию), red (красный) или green (зеленый). Этот цвет доступен как ресурс PhoneAccentColor (Контрастный цвет телефона), и кисть, использующая этот цвет, доступна как ресурс PhoneAccentBrush (Кисть контрастного цвета телефона).

Градиент

Предположим, в приложении решено использовать выбранный пользователем контрастный цвет для градиентной кисти. То есть вы не собираетесь менять оттенок, но хотите сделать его ярче или темнее. Сделать это в коде довольно легко: просто варьируем красным, зеленым и синим компонентами цвета.

Но и в XAML реализовать это не составляет труда, что демонстрирует проект GradientAccent:

Проект Silverlight: GradientAccent Файл: MainPage.xaml (фрагмент)

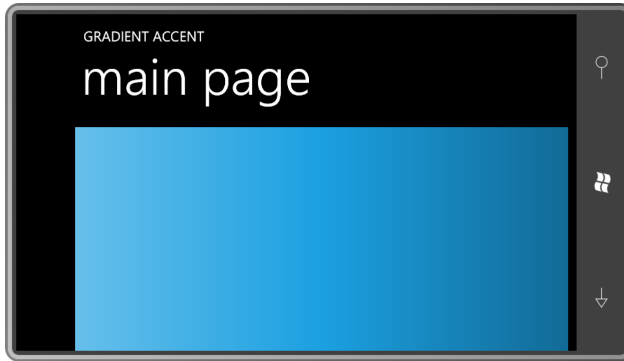
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <GradientStop Offset="0" Color="White" />
            <GradientStop Offset="0.5" Color="{StaticResource PhoneAccentColor}" />
            <GradientStop Offset="1" Color="Black" />
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

Вот как это выглядит на экране:

Изменяя смещения градиента, можно создать более плавный переход. Значения смещений, на самом деле, могут выходить за рамки диапазона от 0 до 1, как это показано в данном фрагменте:

```
<LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
    <GradientStop Offset="-1" Color="White" />
    <GradientStop Offset="0.5" Color="{StaticResource PhoneAccentColor}" />
    <GradientStop Offset="2" Color="Black" />
</LinearGradientBrush>
```

Теперь градиент распространяется от белого (White) в точке со смещением -1 до контрастного цвета в точке со смещением 0,5 и переходит в черный (Black) в точке со смещением 2. Но на экране отображается лишь часть градиента между точками со смещениями 0 и 1, поэтому мы не видим белой и черной границ:



Это всего лишь еще один пример того, что XAML обладает намного большей мощью, чем это может показаться на первый взгляд.

Глава 8

Элементы и свойства

Ранее в данной книге нами были рассмотрены примеры *TextBlock* и *Image*. Эти два элемента, безусловно, являются наиболее важными из поддерживаемых Silverlight элементами. Теперь пришло время исследовать текст и растровые изображения более детально, чем мы и займемся в этой главе. Также познакомимся с другими часто используемыми элементами и некоторыми важными свойствами, которые могут применяться к ним, в том числе и трансформациями. Тем самым мы подготовим почву для работы с элементами *Panel*, которые составляют основу системы динамической компоновки Silverlight (это тема следующей главы), и всей массой остальных элементов управления (глава 10).

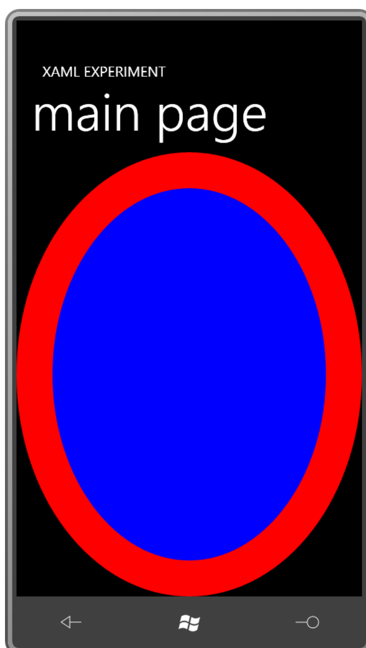
Основные фигуры

Пространство имен *System.Windows.Shapes* включает элементы для отображения векторной графики, т.е. использующие прямые линии и кривые для отрисовки или определения закрашенных областей. Более детальное рассмотрение вопросов векторной графики нас ожидает в главе 13, а сейчас остановимся лишь на двух классах этого пространства имен – *Ellipse* (Эллипс) и *Rectangle* (Прямоугольник) – которые немного отличаются от всех остальных тем, что могут использоваться без задания координат.

Вернемся к приложению XamlExperiment, с которым мы работали в главе 7, и добавим элемент *Ellipse* в сетку для содержимого:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Ellipse Fill="Blue"
             Stroke="Red"
             StrokeThickness="50" />
</Grid>
```

На экране мы увидим синий эллипс с красной окантовкой, занимающий всю область *Grid*:



Теперь присвоим свойствам *HorizontalAlignment* и *VerticalAlignment* значение *Center*. Эллипс исчез. Что произошло?

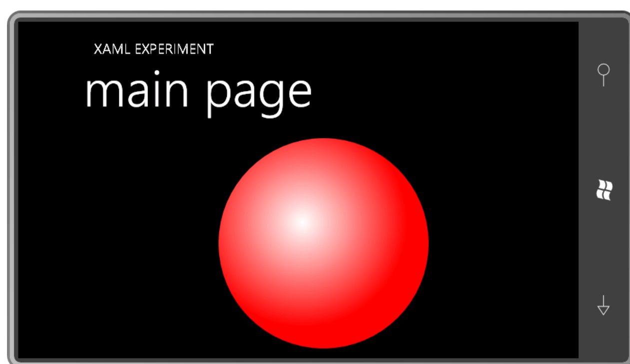
У данного эллипса нет подразумеваемого минимального размера. Если ничто ему не мешает, он занимает весь объем контейнера, но если его размер начинают ограничивать, он становится предельно малым, т.е. сжимается в точку. Это тот случай, когда желательно, а зачастую просто необходимо, явно задавать значения свойств *Width* и *Height* элемента.

Понятия *обводка* (*stroke*) и *заливка* (*fill*) очень распространены в векторной графике. В основе векторной графики лежит определение прямых линий и кривых с помощью координат. Прямые и кривые линии - это математические сущности, которые становятся видимыми только будучи обведенными с использованием определенного цвета и толщины. Линии и кривые также могут определять замкнутые области. Эти области закрашиваются с применением *заливки*. Оба свойства класса *Ellipse*, отвечающие за эту функциональность – *Fill* и *Stroke* – типа *Brush*, поэтому к ним могут применяться градиентные кисти.

Чтобы получить круг, свойствам *Width* и *Height* класса *Ellipse* задают одинаковые значения. Свойство *Fill* можно определить как кисть *RadialGradientBrush*, которая начинается с белого в центре круга и переходит к градиентному цвету на периметре. Как правило, центром градиента является точка с координатами (0.5, 0.5) относительно круга, но можно ввести смещение:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Ellipse Width="300"
             Height="300">
        <Ellipse.Fill>
            <RadialGradientBrush Center="0.4 0.4"
                                 GradientOrigin="0.4 0.4">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="Red" />
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
</Grid>
```

Такое смещение белой точки создает эффект отражения источника света и делает фигуру объемной:



Кроме всех тех же свойств, что и у класса *Ellipse*, класс *Rectangle* определяет свойства *RadiusX* и *RadiusY*, обеспечивающие скругление углов.

Трансформации

До появления Windows Presentation Foundation и Silverlight трансформации были преимущественно инструментом для знатоков графики. С математической точки зрения

трансформация – это применение простой формулы ко всем координатам визуального объекта, что приводит к перемещению, изменению размера или повороту объекта.

В Silverlight трансформации могут применяться к любому объекту, который наследуется от *UIElement* и включает текст, растровые изображения, фильмы, панели и элементы управления. Трансформации возможны в *UIElement* благодаря свойству *RenderTransform* (Трансформация визуального представления), значением которого является объект типа *Transform*. *Transform* – это абстрактный класс, но от него наследуются семь неабстрактных классов:

- *TranslateTransform* (Трансформация переносом) обеспечивает изменение местоположения объекта
- *ScaleTransform* (Трансформация масштабированием) обеспечивает увеличение или уменьшение размера
- *RotateTransform* (Трансформация вращением) обеспечивает вращение вокруг точки
- *SkewTransform* (Трансформация наклонением) обеспечивает смещение одного размера относительного другого
- *MatrixTransform* (Трансформация с использованием матрицы) обеспечивает представление трансформаций с помощью стандартной матрицы
- *TransformGroup* (Группа трансформаций) позволяет комбинировать несколько трансформаций
- *CompositeTransform* (Составная трансформация) позволяет задавать ряд трансформаций в определенном порядке

Трансформации могут быть довольно сложными, особенно если это составные трансформации. Мы рассмотрим здесь лишь самые основные из них. Очень часто трансформации используются в сочетании с анимациями. Анимация трансформации – наиболее эффектный вид анимации визуального объекта.

Предположим, имеется *TextBlock*, и требуется увеличить его в два раза. Нет ничего проще: удваиваем значение его свойства *FontSize*. Теперь, предположим, необходимо сделать текст в два раза шире и в три раза выше. Здесь одним изменением *FontSize* не обойтись. Понадобится вынести свойство *RenderTransform* как свойство-элемент и задать *ScaleTransform* в качестве его значения:

```
<TextBlock … >
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="2" ScaleY="3" />
    </TextBlock.RenderTransform>
</TextBlock>
```

Чаще всего в качестве значения свойства *RenderTransform* используется объект типа *TranslateTransform*, *ScaleTransform* или *RotateTransform*. Трансформации можно объединять в *TransformGroup*. Двухмерные трансформации представляются в виде матриц 3×3, сочетание трансформаций эквивалентно умножению матриц. Общеизвестно, что умножение матриц не является коммутативным, поэтому порядок трансформаций имеет значение.

Несмотря на то, что применение *TransformGroup* считается сложным случаем, я включил *TransformGroup* в свой небольшой проект TransformExperiment (Опыт с трансформациями), который позволяет поэкспериментировать с четырьмя стандартными трансформациями. Изначально в этом проекте все свойства имеют значения по умолчанию:

Проект Silverlight: TransformExperiment Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Transform Experiment"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleX="1" ScaleY="1"
                                CenterX="0" CenterY="0" />
                <SkewTransform AngleX="0" AngleY="0"
                               CenterX="0" CenterY="0" />
                <RotateTransform Angle="0"
                                 CenterX="0" CenterY="0" />
                <TranslateTransform X="0" Y="0" />
            </TransformGroup>
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Можете поэкспериментировать с этим приложением прямо в Visual Studio. Сначала попробуйте каждый тип трансформации отдельно. Хотя *TranslateTransform* и расположен в конце группы, начните с него. Меняя значение свойства *X*, можно смещать текст вправо или влево (задавая отрицательные значения). Изменение значения свойства *Y* обеспечивает перемещение текста вверх или вниз. Задайте *Y* равным -400, и текст окажется вверху в области заголовка!

TranslateTransform используется для создания теней и эффектов выпуклого или вдавленного текста. Просто поместите два элемента с одинаковым текстом в одно и то же место. Все свойства текста должны быть одинаковыми, отличаются только значения свойства *Foreground*. Без применения каких-либо трансформаций второй *TextBlock* просто располагается поверх первого. К любому из двух примените *ScaleTransform*, и получите абсолютно фантастический результат. Эту технику демонстрирует проект EmbossedText (Выпуклый текст). Вот два элемента *TextBlock* в одном *Grid*:

Проект Silverlight: EmbossedText Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="EMBOSS"
                Foreground="{StaticResource PhoneForegroundBrush}"
                FontSize="96"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />

    <TextBlock Text="EMBOSS"
                Foreground="{StaticResource PhoneBackgroundBrush}"
                FontSize="96"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock.RenderTransform>
            <TranslateTransform X="2" Y="2" />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Заметьте, что в качестве значений свойств *Foreground* я использовал цвета темы. В случае применения стандартной темной темы нижний *TextBlock* будет белым, и верхний – черным, как фон, но немного смещенным, так что белый чуть-чуть выглядывает:

Обычно этот прием применяется к черному тексту на белом фоне, но с такой цветовой схемой получается тоже неплохой эффект.

Обратимся опять к проекту TransformExperiment. Вернем значения свойств *TranslateTransform* к применяемым по умолчанию (0) и поэкспериментируем немного со свойствами *ScaleX* и *ScaleY* для *ScaleTransform*. По умолчанию оба свойства имеют значение 1. Значения больше 1 обеспечат увеличение размеров текста в горизонтальном и вертикальном направлениях; значения меньше 1 обусловят уменьшение текста. Можно использовать даже отрицательные значения, чтобы развернуть текст вокруг горизонтальной или вертикальной оси.

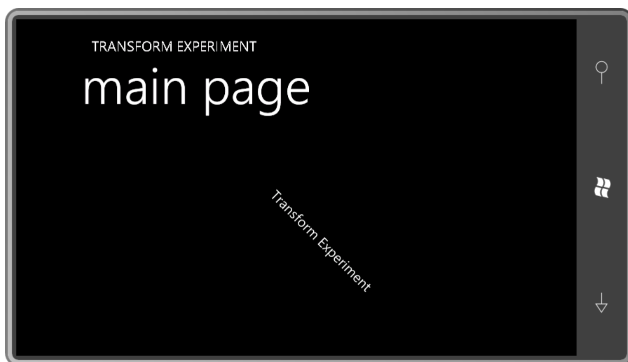
Любое изменение масштаба выполняется относительно верхнего левого угла текста. Другими словами, при увеличении или уменьшении размера текста, его верхний левый угол остается закрепленным на одном месте. Это может быть сложно заметить, поскольку верхний левый угол, на самом деле, располагается немного *выше* горизонтального штриха первой «Т» текстовой строки, в области, зарезервированной для диакритических знаков, таких как знаки ударения и умляуты.

Предположим, требуется изменить масштаб текста относительно другой точки, например, центра текста. Для этого в классе *ScaleTransform* предусмотрены свойства *CenterX* и *CenterY*. Для реализации масштабирования текста вокруг его центра можно вычислить размер текста (или получить его в коде посредством свойств *ActualWidth* и *ActualHeight TextBlock*), разделить полученные значения на 2 и задать результаты в качестве значений *CenterX* и *CenterY*. В проекте TransformExperiment задайте для текстовой строки 96 и 13, соответственно. Теперь масштабирование будет выполняться относительно центра.

Но существует и намного более простой путь. У самого *TextBlock* есть свойство *RenderTansformOrigin* (Центр трансформации визуального представления), которое он наследует от *UIElement*. Это свойство определяет точку в относительной системе координат, где (0, 0) соответствует верхнему левому углу, (1, 1) нижнему правому углу, и (0.5, 0.5) – центру. Верните значения *CenterX* и *CenterY* в *ScaleTransform* к исходному 0 и задайте *RenderTransformOrigin* в *TextBlock* следующим образом:

```
RenderTransformOrigin="0.5 0.5"
```

Не будем менять это значение *RenderTransformOrigin*, вернем свойствам *ScaleX* и *ScaleY* класса *ScaleTransform* значения по умолчанию (1) и займемся *RotateTransform*. Как и масштабирование, вращение всегда выполняется относительно какой-то точки. Эту точку можно задать в абсолютных единицах измерения (т.е. пикселах) относительно вращаемого объекта посредством свойств *CenterX* и *CenterY* или использовать относительные координаты через *RenderTransformOrigin*. Свойство *Angle* (Угол) измеряется в градусах, положительные значения означают вращение по часовой стрелке. На рисунке представлено вращение вокруг центра на 45 градусов.



Трансформацию *SkewTransform* сложно описать, но легко продемонстрировать. На рисунке представлен эффект, получаемый в результате задания свойству *AngleX* значения 30 градусов:

Смещение координат *X* вправо осуществляется на основании значений *Y*, так что по мере увеличения *Y* (в направлении вниз) значения *X* также растут. Применяя отрицательные значения углов можно моделировать наклонный (подобный курсиву) текст. *AngleY* обусловливает смещение текста в вертикальном направлении на основании увеличения координат X. В данном примере *AngleY* задано значение 30 градусов:

Все трансформации, наследуемые от *Transform*, являются аффинными («не бесконечными») преобразованиями. Это означает, что прямоугольник никогда не сможет быть трансформирован во что-то более чем параллелограмм.

Нетрудно убедиться, что порядок трансформаций имеет значение. Например, в проекте TransformExperiment задайте свойствам *ScaleX* и *ScaleY* класса *ScaleTransform* значение 4 и свойствам *X* и *Y* класса *TranslateTransform* значение 100. Таким образом, текст увеличивается в 4 раза и затем переносится на 100 пикселов. Теперь поставьте описание *TranslateTransform* перед *ScaleTransform*. Текст сначала переносится на 100 пикселов и затем масштабируется, но масштаб применяется и к исходным коэффициентам переноса тоже, т.е. текст фактически переносится на 400 пикселов.

Порой удобно поместить *Transform* в *Style*, как сделано в данном фрагменте:

```
<Setter Property="RenderTransform">
    <Setter.Value>
        <TranslateTransform />
    </Setter.Value>
</Setter>
```

Это позволяет выполнять трансформации из кода. Но будьте осторожны: ресурсы используются совместно. Будет существовать только один экземпляр *TranslateTransform*, который будет использоваться совместно всеми элементами, работающими со *Style*. Таким образом, изменение параметров трансформации для одного элемента повлечет за собой изменение всех элементов! Если это то, что требуется, совместное использование трансформации через *Style* – идеальное решение.

Все рассмотренные трансформации можно объединить в том порядке, в каком они применялись в TransformExperiment (масштабирование, наклонение, вращение, перенос), в одном удобном классе *CompositeTransform*.

Давайте напишем приложение, реализующее часы. Это будут не электронные часы, но и не просто часы со стрелками, поэтому я назову их гибридными часами (HybridClock). Часовая, минутная и секундная стрелки реализованы посредством объектов *TextBlock*, вращающихся вокруг центра сетки для содержимого. Рассмотрим XAML:

Проект Silverlight: HybridClock Файл: MainPage.xaml (фрагмент)

```
<Grid Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      SizeChanged="OnContentPanelSizeChanged">
    <TextBlock Name="referenceText"
               Text="THE SECONDS ARE 99"
               Foreground="Transparent" />

    <TextBlock Name="hourHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Name="minuteHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Name="secondHand">
        <TextBlock.RenderTransform>
            <CompositeTransform />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Обратите внимание на обработчик события *SizeChanged* в *Grid*. Файл выделенного кода будет использовать его для корректировки вычислений соответственно размеру *Grid*, который будет зависеть от ориентации.

В данном *Grid* располагается четыре элемента *TextBlock*. Первый *TextBlock* прозрачный и используется только в коде для определения размеров. Остальные три элемента *TextBlock* наследуют свой цвет посредством наследования свойств, и в качестве значений их свойств *RenderTransform* заданы объекты по умолчанию *CompositeTransform*. В файле выделенного кода описаны несколько полей, они будут использоваться в приложении. В конструкторе создается *DispatcherTimer*, для использования которого посредством директивы *using* необходимо подключить пространство имен *System.Windows.Threading*.

Проект Silverlight: HybridClock Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
```

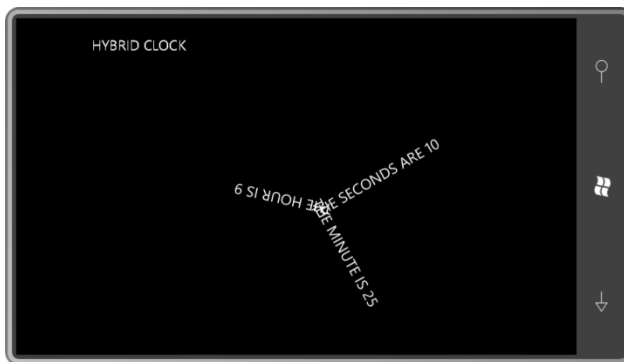
```
    Point gridCenter;
    Size textSize;
    double scale;

    public MainPage()
    {
        InitializeComponent();

        DispatcherTimer tmr = new DispatcherTimer();
        tmr.Interval = TimeSpan.FromSeconds(1);
        tmr.Tick += OnTimerTick;
        tmr.Start();
    }

    void OnContentPanelSizeChanged(object sender, SizeChangedEventArgs args)
    {
        gridCenter = new Point(args.NewSize.Width / 2,
                               args.NewSize.Height / 2);

        textSize = new Size(referenceText.ActualWidth,
                            referenceText.ActualHeight);

        scale = Math.Min(gridCenter.X, gridCenter.Y) / textSize.Width;

        UpdateClock();
    }

    void OnTimerTick(object sender, EventArgs e)
    {
        UpdateClock();
    }

    void UpdateClock()
    {
        DateTime dt = DateTime.Now;
        double angle = 6 * dt.Second;
        SetupHand(secondHand, "THE SECONDS ARE " + dt.Second, angle);
        angle = 6 * dt.Minute + angle / 60;
        SetupHand(minuteHand, "THE MINUTE IS " + dt.Minute, angle);
        angle = 30 * (dt.Hour % 12) + angle / 12;
        SetupHand(hourHand, "THE HOUR IS " + (((dt.Hour + 11) % 12) + 1), angle);
    }

    void SetupHand(TextBlock txtblk, string text, double angle)
    {
        txtblk.Text = text;
        CompositeTransform xform = txtblk.RenderTransform as CompositeTransform;
        xform.CenterX = textSize.Height / 2;
        xform.CenterY = textSize.Height / 2;
        xform.ScaleX = scale;
        xform.ScaleY = scale;
        xform.Rotation = angle - 90;
        xform.TranslateX = gridCenter.X - textSize.Height / 2;
        xform.TranslateY = gridCenter.Y - textSize.Height / 2;
    }
}
```

С помощью метода-обработчика *SizeChanged* HybridClock определяет центр панели для содержимого (*ContentPanel*) и размер *TextBlock* под именем *referenceText* (Текст для справки). (Второй элемент не будет меняться в ходе всего выполнения приложения.) На основании этих двух элементов приложение вычисляет коэффициент масштабирования, посредством которого будет обеспечено соответствие ширины *referenceText* половине меньшего размера *Grid* и пропорциональные размеры остальных элементов *TextBlock*.

В методе обратного вызова для таймера определяется текущее время и вычисляются значения углов поворота для секундной, минутной и часовой стрелок (углы отсчитываются относительно положения, какое стрелки занимают в полдень). Все остальные операции выполняются в *SetupHand* (Настроить стрелку) каждой стрелки в отдельности.

От *CompositeTransform* требуется выполнить несколько операций. В ходе переноса элементы *TextBlock* должны быть перемещены так, чтобы начало текста располагалось в центре *Grid*. Но я не хочу, чтобы верхний левый угол текста находился в центре. Я хочу найти точку, которая смещена под данным углом на половину высоты текста. Для этого используем свойства *TranslateX* и *TranslateY*. В *CompositeTransform* перенос применяется последним, поэтому я поместил эти свойства в конец метода, даже несмотря на то, что порядок их задания не имеет значения.

В качестве значения свойств *ScaleX* и *ScaleY* задается вычисленный ранее коэффициент масштабирования. Значение параметра *angle*, передаваемого в метод, определено относительно положения, занимаемого стрелками в полдень, но элементы *TextBlock* располагаются соответственно 3:00. Поэтому *Rotation* добавляет к *angle* смещение в –90 градусов. И масштабирование, и вращение выполняются относительно *CenterX* и *CenterY*. Эти координаты определяют точку на левом краю текста, смещенную относительно верхнего левого угла на половину высоты текста вниз. На рисунке показано, как часы выглядят в действии:



Windows Phone поддерживает также трансформацию *проекции*. Эта трансформация была введена в Silverlight 3, но используется практически исключительно в связи с анимациями, поэтому отложим ее рассмотрение до главы 15.

Анимация со скоростью видео

Вызов *DispatcherTimer* с интервалом в одну секунду имеет смысл в приложении HybridClock, потому что положения стрелок часов должны обновляться не чаще, чем каждую секунду. Но с переходом к плавно перемещающейся секундной стрелке сразу же возникает вопрос, как часто необходимо обновлять стрелки часов. Учитывая то, что секундная стрелка должна перемещаться лишь на несколько пикселов в секунду, вероятно, установить таймер на 250 миллисекунд было бы приемлемым, а на 100 миллисекунд – более чем достаточно.

Не лишним будет напомнить, что экран устройств, работающих под управлением Windows Phone 7, обновляется примерно 30 раз в секунду или каждые 33-1/3 миллисекунды. Поэтому использовать для анимации таймер с тактовой частотой выше, чем раз в 33-1/3 миллисекунды, вообще не имеет смысла.

Идеальным вариантом для анимации является таймер, синхронный с частотой обновления монитора. Silverlight предоставляет такую возможность посредством одного очень простого

в использовании события *CompositionTarget.Rendering*. Обработчик этого события выглядит следующим образом:

```
void OnCompositionTargetRendering(object sender, EventArgs args)
{
    TimeSpan renderingTime = (args as RenderingEventArgs).RenderingTime;
    ...
}
```

Несмотря на то, что обработчик события должен быть определен аргументом *EventArgs* (Аргументы события), в качестве аргумента фактически выступает объект *RenderingEventArgs* (Аргументы события формирования визуального представления). В результате приведения аргумента к *RenderingEventArgs* получаем объект *TimeSpan* (Диапазон времени), указывающий на время, прошедшее с момента запуска приложения.

CompositionTarget (Композиция приложения) – это статический класс с единственным открытым членом, которым является событие *Rendering*. Добавляем обработчик события следующим образом:

```
CompositionTarget.Rendering += OnCompositionTargetRendering;
```

Если создаваемое приложение не перегружено анимацией, нет необходимости сохранять обработчик события в ходе всего выполнения приложения, поэтому удалим его сразу же после использования:

```
CompositionTarget.Rendering -= OnCompositionTargetRendering;
```

В проекте RotatingText (Вращающийся текст) элемент *TextBlock* располагается в центре сетки для содержимого:

Проект: RotatingText Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="ROTATE!"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate" />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
```

Обратите внимание на атрибут *x:Name* в *RotateTransform*. Здесь не может использоваться *Name*, потому что он определен в *FrameworkElement*. В конструкторе в файле выделенного кода подписываемся на событие *CompositionTarget.Rendering*:

Проект: RotatingText Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    TimeSpan startTime;

    public MainPage()
    {
        InitializeComponent();
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, EventArgs args)
```

```
    {
        TimeSpan renderingTime = (args as RenderingEventArgs).RenderingTime;

        if (startTime.Ticks == 0)
        {
            startTime = renderingTime;
        }
        else
        {
            TimeSpan elapsedTime = renderingTime - startTime;
            rotate.Angle = 180 * elapsedTime.TotalSeconds % 360;
        }
    }
}
```

Обработчик события использует *renderingTime* для отрисовки анимации, т.е. наш текст делает один оборот за две секунды.

Для реализации подобных простых повторяющихся анимаций предпочтительнее использовать не *CompositionTarget.Rendering*, а встроенные возможности анимации Silverlight (которые мы обсудим в главе 15).

Обработка событий манипуляций

Трансформации также являются хорошим способом обработки событий манипуляций. В данном фрагменте описан мяч, располагающийся в центре сетки для содержимого:

Проект Silverlight: DragAndScale Файл: Page.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Ellipse Width="200"
             Height="200"
             RenderTransformOrigin="0.5 0.5"
             ManipulationDelta="OnEllipseManipulationDelta">
        <Ellipse.Fill>
            <RadialGradientBrush Center="0.4 0.4"
                                 GradientOrigin="0.4 0.4">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="{StaticResource PhoneAccentColor}"
/>
            </RadialGradientBrush>
        </Ellipse.Fill>

        <Ellipse.RenderTransform>
            <CompositeTransform  />
        </Ellipse.RenderTransform>
    </Ellipse>
</Grid>
```

Обратите внимание на *CompositeTransform*. У этого класса нет имени, поэтому в коде на него придется ссылаться через элемент *Ellipse*. (Такая стратегия хороша в случае использования одного обработчика события для нескольких элементов.)

В файле выделенного кода обрабатывается только событие *ManipulationDelta* от *Ellipse*:

```
void OnEllipseManipulationDelta(object sender, ManipulationDeltaEventArgs args)
{
    Ellipse ellipse = sender as Ellipse;
    CompositeTransform xform = ellipse.RenderTransform as CompositeTransform;

    if (args.DeltaManipulation.Scale.X > 0 || args.DeltaManipulation.Scale.Y > 0)
```

```
    {
        double maxScale = Math.Max(args.DeltaManipulation.Scale.X,
                                   args.DeltaManipulation.Scale.Y);
        xform.ScaleX *= maxScale;
        xform.ScaleY *= maxScale;
    }

    xform.TranslateX += args.DeltaManipulation.Translation.X;
    xform.TranslateY += args.DeltaManipulation.Translation.Y;

    args.Handled = true;
}
```

Событие *ManipulationDelta* играет критически важную роль при обработке любых манипуляций, сложнее чем одиночное касание. Это событие консолидирует сведения об одном или более касании элемента в данные трансформации переноса и масштабирования. Класс *ManipulationDeltaEventArgs* (Аргументы события приращения в ходе манипуляции) включает два свойства: *CumulativeManipulation* (Совокупная манипуляция) и *DeltaManipulation* (Манипуляция приращением). Оба эти свойства типа *ManipulationDelta*, у которого есть два свойства: *Translation* (Перенос) и *Scale* (Масштаб).

Зачастую использовать *DeltaManipulation* проще, чем *CumulativeManipulation*. Если манипуляции с элементом осуществляются посредством касания в одной точке, действительные значения имеют только коэффициенты *Translation*, и они могут быть использованы только как значения *TranslateX* и *TranslateY* свойств класса *CompositeTransform*. Если касание экрана происходит одновременно в двух точках, значения *Scale* отличны от нуля. Они могут быть отрицательными и часто неодинаковые. Чтобы круг оставался кругом, я использую максимальное из этих значений и умножаю его на существующие коэффициенты масштабирования трансформации. Так реализовываются операции «сведение» и «растяжение».

XAML-файл устанавливает центр трансформации в центре эллипса. Теоретически он должен определяться на основании положения и перемещений двух касаний, но это довольно сложная задача.

Элемент *Border*

В *TextBlock* не предусмотрена рамка, которой можно было бы обвести текст. К счастью в Silverlight есть элемент *Border*, и он может использоваться для создания рамок вокруг *TextBlock* или любого другого элемента. *Border* включает свойство *Child* (Потомок) типа *UIElement*. Это означает, что в *Border* может быть помещен лишь один элемент, но этим элементом может быть панель, в которой может располагаться множество элементов.

Загрузите в Visual Studio приложение XamlExperiment из предыдущей главы и поместите *TextBlock* в *Border* следующим образом:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25">
        <Border.Child>
            <TextBlock Text="Hello, Windows Phone 7!" />
        </Border.Child>
    </Border>
</Grid>
```

Свойство *Child* является атрибутом *ContentProperty* элемента *Border*, поэтому теги *Border.Child* не нужны. Если свойства *HorizontalAlignment* и *VerticalAlignment* не заданы, элемент *Border*

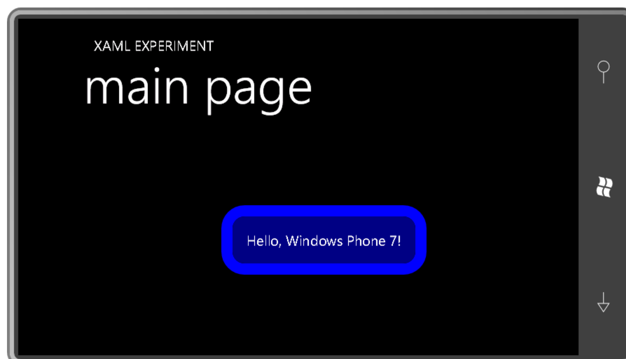
занимает всю область *Grid*; и *TextBlock* занимает всю область *Border*, даже если сам текст располагается в верхнем левом углу. *TextBlock* в *Border* можно центрировать:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25">
        <TextBlock Text="Hello, Windows Phone 7!"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />
    </Border>
</Grid>
```

Или можно центрировать *Border* в *Grid*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
        <TextBlock Text="Hello, Windows Phone 7!" />
    </Border>
</Grid>
```

Здесь *Border* уменьшается до размера, достаточного лишь, чтобы вместить *TextBlock*. Также можно задать свойства *HorizontalAlignment* и *VerticalAlignment* для *TextBlock*, но сейчас это не возымеет никакого действия. Немного увеличить область внутри рамки можно, задавая свойства *Margin* или *Padding* для *TextBlock* или свойство *Padding* самого *Border*:



Теперь наш *TextBlock* обведен симпатичной рамкой. Свойство *BorderThickness* (Толщина рамки) типа *Thickness* (эта же структура используется для *Margin* или *Padding*), что обеспечивает возможность сделать толщину рамки разной со всех сторон. Свойство *CornerRadius* (Радиус скругления углов) типа *CornerRadius*. *CornerRadius* – это структура, которая также позволяет задавать четыре разных значения для четырех углов. Свойства типа *Background* и *BorderBrush*, что позволяет использовать градиентные кисти.

Если требуется получить *Border* «нормальной» толщины, можно использовать предопределенные ресурсы:

```
<Border BorderThickness="{StaticResource PhoneBorderThickness}"
```

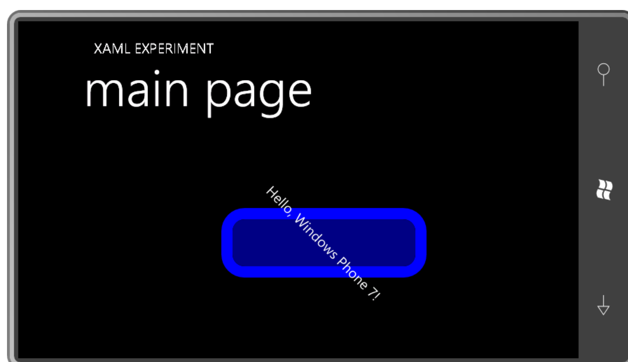
В данном случае получаем рамку толщиной в 3 пиксела. Ресурс *PhoneStrokeThickness* (Толщина обводки в телефоне) обеспечивает такое же значение.

Что произойдет, если задать *RenderTransform* для *TextBlock*? Попробуем:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="Navy"
```

```
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Padding="20">
        <TextBlock Text="Hello, Windows Phone 7!"
                   RenderTransformOrigin="0.5 0.5">
            <TextBlock.RenderTransform>
                <RotateTransform Angle="45" />
            </TextBlock.RenderTransform>
        </TextBlock>
    </Border>
</Grid>
```

Вот что мы получаем:

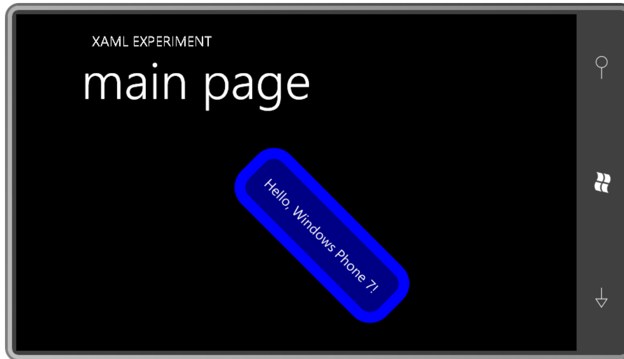


Свойство *RenderTransform* называется трансформацией *визуального представления* не просто так: оно оказывает влияние только на формирование визуального представления и *не* касается того, как элемент рассматривается в системе компоновки. (В Windows Presentation Foundation есть второе свойство под именем *LayoutTransform* (Трансформация компоновки), которое изменяет компоновку. Если бы мы создавали код в WPF и использовали в данном случае *LayoutTransform*, *Border* расширился бы, чтобы вместить развернутый текст, но при этом сам не разворачивался бы. Однако в Silverlight пока нет свойства *LayoutTransform*, хотя, как видите, иногда его явно не хватает.)

Но не стоит падать духом! Перенесем *RenderTransform* (и *RenderTransformOrigin*) из *TextBlock* в *Border*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="Navy"
            BorderBrush="Blue"
            BorderThickness="16"
            CornerRadius="25"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Padding="20"
            RenderTransformOrigin="0.5 0.5">
        <Border.RenderTransform>
            <RotateTransform Angle="45" />
        </Border.RenderTransform>

        <TextBlock Text="Hello, Windows Phone 7!" />
    </Border>
</Grid>
```

Теперь трансформирован не только элемент, к которому применили трансформацию, но и все его дочерние элементы, что отчетливо демонстрирует данный снимок экрана:



Это означает, что трансформации могут применяться к целым разделам дерева визуальных элементов, и внутри этого трансформированного дерева можно использовать дополнительные трансформации.

Свойства и строковые элементы *TextBlock*

Мы обсуждаем *TextBlock* с самых первых страниц данной книги, и, наконец, пришло время рассмотреть его более подробно. Элемент *TextBlock* включает пять свойств для задания шрифтов: *FontFamily*, *FontSize*, *FontStretch*, *FontStyle*, and *FontWeight*.

Как было показано ранее, свойство *FontStyle* может принимать значения *Normal* или *Italic*. Теоретически свойству *FontStretch* можно задать значения *Condensed* (Уплотненный) и *Expanded* (Разреженный), но я никогда не встречал, чтобы это работало в Silverlight. Как правило, для *FontWeight* задают значения *Normal* или *Bold*, хотя существуют и другие варианты: *Black* (Темный), *SemiBold* (Полужирный) и *Light* (Светлый).

У *TextBlock* также имеется свойство *TextDecorations* (Украшения шрифта). Это свойство кажется очень обобщенным, но в Silverlight для него предусмотрено только одно значение:

```
TextDecorations="Underline"
```

Наиболее часто используемым свойством *TextBlock*, безусловно, является *Text*. Строка, задаваемая как значение свойства *Text*, может включать встроенные символы Unicode в стандартном XML-формате, например:

```
Text="&#x03C0; is approximately 3.14159"
```

Если значением свойства *Text* является очень длинная строка, она может не поместиться на экране устройства. Для форматирования такой строки предусмотрены символы возврата каретки или перевода строки (
 или
) либо можно задать

```
TextWrapping="Wrap"
```

и присвоить *TextAlignment* (Выравнивание текста) значения *Left*, *Right* или *Center* (но не *Justify* (Выровнять по ширине)). Также можно задать текст как содержимое элемента *TextBlock*:

```
<TextBlock>
    Некоторый текст.
</TextBlock>
```

Может показаться удивительным, но атрибут *ContentProperty* элемента *TextBlock* не является свойством *Text*. Это совершенно отдельное свойство под именем *Inlines* (Строковые элементы). Свойство *Inlines* типа *InlineCollection* (Коллекция строковых элементов). *InlineCollection* – это коллекция объектов типа *Inline*, а именно *LineBreak* (Разрыв строки) и *Run* (Переход). Их применение делает *TextBlock* намного более универсальным. Использовать *LineBreak* не сложно:

```
<TextBlock>
    Некоторый текст<LineBreak />И еще немного текста.
</TextBlock>
```

Особый интерес представляет *Run*, потому что у него также есть собственные свойства *FontFamily*, *FontSize*, *FontStretch*, *FontStyle*, *FontWeight*, *Foreground* и *TextDecorations*. Это позволяет форматировать текст любым самым причудливым образом:

```
<TextBlock FontSize="36"
           TextWrapping="Wrap">
    This is
    some <Run FontWeight="Bold">bold</Run> text and
    some <Run FontStyle="Italic">italic</Run> text and
    some <Run Foreground="Red">red</Run> text and
    some <Run TextDecorations="Underline">underlined</Run> text
    and some <Run FontWeight="Bold"
                  FontStyle="Italic"
                  Foreground="Cyan"
                  FontSize="72"
                  TextDecorations="Underline">big</Run> text.
</TextBlock>
```

В дизайнере Visual Studio текст, заключенный в теги *Run*, визуально может не отличаться от текста, находящегося вне тегов *Run*. Это ошибка. Если запустить приложение на эмуляторе, все будет выглядеть нормально:

Все это векторные шрифты TrueType. Перед растеризацией символов векторы шрифта масштабируются соответственно заданному размеру шрифта, поэтому независимо от размера символы всегда выглядят сглаженными.

Несмотря на массу преимуществ, *TextBlock* не обеспечивает всех возможностей, необходимых для отображения абзаца текста, какие можно найти в классе *Paragraph* (Абзац) (например, отступа или выступа первой строки). Мне не известен способ реализации выступа первой строки, но отступ создать довольно просто, и я продемонстрирую это в следующей главе.

Использование свойства *Inlines* позволяет нам создать приложение для демонстрации всех возможностей свойства *FontFamily*. В XAML мы можем присвоить *FontFamily* строковое значение. (В коде пришлось бы создавать экземпляр класса *FontFamily*.) По умолчанию применяется шрифт «Portable User Interface». На эмуляторе телефона этот стандартный шрифт проецируется в Segoe WP, разновидность шрифта Segoe для Windows Phone, часто используемый в продуктах Майкрософт и печатных материалах, в том числе и в этой книге.

В приложении FontFamilies перечислены все значения *FontFamily*, предлагаемые системой Intellisense в Visual Studio:

Проект Silverlight: FontFamilies Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock FontSize="24">
        <Run FontFamily="Arial">Arial</Run><LineBreak />
        <Run FontFamily="Arial Black">Arial Black</Run><LineBreak />
        <Run FontFamily="Calibri">Calibri</Run><LineBreak />
        <Run FontFamily="Comic Sans MS">Comic Sans MS</Run><LineBreak />
        <Run FontFamily="Courier New">Courier New</Run><LineBreak />
        <Run FontFamily="Georgia">Georgia</Run><LineBreak />
        <Run FontFamily="Lucida Sans Unicode">Lucida Sans Unicode</Run><LineBreak />
        <Run FontFamily="Portable User Interface">Portable User
Interface</Run><LineBreak />
        <Run FontFamily="Segoe WP">Segoe WP</Run><LineBreak />
        <Run FontFamily="Segoe WP Black">Segoe WP Black</Run><LineBreak />
        <Run FontFamily="Segoe WP Bold">Segoe WP Bold</Run><LineBreak />
        <Run FontFamily="Segoe WP Light">Segoe WP Light</Run><LineBreak />
        <Run FontFamily="Segoe WP Semibold">Segoe WP Semibold</Run><LineBreak />
        <Run FontFamily="Segoe WP SemiLight">Segoe WP SemiLight</Run><LineBreak />
        <Run FontFamily="Tahoma">Tahoma</Run><LineBreak />
        <Run FontFamily="Times New Roman">Times New Roman</Run><LineBreak />
        <Run FontFamily="Trebuchet MS">Trebuchet MS</Run><LineBreak />
        <Run FontFamily="Verdana">Verdana</Run><LineBreak />
        <Run FontFamily="Webdings">Webdings</Run> (Webdings)
    </TextBlock>
</Grid>
```

Вот что мы получаем:

Если сделать ошибку в написании имени шрифта, присваиваемого *FontFamily*, ничего страшного, просто будет использован шрифт по умолчанию.

Предопределенные ресурсы включают четыре ключа, которые возвращают объекты типа *FontFamily*: PhoneFontFamilyNormal, PhoneFontFamilyLight, PhoneFontFamilySemiLight и PhoneFontFamilySemiBold. Они возвращают соответствующие шрифты Segoe WP.

Изображения более подробно

Как было показано в главе 4, элемент *Image* позволяет использовать в приложении на Silverlight растровые изображения в формате JPEG и PNG. Рассмотрим элемент *Image* более подробно.

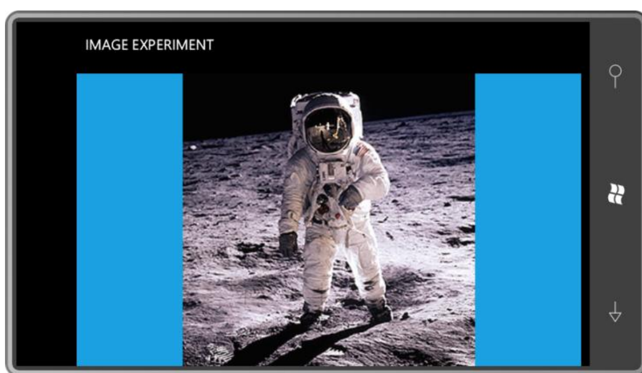
Проект ImageExperiment включает папку Images. В этой папке имеется файл BuzzAldrinOnTheMoon.png. Это популярная фотография, сделанная Нилом Армстронгом 21 июля 1969 года с помощью фотоаппарата Hasselblad. Это изображение размером 288x288 пикселов.

Ссылаемся на файл изображения в MainPage.xaml следующим образом:

Проект Silverlight: ImageExperiment Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      Background="{StaticResource PhoneAccentBrush}">
    <Image Source="Images/BuzzAldrinOnTheMoon.png" />
</Grid>
```

Также для сетки для содержимого я задал кисть *Background* контрастного цвета, просто чтобы фото выглядело несколько отчетливее. Вот как оно отображается в альбомном режиме:



По умолчанию растровое изображение меняет свои размеры, пытаясь максимально занять всю площадь контейнера (в данном случае, сетки для содержимого), но при этом сохраняя собственные пропорции. В зависимости от размеров и пропорций контейнера изображение центрируется по вертикали или по горизонтали. Варьируя значениями свойств *HorizontalAlignment* и *VerticalAlignment*, изображение можно перемещать в ту или иную сторону.

Поведением растяжения управляет свойство *Stretch* элемента *Image*. В качестве значений этого свойства используются члены перечисления *Stretch*. Значением по умолчанию является *Uniform* (Равномерно), которое можно задать явно следующим образом:

```
<Image Source="Images/BuzzAldrinOnTheMoon.png"
       Stretch="Uniform" />
```

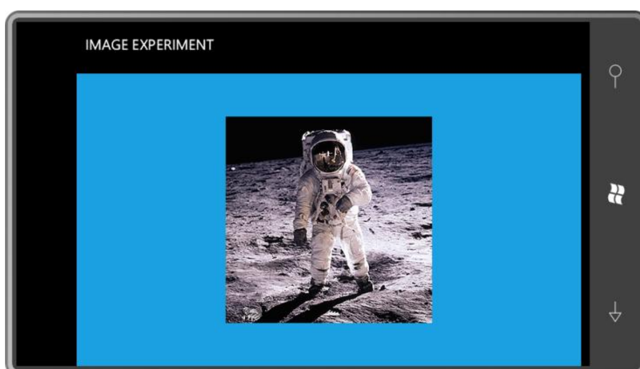
«Равномерно» в данном случае означает одинаково во обоих направлениях, чтобы не нарушить пропорции изображения.

Также свойству *Stretch* можно задать значение *Fill* (Заполнить). Тогда изображение заполнит весь контейнер без сохранения собственных пропорций.

Компромиссным вариантом является значение *UniformToFill* (Равномерно, чтобы заполнить):

Теперь изображение и заполняет контейнер полностью, и растянуто равномерно с сохранением пропорций. Как обеспечивается одновременное выполнение обеих задач? Вообще это возможно только за счет обрезки изображения. Задавая свойства *HorizontalAlignment* и *VerticalAlignment*, можно определить, какой край изображения будет обрезан. То какие параметры используются, зависит от конкретного изображения.

Четвертое возможное значение – *None* (Нет). Оно определяет отсутствие растяжения. Изображение будет выведено в исходном размере, в данном случае, 288x288 пикселов.



Если при выводе изображения требуется обеспечить конкретный размер с сохранением исходных пропорций, задайте явно свойство *Width* или *Height*. Чтобы растянуть изображение до определенного размера без сохранения пропорций, задайте оба свойства, *Width* и *Height*, а также используйте для *Stretch* значение *Fill*.

Применять трансформации к элементу *Image* так же просто, как мы делали это для элементов *TextBlock*:

```
<Image Source="Images/BuzzAldrinOnTheMoon.png"
       RenderTransformOrigin="0.5 0.5">
    <Image.RenderTransform>
```

```
        <RotateTransform Angle="30" />
    </Image.RenderTransform>
</Image>
```

Вот что получилось:

Воспроизведение фильмов

Воспроизвести фильм практически так же просто, как и вывести изображение. Но из-за их большого размера видеофайлы практически никогда не включаются в исполняемые файлы, и практически всегда воспроизведение осуществляется через Веб-подключение. Давайте воспроизведем фильм с моего Веб-сайта. Для этого в проекте ImageExperiment заменим элемент *Image* элементом *MediaElement* (Элемент мультимедиа):

```
<Grid x:Name="ContentGrid" Grid.Row="1" Margin="12,0,12,0"
      Background="{StaticResource PhoneAccentBrush}">
    <MediaElement Source=http://www.charlespetzold.com/Media/Walrus.wmv />
</Grid>
```

Элемент *MediaElement* имеет свойство *AutoPlay* (Автоматическое воспроизведение). Его значением по умолчанию является *true*, т.е. воспроизведение начинается сразу же, как только в буфер загружается достаточный объем видео.

В главе 10 я покажу, как использовать кнопки вместе с *MediaPlayer*, что позволяет управлять воспроизведением фильмов, как на DVD-проигрывателе.

Режимы прозрачности

UIElement определяет свойство *Opacity* (Непрозрачность), которое может принимать значения от 0 до 1, обеспечивая элементу (и его потомкам) разные степени прозрачности. Но несколько более интересным является свойство *OpacityMask* (Маска прозрачности), которое позволяет «скрыть» часть элемента. Значением *OpacityMask* является объект типа *Brush*; чаще всего используются один из двух производных от *GradientBrush* классов. Цвет кисти игнорируется, для определения прозрачности используется только значение альфа-канала.

Например, к свойству *OpacityMask* элемента *Image* можно применить *RadialGradientBrush*:

```
<Image Source="Images/BuzzAldrinOnTheMoon.png">
    <Image.OpacityMask>
        <RadialGradientBrush>
            <GradientStop Offset="0" Color="White" />
            <GradientStop Offset="0.8" Color="White" />
            <GradientStop Offset="1" Color="Transparent" />
        </RadialGradientBrush>
    </Image.OpacityMask>
</Image>
```

Обратите внимание, что *RadialGradientBrush* непрозрачный в центре и остается непрозрачным до 0,8 радиуса, после чего плавно становится полностью прозрачным к краям круга. Получаем очень интересный эффект, глядя на который не верится, что его можно было создать всего с помощью нескольких строк XAML:

Рассмотрим популярную методику, в которой используются два одинаковых элемента, но к одному из них применяется *ScaleTransform*, чтобы перевернуть его, и *OpacityMask*, чтобы сделать его частично размытым:

```
<Image Source="Images/BuzzAldrinOnTheMoon.png"
       Stretch="None"
       VerticalAlignment="Top" />
<Image Source="Images/BuzzAldrinOnTheMoon.png"
       Stretch="None"
       VerticalAlignment="Top"
       RenderTransformOrigin="0.5 1">
    <Image.RenderTransform>
        <ScaleTransform ScaleY="-1" />
    </Image.RenderTransform>
    <Image.OpacityMask>
        <LinearGradientBrush StartPoint="0 0" EndPoint="0 1">
            <GradientStop Offset="0" Color="#00000000" />
            <GradientStop Offset="1" Color="#40000000" />
        </LinearGradientBrush>
    </Image.OpacityMask>
</Image>
```

Эти два элемента одинакового размера и выровнены по верху и центру. Обычно второй размещается над первым. Но для второго элемента задан *RenderTransform* со значением *ScaleTransform*, что переворачивает изображение относительно горизонтальной оси. *RenderTransformOrigin* задано значение (0.5, 1), что соответствует низу элемента. Таким образом, изображение переворачивается относительно нижнего края. После этого к свойству *OpacityMask* применяется *LinearGradientBrush*, что обеспечивает размытие перевернутого изображения:

Заметьте, что значения *GradientStop* применяются к исходному «неотраженному» изображению, поэтому верхняя часть картинки полностью прозрачная (что соответствует значению #00000000) и затем она отражается в нижнюю часть составного изображения.

Часто вот такие несложные приемы позволяют визуальным элементам заиграть в новом свете для пользователя. Но не стоит злоупотреблять *OpacityMask*, особенно в сочетании со сложными анимациями, потому что это может иметь негативное влияние на производительность. Возьмите за правило использовать *OpacityMask*, только если создаваемый эффект действительно того стоит.

Мозаичные кисти, не создающие мозаики

В данной книге уже были представлены примеры *SolidColorBrush*, *LinearGradientBrush* и *RadialGradientBrush*. Рассмотрим полную иерархию потомков класса *Brush*:

Object
- *DependencyObject* (абстрактный)
  - *Brush* (абстрактный)
    - *SolidColorBrush* (запечатанный[1])
    - *GradientBrush* (абстрактный)
      - *LinearGradientBrush* (запечатанный)
      - *RadialGradientBrush* (запечатанный)
    - *TileBrush* (абстрактный)
      - *ImageBrush* (запечатанный)
      - *VideoBrush* (запечатанный)
    - *ImplicitInputBrush* (запечатанный)

Но Windows Phone 7 поддерживает еще одну кисть, *ImageBrush* (Кисть изображения), и, хотя этот класс наследуется от *TileBrush* (Мозаичная кисть), с его помощью невозможно создать мозаичный шаблон. (Это можно сделать в Windows Presentation Foundation, и, вероятно, когда-нибудь станет реальным и в Silverlight.) По сути, *ImageBrush* позволяет задавать для

[1] То есть от него невозможно создать класс-поток (*прим. научного редактора*).

растрового изображения любое свойство типа *Brush*. Вернемся к ImageExperiment, но вместо элемента *Image* применим *ImageBrush* для свойства *Background* сетки для содержимого.

```
<Grid x:Name="ContentGrid" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
        <ImageBrush ImageSource="Images/BuzzAldrinOnTheMoon.png" />
    </Grid.Background>
</Grid>
```

Как и *Image*, *TileBrush* определяет свойство *Stretch*, но его значением по умолчанию является *Fill*, поэтому изображение заполняет область без сохранения пропорций.

Глава 9
Вопросы компоновки

Одним из самых важных классов Silverlight является *Panel*. Этому классу отведена главная роль к системе компоновки Silverlight. Логичным было бы ожидать, что такой важный класс должен определять массу свойств и событий, но в *Panel* только три собственных свойства:

- *Background* типа *Brush*
- *Children* типа *UIElementCollection*
- *IsItemsHost* (Является хостом элементов) типа *bool*

С первым свойством все понятно, и третье является свойством только для чтения и касается роли класса в *ListBox* (Окно списка) и подобных классах.

Самым значимым является свойство *Children*. В предыдущей главе мы видели свойство *Child* типа *UIElement* в классе *Border*. Свойство *Children*, определяемое классом *Panel*, типа *UIElementCollection*. Громадная разница!

Единственный дочерний элемент в классе *Border* не требует принятия особо сложных решений. Он просто располагается внутри *Border* и больше ничего. Но панель может вмещать множество потомков и может делать это по-разному: размещать друг над другом или в сетке, или по краям, или по кругу, или отображать их веером, либо как карусель.

Поэтому сам класс *Panel* является абстрактным. Рассмотрим иерархию класса *Panel* со всеми производными от него классами:

Object
    DependencyObject (абстрактный)
        UIElement (абстрактный)
            FrameworkElement (абстрактный)
                Panel (абстрактный)
                    Canvas
                        InkPresenter (запечатанный)
                    Grid
                    StackPanel
                    VirtualizingPanel (абстрактный)
                        VirtualizingStackPanel
                    PanoramaPanel
                    MapLayerBase (абстрактный)
                        MapLayer (запечатанный)

Silverlight обеспечивает для Windows Phone три стандартных типа панелей: *StackPanel* (вероятно, самый простой вид панелей), *Grid* (основной вариант для наиболее типовых компоновок) и *Canvas* (Холст). *Canvas* в большинстве случаев следует избегать, однако, он имеет некоторые особые свойства, делающие его незаменимым в ряде ситуаций.

Набор инструментов Silverlight for Windows Phone Toolkit включает *WrapPanel* (Панель с переносом), очень подобную правой части Windows Explorer.

В следующей главе я покажу пример использования *InkPresenter* (Элемент отображения рукописного ввода). Опция *VirtualizingPanel* (Виртуализирующая панель) обсуждается в

главе 17 в связи с элементами управления списками. Все остальные (как следует из их имен) используются для специальных целей с элементами управления *Panorama* (Панорама) и *Map* (Карта).

Вы уже видели *Grid* и *StackPanel* в стандартном MainPage.xaml и, вероятно, догадались, что панели могут быть вложенными. Панели являются основными архитектурными элементами Silverlight-страницы.

Также можно создавать собственные панели. В данной главе мы рассмотрим основные моменты, а более сложными вопросами займемся в последующих главах.

Grid с одной ячейкой

Как правило, *Grid* состоит из строк и столбцов, но в предыдущих главах было продемонстрировано, что множество дочерних элементов могут размещаться в *Grid* с одной ячейкой. Приведу простой пример для справки:

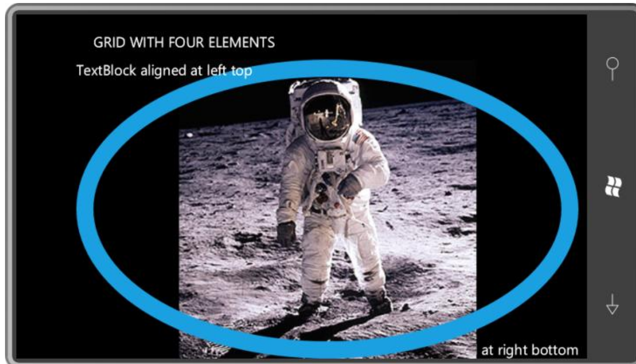
Проект Silverlight: GridWithFourElements Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="TextBlock aligned at right bottom"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom" />

    <Image Source="Images/BuzzAldrinOnTheMoon.png" />

    <Ellipse Stroke="{StaticResource PhoneAccentBrush}"
             StrokeThickness="24" />

    <TextBlock Text="TextBlock aligned at left top"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />
</Grid>
```

Каждый из четырех элементов имеет в своем распоряжении всю область содержимого:

Что касается размера, для всех элементов он немного разный. Размеры двух элементов *TextBlock* зависят от отображаемого ими текста и размера шрифта. Растровое изображение в элементе *Image* старается максимально заполнить область, предоставляемую *Grid*, сохранив при этом собственные пропорции. *Ellipse* просто расползается, занимая все доступное пространство.

Элементы перекрывают друг друга в том порядке, в котором они появляются в разметке, поскольку в этом порядке они добавляются в коллекцию *Children* элемента *Grid*. Я задал свойству *SupportedOrientations* страницы значение *PortraitOrLandscape*, чтобы позволяет понаблюдать за перестановкой элементов при изменении ориентации телефона:



Элемент *StackPanel*

Теперь поместим те же четыре элемента в *StackPanel*, который вложен в сетку для содержимого:

Проект Silverlight: StackPanelWithFourElements Файл: MainPage.xaml (фрагмент)

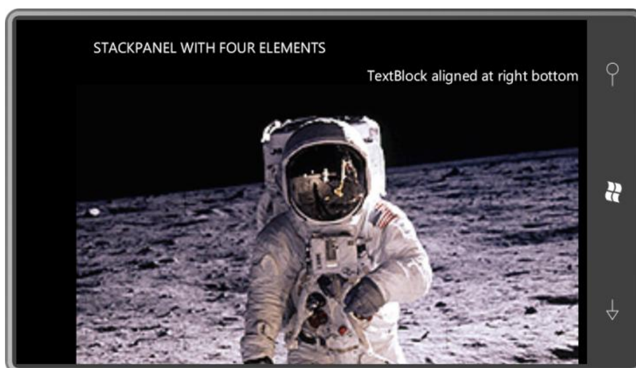
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Name="stackPanel"
                Orientation="Vertical">
        <TextBlock Text="TextBlock aligned at right bottom"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom" />

        <Image Source="Images/BuzzAldrinOnTheMoon.png" />

        <Ellipse Stroke="{StaticResource PhoneAccentBrush}"
                 StrokeThickness="12" />

        <TextBlock Text="TextBlock aligned at left top"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top" />
    </StackPanel>
</Grid>
```

По умолчанию дочерние элементы располагаются в *StackPanel* друг над другом сверху вниз. Дочерние элементы не перекрываются:

Теперь текст, отображаемый двумя элементами *TextBlock*, выглядит несколько странно. Первый *TextBlock* располагается вверху экрана, потому что является первым элементом в коллекции *Children*. Свойство *HorizontalAlignment* обеспечивает его смещение вправо, а вот свойство *VerticalAlignment* (для которого задано значение *Bottom*) явно проигнорировано. Аналогичная ситуация и со вторым *TextBlock*. Элемент *Image* занимает всю область *StackPanel* по ширине, но сохраняет пропорции изображения. Теперь для отображения фотографии целиком требуется лишь, чтобы по вертикали было достаточно пространства для этого.

Элементы *TextBlock* и *Image* занимают минимально необходимое им пространство по вертикали, и *Ellipse*… *Ellipse* совсем пропал. Это кажется странным, но тому есть вполне логичное объяснение. Эллипсу, на самом деле, не нужно вообще никакого пространства по вертикали. Именно это он и получает. (Задайте свойству *Height* элемента *Ellipse* положительное значение, и эллипс вновь появится на экране.)

В результате изменения ориентации экрана *Image* получает в свое распоряжение большую ширину, с которой он соотносит высоту, сохраняя пропорции изображения. Но это приводит к тому, что большая часть растрового изображения выходит за нижнюю границу экрана, выталкивая из области видимости и второй *TextBlock*:



Как известно, способность страницы изменять ориентацию определяется свойством *SupportedOrientations* класса *PhoneApplicationPage*. Допустимыми значениями этого свойства являются члены перечисления *SupportedPageOrientation*. Класс *PhoneApplicationPage* определяет еще одно свойство: *Orientation*. Значениями этого свойства являются члены

перечисления *PageOrientation*, обозначающие текущую ориентацию телефона (портретная или альбомная).

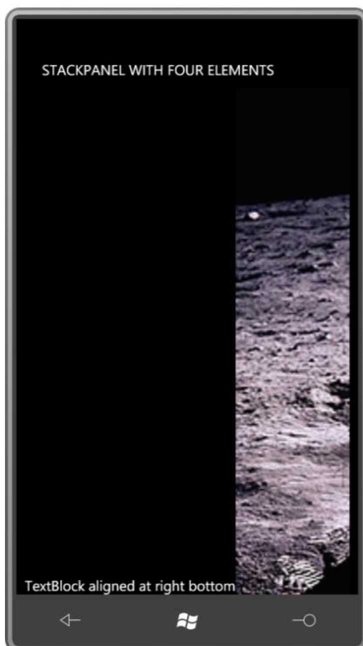
У *StackPanel* есть собственное свойство *Orientation*, но оно не имеет никакого отношения к ориентации страницы. В качестве значений свойства *Orientation* класса *StackPanel* используются члены перечисления *Orientation*: *Horizontal* (Горизонтально) или *Vertical* (Вертикально). Значением по умолчанию является *Vertical*. В приложении StackPanelWithFourElements (Панель с четырьмя элементами) ориентация *StackPanel* меняется по касанию экрана пользователем. Рассмотрим код, реализующий это поведение:

Проект Silverlight: StackPanelWithFourElements Файл: MainPage.xaml.cs (фрагмент)

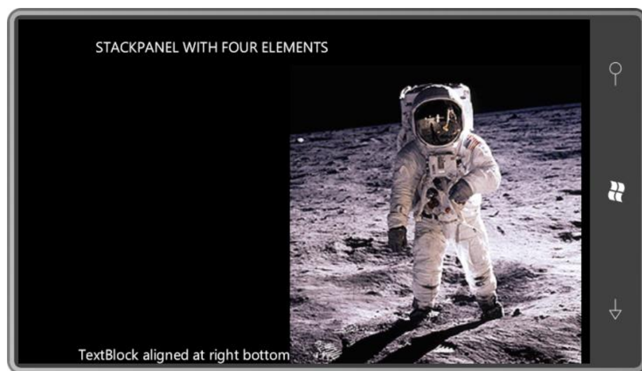
```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    stackPanel.Orientation =
        stackPanel.Orientation == System.Windows.Controls.Orientation.Vertical ?
                                  System.Windows.Controls.Orientation.Horizontal :
                                  System.Windows.Controls.Orientation.Vertical;
    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Перечисление *Orientation* должно быть полностью определено. В противном случае компилятор считает, что используется свойство *Orientation* класса *PhoneApplicationPage*.

Одно касание – и элементы расставлены слева направо:

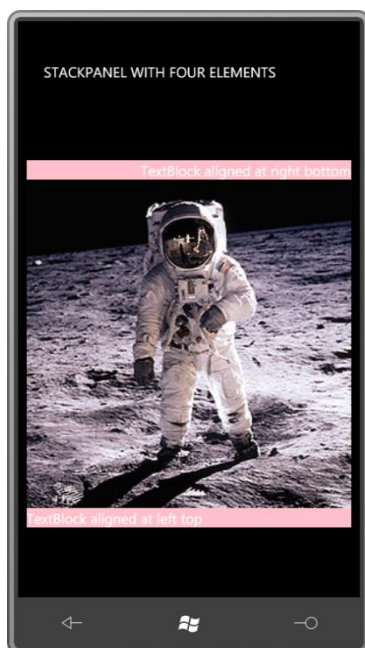


Теперь проигнорировано свойство *HorizontalAlignment* первого *TextBlock*, и значение свойства *VerticalAlignment* обусловливает его размещение внизу экрана. Высота *Image* становится настолько большой, что изображение практически полностью выходит за границы экрана. Если повернуть телефон на бок, ситуация немного улучшится (включая второй *TextBlock*):



StackPanel занимает всю площадь сетки для содержимого, даже если это пространство больше, чем необходимо для размещения его дочерних элементов. Это можно без труда проверить с помощью свойства *Background* элемента *StackPanel*. StackPanel заполняет родительский контейнер, потому что по умолчанию свойства *HorizontalAlignment* и *VerticalAlignment* имеют значение *Stretch*.

Чтобы *StackPanel* занимал именно столько пространства, сколько необходимо, и размещался в сетке для содержимого определенным образом, измените значения свойств *HorizontalAlignment* или *VerticalAlignment*. На рисунке проиллюстрирован пример, когда свойству *Background* задано значение Pink, и свойству *VerticalAlignment* – *Center*.



В данном конкретном приложении свойство *HorizontalAlignment* элемента *StackPanel* вообще не оказывает никакого влияния на компоновку.

Конкатенация текста с помощью *StackPanel*

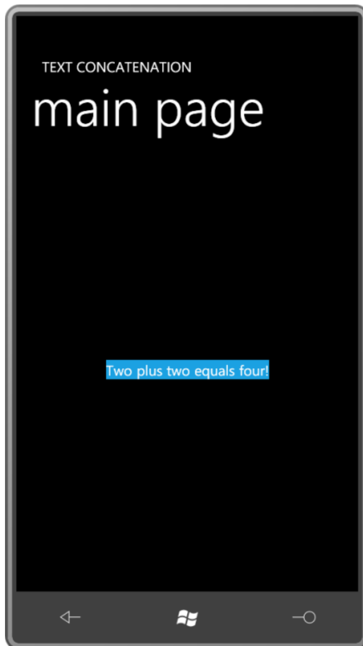
StackPanel с горизонтальной ориентацией может выполнять конкатенацию текста. Это продемонстрировано в проекте TextConcatenation (Конкатенация текста):

Проект Silverlight: TextConcatenation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal"
```

```
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                Background="{StaticResource PhoneAccentBrush}">
            <TextBlock Text="Two " />
            <TextBlock Text="plus " />
            <TextBlock Text="two " />
            <TextBlock Text="equals " />
            <TextBlock Text="four!" />
    </StackPanel>
</Grid>
```

Вот что получается:



Может показаться довольно глупым выполнять конкатенацию текста таким образом, но на самом деле это очень полезная методика. Иногда в приложении некоторый фиксированный текст определен в XAML и объединен с некоторым переменным текстом, поставляемым из кода или привязки данных. *StackPanel* обеспечивает составление этого текста без каких-либо лишних пробелов. (В некоторых случаях для этих целей можно использовать *TextBlock*, задавая в качестве значения его свойства *Inlines* несколько объектов *Run*, но, как будет показано в главе 12, *Run* не может использоваться с привязкой данных.)

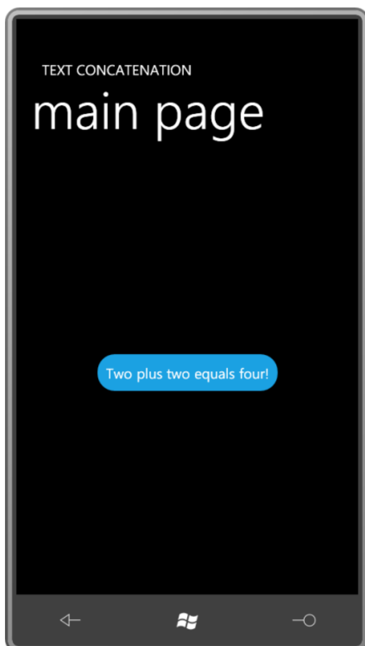
Предположим, требуется, чтобы фон немного выходил за рамки текста, получаемого в результате конкатенации. Свойство *Margin* элемента *StackPanel* не может этого обеспечить, потому что часть текста находится вне этого элемента. У *StackPanel* нет свойства *Padding* (увы!), поэтому придется задавать *Margin* или *Padding* по отдельности для каждого элемента *TextBlock*. Звучит не радостно.

Более простым решением является поместить *StackPanel* в элемент *Border* и перенести все параметры выравнивания и *Background* в этот *Border*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Background="{StaticResource PhoneAccentBrush}"
            Padding="12"
            CornerRadius="24"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Two " />
            <TextBlock Text="plus " />
```

```
            <TextBlock Text="two " />
            <TextBlock Text="equals " />
            <TextBlock Text="four!" />
        </StackPanel>
    </Border>
</Grid>
```

И вот мы получаем красивый фон со скругленными углами:



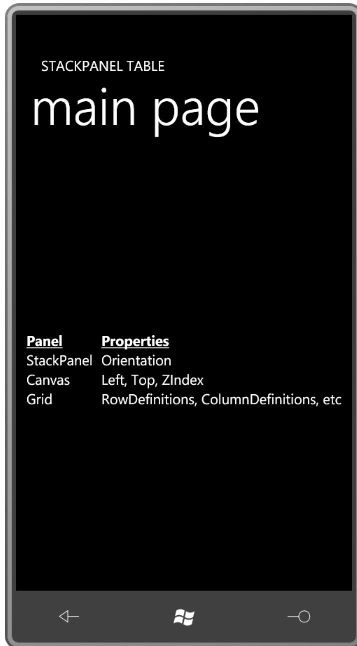
Вложенные панели

StackPanel могут быть вложены друг в друга, что имеет больше смысла в случае, если они по-разному ориентированы. Рассмотрим приложение, где две вертикальные панели вложены в одну горизонтальную:

Проект Silverlight: StackPanelTable Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <StackPanel>
            <TextBlock Text="Panel" FontWeight="Bold"
                                    TextDecorations="Underline" />
            <TextBlock Text="StackPanel" />
            <TextBlock Text="Canvas" />
            <TextBlock Text="Grid" />
        </StackPanel>

        <StackPanel Margin="12 0 0 0">
            <TextBlock Text="Properties" FontWeight="Bold"
                                         TextDecorations="Underline" />
            <TextBlock Text="Orientation" />
            <TextBlock Text="Left, Top, ZIndex" />
            <TextBlock Text="RowDefinitions, ColumnDefinitions, etc" />
        </StackPanel>
    </StackPanel>
</Grid>
```

Для разделения столбцов используется параметр *Margin*:



Ширина вертикального *StackPanel* определяется шириной его самого широкого дочернего элемента, и высота соответствует сумме высот дочерних элементов. Горизонтальный *StackPanel* выровнен по центру экрана, и его ширина соответствует сумме ширин двух его дочерних элементов.

Это не лучший способ создания таблицы! Здесь все выглядит хорошо, потому что высота всех элементов *TextBlock* одинаковая. В противном случае строки не были бы выровнены.

Видимость и компоновка

Класс *UIElement* определяет свойство *Visibility* (Видимость), которое удобно использовать для временного сокрытия элементов. Но это свойство не типа Boolean. *Visibility* типа *Visibility*. Это перечисление, включающее два члена: *Visible* (Видимый) и *Collapsed* (Свернутый).

Вернемся к предыдущему приложению и зададим *Visibility* для одного из элементов:

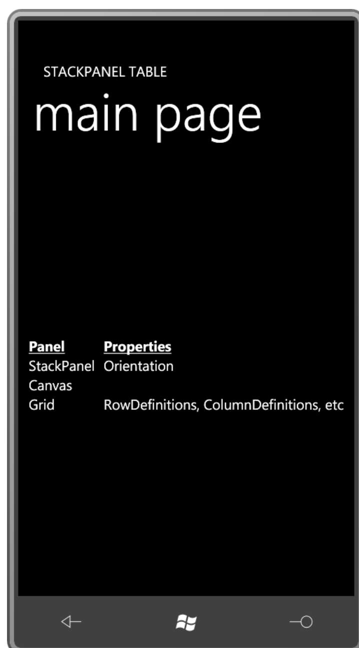
```
<TextBlock Text="Left, Top, ZIndex" Visibility="Collapsed" />
```

Значение *Collapsed* означает, что элемент имеет нулевой размер, т.е. фактически не принимает участия в компоновке. В некоторых случаях именно такое поведение и требуется, но в данной ситуации это приводит к тому, что строки таблицы выстраиваются неверно:

Если требуется скрыть элемент, сохранив его размер и роль в компоновке, используется свойство *Opacity*:

```
<TextBlock Text="Left, Top, ZIndex" Opacity="0" />
```

Теперь *TextBlock* невидимый, но его размер остается прежним:



Такой подход можно считать *почти* правильным. Единственная проблема в том, что *TextBlock* продолжает реагировать на сенсорный ввод. Если требуется скрыть его как из виду, так и от касаний, используйте следующее:

```
<TextBlock Text="Left, Top, ZIndex"
           Opacity="0"
           IsHitTestVisible="False" />
```

Для целей компоновки свойство *Opacity* далеко не так эффективно, как *Visibility*, поэтому следует избегать его использования в случаях, требующих частой смены компоновки.

(Причина того, что *Visibility* не типа Boolean, в Windows Presentation Foundation. В WPF перечисление *Visibility* включает еще один, третий член, под именем *Invisible* (Невидимый), который обеспечивает визуальное сокрытие элемента с сохранением его размера для целей компоновки.)

Свойства *Visibility* и *Opacity* применяются к элементу и его дочерним элементам, поэтому если задать эти свойства для панели, их действие распространится и на дочерние элементы панели.

Если задать свойство *RenderTransform* для панели, ее дочерние элементы также будут подвержены трансформации. Но если задать *RenderTransform* для дочернего элемента, родительская панель при компоновке дочерних элементов проигнорирует все эффекты, обусловливаемые *RenderTransform*.

Два примера использования *ScrollViewer*

Если *StackPanel* включает больше элементов, чем помещается на экране (или любом другом контейнере, в котором располагается *StackPanel*), самые нижние (или правые) элементы будут не видны.

Если есть опасение, что все дочерние элементы *StackPanel* не поместятся на экране телефона, *StackPanel* можно поместить в *ScrollViewer* (Прокручиваемая область). *ScrollViewer* – это элемент управления, определяющий необходимые размеры для отображения своего содержимого и предоставляющий одну или две полосы прокрутки.

В Windows Phone 7 полосы прокрутки, скорее, виртуальные, чем реальные. Мы прокручиваем *ScrollViewer* не с помощью полос прокрутки, а просто перемещаем страницу, проводя пальцем по экрану. Но при обсуждении этой функциональности удобно использовать традиционное понятие полосы прокрутки, поэтому я буду его придерживаться.

По умолчанию вертикальная полоса прокрутки видима, и горизонтальная скрыта, но это поведение можно переопределить с помощью свойств *VerticalScrollBarVisibility* (Видимость вертикальной полосы прокрутки) и *HorizontalScrollBarVisibility* (Видимость горизонтальной полосы прокрутки). Возможные значения этих свойств являются членами перечисления *ScrollBarVisibility* (Видимость полосы прокрутки): *Visible*, *Hidden* (Скрыта), *Auto* (видима только в случае необходимости) и *Disabled* (Отключена) (видима, но не активна).

Следующее приложение-пример является средством для чтения электронных книг. Ну, скажем, не вполне *книг*, скорее, *отрывков из книг*. Также это приложение не отличается универсальностью, оно отображает лишь небольшую юмористическую заметку, написанную Марком Твеном в 1880 году, которая считается первым описанием впечатлений человека, ставшего свидетелем разговора по телефону, без возможности слышать второго собеседника. (Женщина, говорящая по телефону – жена Марка Твена, Оливия.)

Я внес небольшие дополнения в заголовок: выделил его другим цветом и разбил на две строки:

Проект Silverlight: TelephonicConversation Файл: MainPage.xaml (фрагмент)

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="24,24,0,12">
    <TextBlock x:Name="ApplicationTitle"
               Style="{StaticResource PhoneTextNormalStyle}"
               TextAlignment="Center"
               Foreground="{StaticResource PhoneAccentBrush}">
        "A Telephonic Conversation"<LineBreak />by Mark Twain
```

```
    </TextBlock>
</StackPanel>
```

Сетка для содержимого включает собственную коллекцию *Resources* с заданным *Style*. В *Grid* расположен *ScrollViewer*, в котором находится *StackPanel*, включающий два элемента *TextBlock* с рассказом, по одному для каждого абзаца. Обратите внимание на четкое разделение задач: элементы *TextBlock* отображают текст, *StackPanel* обеспечивает компоновку, *ScrollViewer* обеспечивает прокрутку:

Проект Silverlight: TelephonicConversation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Resources>
        <Style x:Key="paragraphStyle"
               TargetType="TextBlock">
            <Setter Property="TextWrapping" Value="Wrap" />
            <Setter Property="Margin" Value="5" />
            <Setter Property="FontSize" Value="{StaticResource PhoneFontSizeSmall}"
/>
        </Style>
    </Grid.Resources>

    <ScrollViewer Padding="5">
        <StackPanel>
            <TextBlock Style="{StaticResource paragraphStyle}">
                 I consider that a conversation by telephone — when you are
                simply sitting by and not taking any part in that conversation —
                is one of the solemnest curiosities of this modern life.
                Yesterday I was writing a deep article on a sublime philosophical
                subject while such a conversation was going on in the
                room. I notice that one can always write best when somebody
                is talking through a telephone close by. Well, the thing began
                in this way. A member of our household came in and asked
                me to have our house put into communication with Mr. Bagley's,
                down town. I have observed, in many cities, that the sex
                always shrink from calling up the central office themselves. I
                don't know why, but they do. So I touched the bell, and this
                talk ensued: —
            </TextBlock>
            <TextBlock Style="{StaticResource paragraphStyle}">
                 <Run FontStyle="Italic">Central Office.</Run>
                [Gruffly.] Hello!
            </TextBlock>
            <TextBlock Style="{StaticResource paragraphStyle}">
                 <Run FontStyle="Italic">I.</Run> Is it the Central Office?
            </TextBlock>

            …

            <TextBlock Style="{StaticResource paragraphStyle}"
                       TextAlignment="Right">
                — <Run FontStyle="Italic">Atlantic Monthly</Run>, June 1880
            </TextBlock>
        </StackPanel>
    </ScrollViewer>
</Grid>
```

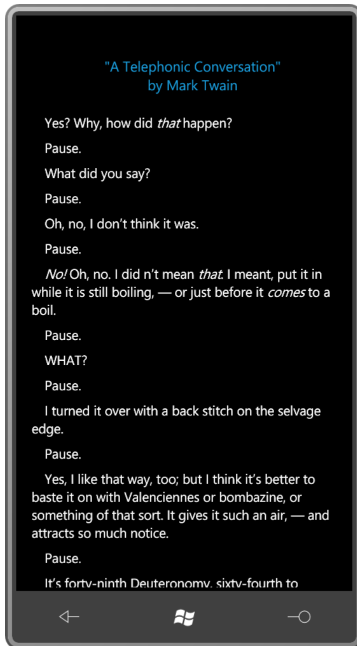
Конечно же, это не весь файл. Большую часть рассказа я заменил здесь на многоточия (…).

Чтобы *StackPanel* не прилегал вплотную к краям, для *ScrollViewer* задан отступ (*Padding*) в 5 пикселов. Кроме того, свойству *Margin* каждого *TextBlock* задано значение 5 пикселов посредством *Style*. Отступ и поле дают в сумме по 10 пикселов справа и слева и 10 пикселов

между каждым *TextBlock*, что делает абзацы более различимыми и текст более удобным для восприятия.

Также я вставил Unicode-символ   в начало каждого абзаца. Это длинный пробел в кодировке Unicode, который эффективно обеспечивает отступ первой строки на ширину одного символа.

По умолчанию *ScrollViewer* обеспечивает вертикальную прокрутку. Элемент управления реагирует на касание, что позволяет без труда «прокручивать» отображаемый текст и прочитать рассказ полностью.



В приложении PublicClasses (Открытые классы), которое мы рассмотрим следующим, также есть *ScrollViewer*, включающий вертикальный *StackPanel*, но заполнение этой панели реализовано полностью в коде. Используя технологию отражения, файл выделенного кода получает все открытые классы, предоставляемые сборками System.Windows, Microsoft.Phone, Microsoft.Phone.Controls и Microsoft.Phone.Controls.Maps, и представляет их список в виде иерархии классов.

Для обеспечения этого в XAML-файл включен пустой *StackPanel*, идентифицируемый по имени:

Проект Silverlight: PublicClasses Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ScrollViewer HorizontalScrollBarVisibility="Auto">
        <StackPanel Name="stackPanel" />
    </ScrollViewer>
</Grid>
```

По умолчанию *VerticalScrollBarVisibility* имеет значение *Visible*, и я задал для свойства *HorizontalScrollBarVisibility* значение *Auto*. Если какая-либо строка текста будет слишком длинной и не поместится на экране, ее можно будет увидеть посредством горизонтальной прокрутки.

Данная горизонтальная прокрутка существенно отличает данное приложение от предыдущего. Когда текст автоматически переносится на новую строку, как это было в проекте TelephonicConversation (Разговор по телефону), в горизонтальной прокрутке нет необходимости. В данном приложении длина строк без переносов может превышать ширину экрана, поэтому желательно предусмотреть горизонтальную полосу прокрутки.

Файл выделенного кода использует отдельный небольшой класс *ClassAndChildren* (Класс и потомки) для хранения иерархии классов, представляемой в виде дерева.

Проект Silverlight: PublicClasses Файл: ClassAndChildren.cs

```
using System;
using System.Collections.Generic;

namespace PublicClasses
{
    class ClassAndChildren
    {
        public ClassAndChildren(Type parent)
        {
            Type = parent;
            SubClasses = new List<ClassAndChildren>();
        }

        public Type Type { set; get; }
        public List<ClassAndChildren> SubClasses { set; get; }
    }
}
```

Объект *ClassAndChildren* создается для каждого класса, отображаемого в дереве. Каждый объект *ClassAndChildren* включает объект *List* (Список) со всеми классами, наследуемыми от данного.

Привожу код класса *MainPage* полностью. Ему необходима директива *using* для подключения пространства имен *System.Reflection*.

Проект Silverlight: PublicClasses Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Brush accentBrush;

    public MainPage()
    {
        InitializeComponent();
        accentBrush = this.Resources["PhoneAccentBrush"] as Brush;

        // Загружаем все сборки
        List<Assembly> assemblies = new List<Assembly>();
        assemblies.Add(Assembly.Load("System.Windows"));
        assemblies.Add(Assembly.Load("Microsoft.Phone"));
        assemblies.Add(Assembly.Load("Microsoft.Phone.Controls"));
        assemblies.Add(Assembly.Load("Microsoft.Phone.Controls.Maps"));

        // Задаем корневой объект (используем DependencyObject,
        // чтобы сделать список короче)
        Type typeRoot = typeof(object);

        // Составляем полный список открытых классов
        List<Type> classes = new List<Type>();
        classes.Add(typeRoot);
```

```
        foreach (Assembly assembly in assemblies)
            foreach (Type type in assembly.GetTypes())
                if (type.IsPublic && type.IsSubclassOf(typeRoot))
                    classes.Add(type);

        // Сортируем эти классы
        classes.Sort(TypeCompare);

        // Теперь помещаем все эти сортированные классы в древовидную структуру
        ClassAndChildren rootClass = new ClassAndChildren(typeRoot);
        AddToTree(rootClass, classes);

        // Выводим дерево на экран
        Display(rootClass, 0);
    }

    int TypeCompare(Type t1, Type t2)
    {
        return String.Compare(t1.Name, t2.Name);
    }

    // Рекурсивный метод
    void AddToTree(ClassAndChildren parentClass, List<Type> classes)
    {
        foreach (Type type in classes)
        {
            if (type.BaseType == parentClass.Type)
            {
                ClassAndChildren subClass = new ClassAndChildren(type);
                parentClass.SubClasses.Add(subClass);
                AddToTree(subClass, classes);
            }
        }
    }

    // Рекурсивный метод
    void Display(ClassAndChildren parentClass, int indent)
    {
        string str1 = String.Format("{0}{1}{2}{3}",
                                    new string(' ', indent * 4),
                                    parentClass.Type.Name,
                                    parentClass.Type.IsAbstract ? " (abstract)" :
"",
                                    parentClass.Type.IsSealed ? " (sealed)" : "");

        string str2 = " " + parentClass.Type.Namespace;

        TextBlock txtblk = new TextBlock();
        txtblk.Inlines.Add(str1);
        txtblk.Inlines.Add(new Run
        {
            Text = str2,
            Foreground = accentBrush
        });

        stackPanel.Children.Add(txtblk);

        foreach (ClassAndChildren child in parentClass.SubClasses)
            Display(child, indent + 1);
    }
}
```

Конструктор сначала сохраняет все открытые классы из основных сборок Silverlight в большую коллекцию. После этого они сортируются по имени и распределяются в объекты *ClassAndChildren* с помощью рекурсивного метода. Второй рекурсивный метод добавляет элементы *TextBlock* в *StackPanel*. Обратите внимание, что каждый элемент *TextBlock* включает

коллекцию *Inlines* с двумя объектами *Run*. Для улучшения восприятия я решил выделить имена пространств имен другим цветом и для удобства использовал контрастный цвет, выбранный пользователем.

На рисунке показана часть иерархии классов, представляющая класс *Panel* и производные от него классы:

Механизм компоновки

Давайте проведем небольшой эксперимент. Перейдем к файлу XAML проекта TelephonicConversation и вставим в тег *ScrollViewer* следующий параметр:

```
HorizontalScrollBarVisibility="Visible"
```

И тут же произойдет разительное изменение. Все элементы *TextBlock* превратятся в длинные строки текста без переносов. Что случилось? Почему свойство элемента *ScrollViewer* имеет такое огромное влияние на элементы *TextBlock*?

Но такое поведение не должно вызывать удивления. Если у *ScrollViewer* есть горизонтальная полоса прокрутки, то должна быть и причина ее существования. Этой причины нет, если все слова всех *TextBlock* переносятся на новую строку в случае необходимости. Если же предполагается использование горизонтальной полосы прокрутки, текст параграфов должен быть выстроен в одну строку.

Однако было бы не лишним не только знать эту странность поведения, но понимать реальный механизм происходящего. Понимание системы компоновки – один из самых важных навыков создания приложений на Silverlight. Система компоновки – мощнейший механизм, но для непосвященных он может казаться довольно странным.

Создание компоновки в Silverlight – это двухпроходный процесс, начинающийся с вершины дерева визуальных элементов и проходящий вниз по всем потомкам элементов. В приложении для телефона на Silverlight он начинается с *PhoneApplicationFrame*, затем идет *PhoneApplicationPage*, после чего, скорее всего, *Grid*, и затем (обычно) *StackPanel* и второй *Grid*. В приложении TelephonicConversation процесс продолжается элементом *ScrollViewer*, который может включать собственный *Border*, затем *StackPanel* и, наконец, элементы

TextBlock. У этих элементов *TextBlock* нет дочерних элементов, поэтому на них цепочка заканчивается.

При первом проходе каждый элемент дерева опрашивает свои дочерние элементы для получения желаемого размера. При втором проходе элементы компонуют дочерние элементы на своей поверхности. Компоновка может быть простой или сложной. Например, у *Border* только один потомок, и для определения местоположения этого потомка относительно себя данному элементу необходимо учесть лишь толщину собственной рамки (*BorderThickness*). А вот компоновка дочерних элементов в производных от *Panel* классах – намного более сложная задача.

Когда родительский элемент запрашивает размер своих дочерних элементов, он, буквально, говорит: «Ты можешь быть такого размера. Насколько большим ты хочешь быть?» – и каждый дочерний элемент вычисляет для себя желаемый размер. Все размеры представляются в форме структуры *Size* со свойствами *Width* и *Height*. Если у дочернего элемента тоже есть потомки, он должен определять свой размер на основании их размеров. И этот процесс продолжается вниз по дереву элементов, вплоть до таких элементов, как *TextBlock*, у которых нет потомков.

Элементы определяют свой размер по-разному, в зависимости от природы элемента. Возьмем для примера *TextBlock*, который включает большой фрагмент текста, и свойству *TextWrapping* (Автоматический перенос текста на новую строку) которого задано значение *Wrap* (Выполнять перенос текста). В этом случае *TextBlock* проверяет значение свойства *Width*, чтобы выяснить, какой размер ему доступен и где должен выполняться разрыв строки. После этого он знает, сколько строк ему необходимо для отображения имеющегося текста, и какое пространство в вертикальном направлении требуется для размещения всех этих строк. Так *TextBlock* вычисляет желаемый размер.

Но тут есть одна сложность. Родительский элемент предоставляет своему потомку доступный размер посредством структуры *Size*, имеющей два свойства, *Width* и *Height*, типа *double*. Иногда свойству *Width* или *Height* (или обоим) родителя задано особое значение с плавающей точкой: *Double.PositiveInfinity* (Плюс бесконечность). В этом случае родитель, буквально, сообщает: «Потомок, я предлагаю в твое распоряжение неограниченное пространство по ширине [или по высоте, или в обоих направлениях]. Сколько тебе надо?»

Дочерний элемент не может ответить: «Я хочу все!». Несмотря на то, что дочерние элементы иногда пытаются занять весь предоставляемый размер, это запрещено. Запрашиваемый желаемый размер дочернего элемента должен быть конечным и неотрицательным.

Вот как *StackPanel* запрашивает размеры своих дочерних элементов. Вертикальный *StackPanel* предлагает каждому из своих потомков доступный размер, ширина которого равна его собственной ширине, и неограниченный по высоте.

Но в этом кроется парадокс: некоторые элементы, такие как *TextBlock* и *Image*, имеют неявно заданный подразумеваемый размер, который определяется размером форматированного текста или исходного растрового изображения. А у некоторых элементов, таких как *Ellipse*, такого подразумеваемого размера нет. Если *Ellipse* задан конкретный размер, он отображается такого размера. Но если *Ellipse* предлагается бесконечный размер, у него нет иного выбора, кроме как сжать себя полностью, до несуществования.

Чтобы разобраться во всем этом механизме основательно, чрезвычайно полезно самостоятельно создать несколько простых панелей.

Панель, взгляд изнутри

Панели полностью описываются в коде без участия XAML. При создании производного от *Panel* класса обычно описывают несколько свойств, чтобы сделать панель более гибкой. Поскольку практически всегда эти свойства являются свойствами-зависимостями, давайте дождемся главы 11, в которой будет рассмотрено, как создавать панели с собственными свойствами.

Кроме определения пользовательских свойств, каждая панель всегда перегружает два метода, *MeasureOverride* (Перегрузка метода измерить) и *ArrangeOverride* (Перегрузка метода скомпоновать), которые соответствуют двум проходам компоновки. За первый проход каждый родительский элемент определяет размеры своих дочерних элементов; во втором проходе родительский элемент компонует дочерние элементы относительно себя.

Для реализации обеих задач панель выполняет доступ к свойству *Children*, унаследованному от *Panel*. (Свойство *Children* типа *UIElementCollection* (Коллекция элементов UI), но вы не можете самостоятельно создать экземпляр *UIElementCollection*, а этот объект выполняет некоторые особые операции в фоновом режиме, о чем вы даже не догадываетесь. Поэтому собственный подобный *Panel* класс без наследования от *Panel* создать невозможно. Если требуется элемент, который может размещать в себе множество дочерних элементов с достаточной гибкостью, он должен быть унаследован от *Panel*.)

Тайной окутано то, кто придумал такие имена – *MeasureOverride* и *ArrangeOverride* – для защищенных виртуальных методов? Почему ключевое слово C# *override* входит в состав имени метода?

Мне это не известно. Эти имена появились в Windows Presentation Foundation и указывают на разницу между классами *UIElement* и *FrameworkElement*. *UIElement* реализовывает сравнительно простую систему компоновки и для ее поддержки у него есть два метода: *Measure* (Измерить) и *Arrange* (Скомпоновать). Эти методы по-прежнему играют очень важную роль в компоновке (как мы увидим далее), но *FrameworkElement* понадобилось добавить в компоновку некоторые более сложные аспекты, а именно *HorizontalAlignment*, *VerticalAlignment* и *Margin*. Эти аспекты довольно сильно запутали систему компоновки, поэтому *FrameworkElement* ввел два новых метода – *MeasureOverride* и *ArrangeOverride* – которые заменили методы *Measure* и *Arrange* класса *UIElement*.

MeasureOverride и *ArrangeOverride* являются защищенными виртуальными методами. *Measure* и *Arrange* – открытые запечатанные методы. Пользовательская панель перегружает *MeasureOverride* и *ArrangeOverride*. В *MeasureOverride* панель вызывает *Measure* всех своих дочерних элементов; в *ArrangeOverride* панель вызывает *Arrange* всех своих дочерних элементов. Затем эти методы *Measure* и *Arrange* в каждом дочернем элементе вызывают его методы *MeasureOverride* и *ArrangeOverride*, продолжая тем самым этот процесс вниз по дереву визуальных элементов.

Панели *нет* необходимости беспокоиться о следующих свойствах, которые могут быть заданы для нее или ее дочерних элементов:

- *HorizontalAlignment* и *VerticalAlignment*
- *Margin*
- *Visibility*
- *Opacity* (не оказывает никакого влияния на компоновку)
- *RenderTransform* (не оказывает никакого влияния на компоновку)

- *Height*, *MinHeight* и *MaxHeight*
- *Width*, *MinWidth* и *MaxWidth*

Все эти свойства обрабатываются автоматически.

Клон *Grid* с одной ячейкой

Простейшей панелью является *Grid*, который не содержит ни строк, ни столбцов. Такие *Grid* называют «*Grid* с одной ячейкой». Мы уже рассматривали использование *Grid* с одной ячейкой на примере *ContentPanel*. Как было показано, в *Grid* могут располагаться несколько дочерних элементов, но они перекрывают друг друга.

Продублируем функциональность *Grid* с одной ячейкой, создав класс *SingleCellGrid* (Сетка с одной ячейкой).

В новом проекте под именем SingleCellGridDemo (Демонстрация сетки с одной ячейкой) щелкнем правой кнопкой имя проекта, выберем в меню Add и New Item, затем выберем в диалоговом окне Class и назовем его SingleCellGrid.cs. В созданном файле убедимся, что класс является открытым и наследуется от *Panel*.

Проект Silverlight: SingleCellGridDemo Файл: SingleCellGrid.cs (фрагмент)

```
namespace SingleCellGridDemo
{
    public class SingleCellGrid : Panel
    {
        …
    }
}
```

Как все панели, этот класс перегружает два метода: *MeasureOverride* и *ArrangeOverride*. Рассмотрим первый из них:

Проект Silverlight: SingleCellGridDemo Файл: SingleCellGrid.cs (фрагмент)

```
protected override Size MeasureOverride(Size availableSize)
{
    Size compositeSize = new Size();

    foreach (UIElement child in Children)
    {
        child.Measure(availableSize);
        compositeSize.Width = Math.Max(compositeSize.Width,
child.DesiredSize.Width);
        compositeSize.Height = Math.Max(compositeSize.Height,
child.DesiredSize.Height);
    }

    return compositeSize;
}
```

В *MeasureOverride* передается параметр *availableSize* (Доступный размер) типа *Size*. Эта структура имеет два свойства, *Width* и *Height*, типа *double*. Это размер, который панель получает от своего родительского элемента. Один или оба из этих размеров могут быть бесконечными.

Метод *MeasureOverride* выполняет две фундаментальные задачи.

Первая – вызов метода *Measure* всех дочерних элементов. Выполнение этой операции имеет важнейшее значение, в противном случае потомок не будет иметь размера и не будет отображен на экране. *MeasureOverride* практически всегда выполняет эту задачу путем перебора коллекции *Children* с помощью цикла *foreach*.

Вторая задача метода *MeasureOverride* – возвратить желаемый размер панели. В данном методе *MeasureOverride* этот размер сохраняется в переменной *compositeSize* (Совокупный размер). Данный размер должен иметь конечное неотрицательное значение. Метод *MeasureOverride* не может возвратить аргумент *availableSize* с предположением, что ему требуется все предлагаемое пространство, потому что аргумент *availableSize* может содержать бесконечные значения размеров.

На момент вызова метода *MeasureOverride* аргумент *availableSize* уже скорректирован должным образом. Если для панели задано свойство *Margin*, это поле не входит в *availableSize*. Если для панели задано любое из свойств *Width*, *MinWidth* (Минимальная ширина), *MaxWidth* (Максимальная ширина), *Height*, *MinHeight* (Минимальная высота) или *MaxHeight* (Максимальная высота), их значения ограничивают доступный размер.

Обе задачи *MeasureOverride* обычно выполняются совместно. Панель вызывает метод *Measure* всех дочерних элементов и предлагает каждому из них доступный размер. Этот размер может быть бесконечным. То какой аргумент *Size* передается в метод *Measure*, зависит от парадигмы конкретной панели. В данном случае *SingleCellGrid* предлагает каждому своему дочернему элементу собственный *availableSize*:

```
child.Measure(availableSize);
```

Панель позволяет каждому дочернему элементу существовать на одном пространстве с собой. Нет никакой проблемы в том, если этот аргумент *availableSize* определяет бесконечные размеры.

По завершении выполнения метода *Measure* свойство *DesiredSize* дочернего элемента задано и имеет действительное значение. Таков механизм определения желаемого размера дочернего элемента его родителем. Полученное значение свойства *DesiredSize* вычисляется методом *Measure* дочернего элемента после вызова собственного метода *MeasureOverride*, который, возможно, опрашивал о желаемых размерах собственные дочерние элементы. Методу *MeasureOverride* нет необходимости беспокоиться о значениях *Margin* или явно заданных *Width* и *Height*. Для этого есть метод *Measure*, который все учтет и настроит *DesiredSize* соответствующим образом. Например, если для дочернего элемента задано свойство *Margin*, *DesiredSize* будет включать это дополнительное поле.

Приведем некоторые примеры. Метод *MeasureOverride* элемента *TextBlock* возвращает размер текста, отображаемого конкретным шрифтом. Метод *MeasureOverride* элемента *Image* возвращает исходные размеры растрового изображения в пикселах. Метод *MeasureOverride* элемента *Ellipse* возвращает нулевой размер.

Свойство *DesiredSize* всегда имеет конечное значение. Метод *MeasureOverride* в *SingleCellGrid* определяет максимальный размер на основании значений свойств *DesiredSize* всех дочерних элементов и сохраняет его в локальной переменной *compositeSize*:

```
compositeSize.Width = Math.Max(compositeSize.Width, child.DesiredSize.Width);
compositeSize.Height = Math.Max(compositeSize.Height, child.DesiredSize.Height);
```

Этот размер отражает самую большую ширину и высоту, необходимые для размещения всех дочерних элементов.

ArrangeOverride – второй обязательный метод наследуемого от *Panel* класса. Рассмотрим, как он описан в классе *SingleCellGrid*:

Проект Silverlight: SingleCellGridDemo Файл: SingleCellGrid.cs (фрагмент)

```
protected override Size ArrangeOverride(Size finalSize)
{
    foreach (UIElement child in Children)
    {
        child.Arrange(new Rect(new Point(), finalSize));
    }

    return base.ArrangeOverride(finalSize);
}
```

Метод принимает аргумент *finalSize* (Окончательный размер). Это область, предоставленная панели его родительским элементом. Она всегда имеет конечные размеры.

Задача метода *ArrangeOverride* – скомпоновать дочерние элементы в предоставленной области. Это осуществляется путем перебора всех дочерних элементов и вызова метода *Arrange* каждого из них. Метод *Arrange* принимает обязательный аргумент типа *Rect*. Это прямоугольник, который определяется объектом *Point*, указывающим на верхний левый угол, и объектом *Size*, обозначающим высоту и ширину. Обычно это единственное появление *Rect* в процессе компоновки. *Rect* задает и местоположение дочернего элемента относительно верхнего левого угла родительского элемента, и его размер.

В данном конкретном случае всем дочерним элементам задан размер *finalSize*, такой же как размер самой панели, и все они помещены в верхний левый угол панели.

Если вы думаете, что размер, передаваемый в *Arrange*, должен быть *DesiredSize* дочернего элемента, это не так (по крайней мере не для этой конкретной панели). Очень часто значение *finalSize* больше, чем *DesiredSize* дочернего элемента. (Экстремальным случаем является *Ellipse*, для которого *DesiredSize* равен нулю.) Вот как в методе *Arrange* дочернего элемента настраиваются значения *HorizontalAlignment* и *VerticalAlignment*. В *SingleCellGrid* при вызове метода *Arrange* дочернего элемента в него передается *finalSize*:

```
child.Arrange(new Rect(new Point(), finalSize));
```

Метод *Arrange* сравнивает этот размер с собственным *DesiredSize* дочернего элемента и затем вызывает метод *ArrangeOverride* дочернего элемента, передавая в него размер и положение, скорректированные на основании параметров *HorizontalAlignment* и *VerticalAlignment*. Так *Ellipse* получает ненулевой размер, даже если его *DesiredSize* равен нулю.

Метод *ArrangeOverride* практически всегда возвращает аргумент *finalSize*, который содержит значение, возвращенное методом базового класса *Panel*.

Теперь протестируем все это. Файл MainPage.xaml в проекте SingleCellGridDemo должен ссылаться на этот пользовательский класс. В корневом элементе описание пространства имен XML связывает имя «local» (локальный) с пространством имен .NET, используемым проектом:

```
xmlns:local="clr-namespace:SingleCellGridDemo"
```

В файле MainPage.xaml *SingleCellGrid* помещается в сетку для содержимого и затем заполняется все теми же четырьмя элементами, с которыми мы работали в первых двух приложениях данной главы:

Проект Silverlight: SingleCellGridDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <local:SingleCellGrid>
        <TextBlock Text="TextBlock aligned at right bottom"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom" />

        <Image Source="Images/BuzzAldrinOnTheMoon.png" />

        <Ellipse Stroke="{StaticResource PhoneAccentBrush}"
                 StrokeThickness="24" />

        <TextBlock Text="TextBlock aligned at left top"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top" />
    </local:SingleCellGrid>
</Grid>
```

Данное приложение обеспечивает абсолютно такое же отображение элементов, как и приложение GridWithFourElements.

Пользовательский вертикальный *StackPanel*

Рассмотрим еще один производный от *Panel* класс – *StackPanel* – и вы увидите, как он отличается от *Grid* с одной ячейкой. Чтобы не усложнять код и избежать описания свойств, я назову этот пользовательский класс *VerticalStackPanel*. Вот метод *MeasureOverride*:

Проект Silverlight: VerticalStackPanelDemo Файл: VerticalStackPanel.cs (фрагмент)

```
protected override Size MeasureOverride(Size availableSize)
{
    Size compositeSize = new Size();

    foreach (UIElement child in Children)
    {
        child.Measure(new Size(availableSize.Width, Double.PositiveInfinity));
        compositeSize.Width = Math.Max(compositeSize.Width,
child.DesiredSize.Width);
        compositeSize.Height += child.DesiredSize.Height;
    }
    return compositeSize;
}
```

Как обычно, метод *MeasureOverride* перебирает все дочерние элементы и вызывает *Measure* для каждого из них. Но заметьте, что в данном случае предлагаемый каждому дочернему элементу *Size* включает ширину самого *VerticalStackPanel* и не ограничен по высоте.

По сути, у дочерних элементов спрашивают, какой высоты они должны быть. Для *TextBlock* высоту определить просто: она равна высоте текста. Для *Ellipse* тоже нет ничего сложного: она равна нулю. А вот элемент *Image* вычисляет высоту по заданной ширине, исходя из неизменных пропорций изображения. Полученное значение высоты может отличаться от высоты *Grid* с одной ячейкой.

Как и в перегруженном *MeasureOverride* класса *SingleCellGrid*, свойство *Width* локальной переменной *compositeSize* основывается на максимальной ширине дочернего элемента. А вот свойство *Height* переменной *compositeSize* в данной панели является накапливаемым. Высота *VerticalStackPanel* должна быть равна сумме высот всех его дочерних элементов.

Если сам *VerticalStackPanel* располагается в *StackPanel* с горизонтальной ориентацией, свойство *Width* переменной *availableSize* будет иметь бесконечное значение, и *Measure* будет

вызываться для каждого дочернего элемента, размер которого бесконечен в обоих направлениях. В этом нет ничего страшного и ничего такого, что требовало бы какого-то особого подхода.

В *SingleCellGrid* метод *ArrangeOverride* разместил все дочерние элементы в одном и том же месте. Панели *VerticalStackPanel* необходимо выстроить дочерние элементы в ряд друг над другом. Для этого он определяет локальные переменные *x* и *y*:

Проект Silverlight: VerticalStackPanelDemo Файл: VerticalStackPanel.cs (фрагмент)

```
protected override Size ArrangeOverride(Size finalSize)
{
    double x = 0, y = 0;

    foreach (UIElement child in Children)
    {
        child.Arrange(new Rect(x, y, finalSize.Width, child.DesiredSize.Height));
        y += child.DesiredSize.Height;
    }
    return base.ArrangeOverride(finalSize);
}
```

Переменная *x* всегда остается равной 0, а в переменной суммируются значения свойства *Height* размера *DesiredSize* каждого дочернего элемента. При вызове метода *Arrange* в него передаются *x* и y, указывающие на местоположение дочернего элемента относительно верхнего левого угла панели. Свойство *Width* этого *Rect* равно свойству *Width finalSize*, но свойство *Height* равно значению *Height DesiredSize* дочернего элемента. Это значение соответствует величине пространства по вертикали, выделенного для каждого дочернего элемента ранее в методе *MeasureOverride*. Предоставление дочернему элементу его собственной желаемой высоты в методе *Arrange*, по сути, аннулирует любое значение *VerticalAlignment*, заданное для дочернего элемента. Этот эффект был обнаружен нами ранее опытным путем при рассмотрении вертикального *StackPanel*.

В общем, если дочернему элементу в методе *MeasureOverride* предлагается неограниченный размер по вертикали или по горизонтали, или в обоих направлениях, этот размер дочернего элемента будет определен в методе *ArrangeOverride* исходя из *DesiredSize*.

Файл MainPage.xaml проекта VerticalStackPanelDemo (Демонстрация вертикальной стек-панели) аналогичен приводимому в начале данной главы, только в нем используется *VerticalStackPanel*:

Проект Silverlight: VerticalStackPanelDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <local:VerticalStackPanel>
        <TextBlock Text="TextBlock aligned at right bottom"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom" />

        <Image Source="Images/BuzzAldrinOnTheMoon.png" />

        <Ellipse Stroke="{StaticResource PhoneAccentBrush}"
                 StrokeThickness="24" />

        <TextBlock Text="TextBlock aligned at left top"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top" />
    </local:VerticalStackPanel>
</Grid>
```

На экране получаем тот же вывод, что и в предыдущем приложении.

Если *VerticalStackPanel* располагается в сетке для содержимого, в его метод *MeasureOverride* передаются размеры, равные размерам самой сетки (за вычетом размера *Margin*, если он задан для *VerticalStackPanel*). Это конечный размер, который мы обсуждали в приложении SilverlightWhatSize в главе 2.

Но поместите *VerticalStackPanel* (или вертикальный *StackPanel*) в *ScrollViewer*, и эффект будет совершенно иным. По умолчанию *ScrollViewer* отображает вертикальную полосу прокрутки, поэтому *ScrollViewer* (или, скорее, один из его дочерних элементов) вызывает *Measure* элемента *StackPanel* с конечной шириной, но бесконечной высотой. После этого *DesiredHeight* вертикального *StackPanel* предоставляет *ScrollViewer* данные, необходимые для задания параметров вертикальной полосы прокрутки.

Когда свойству *HorizontalScrollBarVisibility* элемента *ScrollViewer* задается значение *Visible* или *Auto*, *ScrollViewer* вызывает метод *Measure* элемента *StackPanel* с неограниченной шириной, чтобы определить желаемую ширину панели. *ScrollViewer* использует эти данные для задания параметров горизонтальной полосы прокрутки. Затем *StackPanel* передает эту бесконечную ширину в вызовы *MeasureOverride* его дочерних элементов. Потенциально это может иметь неожиданный эффект для дочерних элементов *StackPanel*.

Например, если свойству *TextWrapping* элемента *TextBlock* задано значение *Wrap*, в вызове собственного *MeasureOverride* для определения необходимого количества строк для размещения текста он использует значение *availableSize.Width*. Но если *availableSize.Width* имеет неограниченный размер – как это происходит в случае, когда *TextBlock* размещен в *ScrollViewer* с активированной горизонтальной полосой прокрутки – у *TextBlock* нет другого выбора, кроме как возвратить свой размер с текстом без переносов.

Вот почему в приложении TelephonicConversation не следует активировать горизонтальную полосу прокрутки в *ScrollViewer*.

Старомодный *Canvas*

Безусловно, *Canvas* является самым старомодным видом панелей. Размещение элементов в *Canvas* осуществляется через задание их координат по вертикали и горизонтали относительно верхнего левого угла.

Canvas обладает двумя необычными характеристиками:

- В методе *MeasureOverride Canvas* всегда вызывает *Measure* своего дочернего элемента, передавая в него неограниченные ширину и высоту. (Соответственно, в *ArrangeOverride Canvas* определяет размеры дочерних элементов на основании их *DesiredSize*.)
- Метод *MeasureOverride* элемента *Canvas* возвращает размер, включающий нулевую ширину и нулевую высоту.

Первое означает, что дочерний элемент *Canvas* всегда отображается минимально возможного размера, что для *Ellipse* и *Rectangle* означает вообще ничего, а для *Image* – оригинальный размер растрового изображения. Любые заданные для дочерних элементов значения свойств *HorizontalAlignment* или *VerticalAlignment* не имеют никакого эффекта в *Canvas*.

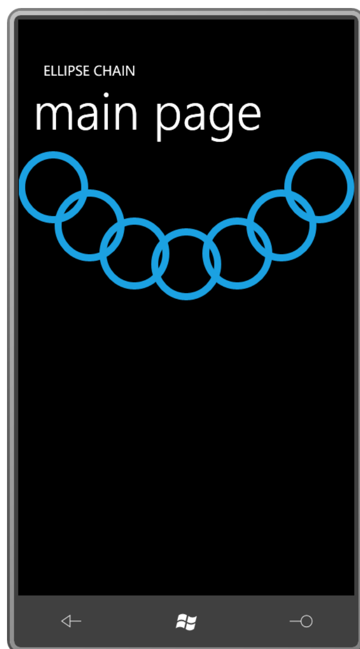
Второе подразумевает, что *Canvas* не имеет собственного места в системе компоновки Silverlight. (Это можно исправить, явно задав *Width* или *Height* для *Canvas*.) На самом деле это очень полезное свойство в случаях, когда требуется, чтобы элемент существовал где-то «вне» системы компоновки и не оказывал влияния на позиционирование других элементов.

Рассмотрим приложение, использующее *Canvas* для вывода на экран семи элементов *Ellipse* в виде цепочки с перекрывающимися звеньями. Объект *Style* (описанный в коллекции *Resources* самого *Canvas*) обеспечивает каждый эллипс конечными значениями *Width* и *Height*; в противном случае они не отображались бы.

Проект Silverlight: EllipseChain Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1">
    <Canvas>
        <Canvas.Resources>
            <Style x:Key="ellipseStyle"
                   TargetType="Ellipse">
                <Setter Property="Width" Value="100" />
                <Setter Property="Height" Value="100" />
                <Setter Property="Stroke" Value="{StaticResource PhoneAccentBrush}"
/>
                <Setter Property="StrokeThickness" Value="10" />
            </Style>
        </Canvas.Resources>

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="0" Canvas.Top="0" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="52" Canvas.Top="53" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="116" Canvas.Top="92" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="190" Canvas.Top="107" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="263" Canvas.Top="92" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="326" Canvas.Top="53" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Canvas.Left="380" Canvas.Top="0" />
    </Canvas>
</Grid>
```

Обратите внимание, что я удалил *Margin* панели содержимого, поэтому значения достигают 480. Вот как это выглядит на экране:



Canvas является идеальным контейнером для размещения элементов произвольным образом. Конечно, это больше относится к вопросам программирования векторной графики, чем компоновки элементов управления.

Но имеем этот абсолютно странный синтаксис, очень сильно отличающийся от всего, что мы видели до сих пор в XAML:

```
<Ellipse Style="{StaticResource ellipseStyle}"
         Canvas.Left="190" Canvas.Top="107" />
```

Эти свойства *Left* и *Top* позиционируют верхний правый угол элемента относительно верхнего правого угла *Canvas*. Свойства, кажется, определены классом *Canvas*, но заданы для элемента *Ellipse*! Когда я впервые увидел такой синтаксис много лет назад, я просто был сбит с толку. Зачем классу *Canvas* определять свойства *Left* и *Top*? Разве это должен делать не *FrameworkElement*?

Конечно, в графических средах разработки прошлого у всех элементов были свойства *Left* и *Top*, потому что так работала система.

Но в Silverlight эти свойства не имеют особого смысла. *Canvas* требует, чтобы для его дочерних элементов были заданы *Left* и *Top*, для других панелей этого не нужно. Для дочерних элементов других панелей, включая пользовательские панели, которые вам еще предстоит написать или даже понять, могут понадобиться совсем другие свойства.

Поэтому Silverlight поддерживает концепцию *присоединенных свойств*. Свойства *Left* и *Top* в самом деле определены классом *Canvas* (и как именно это делается, будет показано в главе 11), но задаются эти свойства для дочерних элементов *Canvas*. (Их можно задать и для элементов, не являющихся дочерними элементами *Canvas*, но они будут проигнорированы.)

Давайте рассмотрим приложение, в коде которого задаются эти присоединенные свойства. В приложении EllipseMesh (Сетка из эллипсов) в сетке для содержимого создается ряд перекрывающихся эллипсов. Файл XAML включает пустой *Canvas* с заданным обработчиком событий *SizeChanged*:

Проект Silverlight: EllipseMesh Файл: MainPage.xaml (фрагмент)

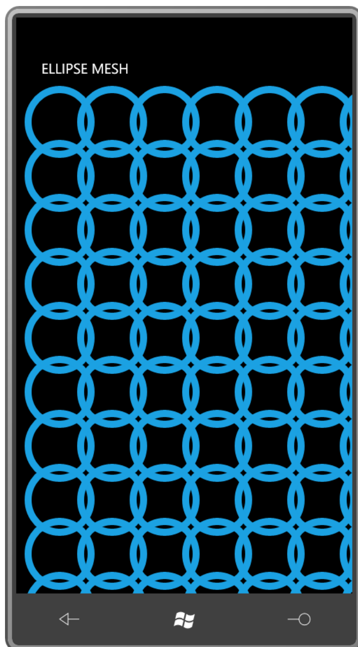
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Canvas Name="canvas"
            SizeChanged="OnCanvasSizeChanged" />
</Grid>
```

Несмотря на то что *Canvas* не имеет собственного места в системе компоновки, у него есть размер и событие *SizeChanged*. При каждом вызове *SizeChanged* обработчик события опустошает *Canvas* (просто для удобства) и заполняет его вновь новыми объектами *Ellipse*:

Проект Silverlight: EllipseMesh Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    void OnCanvasSizeChanged(object sender, SizeChangedEventArgs args)
    {
        canvas.Children.Clear();

        for (double y = 0; y < args.NewSize.Height; y += 75)
            for (double x = 0; x < args.NewSize.Width; x += 75)
            {
                Ellipse ellipse = new Ellipse
                {
                    Width = 100,
                    Height = 100,
                    Stroke = this.Resources["PhoneAccentBrush"] as Brush,
                    StrokeThickness = 10
                };

                Canvas.SetLeft(ellipse, x);
                Canvas.SetTop(ellipse, y);

                canvas.Children.Add(ellipse);
            }
    }
}
```

Вот как это выглядит на экране:



Эти два выражения задают присоединенные свойства *Left* и *Top*:

```
Canvas.SetLeft(ellipse, x);
Canvas.SetTop(ellipse, y);
```

Эти два статических метода определены классом *Canvas*. Их можно вызвать либо до, либо после добавления дочернего элемента в коллекцию *Children* элемента *Canvas*. Это статические методы, поэтому они могут вызываться даже, когда объекта *Canvas* еще не существует.

Еще более показательным является то, как эти статические методы определены в классе *Canvas*. Прямо в приложении EllipseMesh эти два статических метода можно заменить следующими выражениями:

```
ellipse.SetValue(Canvas.LeftProperty, x);
ellipse.SetValue(Canvas.TopProperty, y);
```

Эти эквивалентные вызовы подтверждают, что какие-то значения действительно были заданы для объектов *Ellipse*. Метод *SetValue* (Задать значение) определен классом *DependencyObject* (самый базовый класс в иерархии классов Silverlight), а *LeftProperty* (Свойство слева) и *RightProperty* (Свойство справа), несмотря на их имена, на самом деле являются статическими полями типа *DependencyProperty*, определенными *Canvas*.

Если я не ошибаюсь, *SetValue* выполняет доступ к внутреннему словарю, создаваемому и сохраняемому *DependencyObject*. При этом первый передаваемый в *SetValue* аргумент – это ключ словаря, и второй – значение. Когда *Canvas* компонует свои дочерние элементы в методе *ArrangeOverride*, он может обращаться к этим значениям для конкретного элемента *child*, используя следующий синтаксис:

```
double x = GetLeft(child);
double y = GetTop(child);
```

или эквивалентный ему:

```
double x = (double)child.GetValue(LeftProperty);
double y = (double)child.GetValue(TopProperty);
```

Метод *GetValue* выполняет доступ к внутреннему словарю в дочернем элементе и возвращает объект типа *object*, который должен быть приведен к типу *double*.

В главе 11 я покажу клон *Canvas* и то, как определять собственные присоединенные свойства.

Внимание! Я описал, как заменять в EllipseMesh вызовы *Canvas.SetLeft* и *Canvas.SetTop* эквивалентными вызовами *SetValue*. Но этот вызов:

```
Canvas.SetLeft(ellipse, 57);
```

не является эквивалентным данному вызову:

```
ellipse.SetValue(Canvas.LeftProperty, 57);
```

Второй аргумент *Canvas.SetLeft* должен быть типа *double*, но второй аргумент метода общего назначения *SetValue* определен типа *object*. Когда компилятор C# будет проводить синтаксический разбор этого вызова *SetValue*, он предположит, что число типа *int*. Ошибка будет выявлена только во время выполнения. Избежать проблемы поможет явное задание числа как *double*:

```
ellipse.SetValue(Canvas.LeftProperty, 57.0);
```

Мы говорим здесь о присоединенных свойствах *Left* и *Top* класса *Canvas*, но на самом деле в *Canvas* нет ни одного члена с именем *Left* или *Top*! *Canvas* определяет статические поля *LeftProperty* и *TopProperty* и статические методы *SetLeft*, *SetTop*, *GetLeft* и *GetTop*, но ничего с именами *Left* или *Top*. Представленный здесь синтаксис XAML:

```
<Ellipse Style="{StaticResource ellipseStyle}"
         Canvas.Left="190" Canvas.Top="107" />
```

на самом деле сформирован за счет вызовов *Canvas.SetLeft* и *Canvas.SetTop*.

Мы будем рассматривать и другие присоединенные свойства. В стандартном файле MainPage.xaml присоединенное свойство задано для корневого элемента:

```
shell:SystemTray.IsVisible="True"
```

Кстати, класс *SystemTray* существует исключительно в целях определения этого присоединенного свойства, обеспечивая возможность его задания для производных от *PhoneApplicationPage* классов. *PageApplicationFrame* проверяет значение этого свойства на каждой странице, выясняя, должна ли отображаться панель задач или нет.

Canvas и *ZIndex*

У *Canvas* есть третье присоединенное свойство, *ZIndex* (Индекс по оси Z), которое может использоваться для переопределения компоновки элементов по умолчанию.

Как мы видели, элементы в панели располагаются в том порядке, в каком они появляются в коллекции *Children*. Элементы, объявленные в коллекции раньше, перекрываются элементами, объявленными позже.

Изменить это поведение можно, задавая присоединенное свойство *Canvas.ZIndex* для одного или более элементов. Имя этого свойства относит нас к воображаемой оси Z, перпендикулярной к поверхности экрана. Элементы с большими индексами Z отображаются поверх (и могут совершенно заслонять) элементов с меньшими индексами Z. Если значения присоединенных свойств *Canvas.ZIndex* двух элементов одинаковые (а по умолчанию *Canvas.ZIndex* не задано ни для одного элемента, т.е. предполагается равным нулю), элементы расставляются в порядке, в каком они размещаются в коллекции *Children*.

Несмотря на то что это присоединенное свойство *Canvas.ZIndex* определено классом *Canvas*, оно может использоваться для любых типов панелей. При написании собственного класса

панели не стоит беспокоиться об индексах Z, о них автоматически позаботиться система компоновки.

Canvas и сенсорный ввод

В главе 8 было показано, как реализовать перемещение элементов по экрану посредством сенсорного ввода. Это делалось путем изменения объектов трансформаций, заданных в качестве значений свойства *RenderTransform*. Элементы можно перемещать и в рамках *Canvas*, задавая присоединенные свойства *Left* и *Top* в коде.

Рассмотрим простое приложение TouchCanvas (Сенсорный холст). В *Canvas* располагается три элемента *Ellipse* красного, зеленого и синего цвета:

Проект Silverlight: TouchCanvas Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Canvas Name="canvas">
        <Ellipse Canvas.Left="50"
                 Canvas.Top="50"
                 Width="100"
                 Height="100"
                 Fill="Red" />

        <Ellipse Canvas.Left="150"
                 Canvas.Top="150"
                 Width="100"
                 Height="100"
                 Fill="Green" />

        <Ellipse Canvas.Left="250"
                 Canvas.Top="250"
                 Width="100"
                 Height="100"
                 Fill="Blue" />
    </Canvas>
</Grid>
```

В файле кода перегружаются методы *OnManipulationStarted* и *OnManipulationDelta* класса *MainPage*. В задании свойства *ManipulationContainer* (Контейнер для манипуляций) для *Canvas* в первом перегруженном методе нет крайней необходимости.

Проект Silverlight: TouchCanvas Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        args.ManipulationContainer = canvas;
        base.OnManipulationStarted(args);
    }

    protected override void OnManipulationDelta(ManipulationDeltaEventArgs args)
    {
        UIElement element = args.OriginalSource as UIElement;
        Point translation = args.DeltaManipulation.Translation;
        Canvas.SetLeft(element, Canvas.GetLeft(element) + translation.X);
```

```
        Canvas.SetTop(element, Canvas.GetTop(element) + translation.Y);

        args.Handled = true;
        base.OnManipulationDelta(args);
    }
}
```

Перегруженный метод *OnManipulationDelta* перемещает один из эллипсов. Для этого он извлекает значения *Left* и *Top*, добавляет к ним коэффициенты переноса и повторно задает полученные значения этим свойствам. Все выполняется с помощью довольно коротких и четких выражений.

Grid всемогущий

Если требуется создать панель, выбирайте *Grid*. Он обеспечивает гибкость и мощь, оставаясь при этом простым и универсальным. В данной главе я продемонстрирую всего один пример приложения с использованием *Grid*, но лишь потому, что в этой книге таких примеров бесчисленное множество.

Grid напоминает HTML-таблицу, но имеет несколько отличий. *Grid* не выполняет форматирования, за него всецело отвечает компоновка. В *Grid* нет понятия заголовков или встроенных разделителей ячеек. Кроме того, в отличие от HTML-таблиц, использование *Grid* приветствуется.

Grid имеет определенное число строк и столбцов. Строки могут быть разной высоты, столбцы – разной ширины. Дочерний элемент *Grid* обычно занимает одну определенную строку и столбец, но также может охватывать несколько строк и несколько столбцов. Звучит красиво (и так оно и есть), но бесплатный сыр бывает только в мышеловке, за все приходится платить. В *StackPanel* или *Canvas* дочерние элементы можно добавлять произвольным образом, для *Grid* необходимо четко знать, сколько строк и столбцов требуется для размещения всех дочерних элементов. Строки и столбцы можно добавлять из кода во время выполнения, но если *Grid* полностью описывается в XAML, их количество необходимо знать заранее.

Реализация вложения в *Grid* вполне тривиальна, но не стоит увлекаться, особенно если в приложении выполняется частое обновление компоновки. Слишком сложное вложение может создавать сложности для компоновки.

Grid определяет два свойства: *RowDefinitions* и *ColumnDefinitions* (Описания столбцов). Это коллекции объектов *RowDefinition* (Описание строки) и *ColumnDefinition* (Описание столбца), соответственно. Эти объекты определяют высоту каждой строки и ширину каждого столбца. Они могут быть заданы тремя способами:

- с применением ключевого слова «Auto»
- заданием определенного количества пикселов
- с помощью символа «звездочка» и числа со «звездочкой»

Чаще всего используются первый и последний варианты. Первый указывает на то, что размер ячейки будет соответствовать размеру помещаемого в нее элемента. (Размер этого элемента *Grid* выясняет в ходе вызова метода *MeasureOverride* с использованием неограниченных размеров.) Строки и столбцы, отмеченные звездочкой, используются для пропорционального распределения оставшегося пространства.

Как было показано, очень часто в элементах *StackPanel* располагается дочерних элементов больше, чем может быть отображено на экране; *Grid* обычно описывается так, что этого не происходит.

Конкретная строка и столбец элемента определяется присоединенными свойствами *Grid.Row* и *Grid.Column*. Отсчет строк и столбцов ведется с нуля, начиная сверху слева. С помощью присоединенных свойств *Grid.RowSpan* и *Grid.ColumnSpan* можно задать количество строк и столбцов, занимаемых элементом.

Рассмотрим пример:

Проект Silverlight: SimpleGrid Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0"
               Grid.Column="0"
               Grid.ColumnSpan="2"
               Text="Heading at top of Grid"
               HorizontalAlignment="Center" />

    <Image Grid.Row="1"
           Grid.Column="0"
           Source="Images/BuzzAldrinOnTheMoon.png" />

    <Ellipse Grid.Row="1"
             Grid.Column="1"
             Stroke="{StaticResource PhoneAccentBrush}"
             StrokeThickness="6" />

    <TextBlock Grid.Row="2"
               Grid.Column="0"
               Grid.ColumnSpan="2"
               Text="Footer at bottom of Grid"
               HorizontalAlignment="Center" />
</Grid>
```

В данном примере я просто добавил описания строк и столбцов в существующую сетку для содержимого. Для каждого элемента *Grid* явно заданы *Grid.Row* и *Grid.Column*, но они могут быть опущены, если имеют нулевые значения. И верхний, и нижний элементы *TextBlock* занимают по два столбца, что обеспечивает их центрирование относительно всей сетки.

Два столбца распределены таким образом, что первый из них в два раза шире второго. Ширина первого столбца определяет размер *Image*, после чего в ячейке выполняется центрирование этого изображение по вертикали:

При изменении ориентации экрана строки и столбцы меняют размер, но общая компоновка сохраняется:

Попробуем задать свойства *HorizontalAlignment* и *VerticalAlignment* для этого *Grid*. Обнаружится, что размер сетки ограничен оригинальными размерами растрового изображения.

У *Grid* под именем *ContentPanel* имеется собственное присоединенное свойство *Grid.Row*, но оно относится ко второй строке родительского *Grid*, который называется *LayoutRoot*. Первую строку этого *Grid* занимает *StackPanel* с двумя заголовками.

И теперь, наконец, мы достигли такой концентрации знаний по Silverlight и XAML, когда уже ничего в MainPage.xaml не должно представлять загадки.

Глава 10
Панель приложения и элементы управления

Нет ничего удивительного в том, что Silverlight для Windows Phone поддерживает ряд стандартных элементов управления. Среди них *ScrollBar* (Полоса прокрутки) и *Slider* (Ползунок) для осуществления выбора из непрерывного диапазона значений, *TextBox* для ввода и редактирования текста, а также типовой набор кнопок, включая *CheckBox* (Флажок) для реализации опций вкл/выкл, *RadioButton* (Переключатель) для реализации группы взаимоисключающих опций и простой *Button* для запуска выполнения команд.

Мы уже располагаем достаточными знаниями и абсолютно готовы погрузиться в вопросы, связанные с элементами управления Silverlight. Кроме того, в данной главе приведены несколько действительно полезных приложений, которые можно классифицировать как реальные приложения для Windows Phone 7.

Но прежде чем переходить к обсуждению стандартных элементов управления, я хочу рассмотреть существующую для них альтернативу. В приложениях для Windows Phone 7 базовые команды и опции могут быть реализованы с помощью механизма, специально созданного для телефона с целью обеспечить пользователям унифицированное взаимодействие с устройствами.

Этим механизмом является *ApplicationBar* (Панель приложения), который обычно называют *app bar*.

Значки *ApplicationBar*

ApplicationBar выполняет те же задачи, что и меню или панель инструментов, которые можно встретить в обычном Windows-приложении. Он очень похож на эти традиционные структуры как внешне, так и функционально. Если в приложении необходимо реализовать лишь несколько кнопок для выполнения каких-то простых команд и, возможно, небольшое меню, используйте *ApplicationBar*. Silverlight вообще не предлагает никакого типового меню или панели инструментов (хотя, несомненно, их можно реализовать самостоятельно).

ApplicationBar и связанные с ним классы (*ApplicationBarIconButton* (Кнопка панели приложения) и *ApplicationBarMenuItem* (Элемент меню панели приложения)) описаны в пространстве имен *Microsoft.Phone.Shell*. Эти классы наследуются от *Object* и существуют абсолютно отдельно от всей иерархии классов *DependencyObject*, *UIElement* и *FrameworkElement*, используемых при создании обычных приложений на Silverlight. Строго говоря, *ApplicationBar не* является частью дерева визуальных элементов страницы.

Объект *ApplicationBar* всегда задается как значение свойства *ApplicationBar* объекта *PhoneApplicationPage*. Когда телефон располагается вертикально вверх, *ApplicationBar* находится внизу страницы и сохраняет свое местоположение при изменении ориентации экрана телефона. *ApplicationBar* не является настраиваемым.

ApplicationBar может включать до четырех кнопок. Иногда их называют значками, потому что они всегда представлены в виде каких-то изображений. В качестве изображений, как правило, используются PNG-файлы; изображение должно быть квадратным со стороной 48 пикселов и преимущественно прозрачным. Но его видимая часть должна быть белой и

занимать квадрат со стороной 26 пикселов в центре основного изображения. Если вы установили Expressions Blend for Windows Phone, коллекцию растровых изображений можно найти в папке C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.0\Icons. Прежде чем создавать собственные изображения, внимательно изучите предлагаемые.

Предположим, мы нашли эти растровые изображения. Они хранятся в двух папках: *light* (черные изображения на белом фоне) и *dark* (белые изображения с прозрачным фоном). Несмотря на то что изображения из папки *light* выглядят в Windows Explorer лучше, в приложениях для телефона следует всегда использовать соответствующие файлы из папки *dark*.

Проект MoviePlayer включает элемент *MediaElement* для воспроизведения фильма и *ApplicationBar* со значками для воспроизведения, приостановки воспроизведения, перемотки назад и вперед.

В проекте Visual Studio, использующем *ApplicationBar*, должна быть отдельная папка для значков. Щелкните правой кнопкой мыши имя проекта и выберите в меню Add и New Folder. (Или выберите Add New Folder (Добавить новую папку) в меню Project.) Присвойте этой папке имя *Images* или нечто подобное. Щелкните правой кнопкой мыши имя папки, выберите Add и Existing Item и перейдите к папке *dark* с загруженными растровыми изображениями. Для MoviePlayer I были выбраны:

- appbar.transport.ff.rest.png
- appbar.transport.pause.rest.png
- appbar.transport.play.rest.png
- appbar.transport.rew.rest.png

Это четыре стандартных изображения, ассоциированные с метафорой проигрывания видеопленки.

А теперь обязательно для каждого из этих файлов из папки *Images* откройте страницу Properties (Свойства). (Возможно, придется щелкнуть имя файла правой кнопкой мыши и выбрать Properties из меню.) Задайте полю Build Action значение Content. *ApplicationBar* не сможет найти изображения, если в качестве Build Action будет задано Resource.

ApplicationBar не является частью стандартного Silverlight, поэтому описание пространства имен XML должно связывать пространство имен XML «shell» с пространством имен .NET *Microsoft.Phone.Shell*. Это делается автоматически в файле MainPage.xaml:

```
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
```

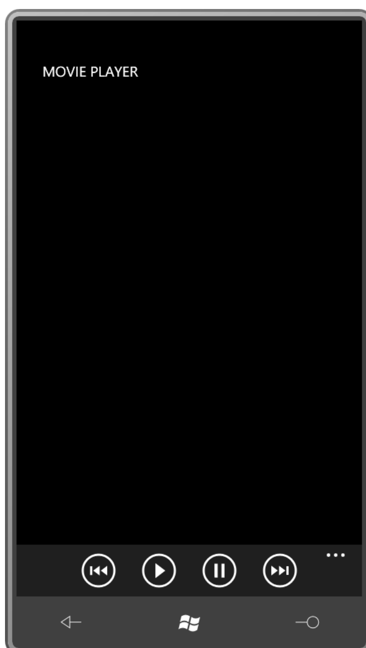
В конце файла MainPage.xaml представлен образец *ApplicationBar*. Можно раскомментировать и вносить изменения в него либо добавить собственный прямо перед закрывающим тегом *phone:PhoneApplicationPage*:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton
                IconUri="Images/appbar.transport.rew.rest.png"
                Text="rewind" />

        <shell:ApplicationBarIconButton
                IconUri="Images/appbar.transport.play.rest.png"
                Text="play" />

        <shell:ApplicationBarIconButton
                IconUri="Images/appbar.transport.pause.rest.png"
```

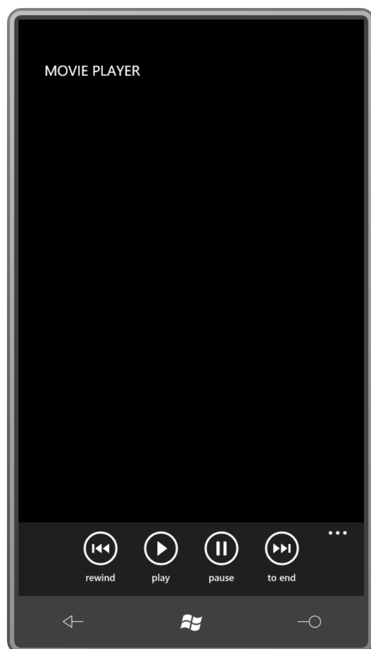
```
                Text="pause" />

        <shell:ApplicationBarIconButton
                IconUri="Images/appbar.transport.ff.rest.png"
                Text="to end" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

У *ApplicationBar* есть свойство *Buttons* (Кнопки), которое является свойством содержимого класса. В коллекции *Buttons* может быть не более четырех объектов *ApplicationBarIconButton*. Поля *IconUri* (URI значка) и *Text* являются обязательными! Текстовое описание должно быть коротким; все буквы преобразованы в строчные в целях удобства отображения.

Выполните сборку и развертывание приложения в том состоянии, в котором оно находится сейчас. На рисунке показано, как все это будет выглядеть на экране. (Я также удалил заголовок страницы.)



Реагируя на нажатие, каждый из значков меняет цвет на инверсный и немного подпрыгивает. Если нажать многоточие, кнопки поднимутся вверх, и под каждой из них будет выведено описание, хранящееся в свойстве *Text*:



Если перевернуть телефон на бок, *ApplicationBar* остается на том же месте, но если свойству *SupportedOrientations* было задано значение *PortraitOrLandscape*, значки повернуться соответственно текущей ориентации экрана:

Будьте внимательны с этим свойством и не используйте значки, которые могут запутать пользователя при альбомном расположении экрана. Если в одном значке имеется горизонтальная черта, и в другом – вертикальная, это может сбить с толку и пользователя, и разработчика.

ApplicationBar можно описать в XAML. Чтобы его не было видно на экране раньше времени, задайте его свойству *IsVisible* значение *false*:

```
<shell:ApplicationBar IsVisible="False">
```

Потом значение этого свойства можно изменить в коде. Нет смысла описывать атрибут *x:Name* для организации доступа к *ApplicationBar* из кода. Это необъяснимо, но факт: ссылаться на объект *ApplicationBar* по имени в коде нельзя. Добраться к нему можно только через свойство *ApplicationBar* объекта *MainPage*:

```
this.ApplicationBar.IsVisible = true;
```

ApplicationBar также описывает свойства *ForegroundColor* (Цвет переднего плана) и *BackgroundColor* (Цвет фона), которые следует игнорировать. Цвета *ApplicationBar* будут меняться по умолчанию при изменении и соответственно выбранной цветовой схеме телефона.

У *ApplicationBar* имеется свойство *Opacity*, под таким знакомым именем которого кроется весьма нетрадиционный эффект. Свойство *Opacity* относится не к переднему плану, а к фону *ApplicationBar*; оно никогда не оказывает влияния на сами изображения.

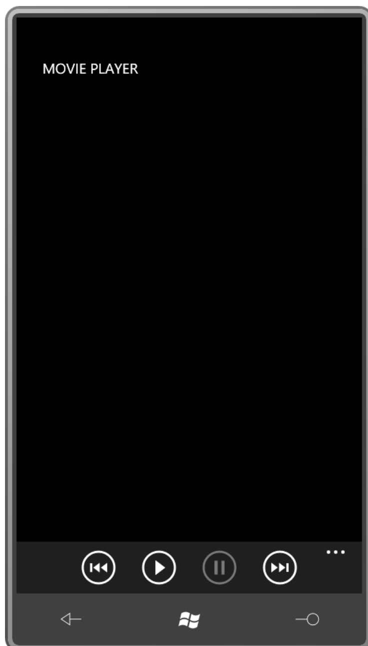
Свойство *Opacity* по умолчанию имеет значение 1, т.е. фон *ApplicationBar* непрозрачный. Цвет фона определяет ресурс *PhoneChromeBrush* (Кисть визуального стиля телефона): очень темный серый для темной цветовой темы и очень светлый серый для светлой темы.

Когда свойство *Opacity* равно 1, *ApplicationBar* занимает область отображения отдельно от содержимого страницы. При любых других значениях *Opacity* содержимое страницы и *ApplicationBar* совместно используют общую область. *ApplicationBar* всегда располагается поверх всего остального содержимого. Когда *Opacity* принимает значение 0, фон *ApplicationBar* становится совершенно прозрачным, и под его значками можно видеть любое содержимое, оказавшееся в этой же области сетки для содержимого. В документации для *Opacity* рекомендуется использовать значения 1, 0,5 или 0.

Если в какой-то момент времени какой-то из значков теряет свою актуальность, для отражения этого в *ApplicationBarIconButton* предусмотрено свойство *IsEnabled*, которому можно задать значение *false*. Например, кнопки Play и Pause (Пауза) не должны быть активными одновременно. Рассмотрим, как можно деактивировать кнопку Pause при запуске:

```
<shell:ApplicationBarIconButton
        IconUri="Images/appbar.transport.pause.rest.png"
        Text="Pause"
        IsEnabled="False" />
```

И вот как это выглядит на экране:



Эту кнопку можно активировать позже в коде, но, опять же, для этого не может использоваться *x:Name*. Это третья кнопка, следовательно, в коллекции *Buttons* она идет под индексом 2. Задать для нее свойство *IsEnabled* в коде можно следующим образом:

```
(this.ApplicationBar.Buttons[2] as ApplicationBarIconButton).IsEnabled = true;
```

Вероятно, одновременно с этим необходимо деактивировать кнопку Play:

```
(this.ApplicationBar.Buttons[1] as ApplicationBarIconButton).IsEnabled = false;
```

Если доступ к фильму происходит через Интернет, вероятно, целесообразно деактивировать все кнопки до тех пор, пока файл мультимедиа не будет открыт, и фильм сможет быть действительно воспроизведен.

Также приложение должно знать, когда произошло нажатие активной кнопки пользователем. Зададим обработчик событий *Click*:

```
<shell:ApplicationBarIconButton
        IconUri="Images/appbar.transport.play.rest.png"
        Text="Play"
        Click="OnAppBarPlayClick" />
```

В файле выделенного кода для реализации обработчика используется делегат *EventHandler* (Обработчик событий):

```
void OnAppbarPlayClick(object sender, EventArgs args)
{
    …
}
```

Окончательный вариант *ApplicationBar* в проекте MoviePlayer выглядит следующим образом:

Проект Silverlight: MoviePlayer Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar x:Name="appbar">
        <shell:ApplicationBarIconButton
                x:Name="appbarRewindButton"
                IconUri="Images/appbar.transport.rew.rest.png"
                Text="rewind"
                IsEnabled="False"
                Click="OnAppbarRewindClick" />

        <shell:ApplicationBarIconButton
                x:Name="appbarPlayButton"
                IconUri="Images/appbar.transport.play.rest.png"
                Text="play"
                IsEnabled="False"
                Click="OnAppbarPlayClick" />

        <shell:ApplicationBarIconButton
                x:Name="appbarPauseButton"
                IconUri="Images/appbar.transport.pause.rest.png"
                Text="pause"
                IsEnabled="False"
                Click="OnAppbarPauseClick" />

        <shell:ApplicationBarIconButton
                x:Name="appbarEndButton"
                IconUri="Images/appbar.transport.ff.rest.png"
                Text="to end"
                IsEnabled="False"
                Click="OnAppbarEndClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Да, здесь всем кнопкам назначены атрибуты *x:Name*, но вскоре мы увидим, что они переопределены в коде.

Сетка для содержимого включает *MediaElement* для воспроизведения фильма и два элемента *TextBlock* для вывода сообщений о состоянии и ошибках:

Проект Silverlight: MoviePlayer Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <MediaElement Name="mediaElement"
                  Source="http://www.charlespetzold.com/Media/Walrus.wmv"
                  AutoPlay="False"
                  MediaOpened="OnMediaElementMediaOpened"
                  MediaFailed="OnMediaElementMediaFailed"
                  CurrentStateChanged="OnMediaElementCurrentStateChanged"  />

    <TextBlock Name="statusText"
               HorizontalAlignment="Left"
               VerticalAlignment="Bottom" />

    <TextBlock Name="errorText"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom"
               TextWrapping="Wrap" />
</Grid>
```

Обратите внимание, что свойству *AutoPlay* элемента *MediaElement* задано значение *false*, так что фильм не начнет воспроизведение сразу же, как будет загружен. Вся эта функциональность обрабатывается в файле выделенного кода.

В конструкторе *MainPage* всем *ApplicationBarIconButton* назначаются соответствующие атрибуты *x:Name*. Это делается для удобства создания ссылок на них в остальном классе:

Проект Silverlight: MoviePlayer Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();

    // Переназначение имен, заданных в XAML-файле
    appbarRewindButton = this.ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    appbarPlayButton = this.ApplicationBar.Buttons[1] as ApplicationBarIconButton;
    appbarPauseButton = this.ApplicationBar.Buttons[2] as ApplicationBarIconButton;
    appbarEndButton = this.ApplicationBar.Buttons[3] as ApplicationBarIconButton;
}
```

Все четыре обработчика кнопок *ApplicationBar* занимают по одной строке кода:

Проект Silverlight: MoviePlayer Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarRewindClick(object sender, EventArgs args)
{
    mediaElement.Position = TimeSpan.Zero;
}

void OnAppbarPlayClick(object sender, EventArgs args)
{
    mediaElement.Play();
}

void OnAppbarPauseClick(object sender, EventArgs args)
{
    mediaElement.Pause();
}

void OnAppbarEndClick(object sender, EventArgs args)
{
    mediaElement.Position = mediaElement.NaturalDuration.TimeSpan;
}
```

Самой запутанной частью приложения для воспроизведения фильмов является активация и деактивация кнопок. Поскольку основной целью данной программы является демонстрация использования *ApplicationBar*, я применил здесь очень простой подход: кнопки Rewind (Назад) и End (Вперед) активируются, когда файл мультимедиа открыт, кнопки Play и Pause активируются на основании значения свойства *CurrentState* (Текущее состояние) элемента *MediaElement*:

Проект Silverlight: MoviePlayer Файл: MainPage.xaml.cs (фрагмент)

```
void OnMediaElementMediaFailed(object sender, ExceptionRoutedEventArgs args)
{
    errorText.Text = args.ErrorException.Message;
}

void OnMediaElementMediaOpened(object sender, RoutedEventArgs args)
{
    appbarRewindButton.IsEnabled = true;
    appbarEndButton.IsEnabled = true;
}

void OnMediaElementCurrentStateChanged(object sender, RoutedEventArgs args)
{
    statusText.Text = mediaElement.CurrentState.ToString();

    if (mediaElement.CurrentState == MediaElementState.Stopped ||
        mediaElement.CurrentState == MediaElementState.Paused)
    {
        appbarPlayButton.IsEnabled = true;
        appbarPauseButton.IsEnabled = false;
    }
    else if (mediaElement.CurrentState == MediaElementState.Playing)
    {
        appbarPlayButton.IsEnabled = false;
        appbarPauseButton.IsEnabled = true;
    }
}
```

Jot и параметры приложения

Приложение Jot (Записка), которое будет рассмотрено следующим – это одно из трех приложений данной главы, которое может быть полезным в повседневной жизни. Его идея возникла из приложения QuickNotes, которое также будет описано в данной главе несколько позже. По сути QuickNotes – это большой элемент *TextBox*, но содержимое этого *TextBox* хранится изолированном хранилище. Мы можем добавлять и удалять текст из него, но каждый раз, открывая это приложение, мы получаем текст, который оставили в нем в предыдущий раз. Данное приложение замечательно подходит для коротких заметок (как предполагает его название), потому что пользователю не приходится загружать или сохранять файлы, все делается автоматически.

Но обсудим QuickNotes более подробно в конце этой главы. А сейчас обратимся к аналогичному и более простому в использовании приложению Jot. Для работы с ним не требуется ни виртуальной, ни реальной клавиатуры, вводить текст или рисовать можно просто с помощью пальца.

Как будет показано, в QuickNotes все реализовано с использованием одного единственного текстового документа, и пользователь может без труда прокручивать его и вводить текст в любом месте. Приложение Jot использует элементы *Canvas*, и их фиксированный размер, кажется, подразумевает, что приложение должно поддерживать множество страниц для множества холстов.

Отображение сенсорного ввода в Jot реализовано с помощью класса *InkPresenter*, который берет начало с интерфейсов для планшетных ПК. *InkPresenter* наследуется от панели *Canvas*, что означает возможность использования его свойства *Children* для создания фонового изображения (традиционной желтой блокнотной странички, например). Или можно полностью игнорировать все возможности *InkPresenter*, обеспечиваемые происхождением от *Canvas*.

Главное назначение *InkPresenter* – отображение рукописного ввода, который он представляет как множество серий соединенных коротких линий – *полилиний*, если говорить на языке графики, или в данном контексте мы называем их *обводками*.

Точка на поверхности экрана описывается классом *StylusPoint* (Штрих стилуса) – это структура, определенная в пространстве имен *System.Windows.Input* и имеющая свойства *X* и *Y*, а также *PressureFactor* (Коэффициент нажима) для устройств, поддерживающих распознавание силы нажима (Windows Phone 7 не поддерживает этого).

Проводя по экрану пальцем (на сенсорных экранах) или стилусом (на планшетных ПК), можно создать множество кривых абсолютно невообразимых форм. Но независимо от того насколько запутанны кривые, они всегда представляются коллекцией объектов *StylusPoint*, которые в совокупности повторяют сложную кривую. Коллекция объектов *StylusPoint*, представляющих непрерывную линию, инкапсулируется в *Stroke*. Это класс из пространства имен *System.Windows.Ink*. Объект *Stroke* инкапсулирует не только точки кривой, но также ее цвет и толщину. Для этого в нем предусмотрены следующие два свойства:

- *StylusPoints* (Штрихи стилуса) типа *StylusPointCollection* (Коллекция штрихов стилуса)
- *DrawingAttributes* (Атрибуты рисования) типа *DrawingAttributes*

Stroke – это непрерывная линия, которая создается, когда пользователь касается экрана, повторяет все движения пальца по нему и завершается, когда пользователь отрывает палец от экрана. Со следующим касанием экрана создается другой *Stroke*. Множество объектов *Stroke* хранится в *StrokeCollection* (Коллекция обводок). Именно они и хранятся в объекте *InkPresenter*: *InkPresenter* описывает свойство *Strokes* типа *StrokeCollection* и формирует визуальное представление этих обводок, которые все вместе образуют непрерывную кривую.

Приложение Jot многостраничное, поэтому ему понадобится еще одна отдельная коллекция для хранения объектов *StrokeCollection* для каждой из страниц.

Мы должны знать все это заранее, потому что я собираюсь начать обсуждение приложения Jot с параметров приложения. Общая идея этого приложения состоит в том, что оно всегда возвращает пользователя туда, на чем он остановился в прошлый раз. Jot не нужны никакие промежуточные данные для захоронения, потому что оно рассматривает его просто как обычный запуск и завершение выполнения. Jot использует изолированное хранилище, потому что оно сохраняет одинаковые данные как при деактивации (захоронении), так и при закрытии, и ему необходимо загружать эти данные как при запуске, так и при повторной активации (т.е. возрождении после захоронения).

Параметры приложения для Jot инкапсулированы в специально предназначенный для этих целей класс *JotAppSettings* (Параметры приложения Jot). Экземпляр *JotAppSettings* сериализуется и сохраняется в изолированном хранилище. Этот класс также имеет методы для сохранения и загрузки параметров. В проект необходимо включить ссылку на библиотеку System.Xml.Serialization; в *JotAppSettings* требуется несколько нестандартных директив *using* для *System.Collection.Generic*, *System.IO*, *System.IO.IsolatedStorage* и *System.Xml.Serialization*.

Рассмотрим открытые свойства *JotAppSettings*, составляющие параметры приложения:

Проект Silverlight: Jot Файл: JotAppSettings.cs (фрагмент)

```
public List<StrokeCollection> StrokeCollections { get; set; }
public int PageNumber { set; get; }
public Color Foreground { set; get; }
public Color Background { set; get; }
public int StrokeWidth { set; get; }
```

Для каждой страницы Jot отображает один *StrokeCollection*, таким образом, приложению необходимо хранить коллекцию объектов *StrokeCollection*. Приложение начинает выполнение со страницы, обозначенной *PageNumber* (Номер страницы).

При первом запуске Jot выбирает цвета *Foreground* и *Background* из темы системы. Но при этом приложение реализует возможность изменения этих цветов для целей рисования, предполагая, что пользователь может предпочесть тему «белое на черном» для телефона в целом, но «черное на белом» для Jot. Поэтому приложение сохраняет и загружает цвета явно. По умолчанию свойство *StokeWidth* (Толщина обводки) имеет значение 3 (значение по умолчанию для *InkPresenter*), но пользователь может изменить его и задать 1 или 5.

Для сохранения этих элементов я попытался использовать класс *IsolatedStorageSettings*, но у меня ничего не получилось, поэтому я обратился к традиционному изолированному хранилищу. Рассмотрим метод *Save*:

Проект Silverlight: Jot Файл: JotAppSettings.cs (фрагмент)

```
public void Save()
{
    IsolatedStorageFile iso = IsolatedStorageFile.GetUserStoreForApplication();
    IsolatedStorageFileStream stream = iso.CreateFile("settings.xml");
    StreamWriter writer = new StreamWriter(stream);

    XmlSerializer ser = new XmlSerializer(typeof(JotAppSettings));
    ser.Serialize(writer, this);

    writer.Close();
    iso.Dispose();
}
```

Метод *Save* создает (или воссоздает) в изолированном хранилище файл settings.xml, получает *StreamWriter* (Модуль записи в поток), ассоциированный с этим файлом, и затем использует класс *XmlSerializer* для сериализации этого конкретного экземпляра класса *JotAppSettings*.

Метод *Load* является статическим, потому что должен создавать экземпляр *JotAppSettings* путем десериализации файла из хранилища. Если этого файла не существует – это означает, что приложение выполняется впервые – он просто создает новый экземпляр.

Проект Silverlight: Jot Файл: JotAppSettings.cs (фрагмент)

```
public static JotAppSettings Load()
{
    JotAppSettings settings;
    IsolatedStorageFile iso = IsolatedStorageFile.GetUserStoreForApplication();

    if (iso.FileExists("settings.xml"))
    {
        IsolatedStorageFileStream stream = iso.OpenFile("settings.xml",
```

```
FileMode.Open);
        StreamReader reader = new StreamReader(stream);

        XmlSerializer ser = new XmlSerializer(typeof(JotAppSettings));
        settings = ser.Deserialize(reader) as JotAppSettings;

        reader.Close();
    }
    else
    {
        // Создаем и инициализируем новый объект JotAppSettings
        settings = new JotAppSettings();
        settings.StrokeCollections = new List<StrokeCollection>();
        settings.StrokeCollections.Add(new StrokeCollection());
    }

    iso.Dispose();
    return settings;
}
```

Конструктор класса задает для некоторых (но не всех) свойств значения по умолчанию:

Проект Silverlight: Jot Файл: JotAppSettings.cs (фрагмент)

```
public JotAppSettings()
{
    this.PageNumber = 0;
    this.Foreground = (Color)Application.Current.Resources["PhoneForegroundColor"];
    this.Background = (Color)Application.Current.Resources["PhoneBackgroundColor"];
    this.StrokeWidth = 3;
}
```

Этот конструктор вызывается, и когда метод *Load* явно создает новый экземпляр при выполнении приложения впервые, и когда выполняется десериализация файла из изолированного хранилища. Во втором случае все значения по умолчанию в конструкторе заменяются. Однако располагать эти настройки в конструкторе является целесообразным, на случай если позже в приложение будет добавлен новый параметр приложения. Этого параметра не будет в существующем файле в изолированном хранилище, но ему будет присвоено значение по умолчанию в этом конструкторе.

Изначально я тоже создавал коллекцию *StrokeCollection* в конструкторе:

```
settings.StrokeCollections = new List<StrokeCollection>();
settings.StrokeCollections.Add(new StrokeCollection());
```

Но обнаружилось, что в этом случае метод *Deserialize* (Десериализовать) класса *XmlSerializer* не создает новый объект *List*, а просто использует тот, который был создан в конструкторе, из-за чего я оставался с одним новым пустым *StrokeCollection* в *List* при каждом выполнении приложения! Поэтому я перенес этот код в метод *Load*.

Методы *Save* и *Load* класса *JotAppSettings* вызываются только из App.xaml.cs в ходе обработки четырех событий *PhoneApplicationService*, обсуждаемых нами в главе 6. Эти события сигнализируют о запуске, деактивации, активации и завершении приложения. App.xaml.cs также предоставляет параметры приложения как открытое свойство:

Проект Silverlight: Jot Файл: App.xaml.cs (фрагмент)

```
public partial class App : Application
{
    // Параметры приложения
```

```
    public JotAppSettings AppSettings { set; get; }

    …

    private void Application_Launching(object sender, LaunchingEventArgs e)
    {
        AppSettings = JotAppSettings.Load();
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
       AppSettings = JotAppSettings.Load();
    }

    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        AppSettings.Save();
    }

    private void Application_Closing(object sender, ClosingEventArgs e)
    {
        AppSettings.Save();
    }
}
```

В рамках *MainPage* все ссылки на свойства, составляющие параметры приложения, выполняются через свойство *AppSettings* класса *App*.

Jot и сенсорный ввод

Область содержимого в Jot невелика, но играет важную роль:

Проект Silverlight: Jot Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <InkPresenter Name="inkPresenter" />
</Grid>
```

Как предполагает его имя, класс *InkPresenter* формирует виртуальный рукописный фрагмент на основе ввода со стилуса или касания. *InkPresenter* не распознает этот ввод самостоятельно, об этом должен позаботиться разработчик. (И в Silverlight нет встроенной функциональности для распознавания рукописного ввода, но нам ничего не мешает реализовать ее самостоятельно.)

В файле выделенного кода с помощью директивы *using* необходимо подключить пространство имен *System.Windows.Ink* и описать всего два закрытых поля:

Проект Silverlight: Jot Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    JotAppSettings appSettings = (Application.Current as App).AppSettings;
    Dictionary<int, Stroke> activeStrokes = new Dictionary<int, Stroke>();

    public MainPage()
    {
        InitializeComponent();

        inkPresenter.Strokes =
appSettings.StrokeCollections[appSettings.PageNumber];
```

```
        inkPresenter.Background = new SolidColorBrush(appSettings.Background);
        …
        TitleAndAppbarUpdate();

        Touch.FrameReported += OnTouchFrameReported;
    }
    …
}
```

Первое поле обеспечивает удобный доступ к параметрам приложения, предоставляемым классом *App*. Второе – предназначено для сохранения данных мультисенсорного ввода. Конструктор инициализирует свойства *Strokes* и *Background* класса *InkPresenter*, используя данные приложения, и завершается заданием обработчика для низкоуровневого события *Touch.FrameReported*. (Метод *TitleAndAppbarUpdate* (Обновить заголовок и панель приложения) обсудим несколько позже.)

Я решил использовать простой сенсорный ввод, потому что данное приложение не выполняет никакой обработки. Оно просто принимает координаты точек касания соответственно перемещениям пальца по экрану. Событие *Touch.FrameReported* позволяет приложению принимать ввод от нескольких касаний одновременно, но, безусловно, это требует определенных ухищрений на практике.

Если помните, при использовании события *Touch.FrameReported* каждое касание идентифицируется целочисленным ID с момента прикосновения к экрану до момента снятия касания. В данном приложении в результате прикосновения формируется новый *Stroke* для свойства *Strokes* класса *InkPresenter*. Для сохранения ID касания, ассоциированного с соответствующим *Stroke*, предусмотрен словарь (*Dictionary*), который я назвал *activeStrokes* (Активные обводки):

```
Dictionary<int, Stroke> activeStrokes = new Dictionary<int, Stroke>();
```

Рассмотрим обработчик события *OnTouchFrameReported* (Уведомление о касании):

Проект Silverlight: Jot   Файл: MainPage.xaml.cs (фрагмент)

```
void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
{
    TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);

    if (primaryTouchPoint != null && primaryTouchPoint.Action == TouchAction.Down)
        args.SuspendMousePromotionUntilTouchUp();

    TouchPointCollection touchPoints = args.GetTouchPoints(inkPresenter);

    foreach (TouchPoint touchPoint in touchPoints)
    {
        Point pt = touchPoint.Position;
        int id = touchPoint.TouchDevice.Id;

        switch (touchPoint.Action)
        {
            case TouchAction.Down:
                Stroke stroke = new Stroke();
                stroke.DrawingAttributes.Color = appSettings.Foreground;
                stroke.DrawingAttributes.Height = appSettings.StrokeWidth;
                stroke.DrawingAttributes.Width = appSettings.StrokeWidth;
                stroke.StylusPoints.Add(new StylusPoint(pt.X, pt.Y));

                inkPresenter.Strokes.Add(stroke);
                activeStrokes.Add(id, stroke);
                break;
```

```
            case TouchAction.Move:
                activeStrokes[id].StylusPoints.Add(new StylusPoint(pt.X, pt.Y));
                break;

            case TouchAction.Up:
                activeStrokes[id].StylusPoints.Add(new StylusPoint(pt.X, pt.Y));
                activeStrokes.Remove(id);

                TitleAndAppbarUpdate();
                break;
        }
    }
}
```

При первом касании экрана (о чем сигнализирует значение *TouchAction.Down* свойства *Action* (Действие)) этот метод создает новый объект *Stroke*. Он добавляется в коллекцию *Strokes* класса *InkPresenter*, и *InkPresenter* повторно визуализирует его при добавлении каждой новой точки. Данный объект *Stroke* вместе с его ID также сохраняется в словаре *activeStrokes*. Это позволяет достраивать объект *Stroke* с каждым новым событием *TouchAction.Move*. Когда касание прекращается, запись удаляется из словаря.

Jot и *ApplicationBar*

ApplicatonBar приложения Jot включает четыре кнопки: для добавления новой страницы, возвращения к предыдущей странице, перехода на следующую страницу и удаления текущей страницы. (Если текущая страница является единственной, удаляется только то, что было введено на ней.) Для каждой кнопки описан собственный обработчик события *Click*:

Проект Silverlight: Jot   Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton x:Name="appbarAddButton"
                                        IconUri="/Images/appbar.add.rest.png"
                                        Text="add page"
                                        Click="OnAppbarAddClick" />

        <shell:ApplicationBarIconButton x:Name="appbarLastButton"
                                        IconUri="/Images/appbar.back.rest.png"
                                        Text="last page"
                                        Click="OnAppbarLastClick" />

        <shell:ApplicationBarIconButton x:Name="appbarNextButton"
                                        IconUri="/Images/appbar.next.rest.png"
                                        Text="next page"
                                        Click="OnAppbarNextClick" />

        <shell:ApplicationBarIconButton x:Name="appbarDeleteButton"
                                        IconUri="/Images/appbar.delete.rest.png"
                                        Text="delete page"
                                        Click="OnAppbarDeleteClick" />
        <shell:ApplicationBar.MenuItems>
            <shell:ApplicationBarMenuItem Text="swap colors"
                                          Click="OnAppbarSwapColorsClick" />

            <shell:ApplicationBarMenuItem Text="light stroke width"
                                          Click="OnAppbarSetStrokeWidthClick" />

            <shell:ApplicationBarMenuItem Text="medium stroke width"
                                          Click="OnAppbarSetStrokeWidthClick" />

            <shell:ApplicationBarMenuItem Text="heavy stroke width"
```

```
                                      Click="OnAppbarSetStrokeWidthClick" />
        </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Меню тоже реализовано посредством свойства-элемента *MenuItems* (Пункты меню) с коллекцией объектов *ApplicationBarMenuItem* (Пункты меню панели приложения). Пункты меню отображаются при нажатии многоточия на панели приложения. Каждый из них включает только короткую строку текста, написанную строчными буквами. (Количество пунктов меню не должно превышать пяти, текст должен быть не более 20 символов.) Для первого пункта меню (для изменения цветов) предусмотрен собственный обработчик события *Click*; остальные три совместно используют один обработчик *Click*.

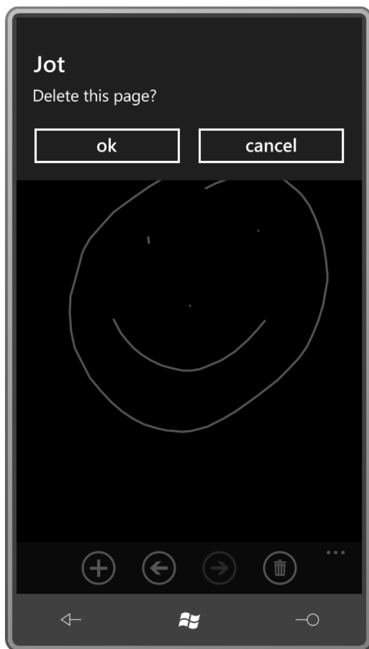
Рассмотрим обработчики событий *Click* для всех четырех кнопок:

Проект Silverlight: Jot Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarAddClick(object sender, EventArgs args)
{
    StrokeCollection strokes = new StrokeCollection();
    appSettings.PageNumber += 1;
    appSettings.StrokeCollections.Insert(appSettings.PageNumber, strokes);
    inkPresenter.Strokes = strokes;
    TitleAndAppbarUpdate();
}

void OnAppbarLastClick(object sender, EventArgs args)
{
    appSettings.PageNumber -= 1;
    inkPresenter.Strokes = appSettings.StrokeCollections[appSettings.PageNumber];
    TitleAndAppbarUpdate();
}

void OnAppbarNextClick(object sender, EventArgs args)
{
    appSettings.PageNumber += 1;
    inkPresenter.Strokes = appSettings.StrokeCollections[appSettings.PageNumber];
    TitleAndAppbarUpdate();
}

void OnAppbarDeleteClick(object sender, EventArgs args)
{
    MessageBoxResult result = MessageBox.Show("Delete this page?", "Jot",
                                              MessageBoxButton.OKCancel);

    if (result == MessageBoxResult.OK)
    {
        if (appSettings.StrokeCollections.Count == 1)
        {
            appSettings.StrokeCollections[0].Clear();
        }
        else
        {
            appSettings.StrokeCollections.RemoveAt(appSettings.PageNumber);

            if (appSettings.PageNumber == appSettings.StrokeCollections.Count)
                appSettings.PageNumber -= 1;

            inkPresenter.Strokes =
appSettings.StrokeCollections[appSettings.PageNumber];
        }
        TitleAndAppbarUpdate();
    }
}
```

Некоторую сложность представляет лишь удаление страницы. Но обратите внимание, что оно начинается с запроса подтверждения от пользователя посредством вызова *MessageBox.Show*! В данном контексте окно сообщений кажется очень архаичным, но в телефоне ничто не может сравниться с ним по простоте, и это самое главное его преимущество. Если требуется проинформировать пользователя о чем-то и получить подтверждение о том, что информация принята к сведению посредством кнопки OK, или если необходимо задать вопрос, требующий ответа в виде OK и Cancel, окно сообщений – оптимальный выбор.

Окно сообщений отображается вверху экрана и деактивирует все приложение до тех пор, пока пользователь не закроет его:



Я покажу более сложные диалоговые окна далее в данной главе и также в главе 14.

Четыре пункта меню обрабатываются здесь:

Проект Silverlight: Jot Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarSwapColorsClick(object sender, EventArgs args)
{
    Color foreground = appSettings.Background;
    appSettings.Background = appSettings.Foreground;
    appSettings.Foreground = foreground;
    inkPresenter.Background = new SolidColorBrush(appSettings.Background);

    foreach (StrokeCollection strokeCollection in appSettings.StrokeCollections)
        foreach (Stroke stroke in strokeCollection)
            stroke.DrawingAttributes.Color = appSettings.Foreground;
}

void OnAppbarSetStrokeWidthClick(object sender, EventArgs args)
{
    ApplicationBarMenuItem item = sender as ApplicationBarMenuItem;

    if (item.Text.StartsWith("light"))
        appSettings.StrokeWidth = 1;

    else if (item.Text.StartsWith("medium"))
        appSettings.StrokeWidth = 3;
```

```
    else if (item.Text.StartsWith("heavy"))
        appSettings.StrokeWidth = 5;
}
```

При изменении цветов новые цвета должны сохраняться в параметрах приложения, но также должны быть изменены текущие цвета всех объектов *Stroke* на каждой странице. К счастью для этого потребуется лишь несколько циклов *foreach*.

Метод *OnAppbarSetStrokeWidthClick* (По щелчку панели приложения задать толщину обводки) обслуживает три подобных пункта меню. Обратите внимание, что объект *sender* – это конкретный *ApplicationBarMenuItem*, который был выбран. Логика здесь проста, но зависит от значений свойства *Text* этих трех элементов. В данном случае можно использовать методику без зависимости от значения *Text*, например, три отдельных обработчика.

Мы уже видели несколько обращений к *TitleAndAppbarUpdate* – заключительному методу файла выделенного кода для *MainPage*. Рассмотрим его:

Проект Silverlight: Jot Файл: MainPage.xaml.cs (фрагмент)

```
void TitleAndAppbarUpdate()
{
    pageInfoTitle.Text = String.Format(" - PAGE {0} OF {1}",
                                   appSettings.PageNumber + 1,
                                   appSettings.StrokeCollections.Count);

    appbarLastButton.IsEnabled = appSettings.PageNumber > 0;
    appbarNextButton.IsEnabled =
            appSettings.PageNumber < appSettings.StrokeCollections.Count - 1;
    appbarDeleteButton.IsEnabled = (appSettings.StrokeCollections.Count > 1) ||
                                   (appSettings.StrokeCollections[0].Count > 0);
}
```

Последние три выражения обеспечивают деактивацию различных кнопок панели приложения, если в них нет необходимости в текущем контексте. (Логика обработчиков строится на том факте, что кнопка не будет вызвана, если представляемая ею опция недействительна.) В данных выражениях могут использоваться имена, присвоенные трем кнопкам в файле XAML, потому что я переназначил эти имена в конструкторе *MainPage*:

Проект Silverlight: Jot Файл: MainPage.xaml.cs (фрагмент)

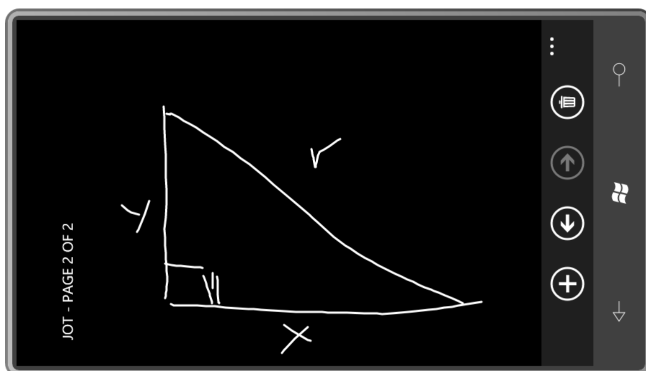
```
public MainPage()
{
    InitializeComponent();
    …
    appbarLastButton = this.ApplicationBar.Buttons[1] as ApplicationBarIconButton;
    appbarNextButton = this.ApplicationBar.Buttons[2] as ApplicationBarIconButton;
    appbarDeleteButton = this.ApplicationBar.Buttons[3] as ApplicationBarIconButton;
    …
}
```

Первое выражение в *TitleAndAppbarUpdate* ссылается на *TextBlock*, добавленный мною в заголовок приложения в XAML-файле:

Проект Silverlight: Jot Файл: MainPage.xaml (фрагмент)

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="24,24,0,12"
            Orientation="Horizontal">
    <TextBlock x:Name="ApplicationTitle" Text="JOT"
               Style="{StaticResource PhoneTextNormalStyle}"
               Margin="12 0 0 0" />
    <TextBlock Name="pageInfoTitle"
               Style="{StaticResource PhoneTextNormalStyle}"
               Margin="0" />
</StackPanel>
```

Приложение Jot не поддерживает изменения ориентации. Если пользователь набросал на экране какое-то изображение, определенного размера и пропорций, вряд ли он хочет, чтобы картинка была перевернута, и часть ее оказалась за границами экрана. Но рисовать в Jot можно, ориентируя экран в любом направлении, приложению все равно, как пользователь держал экран при вводе графических данных:



Элементы и элементы управления

ApplicationBar существует абсолютно независимо от обычной иерархии классов Silverlight в особом разделе Silverlight для Windows Phone. Далее в данной главе мы вернемся к более привычному царству классов.

Большинство визуальных объектов, обсуждаемых до сих пор в данной книге, часто называют *элементами*, главным образом потому, что в иерархии классов они наследуются от *FrameworkElement*:

Object
    DependencyObject (абстрактный)
        UIElement (абстрактный)
            FrameworkElement (абстрактный)

К этим производным от *FrameworkElement* относятся *TextBlock*, *Image*, *Border*, *MediaElement*, *Shape* (который является родительским классом для *Rectangle* и *Ellipse*) и *Panel*, который является родителем для *Grid*, *StackPanel* и *Canvas*.

Далее в данной главе я собираюсь сосредоточиться на некоторых классах, которые наследуются от *Control*:

Object
    DependencyObject (абстрактный)
        UIElement (абстрактный)
            FrameworkElement (абстрактный)
                Control (абстрактный)

Большинство производных от *Control* описываются в пространстве имен *System.Windows.Controls*, но некоторые из них скрываются в пространстве имен *System.Windows.Controls.Primitives*.

Для разработчиков, имеющих опыт работы в графических средах, термин *элемент управления* более привычен, чем *элемент*. И здесь возникает вопрос: если визуальные объекты, которые обычно называют *элементами управления*, в Silverlight более правильно называть *элементами*, в чем разница между элементами и визуальными объектами, которые *действительно* являются элементами управления Silverlight?

Важным отличием между этими концепциями является то, что элементы обычно относятся к *представлению*, тогда как элементы управления предназначены для *взаимодействия*. В Silverlight *Control* является родителем таких классов, как *Button*, *Slider* и *TextBox*, т.е. такое объяснение кажется вполне убедительным. Можно также заметить, что класс *Control* реализует свойство *IsEnabled*, а также три свойства, участвующих в навигации с использованием клавиши табуляции: *IsTabStop* (Является позицией табуляции), *TabIndex* (Индекс позиции табуляции) и *TabNavigation* (Навигация табуляцией).

С другой стороны, события пользовательского ввода для клавиатуры, мыши, стилуса и касания описываются классом *UIElement*, так что элементы, такие как *TextBlock* и *Image*, могут получать пользовательский ввод и – с соответствующей поддержкой в разметке или коде – отвечать на него.

Возможно, более существенным различием является то, что элементы управления образованы элементами. Элементы можно рассматривать как *визуальные примитивы*. Элементы управления составлены из этих элементов и других элементов управления. Например, *Button* это не что иное, как *Border* с некоторым содержимым. Как правило, в роли этого содержимого выступает *TextBlock*, но этим список не ограничивается. *Slider* это не что иное, как пара элементов *Rectangle* и несколько специальных элементов управления *RepeatButton* (Кнопка повтора).

Визуальные элементы, производные от *Control*, всегда определяются деревом производных от *FrameworkElement*. Это дерево также может включать объекты, которые тоже наследуются от *Control*, но описываются деревом других производных от *FrameworkElement* и *Control*.

Несмотря на то что визуальные элементы *Control* всегда входят в состав дерева производных от *FrameworkElement*, это дерево не является неизменным. Его можно заменить и полностью переопределить визуальные элементы элемента управления. Класс *Control* описывает свойство *Template* (Шаблон) типа *ControlTemplate* (Шаблон элемента управления). В главе 16 мы рассмотрим, как заменять шаблон.

Переопределение визуальных элементов с помощью шаблонов – мощный инструмент для настройки элементов управления. Однако сложно себе представить, как можно переопределить визуальные элементы производного от *FrameworkElement* класса. Конечно, внешний вид *TextBlock* будет разным в зависимости от значений свойств *Text*, *FontFamily* *FontSize* и *Foreground*, но какой-либо целесообразности в том, чтобы делать *TextBlock* визуально отличным по каким-то иным параметрам, нет. Напротив, вспомним, как сильно менялся внешний вид базовой кнопки в различных версиях Windows.

В Silverlight невозможно унаследовать класс от *FrameworkElement*. (Вообще фактически можно, но сделать что-либо полезное в унаследованном классе не получится.) И, кроме *Panel*, большинство производных от *FrameworkElement* являются запечатанными. *Shape* не запечатанный, но наследоваться от него нельзя, а все производные от *Shape* классы запечатанные.

Но можно наследоваться от *Control* и многих его производных. Один из производных от *Control* классов, *UserControl*, существует исключительно в целях создания пользовательских классов. Он позволяет не только настраивать визуальные элементы существующего элемента управления с помощью шаблонов, но и создавать собственные элементы. Однако независимо от метода реализации визуальные элементы элемента управления всегда будут определены как дерево элементов и других элементов управления.

В документации для *Control* можно увидеть, что этот класс значительно расширяет *FrameworkElement* удобными свойствами. Среди них несколько свойств шрифта, которые ассоциированы с *TextBlock*: *FontSize*, *FontFamily*, *FontStyle*, *FontWeight* и *FontStretch*. Также *Control* вводит несколько свойств из класса *Border – BorderBrush*, *BorderThickness*, *Background* и *Padding* – и еще два свойства, касающихся содержимого элемента управления – *HorizontalContentAlignment* и *VerticalContentAlignment*. Сам класс *Control* не использует эти свойства, они определены исключительно для удобства классов, наследуемых от *Control*.

Аналогично *Control* определяет ряд защищенных виртуальных методов, соответствующих событиям пользовательского ввода, описанным в *UIElement*. Для разработчиков на платформе Windows Phone 7 самое большое значение имеют, безусловно, те из этих методов, которые участвуют в обработке мультисенсорного ввода: *OnManipulationStarted*, *OnManipulationDelta* и *OnManipulationCompleted*.

RangeBase и *Slider*

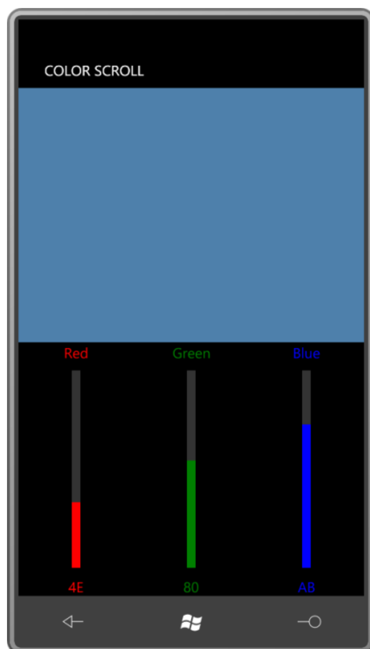
Из приведенных ранее в данной книге примеров можно догадаться, что полосы прокрутки и ползунки не имеют такого широкого применения в приложениях для устройств с мультисенсорным вводом, как в средах, где основной ввод осуществляется посредством мыши. В примере предыдущей главы *ScrollViewer* отвечает непосредственно на касание, а не на какие-либо манипуляции с его полосами прокрутки, которые существуют практически исключительно как концепции.

Но при этом полосы прокрутки и ползунки по-прежнему пригодятся для задач выбора из непрерывного диапазона значений. Этим элементам управления отведен данный небольшой подраздел иерархии классов:

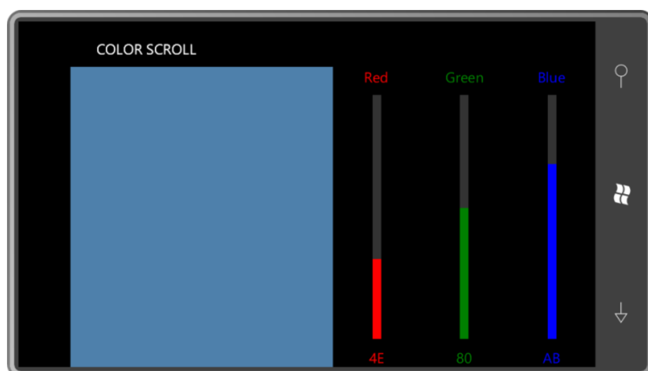
Control (абстрактный)
- *RangeBase* (абстрактный)
  - *ProgressBar*
  - *ScrollBar* (запечатанный)
  - *Slider*

Класс *RangeBase* (Диапазон) описывает свойства *Minimum* (Минимум), *Maximum* (Максимум), *SmallChange* (Небольшое изменение) и *LargeChange* (Большое изменение) для определения параметров прокрутки, а также свойство *Value* для хранения выбора пользователя и событие *ValueChanged* (Значение изменилось), которое сигнализирует об изменении значения *Value*. (Обратите внимание, что *ProgressBar* (Индикатор выполнения) также наследуется от *RangeBase*, но его свойство *Value* всегда управляется программно, а не задается пользователем.)

Я остановил свой выбор на *Slider* в качестве примера, потому что его версия в Windows Phone 7 кажется более соответствующей приложению для телефона, чем *ScrollBar*. Наша задача – с помощью трех элементов управления *Slider* создать приложение ColorScroll, имеющее следующий интерфейс:



Перемещая красный, зеленый и синий ползунки, пользователь определяет составной цвет. Чтобы сделать приложение более интересным, я решил добавить функциональность изменения ориентации визуальных элементов при повороте телефона на бок:



Самый простой способ создать такой интерфейс – с помощью вложенных сеток. Возьмем сетку с тремя строками и тремя столбцами, в которых располагаются три элемента управления *Slider* и шесть элементов *TextBlock*. Эту сетку поместим в другую сетку, включающую всего две ячейки. В ее второй ячейке (не занятой первым *Grid*) располагается элемент *Rectangle*, и значением его свойства *Fill* является *SolidColorBrush*, цвет которого определяется значениями, выбранными с помощью ползунков.

Больший *Grid* с двумя ячейками является обычным *Grid* под именем *ContentPanel*. Являются ли эти две ячейки двумя строками или двумя столбцами, определяется в файле выделенного кода на основании текущего значения свойства *Orientation*.

XAML-файл включает коллекцию *Resources* с описаниями *Style* для *TextBlock* и *Slider*:

Проект Silverlight: ColorScroll Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="textStyle" TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
    </Style>
```

```
    <Style x:Key="sliderStyle" TargetType="Slider">
        <Setter Property="Minimum" Value="0" />
        <Setter Property="Maximum" Value="255" />
        <Setter Property="Orientation" Value="Vertical" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Style всего с одним *Setter* кажется несколько излишним, но он не помешает, если в будущем предполагается добавить другой *Setter* для *Margin* или *FontSize*. Диапазон допустимых значений по умолчанию для *Slider* – от 0 до 10. Я изменил его, чтобы привести в соответствие однобайтовому значению.

У *ScrollBar* и *Slider* есть собственные свойства *Orientation*, совершенно не связанные со свойством *Orientation* класса *PhoneApplicationPage*, но имеющие некоторое отношение к свойству *Orientation* класса *StackPanel*, поскольку совместно используют одно и то же перечисление *Orientation* со значениями *Horizontal* и *Vertical*.

По умолчанию свойство *Orientation* класса *Slider* имеет значение *Horizontal*. (Для *ScrollBar* это значение *Vertical*; в чем разница, я никогда не понимал.)

По умолчанию предельно верхнее положение вертикального *Slider* ассоциировано со значением *Maximum*. Это вполне подходит для данного приложения, но положение вещей можно изменить, задав свойству *IsDirectionReversed* (Изменить направление) значение *true*.

Рассмотрим панель содержимого полностью:

Проект Silverlight: ColorScroll Файл: MainPage.xaml (фрагмент)

```
<Grid  x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Rectangle Name="rect"
               Grid.Row="0"
               Grid.Column="0" />

    <Grid Name="controlGrid"
          Grid.Row="1"
          Grid.Column="0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <!-- Красный столбец -->
        <TextBlock Grid.Column="0"
                   Grid.Row="0"
                   Text="Red"
                   Foreground="Red"
                   Style="{StaticResource textStyle}" />

        <Slider Name="redSlider"
                Grid.Column="0"
                Grid.Row="1"
```

```
                Foreground="Red"
                Style="{StaticResource sliderStyle}"
                ValueChanged="OnSliderValueChanged" />

        <TextBlock Name="redText"
                   Grid.Column="0"
                   Grid.Row="2"
                   Text="0"
                   Foreground="Red"
                   Style="{StaticResource textStyle}" />

        <!-- Зеленый столбец -->
        <TextBlock Grid.Column="1"
                   Grid.Row="0"
                   Text="Green"
                   Foreground="Green"
                   Style="{StaticResource textStyle}" />

        <Slider Name="greenSlider"
                Grid.Column="1"
                Grid.Row="1"
                Foreground="Green"
                Style="{StaticResource sliderStyle}"
                ValueChanged="OnSliderValueChanged" />

        <TextBlock Name="greenText"
                   Grid.Column="1"
                   Grid.Row="2"
                   Text="0"
                   Foreground="Green"
                   Style="{StaticResource textStyle}" />

        <!-- Синий столбец -->
        <TextBlock Grid.Column="2"
                   Grid.Row="0"
                   Text="Blue"
                   Foreground="Blue"
                   Style="{StaticResource textStyle}" />

        <Slider Name="blueSlider"
                Grid.Column="2"
                Grid.Row="1"
                Foreground="Blue"
                Style="{StaticResource sliderStyle}"
                ValueChanged="OnSliderValueChanged" />

        <TextBlock Name="blueText"
                   Grid.Column="2"
                   Grid.Row="2"
                   Text="0"
                   Foreground="Blue"
                   Style="{StaticResource textStyle}" />
    </Grid>
</Grid>
```

Для всех событий *ValueChanged* всех элементов управления *Slider* задан один и тот же обработчик. Этот обработчик использует очень простой подход и не утруждает себя определять, какой именно из *Slider* сформировал событие:

Проект Silverlight: ColorScroll Файл: MainPage.xaml.cs (фрагмент)

```
void OnSliderValueChanged(object sender, RoutedPropertyChangedEventArgs<double>
args)
{
    Color clr = Color.FromArgb(255, (byte)redSlider.Value,
                                    (byte)greenSlider.Value,
```

```
                                          (byte)blueSlider.Value);

    rect.Fill = new SolidColorBrush(clr);

    redText.Text = clr.R.ToString("X2");
    greenText.Text = clr.G.ToString("X2");
    blueText.Text = clr.B.ToString("X2");
}
```

Как можно заметить, XAML-файл не инициализирует свойство *Value* ни одного из *Slider*, и вот почему.

При создании страницы создаются различные элементы и элементы управления, обработчики событий подключаются к событиям и задаются значения свойств. Когда при создании этой страницы свойству *Value* любого из *Slider* задается какое-либо значение, этот *Slider* формирует событие *ValueChanged*. Весьма вероятно, что это приведет к вызову метода *OnSliderValueChanged* (При изменении значения ползунка) класса *MainPage* до того, как страница будет полностью создана. Но метод *OnSliderValueChanged* ссылается и на другие элементы дерева визуальных элементов. Если эти элементы не будут существовать на этот момент, возникнет ошибка времени выполнения.

Хотите увидеть это в действии? Попробуйте задать

```
<Setter Property="Value" Value="128" />
```

в описании *Style* для *Slider*.

Формирование события до того, как дерево визуальных элементов полностью построено, является общей проблемой. Чтобы избежать этого, можно либо повысить надежность обработчиков событий проверкой элементов и элементов управления на эквивалентность *null*, либо можно сделать то, что сделал я в ColorScroll: все свойства, формирующие события на странице, задаются в конструкторе класса после вызова метода *InitializeComponent*, когда дерево визуальных элементов уже полностью сформировано:

Проект Silverlight: ColorScroll Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();

    redSlider.Value = 128;
    greenSlider.Value = 128;
    blueSlider.Value = 128;
}
```

Для обработки изменения ориентации телефона *MainPage* перегружает свой метод *OnOrientationChanged*. Среди аргументов этого события имеется свойство *Orientation* типа *PageOrientation*.

Полезно знать, что значениями перечисления *PageOrientation* являются битовые флаги со следующими значениями:

None	0000-0000
Portrait	0000-0001
Landscape	0000-0010
PortraitUp	0000-0101
PortraitDown	0000-1001

LandscapeLeft	0001-0010
LandscapeRight	0010-0010

Проверка заданной ориентации осуществляется посредством побитовой операции ИЛИ между свойством *Orientation* и членами *Portrait* или *Landscape* с последующей проверкой на ненулевой результат. Это несколько упрощает код:

Проект Silverlight: ColorScroll Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnOrientationChanged(OrientationChangedEventArgs args)
{
    ContentPanel.RowDefinitions.Clear();
    ContentPanel.ColumnDefinitions.Clear();

    // Альбомный
    if ((args.Orientation & PageOrientation.Landscape) != 0)
    {
        ColumnDefinition coldef = new ColumnDefinition();
        coldef.Width = new GridLength(1, GridUnitType.Star);
        ContentPanel.ColumnDefinitions.Add(coldef);

        coldef = new ColumnDefinition();
        coldef.Width = new GridLength(1, GridUnitType.Star);
        ContentPanel.ColumnDefinitions.Add(coldef);

        Grid.SetRow(controlGrid, 0);
        Grid.SetColumn(controlGrid, 1);
    }
    // Портретный
    else
    {
        RowDefinition rowdef = new RowDefinition();
        rowdef.Height = new GridLength(1, GridUnitType.Star);
        ContentPanel.RowDefinitions.Add(rowdef);

        rowdef = new RowDefinition();
        rowdef.Height = new GridLength(1, GridUnitType.Star);
        ContentPanel.RowDefinitions.Add(rowdef);

        Grid.SetRow(controlGrid, 1);
        Grid.SetColumn(controlGrid, 0);
    }
    base.OnOrientationChanged(args);
}
```

Объект *ContentPanel* должен переключаться между двумя строками для портретного и двумя столбцами для альбомного режима отображения, поэтому он создает объекты *GridDefinition* (Описание сетки) и *ColumnDefinition* для новой ориентации. (Или он мог бы создать эти коллекции заранее и просто переключаться между ними. Или он мог бы создать объект *Grid* 2 x 2 в XAML-файле и задать неиспользуемой строке или столбцу нулевую высоту или ширину.)

Элемент *Rectangle* всегда располагается в ячейке с параметрами *Grid.Row* и *Grid.Column* равными нулю. Но присоединенные свойства *Grid.Row* и *Grid.Column* сетки *controlGrid* (Сетка элементов управления) должны быть заданы с использованием синтаксиса, который мы обсуждали в предыдущей главе.

В следующей главе мы рассмотрим, как наследоваться от класса *UserControl* , чтобы сделать приложение модульным и превратить его в элемент управления.

Простой *Button*

Стандартная кнопка Silverlight намного более гибкая, чем кнопки *ApplicationBar*, и при этом с ней легче работать. Нет ничего проще, чем поместить *Button* в сетку для содержимого:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Content="Click me!" />
</Grid>
```

По умолчанию *Button* заполняет весь *Grid*:

Эта кнопка обведена простой белой рамкой и выводит на экран строку текста, которая задана в ее свойстве *Content*. Если поместить *Button* в горизонтальный *StackPanel*, ее ширина будет точно такой, какая необходима для вмещения ее содержимого. Противоположный эффект имеет место, если изменить ориентацию *StackPanel* на *Vertical*. Или можно задать свойствам *HorizontalAlignment* и *VerticalAlignment* любое другое значение, отличное от *Stretch*.

Очевидно, что обычный *Button* был немного изменен для телефона. В нем немного расширена область вокруг рамки для создания большей мишени для касания.

В роли рамки в *Button* фактически выступает *Border*, и содержимым *Button* (в данном примере) является *TextBlock*. Ранее я упоминал, что класс *Control* определяет ряд свойств, которые обычно ассоциированы с *Border* и *TextBlock*. Некоторые из этих свойств можно задать следующим образом:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Content="Click me!"
            FontSize="48"
            FontStyle="Italic"
            Foreground="Red"
            Background="Blue"
            BorderThickness="10"
            BorderBrush="Yellow"
            Padding="20"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />
</Grid>
```

Как можно надеяться (или, возможно, опасаться), эти настройки будут отражены во внешнем виде кнопки:

Класс также определяет свойства *HorizontalContentAlignment* (Выравнивание содержимого по горизонтали), *VerticalContentAlignment* (Выравнивание содержимого по вертикали) и *Padding*, которые можно задать следующим образом:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Content="Click me!"
            Padding="50 100"
            HorizontalContentAlignment="Right"
            VerticalContentAlignment="Bottom" />
</Grid>
```

Теперь содержимое располагается в нижнем правом углу, но отстоит на 50 пикселов от правого и на 100 пикселов от нижнего края кнопки:

Класс *Control* описывает свойство *IsEnabled*. Если ему задано значение *false*, *Button* становится тусклым (неактивным) и не реагирует на касания.

Практически всегда требуется задать обработчик события *Click* класса *Button*, что позволяет знать, когда произошло нажатие кнопки. Событие *Click* формируется только, когда пользователь нажимает кнопку и снимает палец с экрана без перемещений по нему. Если свойству *ClickMode* (Режим нажатия) задано значение *Press* (Нажатие), *Button* формирует событие *Click*, как только палец касается экрана.

Концепция свойства *Content*

Button наследуется от класса *Control*, но также и от *ContentControl* (Элемент управления содержимым). *ContentControl* – это класс, описывающий свойство *Content* кнопки. Свойство *Content* можно вынести как свойство-элемент:

```
<Button>
    <Button.Content>

    </Button.Content>
</Button>
```

Но вот что странно – в него невозможно вставить текст:

```
<!-- Не работает! -->
<Button>
    <Button.Content>
        Click this button!
    </Button.Content>
</Button>
```

В этом синтаксисе нет никакой ошибки, но Silverlight не допускает его. Для того чтобы действительно сделать возможным вставку текста, необходимо включить описание пространства имен XML для пространства имен *System* (Система):

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

После этого можно вставлять строку между тегами, которые явно указывают синтаксическому анализатору XAML на то, что строка на самом деле типа *String*:

```
<Button>
    <Button.Content>
        <system:String>Click this button¹</system:String>
    </Button.Content>
</Button>
```

Как и в любом производном от *ContentControl*, мы можем опустить теги свойства-элемента для *Content*:

```
<Button>
    <system:String>Click this button</system:String>
</Button>
```

Свойство *Content* типа *object*, и мы на самом деле *можем* использовать в качестве значения *Content* практически все, что угодно:

```
<Button>
   <system:Double>1E5</system:Double>
</Button>
```

То что данное значение интерпретируется как число, можно понять, когда в *Button* будет отображено 10000. Попробуем следующее:

```
<Button>
    <system:DateTime>October 1, 2010, 9:30 PM</system:DateTime>
</Button>
```

---

[1] Щелкните эту кнопку (*прим. переводчика*).

Это будет отображено как 10/1/2010 9:30:00 PM. Сначала метод *Parse* для *Double* или *DateTime* выполняет синтаксический разбор текста, который должен быть отображен объектом *Button*. Затем метод *ToString* полученного объекта *Double* или *DateTime* обеспечивает его текстовое визуальное представление.

В *Button* можно поместить объект типа *Color*:

```
<Button>
    <Color>Cyan</Color>
</Button>
```

Но на кнопке будет выведено шестнадцатеричное представление значения этого цвета: «#FF00FFFF». Можно попробовать задать *SolidColorBrush*:

```
<Button>
    <SolidColorBrush Color="Cyan" />
</Button>
```

Но этот варианта даже еще хуже. Теперь будет выведен текст: «System.Windows.Media.SolidColorBrush». *SolidColorBrush* не определяет метода *ToString*, поэтому на экран выводится полное имя класса. *SolidColorBrush* может использоваться как значение свойств *Foreground*, *Background* или *BorderBrush* (Кисть рамки) кнопки, но не для *Content*.

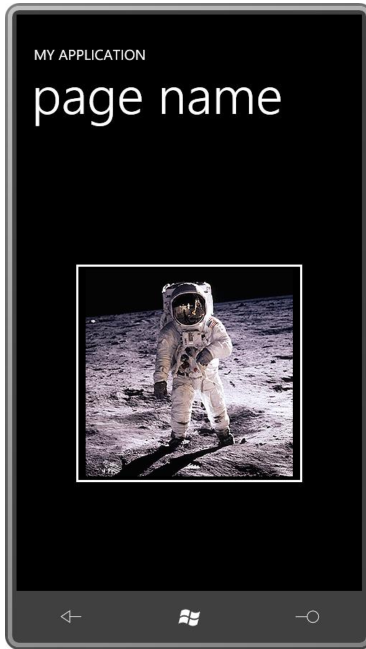
Тем не менее существует вполне определенная причина того, что свойство *Content* кнопки типа *object*. В главе 16 я покажу, как создать *DataTemplate* (Шаблон данных) для отображения объектов, таких как *SolidColorBrush*. А до тех пор, если хотите, чтобы *Button* отображал что-то отличное от просто текста, задавайте в качестве свойства *Content* объект, наследуемый от *FrameworkElement*. Например, чтобы вывести на кнопке форматированный текст, явно задайте в качестве значения *Content* объект *TextBlock*:

```
<Button>
    <TextBlock>
        Click <Run FontStyle="Italic">this</Run> button!
    </TextBlock>
</Button>
```

Или можно вставить элемент *Image*:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Image Source="Images/BuzzAldrinOnTheMoon.png"
           Stretch="None" />
</Button>
```

На экран будет выведено следующее:



Свойство *Content* типа *object*, поэтому в качестве его значения не может быть задано множество объектов, но это может быть одна из панелей (*Panel*):

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <StackPanel>
        <Image Source="Images/BuzzAldrinOnTheMoon.png"
               Stretch="None" />
        <TextBlock Text="Click this button!"
                   TextAlignment="Center" />
    </StackPanel>
</Button>
```

На рисунке показана получаемая кнопка, на которой выведены и изображение, и текст:

И да, в кнопку можно вставить другую кнопку или ползунок, вот только как пользователи отнесутся к этому?

Мы уже рассматривали производные от *ContentControl*: *ScrollViewer*, который обсуждался в прошлой главе, наследуется от *ContentControl*. В качестве значения свойства *Content* класса *ScrollViewer* часто используют *StackPanel*, но можно также задать *Image* большего, чем кнопка, размера. *PhoneApplicationFrame* также наследуется от *ContentControl* через *Frame*, но обычно этот класс используется несколько иначе, чем все остальные производные от *ContentControl*, потому что должен реализовывать навигацию по страницам.

ContentControl не единственный класс, имеющий свойство *Content*. *UserControl* – класс, от которого наследуется *PhoneApplicationPage* через *Page* – также описывает свойство *Content*. Естественно предположить, что *ContentControl* и *UserControl* каким-то образом взаимосвязаны. Как показывает данная иерархия классов, они являются элементами одного уровня:

Control (абстрактный)
- *ContentControl*
  - *Frame*
    - *PhoneApplicationFrame*
- *UserControl*
  - *Page*
    - *PhoneApplicationPage*

Свойство *Content* в *ContentControl* типа *object*, свойство *Content* в *UserControl* типа *UIElement*, т.е. оно несколько менее универсальное.

Наследование от *UserControl* используется очень широко (само его имя предполагает это), и мы рассмотрим данный вопрос следующей главе.

Стили темы и приоритетность

Проведем небольшой, но любопытный эксперимент. Поместим в сетку для содержимого простой *TextBlock* с очень большим размером текста (*FontSize*):

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Hello!"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Как известно, задание *FontSize* можно перенести из тега *TextBlock* в *PhoneApplicationPage*, и эффект останется тем же:

```
<phone:PhoneApplicationPage …
                            FontSize="96"
                            … >
        …
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <TextBlock Text="Hello!"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center" />
        </Grid>
        …
</phone:PhoneApplicationPage>
```

Это пример наследования свойств в действии. Теперь поместим *TextBlock* в *Button*. Размер текста можно задать, используя свойство *FontSize* объекта *TextBlock*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button HorizontalAlignment="Center"
            VerticalAlignment="Center">
```

```
        <TextBlock Text="Hello!"
                   FontSize="96" />
    </Button>
</Grid>
```

Или того же эффекта можно достичь, задавая *FontSize* самого *Button*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button HorizontalAlignment="Center"
            VerticalAlignment="Center"
            FontSize="96">
        <TextBlock Text="Hello!" />
    </Button>
</Grid>
```

А вот если задать *FontSize* для *PhoneApplicationPage*, ничего не получится, хотя, кажется, наследование свойств должно обеспечивать передачу этого значения в *TextBlock*:

```
<phone:PhoneApplicationPage …
                            FontSize="96"
                            … >
        …
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Button HorizontalAlignment="Center"
                    VerticalAlignment="Center">
                <TextBlock Text="Hello!" />
            </Button>
        </Grid>
        …
</phone:PhoneApplicationPage>
```

Но ничего не получается. Что-то мешает *TextBlock* наследовать это значение *FontSize*.

Класс *Button* описан в библиотеке System.Windows. Также эта библиотека содержит стиль и шаблон по умолчанию для *Button*, что называют *стилем темы*. Для *Button* стиль темы включает свойство *FontSize*. Этого нет в обычном Silverlight, но разработчики Windows Phone 7, вероятно, решили, что по умолчанию текст в *Button* должен быть несколько крупнее, чтобы обеспечить большую мишень для касания. Поэтому они включили в стандартный стиль темы свойство *FontSize*, и этот параметр имеет приоритет над наследованием свойств.

Вспомним схему приоритетности свойств, которую мы составили в главе 7, и дополним ее:

Локальные параметры имеют приоритет над

Параметрами стиля, которые являются более приоритетными, чем

Стиль темы, который является более приоритетным, чем

Унаследованные свойства, которые имеют приоритет над

Значениями по умолчанию

Иерархия класса Button

Это полная иерархия классов ветки класса *ButtonBase* (Базовая кнопка):

Control (абстрактный)
  ContentControl
    ButtonBase (абстрактный)
      Button
      HyperlinkButton
      RepeatButton (запечатанный)

ToggleButton
CheckBox
RadioButton

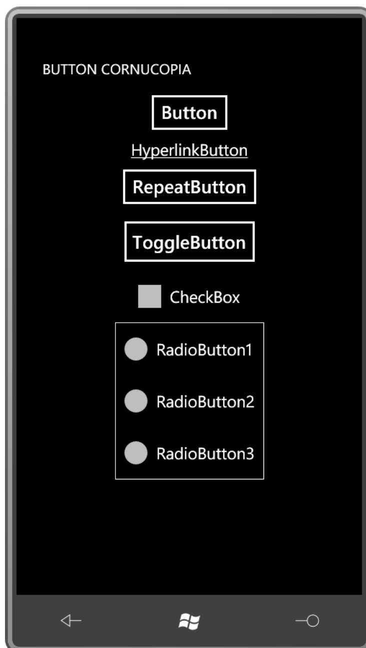
Именно *ButtonBase* описывает событие *Click* и свойство *ClickMode*.

В проекте ButtonCornucopia (Многообразие кнопок) создаются экземпляры всех этих кнопок с минимально необходимым набором свойств:

Проект Silverlight: ButtonCornucopia Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <Button Content="Button" HorizontalAlignment="Center" />
        <HyperlinkButton Content="HyperlinkButton" HorizontalAlignment="Center" />
        <RepeatButton Content="RepeatButton" HorizontalAlignment="Center" />
        <ToggleButton Content="ToggleButton" HorizontalAlignment="Center" />
        <CheckBox Content="CheckBox" HorizontalAlignment="Center" />

        <Border BorderBrush="White"
                BorderThickness="1"
                HorizontalAlignment="Center">
            <StackPanel>
                <RadioButton Content="RadioButton1" />
                <RadioButton Content="RadioButton2" />
                <RadioButton Content="RadioButton3" />
            </StackPanel>
        </Border>
    </StackPanel>
</Grid>
```

Кнопки и расстояния между ними могут казаться очень большими, но не забывайте, что они являются мишенями для касания и должны обеспечивать достаточное пространство для этого:



HyperlinkButton (Гиперссылка) используется в Silverlight для перехода на определенную страницу и включает свойство *NavigateUri* (URI перехода) для этой цели. *RepeatButton* при

удерживании кнопки формирует множество событий *Click*. Его основное применение – в *ScrollBar* и *Slider*, и его редко используют где-либо еще.

Между *ToggleButton* (Выключатель) и *CheckBox* нет функциональной разницы. Они отличаются лишь внешне, и их визуальное представление можно изменять с помощью шаблона (как будет продемонстрировано в главе 16). В наборе инструментов Silverlight for Windows Phone Toolkit появляется производный от *ToggleButton* под именем *ToggleSwitchButton* (Тумблер), который можно увидеть в разделе Settings телефона, работающего под управлением Windows Phone.

Выбор одного из *RadioButton* обусловливает снятие выбора со всех остальных *RadioButton*. Для объектов *RadioButton* не предусмотрено никакого специального контейнера. Просто поместите два или более *RadioButton* на панель (практически всегда для этого используется *StackPanel*) и они будут синхронизироваться самостоятельно. *RadioButton* описывает свойство *GroupName* (Имя группы), позволяющее различать группы переключателей, которые могут располагаться на одной панели.

Обычно *ToggleButton* и *CheckBox* являются визуальной реализацией величины типа Boolean. Каждый последующий щелчок обеспечивает их включение или выключение. Но в *ToggleButton* описано свойство *IsThreeState* (Три состояния), которое позволяет включать третье «неопределенное» состояние. Как правило, оно используется только для *CheckBox*; в *ToggleButton* даже не предусмотрено визуального представления для такого неопределенного состояния.

Следовательно, свойство *IsChecked*, определенное *ToggleButton*, не типа *bool*. Оно типа *Nullable<bool>* (опять *bool?*) с тремя допустимыми значениями: *true*, *false* и *null*. Чтобы задать свойству *IsChecked* значение *null* в XAML, можно воспользоваться специальным расширением разметки:

```
IsChecked="{x:Null}"
```

Для использования в роли обычного переключателя *IsChecked*, как правило, приводят к типу *bool*.

ToggleButton определяет три события: *Checked* (Установлен) формируется, когда флажок установлен, *Unchecked* (Снят) – когда флажок снят, и *Indeterminate* (Не определен) – когда объект переходит в третье состояние. В большинстве случаев приложению, использующему *ToggleButton* или *CheckBox*, необходимо обрабатывать и события *Checked*, и события *Unchecked*, но это можно делать с помощью одного обработчика событий.

Реализация секундомера

Очень полезным приложением в телефоне является секундомер. Также это идеальный пример использования как *ToggleButton*, так и класса *Stopwatch* (Секундомер), который описан в пространстве имен *System.Diagnostics*.

Я намеренно использовал прописные буквы в написании имени проекта StopWatch (Секундомер), чтобы избежать путаницы с .NET-классом *Stopwatch*. Я решил сделать приложение более интересным и реализовал отображение истекшего времени в трех разных форматах соответственно членам следующего перечисления:

Проект Silverlight: StopWatch Файл: ElapsedTimeFormat.cs

```
namespace StopWatch
{
    public enum ElapsedTimeFormat
```

```
    {
        HourMinuteSecond,
        Seconds,
        Milliseconds
    }
}
```

В StopWatch формат отображения истекшего времени является параметром приложения, поэтому он предоставляется как открытое свойство в классе *App*. Как это обычно бывает с параметрами приложения, формат отображения истекшего времени сохраняется в изолированном хранилище, когда приложение деактивируется или закрывается, и извлекается при запуске или активации приложения:

Проект Silverlight: StopWatch Файл: App.xaml.cs (фрагмент)

```
public partial class App : Application
{
    // Параметры приложения
    public ElapsedTimeFormat ElapsedTimeFormat { set; get; }

    …

    private void Application_Launching(object sender, LaunchingEventArgs e)
    {
        LoadSettings();
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        LoadSettings();
    }

    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        SaveSettings();
    }

    private void Application_Closing(object sender, ClosingEventArgs e)
    {
        SaveSettings();
    }

    void LoadSettings()
    {
        IsolatedStorageSettings settings =
IsolatedStorageSettings.ApplicationSettings;

        if (settings.Contains("elapsedTimeFormat"))
            ElapsedTimeFormat = (ElapsedTimeFormat)settings["elapsedTimeFormat"];
        else
            ElapsedTimeFormat = ElapsedTimeFormat.HourMinuteSecond;
    }

    void SaveSettings()
    {
        IsolatedStorageSettings settings =
IsolatedStorageSettings.ApplicationSettings;
        settings["elapsedTimeFormat"] = ElapsedTimeFormat;
        settings.Save();
    }
}
```

Описание области содержимого в XAML-файле немного более пространное, чем можно ожидать, потому что включает тип «диалогового окна», используемого пользователем для

выбора формата отображения истекшего времени. Но чтобы не пугать читателя, я привожу здесь лишь часть области содержимого, связанную с функциональностью секундомера. Она включает только *ToggleButton* для включения и выключения секундомера и *TextBlock* для отображения истекшего времени.

Проект Silverlight: StopWatch Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

    <!-- Экран секундомера -->
    <Grid VerticalAlignment="Center"
          Margin="25 0">
        <Grid.RowDefinitions>
             <RowDefinition Height="Auto" />
             <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Name="elapsedText"
                   Text="0"
                   Grid.Row="0"
                   FontFamily="Arial"
                   FontSize="{StaticResource PhoneFontSizeExtraLarge}"
                   TextAlignment="Center"
                   Margin="0 0 0 50"/>

        <ToggleButton Name="startStopToggle"
                      Content="Start"
                      Grid.Row="1"
                      Checked="OnToggleButtonChecked"
                      Unchecked="OnToggleButtonChecked" />
    </Grid>

    <!-- Прямоугольник для имитации неактивного состояния -->
    …
    <!-- "Диалоговое окно" для выбора формата TimeSpan -->
    …
</Grid>
```

Файл выделенного кода описывает всего три поля и должен обязательно включать директивы *using* для *System.Diagnostics* и *System.Globaliztion*.

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Stopwatch stopwatch = new Stopwatch();
    TimeSpan suspensionAdjustment = new TimeSpan();
    string decimalSeparator = NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;

    public MainPage()
    {
        InitializeComponent();
        DisplayTime();
    }
    …
    void DisplayTime()
    {
        TimeSpan elapsedTime = stopwatch.Elapsed + suspensionAdjustment;
        string str = null;

        switch ((Application.Current as App).ElapsedTimeFormat)
        {
            case ElapsedTimeFormat.HourMinuteSecond:
```

```
                str = String.Format("{0:D2} {1:D2} {2:D2}{3}{4:D2}",
                                    elapsedTime.Hours, elapsedTime.Minutes,
                                    elapsedTime.Seconds, decimalSeparator,
                                    elapsedTime.Milliseconds / 10);
                break;

            case ElapsedTimeFormat.Seconds:
                str = String.Format("{0:F2} sec", elapsedTime.TotalSeconds);
                break;

            case ElapsedTimeFormat.Milliseconds:
                str = String.Format("{0:F0} msec", elapsedTime.TotalMilliseconds);
                break;
        }
        elapsedText.Text = str;
    }
    …
}
```

Самым важным полем является экземпляр *Stopwatch*. Обычно с помощью этого класса разработчики определяют, сколько времени приложение проводит в определенном методе. Он редко используется как настоящий секундомер.

Рассмотрим, как используется поле *suspensionAdjustment* (Настройка задержки) в связи с захоронением.

.NET-объект *Stopwatch* предоставляет истекшее время в форме объекта *TimeSpan*. У меня не получилось «уговорить» объект *TimeSpan* отображать истекшее время именно в том формате, в котором мне хотелось бы, поэтому пришлось выполнять форматирование самостоятельно. Поле *decimalSeparator* (Десятичный разделитель) является красивым жестом для локализации.

Метод *DisplayTime* (Вывести время) задает значение свойства *Text* объекта *TextBlock*. Он выполняет доступ к свойству *Elapsed* (Истекший) объекта *Stopwatch* и добавляет значение *suspensionAdjustment*. Полученное значение форматируется одним из трех способов в зависимости от значения свойства *ElapsedTimeFormat* (Формат истекшего времени) класса *App*.

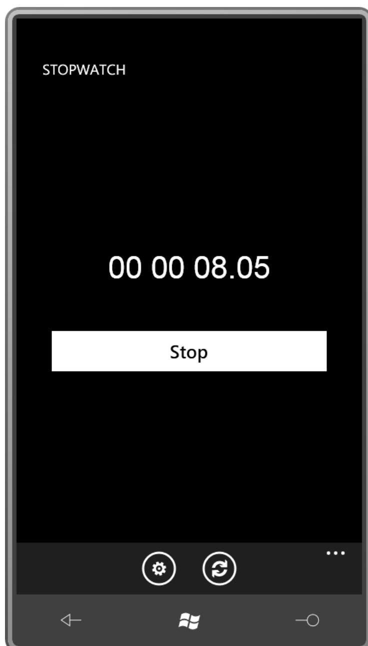
При нажатии объект *ToggleButton* формирует события *Checked* и *Unchecked*. Они оба обрабатываются методом *OnToggleButtonChecked* (При выборе переключателя). Этот метод использует значение свойства *IsChecked* объекта *ToggleButton* для запуска или остановки объекта *Stopwatch* и также для изменения текста, отображаемого кнопкой. Чтобы текст обновлялся оперативно, событие *CompositionTarget.Rendering* просто вызывает *DisplayTime*:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
void OnToggleButtonChecked(object sender, RoutedEventArgs e)
{
    if ((bool)startStopToggle.IsChecked)
    {
        stopwatch.Start();
        startStopToggle.Content = "Stop";
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }
    else
    {
        stopwatch.Stop();
        startStopToggle.Content = "Start";
        CompositionTarget.Rendering -= OnCompositionTargetRendering;
    }
}
```

```
void OnCompositionTargetRendering(object sender, EventArgs args)
{
    DisplayTime();
}
```

Посмотрим, что получилось:



Как видите, приложение также включает *ApplicationBar*. Его две кнопки обозначены как «format» (форматировать) и «reset» (сброс). Рассмотрим описание *ApplicationBar* в XAML-файле:

Проект Silverlight: StopWatch Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.feature.settings.rest.png"
                                        Text="format"
                                        Click="OnAppbarFormatClick" />

        <shell:ApplicationBarIconButton IconUri="/Images/appbar.refresh.rest.png"
                                        Text="reset"
                                        Click="OnAppbarResetClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Более простым из двух методов *Click* является тот, который обеспечивает сброс секундомера. Сброс .NET-объекта *Stopwatch* также останавливает отсчет времени, поэтому выполняется явная отмена выбора *ToggleButton* и *suspensionAdjustment* устанавливается равным нулю:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarResetClick(object sender, EventArgs args)
{
    stopwatch.Reset();
    startStopToggle.IsChecked = false;
```

```
    suspensionAdjustment = new TimeSpan();
    DisplayTime();
}
```

Выбрать формат отображения истекшего времени несколько сложнее. Я решил реализовать эту функциональность не с помощью пунктов меню *ApplicationBar*, а с помощью небольшого диалогового окна. Этот диалог описан прямо в XAML-файле в той же ячейке *Grid*, что и главное окно:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

    <!-- Экран секундомера -->
    …
    <!-- Прямоугольник для имитации неактивного состояния -->
    <Rectangle Name="disableRect"
               Fill="#80000000"
               Visibility="Collapsed" />

    <!-- "Диалоговое окно" для выбора формата TimeSpan -->
    <Border Name="formatDialog"
            Background="{StaticResource PhoneChromeBrush}"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="3"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Visibility="Collapsed">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <StackPanel Name="radioButtonPanel"
                        Grid.Row="0"
                        Grid.Column="0"
                        Grid.ColumnSpan="2"
                        HorizontalAlignment="Center">

                <RadioButton Content="Hour/Minute/Seconds"
                             Tag="HourMinuteSecond" />

                <RadioButton Content="Seconds"
                             Tag="Seconds" />

                <RadioButton Content="Milliseconds"
                             Tag="Milliseconds" />
            </StackPanel>

            <Button Grid.Row="1" Grid.Column="0"
                    Content="ok"
                    Click="OnOkButtonClick" />

            <Button Grid.Row="1" Grid.Column="1"
                    Content="cancel"
                    Click="OnCancelButtonClick" />
        </Grid>
    </Border>
</Grid>
```

Обратите внимание, что свойству *Visibility* обоих объектов, *Rectangle* и *Border*, задано значение *Collapsed*, поэтому в обычном состоянии на экране их не видно. *Rectangle* покрывает всю область содержимого и используется исключительно для обеспечения визуального представления неактивного состояния. *Border* структурирован во многом аналогично традиционному диалоговому окну и включает три элемента управления *RadioButton* и два *Button* с надписями «ok» и «cancel».

Обратите внимание, что в элементах управления *RadioButton* не заданы обработчики для событий *Checked*, но у них есть текстовые строки для задания свойств *Tag* (Тег). Свойство *Tag* определено классом *FrameworkElement* и позволяет прикреплять данные произвольного характера к элементам и элементам управления. И это не просто совпадение, что текстовые строки, заданные мною как значения свойств *Tag*, точно повторяют члены перечисления *ElapsedTimeFormat*.

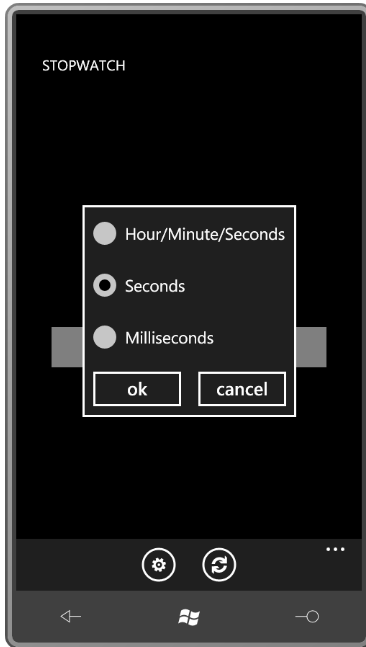
Когда пользователь нажимает кнопку *ApplicationBar* с надписью «format», вызывается метод *OnAppbarFormatClick* (По щелчку кнопки форматировать панели приложения) и делает элементы *disableRect* (Прямоугольник неактивного состояния) и *formatDialog* (Диалоговое окно форматирования) видимыми:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarFormatClick(object sender, EventArgs args)
{
    disableRect.Visibility = Visibility.Visible;
    formatDialog.Visibility = Visibility.Visible;

    // Инициализируем переключатели
    ElapsedTimeFormat currentFormat = (Application.Current as
App).ElapsedTimeFormat;

    foreach (UIElement child in radioButtonPanel.Children)
    {
        RadioButton radio = child as RadioButton;
        ElapsedTimeFormat radioFormat =
            (ElapsedTimeFormat)Enum.Parse(typeof(ElapsedTimeFormat),
                                          radio.Tag as string, true);
        radio.IsChecked = currentFormat == radioFormat;
    }
}
```

Согласно логике задание свойства *IsChecked*, определенного *RadioButton*, выполняется если значение его свойства *Tag* (после преобразования в член перечисления *ElapsedTimeFormat*) равно значению *ElapsedTimeFormat*, хранящемуся как параметр приложения. (Более простая логика была бы возможна, если бы значениями свойства *Tag* были просто 0,1 и 2 для целочисленных значений членов перечисления.)

Получаем такое диалоговое окно:



С элементами управления *RadioButton* не ассоциирован ни один обработчик событий. Следующим событием, полученным приложением после вывода диалогового окна на экран, будет сигнал о нажатии пользователем кнопки «ok» или «cancel»:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
void OnOkButtonClick(object sender, RoutedEventArgs args)
{
    foreach (UIElement child in radioButtonPanel.Children)
    {
        RadioButton radio = child as RadioButton;
        if ((bool)radio.IsChecked)
            (Application.Current as App).ElapsedTimeFormat =
                (ElapsedTimeFormat)Enum.Parse(typeof(ElapsedTimeFormat),
                                              radio.Tag as string, true);
    }
    OnCancelButtonClick(sender, args);
}

void OnCancelButtonClick(object sender, RoutedEventArgs args)
{
    disableRect.Visibility = Visibility.Collapsed;
    formatDialog.Visibility = Visibility.Collapsed;
    DisplayTime();
}
```

Процедура обработки нажатия кнопки «ok» проверяет, какой объект *RadioButton* выбран и задает параметру приложения соответствующее значение. Также вызывается обработчик нажатия кнопки «cancel», который «удаляет» диалоговое окно, возвращая свойству *Visibility* объектов *disableRect* и *formatDialog* значение *Collapsed*.

Такое приложение представляет некоторые сложности с точки зрения захоронения, но я решил не заострять внимания на проблемах, связанных с диалоговым окном. Если пользователь решит покинуть приложение в момент, когда диалоговое окно выведено на экран, нет ничего страшного в том, что по его возвращении оно не будет отображаться.

В идеале секундомер должен продолжать отсчет времени, даже если пользователь переходит к другому приложению. Конечно, это невозможно, потому что приложение завершает выполнение.

Однако приложение *может* сохранять текущее значение истекшего времени и время, когда оно было захоронено. При возобновлении выполнения оно может использовать эти данные для настройки показателей секундомера. Эту функциональность реализуют методы *OnNavigatedFrom* и *OnNavigatedTo*:

Проект Silverlight: StopWatch Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    PhoneApplicationService service = PhoneApplicationService.Current;
    service.State["stopWatchRunning"] = (bool)startStopToggle.IsChecked;
    service.State["suspensionAdjustment"] = suspensionAdjustment +
stopwatch.Elapsed;
    service.State["tombstoneBeginTime"] = DateTime.Now;

    base.OnNavigatedFrom(args);
}

protected override void OnNavigatedTo(NavigationEventArgs args)
{
    PhoneApplicationService service = PhoneApplicationService.Current;

    if (service.State.ContainsKey("stopWatchRunning"))
    {
        suspensionAdjustment = (TimeSpan)service.State["suspensionAdjustment"];

        if ((bool)service.State["stopWatchRunning"])
        {
            suspensionAdjustment += DateTime.Now -

(DateTime)service.State["tombstoneBeginTime"];
            startStopToggle.IsChecked = true;
        }
        else
        {
            DisplayTime();
        }
    }
    base.OnNavigatedTo(args);
}
```

При любом повторном запуске приложения исходное значение истекшего времени всегда равно нулю. Настроить .NET-объект *Stopwatch* напрямую нельзя. Для этого используется поле *suspensionAdjustment*, которое представляет время, прошедшее с момента захоронения приложения, плюс значение истекшего времени, зафиксированное *Stopwatch* на момент захоронения. Пользователь может покидать приложение несколько раз при включенном секундомере, поэтому это поле может содержать данные за несколько периодов захоронения.

Самый простой вариант *OnNavigatedTo* – возвращение в приложение при выключенном секундомере. В этом случае требуется лишь задать *suspensionAdjustment* сохраненное значение. Но если секундомер «был включен» все это время, значение *suspensionAdjustment* должно быть увеличено на отрезок времени, прошедший с момента захоронения, на основании значения, возвращаемого *DateTime.Now*.

При реальном применении приложение StopWatch будет создавать иллюзию выполнения и отслеживать истекшее время, даже в периоды захоронения, и это та иллюзия, которая делает данное приложение намного более полезным, чем оно было бы в противном случае.

Кнопки и стили

Свойство *Style* описано классом *FrameworkElement*, поэтому, несомненно, оно наследуется классами *Control*, *ButtonBase* и *Button*. Рассмотрим приложение, описывающее *Style* для *Button* в разделе *Resources* страницы:

Project: ButtonStyles   Файл: MainPage.xaml (фрагмент)

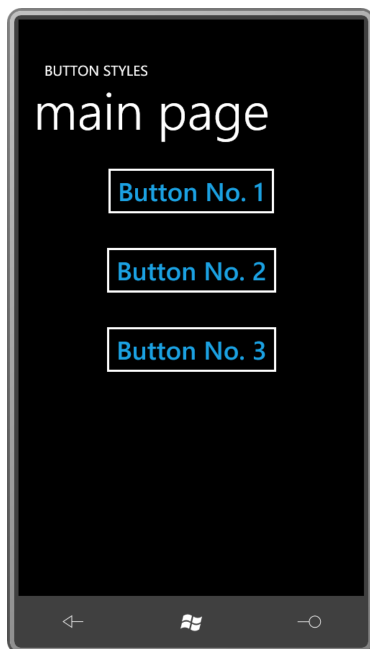
```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="btnStyle" TargetType="Button">
        <Setter Property="Foreground" Value="SkyBlue" />
        <Setter Property="FontSize" Value="36" />
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="Margin" Value="12" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Как обычно, у свойства *Style* есть атрибут *x:Key* и свойство *TargetType*. Три элемента управления *Button* размещены в *StackPanel*. Каждый имеет ссылку на ресурс *Style*:

Проект: ButtonStyles   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name=" ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <Button Content="Button No. 1"
                Style="{StaticResource btnStyle}" />

        <Button Content="Button No. 2"
                Style="{StaticResource btnStyle}" />

        <Button Content="Button No. 3"
                Style="{StaticResource btnStyle}" />
    </StackPanel>
</Grid>
```

Так это выглядит на экране:



Теперь заменим один из этих трех объектов *Button* объектом *ToggleButton*:

```
<ToggleButton Content="Button No. 2"
              Style="{StaticResource btnStyle}" />
```

Это приводит к ошибке времени выполнения, потому что выполняется попытка настройки *ToggleButton* из ресурса *Style*, в качестве *TargetType* которого задан объект *Button*.

Но если взглянуть на иерархию классов, можно увидеть, что оба класса, *Button* и *ToggleButton*, наследуются от *ButtonBase*. Зададим его в качестве *TargetType* для *Style*:

```
<Style x:Key="btnStyle" TargetType="ButtonBase">
    <Setter Property="Foreground" Value="SkyBlue" />
    <Setter Property="FontSize" Value="36" />
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="Margin" Value="12" />
</Style>
```

Теперь все нормально. Можно даже менять *TargetType* на *Control*, но это максимальная вольность, которую можно себе позволить в данном конкретном примере. Если изменить *TargetType* на *FrameworkElement*, опять получим ошибку времени выполнения, потому что у *FrameworkElement* нет свойств *Foreground* или *FontSize*.

Следует взять за правило задавать в качестве *TargetType* самый общий класс, у которого имеются все свойства, определенные в *Style*. Можно наследовать свойства на основании производных классов. Например, можно начать со *Style*, *TargetType* которого является *ButtonBase*, и затем создать два производных стиля с *TargetType* типа *Button* и *TargetType* типа *ToggleButton*.

TextBox и ввод с клавиатуры

В Silverlight для Windows Phone предлагается два типа элементов управления, обеспечивающих текстовый ввод. Это *TextBox*, который позволяет вводить и редактировать неформатированный текст в одну или много строк, и *PasswordBox* (Поле для введения пароля), который несколько мгновений показывает введенный символ и затем заменяет его другим символом, по умолчанию звездочкой.

Приложение может принимать ввод с аппаратной клавиатуры телефона (если таковая существует) или вызывать Software Input Panel (SIP), виртуальную экранную клавиатуру, только этими двумя способами.

Сразу перейдем к практике. Назначение приложения OneTimeText (Разовое текстовое сообщение) – отправка коротких текстовых сообщений SMS (Short Message Service)[1] на определенный телефонный номер. Приложение требует ввести этот телефонный номер, но нигде его не сохраняет. Поэтому я использовал в его имени слово «разовый» (one time).

Рассмотрим область содержимого:

Проект Silverlight: OneTimeText Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid Margin="24">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Grid.Row="0"
                   Text="phone number"
                   Style="{StaticResource PhoneTextSmallStyle}" />

        <TextBox Name="toTextBox"
                 Grid.Row="1"
                 InputScope="TelephoneNumber"
                 TextChanged="OnTextBoxTextChanged" />

        <TextBlock Grid.Row="2"
                   Text="text message"
                   HorizontalAlignment="Left"
                   Style="{StaticResource PhoneTextSmallStyle}" />

        <TextBlock Name="charCountText"
                   Grid.Row="2"
                   HorizontalAlignment="Right"
                   Style="{StaticResource PhoneTextSmallStyle}" />

        <TextBox Name="bodyTextBox"
                 Grid.Row="3"
                 MaxLength="160"
                 TextWrapping="Wrap"
                 VerticalScrollBarVisibility="Auto"
                 TextChanged="OnTextBoxTextChanged" />

        <Button Name="sendButton"
                Grid.Row="4"
                Content="send"
                IsEnabled="False"
                HorizontalAlignment="Center"
                Click="OnSendButtonClick" />
    </Grid>
</Grid>
```

Первый *TextBox* предназначен для ввода телефонного номера, поэтому ему достаточно одной строки. Второй *TextBox* – для ввода тела сообщения. Он занимает все пространство *Grid*, не используемое другими элементами. Его свойству *TextWrapping* задано значение

[1] Служба коротких сообщений (*прим. переводчика*).

Wrap. Это обеспечивает возможность ввода многострокового текста, которая обычно используется в сочетании с вертикальной прокруткой.

Кнопка с надписью «send» (отправить) изначально неактивна, потому что еще ничего не введено ни в один *TextBox*. По этой причине событие *TextChanged* (Текст изменен) задано для обоих *TextBox*.

Свойству *InputScope* (Тип вводимых данных) первого *TextBox* задано значение *TelephoneNumber* (Телефонный номер). Когда пользователь касается этого *TextBox*, появляется цифровая клавиатура:

Для второго *TextBox* свойство *InputScope* не задано, поэтому для него выводится стандартная клавиатура общего назначения:

Для второго *TextBox* задано свойство *MaxLength*, поэтому в него нельзя ввести более 160 символов – максимальная длина SMS.

Допустимые значения *InputScope* являются членами перечисления *InputScopeNameValue* (Имена типов вводимых данных), описанного в пространстве имен *System.Windows.Input*. Чтобы иметь возможность пользоваться подсказками Intellisense в Visual Studio, необходимо вынести *InputScope* как свойство-элемент и задать его следующим образом:

```
<TextBox Name="toTextBox"
         Grid.Row="1"
         TextChanged="OnTextBoxTextChanged">
    <TextBox.InputScope>
        <InputScope>
            <InputScopeName NameValue="TelephoneNumber" />
        </InputScope>
    </TextBox.InputScope>
</TextBox>
```

Теперь сразу же после ввода знака равно за *NameValue*, система предложит список возможных вариантов значений.

В XAML нет самого важного свойства *TextBox*, каковым является свойство *Text* типа *string*. В любой момент времени программно можно выполнить доступ к этому свойству и получить или задать ему значение для инициализации содержимого. Также можно вставить что-то в существующее содержимое *TextBox* или удалить что-то. Операция удаления включает следующие этапы: получение текущего значения свойства *Text*, применение обычных методов класса *String* для создания новой строки, содержащей новый текст, и затем задание этой строки как нового значения свойства *Text*.

Привожу довольно большой фрагмент файла выделенного кода *MainPage*:

Проект Silverlight: OneTimeText Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    PhoneApplicationService appService = PhoneApplicationService.Current;
    SmsComposeTask smsTask;

    public MainPage()
    {
        InitializeComponent();

        smsTask = new SmsComposeTask();
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        if (sender == bodyTextBox)
            charCountText.Text = String.Format("{0} chars",
bodyTextBox.Text.Length);

        sendButton.IsEnabled = toTextBox.Text.Length > 0 && bodyTextBox.Text.Length
> 0;
    }

    void OnSendButtonClick(object sender, RoutedEventArgs e)
    {
        smsTask.To = toTextBox.Text;
        smsTask.Body = bodyTextBox.Text;
        smsTask.Show();
    }
    …
}
```

Единственный обработчик события *TextChanged* может различать, какой из элементов *TextBox* сформировал событие, путем сравнения аргумента *sender* с именами, определенными в XAML-файле. Для второго *TextBox* выполняется обновление экрана и отображается количество введенных символов. Кнопка «send» остается неактивной, если один из *TextBox* пуст.

При нажатии *Button* приложение вызывает *SmsComposeTask* (Составление SMS) – стандартное приложение для набора текстов в телефоне. Оно обеспечивает пользователю более удобный интерфейс, позволяющий отправить текст, редактировать его или отправлять другие тексты.

В некоторый момент пользователь может вернуться к приложению OneTimeText. Объект *SmsComposeTask* не возвращает никаких данных приложению, его вызвавшему – это *задача выполнения*, а не *задача выбора* – но было бы неплохо, если бы пользователь мог видеть ранее введенный текст. Для этого приложение перегружает методы *OnNavigationFrom* и *OnNavigationTo*, чтобы обеспечить сохранение и восстановление состояния приложения:

Проект Silverlight: OneTimeText Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    appService.State["toText"] = toTextBox.Text;
    appService.State["bodyText"] = bodyTextBox.Text;

    base.OnNavigatedFrom(args);
}

protected override void OnNavigatedTo(NavigationEventArgs args)
{
    object text;

    if (appService.State.TryGetValue("toText", out text))
        toTextBox.Text = text as string;

    if (appService.State.TryGetValue("bodyText", out text))
        bodyTextBox.Text = text as string;

    base.OnNavigatedTo(args);
}
```

Последним примером, который мы рассмотрим в данной главе, является приложение QuickNotes. Его задача – обеспечивать возможность быстрого ввода коротких заметок и гарантировать их сохранение без каких-либо явных операций сохранения или загрузки. По сути, это версия приложения Notepad (Блокнот) для Windows Phone 7, но QuickNotes может работать только с одним файлом.

Приложение также позволяет менять размер шрифта, поэтому в классе параметров приложения *QuickNotesSettings* (Параметры QuickNotes) имеется два открытых свойства, *Text* и *FontSize*, а также методы для сохранения и загрузки этих свойств в/из изолированного хранилища:

Проект Silverlight: QuickNotes Файл: QuickNotesSettings.cs

```
public class QuickNotesSettings
{
    public QuickNotesSettings()
    {
        this.Text = "";
        this.FontSize =
(double)Application.Current.Resources["PhoneFontSizeMediumLarge"];
```

```
    }

    public string Text { set; get; }
    public double FontSize { set; get; }

    public static QuickNotesSettings Load()
    {
        IsolatedStorageSettings isoSettings =
IsolatedStorageSettings.ApplicationSettings;
        QuickNotesSettings settings;

        if (!isoSettings.TryGetValue<QuickNotesSettings>("settings", out settings))
            settings = new QuickNotesSettings();

        return settings;
    }

    public void Save()
    {
        IsolatedStorageSettings isoSettings =
IsolatedStorageSettings.ApplicationSettings;
        isoSettings["settings"] = this;
    }
}
```

Как и в приложении Jot, за сохранение, загрузку и предоставление этих параметров отвечает класс *App*:

Проект Silverlight: QuickNotes Файл: App.xaml.cs

```
public partial class App : Application
{
    // Параметры приложения
    public QuickNotesSettings AppSettings { set; get; }

    …

    private void Application_Launching(object sender, LaunchingEventArgs e)
    {
        AppSettings = QuickNotesSettings.Load();
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        AppSettings = QuickNotesSettings.Load();
    }

    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        AppSettings.Save();
    }

    private void Application_Closing(object sender, ClosingEventArgs e)
    {
        AppSettings.Save();
    }
    …
}
```

В XAML-файле создается многострочный *TextBox*, занимающий всю область содержимого. Кроме определения свойства *TextWrapping*, обеспечивающего возможность многострочного редактирования, в разметке задается значение *true* свойству *AcceptsReturn* (Допускает возврат строки). Благодаря этому по нажатию клавиши Enter будет выполняться переход на новую строку, что вполне соответствует требованиям, предъявляемым к данному

приложению. (В контексте диалогового окна обычно ожидается, что по нажатию клавиши Enter инициируется кнопка OK, даже если это происходит в процессе введения пользователем текста в *TextBox*.)

Проект Silverlight: QuickNotes   Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBox Name="txtbox"
             TextWrapping="Wrap"
             AcceptsReturn="True"
             VerticalScrollBarVisibility="Auto"
             TextChanged="OnTextBoxTextChanged" />
</Grid>
```

XAML-файл также включает *ApplicationBar* с двумя кнопками, которые я создал самостоятельно для увеличения и уменьшения размера шрифта:

Проект Silverlight: QuickNotes   Файл: MainPage.xaml

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton IconUri="/Images/littleletter.icon.png"
                                        Text="smaller font"
                                        Click="OnAppBarSmallerFontClick" />

        <shell:ApplicationBarIconButton IconUri="/Images/bigletter.icon.png"
                                        Text="larger font"
                                        Click="OnAppBarLargerFontClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

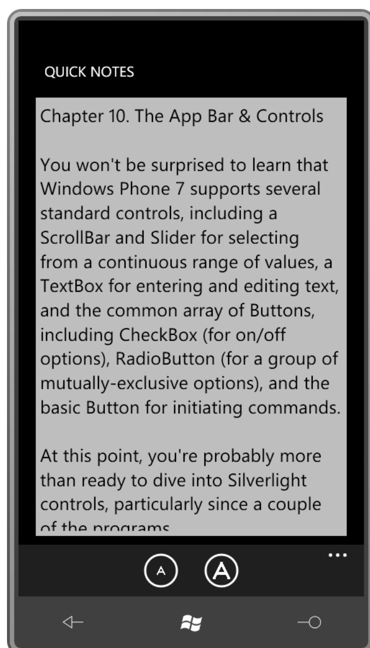
Благодаря всем этим приготовлениям, сам файл кода для *MainPage* довольно мал и прост:

Проект Silverlight: QuickNotes   Файл: MainPage.xaml.cs

```
public partial class MainPage : PhoneApplicationPage
{
    QuickNotesSettings appSettings = (Application.Current as App).AppSettings;

    public MainPage()
    {
        InitializeComponent();

        txtbox.Text = appSettings.Text;
        txtbox.FontSize = appSettings.FontSize;
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        appSettings.Text = txtbox.Text;
    }

    void OnAppBarSmallerFontClick(object sender, EventArgs args)
    {
        txtbox.FontSize = Math.Max(12, txtbox.FontSize - 1);
        appSettings.FontSize = txtbox.FontSize;
    }

    void OnAppBarLargerFontClick(object sender, EventArgs args)
    {
        txtbox.FontSize = Math.Min(48, txtbox.FontSize + 2);
```

```
        appSettings.FontSize = txtbox.FontSize;
    }
}
```

При любом изменении текста в *TextBox* метод *OnTextBoxChanged* (При изменении текстового поля) сохраняет новую версию в параметрах приложения. Два метода для увеличения и уменьшения размера шрифта тоже сохраняют новое значение и используют его для задания свойства *FontSize* элемента *TextBox*. На рисунке показано, как все это выглядит в действии:



Вот что не делает данное приложение, так это не сохраняет точки вставки текста (визуально она обозначается знаком вставки *TextBox*), поэтому при любом запуске приложения пользователю придется самому указывать посредством касания, где он хочет продолжить ввод текста. Даже если пользователь вышел из приложения, находясь в конце файла, QuickNotes при запуске всегда переходит в его начало.

Я немного поупражнялся с решением этой проблемы. Точка вставки доступна через свойство *SelectionStart* (Начало выбранной подстроки). Как следует из его имени, это свойство используется в связи с выбором текста. Также существует свойство *SelectionLength* (Длина выбранной подстроки), значение которого равно 0, если никакой текст не выбран. (Выполнять доступ к или задавать выбранный текст можно также с помощью свойства *SelectedText* (Выбранный текст).)

У *TextBox* также имеется событие *SelectionChanged* (Выбранная подстрока изменилась), поэтому QuickNotes, несомненно, может сохранять новое значение *SelectionStart* в параметрах приложения при каждом его изменении. Тогда вся задача сводится к заданию свойства *SelectionStart*, а также *Text* и *FontSize* в конструкторе *MainPage*.

Однако такой подход не вполне работоспособен. При запуске или возвращении к QuickNotes фокус ввода находится не на *TextBox*. Пользователь должен коснуться экрана, чтобы *TextBox* получил фокус, и начать вводить что-то. Но при касании экрана пользователь задает новую точку ввода!

Решение этой небольшой проблемы в программном задании фокуса ввода для *TextBox*. Чтобы сделать это в конструкторе *MainPage*, необходимо установить обработчик события *Loaded*:

```
txtbox.Focus();
```

Однако это приводит к довольно драматичным последствиям при входе в приложение: как только оно запускается, на экране появляется виртуальная клавиатура! Я пытался бороться с этой проблемой, но в конце концов решил, что от этого больше вреда, чем пользы.

Кто знает, возможно, я верну эту функциональность несколько позже. Может, поэтому оно и называется *программным обеспечением*, что ничего невозможно запрограммировать или спрогнозировать.

Глава 11
Свойства-зависимости

Данная глава посвящена созданию пользовательских классов элементов управления в Silverlight. В ней рассказывается, как сделать их доступными через библиотеки динамической компоновки, и как работать с ними в коде и разметке.

Наследование одного класса от другого является настолько базовым аспектом объектно-ориентированного программирования, что в том, чтобы посвящать ему целую главу, кажется, нет необходимости. С одной стороны, в наследовании пользовательских классов от существующих классов Silverlight нет ничего особенного. Полученный класс можно использовать в XAML, просто объявив пространство имен XML, чтобы связать префикс XML с пространством имен .NET. Именно это было продемонстрировано мною в главе 9 в двух проектах, представлявших примеры создания пользовательских панелей.

С другой стороны, если создается класс пользовательского элемента управления, и этот класс определяет новые свойства, и если требуется задавать значения этих свойств через стили или привязки данных, или требуется назначить их целью анимации, с этими свойствами требуется сделать нечто очень особенное.

Их необходимо сделать *свойствами-зависимостями*.

Описание проблемы

Для иллюстрации разницы, обеспечиваемой свойствами-зависимостями, рассмотрим сначала класс пользовательского элемента управления, написанный начинающим разработчиком.

Предположим, требуется создать приложение с множеством кнопок, закрашенных с применением различных градиентных кистей. Мы решаем, что удобней будет задать два цвета как свойства кнопок. Назовем свойства *Color1* и *Color2*. Итак, открываем проект NaiveGradientButtonDemo (Демонстрация простой кнопки с градиентом) и добавляем новый класс под именем *NaiveGradientButton* (Простая кнопка с градиентом). Вот этот класс:

Проект Silverlight: NaiveGradientButtonDemo Файл: NaiveGradientButton.cs (фрагмент)

```
public class NaiveGradientButton : Button
{
    GradientStop gradientStop1, gradientStop2;

    public NaiveGradientButton()
    {
        LinearGradientBrush brush = new LinearGradientBrush();
        brush.StartPoint = new Point(0, 0);
        brush.EndPoint = new Point(1, 0);

        gradientStop1 = new GradientStop();
        gradientStop1.Offset = 0;
        brush.GradientStops.Add(gradientStop1);

        gradientStop2 = new GradientStop();
        gradientStop2.Offset = 1;
        brush.GradientStops.Add(gradientStop2);

        Foreground = brush;
```

```
    }

    public Color Color1
    {
        set { gradientStop1.Color = value; }
        get { return (Color)gradientStop1.Color; }
    }

    public Color Color2
    {
        set { gradientStop2.Color = value; }
        get { return (Color)gradientStop2.Color; }
    }
}
```

Как и ожидалось, *NaiveGradientButton* наследуется от *Button* и включает два новых свойства типа *Color*: *Color1* и *Color2*. Конструктор создает *LinearGradientBrush*, задает значения свойств *StartPoint* и *EndPoint*, создает два объекта *GradientStop*, хранящихся как поля, добавляет их в *LinearGradientBrush* и затем определяет эту кисть как значение свойства *Foreground* кнопки.

Этот класс не препятствует повторному заданию свойства *Foreground* объекта *GradientBrush* в коде или XAML после создания объекта, но поскольку заданное здесь значение *Foreground* является локальным параметром, свойство *Foreground* не будет наследоваться, и на него не будет распространяться действие *Style*, целевым свойством которого является *Foreground*.

Как видите, методы доступа для получения и задания значений свойств *Color1* и *Color2* просто выполняют доступ к свойству *Color* в соответствующем *GradientStop*.

Этот класс используется в файле MainPage.xaml проекта NaiveGradientButtonDemo. Корневой элемент включает описание пространства имен XML, которое связывает префикс пространства имен «local» с пространством имен CLR *NaiveGradientButton*:

```
xmlns:local="clr-namespace:NaiveGradientButtonDemo"
```

Коллекция *Resources* в MainPage.xaml определяет *Style* для *NaiveGradientButton*:

Проект Silverlight: NaiveGradientButtonDemo Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="gradientButtonStyle"
           TargetType="local:NaiveGradientButton">
        <Setter Property="HorizontalAlignment" Value="Center" />

        <!--
        <Setter Property="Color1" Value="Cyan" />
        <Setter Property="Color2" Value="Pink" />
        -->

    </Style>
</phone:PhoneApplicationPage.Resources>
```

Обратите внимание, что ссылка на пользовательский класс в *TargetType* начинается с пространства имен XML, за которым уже следует имя класса.

Также нетрудно заметить, что я закомментировал теги *Setter*, где задаются значения свойствам *Color1* и *Color2*. (Наверное, я не так наивен, как иногда притворяюсь.)

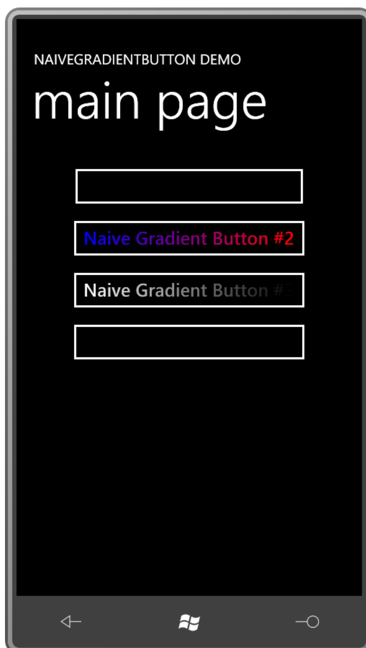
В области содержимого, описываемой XAML-файлом, располагается четыре экземпляра *NaiveGradientButton*, значения свойств *Color1* и *Color2* которых заданы самыми различными способами:

Проект Silverlight: NaiveGradientButtonDemo   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <local:NaiveGradientButton Content="Naive Gradient Button #1"
                                   HorizontalAlignment="Center" />

        <local:NaiveGradientButton Content="Naive Gradient Button #2"
                                   Color1="Blue" Color2="Red"
                                   HorizontalAlignment="Center" />

        <local:NaiveGradientButton Content="Naive Gradient Button #3"
                                   Color1="{StaticResource PhoneForegroundColor}"
                                   Color2="{StaticResource PhoneBackgroundColor}"
                                   HorizontalAlignment="Center" />

        <local:NaiveGradientButton Content="Naive Gradient Button #4"
                                   Style="{StaticResource gradientButtonStyle}" />
    </StackPanel>
</Grid>
```

Первая кнопка использует для *Color1* и *Color2* значения по умолчанию; для второй цвета задаются явно; третья ссылается на цвета темы; и четвертая использует *Style*, описанный в коллекции *Resources*.

При запуске приложения обнаруживается, что со второй и третьей кнопками все в порядке, а вот в первой и четвертой, кажется, отсутствует содержимое:



Color1 и *Color2* не имеют значений по умолчанию. Если они не заданы явно, для цветов в градиенте все свойства *A*, *R*, *G* и *B* будут равны 0, а это соответствует прозрачному черному.

Раскомментируем два тега *Setter* в *Style*. Окно ошибок Visual Studio сообщит о том, что «Object reference not set to an instance of an object[1]» (несомненно, мое самое любимое сообщение об ошибке). Если попытаться запустить приложение в отладчике, будет сформировано исключение *XamlParseException* (Исключение синтаксического анализатора

[1] Объектная ссылка не указывает на экземпляр объекта (*прим. переводчика*).

XAML) и выведено сообщение: «Invalid attribute value Color1 for property Property[1]». Вот это немного лучше и говорит о том, что свойству *Property* тега не может быть присвоено значение *Color1*.

Но лучше бы сообщения об ошибках говорили: «Не будь таким наивным! Используй свойства-зависимости.»

В чем отличие свойств-зависимостей

В Silverlight существует несколько разных способов задания свойств. Опытным путем мы выяснили, что если одно и то же свойство задается через наследование свойств либо темой, либо стилем, либо как локальный параметр, действует строгая приоритезация. Вспомним небольшую схему, созданную нами в главе 7:

> **Локальные параметры** имеют приоритет над
>
> **Параметрами стиля**, которые являются более приоритетными, чем
>
> **Стиль темы**, который является более приоритетным, чем
>
> **Унаследованные свойства**, которые имеют приоритет над
>
> **Значениями по умолчанию**

В следующих главах будет показано, что свойства могут задаваться из анимаций и шаблонов, и это тоже укладывается в диаграмму приоритетности.

Такая строгая приоритетность позволяет избежать коллизий стилей и анимаций, и всего остального. Если бы не приоритеты, воцарился бы полный хаос, а это противоречит нашему желанию иметь абсолютно детерминированный код.

Silverlight обеспечивает инфраструктуру для управления всеми возможными способами задания свойств и устанавливает тем самым некоторый порядок. Свойства-зависимости являются основной составляющей этой инфраструктуры. Их называют свойствами-зависимостями потому, что они *зависят* от ряда внешних факторов и выступают посредниками между ними и приложением.

Свойства-зависимости строятся на базе существующих .NET-свойств. Для этого надо немного потрудиться и пописать код, но вы будете создавать свойства-зависимости автоматически уже до того, как осознаете это.

Кроме всего прочего, свойства-зависимости обеспечивают приоритетность задания свойств. Все это происходит автоматически в фоновом режиме, и в это не надо вмешиваться. Свойства-зависимости также предлагают очень структурированный способ задания свойствам значений по умолчанию и предоставления методов обратного вызова, которые вызываются при изменении значения свойства.

Практически все свойства классов Silverlight, с которыми мы до сих пор сталкивались, были на самом деле свойствами-зависимостями. Проще назвать свойства, которые ими не были бы! Первыми на ум приходят свойство *Children* класса *Panel* и свойство *Text* класса *Run*[2].

Все классы, реализующие свойства-зависимости, должны наследоваться от *DependencyObject*. *DependencyObject* – один из базовых классов в иерархии классов Silverlight. Многие классы

---

[1] Недействительное значение атрибута Color1 для свойства Property (*прим. переводчика*).
[2] И даже это временно, т.к. в обычном Silverlight 4.0 свойство Text у Run уже стало свойством-зависимостью (*прим. научного редактора*)

Silverlight наследуются от него, включая и самый значимый: *UIElement*. Это означает, что класс *Button* наследуется от *DependencyObject*, из чего, несомненно, следует, что любой класс, унаследованный от *Button*, может реализовывать свойства-зависимости.

Класс *BetterGradientButton* (Более сложная кнопка с градиентом) со свойствами-зависимостями начинается, как обычно:

```
public class BetterGradientButton : Button
{

}
```

Как и *NaiveGradientButton*, *BetterGradientButton* описывает два свойства: *Color1* и *Color2*. Свойства-зависимости начинаются с открытого поля типа *DependencyProperty*, имя которого совпадает с именем свойства, но с добавлением слова *Property*. Итак, первым шагом в определении свойства *Color1* в классе *BetterGradientButton* является описание открытого поля типа *DependencyProperty* под именем *Color1Property*.

```
public class BetterGradientButton : Button
{
    public static readonly DependencyProperty Color1Property =
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));
}
```

Это не просто поле, но *открытое статическое* поле, и обычно оно также является полем *только для чтения* (это означает, что после того, как оно определено, его нельзя изменить). *DependencyProperty* (Свойство-зависимость) создается для определенного класса и после этого не меняется и используется совместно всеми экземплярами этого класса.

Как правило, объект типа *DependencyProperty* создается путем вызова статического метода *DependencyProperty.Register* (Зарегистрировать). (Единственным исключением являются присоединенные свойства.) Первый аргумент – это текстовая строка имени свойства; второй аргумент типа свойства, в данном случае это *Color*; третий аргумент – это класс, описывающий свойство, в данном примере это класс *BetterGradientButton*.

Последний аргумент – объект типа *PropertyMetadata* (Метаданные свойства). В конструкторе *PropertyMetadata* могут быть предоставлены только два типа сведений. Первое – значение свойства по умолчанию, т.е. значение, используемое для свойства, если оно не задано никаким другим способом. Если для ссылочного типа не обеспечить значение по умолчанию, оно принимается равным *null*. Для типа значения – это значение по умолчанию этого типа.

Я решил, что значением по умолчанию свойства *Color1* должно быть *Colors.Black*.

Вторая часть конструктора *PropertyMetadata* – имя обработчика, вызываемого при изменении значения свойства. Этот обработчик вызывается, только если значение свойства действительно меняется. Например, если значением свойства по умолчанию является *Colors.Black*, и свойству присваивается значение *Colors.Black*, обработчик события изменения значения свойства вызываться *не* будет.

Кажется странным для сущности, названной *свойством-зависимостью* типа *DependencyProperty* с именем *Color1Property*, быть определенной как поле, но так оно и есть.

Эти два класса, *DependencyObject* и *DependencyProperty*, легко спутать. Все классы, имеющие свойства-зависимости, должны наследоваться от *DependencyObject*, так же как обычные

классы наследуются от *Object*. После этого класс создает объекты типа *DependencyProperty*, как и любой другой обычный класс определял бы обычные свойства.

Нет необходимости определять весь *DependencyProperty* в статическом поле. Некоторые разработчики предпочитают инициализировать поле *DependencyProperty* в статическом конструкторе:

```
public class BetterGradientButton : Button
{
    public static readonly DependencyProperty Color1Property;

    static BetterGradientButton()
    {
        Color1Property = DependencyProperty.Register("Color1",
                                typeof(Color),
                                typeof(BetterGradientButton),
                                new PropertyMetadata(Colors.Black, OnColorChanged));
    }
}
```

На самом деле между этими двумя техниками нет разницы.

Кроме статического поля типа *DependencyProperty*, нам понадобиться обычное описание .NET-свойства для свойства *Color1*:

```
public class BetterGradientButton : Button
{
    public static readonly DependencyProperty Color1Property =
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));

    public Color Color1
    {
        set { SetValue(Color1Property, value); }
        get { return (Color)GetValue(Color1Property); }
    }
}
```

Это описание свойства *Color1* является стандартным. Метод доступа *set* (задать) вызывает *SetValue*, который ссылается на свойство-зависимость *Color1Property*, и метод доступа *get* (получить) вызывает метод *GetValue* (Получить значение), также ссылающийся на *Color1Property*.

Откуда взялись эти два метода, *SetValue* и *GetValue*? *SetValue* и *GetValue* – два открытых метода, которые описаны *DependencyObject* и наследуются всеми производными от него классами. Обратите внимание, что второй аргумент в *SetValue* – это значение, которое было задано свойству. Возвращаемое значение *GetValue* типа *object*, поэтому оно должно быть приведено к *Color*.

В связи со свойствами-зависимостями описание свойства *Color1* можно назвать описанием *CLR-свойства* (свойства общеязыковой среды выполнения .NET), чтобы отличить его от объекта *DependencyProperty*, определенного как открытое статическое поле. Иногда говорят, что CLR-свойство *Color1* «продублировано» свойством-зависимостью *Color1Property*. Такая терминология удобна, если вы хотите различить описание свойства от описания открытого статического поля. Но так же часто обе части – и открытое статическое поле, и описание свойства – вместе называют «свойством-зависимостью» или (для самых крутых) «DP[1]»"

---

[1] Аббревиатура английского «dependency property» – свойство-зависимости (*прим. переводчика*).

Очень важно, чтобы CLR-свойство не делало ничего, кроме вызова методов *SetValue* и *GetValue*, здесь не место для проведения каких-либо проверок достоверности или обработки событий изменения значения свойства. Причиной этому является то, что мы никогда не можем точно знать, как задается свойство-зависимость. Можно подумать, что свойство всегда задается так:

```
btn.Color1 = Colors.Red;
```

Но методы *SetValue* и *GetValue*, описанные *DependencyObject*, являются открытыми, поэтому свойство вполне может быть задано и так:

```
btn.SetValue(GradientButton2.Color1Property, Colors.Red);
```

Или оно может быть задано способом, известным только создателям Silverlight.

С другой стороны, не следует ошибочно полагать, что CLR-свойство можно опустить. Иногда, если задать только поле *DependencyProperty* и забыть о CLR-свойстве, некоторые вещи не работают.

Рассмотрим также класс, определяющий *DependencyProperty* и CLR-свойство для *Color2*:

```
public class BetterGradientButton : Button
{
    public static readonly DependencyProperty Color1Property =
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));

    public static readonly DependencyProperty Color2Property =
        DependencyProperty.Register("Color2",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.White, OnColorChanged));

    public Color Color1
    {
        set { SetValue(Color1Property, value); }
        get { return (Color)GetValue(Color1Property); }
    }

    public Color Color2
    {
        set { SetValue(Color2Property, value); }
        get { return (Color)GetValue(Color2Property); }
    }
}
```

В описании *DependencyProperty* для *Color2* в качестве значения по умолчанию я задал *Colors.White*.

Оба поля *DependencyProperty* ссылаются на обработчик события изменения значения свойства *OnColorChanged* (При изменении цвета). Поскольку ссылка на этот метод выполняется в описании статического поля, сам метод тоже должен быть статическим. Вот как это выглядит:

```
static void OnColorChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
    …
}
```

Это статический метод, т.е. один и тот же метод используется для всех экземпляров *BetterGradientButton*, и в этом кроется небольшая проблема. Обычно из статического метода

невозможен доступ к любым нестатическим свойствам или методам. Отсюда можно предположить, что этот метод не может ссылаться на что-либо, связанное с конкретным экземпляром *BetterGradientButton*.

Но обратите внимание, что первый аргумент, передаваемый в этот обработчик события изменения значения свойства, типа *DependencyObject*. Фактически этим аргументом является конкретный экземпляр *BetterGradientButton*, свойство которого изменилось. То есть этот первый аргумент можно свободно приводить к объекту типа *BetterGradientButton*:

```
static void OnColorChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
    BetterGradientButton btn = obj as BetterGradientButton;
    …
}
```

После этого можно воспользоваться переменной *btn* для доступа ко всем свойствам и методам экземпляра в классе.

Второй передаваемый в обработчик аргумент обеспечивает сведения о том, значение какого именно свойства изменилось, а также старое и новое значения этого свойства.

Привожу класс *BetterGradientButton* полностью:

Проект Silverlight: BetterGradientButtonDemo   Файл: BetterGradientButton.cs (фрагмент)

```
public class BetterGradientButton : Button
{
    GradientStop gradientStop1, gradientStop2;

    public static readonly DependencyProperty Color1Property =
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));

    public static readonly DependencyProperty Color2Property =
        DependencyProperty.Register("Color2",
            typeof(Color),
            typeof(BetterGradientButton),
            new PropertyMetadata(Colors.White, OnColorChanged));

    public BetterGradientButton()
    {
        LinearGradientBrush brush = new LinearGradientBrush();
        brush.StartPoint = new Point(0, 0);
        brush.EndPoint = new Point(1, 0);

        gradientStop1 = new GradientStop();
        gradientStop1.Offset = 0;
        gradientStop1.Color = Color1;
        brush.GradientStops.Add(gradientStop1);

        gradientStop2 = new GradientStop();
        gradientStop2.Offset = 1;
        gradientStop2.Color = Color2;
        brush.GradientStops.Add(gradientStop2);

        Foreground = brush;
    }

    public Color Color1
    {
        set { SetValue(Color1Property, value); }
        get { return (Color)GetValue(Color1Property); }
```

```
    }

    public Color Color2
    {
        set { SetValue(Color2Property, value); }
        get { return (Color)GetValue(Color2Property); }
    }

    static void OnColorChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        BetterGradientButton btn = obj as BetterGradientButton;

        if (args.Property == Color1Property)
            btn.gradientStop1.Color = (Color)args.NewValue;

        if (args.Property == Color2Property)
            btn.gradientStop2.Color = (Color)args.NewValue;
    }
}
```

Как и ранее в *NaiveGradientButton*, в данном классе имеется два закрытых поля экземпляра типа *gradientStop1* и *gradientStop2*. Конструктор также довольно похож на предыдущую версию, за исключением одного существенного отличия: свойство *Color* каждого объекта *GradientStop* инициализируется из свойств *Color1* и *Color2*:

```
gradientStop1.Color = Color1;

gradientStop2.Color = Color2;
```

Попытки доступа к свойствам *Color1* и *Color2* приводят к вызову метода *GetValue* с передачей в него аргументов *Color1Property* и *Color2Property*. *GetValue* возвращает значения по умолчанию, определенные в поле *DependencyProperty*: *Colors.Black* и *Colors.White*. Это обеспечивает создание *LinearGradientBrush*, использующего цвета по умолчанию.

Существует множество способов написания обработчика события изменения значения свойства. Обычно этот метод описывается в конце класса. В *DependencyPropertyChangedEventArgs* (Аргументы события изменения значения свойства-зависимости) класса *BetterGradientButton* я использовал два свойства. Свойство *Property* типа *DependencyProperty* указывает на конкретное свойство-зависимость, значение которого изменилось. Это очень удобно в случае, если обработчик события изменения значения свойства совместно используется несколькими свойствами, как в нашем примере.

DependencyPropertyChangedEventArgs также определяет свойства *OldValue* (Старое значение) и *NewValue* (Новое значение). Эти два значения всегда будут различными. Обработчик события изменения значения свойства вызывается, только если значение свойства действительно изменяется.

На момент вызова обработчика события изменения значения свойства значение уже изменилось, так что обработчик может быть реализован через доступ к этим свойствам напрямую. Рассмотрим простой альтернативный вариант:

```
static void OnColorChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
    BetterGradientButton btn = obj as BetterGradientButton;

    btn.gradientStop1.Color = btn.Color1;
    btn.gradientStop2.Color = btn.Color2;
}
```

В данной версии не проверяется, какое именно свойство изменило значение, поэтому для любого отдельно взятого вызова *OnColorChanged* одно из этих выражений является избыточным. Полезно знать, что *GradientStop* наследуется от *DependencyObject*, и свойство *Color* является свойством-зависимостью, поэтому обработчик события изменения значения свойства в *GradientStop* вызывается только в случае, если значение одного из свойств действительно изменилось.

Рассмотрим технику, к которой я прибегаю довольно часто и чрезвычайно доволен результатом.

Вместо того, чтобы дотягиваться до правого уха левой рукой и ссылаться в статическом методе на конкретный экземпляр класса, я использую этот статический метод исключительно для вызова метода экземпляра с таим же именем. Вот как данная техника будет выглядеть в *BetterGradientBrush*:

```
static void OnColorChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
    (obj as BetterGradientButton).OnColorChanged(args);
}

void OnColorChanged(DependencyPropertyChangedEventArgs args)
{
    if (args.Property == Color1Property)
        gradientStop1.Color = (Color)args.NewValue;

    if (args.Property == Color2Property)
        gradientStop2.Color = (Color)args.NewValue;
}
```

Данный метод экземпляра может делать все то же самое, что и статический метод, но без всех этих сложностей, связанных с использованием ссылки на конкретный экземпляр класса.

Теперь посмотрим на этот новый класс в действии. Будет обидно, если все наши усилия пойдут насмарку и никак не улучшат ситуацию. Как в предыдущем приложении, коллекция *Resources* в MainPage.xaml включает элемент *Style*, целью которого является пользовательская кнопка. Но теперь теги *Setter* для *Color1* и *Color2* оптимистично раскомментированы:

Проект Silverlight: BetterGradientButtonDemo Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="gradientButtonStyle"
           TargetType="local:BetterGradientButton">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="Color1" Value="Cyan" />
        <Setter Property="Color2" Value="Pink" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

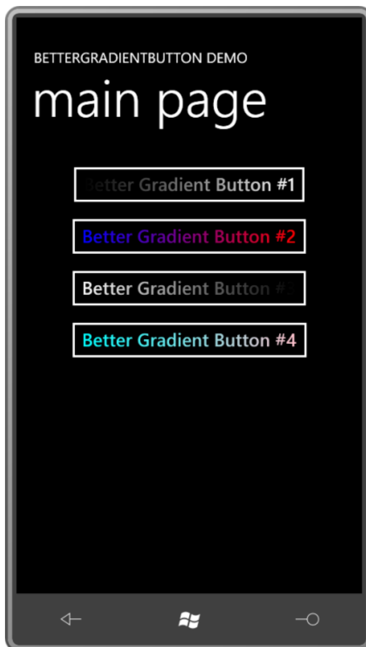
Область содержимого в целом не изменилась:

Проект Silverlight: BetterGradientButtonDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <local:BetterGradientButton Content="Better Gradient Button #1"
                                    HorizontalAlignment="Center" />

        <local:BetterGradientButton Content="Better Gradient Button #2"
                                    Color1="Blue" Color2="Red"
```

```
                                   HorizontalAlignment="Center" />

        <local:BetterGradientButton Content="Better Gradient Button #3"
                                   Color1="{StaticResource PhoneForegroundColor}"
                                   Color2="{StaticResource PhoneBackgroundColor}"
                                   HorizontalAlignment="Center" />

        <local:BetterGradientButton Content="Better Gradient Button #4"
                                   Style="{StaticResource gradientButtonStyle}" />
    </StackPanel>
</Grid>
```

На самом деле радует то, что теперь можно увидеть на экране:



Первая кнопка представляет эффект от применения значений по умолчанию – свойства-зависимости имеют встроенную реализацию этой концепции – и последняя кнопка демонстрирует, что техника с применением *Style* работает.

Обратим внимание на несколько моментов:

Как можно было заметить, мы не имеем прямого доступа к фактическим значениям свойств-зависимостей. Без всякого сомнения, они хранятся где-то в закрытой сущности, доступ к которой возможен только посредством методов *SetValue* и *GetValue*. Вероятно, в *DependencyObject* есть словарь для хранения коллекции свойств-зависимостей и их значений. Это должен быть словарь, потому что *SetValue* может использоваться для хранения определенных типов свойств-зависимостей – в частности, присоединенных свойств – в любом объекте *DependencyObject*.

Обратите внимание на первый аргумент конструктора *PropertyMetadata*. Он определен типа *object*. Предположим, мы создаем *DependencyProperty* для свойства типа *double* и хотим задать ему значение по умолчанию 10:

```
public static readonly DependencyProperty MyDoubleProperty =
    DependencyProperty.Register("MyDouble",
        typeof(double),
        typeof(SomeClass),
        new PropertyMetadata(10, OnMyDoubleChanged));
```

Компилятор C# интерпретирует это значение как *int* и сформирует код для передачи целого значения 10 в конструктор *PropertyMetadata*. Конструктор сделает попытку сохранить целое значение в свойство-зависимость типа *double*. Возникнет ошибка времени выполнения. Типы данных должны быть заданы явно:

```
public static readonly DependencyProperty MyDoubleProperty =
    DependencyProperty.Register("MyDouble",
        typeof(double),
        typeof(SomeClass),
        new PropertyMetadata(10.0, OnMyDoubleChanged));
```

Может возникнуть необходимость в создании свойства-зависимости только для чтения. (Например, для свойств *ActualWidth* и *ActualHeight* класса *FrameworkElement* имеются только методы доступа, возвращающие значения.) На первый взгляд, кажется, все просто:

```
public double MyDouble
{
    private set { SetValue(MyDoubleProperty, value); }
    get { return (double)GetValue(MyDoubleProperty); }
}
```

Теперь только сам класс может задавать свойство.

Но подождите! Как говорилось выше, метод *SetValue* является открытым, так что любой класс может вызвать *SetValue*, чтобы задать значение этого свойства. Чтобы защитить свойство-зависимость от неавторизованного доступа, понадобится создать исключение, которое будет формироваться при попытке задания значения свойства из кода, внешнего по отношению к классу. Вероятно, самой простой логикой будет устанавливать закрытый флаг при задании свойства внутри класса и затем в обработчике события изменения значения свойства проводить проверку на наличие этого закрытого флага.

Из документации без труда можно понять, подкреплено ли конкретное свойство существующего класса свойством-зависимостью. Просто поищите в разделе Fields (Поля) статическое поле типа *DependencyProperty*, имя которого соответствует имени свойства, дополненному словом *Property*.

Наличие статического поля *DependencyProperty* позволяет коду или разметке ссылаться на конкретное свойство, описанное классом, независим от какого-либо экземпляра данного класса, даже если этот экземпляр еще не создан. В некоторых методах – например, методе *SetBinding* (Задать привязку), определенным классом *FrameworkElement* – есть аргументы, которые позволяют ссылаться на конкретное свойство, и свойство-зависимость идеально подходит для этого.

Наконец, не чувствуйте себя обязанным делать *все* свойства своих классов свойствами-зависимостями. Если свойство никогда не будет целевым для стиля, привязки данных или анимации, нет никакой проблемы в том, если оно будет просто обычным свойством.

Например, для обеспечения возможности пользователю выбирать объект типа *Color* планируется использовать несколько элементов управления *RadioButton*. Для этого можно создать класс, производный от *RadioButton*, и определить в нем свойство, которое связывало бы каждый *RadioButton* с объектом *Color*:

```
public class ColorRadioButton : RadioButton
{
    public Color ColorTag { set; get; }
}
```

Теперь данное свойство можно задавать в XAML и затем использовать в коде для определения того, какой *Color* представляет каждый *RadioButton*. В таком простом случае свойство-зависимость ни к чему.

Наследование от *UserControl*

Как было показано, для введения некоторых дополнительных свойств можно наследоваться от класса, производного от *Control*. Чтобы создать совершенно новый элемент управления, можно наследоваться напрямую от *Control* (или от *ContentControl*, если элемент управления должен иметь свойство *Content*). Но наследование от *Control* или *ContentControl* по всем правилам подразумевает создание в XAML шаблона по умолчанию, который описывал бы внешний вид элемента управления, и обеспечение возможности замены этого шаблона в случае необходимости переопределения визуального представления элемента управления.

Все это не так сложно, но требует тщательного анализа того, как будет настроен элемент управления. Некоторые из связанных с этим вопросы рассматриваются в главе 16.

Если вы занимаетесь созданием элементов управления на коммерческой основе, создаваемые и продаваемые вами пользовательские элементы управления должны наследоваться от *Control* или *ContentControl*, или *ItemsControl* (Элемент управления списками) (глава 17). Сменный шаблон является важнейшей составляющей коммерческих элементов управления.

Но некоторые пользовательские элементы управления не требуют дополнительной настройки визуального представления. Элементы управления, используемые только в определенном проекте либо только одним разработчиком или группой разработки компании, или элементы управления, наследующие внешний вид, обычно не нуждаются в обеспечении возможности замены шаблона.

Для таких элементов управления идеальным решением часто является наследование от *UserControl*. (*User* (Пользователь) в *UserControl* – это вы, разработчик.) Более того, у нас уже имеется опыт наследования от *UserControl*! Класс *PhoneApplicationPage* наследуется от *Page*, который является производным от *UserControl*.

UserControl имеет свойство *Content* типа *UIElement*. При наследовании от *UserControl* новые свойства обычно определяются в коде (и часто также методы и события), но визуальные элементы описываются в XAML через задание дерева визуальных элементов для свойства *Content*. Это делает невозможным использование свойства *Content* для каких-либо иных целей. Если производный от *UserControl* класс должен включать свойство *Content*, следует описать новое свойство *Child* или что-то подобное для использования в тех же целях.

Широкое применение *UserControl* – идеальный способ разбиения визуальных элементов приложения на модули.

Например, в XAML-файле приложения *ColorScroll* из предыдущей главы было много повторяющихся фрагментов: три строки, каждая из которых включала *Slider* и два элемента *TextBlock*. Чтобы адаптировать идею *ColorScroll* в пригодный для повторного использования элемент управления, возможно, следует начать с наследования класса *ColorColumn* (Цветовая дорожка) от *UserControl* и затем поместить три элемента управления *ColorColumn* в производный от *UserControl* класс *RgbColorScroller* (Полоса прокрутки RGB-цвета).

Оба класса, *ColorColumn* и *RgbColorScroller*, можно найти в проекте библиотеки динамической компоновки (DLL) под названием Petzold.Phone.Silverlight. Создать DLL в Visual Studio для приложений Windows Phone просто: в диалоговом окне New Project в левой панели выбираем Silverlight for Windows Phone и в средней панели – Windows Phone Class Library (Библиотека классов Windows Phone). (Для целей тестирования следует либо создать второй проект приложения в том же решении в качестве библиотеки; либо пользовательские классы можно создавать в проекте приложения и переносить их в библиотеку после тестирования.)

В проект Petzold.Phone.Silverlight (или любой другой проект библиотеки) можно добавить новый элемент. Для этого в Solution Explorer щелкните правой кнопкой мыши имя проекта и выберите в появившемся меню Add и New Item.

Чтобы создать новый *UserControl* в проекте приложения или библиотеки, в диалоговом окне Add New Item выберите Windows Phone User Control (Пользовательский элемент управления Windows Phone) и задайте его имя. В результате этого будет создано два файла: XAML-файл и файл выделенного кода.

XAML-файл намного проще того, который был создан для класса *PhoneApplicationPage*. Корневым элементом в нем является *UserControl*. Он включает атрибут *x:Class*, обозначающий, что это производный класс, и единственный вложенный элемент – *Grid* под именем *LayoutRoot*. В этом *Grid* нет необходимости, но с ним удобней.

Корневой элемент включает атрибуты для задания следующих свойств:

```
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
```

В этих атрибутах нет необходимости и их можно удалить. Все три свойства наследуются по дереву визуальных элементов, поэтому *UserControl* обычно получает их значения от *MainPage*. Задавая эти свойства здесь, мы аннулируем всю разметку (или код) их задания в создаваемом элементе управления. Эти свойства следует оставить в *UserControl*, только если они используются создаваемым элементом управления.

Также я удалил атрибуты, связанные с дизайнером. Итак, рассмотрим файл ColorColumn.xaml полностью. Обратите внимание, что я изменил значение свойства *Background* для *Grid*. Ранее для него использовался *StaticResource*, ссылающийся на *PhoneChromeBrush*, теперь его значение *Transparent*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ColorColumn.xaml

```
<UserControl
    x:Class="Petzold.Phone.Silverlight.ColorColumn"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Name="colorLabel"
                   Grid.Row="0"
                   TextAlignment="Center" />

        <Slider Name="slider"
                Grid.Row="1"
                Orientation="Vertical"
                Minimum="0"
                Maximum="255"
                ValueChanged="OnSliderValueChanged" />

        <TextBlock Name="colorValue"
                   Grid.Row="2"
                   Text="00"
                   TextAlignment="Center" />
    </Grid>
</UserControl>
```

Grid включает три строки. В верхней строке располагается *TextBlock* под именем *colorLabel* (Цвет), затем следует *Slider* с диапазоном значений от 0 до 255 и еще один *TextBlock*, называющийся *colorValue* (Значение цвета). В *Slider* задан обработчик события *OnSliderValueChanged*.

В примере ColorScroll предыдущей главы элементы управления *Slider* и элементы *TextBlock* закрашивались красным, зеленым или синим посредством свойства *Foreground*. Поскольку свойство *Foreground* наследуется через дерево визуальных элементов, достаточно задать его один раз для любого экземпляра *ColumnColumn* и позволить ему наследоваться вниз по дереву.

Элемент *colorLabel* будет отображать текст, соответствующий заданному цвету. Но я решил обработать этот текст несколько иначе с помощью свойства, специально предусмотренного для данной цели.

Это означает, что класс *ColorColumn* описывает два свойства: свойство *Label* (Надпись), а также более ожидаемое свойство *Value*, значение которого соответствует положению *Slider*. Как и сам *Slider*, класс *ColorColumn* тоже определяет событие *ValueChanged*, сигнализирующее о моменте изменения значения *Slider*.

Как правило, производный от *UserControl* класс описывает собственные свойства и события, и очень часто эти свойства и события повторяют свойства и события элементов его визуального дерева. Типично для такого класса, как *ColorColumn*, иметь свойство *Label*, соответствующее свойству *Text* элемента *TextBlock*, и свойство *Value*, соответствующее свойству *Value* элемента *Slider*, а также событие *ValueChanged*, соответствующее событию *ValueChanged* элемента *Slider*.

Рассмотрим фрагмент файла выделенного кода *ColorColumn*, описывающий свойство *Label*, которое обеспечивает вывод надписи над *Slider*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ColorColumn.xaml.cs (фрагмент)

```
public partial class ColorColumn : UserControl
{
    …
    public static readonly DependencyProperty LabelProperty =
        DependencyProperty.Register("Label",
            typeof(string),
            typeof(ColorColumn),
            new PropertyMetadata(OnLabelChanged));
    …
    public string Label
    {
        set { SetValue(LabelProperty, value); }
        get { return (string)GetValue(LabelProperty); }
    }
    …
    static void OnLabelChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as ColorColumn).colorLabel.Text = args.NewValue as string;
    }
}
```

Обработчик изменения значения свойства *Label* просто задает значение свойства *Text* элемента *colorLabel*. Это один из способов передачи значения свойства из пользовательского элемента управления в свойство элемента дерева визуальных элементов. В следующей главе я продемонстрирую более простой подход с использованием привязки данных.

Свойство *Value* в *ColorColumn* несколько сложнее, потом что оно должно формировать событие *ValueChanged*. Это свойство *Value* в итоге используется при вычислении значения *Color*, поэтому я сделал его типа *byte*, а не *double*. Рассмотрим код класса, описывающий свойство *Value* и событие *ValueChanged*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ColorColumn.xaml.cs (фрагмент)

```
public partial class ColorColumn : UserControl
{
    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value",
            typeof(byte),
            typeof(ColorColumn),
            new PropertyMetadata((byte)0, OnValueChanged));
    …
    public event RoutedPropertyChangedEventHandler<byte> ValueChanged;
    …
    public byte Value
    {
        set { SetValue(ValueProperty, value); }
        get { return (byte)GetValue(ValueProperty); }
    }
    …
    static void OnValueChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as ColorColumn).OnValueChanged((byte)args.OldValue,
(byte)args.NewValue);
    }

    protected virtual void OnValueChanged(byte oldValue, byte newValue)
    {
        slider.Value = newValue;
        colorValue.Text = newValue.ToString("X2");

        if (ValueChanged != null)
            ValueChanged(this,
                new RoutedPropertyChangedEventArgs<byte>(oldValue, newValue));
    }
    …
}
```

Для описания события *ValueChanged* я использовал универсальный *RoutedPropertyChangedEventHandler* (Обработчик маршрутизируемого события изменения значения свойства) и соответствующий *RoutedPropertyChangedEventArgs* (Аргументы маршрутизируемого события изменения значения свойства). Они хороши для сигнализации об изменении значения свойства-зависимости, потому что обеспечивают передачу старого и нового значения.

Статический метод *OnValueChanged* вызывает защищенный виртуальный метод экземпляра с таким же именем, *OnValueChanged*, но его аргументы указывают на старое и новое значения свойства. (На создание моей версии меня вдохновил метод *OnValueChanged* класса *RangeBase*.) Этот метод экземпляра устанавливает *Slider* и задает значение *TextBlock* соответственно текущему значению и формирует событие *ValueChanged*.

В коде *ColorColumn* нам осталось обсудить только лишь конструктор и обработчик события *ValueChanged* объекта *Slider*. Этот обработчик события просто приводит свойство *Value* класса *Slider* к типу *byte* и присваивает ему значение свойства *Value* класса *ColorColumn*.

Проект Silverlight: Petzold.Phone.Silverlight Файл: ColorColumn.xaml.cs (фрагмент)

```
public partial class ColorColumn : UserControl
{
    …
    public ColorColumn()
    {
        InitializeComponent();
    }
    …
    void OnSliderValueChanged(object sender,
                              RoutedPropertyChangedEventArgs<double> args)
    {
        Value = (byte)args.NewValue;
    }
}
```

И здесь можно выявить бесконечный цикл: пользователь перемещает ползунок, объект *Slider* формирует событие *ValueChanged*, метод *OnSliderValueChanged* задает значение свойству *Value* объекта *ColorColumn*, вызывается статический обработчик события изменения значения свойства, статический метод вызывает метод экземпляра *OnValueChanged*, который задает значение свойства *Value* объекта *Slider*, что приводит к формированию следующего события *ValueChanged*, и т.д.

В реальности этого не происходит, потому что в некоторый момент одному из этих свойств *Value* – либо *Value* объекта *Slider*, либо *Value* объекта *ColorColumn* – будет присвоено его текущее значение, т.е. событие изменения свойства сформировано не будет. Цикл прервется.

Класс *RgbColorScroller* также наследуется от *UserControl* и включает три элемента управления *ColorColumn*. Привожу XAML-файл полностью:

Проект Silverlight: Petzold.Phone.Silverlight Файл: RgbColorScroller.xaml

```
<UserControl
    x:Class="Petzold.Phone.Silverlight.RgbColorScroller"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:petzold="clr-namespace:Petzold.Phone.Silverlight">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <petzold:ColorColumn x:Name="redColumn"
                             Grid.Column="0"
                             Foreground="Red"
                             Label="Red"
                             ValueChanged="OnColorColumnValueChanged" />

        <petzold:ColorColumn x:Name="greenColumn"
                             Grid.Column="1"
                             Foreground="Green"
                             Label="Green"
                             ValueChanged="OnColorColumnValueChanged" />

        <petzold:ColorColumn x:Name="blueColumn"
                             Grid.Column="2"
                             Foreground="Blue"
                             Label="Blue"
                             ValueChanged="OnColorColumnValueChanged" />
    </Grid>
</UserControl>
```

Свойству *Foreground* каждого из трех элементов управления *ColorColumn* задан один из трех цветов. Свойствам *Label* заданы аналогичные значения, но типа *string*, а не *Color*.

Обратите внимание, каждый *ColorColumn* идентифицирован с помощью атрибута *x:Name*, а не *Name*. Обычно я использую *Name*, но для ссылки на класс из той же сборки *Name* использоваться не может, а у нас оба класса, *ColorColumn* и *RgbColorScroller*, располагаются в сборке Petzold.Phone.Silverlight.

Класс *RgbColorScroller* определяет одно свойство с именем *Color* (типа *Color*, конечно) и событие *ColorChanged*. Рассмотрим класс полностью:

Проект Silverlight: Petzold.Phone.Silverlight Файл: RgbColorScroller.xaml.cs (фрагмент)

```
public partial class RgbColorScroller : UserControl
{
    public static readonly DependencyProperty ColorProperty =
        DependencyProperty.Register("Color",
            typeof(Color),
            typeof(RgbColorScroller),
            new PropertyMetadata(Colors.Gray, OnColorChanged));

    public event RoutedPropertyChangedEventHandler<Color> ColorChanged;

    public RgbColorScroller()
    {
        InitializeComponent();
    }

    public Color Color
    {
        set { SetValue(ColorProperty, value); }
        get { return (Color)GetValue(ColorProperty); }
    }

    void OnColorColumnValueChanged(object sender,
                    RoutedPropertyChangedEventArgs<byte> args)
    {
        Color = Color.FromArgb(255, redColumn.Value,
                                    greenColumn.Value,
                                    blueColumn.Value);
    }

    static void OnColorChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as RgbColorScroller).OnColorChanged((Color)args.OldValue,
                                                 (Color)args.NewValue);
    }

    protected virtual void OnColorChanged(Color oldValue, Color newValue)
    {
        redColumn.Value = newValue.R;
        greenColumn.Value = newValue.G;
        blueColumn.Value = newValue.B;

        if (ColorChanged != null)
            ColorChanged(this,
                new RoutedPropertyChangedEventArgs<Color>(oldValue, newValue));
    }
}
```

При изменении значения свойства *Color* вызывается два метода *OnColorChanged*. Они приводят значения свойства *Color* к типу *byte*, задают значения свойствам *Value* каждого из объектов *ColorColumn* и формируют событие *ColorChanged*.

Обработчик *OnColorColumnValueChanged* (При изменении значения цвета столбца) вызывается при формировании события *ValueChanged* одним из трех элементов управления *ColorColumn*. Этот обработчик выполняет агрегацию трех значений цвета типа *byte*, полученных от трех элементов управления *ColorColumn*, в одно значение *Color*.

Опять же, создается впечатление, что может возникнуть бесконечный цикл, но в реальности этого не происходит.

Чтобы использовать класс *RgbColorScroller* из библиотеки Petzold.Phone.Silverlight, создаем новый проект приложения. Назовем его SelectTwoColors (Выбор двух цветов). В Solution Explorer щелкнем правой кнопкой мыши заголовок References под именем проекта и выберем Add Reference. В открывшемся диалоговом окне Add Reference выбираем вкладку Browse (Обзор). Переходим к файлу DLL (в данном случае к Petzold.Phone.Silverlight.dll) и выбираем его.

В файл MainPage.xaml понадобится включить объявление пространства имен XML для этой библиотеки. Поскольку библиотека является отдельной сборкой, в этом объявлении должен быть раздел assembly (сборка) для ссылки на файл DLL:

```
xmlns:petzold="clr-
namespace:Petzold.Phone.Silverlight;assembly=Petzold.Phone.Silverlight"
```

В XAML-файле SelectTwoColors имеется два элемента управления, каждый из которых располагается в *Border*, и между ними элемент *Rectangle*. События *ColorChanged* обоих *RgbColorScroll* ассоциированы с одним обработчиком:

Проект Silverlight: SelectTwoColors   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Border Grid.Column="0"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="2"
            Margin="12"
            Padding="12">

        <petzold:RgbColorScroller
                    Name="colorScroller1"
                    ColorChanged="OnColorScrollerColorChanged" />
    </Border>

    <Rectangle Name="rectangle"
               Grid.Column="1"
               StrokeThickness="24"
               Margin="12" />

    <Border Grid.Column="2"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="2"
            Margin="12"
            Padding="12">

        <petzold:RgbColorScroller
```

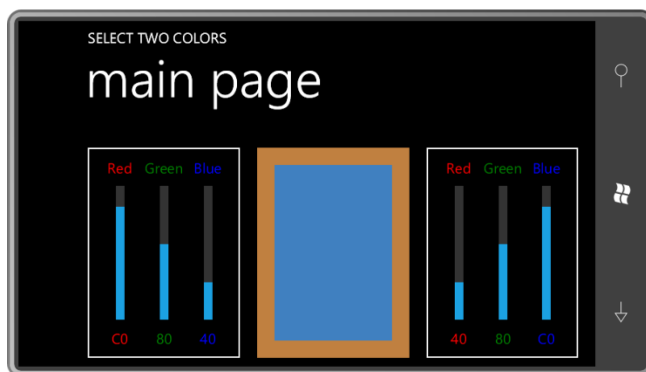
```
                    Name="colorScroller2"
                    ColorChanged="OnColorScrollerColorChanged" />

    </Border>
</Grid>
```

Конструктор файла выделенного кода инициализирует эти два элемента управления *RgbColorScroller*, задавая каждому значение цвета. Это приводит к формированию первых событий *ColorChanged*. Обработчик событий обрабатывает их и задает цвета *Rectangle*:

Проект Silverlight: SelectTwoColors Файл: MainPage.xaml.cs (фрагмент)

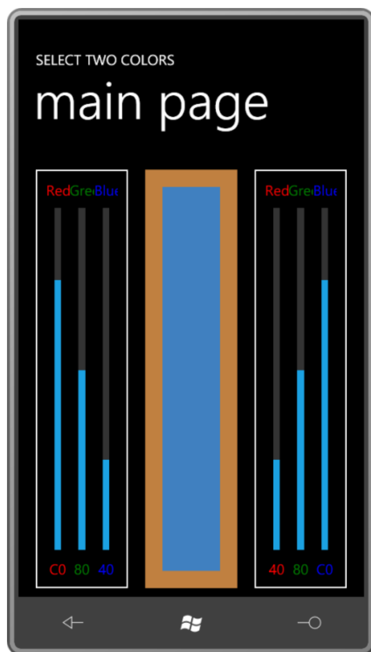
```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();

        colorScroller1.Color = Color.FromArgb(0xFF, 0xC0, 0x80, 0x40);
        colorScroller2.Color = Color.FromArgb(0xFF, 0x40, 0x80, 0xC0);
    }

    void OnColorScrollerColorChanged(object sender,
                                     RoutedPropertyChangedEventArgs<Color> args)
        {
        Brush brush = new SolidColorBrush(args.NewValue);

        if (sender == colorScroller1)
            rectangle.Stroke = brush;

        else if (sender == colorScroller2)
            rectangle.Fill = brush;
    }
}
```

Посмотрим, что мы получаем в альбомном режиме отображения:



Обратите внимание, подписи в каждом элементе управления *ColorColumn* подхватили значение свойства *Foreground* через наследование свойств. Со *Slider* этого не произошло. Я подозреваю, что свойство *Foreground* задано в стиле темы *Slider* и это блокирует наследование свойств. Если бы наследование этого свойства было действительно важным, возможно, я бы описал новое свойство *Color* для *ColorColumn* и использовал его для программного задания свойства *Foreground* объекта *Slider*.

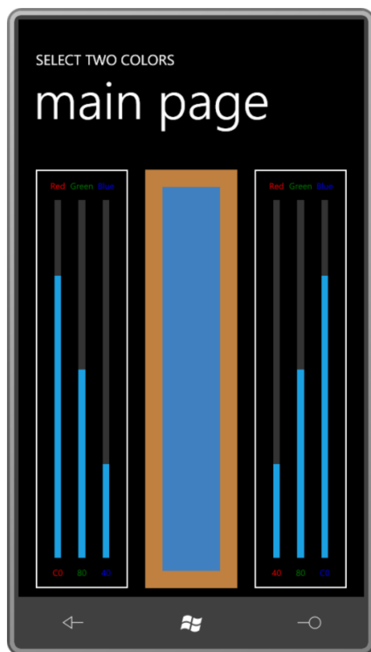
Я намеренно спроектировал компоновку SelectTwoColors так, чтобы она плохо смотрелась в портретном режиме:



Как видите, текст сплывается. Но ничего страшного, достаточно лишь задать меньшее значение свойству *FontSize* прямо в элементах управления *RgbColorScroller*:

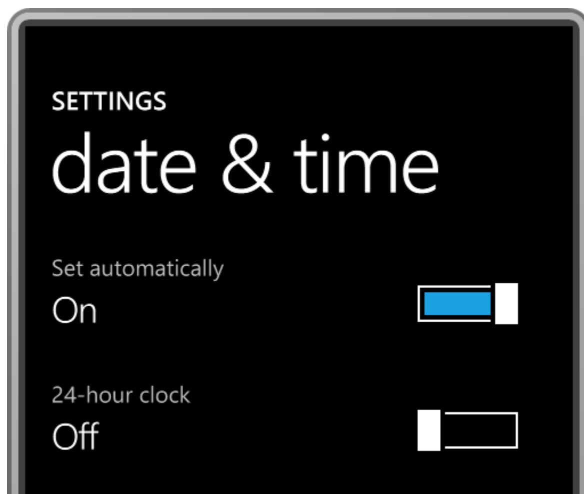
```
FontSize="12"
```

Это одно свойство определяет размер текста всех шести элементов *TextBlock* в обоих элементах управления. Конечно, текст становится очень маленьким, но больше не перекрывается:



Новый тип переключателя

В некоторых телефонах, работающих под управлением Windows Phone 7, можно увидеть новый стиль переключателей. Вот как они выглядят на странице для задания даты и времени при практически двукратном увеличении:



Если немного поэкспериментировать с этими элементами управления, можно обнаружить, что переключение происходит просто в результате касания. Но также можно «перетягивать» ползунок переключателя, проводя пальцем по экрану, но он имеет тенденцию перескакивать в крайнюю левую или крайнюю правую позицию.

В своем примере я не собираюсь дублировать более сложное движение. Моя версия переключателя будет отвечать только на касания. Поэтому я назвал его *TapSlideToggle* (Переключатель, переключающийся по касанию). Кнопка – это производный от *UserControl* класс, размещающийся в библиотеке *Petzold.Phone.Silverlight*. (Должен заметить, что-то подобное можно реализовать полностью в шаблоне, применяемом к существующему *ToggleButton*, и набор инструментов Silverlight for Windows Phone Toolkit реализовывает этот элемент управления под именем *ToggleSwitchButton*.) Рассмотрим полный XAML-файл моей версии реализации:

Проект Silverlight: Petzold.Phone.Silverlight Файл: TapSlideToggle.xaml

```
<UserControl x:Class="Petzold.Phone.Silverlight.TapSlideToggle"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             mc:Ignorable="d"
             d:DesignHeight="36" d:DesignWidth="96">

    <Grid x:Name="LayoutRoot"
          Background="Transparent"
          Width="96" Height="36">

        <Border BorderBrush="{StaticResource PhoneForegroundBrush}"
                BorderThickness="2"
                Margin="4 2"
                Padding="4">
            <Rectangle Name="fillRectangle"
                       Fill="{StaticResource PhoneAccentBrush}"
                       Visibility="Collapsed" />
        </Border>

        <Border Name="slideBorder"
                BorderBrush="{StaticResource PhoneBackgroundBrush}"
                BorderThickness="4 0"
                HorizontalAlignment="Left">
            <Rectangle Stroke="{StaticResource PhoneForegroundBrush}"
                       Fill="White"
                       StrokeThickness="2"
```

```
                       Width="20" />
        </Border>
    </Grid>
</UserControl>
```

Размер кнопки задается в *Grid*. Размер элемента управления лучше задавать именно в *Grid*, а не через свойства *Height* и *Width* самого элемента управления. Я также изменил связанные с дизайнером атрибуты так, чтобы почувствовать, как эти элементы управления выглядят в дизайнере.

Должен признаться, я не вполне доволен выбранным здесь подходом. Чтобы не усложнять приложение, я ограничился двумя элементами *Border*, каждый из которых включает по элементу *Rectangle*, но, чтобы создать зазор между большим ползунком и белым фоном, я задал свойству *BorderBrush* второго *Border* значение, соответствующее цвету фона. Кнопка будет хорошо смотреться только на поверхности, закрашенной с помощью ресурса *PhoneBackgroundBrush* (Кисть фона телефона).

Чтобы сделать наш переключатель хоть немного похожим на обычный *ToggleButton* (но без опции трех состояний), в файле выделенного кода определяем свойство-зависимость *IsChecked* типа *bool* и два события: *Checked* и *Unchecked*. При изменении значения свойства *IsChecked* формируется одно из этих событий:

Проект Silverlight: Petzold.Phone.Silverlight Файл: TapSlideToggle.xaml.cs (фрагмент)

```
public partial class TapSlideToggle : UserControl
{
    public static readonly DependencyProperty IsCheckedProperty =
        DependencyProperty.Register("IsChecked",
            typeof(bool),
            typeof(TapSlideToggle),
            new PropertyMetadata(false, OnIsCheckedChanged));

    public event RoutedEventHandler Checked;
    public event RoutedEventHandler Unchecked;

    public TapSlideToggle()
    {
        InitializeComponent();
    }

    public bool IsChecked
    {
        set { SetValue(IsCheckedProperty, value); }
        get { return (bool)GetValue(IsCheckedProperty); }
    }

    …

    static void OnIsCheckedChanged(DependencyObject obj,
                                   DependencyPropertyChangedEventArgs args)
    {
        (obj as TapSlideToggle).OnIsCheckedChanged(args);
    }

    void OnIsCheckedChanged(DependencyPropertyChangedEventArgs args)
    {
        fillRectangle.Visibility = IsChecked ? Visibility.Visible :
                                               Visibility.Collapsed;

        slideBorder.HorizontalAlignment = IsChecked ? HorizontalAlignment.Right :
                                                      HorizontalAlignment.Left;
```

```
        if (IsChecked && Checked != null)
            Checked(this, new RoutedEventArgs());

        if (!IsChecked && Unchecked != null)
            Unchecked(this, new RoutedEventArgs());
    }
}
```

Статический обработчик событий изменения значения свойства вызывает метод-обработчик экземпляра с таким же именем, который немного изменяет визуальные элементы в XAML и формирует одно из двух событий. В приведенном выше коде не хватает только переопределений двух событий *Manipulation*. Рассмотрим их:

Проект Silverlight: Petzold.Phone.Silverlight Файл: TapSlideToggle.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    args.Handled = true;
    base.OnManipulationStarted(args);
}

protected override void OnManipulationCompleted(ManipulationCompletedEventArgs args)
{
    Point pt = args.ManipulationOrigin;

    if (pt.X > 0 && pt.X < this.ActualWidth &&
        pt.Y > 0 && pt.Y < this.ActualHeight)
        IsChecked ^= true;

    args.Handled = true;
    base.OnManipulationCompleted(args);
}
```

Я решил переключать кнопку, только если пользователь нажимает и затем отпускает ее, оставляя при этом палец на кнопке. Это обычный подход. Перегруженное событие *OnManipulationStarted* задает свойству *Handled* (Обработано) значение *true*, чтобы не допустить распространение события вверх по дереву визуальных элементов, и как результат просигнализировать, что кнопка производит данную манипуляцию. После этого перегруженное событие *OnManipulationCompleted* проверяет, попадает ли значение свойства *ManipulationOrigin* (Центр манипуляции) в границы элемента управления. Если да, свойство *IsChecked* меняет значение на противоположное:

```
IsChecked ^= true;
```

Протестируем функциональность с помощью приложения TapSlideToggleDemo. Область содержимого определяет два экземпляра *TapSlideToggle* и два элемента *TextBlock* для отображения их текущего состояния:

Проект Silverlight: TapSlideToggleDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
```

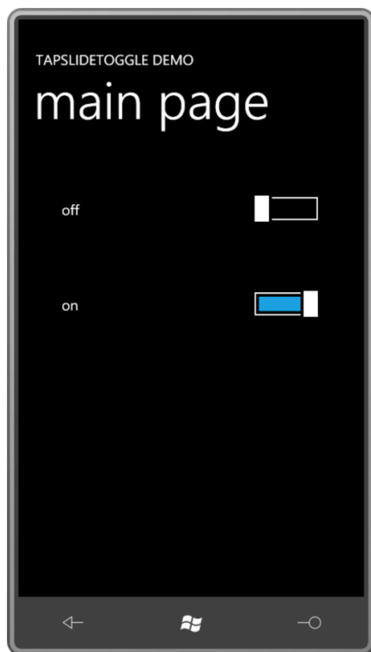
```
    <TextBlock Name="option1TextBlock"
               Grid.Row="0" Grid.Column="0"
               Text="off"
               Margin="48"
               VerticalAlignment="Center" />

    <petzold:TapSlideToggle Name="slideToggle1"
                            Grid.Row="0" Grid.Column="1"
                            Margin="48"
                            HorizontalAlignment="Right"
                            Checked="OnSlideToggle1Checked"
                            Unchecked="OnSlideToggle1Checked" />

    <TextBlock Name="option2TextBlock"
               Grid.Row="1" Grid.Column="0"
               Text="off"
               Margin="48"
               VerticalAlignment="Center" />

    <petzold:TapSlideToggle Name="slideToggle2"
                            Grid.Row="1" Grid.Column="1"
                            Margin="48"
                            HorizontalAlignment="Right"
                            Checked="OnSlideToggle2Checked"
                            Unchecked="OnSlideToggle2Checked" />
</Grid>
```

События *Checked* и *Unchecked* экземпляра *TapSlideToggle* обрабатываются одним обработчиком, но для каждого экземпляра *TapSlideToggle* существует свой обработчик этих событий. Благодаря этому каждый обработчик может определять состояние кнопки, получая значение свойства *IsChecked* и задавая соответствующее значение *TextBlock*:

Проект Silverlight: TapSlideToggleDemo Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        slideToggle2.IsChecked = true;
    }

    void OnSlideToggle1Checked(object sender, RoutedEventArgs args)
    {
        TapSlideToggle toggle = sender as TapSlideToggle;
        option1TextBlock.Text = toggle.IsChecked ? "on" : "off";
    }

    void OnSlideToggle2Checked(object sender, RoutedEventArgs args)
    {
        TapSlideToggle toggle = sender as TapSlideToggle;
        option2TextBlock.Text = toggle.IsChecked ? "on" : "off";
    }
}
```

И вот результат:



Для этой кнопки не предусмотрено визуальное представление неактивного состояния. Когда свойству *IsEnabled* задано значение *false*, элемент управления автоматически перестает получать пользовательский ввод, но визуальная передача этого состояния должна быть реализована самим элементом управления. Обычно для этого используют полупрозрачный черный *Rectangle*, который перекрывает весь элемент управления. Когда кнопка активна, свойство *Visibility* этого прямоугольника имеет значение *Collapsed*. Когда свойство *IsEnabled* принимает значение *true*, свойству *Visibility* этого *Rectangle* задается значение *Visible*, что приводит к эффекту «затенения» визуальных элементов элемента управления.

Панели и свойства

В главе 9 мы рассмотрели создание пользовательских панелей, но они были довольно примитивными, потому что не имели свойств. У большинства пользовательских панелей есть специальные свойства, и некоторые из них также описывают присоединенные свойства.

В Windows Presentation Foundation есть панель, которую я нахожу весьма полезной. Это *UniformGrid* (Унифицированная сетка). Как можно предположить из ее имени, *UniformGrid* разделяет свою область содержимого на равные ячейки.

По умолчанию *UniformGrid* автоматически определяет число строк и столбцов по округленному в большую сторону значению квадратного корня из количества дочерних элементов. Например, если имеется 20 дочерних элементов, *UniformGrid* создаст 5 строк и столбцов, даже несмотря на то что более логичным кажется получить 5 строки и 4 столбца или 4 строки и 5 столбцов. Такое поведение по умолчанию можно переопределить, задав свойству *Rows* (Строки) или *Columns* (Столбцы) объекта *UniformGrid* отличные от нуля числовые значения.

Практически всегда я задаю *Rows* или *Columns* равным 1, получая в результате один столбец или строку, разбитые на равные ячейки. В данном случае все происходит не так, как со *StackPanel*, который выходит за границы экрана, если содержит слишком много дочерних элементов. Это скорее похоже на поведение *Grid* с одним столбцом или одной строкой, когда свойство *GridLength* (Протяженность сетки) в *RowDefinition* или *ColumnDefinition* имеет значение *Star* (Звезда), и, следовательно, обеспечивается равномерное распределение имеющегося пространства между ячейками.

Своей версии *UniformGrid* я дал имя *UniformStack* (Унифицированный стек). В этом классе нет свойства *Rows* или *Columns*, но имеется свойство *Orientation* – такое же свойство описано в *StackPanel* – для обозначения того, как будет ориентирована панель, вертикально или горизонтально.

Рассмотрим фрагмент класса *UniformStack*, в котором описывается единственное свойство-зависимость и обработчик события изменения значения свойства:

Проект Silverlight: Petzold.Phone.Silverlight Файл: UniformStack.cs (фрагмент)

```
public class UniformStack : Panel
{
    public static readonly DependencyProperty OrientationProperty =
        DependencyProperty.Register("Orientation",
            typeof(Orientation),
            typeof(UniformStack),
            new PropertyMetadata(Orientation.Vertical, OnOrientationChanged));

    public Orientation Orientation
    {
        set { SetValue(OrientationProperty, value); }
        get { return (Orientation)GetValue(OrientationProperty); }
    }

    static void OnOrientationChanged(DependencyObject obj,
                                     DependencyPropertyChangedEventArgs args)
    {
        (obj as UniformStack).InvalidateMeasure();
    }

    …
}
```

В описаниях свойства-зависимости и CLR-свойства нет ничего сложного. Обработчик события изменения значения свойства приводит первый аргумент к типу класса, как обычно, и затем просто вызывает метод *InvalidateMeasure* (Аннулировать размеры). Этот метод определен в *UIElement* и буквально говорит системе компоновки: «Даже если ты думаешь, что знаешь мои размеры, забудь. Я совершенно другой». Этот код инициирует пересчет размеров всех элементов компоновки, начиная от корневого элемента вниз по дереву визуальных элементов, потому что размер этой панели может повлиять на размеры родительских элементов. За пересчетом размеров элементов компоновки сразу же автоматически следует их перекомпоновка. (Пересчет размеров выполняется при каждом изменении размеров панели, либо при добавлении или удалении элементов из ее коллекции *Children*, либо при изменении размеров любого из имеющихся дочерних элементов.)

Также имеется метод *InvalidateArrange* (Аннулировать компоновку), который запускает вторую половину процесса перекомпоновки, но такой сценарий используется довольно редко. Этот метод пригодится для панели, элементы которой могут перемещаться, но при этом ни они, ни сама панель не меняют размеров.

Вызов *InvalidateMeasure* в итоге приводит к вызову метода *MeasureOverride*. Задумаемся на мгновение, что должно быть сделано.

Рассмотрим горизонтальный *UniformStack*. Предположим, на этой панели расположено пять дочерних элементов. Согласно значению *availableSize*, панель может занять область, шириной (*Width*) 400 и высотой (*Height*) 200 пикселов. Каждому дочернему элементу должна быть предложена область шириной 80 (1/5 общей доступной ширины) и высотой 200 пикселов. Таков принцип панели.

Ну а если свойство *Width* объекта *availableSize* имеет бесконечное значение? Что происходит в этом случае?

А вот это не вполне ясно. Безусловно, у панели нет другого выбора, кроме как предложить каждому дочернему элементу неограниченную ширину. После этого единственным логичным решением будет возвращение методом *MeasureOverride* значения *Width*, которое в пять раз больше значения *Width* самого широкого дочернего элемента.

Именно это я и делаю в следующем фрагменте кода:

Проект Silverlight: Petzold.Phone.Silverlight Файл: UniformStack.cs (фрагмент)

```
protected override Size MeasureOverride(Size availableSize)
{
    if (Children.Count == 0)
        return new Size();

    Size availableChildSize = new Size();
    Size maxChildSize = new Size();
    Size compositeSize = new Size();

    // Вычисляем доступный размер для каждого дочернего элемента
    if (Orientation == Orientation.Horizontal)
        availableChildSize = new Size(availableSize.Width / Children.Count,
                                      availableSize.Height);
    else
        availableChildSize = new Size(availableSize.Width,
                                      availableSize.Height / Children.Count);

    // Перебираем все дочерние элементы и находим максимальную ширину и высоту
    foreach (UIElement child in Children)
    {
        child.Measure(availableChildSize);
        maxChildSize.Width = Math.Max(maxChildSize.Width, child.DesiredSize.Width);
        maxChildSize.Height = Math.Max(maxChildSize.Height,
child.DesiredSize.Height);
    }

    // Теперь определяем совокупный размер, зависящий от доступной
    // неограниченной ширины или высоты
    if (Orientation == Orientation.Horizontal)
    {
        if (Double.IsPositiveInfinity(availableSize.Width))
            compositeSize = new Size(maxChildSize.Width * Children.Count,
                                     maxChildSize.Height);
        else
            compositeSize = new Size(availableSize.Width, maxChildSize.Height);
    }
    else
    {
        if (Double.IsPositiveInfinity(availableSize.Height))
            compositeSize = new Size(maxChildSize.Width,
                                     maxChildSize.Height * Children.Count);
        else
            compositeSize = new Size(maxChildSize.Width, availableSize.Height);
    }

    return compositeSize;
}
```

Метод начинается с проверки наличия дочерних элементов у панели, это предупреждает деление на нуль впоследствии.

availableChildSize (Доступный размер дочернего элемента) вычисляется на основании значения свойства *Orientation*. При этом наличие неограниченного размера в *availableSize* для панели игнорируется. (Бесконечность, деленная на количество дочерних элементов, все равно останется бесконечностью; именно это требовалось в данном случае.) При переборе дочерних элементов для каждого из них вызывается метод *Measure* с этим *availableChildSize*. Логика вычисления *DesiredSize* дочернего элемента также игнорирует неограниченные размеры, но находит *maxChildSize* (Максимальный размер дочернего элемента). Это свойство представляет ширину самого широкого дочернего элемента и высоту самого высокого дочернего элемента. Возможно также, что несколько дочерних элементов будут иметь размеры, соответствующие параметрам *maxChildSize*.

При окончательном вычислении *compositeSize* учитываются и *Orientation*, и возможность наличия неограниченного размера. Обратите внимание, что *compositeSize* иногда берет один из размеров *availableSize* за базовый; вообще это не очень правильно, но метод делает это, только если знает, что этот размер не бесконечный.

Метод *ArrangeOverride* вызывает *Arrange* для каждого дочернего элемента, передавая в него один и тот же размер (в методе это параметр *finalChildSize* (Окончательный размер дочернего элемента)), но разные координаты *x* и *y* относительно панели, они зависят от ориентации:

Проект Silverlight: Petzold.Phone.Silverlight Файл: UniformStack.cs (фрагмент)

```
protected override Size ArrangeOverride(Size finalSize)
{
    if (Children.Count > 0)
    {
        Size finalChildSize = new Size();
        double x = 0;
        double y = 0;

        if (Orientation == Orientation.Horizontal)
            finalChildSize = new Size(finalSize.Width / Children.Count,
                                      finalSize.Height);
        else
            finalChildSize = new Size(finalSize.Width,
                                      finalSize.Height / Children.Count);

        foreach (UIElement child in Children)
        {
            child.Arrange(new Rect(new Point(x, y), finalChildSize));

            if (Orientation == Orientation.Horizontal)
                x += finalChildSize.Width;
            else
                y += finalChildSize.Height;
        }
    }

    return base.ArrangeOverride(finalSize);
}
```

Давайте на базе *UniformStack* создадим гистограмму!

Вообще в приложении QuickBarChart используется три панели *UniformStack*:

Проект Silverlight: QuickBarChart Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <petzold:UniformStack Orientation="Vertical">
```

```
        <petzold:UniformStack x:Name="barChartPanel"
                              Orientation="Horizontal" />

        <petzold:UniformStack Orientation="Horizontal">

            <Button Content="Add 10 Items"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center"
                    Click="OnButtonClick" />

            <TextBlock Name="txtblk"
                       Text="0"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center" />
        </petzold:UniformStack>
    </petzold:UniformStack>
</Grid>
```

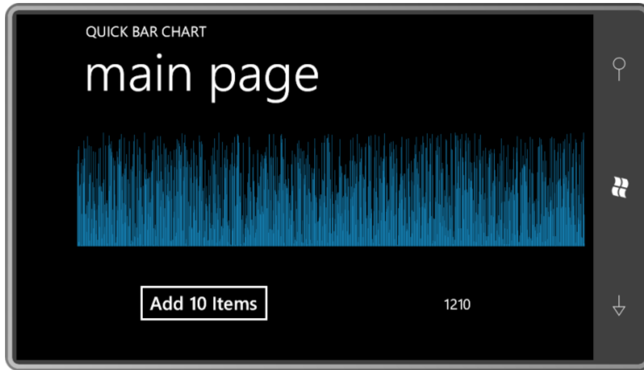
Первый вертикальный *UniformStack* просто разделяет область содержимого на две равные области. (Посмотрите, насколько проще использовать *UniformStack*, чем обычный *Grid*!) В верхней области располагается другой *UniformStack*, который пока пуст. В нижней области находится горизонтальный *UniformStack*, в котором мы разместим кнопку (*Button*) и текстовое поле (*TextBlock*).

По щелчку *Button* файл выделенного кода добавляет по 10 элементов *Rectangle* на панель *UniformStack*, которую я назвал *barChartPanel* (Панель гистограммы):

Проект Silverlight: QuickBarChart   Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();

    public MainPage()
    {
        InitializeComponent();
    }

    void OnButtonClick(object sender, RoutedEventArgs args)
    {
        for (int i = 0; i < 10; i++)
        {
            Rectangle rect = new Rectangle();
            rect.Fill = this.Resources["PhoneAccentBrush"] as Brush;
            rect.VerticalAlignment = VerticalAlignment.Bottom;
            rect.Height = barChartPanel.ActualHeight * rand.NextDouble();
            rect.Margin = new Thickness(0, 0, 0.5, 0);

            barChartPanel.Children.Add(rect);
        }

        txtblk.Text = barChartPanel.Children.Count.ToString();
    }
}
```

Обратите внимание, что у каждого *Rectangle* справа есть небольшое поле (*Margin*) в полпиксела. Это обеспечивает разделение элементов. Просто удивительно, как много прямоугольников можно поместить на экране, прежде чем падет логика отображения:



Присоединенные свойства

Присоединенные свойства на первый взгляд кажутся очень загадочными. Как мы уже знаем из главы 9, так они могут выглядеть в XAML:

```
<Canvas>
    …
    <Ellipse Style="{StaticResource ellipseStyle}"
             Canvas.Left="116" Canvas.Top="92" />
    …
</Canvas>
```

Это фрагмент приложения EllipseChain.

Canvas.Left и *Canvas.Top* – присоединенные свойства. Это свойства, описанные классом *Canvas* и задаваемые для дочерних элементов *Canvas*.

Как я уже говорил в главе 9, в классе *Canvas* нет ни одного члена с именем *Left* или *Top*. Для задания этих присоединенных свойств в коде мы используем два статических метода, определенных классом *Canvas*:

```
Canvas.SetLeft(ellipse, 116);
Canvas.SetTop(ellipse, 92);
```

Или можно воспользоваться методом *SetValue*, унаследованным классом *Ellipse* от *DependencyObject*, и обратиться к свойствам-зависимостям, описанным классом *Canvas*:

```
ellipse.SetValue(Canvas.LeftProperty, 116.0);
ellipse.SetValue(Canvas.TopProperty, 92.0);
```

Это те же самые методы *SetValue*, которые класс вызывает в CLR-свойстве для задания свойства-зависимости.

Это практически все, что необходимо знать для описания собственных присоединенных свойств. В проекте CanvasCloneDemo (Демонстрация клона холста) используется класс *CanvasClone* (Клон холста). Этот клон определяет два поля *DependencyProperty*: *LeftProperty* и *TopProperty*:

Проект: CanvasCloneDemo Файл: CanvasClone.cs (фрагмент)

```
public class CanvasClone : Panel
{
    public static readonly DependencyProperty LeftProperty =
        DependencyProperty.RegisterAttached("Left",
            typeof(double),
            typeof(CanvasClone),
            new PropertyMetadata(0.0, OnLeftOrTopPropertyChanged));
```

```
    public static readonly DependencyProperty TopProperty =
        DependencyProperty.RegisterAttached("Top",
            typeof(double),
            typeof(CanvasClone),
            new PropertyMetadata(0.0, OnLeftOrTopPropertyChanged));
    …
}
```

Обратите внимание, ранее в данной главе объекты *DependencyProperty* создавались с помощью статического метода *DependencyProperty.Register*. Поля *DependencyObject* в *CanvasClone* создаются единственно возможным альтернативным способом: с помощью метода *DependencyProperty.RegisterAttached* (Зарегистрировать присоединенное свойство). Это делает свойства присоединенными и позволяет задавать их для классов, в которых они не описаны.

Первый аргумент конструктора *PropertyMetadata* явно определен типа *double*, благодаря этому не будет возникать ошибки времени выполнения из-за того, что компилятор C# принимает передаваемое значение за *int*.

После того как поля *DependencyProperty* описаны, нам понадобятся статические методы для доступа к присоединенным свойствам. Имена этих методов начинаются с *Set* и *Get*, за которыми следуют имена присоединенных свойств, в данном случае это *Left* и *Top*.

Проект: CanvasCloneDemo Файл: CanvasClone.cs (фрагмент)

```
public static void SetLeft(DependencyObject obj, double value)
{
    obj.SetValue(LeftProperty, value);
}

public static double GetLeft(DependencyObject obj)
{
    return (double)obj.GetValue(LeftProperty);
}

public static void SetTop(DependencyObject obj, double value)
{
    obj.SetValue(TopProperty, value);
}

public static double GetTop(DependencyObject obj)
{
    return (double)obj.GetValue(TopProperty);
}
```

Данные методы вызываются либо явно из кода, либо неявно из синтаксического анализатора XAML. Первый аргумент – это объект, для которого задается присоединенное свойство, т.е. первым аргументом, по всей вероятности, будет дочерний элемент *CanvasClone*. Тело метода использует этот аргумент для вызова методов *SetValue* и *GetValue* дочернего элемента. Эти же методы определены классом *DependencyObject* для задания и возвращения значений свойств-зависимостей.

При изменении значений этих свойств вызывается обработчик события изменения значения свойства, определенный в конструкторе *PropertyMetadata*. Сигнатура этого метода аналогична стандартному методу-обработчику события изменения значения свойства для обычных свойств-зависимостей.

```
static void OnLeftOrTopPropertyChanged(DependencyObject obj,
                                       DependencyPropertyChangedEventArgs args)
```

```
{
    ...
}
```

Опять же, это статический метод. Но первый аргумент не является объектом типа *CanvasClone*. Это *дочерний элемент* объекта *CanvasClone*. Или, *скорее*, это дочерний элемент *CanvasClone*. Метод *CanvasClone.SetLeft* может быть вызван для элемента, который не является дочерним элементом панели. Более того, методы *CanvasClone.SetLeft* и *OnLeftOrTopPropertyChanged* (При изменении свойства Слева или Сверху) могут быть вызваны в условиях, когда не существует никакого экземпляра *CanvasClone*!

По этой причине в теле метода должны быть предприняты определенные меры предосторожности. С помощью вызова такого удобного статического метода *VisualTreeHelper.GetParent* (Получить родителя) он получает аргумент *DependencyObject* и приводит его к *CanvasClone*:

Проект: CanvasCloneDemo Файл: CanvasClone.cs (фрагмент)

```
static void OnLeftOrTopPropertyChanged(DependencyObject obj,
                                       DependencyPropertyChangedEventArgs args)
{
    CanvasClone parent = VisualTreeHelper.GetParent(obj) as CanvasClone;

    if (parent != null)
        parent.InvalidateArrange();
}
```

Если родителем объекта, вызвавшего *CanvasClone.SetLeft* или *CanvasClone.SetTop*, на самом деле является *CanvasClone*, метод вызывает *InvalidateArrange* родителя, т.е. *CanvasClone*.

В общем случае при обработке изменения одного из присоединенных свойств панель, возможно, вызовет метод *InvalidateMeasure* панели, чтобы запустить полный перерасчет компоновки. Однако как видно в следующем методе *MeasureOverride*, при изменении местоположения дочерних элементов размер *CanvasClone* не меняется:

Проект: CanvasCloneDemo Файл: CanvasClone.cs (фрагмент)

```
protected override Size MeasureOverride(Size availableSize)
{
    foreach (UIElement child in Children)
        child.Measure(new Size(Double.PositiveInfinity,
                               Double.PositiveInfinity));

    return Size.Empty;
}
```

Canvas так построен, что *MeasureOverride* всегда возвращает нуль независимо от наличия дочерних элементов, поэтому *CanvasClone* делает то же самое. *MeasureOverride* по-прежнему должен вызвать *Measure* для всех дочерних элементов, в противном случае для них не будет задан размер, но *Measure* вызывается с неограниченными размерами, что заставляет дочерние элементы принимать минимально возможные размеры.

Когда панель вызывает собственный *InvalidateArrange*, начинается перекомпновка и вызывается *ArrangeOverride*. Этот метод заставляет панель расставить элементы на своей поверхности. По сути, он задает размер и местоположение каждого дочернего элемента.

Проект: CanvasCloneDemo Файл: CanvasClone.cs (фрагмент)

```
protected override Size ArrangeOverride(Size finalSize)
{
    foreach (UIElement child in Children)
        child.Arrange(new Rect(
            new Point(GetLeft(child), GetTop(child)), child.DesiredSize));

    return base.ArrangeOverride(finalSize);
}
```

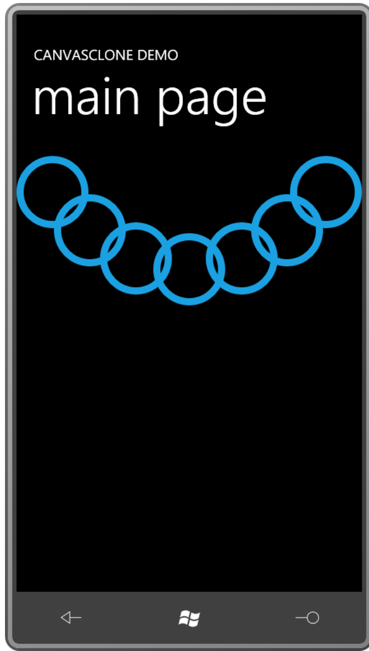
ArrangeOverride вызывает собственные статические методы *GetLeft* и *GetTop* для каждого дочернего элемента, чтобы определить, куда должен быть перемещен дочерний элемент относительно самого себя. Размер каждого дочернего элемента – это просто *DesiredSize*, который был вычислен при пересчете размеров.

XAML-файл в CanvasCloneDemo полностью аналогичен XAML-файлу проекта *EllipseChain*, только *Canvas* заменен на *CanvasClone*:

Проект: CanvasCloneDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1">
    <local:CanvasClone>
        <local:CanvasClone.Resources>
            <Style x:Key="ellipseStyle"
                   TargetType="Ellipse">
                <Setter Property="Width" Value="100" />
                <Setter Property="Height" Value="100" />
                <Setter Property="Stroke" Value="{StaticResource PhoneAccentBrush}"
/>
                <Setter Property="StrokeThickness" Value="10" />
            </Style>
        </local:CanvasClone.Resources>

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="0" local:CanvasClone.Top="0" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="52" local:CanvasClone.Top="53" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="116" local:CanvasClone.Top="92" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="190" local:CanvasClone.Top="107" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="263" local:CanvasClone.Top="92" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="326" local:CanvasClone.Top="53" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 local:CanvasClone.Left="380" local:CanvasClone.Top="0" />
    </local:CanvasClone>
</Grid>
```

С восторгом мы обнаруживаем на экране изображение, абсолютно аналогичное получаемому при выполнении первого приложения:


CANVASCLONE DEMO
main page

Глава 12

Привязка данных

Предположим, требуется, чтобы текущее значение *Slider*, перемещаемого пользователем, отображалось в *TextBlock*, как это было в приложении ColorScroll. Пара пустяков. Просто установим обработчик события *ValueChanged* объекта *Slider* и при каждом вызове обработчика будем получать значение свойства *Value* этого *Slider*, преобразовывать его в строку и задавать эту строку в качестве значения свойства *Text* элемента *TextBlock*.

Подобные задачи настолько широко распространены, что Silverlight обеспечивает элегантный механизм для их реализации. Он известен как *привязка данных* или просто *привязка*. Привязка данных – это связь, устанавливаемая между двумя свойствами двух объектов, благодаря которой при изменении свойства одного объекта связанное с ним свойство другого объекта обновляется его значением. Привязки также могут быть двунаправленными. В этом случае изменение любого из связанных свойств ведет к изменению другого.

Реализация привязки данных вполне ожидаема. Устанавливается обработчик событий, который обеспечивает обновление одного свойства значением другого с применением возможного преобразования данных. Обычно привязка данных описывается полностью в XAML, т.е. для нее не нужен вообще никакой код.

Проще всего продемонстрировать привязку данных на примере двух элементов, таких как *Slider* и *TextBlock*. С этого мы и начнем. Но намного большая мощь привязки данных обнаруживается при связывании визуальных элементов с источниками данных.

Цель этой главы – избежать применения явных обработчиков событий в файлах выделенного кода (только в примере в конце главы я был просто вынужден использовать их). Конечно, часто для поддержки привязки данных в XAML приходится написать *немного* кода, но этот код правильнее будет классифицировать как бизнес-объекты, а не элементы пользовательского интерфейса.

Источник и цель

В обычной привязке данных свойство одного объекта обновляется автоматически значением свойства другого объекта. Объект, предоставляющий данные – *Slider*, к примеру – называют *источником* привязки данных. Объект, принимающий данные – такой как *TextBlock* – это *цель* привязки.

Источнику привязки данных обычно дают имя:

```
<Slider Name="slider" … />
```

Целевое свойство можно вынести как свойство-элемент и задать в качестве его значения объект типа *Binding* (Привязка):

```
<TextBlock … >
    <TextBlock.Text>
        <Binding ElementName="slider" Path="Value" />
    </TextBlock.Text>
</TextBlock>
```

В свойстве *ElementName* (Имя элемента) указывается имя элемента-источника; в свойстве *Path* (Путь) – имя свойства-источника, каковым в данном случае является свойство *Value*

объекта *Slider*. Такой тип привязки иногда называют привязка к *элементу*, потому что источником привязки является визуальный элемент, ссылка на который выполняется по имени.

Чтобы сделать синтаксис немного более понятным, Silverlight предоставляет расширение разметки *Binding*, в котором вся привязка описывается в пределах фигурных скобок. (Это одно из расширений разметки Silverlight for Windows Phone. Мы уже познакомились с расширением *StaticResource* в главе 7 и в главе 16 рассмотрим *TemplateBinding* (Привязка шаблона).) Получаем более лаконичный синтаксис:

```
<TextBlock … Text="{Binding ElementName=slider, Path=Value}" … />
```

Обратите внимание, что параметры *ElementName* и *Path* разделены запятой, и что имена *slider* и *Value* больше не заключены в кавычки. Кавычки никогда не используются в фигурных скобках расширения разметки.

Приложение SliderBindings (Привязки ползунка) включает эту привязку и позволяет немного поэкспериментировать. Все описывается в XAML-файле:

Проект Silverlight: SliderBindings Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Slider Name="slider"
            Value="90"
            Grid.Row="0"
            Maximum="180"
            Margin="24" />

    <TextBlock Name="txtblk"
               Text="{Binding ElementName=slider, Path=Value}"
               Grid.Row="1"
               FontSize="48"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <Rectangle Grid.Row="2"
               Width="{Binding ElementName=slider, Path=Value}"
               RenderTransformOrigin="0.5 0.5"
               Fill="Blue">
        <Rectangle.RenderTransform>
            <RotateTransform x:Name="rotate"
                             Angle="90" />
        </Rectangle.RenderTransform>
    </Rectangle>
</Grid>
```

На странице располагается объект *Slider* с диапазоном значений от 0 до 180, объект *TextBlock*, свойство *Text* которого связано со свойством *Value* объекта *Slider*, и *Rectangle*, свойство *Width* которого связано все с тем же свойством *Value*. Для *Rectangle* также задано *RotateTransform*, что обеспечивает поворот элемента на 90°, которые заданы как константа.

При перемещении ползунка *TextBlock* выводит на экран его текущее значение, и соответствующим образом меняется высота *Rectangle*. (Целью *Binding* является свойство *Width* объекта *Rectangle*, но *Rectangle* повернут на 90°.)

Порядок перечисления свойств в расширении разметки *Binding* не имеет значения. Свойство *Path* можно поставить первым:

```
<TextBlock … Text="{Binding Path=Value, ElementName=slider}"
```

Кстати, если *Path* идет первым, можно опустить часть «Path=» и просто использовать имя свойства:

```
<TextBlock … Text="{Binding Value, ElementName=slider}"
```

Далее в этой и последующих главах я буду использовать такой сокращенный синтаксис, но мне не нравится применять его для привязок к элементу, потому что теряется понимание того, как работает привязка. Классу *Binding* надо *прежде всего* найти элемент с именем *slider* в дереве визуальных элементов и *после этого* с помощью технологии отражения найти свойство *Value* в этом элементе. Я предпочитаю синтаксис, в котором порядок свойств повторяет последовательность операций процесса:

```
<TextBlock … Text="{Binding ElementName=slider, Path=Value}"
```

Почему это свойство класса *Binding* называется *Path*, а не *Property*? В конце концов, у класса *Style* есть свойство *Property*. В чем же дело с *Binding*?

Ответ прост – значение *Path* может быть составлено из имен нескольких свойств. Например, у *Slider* нет имени. Если известно, что этот *Slider* является первым элементом коллекции *Children* элемента *ContentPanel*, на него можно сослаться косвенно следующим образом:

```
Text="{Binding ElementName=ContentPanel, Path=Children[0].Value}"
```

Или перемещаясь вверх по дереву визуальных элементов:

```
Text="{Binding ElementName=LayoutRoot, Path=Children[1].Children[0].Value}"
```

Составляющие компоненты пути должны быть свойствами или индексами. Между ними ставятся точки.

Цель и режим

У привязок есть источник и цель. Цель привязки – это свойство, для которого задается привязка. Это свойство всегда должно быть продублировано свойством-зависимостью. Всегда, без каких-либо исключений. Это ограничение особенно очевидно, когда привязка создается в коде.

Чтобы попробовать это в *SliderBindings*, удалим привязку, заданную для свойства *Text* элемента *TextBlock*. В файле MainPage.xaml.cs с помощью директивы *using* необходимо подключить пространство имен *System.Windows.Data*, в котором располагается класс *Binding*. В конструкторе после вызова *InitializeComponent* создадим объект типа *Binding* и зададим его свойства:

```
Binding binding = new Binding();
binding.ElementName = "slider";
binding.Path = new PropertyPath("Value");
```

Свойства *ElementName* и *Path* ссылаются на источник привязки. Но посмотрите на код, описывающий свойство *Text* элемента *TextBlock* как цель привязки:

```
txtblk.SetBinding(TextBlock.TextProperty, binding);
```

Метод *SetBinding* описывается классом *FrameworkElement*, и его первый аргумент является свойством-зависимостью. Это и есть целевое свойство. Цель – это элемент, для которого вызывается *SetBinding*. Или как альтернативный вариант привязку для цели можно задать, используя статический метод *BindingOperations.SetBinding*:

```
BindingOperations.SetBinding(txtblk, TextBlock.TextProperty, binding);
```

Но нам по-прежнему необходимо свойство-зависимость. Это еще одна причина, почему свойства визуальных объектов должны быть зависимыми свойствами. Это позволяет не только применять к этим свойствам стили и назначать их целями анимаций, но также использовать их в качестве целей привязок данных.

С точки зрения приоритетности свойств зависимостей привязки приравнены к локальным параметрам.

Метод *BindingOperations.SetBinding* подразумевает возможность задания привязки для *любого* свойства-зависимости. Но это не так в Silverlight for Windows Phone. Целью привязки может быть только свойство класса *FrameworkElement*.

Например, можно заметить, что свойству *RenderTransform* элемента *Rectangle* в MainPage.xaml задан *RotateTransform*. Попробуем задать для свойства *Angle* ту же привязку, что и для свойства *Text* объекта *TextBlock* и свойства *Width* объекта *Rectangle*:

```
<RotateTransform x:Name="rotate"
                 Angle="{Binding ElementName=slider, Path=Value}" />
```

Все выглядит нормально, но не работает. Во время выполнения будет сформировано исключение *XamlParseException*. *Angle* продублировано свойством-зависимостью, все правильно, а вот *RotateTransform* не наследуется от *FrameworkElement*, поэтому не может быть целью привязки. (Если задать *Binding* для свойства *Angle* класса *RotateTransform*, в Silverlight 4 все будет работать, но Silverlight for Windows Phone – это преимущественно Silverlight 3)

Давайте поэкспериментируем и удалим эту привязку для свойства *Angle RotateTransform* и любой код, который мог быть добавлен в MainPage.xaml.cs. Исходное значение свойства *Value* объекта *Slider* – 90:

```
<Slider Name="slider"
        Value="90" … />
```

Целью привязки является свойство *Text* объекта *TextBlock*:

```
<TextBlock Name="txtblk"
           Text="{Binding ElementName=slider, Path=Value}" … />
```

Теперь изменим ситуацию на противоположную. Зададим для свойства *Text* объекта *TextBlock* исходное значение 90:

```
<TextBlock Name="txtblk"
           Text="90" … />
```

И сделаем целью привязки свойство *Value* объекта *Slider*:

```
<Slider Name="slider"
        Value="{Binding ElementName=txtblk, Path=Text}" … />
```

Сначала кажется, что все работает. При запуске приложения ползунок *Slider* располагается в центре шкалы, указывая на значение 90, полученное от *TextBlock*, и размер *Rectangle* тоже все еще связан со *Slider*. Но при перемещении ползунка прямоугольник меняет высоту, а вот выводимое *TextBlock* значение остается неизменным. Объект *Binding*, заданный для *Slider*, ожидает изменения свойства *Text* объекта *TextBlock*, а оно остается неизменным.

Теперь добавим в привязку для *Slider* параметр *Mode*, чтобы обозначить двунаправленную привязку:

```
<Slider Name="slider"
        Value="{Binding ElementName=txtblk, Path=Text, Mode=TwoWay}" … />
```

Заработало! Целью привязки по-прежнему считается свойство *Value* объекта *Slider*. Любые изменения свойства *Text* объекта *TextBlock* отражаются в свойстве *Value* объекта *Slider*, но теперь и изменения *Slider* также приводят к изменению *TextBlock*.

В качестве значений свойства *Mode* (Режим) используются члены перечисления *BindingMode* (Режим привязки). Значением по умолчанию является *BindingMode.OneWay* (Однонаправленный). Кроме него имеется еще два значения: *BindingMode.TwoWay* (Двунаправленный) и *BindingMode.OneTime* (Однократно), обеспечивающий передачу данных от источника к цели только один раз.

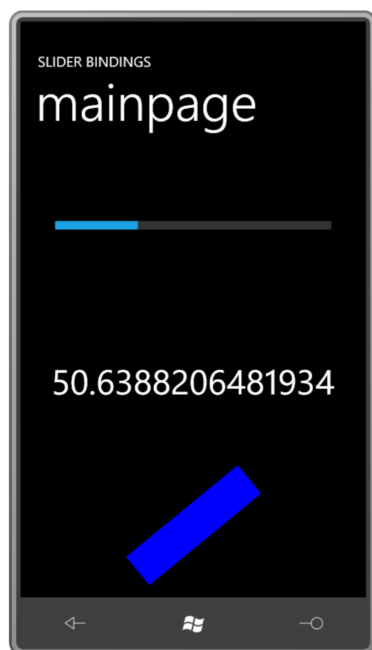
Применяя эту же технику, можно установить привязку к свойству *Angle* объекта *RotateTransform*. Сначала вернем в *TextBlock* исходную привязку:

```
<TextBlock Name="txtblk"
           Text="{Binding ElementName=slider, Path=Value}" … />
```

Теперь зададим для *Slider* двунаправленную привязку к свойству *Angle* объекта *RotateTransform*:

```
<Slider Name="slider"
        Value="{Binding ElementName=rotate, Path=Angle, Mode=TwoWay}" … />
```

И это работает! Элемент *Rectangle* поворачивается при перемещении *Slider*:



Конвертеры привязок

Экспериментируя с приложением SliderBindings (или увидев снимок экрана выше), можно заметить, что *TextBlock* по-разному отображает значения *Slider*: то это целое число, то десятичная дробь с одним или несколькими знаками после запятой, но чаще всего это числа с полной выкладкой 15 разрядов после запятой, предусмотренных для значений с плавающей точкой двойной точности.

Можно ли каким-то образом исправить это?

Да, безусловно. В классе *Binding* есть свойство *Converter* (Конвертер). Это свойство ссылается на класс, который преобразовывает данные по пути от источника к цели и (если необходимо) в обратном направлении. Очевидно, что некоторое неявное преобразование данных

выполняется в любом случае, поскольку числа преобразовываются в строки и строки преобразовываются в числа. Но у нас есть возможность обеспечить более явную поддержку этим преобразованиям.

Свойство *Converter* класса *Binding* типа *IValueConverter* (Конвертер значений). Это интерфейс, которому необходимы лишь два метода: *Convert* (Преобразовать) и *ConvertBack* (Преобразовать в обратном направлении). Метод *Convert* обрабатывает данные на пути от источника к целевому объекту, и метод *ConvertBack* обеспечивает преобразование в обратном направлении для двунаправленных привязок.

Если класс конвертера не предполагается использовать с двунаправленными привязками, просто возвращайте *null* из *ConvertBack*.

Чтобы добавить простой конвертер в SliderBindings, введем в проект новый класс и назовем его *TruncationConverter* (Конвертер для усечения разрядов). На самом деле этот класс уже есть в проекте, рассмотрим его:

Проект Silverlight: SliderBindings Файл: TruncationConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace SliderBindings
{
    public class TruncationConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            if (value is double)
                return Math.Round((double)value);

            return value;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return value;
        }
    }
}
```

Аргумент value метода *Convert* – это объект, передаваемый от источника к цели. Этот метод просто проверяет, является ли он типа *double*. Если да, он явно приводит его к *double* для метода *Math.Round*.

MainPage.xaml должен ссылаться на этот класс, т.е. мы должны объявить пространство имен XML:

```
xmlns:local="clr-namespace:SliderBindings"
```

После этого класс *TruncationConverter* задается как ресурс:

```
<phone:PhoneApplicationPage.Resources>
    <local:TruncationConverter x:Key="truncate" />
    …
</phone:PhoneApplicationPage.Resources>
```

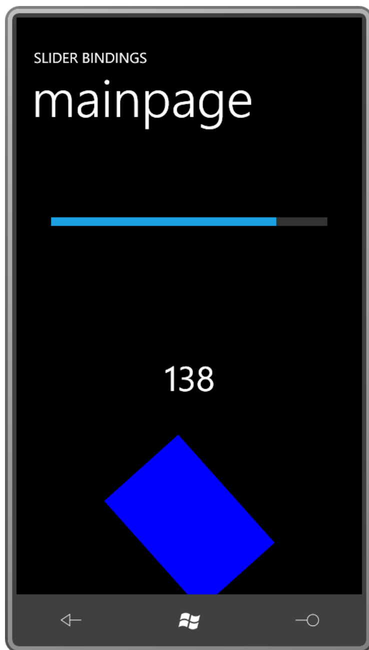
Все эти дополнения уже сделаны в файле MainPage.xaml проекта SliderBindings.

После этого расширение разметки *Binding* ссылается на этот ресурс:

```
<TextBlock Name="txtblk"
           Text="{Binding ElementName=slider,
                          Path=Value,
                          Converter={StaticResource truncate}}" … />
```

Я разнес расширение разметки на три строки, чтобы все три компонента было отчетливо видно. Обратите внимание, что *StaticResource*, другое расширение разметки, встроено в первое расширение разметки, поэтому в конце этого выражения стоят две фигурные скобки.

Теперь число отображаемых знаков при выводе значений в *TextBlock* ограничено:



Не забывайте указывать конвертер как *StaticResource*. Часто так и хочется просто присвоить имя ключа в качестве значения свойства *Converter* привязки:

```
<!-- Это неверно! -->
<TextBlock Name="txtblk"
           Text="{Binding ElementName=slider,
                          Path=Value,
                          Converter=truncate}" … />
```

Я до сих пор нередко допускаю такую ошибку, а потом выискивать причины возникшей проблемы очень сложно.

Описание конвертера как ресурса является, несомненно, самым распространенным подходом при использовании конвертеров, но не единственным. Применяя синтаксис свойство-элемент для *Binding*, класс *TrunctionConverter* можно встроить непосредственно в разметку:

```
<TextBlock … >
    <TextBlock.Text>
        <Binding ElementName="slider"
                 Path="Value">
            <Binding.Converter>
                <local:TruncationConverter />
            </Binding.Converter>
        </Binding>
    </TextBlock.Text>
</TextBlock>
```

Но если XAML-файл ссылается на один и тот же конвертер многократно, предпочтительнее определить его как ресурс, это обеспечивает возможность совместного использования одного его экземпляра.

На самом деле *TrucationConverter* – ужасный конвертер. Несомненно, он выполняет свою задачу, но при этом не отличается гибкостью. Вы собираетесь вызывать метод *Math.Round* в классе конвертера, но не было бы лучше иметь опцию для округления значений до определенного количества знаков после запятой? А не имело бы больше смысла обеспечить все виды форматирования: не только для чисел, но и для других типов данных?

Такие чудеса обеспечивает класс из библиотеки Petzold.Phone.Silverlight под именем *StringFormatConverter* (Конвертер с форматированием строк):

Проект Silverlight: Petzold.Phone.Silverlight Файл: StringFormatConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Petzold.Phone.Silverlight
{
    public class StringFormatConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            if (targetType == typeof(string) && parameter is string)
                return String.Format(parameter as string, value);

            return value;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return value;
        }
    }
}
```

Кроме свойства *Converter*, в классе *Binding* также имеется свойство *ConverterParameter* (Параметр преобразования). Значение этого свойства передается в вызов метода *Convert* как аргумент *parameter* (параметр). Здесь метод *Convert* принимает аргумент *parameter* как стандартную строку форматирования .NET, которая может использоваться в вызове *String.Format*.

Чтобы использовать этот конвертер в приложении SliderBindings, необходимо указать ссылку на библиотеку Petzold.Phone.Silverlight. (Это уже сделано.) Также в наш файл уже добавлено объявление пространства имен XML:

```
xmlns:petzold="clr-
namespace:Petzold.Phone.Silverlight;assembly=Petzold.Phone.Silverlight"
```

Создадим экземпляр класса *StringFormatConverter* в коллекции *Resources* страницы:

```
<phone:PhoneApplicationPage.Resources>
    …
    <petzold:StringFormatConverter x:Key="stringFormat" />
</phone:PhoneApplicationPage.Resources>
```

Теперь этот конвертер можно использовать в расширении разметки *Binding*. В качестве значения *ConverterParameter* зададим строку форматирования .NET с одним полем подстановки:

```
Text="{Binding ElementName=slider,
               Path=Value,
               Converter={StaticResource stringFormat},
               ConverterParameter=…}"
```

Но при вводе строки форматирования .NET сразу же выявляется проблема. В стандартной строке форматирования .NET используются фигурные скобки. Можно с уверенностью сказать, что синтаксический анализатор XAML при разборе расширения разметки *Binding* не будет рад встроенным фигурным скобкам.

Простейшее решение – заключить значение *ConverterParameter* в одинарные кавычки:

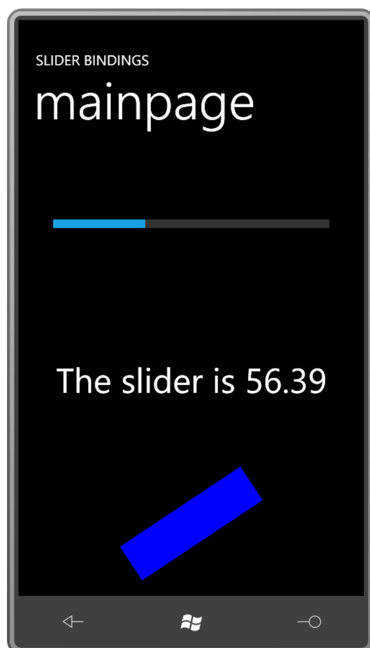
```
Text="{Binding ElementName=slider,
               Path=Value,
               Converter={StaticResource stringFormat},
               ConverterParameter='{0:F2}'}"
```

Средство синтаксического анализа XAML и визуальный дизайнер в Visual Studio тоже не распознают этот синтаксис, но он не создает проблем во время выполнения. Чтобы дизайнер принял такую запись, вставьте пробел (или какой-то другой символ) после первой одинарной кавычки.

Мы знаем, что *ConverterParameter* используется как первый аргумент при вызове *String.Format*, поэтому можем немного приукрасить его:

```
Text="{Binding ElementName=slider,
               Path=Value,
               Converter={StaticResource stringFormat},
               ConverterParameter='The slider is {0:F2}'}"
```

И получаем вот такой результат:



Относительный источник

Silverlight for Windows Phone поддерживает три основных типа привязок, систематизированных по источнику данных. До сих пор в данной главе мы рассматривали привязки *ElementName*, в которых привязка ссылается на именованный элемент. Но далее для подключения источника данных мы будем использовать преимущественно свойство *Source*, а не *ElementName*.

Третий тип привязок называют *RelativeSource* (Относительный источник). В Windows Presentation Foundation *RelativeSource* намного более гибок, чем его версия в Silverlight, которая может не произвести должного впечатления. Одно из применений *RelativeSource* связано с шаблонами и будет рассмотрено в главе 16. Кроме этого, он позволяет определять привязку, ссылающуюся на свойство того же элемента, обозначаемого как *Self* (Сам объект). Синтаксис продемонстрирован в следующем приложении:

Проект Silverlight: BindToSelf Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">

        <TextBlock Text="{Binding RelativeSource={RelativeSource Self},
                                  Path=FontFamily}" />

        <TextBlock Text=" - " />

        <TextBlock Text="{Binding RelativeSource={RelativeSource Self},
                                  Path=FontSize}" />

        <TextBlock Text=" pixels" />
    </StackPanel>
</Grid>
```

В качестве значения свойства *RelativeSource* задается другое расширение разметки, включающее *RelativeSource* и *Self*. *Path* ссылается на другое свойство того же элемента. Таким образом, элементы *TextBlock* отображают собственные значения *FontFamily* и *FontSize*.

Источник «this»

Предположим, поставлена задача вывести на экран множество коротких строк текста, каждую из которых должна быть заключена в рамку. Принято решение создать для этого элемент управления *BorderedText* (Текст в рамке), унаследовав его от *UserControl*, и описать его следующим образом:

```
<petzold:BorderedText Text="Ta Da!"
                      FontFamily="Times New Roman"
                      FontSize="96"
                      FontStyle="Italic"
                      FontWeight="Bold"
                      TextDecorations="Underline"
                      Foreground="Red"
                      Background="Lime"
                      BorderBrush="Blue"
                      BorderThickness="8"
                      CornerRadius="36"
                      Padding="16 4"
                      HorizontalAlignment="Center"
                      VerticalAlignment="Center" />
```

Как видно из префикса пространства имен XML, этот класс уже входит в состав библиотеки Petzold.Phone.Silverlight.

BorderedText наследуется от *UserControl*, и *UserControl* наследуется от *Control*. Таким образом, нам известно, что *BorderedText* уже будет иметь некоторые из этих свойств посредством наследования классов. В самом *BorderedText* должны быть описаны свойства *Text*, *TextDecorations*, *CornerRadius* и, возможно, еще пара других, что обеспечит классу большую гибкость.

Весьма вероятно, что дерево визуальных элементов файла BorderedText.xaml будет состоять из *TextBlock*, расположенного в *Border*. Все многочисленные свойства *TextBlock* и этого *Border* должны задаваться из свойств *BorderedText*.

Один из способов сделать это был представлен в предыдущей главе. В том случае в классе *ColorColumn* описывались свойства *Label* и *Value*. Задание новых значений этим свойствам в элементах дерева визуальных элементов осуществлялось с помощью обработчиков событий изменения значения свойства. Намного проще сделать это с помощью привязки данных.

В файле выделенного кода для *BorderedText* просто описываются все свойства, которые недоступны посредством наследования от *Control*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: BorderedText.xaml.cs

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace Petzold.Phone.Silverlight
{
    public partial class BorderedText : UserControl
    {
        public static readonly DependencyProperty TextProperty =
            DependencyProperty.Register("Text",
                typeof(string),
                typeof(BorderedText),
                new PropertyMetadata(null));

        public static readonly DependencyProperty TextAlignmentProperty =
            DependencyProperty.Register("TextAlignment",
                typeof(TextAlignment),
                typeof(BorderedText),
                new PropertyMetadata(TextAlignment.Left));

        public static readonly DependencyProperty TextDecorationsProperty =
            DependencyProperty.Register("TextDecorations",
                typeof(TextDecorationCollection),
                typeof(BorderedText),
                new PropertyMetadata(null));

        public static readonly DependencyProperty TextWrappingProperty =
            DependencyProperty.Register("TextWrapping",
                typeof(TextWrapping),
                typeof(BorderedText),
                new PropertyMetadata(TextWrapping.NoWrap));

        public static readonly DependencyProperty CornerRadiusProperty =
            DependencyProperty.Register("CornerRadius",
                typeof(CornerRadius),
                typeof(BorderedText),
                new PropertyMetadata(new CornerRadius()));

        public BorderedText()
        {
```

```
            InitializeComponent();
        }

        public string Text
        {
            set { SetValue(TextProperty, value); }
            get { return (string)GetValue(TextProperty); }
        }

        public TextAlignment TextAlignment
        {
            set { SetValue(TextAlignmentProperty, value); }
            get { return (TextAlignment)GetValue(TextAlignmentProperty); }
        }

        public TextDecorationCollection TextDecorations
        {
            set { SetValue(TextDecorationsProperty, value); }
            get { return
(TextDecorationCollection)GetValue(TextDecorationsProperty); }
        }

        public TextWrapping TextWrapping
        {
            set { SetValue(TextWrappingProperty, value); }
            get { return (TextWrapping)GetValue(TextWrappingProperty); }
        }

        public CornerRadius CornerRadius
        {
            set { SetValue(CornerRadiusProperty, value); }
            get { return (CornerRadius)GetValue(CornerRadiusProperty); }
        }
    }
}
```

Много кода, но ничего сложного, ведь это всего лишь описания свойств. Нет никаких обработчиков событий изменения значений свойств. Рассмотрим XAML-файл с *Border* и *TextBlock*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: BorderedText.xaml

```
<UserControl x:Class="Petzold.Phone.Silverlight.BorderedText"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             Name="this">

    <Border Background="{Binding ElementName=this, Path=Background}"
            BorderBrush="{Binding ElementName=this, Path=BorderBrush}"
            BorderThickness="{Binding ElementName=this, Path=BorderThickness}"
            CornerRadius="{Binding ElementName=this, Path=CornerRadius}"
            Padding="{Binding ElementName=this, Path=Padding}">

        <TextBlock Text="{Binding ElementName=this, Path=Text}"
                   TextAlignment="{Binding ElementName=this, Path=TextAlignment}"
                   TextDecorations="{Binding ElementName=this,
Path=TextDecorations}"
                   TextWrapping="{Binding ElementName=this, Path=TextWrapping}" />
    </Border>
</UserControl>
```

Обратите внимание, что у корневого элемента есть имя:

```
Name="this"
```

Этому корневому элементу можно задать любое имя, но традиционно используется ключевое слово C# *this*, потому что в контексте XAML-файла *this* ссылается на текущий экземпляр класса *BorderedText*. Это хорошо знакомая концепция. Наличие этого имени означает, что есть возможность связывать свойства *BorderedText* и свойства элементов, образующих дерево визуальных элементов.

В данном файле для свойства *Foreground* или остальных свойств для задания шрифта не требуется использовать привязки данных, потому что они наследуются по дереву визуальных элементов. Единственное свойство *TextBlock*, об утрате которого в этом элементе управления я пожалел – *Inlines*, но *TextBlock* описывает это свойство как свойство только для чтения, поэтому мы не можем задавать привязку для него.

Приложение BorderedTextDemo тестирует наш новый элемент управления:

Проект Silverlight: BorderedTextDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <petzold:BorderedText Text="Ta Da!"
                          FontFamily="Times New Roman"
                          FontSize="96"
                          FontStyle="Italic"
                          FontWeight="Bold"
                          TextDecorations="Underline"
                          Foreground="Red"
                          Background="Lime"
                          BorderBrush="Blue"
                          BorderThickness="8"
                          CornerRadius="36"
                          Padding="16 4"
                          HorizontalAlignment="Center"
                          VerticalAlignment="Center" />
</Grid>
```

Механизмы уведомления

Чтобы привязки данных работали, источник данных должен реализовывать некоторый *механизм уведомления*. Этот механизм уведомления сигнализирует об изменении значения свойства, сообщая о том, что новое значение может быть извлечено из источника и передано в цель. При связывании свойства *Value* объекта *Slider* и свойства *Text* объекта *TextBlock* мы имеем дело с двумя свойствами-зависимостями. Этого нельзя заметить в открытых программных интерфейсах, но свойства-зависимости обеспечивают именно такой механизм уведомления.

Безусловно, соединять два визуальных элемента с помощью привязки данных удобно, но в большинстве случаев целевым объектом привязки данных является визуальный элемент, а вот источник нет. Чаще всего это сущность, которую принято называть *бизнес-объектом*.

И теперь внимание!

Иногда, когда разработчики знакомятся с новой и важной возможностью операционной системы – такой как свойства-зависимости, которые обсуждались в предыдущей главе – им хочется применить эту возможность везде, вероятно, просто чтобы опробовать ее и поупражняться. Но со свойствами-зависимостями это не очень хорошая идея. Несомненно, свойства-зависимости должны использоваться при наследовании от классов, производных от *DependencyObject*, но, вероятно, не следует наследоваться от *DependencyObject* исключительно для того, чтобы применить свойства-зависимости.

Иначе говоря, не бросайтесь переписывать свои бизнес-объекты ради использования свойств-зависимостей в них!

Целями привязок данных должны быть свойства-зависимости, но к источникам привязок это требование *не* предъявляется. Источниками привязок могут быть старые добрые свойства старых добрых классов. Но источник должен реализовывать некоторый механизм уведомления об изменении собственного значения, чтобы обеспечить обновление значения цели.

Практически всегда в бизнес-объектах, выступающих в роли источников привязок, механизм уведомления реализовывается через интерфейс *INotifyPropertyChanged* (Уведомление об изменении свойства). *INotifyPropertyChanged* описан в пространстве имен *System.ComponentModel* (Компонентная модель). Это явный признак того, что данный интерфейс не ограничен лишь рамками Silverlight и играет очень важную роль в .NET. Он является средством, с помощью которого бизнес-объекты уведомляют об изменении своих данных.

При этом *INotifyPropertyChanged* исключительно прост. Он описывается следующим образом:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged:
}
```

Класс может реализовать *INotifyPropertyChanged*, просто описав открытое событие *PropertyChanged* (Значение свойства изменилось). В сущности, это событие классу абсолютно *ни к чему*, но правила приличия требуют, чтобы класс формировал это событие при каждом изменении значения одного из его свойств.

Делегат *PropertyChangedEventHandler* (Обработчик события изменения значения свойства) ассоциирован с классом *PropertyChangedEventArgs* (Аргументы события изменения значения свойства), у которого имеется одно единственное открытое свойство только для чтения *PropertyName* типа string. Имя свойства, значение которого изменилось, передается в конструктор *PropertyChangedEventArgs*.

Иногда в классе, реализующем *INotifyPropertyChanged*, можно увидеть защищенный виртуальный метод *OnPropertyChanged* (При изменении значения свойства) с аргументом типа *PropertyChangedEventArgs*. Применение данного метода не является обязательным, но удобно для производных классов. Этот метод является замечательным местом для формирования события изменения значения свойства, и я использую его в своих примерах.

Бизнес-объекты, которые реализуют *INotifyPropertyChanged*, не наследуются от *FrameworkElement*, т.е. они не являются частью дерева визуальных элементов XAML-файла. Обычно их экземпляры создаются как XAML-ресурсы или в файле выделенного кода.

Простой сервер привязки

Иногда я рассматривают бизнес-объекты, на которые предполагается ссылаться в XAML-файлах посредством привязок, как *серверы привязок*. Они предоставляют открытые свойства и формируют события *PropertyChanged* при изменении значений этих свойств.

Например, в приложении для Windows Phone 7 должно отображаться текущее время, при этом требуется обеспечить довольно большую гибкость в части выводимых на экран данных, например, показывать только секунды. Вы хотите реализовать все это полностью в XAML. Скажем, вывод будет реализован в виде фразы «Текущее время в секундах – », за которой

следует число, изменяющееся каждую секунду. Технику, которую я продемонстрирую здесь, можно распространить на многие другие типы приложений, не только для реализации часов.

Даже несмотря на то что мы собираемся реализовывать все визуальные элементы в XAML, нам понадобиться некоторый код, возможно, класс с простым именем *Clock* и свойствами *Year*, *Month*, *Day*, *DayOfWeek* (День недели), *Hour*, *Minute* и *Second*. Экземпляр этого класса будем создавать в XAML-файле и выполнять доступ к свойствам посредством привязок данных.

Как известно, в .NET уже есть готовая структура со свойствами *Year*, *Month*, *Day* и т.д. Это *DateTime*. И хотя *DateTime* необходим для написания класса *Clock*, он не удовлетворяет всем целям, потому что свойства класса *DateTime* не меняются динамически. Каждый объект *DateTime* представляет конкретные фиксированные дату и время. Для сравнения, класс *Clock*, который будет продемонстрирован, имеет свойства, изменяющиеся для отражения текущего значения времени. Об этих изменениях класс *Clock* будет уведомлять с помощью события *PropertyChanged*.

Класс *Clock* находится в библиотеке Petzold.Phone.Silverlight. Рассмотрим его:

Проект Silverlight: Petzold.Phone.Silverlight Файл: Clock.cs

```
using System;
using System.ComponentModel;
using System.Windows.Threading;

namespace Petzold.Phone.Silverlight
{
    public class Clock : INotifyPropertyChanged
    {
        int hour, min, sec;
        DateTime date;

        public event PropertyChangedEventHandler PropertyChanged;

        public Clock()
        {
            OnTimerTick(null, null);

            DispatcherTimer tmr = new DispatcherTimer();
            tmr.Interval = TimeSpan.FromSeconds(0.1);
            tmr.Tick += OnTimerTick;
            tmr.Start();
        }

        public int Hour
        {
            protected set
            {
                if (value != hour)
                {
                    hour = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Hour"));
                }
            }
            get
            {
                return hour;
            }
        }

        public int Minute
```

```
        {
            protected set
            {
                if (value != min)
                {
                    min = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Minute"));
                }
            }
            get
            {
                return min;
            }
        }

        public int Second
        {
            protected set
            {
                if (value != sec)
                {
                    sec = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Second"));
                }
            }
            get
            {
                return sec;
            }
        }

        public DateTime Date
        {
            protected set
            {
                if (value != date)
                {
                    date = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Date"));
                }
            }
            get
            {
                return date;
            }
        }

        protected virtual void OnPropertyChanged(PropertyChangedEventArgs args)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, args);
        }

        void OnTimerTick(object sender, EventArgs args)
        {
            DateTime dt = DateTime.Now;
            Hour = dt.Hour;
            Minute = dt.Minute;
            Second = dt.Second;
            Date = DateTime.Today;
        }
    }
}
```

Класс *Clock* реализует интерфейс *INotifyPropertyChanged* и, следовательно, включает открытое событие *PropertyChanged*. Ближе к концу кода класса можно также увидеть защищенный метод *OnPropertyChanged*, который фактически формирует событие. Конструктор класса

задает обработчик события *Tick* класса *DispatcherTimer*, которое формируется с интервалом в 1/10 секунды. Обработчик *OnTimerTick* (в самом конце класса) задает новые значения свойствам *Hour*, *Minute*, *Second* и *Date*, которые имеют очень схожую структуру.

Например, посмотрим на свойство *Hour*:

```
public int Hour
{
    protected set
    {
        if (value != hour)
        {
            hour = value;
            OnPropertyChanged(new PropertyChangedEventArgs("Hour"));
        }
    }
    get
    {
        return hour;
    }
}
```

Метод доступа *set* является защищенным. Значение задается только внутри приложения, и мы не хотим, чтобы внешние классы вмешивались в это. Метод доступа *set* сравнивает значение, задаваемое свойству, и значение, хранящееся как поле. Если они не равны, он задает полю *hour* (час) новое значение и вызывает метод *OnPropertyChanged* для формирования события.

Некоторые разработчики не включают выражение *if* для проверки того, действительно ли новое значение свойства отличается от его текущего значения. В результате событие *PropertyChanged* формируется при любом задании свойства, даже если его значение не меняется. Это неправильно, особенно для таких классов, как этот. Нам на самом деле не нужно, чтобы событие *PropertyChanged* оповещало об изменении свойства *Hour* каждую 1/10 секунды, если значение фактически меняется только раз в час.

Чтобы использовать класс *Clock* в XAML-файле, необходимо включить в него ссылку на библиотеку Petzold.Phone.Silverlight и объявление пространства имен XML:

```
xmlns:petzold="clr-
namespace:Petzold.Phone.Silverlight;assembly=Petzold.Phone.Silverlight"
```

Если источник привязки не наследуется от *DependencyObject*, в *Binding* используется не *ElementName*, а *Source*. Привязки, которые мы хотим создать, в качестве значения *Source* задают объект *Clock*, описанный в библиотеке Petzold.Phone.Silverlight.

При использовании синтаксиса свойство-элемент ссылку на класс *Clock* можно вставить непосредственно в *Binding*:

```
<TextBlock>
    <TextBlock.Text>
        <Binding Path="Second">
            <Binding.Source>
                <petzold:Clock />
            </Binding.Source>
        </Binding>
    </TextBlock.Text>
</TextBlock>
```

Свойство *Source* объекта *Binding* вынесено как свойство-элемент, и в качестве его значения задан экземпляр класса *Clock*. Свойство *Path* указывает на свойство *Second* класса *Clock*.

Или, что еще более удобно, *Clock* можно определить как XAML-ресурс:

```
<phone:PhoneApplicationPage.Resources>
    <petzold:Clock x:Key="clock" />
    …
</phone:PhoneApplicationPage.Resources>
```

Тогда расширение разметки *Binding* может ссылаться на этот ресурс:

```
TextBlock Text="{Binding Source={StaticResource clock}, Path=Second}" />
```

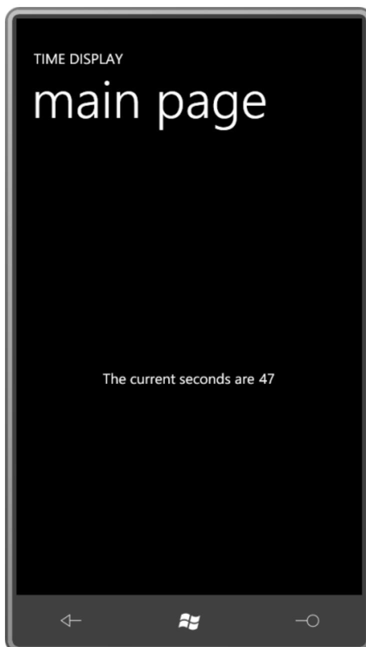
Обратите внимание на встроенное расширение разметки для *StaticResource*.

Этот подход продемонстрирован в проекте TimeDisplay, в котором для конкатенации текста используется горизонтальный *StackPanel*:

Проект Silverlight: TimeDisplay Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="The current seconds are " />
        <TextBlock Text="{Binding Source={StaticResource clock},
                                  Path=Second}" />
    </StackPanel>
</Grid>
```

И вот что получаем:



Обращаю внимание еще раз: цель привязки (свойство *Text* объекта *TextBlock*) должно быть свойством-зависимостью. Это обязательное условие. Чтобы целевой объект обновлялся с изменением значений объекта-источника привязки (свойство *Second* объекта *Clock*), источник должен реализовывать некоторый механизм уведомления, что он и делает.

Конечно, мне не нужен *StackPanel* с множеством элементов *TextBlock*. Применяя *StringFormatConverter* (который я включил в TimeDisplay как ресурс с ключом «stringFormat», чтобы можно было поэкспериментировать с ним), я просто вставляю весь текст следующим образом:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Text="{Binding Source={StaticResource clock},
                              Path=Second,
                              Converter={StaticResource stringFormat},
                              ConverterParameter='The current seconds are {0}'}" />
</Grid>
```

Теперь расширение разметки *Binding* включает *два* встроенных расширения разметки.

Если требуется выводить на экран значения нескольких свойств класса *Clock*, придется вернуться к применению множества элементов *TextBlock*. Например, такая разметка обеспечит вывод значений времени с разделением часов, минут и секунд двоеточиями и добавлением нулей перед значениями минут и секунд:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="{Binding Source={StaticResource clock},
                                  Path=Hour}" />
        <TextBlock Text="{Binding Source={StaticResource clock},
                                  Path=Minute,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
        <TextBlock Text="{Binding Source={StaticResource clock},
                                  Path=Second,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
    </StackPanel>
</Grid>
```

Как видите, все три привязки включают один и тот же параметр *Source*. Можно ли каким-то образом избежать повторений? Да, можно, и эта техника также иллюстрирует очень важную концепцию.

Задание *DataContext*

Класс *FrameworkElement* описывает свойство *DataContext* (Контекст данных), в качестве значения которого может использоваться практически любой объект (в коде), но обычно это привязка (в XAML). *DataContext* – одно из тех свойств, которое передается вниз по дереву визуальных элементов и может сочетаться с локальными привязками. Как минимум *DataContext* позволяет упростить отдельные привязки, устраняя повторения. В более широком понимании DataContext – это средство для связывания данных с деревьями визуальных элементов.

В этом конкретном примере в качестве значения свойства *DataContext* можно задать любой элемент, являющийся родителем элементов *TextBlock*. Возьмем ближайшего родителя, каковым является *StackPanel*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{Binding Source={StaticResource clock}}"
                Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="{Binding Path=Hour}" />
        <TextBlock Text="{Binding Path=Minute,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
        <TextBlock Text="{Binding Path=Second,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
```

```
    </StackPanel>
</Grid>
```

Теперь в качестве значения *DataContext* свойства *DataContext* класса *StackPanel* задан элемент *Binding*, который просто ссылается на объект-источник привязки: ресурс *Clock*. Всем потомкам *StackPanel* нет необходимости явно ссылаться на этот *Source*, он включен в привязки каждого отдельного элемента *TextBlock*.

В качестве значения *DataContext* можно задать объект *Binding*, как это сделал я:

```
DataContext="{Binding Source={StaticResource clock}}"
```

Или в данном случае для *DataContext* задается источник напрямую:

```
DataContext="{StaticResource clock}"
```

Любой из вариантов допустим, и оба можно найти в моих примерах.

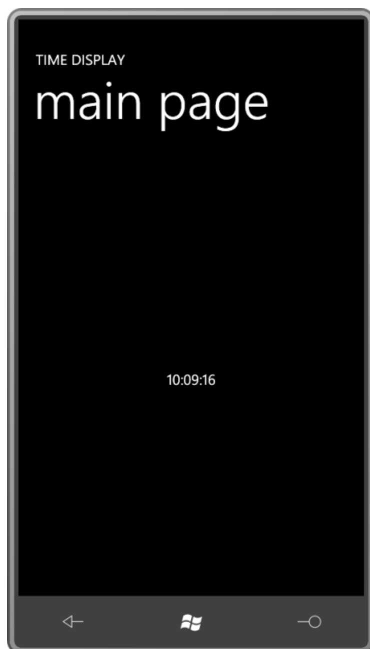
Когда мы убрали свойство *Source* из отдельных расширений *Binding*, кажется, более естественным будет удалить из них и часть «Path=»:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{Binding Source={StaticResource clock}}"
                Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="{Binding Hour}" />
        <TextBlock Text="{Binding Minute,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
        <TextBlock Text="{Binding Second,
                                  Converter={StaticResource stringFormat},
                                  ConverterParameter=':{0:D2}'}" />
    </StackPanel>
</Grid>
```

Помните, что часть «Path=» расширения разметки *Binding* может быть удалена, только если *Path* является первым элементом. Теперь кажется, что каждая привязка ссылается на определенное свойство *DataContext*:

```
<TextBlock Text="{Binding Hour}" />
```

Вот что мы получаем на экране:



DataContext исключительно полезен, если страница или элемент управления должны отображать свойства одного конкретного класса. В коде можно задать, чтобы *DataContext* переключался между различными экземплярами этого класса.

Хотя, конечно, это не так распространено, но *DataContext* может использоваться с привязками *ElementName*. Вспомним дерево визуальных элементов файла BorderText.xaml, которое мы видели ранее:

```
<Border Background="{Binding ElementName=this, Path=Background}"
        BorderBrush="{Binding ElementName=this, Path=BorderBrush}"
        BorderThickness="{Binding ElementName=this, Path=BorderThickness}"
        CornerRadius="{Binding ElementName=this, Path=CornerRadius}"
        Padding="{Binding ElementName=this, Path=Padding}">

    <TextBlock Text="{Binding ElementName=this, Path=Text}"
               TextAlignment="{Binding ElementName=this, Path=TextAlignment}"
               TextDecorations="{Binding ElementName=this, Path=TextDecorations}"
               TextWrapping="{Binding ElementName=this, Path=TextWrapping}" />
</Border>
```

Для *DataContext* объекта *Border* можно задать *Binding* с использованием *ElementName*. Это существенно упростит все остальные привязки:

```
<Border DataContext="{Binding ElementName=this}"
        Background="{Binding Background}"
        BorderBrush="{Binding BorderBrush}"
        BorderThickness="{Binding BorderThickness}"
        CornerRadius="{Binding CornerRadius}"
        Padding="{Binding Padding}">

    <TextBlock Text="{Binding Path=Text}"
               TextAlignment="{Binding Path=TextAlignment}"
               TextDecorations="{Binding Path=TextDecorations}"
               TextWrapping="{Binding Path=TextWrapping}" />
</Border>
```

Вернемся к *Clock*. Я немного поленился при написании класса и не определил многие свойства компонентов даты, такие как *Month* (Месяц) и *Year* (Год). Вместо этого я просто использовал свойство *Date* типа *DateTime*. Обработчик *OnTimerTick* задает в качестве значения этого свойства статическое свойство *DateTime.Today* (Сегодня). *DateTime.Today* – это объект *DateTime*, значение времени которого соответствует полуночи. То есть данное

свойство *Date* не формирует события *PropertyChanged* каждую 1/10 секунды. Оно формирует одно событие при запуске приложения и затем каждый раз при наступлении полуночи.

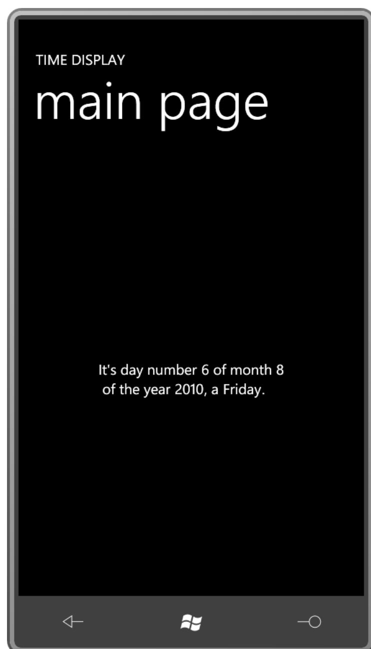
Обратиться к отдельным свойствам свойства *Date* можно следующим образом:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="It's day number " />
            <TextBlock Text="{Binding Source={StaticResource clock},
                                      Path=Date.Day}" />
            <TextBlock Text=" of month " />
            <TextBlock Text="{Binding Source={StaticResource clock},
                                      Path=Date.Month}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text=" of the year " />
            <TextBlock Text="{Binding Source={StaticResource clock},
                                      Path=Date.Year}" />
            <TextBlock Text=", a " />
            <TextBlock Text="{Binding Source={StaticResource clock},
                                      Path=Date.DayOfWeek}" />
            < TextBlock Text="." />
        </StackPanel>
    </StackPanel>
</Grid>
```

Либо можно задать *DataContext* для *StackPanel*, как раньше, и удалить часть «Path=» из привязок:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{StaticResource clock}"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="It's day number " />
            <TextBlock Text="{Binding Date.Day}" />
            <TextBlock Text=" of month " />
            <TextBlock Text="{Binding Date.Month}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text=" of the year " />
            <TextBlock Text="{Binding Date.Year}" />
            <TextBlock Text=", a " />
            <TextBlock Text="{Binding Date.DayOfWeek}" />
            <TextBlock Text="." />
        </StackPanel>
    </StackPanel>
</Grid>
```

Любой из вариантов обеспечивает вывод на экран двух строк текста:



Date – это свойство класса *Clock* типа *DateTime*; *Day*, *Month*, *Year* и *DayOfWeek* – свойства *DateTime*. В них не выполняется никакого форматирования, кроме обеспечиваемого вызовами *ToString* (К строке) по умолчанию. Значения свойств *Day*, *Month* и *Year* отображаются как числа. Значениями свойства *DayOfWeek* являются члены перечисления *DayOfWeek*, поэтому на экран будет выводиться текст, например Wednesday (Среда), но он не будет отвечать локализации приложения. Члены *DayOfWeek* соответствуют английским названиям дней недели, они и будут отображаться.

Также можно задать *DataContext* так, чтобы оно ссылалось и на свойство *Source*, и на свойство *Date*. Тогда отдельные привязки будут просто ссылаться на свойства *DateTime*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{Binding Source={StaticResource clock},
                                      Path=Date}"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="It's day number " />
            <TextBlock Text="{Binding Day}" />
            <TextBlock Text=" of month " />
            <TextBlock Text="{Binding Month}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text=" of the year " />
            <TextBlock Text="{Binding Year}" />
            <TextBlock Text=", a " />
            <TextBlock Text="{Binding DayOfWeek}" />
            <TextBlock Text="." />
        </StackPanel>
    </StackPanel>
</Grid>
```

Несомненно, существует множество вариантов форматирования дат, которые документируются классом *DateTimeFormatInfo* (Сведения о форматировании даты и времени) из пространства имен *System.Globalization*. Также можно использовать *StringFormatConverter*.

Предположим, требуется вставить имя текущего месяца где-то в середине абзаца, выводимого на экран с помощью *TextBlock*. При использовании *TextBlock* для отображения абзаца текста, свойству *TextWrapping* обычно задают значение *Wrap*. Но в данном случае

StackPanel не может использоваться для конкатенации множества элементов *TextBlock*. Здесь придется включить весь текст в один *TextBlock*, имя месяца в том числе. Как это сделать?

Считайте себя гением, если вспомнили о классе *Run*. В конце главы 8 мы рассматривали, как класс *Run* наследуется от *Inline* и позволяет задавать форматирование для отдельного фрагмента текста в рамках *TextBlock*. У класса *Run* есть свойство *Text*, так что он кажется идеальным средством для встраивания имени месяца (или другой привязки) в абзац:

```
<!-- Это не будет работать! -->
<TextBlock TextWrapping="Wrap">
    Здесь располагается какой-то длинный текст. В него необходимо вставить имя месяца
    <Run Text="{Binding Source={StaticResource clock},
                        Path=Date,
                        Converter={StaticResource stringFormat},
                        ConverterParameter='{0:MMMM}'}" />
    и затем продолжить повествование.
</TextBlock>
```

Это именно то, что нам надо. Единственная проблема – такой вариант не работает! Ничего не получится, потому что свойство *Text* класса *Run* не продублировано свойством-зависимостью, а целевые объекты привязок данных обязаны быть свойствами-зависимостями всегда.

Кажется несправедливым, что у класса *Run* есть эта небольшая проблема, но объектные структуры во многом похожи на жизнь, а жизнь не всегда справедлива.

На сегодняшний день эту задачу нельзя реализовать только в XAML. Объекту *Run* придется присвоить имя и задать его свойство *Text* из кода.

Простые решения

XAML не является настоящим языком программирования. Он не включает ничего похожего на выражения *if*. XAML не может принимать решения.

Но это не означает, что мы не будем пытаться это сделать.

Как можно было заметить, класс *Clock* использовал обычное свойство *Hour* класса *DateTime*, значением которого является показание времени в 24-часовом формате. Что делать, если мы хотим использовать 12-часовой формат и выводить рядом с показаниями времени текст «AM» или «PM» для обозначения первой или второй половины суток.

Обычно это делается путем форматирования времени (если класс *Clock* действительно предоставил объект *DateTime*, отражающий время). Однако предположим, мы хотим иметь большую гибкость в отображении данных AM и PM – возможно, выводить текст «утра» или «вечера» – и мы хотим делать это в XAML.

Рассмотрим новый класс *TwelveHourClock* (Время в 12-часовом формате), который наследуется от *Clock*.

Проект Silverlight: Petzold.Phone.Silverlight   Файл: TwelveHourClock.cs

```
using System;
using System.ComponentModel;

namespace Petzold.Phone.Silverlight
{
    public class TwelveHourClock : Clock
    {
        int hour12;
```

```
        bool isam, ispm;

        public int Hour12
        {
            protected set
            {
                if (value != hour12)
                {
                    hour12 = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Hour12"));
                }
            }
            get
            {
                return hour12;
            }
        }

        public bool IsAm
        {
            protected set
            {
                if (value != isam)
                {
                    isam = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("IsAm"));
                }
            }
            get
            {
                return isam;
            }
        }

        public bool IsPm
        {
            protected set
            {
                if (value != ispm)
                {
                    ispm = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("IsPm"));
                }
            }
            get
            {
                return ispm;
            }
        }

        protected override void OnPropertyChanged(PropertyChangedEventArgs args)
        {
            if (args.PropertyName == "Hour")
            {
                Hour12 = (Hour - 1) % 12 + 1;
                IsAm = Hour < 12;
                IsPm = !IsAm;
            }

            base.OnPropertyChanged(args);
        }
    }
}
```

Класс *TwelveHourClock* описывает три свойства: *Hour12* и два свойства типа Boolean, *IsAm* и *IsPm*. Каждое из них формирует событие *PropertyChanged*. Перегруженный метод *OnPropertyChanged* проверяет, является измененное свойство свойством *Hour*. Если да, он

вычисляет новые значения для всех трех свойств, что само по себе опять приводит к вызову *OnPropertyChanged*.

isAm является просто логическим отрицанием *isPM*. Отсюда возникает справедливый вопрос, чем обусловлена необходимость в наличии двух свойств. Это объясняется просто: XAML не может самостоятельно выполнить операцию логического отрицания, поэтому и нужны два свойства.

Создадим экземпляр класса *TwelveHourClock* в коллекции *Resources* и присвоим ему ключ «clock12»:

```
<phone:PhoneApplicationPage.Resources>
    <petzold:TwelveHourClock x:Key="clock12" />
</phone:PhoneApplicationPage.Resources>
```

Чтобы с помощью XAML вывести на экран примерно такой текст: «It's after 9 in the morning[1]» - мы сделали бы следующее:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{StaticResource clock12}"
                Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="It's after " />
        <TextBlock Text="{Binding Hour}" />
        <TextBlock Text=" in the morning." />
        <TextBlock Text=" in the afternoon." />
    </StackPanel>
</Grid>
```

В данном XAML предлагаются две отдельные строки для утра и дня, но на экран должна выводиться только одна из них в зависимости от того, какое из свойств, *IsAm* или *IsPm*, имеет значение true. Как вообще это можно реализовать?

Необходим еще один конвертер. Его имя *BooleanToVisibilityConverter* (Конвертер логического значения в значение видимости), и он будет использоваться довольно часто. Данный конвертер предполагает, что значение источника типа Boolean, а целевой объект – свойство типа *Visibility*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: BooleanToVisibilityConverter.cs

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace Petzold.Phone.Silverlight
{
    public class BooleanToVisibilityConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return (bool)value ? Visibility.Visible : Visibility.Collapsed;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return (Visibility)value == Visibility.Visible;
        }
```

[1] Начало десятого утра (*прим. переводчика*).

```
    }
}
```

Добавим этот класс в коллекцию *Resources*:

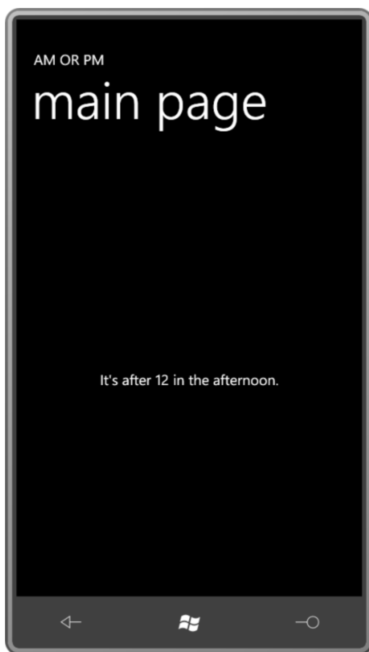
```
<phone:PhoneApplicationPage.Resources>
    <petzold:TwelveHourClock x:Key="clock12" />
    <petzold:BooleanToVisibilityConverter x:Key="booleanToVisibility" />
</phone:PhoneApplicationPage.Resources>
```

Теперь используя *BooleanToVisibilityConverter*, свяжем посредством привязки свойства *Visibility* последних двух элементов *TextBlock* со свойствами *IsAm* и *IsPm*. Рассмотрим разметку из проекта AmOrPm:

Проект Silverlight: AmOrPm Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{StaticResource clock12}"
                Orientation="Horizontal"
                HorizontalAlignment="Center"
                VerticalAlignment="Center">
        <TextBlock Text="It's after " />
        <TextBlock Text="{Binding Hour}" />
        <TextBlock Text=" in the morning."
                   Visibility="{Binding IsAm,
                                Converter={StaticResource booleanToVisibility}}"
/>
        <TextBlock Text=" in the afternoon."
                   Visibility="{Binding IsPm,
                                Converter={StaticResource
booleanToVisibility}}"/>
    </StackPanel>
</Grid>
```

И это работает:



Конвертеры со свойствами

Нет ничего безрассудного в создании конвертера привязки данных, настолько специализированного или причудливого, что он будет иметь очень узкое применение. Например, рассмотрим класс *DecimalBitToBrushConverter* (Конвертер десятичного разряда в кисть). Этот конветер включает два открытых свойства: *ZeroBitBrush* (Кисть нулевого двоичного разряда) и *OneBitBrush* (Кисть единичного двоичного разряда).

Проект Silverlight: BinaryClock Файл: DecimalBitToBrushConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;
using System.Windows.Media;

namespace BinaryClock
{
    public class DecimalBitToBrushConverter : IValueConverter
    {
        public Brush ZeroBitBrush { set; get; }
        public Brush OneBitBrush { set; get; }

        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            int number = (int)value;
            int bit = Int32.Parse(parameter as string);
            int digit = number / PowerOfTen(bit / 4) % 10;

            return ((digit & (1 << (bit % 4))) == 0) ? ZeroBitBrush : OneBitBrush;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return null;
        }

        int PowerOfTen(int exp)
        {
            int value = 1;

            for (int i = 0; i < exp; i++)
                value *= 10;

            return value;
        }
    }
}
```

Метод *Convert* ожидает аргумент *value* типа *int* и действительный аргумент *parameter*. Если свойству *ConverterParameter* в XAML задано строковое значение, оно будет передано в метод *Convert* как объект типа *string*. Вот здесь потребуется преобразовать его в требуемый тип вручную. (Чтобы переопределить это поведение, необходимо использовать для *ConverterParameter* синтаксис свойство-элемент и задавать тип с помощью тегов элемента.) Данный метод *Convert* ожидает, что передаваемая в него строка представляет другой тип *int*, поэтому передает ее в метод *Int32.Parse*.

Аргумент *value* имеет числовое значение, например, 127. Аргумент *parameter*, преобразованный в *int*, представляет двоичный разряд, например, 6. По сути метод разбивает передаваемое в него число на десятичные разряды (в данном примере это 1, 2 и

7) и затем выбирает разряд, соответствующий заданному двоичному разряду. 7 в числе 127 соответствует двоичным разрядам от 0 до 3; 2 в числе 127 соответствует двоичным разрядам от 4 до 7; 1 в числе 127 соответствует двоичным разрядам от 8 до 11.

Если в данном двоичном разряде хранится 1, *Convert* возвращает *OneBitBrush*; если 0, *Convert* возвращает *ZeroBitBrush*.

Я использую этот конвертер в проекте BinaryClock (Двоичные часы). На него ссылается производный от *UserControl* класс *BinaryNumberRow* (Строка двоичного числа). Обратите внимание, два открытых свойства *DecimalBitToBrushConverter* задаются прямо в коллекции *Resources*, которая также включает *Style* для *Ellipse*.

Проект Silverlight: BinaryClock Файл: BinaryNumberRow.xaml

```
<UserControl
  x:Class="BinaryClock.BinaryNumberRow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:petzold="clr-
namespace:Petzold.Phone.Silverlight;assembly=Petzold.Phone.Silverlight"
  xmlns:local="clr-namespace:BinaryClock">

    <UserControl.Resources>
        <Style x:Key="ellipseStyle" TargetType="Ellipse">
            <Setter Property="Width" Value="48" />
            <Setter Property="Height" Value="48" />
            <Setter Property="Stroke" Value="{StaticResource PhoneForegroundBrush}"
/>
            <Setter Property="StrokeThickness" Value="2" />
        </Style>

        <local:DecimalBitToBrushConverter x:Key="converter"
                                          ZeroBitBrush="{x:Null}"
                                          OneBitBrush="Red" />
    </UserControl.Resources>

    <petzold:UniformStack Orientation="Horizontal">
        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=6}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=5}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=4}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Stroke="{x:Null}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=3}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=2}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
                 Fill="{Binding Converter={StaticResource converter},
                               ConverterParameter=1}" />

        <Ellipse Style="{StaticResource ellipseStyle}"
```

```
                 Fill="{Binding Converter={StaticResource converter},
                                ConverterParameter=0}" />
    </petzold:UniformStack>
</UserControl>
```

Тело дерева визуальных элементов *BinaryNumberRow* образует горизонтально ориентированный *UniformStack*, содержащий семь элементов *Ellipse*. У каждого из них есть *Binding*, задающий в качестве значения свойства *Converter* класс *DecimalBitToBrushConverter*, и *ConverterParameter*, значения которого меняются от 0 для крайнего справа *Ellipse* до 6 для крайнего слева *Ellipse*. Ни одна из привязок не включает параметров *Source* или *Path*! Очевидно, что они задаются где-то в другом месте в *DataContext* для экземпляра *BinaryNumberRow*.

В файле MainPage.xaml проекта BinaryClock экземпляр объекта *TwelveHourClock* создается в разделе *Resources*:

Проект Silverlight: BinaryClock Файл: MainPage.xaml

```
<phone:PhoneApplicationPage.Resources>
    <petzold:TwelveHourClock x:Key="clock12" />
</phone:PhoneApplicationPage.Resources>
```

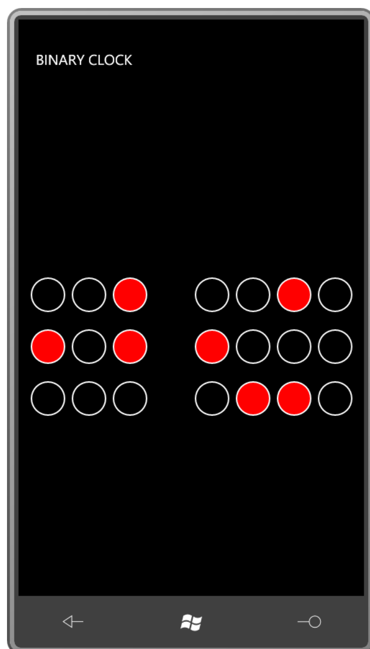
Область содержимого включает центрированный по вертикали *StackPanel* с тремя экземплярами *BinaryNumberRow*:

Проект Silverlight: BinaryClock Файл: MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

    <StackPanel DataContext="{StaticResource clock12}"
                VerticalAlignment="Center">

        <local:BinaryNumberRow DataContext="{Binding Hour12}"
                               Margin="0 12" />

        <local:BinaryNumberRow DataContext="{Binding Minute}"
                               Margin="0 12" />

        <local:BinaryNumberRow DataContext="{Binding Second}"
                               Margin="0 12" />
    </StackPanel>
</Grid>
```

Обратите внимание на параметры *DataContext*. В качестве значения свойства *DataContext* объекта *StackPanel* задан сам *TwelveHourClock*. *DataContext* каждого элемента управления задано одно из свойств *TwelveHourClock*. Вот поэтому в описаниях *Binding* класса *BinaryNumberRow* должны присутствовать только *Converter* и *ConverterParameter*.

В результате получаем, конечно же, двоичные часы:



И время, хммм, 12:58:06.

А можно ли еще больше сократить разметку *Binding*? Предположим, для всех элементов *Ellipse* задан один и тот же *Converter*. Можно ли и этот параметр перенести в описание *DataContext* для *StackPanel*? Нет, нельзя. Параметры *Converter* и *ConverterParameter* должны располагаться вместе в одном описании *Binding*.

Передача и прием

Два сервиса привязки, которые мы обсудили до сих пор, просто предоставляют некоторые данные. В XAML можно также создавать привязки, передающие данные в сервис привязки и возвращающие некоторый результат. В качестве очень простой демонстрации рассмотрим сервис привязки, осуществляющий такую важную операцию, как сложение двух чисел. Я назвал его *Adder* (Сумматор).

Проект Silverlight: Petzold.Phone.Silverlight Файл: Adder.cs

```
using System.ComponentModel;

namespace Petzold.Phone.Silverlight
{
    public class Adder : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        double augend = 0;
        double addend = 0;
        double sum = 0;

        public double Augend
        {
            set
            {
                if (augend != value)
                {
                    augend = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Augend"));
                    CalculateNewSum();
```

```
                }
            }
            get
            {
                return augend;
            }
        }

        public double Addend
        {
            set
            {
                if (addend != value)
                {
                    addend = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Addend"));
                    CalculateNewSum();
                }
            }
            get
            {
                return addend;
            }
        }

        public double Sum
        {
            protected set
            {
                if (sum != value)
                {
                    sum = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Sum"));
                }
            }

            get
            {
                return sum;
            }
        }

        void CalculateNewSum()
        {
            Sum = Augend + Addend;
        }

        protected virtual void OnPropertyChanged(PropertyChangedEventArgs args)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, args);
        }
    }
}
```

Участники операции сложения – это первое слагаемое (*Augend*) и второе слагаемое (*Addend*). Так и назовем наши свойства. Оба свойства типа *double* и оба совершенно открытые. Когда любому из них задается новое значение, оно формирует событие *PropertyChanged* и также вызывает метод *CalculateNewSum* (Вычислить новую сумму).

Метод *CalculateNewSum* выполняет сложение значений свойств *Augend* и *Addend* и полученный результат задает как значение свойства *Sum* (Сумма). Свойство *Sum* несколько отличается от двух других, потому что его метод *set* является защищенным, так что *Sum* не может быть задан извне этого класса. Но так оно и должно быть.

Проект *SliderSum* (Сумма значений ползунков) демонстрирует один из вариантов использования этого сервиса привязки в приложении. Коллекция *Resources* ссылается на два файла из библиотеки Petzold.Phone.Silverlight:

Проект Silverlight: SliderSum Файл: MainPage.xaml (фрагмент)

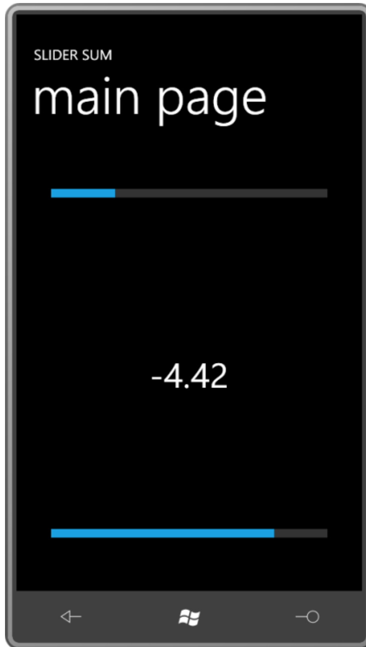
```
<phone:PhoneApplicationPage.Resources>
    <petzold:Adder x:Key="adder" />
    <petzold:StringFormatConverter x:Key="stringFormat" />
</phone:PhoneApplicationPage.Resources>
```

В области содержимого располагаются два элемента *Slider*, сверху и снизу, и довольно большую площадь занимает *TextBlock*:

Проект Silverlight: SliderSum Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource adder}}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Slider Grid.Row="0"
            Minimum="-100"
            Maximum="100"
            Margin="24"
            Value="{Binding Augend, Mode=TwoWay}" />

    <Slider Grid.Row="2"
            Minimum="-100"
            Maximum="100"
            Margin="24"
            Value="{Binding Addend, Mode=TwoWay}" />

    <TextBlock Grid.Row="1"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               FontSize="48"
               Text="{Binding Sum,
                              Converter={StaticResource stringFormat},
                              ConverterParameter=' {0:F2} '}" />
</Grid>
```

Обратите внимание на *DataContext* в *Grid*. Элементы управления *Slider* поставляют значения свойствам *Augend* и *Addend*, но эти свойства не могут быть целями привязки, потому что они не продублированы свойствами-зависимостями. *Adder* должен быть источником привязки, и элементы управления *Slider* должны быть целями привязки. Поэтому для обеих привязок задан двунаправленный режим. Изначально привязкам *Slider* заданы срединные значения диапазонов значений, заданных по умолчанию в классе *Adder*, но затем уже значения *Slider* передаются в свойства *Augend* и *Addend*. Привязка между *TextBlock* и свойством *Sum* обеспечивает также некоторое форматирование строки.



Предположим, требуется, чтобы отрицательные значения отображались красным. (Возможно, приложение используется для бухгалтеров.) Мы уже знаем, что для этого необходимо привлечь конвертер привязки. Обобщим его, включив сравнение передаваемого в конвертер значения с определенным критерием. Каждый из трех возможных вариантов (больше, равно или меньше) обеспечивает возвращение конвертером определенной кисти. Как и в конвертере проекта BinaryClock, эти данные могут предоставляться посредством открытых свойств класса конвертера, как здесь:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ValueToBrushConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;
using System.Windows.Media;

namespace Petzold.Phone.Silverlight
{
    public class ValueToBrushConverter : IValueConverter
    {
        public double Criterion { set; get; }
        public Brush GreaterThanBrush { get; set; }
        public Brush EqualToBrush { get; set; }
        public Brush LessThanBrush { get; set; }

        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            double doubleVal = (value as IConvertible).ToDouble(culture);
            return doubleVal >= Criterion ? doubleVal == Criterion ? EqualToBrush :

GreaterThanBrush :
                                                                     LessThanBrush;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return null;
        }
```

```
    }
}
```

Изначально этот конвертер просто приводил аргумент *value* к типу *double*. Потом я решил использовать его с другими числовыми типами данных, поэтому пришел к более универсальному преобразованию с помощью метода *IConvertible* (Поддающийся преобразованию).

В проекте SliderSumWithColor этот конвертер добавляется в неуклонно увеличивающуюся коллекцию *Resources*:

Проект Silverlight: SliderSumWithColor Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <petzold:Adder x:Key="adder" />
    <petzold:StringFormatConverter x:Key="stringFormat" />
    <petzold:ValueToBrushConverter x:Key="valueToBrush"
                        Criterion="0"
                        LessThanBrush="Red"
                        EqualToBrush="{StaticResource PhoneForegroundBrush}"
                        GreaterThanBrush="{StaticResource PhoneForegroundBrush}" />
</phone:PhoneApplicationPage.Resources>
```

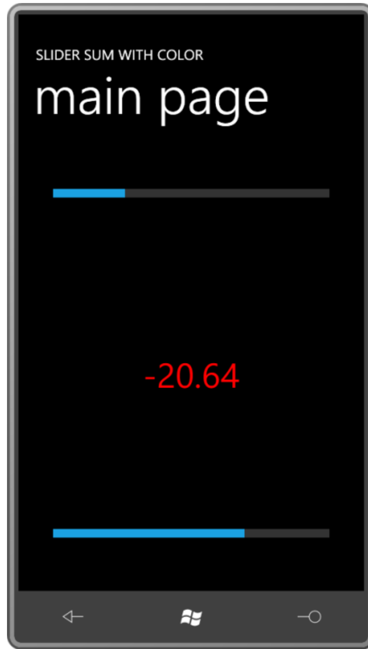
Значения свойств класса *ValueToBrushConverter* показывают, что значения меньше нуля будут отображаться красным, все остальные значения отображаются цветом *PhoneForegroundBrush* (Кисть фона телефона).

Все остальное аналогично предыдущему приложению, кроме *TextBlock*. Теперь для его свойства *Foreground* задана привязка *Binding* с конвертером *ValueToBrushConverter*:

Проект Silverlight: SliderSumWithColor Файл: MainPage.xaml (фрагмент)

```
<Grid … >
    …
    <TextBlock Grid.Row="1"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               FontSize="48"
               Text="{Binding Sum,
                              Converter={StaticResource stringFormat},
                              ConverterParameter=' {0:F2} '}"

               Foreground="{Binding Sum,
                                    Converter={StaticResource valueToBrush}}" />
</Grid>
```

Вообще довольно волнительно видеть, как значения, отображаемые в *TextBlock*, меняют цвет на красный, и знать при этом, что в приложении нет никакого явного обработчика событий, который обеспечивал бы эти изменения:



Обновления привязок *TextBox*

Свойство *Text* объекта *TextBox* может быть целью привязки данных, но при этом существуют потенциальные проблемы. Как только пользователь получает возможность вводить что-либо в *TextBox*, мы должны быть готовы к обработке ошибочного ввода.

Предположим, мы хотим написать приложение для решения квадратных уравнений, т.е. уравнений вида

$$Ax^2 + Bx + C = 0$$

Чтобы сделать приложение более универсальным, вероятно, необходимо предусмотреть три элемента управления *TextBox*, в которые пользователь сможет вводить значения *A*, *B* и *C*. Также включим кнопку *Button* с надписью «calculate» (Вычислить), по нажатию которой производится вычисление двух решений уравнения по стандартной формуле:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

После этого решения выводятся в *TextBlock*.

Но багаж знаний о привязках данных и пример сервера привязки *Adder* наводят на мысли о применении другого подхода. Мы можем оставить три элемента управления *TextBox* и использовать *TextBlock* для отображения результатов, но при этом связать все эти элементы управления со свойствами сервера привязки.

А куда делся *Button*? Похоже, в *Button* нет особой необходимости.

Для начала возьмем из библиотеки Petzold.Phone.Silverlight класс под именем *QuadraticEquationSolver* (Модуль для решения квадратных уравнений). Он реализует интерфейс *INotifyPropertyChanged*, включает три свойства, названные *A*, *B* и *C*, а также свойства только для чтения *Solution1* (Решение1) и *Solution2*. Есть еще два дополнительных свойства только для чтения типа *bool*: *HasTwoSolutions* (Имеет два решения) и *HasOneSolution* (Имеет одно решение).

Solution: Petzold.Phone.Silverlight Файл: QuadaticEquationSolver.cs

```
using System;
using System.ComponentModel;

namespace Petzold.Phone.Silverlight
{
    public class QuadraticEquationSolver : INotifyPropertyChanged
    {
        Complex solution1;
        Complex solution2;
        bool hasTwoSolutions;
        double a, b, c;

        public event PropertyChangedEventHandler PropertyChanged;

        public double A
        {
            set
            {
                if (a != value)
                {
                    a = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("A"));
                    CalculateNewSolutions();
                }
            }
            get
            {
                return a;
            }
        }

        public double B
        {
            set
            {
                if (b != value)
                {
                    b = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("B"));
                    CalculateNewSolutions();
                }
            }
            get
            {
                return b;
            }
        }

        public double C
        {
            set
            {
                if (c != value)
                {
                    c = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("C"));
                    CalculateNewSolutions();
                }
            }
            get
            {
                return c;
            }
        }

        public Complex Solution1
```

```
        {
            protected set
            {
                if (!solution1.Equals(value))
                {
                    solution1 = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Solution1"));
                }
            }

            get
            {
                return solution1;
            }
        }

        public Complex Solution2
        {
            protected set
            {
                if (!solution2.Equals(value))
                {
                    solution2 = value;
                    OnPropertyChanged(new PropertyChangedEventArgs("Solution2"));
                }
            }

            get
            {
                return solution2;
            }
        }

        public bool HasTwoSolutions
        {
            protected set
            {
                if (hasTwoSolutions != value)
                {
                    hasTwoSolutions = value;
                    OnPropertyChanged(new
PropertyChangedEventArgs("HasTwoSolutions"));
                    OnPropertyChanged(new
PropertyChangedEventArgs("HasOneSolution"));
                }
            }

            get
            {
                return hasTwoSolutions;
            }
        }

        public bool HasOneSolution
        {
            get
            {
                return !hasTwoSolutions;
            }
        }

        void CalculateNewSolutions()
        {
            if (A == 0 && B == 0 && C == 0)
            {
                Solution1 = new Complex(0, 0);
                HasTwoSolutions = false;
                return;
            }
```

```
            if (A == 0)
            {
                Solution1 = new Complex(-C / B, 0);
                HasTwoSolutions = false;
                return;
            }

            double discriminant = B * B - 4 * A * C;
            double denominator = 2 * A;
            double real = -B / denominator;
            double imaginary =
                Math.Sqrt(Math.Abs(discriminant)) / denominator;

            if (discriminant == 0)
            {
                Solution1 = new Complex(real, 0);
                HasTwoSolutions = false;
                return;
            }

            Solution1 = new Complex(real, imaginary);
            Solution2 = new Complex(real, -imaginary);
            HasTwoSolutions = true;
        }

        protected virtual void OnPropertyChanged(PropertyChangedEventArgs args)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, args);
        }
    }
}
```

Свойства *Solution1* и *Solution2* типа *Complex* (Комплексный). Эта структура также включена в проект *Petzold.Phone.Silverlight*, но не реализует ни одной операции. Структура *Complex* существует исключительно для обеспечения методов *ToString*. (Silverlight 4 включает класс *Complex* в пространстве имен *System.Numerics*, но он недоступен в Silverlight for Windows Phone 7.)

Проект Silverlight: Petzold.Phone.Silverlight Файл: Complex.cs

```
using System;

namespace Petzold.Phone.Silverlight
{
    public struct Complex : IFormattable
    {
        public double Real { get; set; }
        public double Imaginary { get; set; }

        public Complex(double real, double imaginary) : this()
        {
            Real = real;
            Imaginary = imaginary;
        }

        public override string ToString()
        {
            if (Imaginary == 0)
                return Real.ToString();

            return String.Format("{0} {1} {2}i",
                                 Real,
                                 Math.Sign(Imaginary) >= 1 ? "+" : "-",
```

```
                                 Math.Abs(Imaginary));
        }
        public string ToString(string format, IFormatProvider provider)
        {
            if (Imaginary == 0)
                return Real.ToString(format, provider);

            return String.Format(provider,
                                 "{0} {1} {2}i",
                                 Real.ToString(format, provider),
                                 Math.Sign(Imaginary) >= 1 ? "+" : "-",
                                 Math.Abs(Imaginary).ToString(format, provider));
        }

    }
}
```

Complex реализует интерфейс *IFormattable* (Поддающийся форматированию), т.е. у нее есть дополнительный метод *ToString*, включающий строку форматирования. Это необходимо, если мы собираемся использовать в методе *String.Format* спецификации форматирования числовых значений для форматирования комплексных чисел, как это делает *StringFormatConverter*.

Проект QuadraticEquations1 (Квадратные уравнения 1) – это первая попытка предоставления пользовательского интерфейса для класса *Complex*. Коллекция *Resources* класса *MainPage* включает ссылки на класс *QuadraticEquationSolver* и два конвертера, которые мы рассмотрели ранее:

Проект Silverlight: QuadraticEquations1 Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <petzold:QuadraticEquationSolver x:Key="solver" />
    <petzold:StringFormatConverter x:Key="stringFormat" />
    <petzold:BooleanToVisibilityConverter x:Key="booleanToVisibility" />
</phone:PhoneApplicationPage.Resources>
```

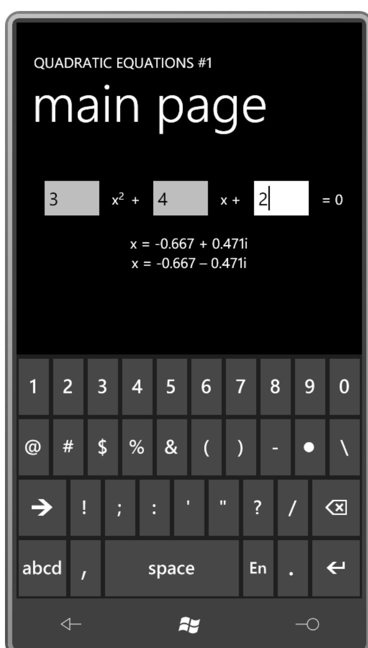
Область содержимого включает два вложенных элемента *StackPanel*. В горизонтальном *StackPanel* располагаются три элемента управления *TextBox* фиксированной ширины. Для них заданы двунаправленные привязки для ввода значений *A*, *B* и *C*. Обратите внимание, что свойству *InputScope* задано значение *Number*. Это обеспечит использование только цифровой клавиатуры.

Проект Silverlight: QuadraticEquations1 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{Binding Source={StaticResource solver}}">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center"
                    Margin="12">

            <TextBox Text="{Binding A, Mode=TwoWay}"
                     InputScope="Number"
                     Width="100" />

            <TextBlock Text=" x" VerticalAlignment="Center" />
            <TextBlock Text="2" VerticalAlignment="Center">
                <TextBlock.RenderTransform>
                    <ScaleTransform ScaleX="0.7" ScaleY="0.7" />
                </TextBlock.RenderTransform>
            </TextBlock>
```

```
            <TextBlock Text=" + " VerticalAlignment="Center" />

            <TextBox Text="{Binding B, Mode=TwoWay}"
                     InputScope="Number"
                     Width="100" />

            <TextBlock Text=" x + " VerticalAlignment="Center" />

            <TextBox Text="{Binding C, Mode=TwoWay}"
                     InputScope="Number"
                     Width="100" />

            <TextBlock Text=" = 0" VerticalAlignment="Center" />
        </StackPanel>

        <TextBlock Text="{Binding Solution1,
                               Converter={StaticResource stringFormat},
                               ConverterParameter='x = {0:F3}'}"
                   HorizontalAlignment="Center" />

        <TextBlock Text="{Binding Solution2,
                               Converter={StaticResource stringFormat},
                               ConverterParameter='x = {0:F3}'}"
                   Visibility="{Binding HasTwoSolutions,
                                     Converter={StaticResource
booleanToVisibility}}"
                   HorizontalAlignment="Center" />
    </StackPanel>
</Grid>
```

Два элемента *TextBlock*, которые описываются в конце этого фрагмента, выводят на экран два решения. Свойство *Visibility* второго *TextBlock* связано со свойством *HasTwoSolutions* класса *QuadraticEquationSolver*, поэтому второй *TextBlock* не будет видимым, если уравнение имеет только одно решение.



Наверное, первое, на что сразу обращаешь внимание – это то, что введение числа в *TextBox* не влияет на решения! Сначала кажется, что приложение вообще не работает. Значение передается в свойство *A*, *B* или *C* класса *QuadraticEquationSolver* только после того, как *TextBox* теряет фокус ввода.

Такое поведение реализовано намеренно. В общем случае элементы управления могут быть связаны с бизнес-объектами по сети и, вероятно, не было бы ничего хорошего, если бы *TextBox* обновлялся при каждом касании клавиши экранной клавиатуры. Пользователи делают множество ошибок и часто удаляют введенные символы. В некоторых случаях того времени, которое пользователь тратит на ввод, абсолютно достаточно, чтобы «передать» окончательное значение.

Но в данном конкретном приложении такое поведение не соответствует нашим требованиям. Чтобы изменить его, зададим свойству *UpdateSourceTrigger* (Триггер обновления источника) объекта *Binding* каждого из элементов управления *TextBox* значение *Explicit* (Явный):

```
<TextBox Text="{Binding A, Mode=TwoWay,
                          UpdateSourceTrigger=Explicit}"
         InputScope="Number"
         Width="100" />
```

Свойство *UpdateSourceTrigger* управляет тем обновлением источника (в данном случае это свойство *A*, *B* или *C* класса *QuadraticEquationSolver*) из цели (*TextBox*) при двунаправленной привязке данных. Значением этого свойства является член перечисления *UpdateSourceTrigger*. В WPF-версии *UpdateSourceTrigger* также доступны члены *LostFocus* (Потеря фокуса) и *PropertyChanged*, но в Silverlight есть только два варианта: *Default* и *Explicit*.

Default означает «поведение по умолчанию для целевого элемента управления», что в случае, когда целью является *TextBox*, значит обновление объекта-источника при потере фокуса объектом *TextBox*. Если задано значение *Explicit*, должен быть предоставлен некоторый код, который будет инициировать передачу данных от цели в источник. Эту роль может выполнять *Button* с надписью «calculate».

Если не хотите использовать *Button*, можно инициировать передачу при изменении текста в *TextBox*. В этом случае кроме задания свойства *UpdateSourceTrigger* для *Binding*, потребуется обеспечить обработчик события *TextChanged* объекта *TextBox*:

```
<TextBox Text="{Binding A, Mode=TwoWay,
                          UpdateSourceTrigger=Explicit}"
         InputScope="Number"
         Width="100"
         TextChanged="OnTextBoxTextChanged" />
```

В обработчике событий *TextChanged* необходимо «вручную» обновить источник, вызвав метод *UpdateSource* (Обновить источник), описанный классом *BindingExpression* (Выражение привязки).

Ранее в данной главе был продемонстрирован вызов метода *SetBinding*, описанного классом *FrameworkElement*, или статического метода *BindingOperations.SetBinding* для задания привязки для свойства в коде. (Метод *SetBinding*, описанный *FrameworkElement*, является сокращенным вариантом метода *BindingOperations.SetBinding*.) Оба метода возвращают объект типа *BindingExpression*.

Если вы не вызвали эти методы в коде, вам будет приятно узнать, что *FrameworkElement* сохраняет объект *BindingExpression*, и его можно извлечь с помощью открытого метода *GetBindingExpression* (Получить выражение привязки). В этот метод необходимо передать конкретное свойство, являющееся целью привязки данных, которое всегда, конечно же, будет свойством-зависимостью.

Рассмотрим код для обновления источника при изменении текста *TextBox*:

```
void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
{
```

```
    TextBox txtbox = sender as TextBox;
    BindingExpression bindingExpression =
txtbox.GetBindingExpression(TextBox.TextProperty);
    bindingExpression.UpdateSource();
}
```

Еще одна проблема с *TextBox* – пользователь может ввести строку символов, которая не может быть распознана как число. Хотя мы и не замечаем этого, но для задания свойств *A*, *B* или *C* класса *QuadraticEquationSolver* объект *string* из *TextBox* преобразовывается в *double* с помощью конвертера. Этот скрытый конвертер, вероятно, использует метод *Double.Parse* или *Double.TryParse*.

Можно перехватывать исключения, формируемые конвертером. Для этого понадобится задать значение *true* еще двум свойствам класса *Binding*, как показано в следующем фрагменте:

```
<TextBox Text="{Binding A, Mode=TwoWay,
                           UpdateSourceTrigger=Explicit,
                           ValidatesOnExceptions=True,
                           NotifyOnValidationError=True}"
         InputScope="Number"
         Width="100"
         TextChanged="OnTextBoxTextChanged" />
```

Это приводит к формированию события *BindingValidationError* (Ошибка валидации привязки). Это маршрутизируемое событие, поэтому оно может обрабатываться в любом элементе дерева визуальных элементов, располагающемся над *TextBox*. Удобнее всего в небольшом приложении задавать обработчик события прямо в конструкторе MainPage:

```
readonly Brush okBrush;
static readonly Brush errorBrush = new SolidColorBrush(Colors.Red);

public MainPage()
{
    InitializeComponent();
    okBrush = new TextBox().Foreground;
    BindingValidationError += OnBindingValidationError;
}
```

Заметьте, что обычная кисть *Foreground* для *TextBox* сохраняется как поле. Привожу простой обработчик события, который в случае недействительного ввода закрашивает текст *TextBox* красным:

```
void OnBindingValidationError(object sender, ValidationErrorEventArgs args)
{
    TextBox txtbox = args.OriginalSource as TextBox;
    txtbox.Foreground = errorBrush;
}
```

Конечно, цвет текста должен возвращаться к исходному сразу же при изменении текста *TextBox*. Это можно сделать в методе *OnTextBoxTextChanged* (При изменении текста текстового поля):

```
void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
{
    TextBox txtbox = sender as TextBox;
    txtbox.Foreground = okBrush;
    …
}
```

Объединим эти две техники – обновления при каждом нажатии клавиши и визуализации недействительного ввода – в проекте QuadraticEquations2. Рассмотрим XAML-файл:

Проект Silverlight: QuadraticEquations2 Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel DataContext="{Binding Source={StaticResource solver}}">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center"
                    Margin="12">

            <TextBox Text="{Binding A, Mode=TwoWay,
                                   UpdateSourceTrigger=Explicit,
                                   ValidatesOnExceptions=True,
                                   NotifyOnValidationError=True}"
                     InputScope="Number"
                     Width="100"
                     TextChanged="OnTextBoxTextChanged" />

            <TextBlock Text=" x" VerticalAlignment="Center" />
            <TextBlock Text="2" VerticalAlignment="Center">
                <TextBlock.RenderTransform>
                    <ScaleTransform ScaleX="0.7" ScaleY="0.7" />
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock Text=" + " VerticalAlignment="Center" />

            <TextBox Text="{Binding B, Mode=TwoWay,
                                   UpdateSourceTrigger=Explicit,
                                   ValidatesOnExceptions=True,
                                   NotifyOnValidationError=True}"
                     InputScope="Number"
                     Width="100"
                     TextChanged="OnTextBoxTextChanged" />

            <TextBlock Text=" x + " VerticalAlignment="Center" />

            <TextBox Text="{Binding C, Mode=TwoWay,
                                   UpdateSourceTrigger=Explicit,
                                   ValidatesOnExceptions=True,
                                   NotifyOnValidationError=True}"
                     InputScope="Number"
                     Width="100"
                     TextChanged="OnTextBoxTextChanged" />

            <TextBlock Text=" = 0" VerticalAlignment="Center" />
        </StackPanel>
        <StackPanel Name="result"
                    Orientation="Horizontal"
                    HorizontalAlignment="Center">

            <TextBlock Text="{Binding Solution1.Real,
                                      Converter={StaticResource stringFormat},
                                      ConverterParameter='x = {0:F3} '}" />
            <TextBlock Text="+"
                       Visibility="{Binding HasOneSolution,
                                    Converter={StaticResource
booleanToVisibility}}" />

            <TextBlock Text="&#x00B1;"
                       Visibility="{Binding HasTwoSolutions,
                                    Converter={StaticResource
booleanToVisibility}}" />

            <TextBlock Text="{Binding Solution1.Imaginary,
                                      Converter={StaticResource stringFormat},
                                      ConverterParameter=' {0:F3}i'}" />
        </StackPanel>
    </StackPanel>
</Grid>
```

Как видите, я полностью изменил представление решений. Вместо двух элементов *TextBlock* для отображения двух решений я использовал четыре элемента *TextBlock* и с их помощью вывожу на экран одно решение, которое может содержать знак ± (Unicode-код 0x00B1).

В файле выделенного кода реализованы обновление и обработка ошибок:

Проект Silverlight: QuadraticEquationSolver2 Файл: MainPage.xaml.cs

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Media;
using Microsoft.Phone.Controls;

namespace QuadraticEquationSolver2
{
    public partial class MainPage : PhoneApplicationPage
    {
        readonly Brush okBrush;
        static readonly Brush errorBrush = new SolidColorBrush(Colors.Red);

        public MainPage()
        {
            InitializeComponent();
            okBrush = new TextBox().Foreground;
            BindingValidationError += OnBindingValidationError;
        }

        void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
        {
            TextBox txtbox = sender as TextBox;
            txtbox.Foreground = okBrush;

            BindingExpression bindingExpression =
                txtbox.GetBindingExpression(TextBox.TextProperty);
            bindingExpression.UpdateSource();
        }

        void OnBindingValidationError(object sender, ValidationErrorEventArgs args)
        {
            TextBox txtbox = args.OriginalSource as TextBox;
            txtbox.Foreground = errorBrush;
        }
    }
}
```

На снимке экрана представлен *TextBox*, указывающий на недействительность ввода:

Если вы уже писали приложение для решения квадратных уравнений для Windows Phone 7 до прочтения этой главы, его представление на экране может быть практически таким же, а вот структура, я думаю, совершенно иная. Я абсолютно уверен в этом, если приложение предназначалось для среды разработки с применением только кода, такой как Windows Forms.

Обратите внимание, как преобразование приложения в преимущественно XAML-решение заставляет нас полностью переформировать всю его архитектуру. У меня вызывает неизменный интерес то, как наши инструменты, кажется, определяют способ решения наших задач. Но в некотором смысле это хорошо, и, если код создается специально для использования в XAML (например, сервисы привязок и конвертеры данных), вы на правильном пути.

Глава 13
Векторная графика

Мир двухмерной компьютерной графики объединяет в себе векторную и растровую графику – графику линий и графику пикселов, графику приложений для *рисования* и приложений *машинной графики*, графику рисованных изображений и графику фотографий.

Векторная графика – это визуальная реализация аналитической геометрии. Координаты в двухмерной системе координат, представленные в форме (*x*, y), определяют прямые линии и кривые. В Silverlight эти кривые могут быть частями контура эллипса или кривыми Безье, как в обычной кубической форме, так и в упрощенной квадратичной. Эти линии могут быть обведены с применением определенной кисти и стиля. Последовательности соединенных линий и кривых могут определять замкнутую область, которая может быть закрашена с помощью кисти.

Растровая графика (которая будет обсуждаться в следующей главе) работает с растровыми изображениями. В Silverlight очень просто организовать отображение файла в формате PNG или JPEG с помощью элемента *Image*, как было продемонстрировано ранее в главе 4 данной книги. Но как будет показано в следующей главе, растровые изображения также могут формироваться алгоритмически в коде с использованием класса *WriteableBitmap* (Растровое изображение с возможностью записи). Миры растровой и векторной графики пересекаются, когда *ImageBrush* используется для заполнения области, или когда векторная графика используется для формирования изображения в *WriteableBitmap*.

Библиотека *Shapes*

Для работы с векторной графикой приложение на Silverlight использует классы пространства имен *System.Windows.Shapes*, которое обычно называют библиотекой *Shapes*. Это пространство включает абстрактный класс и шесть запечатанных классов, производных от *Shape*:

Object
- *DependencyObject* (абстрактный)
  - *FrameworkElement* (абстрактный)
    - *Shape* (абстрактный)
      - *Rectangle* (запечатанный)
      - *Ellipse* (запечатанный)
      - *Line* (запечатанный)
      - *Polyline* (запечатанный)
      - *Polygon* (запечатанный)
      - *Path* (запечатанный)

Класс *Shape* наследуется от *FrameworkElement*, т.е. этот класс может принимать сенсорный ввод, участвовать в компоновке, и к нему могут применяться трансформации. В Silverlight этих сведений не достаточно, чтобы позволить наследование от самого *Shape*.

Мы уже рассматривали *Rectangle* и *Ellipse*, но это два класса стоят особником в царстве векторной графики, поскольку они не включают никаких упоминаний о координатах. *Ellipse* можно просто вставить в *UserControl*, и он заполнит собой весь элемент управления. Мы можем менять размер этого элемента, но чтобы поместить его в определенную точку,

необходимо привлекать свойства *Margin* или *Padding* либо применять *RenderTransform*, либо поместить его в *Canvas* и использовать присоединенные свойства *Left* и *Top*.

Остальные четыре класса библиотеки *Shape* сильно отличаются от этих двух. Они позволяют позиционировать элементы посредством задания координат. Мы будем рассматривать класс *Path* последним, но он настолько универсален, что его одного может быть достаточно для выполнения всех задач, связанных с векторной графикой. Требуется ли нарисовать дугу или сплайн Безье – во всех этих случаях будет использоваться класс *Path*.

Shape определяет 11 задаваемых свойств, которые наследуются всеми производными от него классами:

- *Fill* типа *Brush*
- *Stroke* типа *Brush*
- *StrokeThickness* (Толщина обводки) типа *double*
- *StrokeStartLineCap* (Наконечник начала линии обводки) и *StrokeEndLineCap* (Наконечник конца линии обводки) типа *PenLineCap* (Наконечник линий)
- *StrokeLineJoin* (Соединение линий обводки) типа *PenLineJoin* (Соединение линий)
- *StrokeMiterLimit* (Предельная длина конуса) типа *double*
- *StrokeDashArray* (Массив длин элементов обводки пунктиром) типа *DoubleCollection* (Коллекция элементов типа double)
- *StrokeDashCap* (Наконечник элементов обводки пунктиром) типа *PenLineCap*
- *StrokeDashOffset* (Смещение обводки пунктиром) типа *double*
- *Stretch* типа *Stretch*

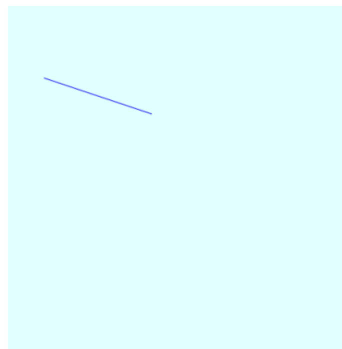
Первые три свойства мы рассматривали, когда обсуждали классы *Rectangle* и *Ellipse*. Свойство *Fill* определяет кисть *Brush*, используемую для заполнения внутреннего пространства фигуры. Свойство *Stroke* – это *Brush*, который используется для закрашивания обводки фигуры. *StrokeThickness* определяет толщину этой обводки.

Все остальные свойства тоже могут использоваться с *Rectangle* и *Ellipse*. Как мы видим, здесь имеется два перечисления (*PenLineCap* и *PenLineJoin*), которые ссылаются на *Pen* (Перо), но в Silverlight нет класса *Pen*. Концептуально все свойства, начинающиеся со слова *Stroke*, составляют объект, традиционно расцениваемый как перо.

Canvas и *Grid*

Класс *Line* (Линия) определяет четыре свойства типа *double*: *X1*, *Y1*, *X2* и *Y2*. Линия проводится из точки с координатами (*X1*, *Y1*) в точку с координатами (*X2*, *Y2*) относительно ее родительского элемента:

```
<Canvas Background="LightCyan">
    <Line X1="50" Y1="100"
          X2="200" Y2="150"
          Stroke="Blue" />
</Canvas>
```



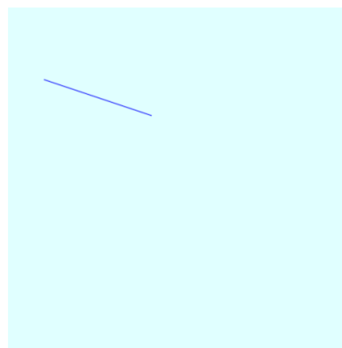
Многие примеры данного приложения будут представлены здесь как фрагмент XAML и соответствующее изображение, выведенное в квадратной области со стороной 480 пикселов. В конце главы мы рассмотрим приложение, создающее эти изображения. Для печатного варианта я задал разрешение этих изображений в 240 точек на дюйм, так что их размер примерно соответствует тому, который мы видим на экране телефона.

Линия начинается в точке с координатами (50, 100) и заканчивается в точке (200, 150). Все координаты отсчитываются относительно верхнего левого угла родительского элемента; значения *X* увеличиваются слева направо; значения *Y* увеличиваются сверху вниз.

Все свойства *X1*, *Y1*, *X2* и *Y2* дублируются свойствами-зависимостями, т.е. могут использоваться как цели стилей, привязок данных и анимаций.

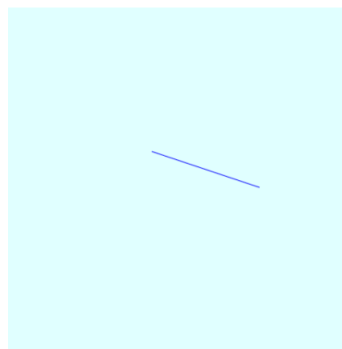
Хотя применение панели *Canvas* кажется более естественным для векторной графики, в *Grid* с одной ячейкой мы получим абсолютно аналогичное изображение:

```
<Grid Background="LightCyan">
    <Line X1="50" Y1="100"
          X2="200" Y2="150"
          Stroke="Blue" />
</Grid>
```



Как правило, для позиционирования элементов в рамках *Canvas* используются присоединенные свойства *Canvas.Left* и *Canvas.Top*. Эти свойства не нужны для *Line*, потому что *Line* имеет собственные координаты. Присоединенные свойства могут использоваться с *Line*, но их значения будут объединяться с координатами:

```
<Canvas Background="LightCyan">
    <Line X1="50" Y1="100"
          X2="200" Y2="150"
          Canvas.Left="150"
          Canvas.Top="100"
          Stroke="Blue" />
</Canvas>
```



Обычно при работе с элементами, имеющими точные координаты, присоединенные свойства *Canvas.Left* и *Canvas.Top* используются только для особых целей, например, перемещения объекта относительно *Canvas*.

Можно вспомнить, что *Canvas* всегда сообщает системе компоновки о том, что имеет нулевой размер. Если задать для *Canvas* любое другое выравнивание, отличное от *Stretch*, он сожмется в ничто независимо от его содержимого.

Поэтому я предпочитаю использовать для векторной графики не *Canvas*, а *Grid* с одной ячейкой.

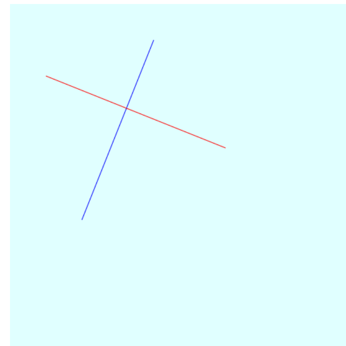
Если в *Grid* располагается один или более элементов *Line* (или любые другие элементы, строящиеся по координатам), *Grid* возвращает размер, соответствующий максимальной неотрицательной координате *X* и максимальной неотрицательной координате *Y* среди всех его дочерних элементов. Иногда это может выглядеть немного странным. Если в *Grid* располагается *Line*, начинающийся в точке (200, 300) и заканчивающийся в точке (210, 310), свойство *ActualWidth* объекта *Line* возвращает значение 210, и свойство *ActualHeight* возвращает значение 310, а *Grid* будет 210 пикселов шириной и 310 пикселов высотой даже несмотря на то, что *Line* занимает лишь небольшую часть этого пространства. (На самом деле *Line* и *Grid* будут на несколько пикселов больше, учитывая *StrokeThickness* отображаемого *Line*.)

Координаты могут иметь отрицательные значения, но *Grid* не учитывает отрицательные координаты. Элементы с отрицательными координатами будут отображаться слева или над *Grid*. Я много размышлял над таким поведением и убедился в его правильности.

Перекрытие и *ZIndex*

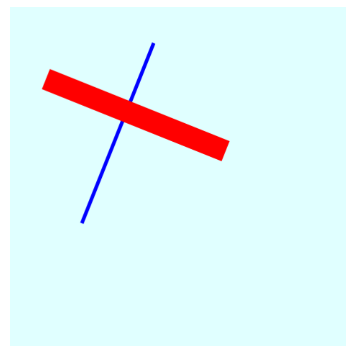
Возьмем две линии:

```
<Grid Background="LightCyan">
    <Line X1="100" Y1="300"
          X2="200" Y2="50"
          Stroke="Blue" />

    <Line X1="50"  Y1="100"
          X2="300" Y2="200"
          Stroke="Red" />
</Grid>
```



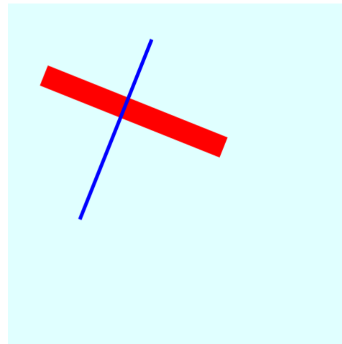
Вторая линия перекрывает первую. Более отчетливо это можно увидеть, если увеличить толщину линии, изменив значение *StrokeThickness* (по умолчанию толщина равна 1 пикселу):

```
<Grid Background="LightCyan">
    <Line X1="100" Y1="300"
          X2="200" Y2="50"
          Stroke="Blue"
          StrokeThickness="5" />

    <Line X1="50" Y1="100"
          X2="300" Y2="200"
          Stroke="Red"
          StrokeThickness="30" />
</Grid>
```



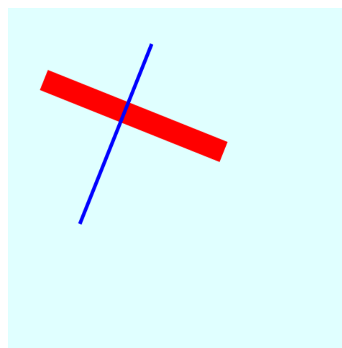
Если требуется, чтобы синяя линия располагалась поверх красной, существует два варианта решения. Можно просто изменить порядок размещения линий в *Grid*:

```
<Grid Background="LightCyan">
    <Line X1="50" Y1="100"
          X2="300" Y2="200"
          Stroke="Red"
          StrokeThickness="30" />

    <Line X1="100" Y1="300"
          X2="200" Y2="50"
          Stroke="Blue"
          StrokeThickness="5" />
</Grid>
```



Или можно задать значение свойства *Canvas.ZIndex*. Это свойство описано в *Canvas*, но применимо для всех типов панелей:

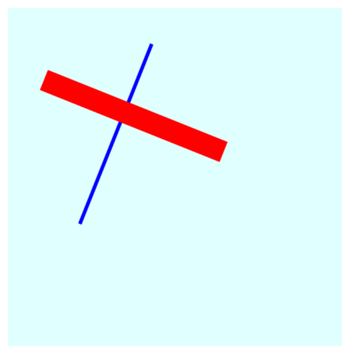
```
<Grid Background="LightCyan">
    <Line Canvas.ZIndex="1"
          X1="100" Y1="300"
          X2="200" Y2="50"
          Stroke="Blue"
          StrokeThickness="5" />

    <Line Canvas.ZIndex="0"
          X1="50" Y1="100"
          X2="300" Y2="200"
          Stroke="Red"
          StrokeThickness="30" />
</Grid>
```



Полилинии и произвольные кривые

Элемент *Line* выглядит просто, а вот разметка для него несколько раздута. Сократить разметку для отрисовки одиночной линии можно, применив не *Line*, а *Polyline* (Полилиния):

```
<Grid Background="LightCyan">
    <Polyline Points="100 300 200 50"
              Stroke="Blue"
              StrokeThickness="5" />
    <Polyline Points="50 100 300 200"
              Stroke="Red"
              StrokeThickness="30" />
</Grid>
```



Свойство *Points* (Точки) класса *Polyline* типа *PointCollection* (Коллекция точек). Это коллекция объектов *Point* (Точка). В XAML множество точек задается просто набором чередующихся координат *X* и *Y*. Пары значений можно задавать в одну строку, записывая их через пробел, как в предыдущем примере, или можно сделать разметку несколько более понятной и добавить запятые. Некоторые разработчики предпочитают ставить запятые между координатами *X* и *Y*:

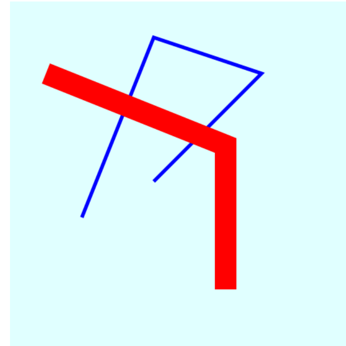
```
<Polyline Points="100,300 200,50" …
```

Другие, и я в том числе, предпочитают разделять запятыми координаты отдельных точек:

```
<Polyline Points="100 300, 200 50"
```

Преимущества *Polyline* в том, что в нем можно задавать любое количество точек:

```
<Grid Background="LightCyan">
    <Polyline Points="100 300, 200  50,
                      350 100, 200 250"
              Stroke="Blue"
              StrokeThickness="5" />

    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="Red"
              StrokeThickness="30" />
</Grid>
```



Каждая дополнительная точка продлевает полилинию на один сегмент.

Но у *Polyline* есть один существенный недостаток, которого нет у *Line*. Поскольку в *Polyline* мы имеем дело с коллекцией объектов *Point*, отдельный объект не может быть целью стиля, привязки данных или анимации. Но это не означает, что мы не можем *изменять* *PointCollection* во время выполнения. Это, несомненно, возможно, и все изменения будут отражены в формируемом визуальном представлении *Polyline*. Я продемонстрирую это в приложении GrowingPolygons (Увеличивающиеся многоугольники) далее в данной главе.

С помощью *Polyline* можно отрисовывать простые соединенные линии, но его истинное предназначение – создание сложных кривых, которые обычно формируются алгоритмически в коде. *Polyline* – это всегда коллекция прямых линий, но если сделать эти линии достаточно короткими, и если их будет очень много, результирующая линия будет выглядеть как настоящая кривая.

Например, используем *Polyline* для отрисовки круга. Обычно круг определяется как фигура с центром в точке (0, 0) и радиусом *R*, координаты (*x*, *y*) всех точек которой удовлетворяют уравнению:

$$x^2 + y^2 = R^2$$

Это всем известная теорема Пифагора.

Но в задачах вычисления координат точек для отрисовки круга эту формулу использовать неудобно. Только представьте, нам придется выбирать значения *x* в диапазоне от –*R* до *R*, затем вычислять *y*, не забывая при этом, что большинству значений *x* соответствуют два значения *y*. И даже если мы будем выбирать значения *x* с регулярным интервалом, мы получим большую плотность точек в области, где *x* стремиться к 0, чем в области, где *y* стремиться к 0.

Для компьютерной графики лучше использовать параметрические уравнения, в которых и *x*, и *y* являются функциями какой-то третьей переменной. Эту переменную иногда называют *t*, предполагая время. В нашем случае этой третьей переменной является угол с диапазоном значений от 0 до 360°.

Предположим, центр круга радиусом *R* располагается в точке (0, 0). Этот круг будет вписан в квадрат со стороной *2R*, где соответственно принятому в Silverlight соглашению значения *x* будут меняться слева направо от –*R* до +*R*, и значения *y* будут меняться сверху вниз от –*R* до +*R*.

Начнем с угла 0°, что соответствует точке (*R*, 0), и будем двигаться по часовой стрелке вдоль по окружности. По мере увеличения угла от 0° to 90° *x* меняется от *R* до 0 и затем до –*R*,

когда угол равен 180°, и возвращается опять к нулю при угле в 270°, и опять становится равным *R*, когда угол достигает значения 360°. Всем известное уравнение:

$$x = R \cdot cos(\alpha)$$

В то же время значения *y* меняются от 0 до *R* и опять к 0, и до –*R*, и назад к 0, что описывается уравнением

$$y = R \cdot sin(\alpha)$$

В зависимости от того, где начинается окружность, и в каком направлении мы идем, формулы могут немного отличаться: косинус и синус могут меняться местами, один из них или оба могут использоваться со знаком минус.

Если в этих формулах использовать разные значения *R*, получим эллипс. Круг можно центрировать в точке (C_x, C_y). Для этого просто добавляем эти координаты в соответствующие формулы:

$$x = C_x + R \cdot cos(\alpha)$$

$$y = C_y + R \cdot sin(\alpha)$$

В приложении эти две формулы помещаются в цикл *for*, который обеспечивает равномерное приращение переменной *angle* (угол) в диапазоне значений от 0 до 360 для формирования коллекции точек.

Какая дискретизация значений позволит получить сглаженную окружность? В данном конкретном примере это зависит от радиуса. Длина окружности вычисляется по формуле 2π*R*, так что если радиус равен 240 пикселам, к примеру, длина окружности будет приблизительно 1500 пикселов. Разделим на 360° и получим примерно 4, т.е. если в цикле *for* угол будет каждый раз получать приращение в 0,25°, результирующие точки будут отстоять друг от друга примерно на один пиксел. (Позже в данной главе я покажу, что можно обойтись и намного меньшим количеством точек.)

Создадим новый проект. Откроем файл MainPage.cs и зададим обработчик события *Loaded*, что позволит выполнять доступ к размерам сетки *ContentPanel*. Вычисляем центр и радиус окружности так, чтобы круг был центрирован относительно панели для содержимого и занимал всю ее площадь:

```
Point center = new Point(ContentPanel.ActualWidth / 2,
                         ContentPanel.ActualHeight / 2 - 1);
double radius = Math.Min(center.X - 1, center.Y - 1);
```

Обратите внимание на то, что при вычислении радиуса мы вычитаем по пикселу от каждой координаты. Таким образом, размер круга будет несколько меньше, чем размер области содержимого. Любой контур имеет некоторую толщину, если размер круга будет таким же, как и размер занимаемой им области, его контур будет отсечен по краям.

Теперь создадим *Polyline* и зададим свойства *Stroke* и *StrokeThickness*:

```
Polyline polyline = new Polyline();
polyline.Stroke = this.Resources["PhoneForegroundBrush"] as Brush;
polyline.StrokeThickness = (double)this.Resources["PhoneStrokeThickness"];
```

Вычисляем объекты *Point* в цикле *for*, используя рассмотренные выше формулы, и добавляем их в коллекцию *Points* объекта *polyline*:

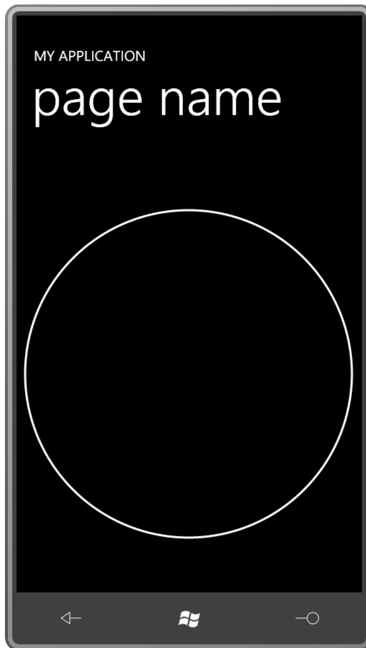
```
for (double angle = 0; angle < 360; angle += 0.25)
{
    double radians = Math.PI * angle / 180;
```

```
    double x = center.X + radius * Math.Cos(radians);
    double y = center.Y + radius * Math.Sin(radians);
    polyline.Points.Add(new Point(x, y));
}
```

Теперь добавим *Polyline* в *Grid*:

```
ContentPanel.Children.Add(polyline);
```

И получаем такой результат:



Мы отрисовали окружность, идя сложным путем. Также мы получили незамкнутую окружность, потому что переменная *angle* в цикле *for* не достигла значения 360. Так что на самом деле там есть небольшой разрыв справа.

Но давайте не будем заниматься устранением проблемы, а сделаем кое-что другое. Пусть верхняя граница диапазона значений угла будет 3600:

```
for (double angle = 0; angle < 3600; angle += 0.25)
```

Теперь цикл обеспечит 10 полных оборотов по окружности. Возьмем этот *angle* и исходное значение *radius* и вычислим *scaledRadius* (Переменный радиус):

```
double scaledRadius = radius * angle / 3600;
```

Подставим это значение *scaledRadius* в наши формулы с синусом и косинусом. В результате получаем архимедову спираль:

Привожу класс полностью:

Проект Silverlight: Spiral Файл: MainPage.xaml.cs (фрагмент)

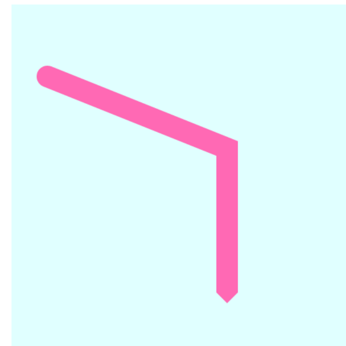
```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        Loaded += OnLoaded;
    }

    void OnLoaded(object sender, RoutedEventArgs args)
    {
        Point center = new Point(ContentPanel.ActualWidth / 2,
                                 ContentPanel.ActualHeight / 2 - 1);
        double radius = Math.Min(center.X - 1, center.Y - 1);

        Polyline polyline = new Polyline();
        polyline.Stroke = this.Resources["PhoneForegroundBrush"] as Brush;
        polyline.StrokeThickness = (double)this.Resources["PhoneStrokeThickness"];

        for (double angle = 0; angle < 3600; angle += 0.25)
        {
            double scaledRadius = radius * angle / 3600;
            double radians = Math.PI * angle / 180;
            double x = center.X + scaledRadius * Math.Cos(radians);
            double y = center.Y + scaledRadius * Math.Sin(radians);
            polyline.Points.Add(new Point(x, y));
        }
        ContentPanel.Children.Add(polyline);
    }
}
```

Не обязательно создавать объект *Polyline* в коде, его можно описать в XAML и затем просто использовать для размещения точек в коллекции *Points*. В главе 15 я покажу, как применить анимацию вращения к спирали и создать гипнотизирующую спираль.

Наконечники, соединения и пунктир

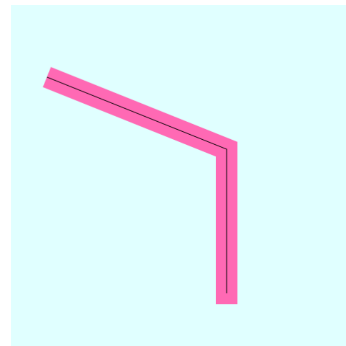
При отрисовке толстых линий можно задавать, как будут выглядеть их концы. Их называют *наконечниками* линий («наконечники», как у стрел). Предлагаемые значения для наконечников являются членами перечисления *PenLineCap*: *Flat* (Плоский) (по умолчанию), *Square* (Квадратный), *Round* (Скругленный) и *Triangle* (Треугольный). Зададим одно из этих значений свойству *StrokeStartLineCap* для наконечника в начале линии и свойству *StrokeEndLineCap* для наконечника в конце линии. И вот как будет выглядеть линия длиной 30 пикселов со скругленным и треугольным наконечниками:

```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Triangle"
/>
</Grid>
```



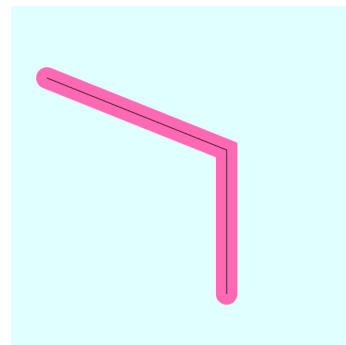
Разница между значениями *Flat* и *Square* не так очевидна на первый взгляд. Чтобы подчеркнуть ее, следующая разметка обеспечивает отрисовку поверх толстой более тонкой линии с такими же координатами, обозначая с ее помощью геометрическое начало и конец линии:

```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Flat"
              StrokeEndLineCap="Square"
/>

    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="Black" />
</Grid>
```



Наконечник со значением *Flat* (сверху слева) обрезает линию точно по ее геометрическим размерам. Значение *Square* продлевает линии на половину ее толщины. Я больше всего люблю скругленные наконечники:

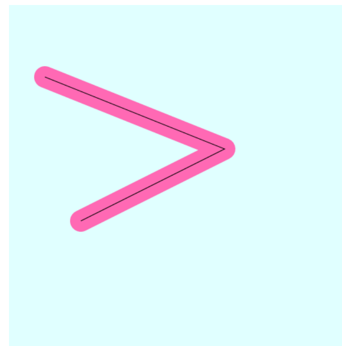
```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
/>

    <Polyline Points=" 50 100, 300 200,
                      300 400"
              Stroke="Black" />
</Grid>
```



Как видите, они тоже продлевают линию на половину ее толщины.

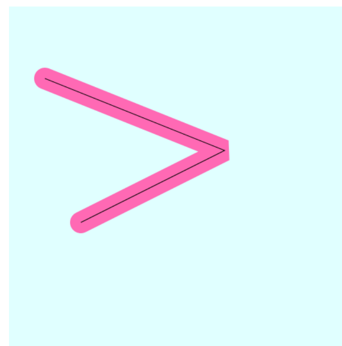
Также можно задавать, как будут отрисовываться углы. Для этого предусмотрено свойство *StrokeLineJoin*, значениями которого являются члены перечисления *PenLineJoin*. Рассмотрим результат применения значения *Round*:

```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round" />

    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="Black" />
</Grid>
```



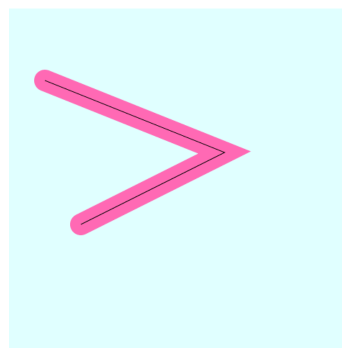
Или значения *Bevel* (Скос):

```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Bevel" />

    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="Black" />
</Grid>
```



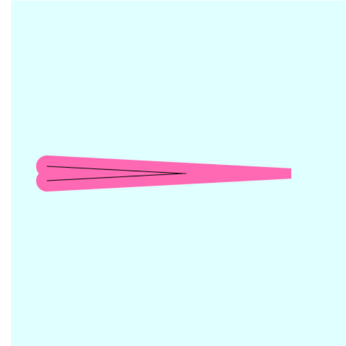
Или значения *Miter* (Конус), которое является значением по умолчанию:

```
<Grid Background="LightCyan">
    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Miter" />

    <Polyline Points=" 50 100, 300 200,
                      100 300"
              Stroke="Black" />
</Grid>
```



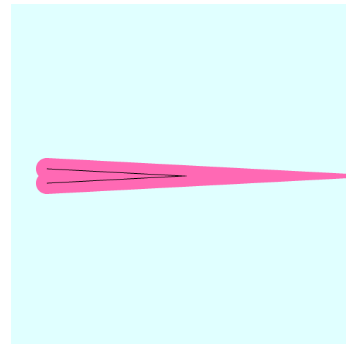
С соединением *Miter* могут возникать небольшие проблемы. Если линии состыковываются под очень острым углом, конус может быть очень длинным. Например, конус в месте соединения для линий шириной 10 пикселов, пересекающихся под углом 1°, будет более 500 пикселов! Для защиты от таких неприятностей предусмотрено свойство *StrokeMiterLimit*:

```
<Grid Background="LightCyan">
    <Polyline Points="50 230, 240 240,
                      50 250"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Miter" />

    <Polyline Points="50 230, 240 240,
                      50 250"
              Stroke="Black" />
</Grid>
```



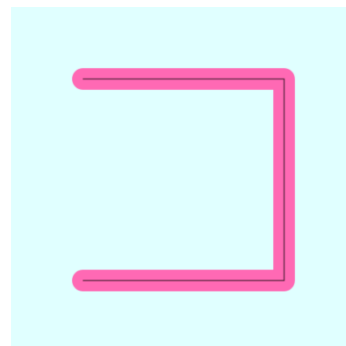
Значение по умолчанию для *StrokeMiterLimit* – 10 (но оно зависит от половины *StrokeThickness*), но его можно и увеличить:

```
<Grid Background="LightCyan">
    <Polyline Points="50 230, 240 240,
                      50 250"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Miter"
              StrokeMiterLimit="50" />

    <Polyline Points="50 230, 240 240,
                      50 250"
              Stroke="Black" />
</Grid>
```



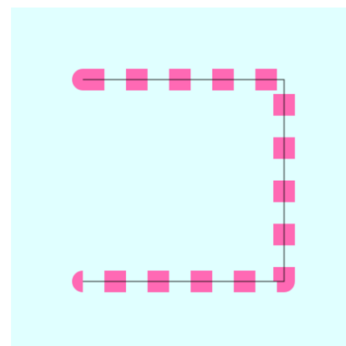
Рассмотрим две линии: одна толстая, другая тонкая. Тонкая располагается поверх толстой. Обе имеют одинаковые геометрические координаты. Начальная точка – в верхнем левом углу, конечная – в нижнем левом:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



Линию можно сделать пунктирной. Для этого зададим *StrokeDashArray*, который обычно включает всего два числа, например, 1 и 1:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="1 1" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
```

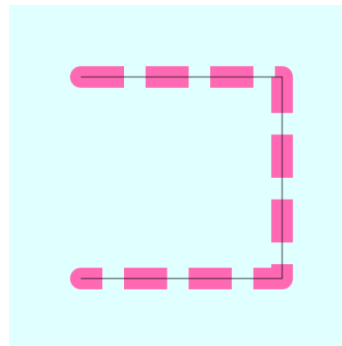


```
</Grid>
```

Это означает, что штрихи и пробелы пунктирной линии будут равны толщине линии (30 пикселов в данном случае). Как видим, наконечники обрабатываются несколько иначе: они отрисовываются или не отрисовываются в зависимости от того, как завершается линия, штрихом или пробелом.

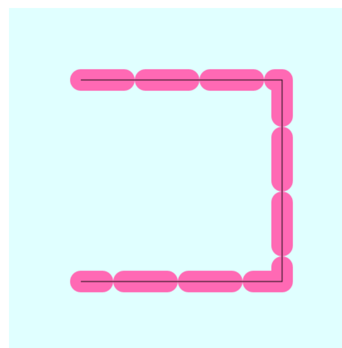
Чтобы увеличить размер штрихов пунктирной линии, увеличиваем первое значение

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="2 1" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



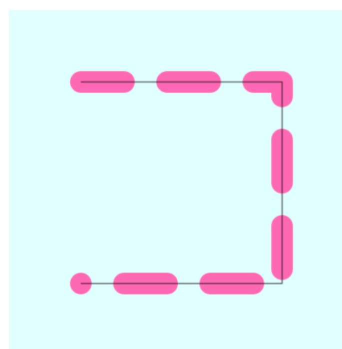
Также есть возможность задать собственные наконечники для штрихов пунктирной линии. Для этого зададим в качестве значения свойства *StrokeDashCap* один из членов перечисления *PenLineCap*: *Flat* (по умолчанию), *Triangle*, *Square* или *Round* (мое любимое):

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="2 1"
              StrokeDashCap="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



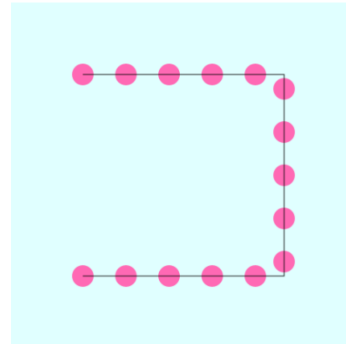
Это вскрывает небольшую проблему. Скругленные наконечники каждого из штрихов увеличивают его длину с каждого конца на половину толщины линии, т.е. теперь штрихи соприкасаются. Исправить это можно, увеличив расстояния между ними:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="2 2"
              StrokeDashCap="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



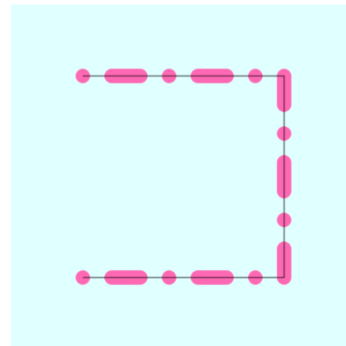
Чтобы сделать пунктир точечным, несомненно, надо использовать наконечники *Round*. Также сделаем так, чтобы каждая точка отстояла от соседней на ширину точки. Значения *StrokeDashArray*, задаваемые в этом случае, могут вызвать некоторое недоумение: длина штриха равна 0 и расстояние между соседними штрихами равно 2:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="0 2"
              StrokeDashCap="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



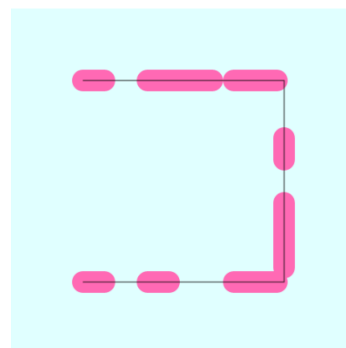
Можно задать более двух чисел. Это позволит получить, например, такое сочетание точек и штрихов:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="0 2 2 2"
              StrokeDashCap="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



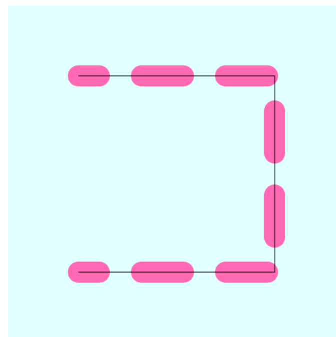
Даже совсем не обязательно задавать четное количество чисел:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="1 2 3"
              StrokeDashCap="Round" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



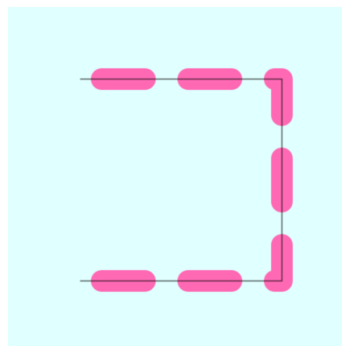
Еще одно свойство, используемое при описании пунктира – *StrokeDashOffset*. Оно также связано и с толщиной линии. Это свойство позволяет отрисовывать штрихи, начиная с середины, из-за чего первый штрих (в верхнем левом углу) короче остальных:

```
<Grid Background="LightCyan">
        <Polyline Points="100 100, 380
100,
                          380 380, 100
380"
                  Stroke="HotPink"
                  StrokeThickness="30"

StrokeStartLineCap="Round"

StrokeEndLineCap="Round"
                  StrokeLineJoin="Round"
                  StrokeDashArray="2 2"
                  StrokeDashCap="Round"
                  StrokeDashOffset="1" />

        <Polyline Points="100 100, 380
100,
                          380 380, 100
380"
                  Stroke="Black" />
</Grid>
```



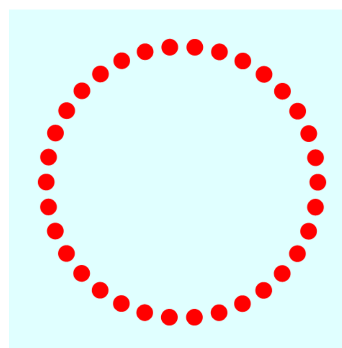
Или можно начать с пробела:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="HotPink"
              StrokeThickness="30"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="2 2"
              StrokeDashCap="Round"
              StrokeDashOffset="3" />

    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Black" />
</Grid>
```



Таким точечным пунктиром можно обвести контур эллипса, например:

```
<Grid Background="LightCyan">
    <Ellipse Width="400" Height="400"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"
             Stroke="Red"
             StrokeThickness="23.22"
             StrokeDashArray="0 1.5"
             StrokeDashCap="Round" />
</Grid>
```

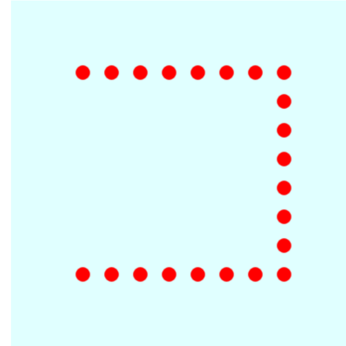


Такой контур выглядит необычно, и чтобы все было красиво, приходится экспериментировать или выполнять некоторые вычисления.

Многоугольник и заливка

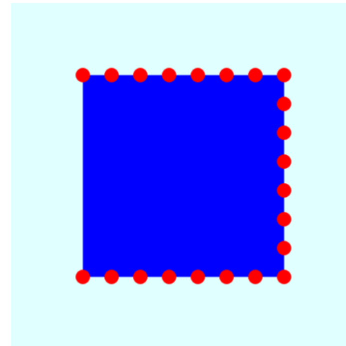
Для демонстрации пунктиров я использовал полилинию, которая образует три стороны квадрата:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Red"
              StrokeThickness="20"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="0 2"
              StrokeDashCap="Round" />
</Grid>
```



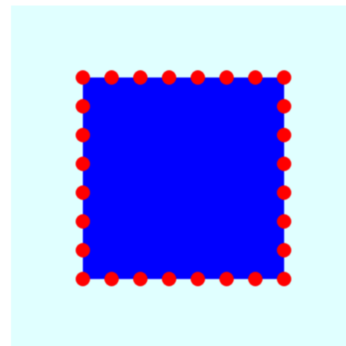
Но если задать кисть *Fill*, внутренняя область полилинии закрашивается так, как будто она описывает замкнутую фигуру:

```
<Grid Background="LightCyan">
    <Polyline Points="100 100, 380 100,
                      380 380, 100 380"
              Stroke="Red"
              StrokeThickness="20"
              Fill="Blue"
              StrokeStartLineCap="Round"
              StrokeEndLineCap="Round"
              StrokeLineJoin="Round"
              StrokeDashArray="0 2"
              StrokeDashCap="Round" />
</Grid>
```



Чтобы получить действительно замкнутую фигуру, добавим в коллекцию *Points* еще одну точку, координаты которой совпадают с координатами первой точки, или используем вместо *Polyline* класс *Polygon* (Многоугольник):

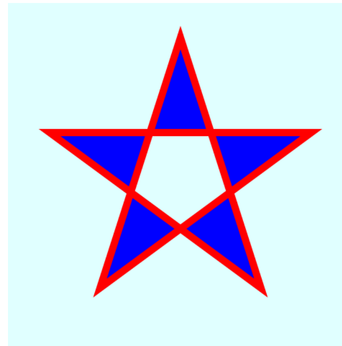
```
<Grid Background="LightCyan">
    <Polygon Points="100 100, 380 100,
                     380 380, 100 380"
             Stroke="Red"
             StrokeThickness="20"
             Fill="Blue"
             StrokeStartLineCap="Round"
             StrokeEndLineCap="Round"
             StrokeLineJoin="Round"
             StrokeDashArray="0 2"
             StrokeDashCap="Round" />
</Grid>
```



Оба элемента имеют коллекцию *Points*, но *Polygon* в случае необходимости замыкается автоматически.

Как только мы начинаем закрашивать область, ограниченную *Polygon*, возникает вопрос: как должны обрабатываться пересечения контурных линий. Класс *Polygon* определяет свойство *FillRule* (Правило заливки), которое позволяет сделать выбор. Классический пример – пятиконечная звезда. В данном случае свойству *FillRule* задано значение по умолчанию, *EvenOdd* (Заливка с пустыми областями):

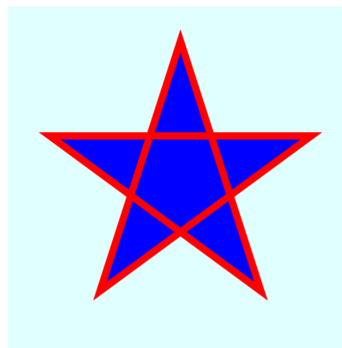
```
<Grid Background="LightCyan">
    <Polygon Points="240  48, 352 396,
                      58 180, 422 180,
                     128 396"
             Stroke="Red"
             StrokeThickness="10"
             Fill="Blue"
             FillRule="EvenOdd" />
</Grid>
```



Алгоритм *EvenOdd* определяет, должна ли быть закрашена замкнутая область, концептуально следующим образом: выбирается точка в этой области, скажем, где-то в центре, и из нее проводится воображаемая бесконечная линия. Эта воображаемая линия пересечет какие-то из контурных линий. Если она пересекает нечетное число этих линий, как это происходит в пяти точках, область закрашивается. Если пересекается четное число контуров, как в центре, область не закрашивается.

Альтернативным значением *FillRule* является *NonZero* (Полная заливка):

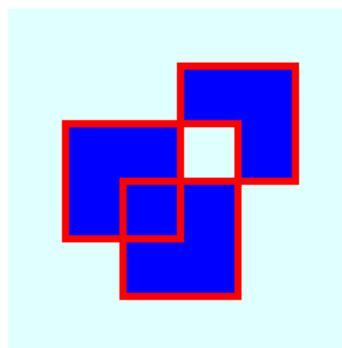
```
<Grid Background="LightCyan">
    <Polygon Points="240  48, 352 396,
                      58 180, 422 180,
                     128 396"
             Stroke="Red"
             StrokeThickness="10"
             Fill="Blue"
             FillRule="NonZero" />
</Grid>
```



Правило заливки *NonZero* несколько сложнее, потому что в нем учитываются направления отрисовки контурных линий. Если количество контурных линий, отрисованных в одном направлении, соответствует количеству этих линий, отрисованных в другом направлении, область не закрашивается. Но в любом секторе этой звезды все контурные линии проведены в одном направлении.

Ни один из этих двух вариантов *FillRule* не гарантирует закрашивания всех внутренних областей. Рассмотрим, например, такую замысловатую фигуру. В ней есть замкнутая область, которая остается незакрашенной, даже если задано значение *NonZero*:

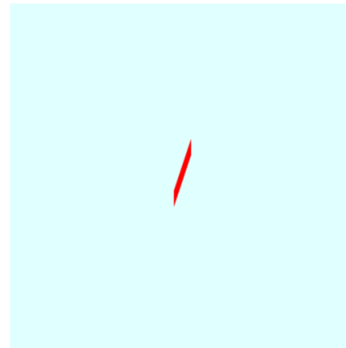
```
<Grid Background="LightCyan">
    <Polygon Points=" 80 160,  80 320,
                     240 320, 240  80,
                     400  80, 400 240,
                     160 240, 160 400,
                     320 400, 320 160"
             Stroke="Red"
             StrokeThickness="10"
             Fill="Blue"
             FillRule="NonZero" />
</Grid>
```



Свойство *Stretch*

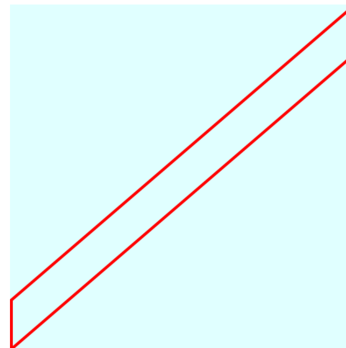
Единственное задаваемое свойство *Shape*, которое мы пока обошли нашим вниманием – это *Stretch*. Аналогичное свойство есть у элемента *Image*. Его значениями являются члены перечисления *Stretch*: *None* (по умолчанию), *Fill*, *Uniform* или *UniformToFill*. Возьмем простой маленький *Polygon*:

```
<Grid Background="LightCyan">
    <Polygon Points="250 200, 250 210,
                     230 270, 230 260"
             Stroke="Red"
             StrokeThickness="4" />
</Grid>
```



Теперь тот же *Polygon*, свойству *Stretch* которого задано значение *Fill*.

```
<Grid Background="LightCyan">
    <Polygon Points="250 200, 250 210,
                     230 270, 230 260"
             Stroke="Red"
             StrokeThickness="4"
             Stretch="Fill" />
</Grid>
```



Независимо от заданных координат, прямоугольник растягивается, заполняя контейнер полностью, без сохранения пропорций. Сохранение пропорций обеспечивают значения *Uniform* и *UniformToFill*, так же как и для элемента *Image*.

В векторной графике свойство *Stretch* для *Shape* используется не часто, но если в конкретном векторном изображении требуется закрасить область произвольного размера, оно очень пригодится.

Динамические многоугольники

Как мы видели, если свойство, продублированное свойством-зависимостью, меняется во время выполнения, соответственно меняется и элемент, свойство которого было изменено. Такое поведение обеспечивает обработчик события изменения значения свойства, встроенный в свойства-зависимости.

Некоторые коллекции также будут реагировать на изменения. Классы *Collection* (Коллекция), наследуемые от *PresentationFrameworkCollection* (Коллекция инфраструктуры представления), реагируют на изменения, обусловленные добавлением или удалением объекта из коллекции. Уведомление об изменении передается вверх по дереву наследования элементу, включающему коллекцию. В некоторых случаях изменения свойств-зависимостей членов коллекции также формируют уведомления. (К сожалению, все детали процесса нотификации в данном случае скрыты от разработчика приложения.) Класс *UIElementCollection*, используемый классами *Panel* в качестве значения свойства *Children*, наследуется от класса *PresentationFrameworkCollection*, как и *PointCollection* в *Polyline* и *Polygon*.

Во время выполнения объекты *Point* могут динамически добавляться или удаляться из *PointCollection*, что приведет к соответствующим изменениям *Polyline* или *Polygon*.

В проекте GrowingPolygons имеется файл MainPage.xaml, в котором создается экземпляр элемента *Polygon* и задаются значения паре его свойств:

Проект Silverlight: GrowingPolygons Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Polygon Name="polygon"
             Stroke="{StaticResource PhoneForegroundBrush}"
             StrokeThickness="{StaticResource PhoneStrokeThickness}" />
</Grid>
```

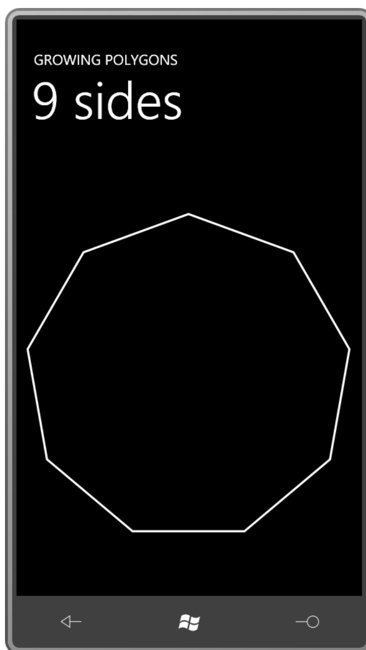
После того как сформировано событие *Loaded*, файл выделенного кода определяет размер панели для содержимого (как и в приложении Spiral) и начинает с получения тех же данных. Но обработчик события *OnLoaded* (Когда загружен) просто добавляет две точки *Points* в коллекцию класса *Polygon*, чтобы определить вертикальную линию. Все остальное происходит при обработке событий *Tick* класса *DispatcherTimer* (для использования которого, конечно же, посредством директивы *using* необходимо подключить пространство имен *System.Windows.Threading*):

Проект Silverlight: GrowingPolygons Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Point center;
    double radius;
    int numSides = 2;

    public MainPage()
    {
        InitializeComponent();
        Loaded += OnLoaded;
    }

    void OnLoaded(object sender, RoutedEventArgs args)
    {
        center = new Point(ContentPanel.ActualWidth / 2 - 1,
                           ContentPanel.ActualHeight / 2 - 1);
        radius = Math.Min(center.X, center.Y);

        polygon.Points.Add(new Point(center.X, center.Y - radius));
        polygon.Points.Add(new Point(center.X, center.Y + radius));

        DispatcherTimer tmr = new DispatcherTimer();
        tmr.Interval = TimeSpan.FromSeconds(1);
        tmr.Tick += OnTimerTick;
        tmr.Start();
    }

    void OnTimerTick(object sender, EventArgs args)
    {
        numSides += 1;

        for (int vertex = 1; vertex < numSides; vertex++)
        {
            double radians = vertex * 2 * Math.PI / numSides;
            double x = center.X + radius * Math.Sin(radians);
            double y = center.Y - radius * Math.Cos(radians);
```

```
            Point point = new Point(x, y);

            if (vertex < numSides - 1)
                polygon.Points[vertex] = point;
            else
                polygon.Points.Add(point);
        }

        PageTitle.Text = "" + numSides + " sides";
    }
}
```

Каждую секунду приложение заменяет в коллекции *Points* класса *Polygon* все объекты *Point*, кроме одного. Единственным неизменным остается первый в коллекции *Point* – это верхний центральный *Point*. Кроме того, обработчик событий *Tick* добавляет новый объект в конец коллекции. В результате получаем многоугольник, каждую секунду увеличивающийся на одну грань:



Теперь можете сами проверить, сколько точек необходимо, чтобы многоугольник визуально превратился в круг!

Заметьте, что приложение не пытается изменить свойства *X* и *Y* существующего объекта *Point* коллекции, а полностью заменяет его. *Point* – это структура, и он не реализует никакого механизма уведомления. У *PointCollection* нет никакой возможности узнать об изменении свойства одного из *Point* в коллекции, уведомление обеспечивается только в случае замены объекта *Point*.

В реальном приложении при внесении большого числа изменений в *PointCollection* лучше отсоединить его от *Polygon*. Это предупредит формирование и передачу в *Polygon* массы уведомлений об изменениях *PointCollection*. Код будет выглядеть примерно так:

```
PointCollection points = polygon.Points;
polygon.Points = null;

// ... вносит изменения в коллекцию точек

polygon.Points = points;
```

Отсоединение *PointCollection* осуществляется путем сохранения ссылки на него и задания свойству *Points* значения *null*. После внесения всех необходимых изменений *PointCollection* повторно присоединяется к *Polygon*, и *Polygon* перестраивается соответственно новой коллекции точек.

Элемент *Path*

Хотя классы *Line*, *Polyline* и *Polygon* удобны и просты в использовании, последний потомок *Shape*, класс *Path* (Контур), объединяет в себе практически всю их функциональность.

Класс *Path* самостоятельно определяет всего одно свойство: *Data* типа *Geometry* (Геометрический элемент). Геометрические элементы являются очень важной концепцией в векторной графике на Silverlight. В общем случае геометрический элемент – это коллекция прямых линий и кривых, некоторые из которых могут соединяться друг с другом (или нет) и определять замкнутые области (или нет). В некоторых графических средах разработки геометрические элементы называют *контурами*. В Silverlight *Path* – это элемент, использующий объект *Geometry* как значение свойства *Data*.

Важно понимать, что объект *Geometry* – это лишь голые координатные точки. Для геометрического элемента нет понятия кистей, или толщины линии, или стилей. Поэтому чтобы действительно получить какое-то визуальное представление на экране, *Geometry* необходимо комбинировать с элементом *Path*. *Geometry* определяет координаты, *Path* определяет кисти обводки и заливки.

В иерархии классов Silverlight *Geometry* следующую позицию:

Object
    DependencyObject (абстрактный)
        Geometry (абстрактный)
            LineGeometry (запечатанный)
            RectangleGeometry (запечатанный)
            EllipseGeometry (запечатанный)
            GeometryGroup (запечатанный)
            PathGeometry (запечатанный)

Если среди производных от *Shape* мы будем пользоваться преимущественно одним только *Path*, то среди потомков *Geometry* нас интересует главным образом класс *PathGeometry*. Но, конечно, мы обсудим все классы, потому что часто удобно использовать именно их. Наследоваться от *Geometry* напрямую нельзя.

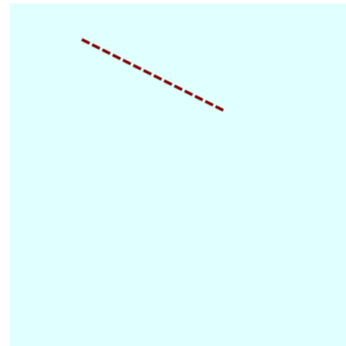
Geometry определяет четыре открытых свойства:

- статическое свойство только для чтения *Empty* (Пустой) типа *Geometry*
- статическое свойство только для чтения *StandardFlatteningTolerance* (Стандартный допуск, используемый для кусочно-линейной аппроксимации) типа *double*
- свойство только для чтения *Bounds* (Границы) типа *Rect*
- свойство *Transform* типа *Transform*

Самыми полезными являются два последних. Свойство *Bounds* задает минимальный прямоугольник, который может вместить геометрический элемент, и *Transform* позволяет применять преобразования к данному геометрическому элементу (как будет продемонстрировано ниже).

LineGeometry (Геометрический элемент линия) определяет два свойства типа *Point*: *StartPoint* и *EndPoint*:

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="4"
        StrokeDashArray="3 1">
    <Path.Data>
      <LineGeometry StartPoint="100 50"
                    EndPoint="300 150" />
   </Path.Data>
 </Path>
</Grid>
```

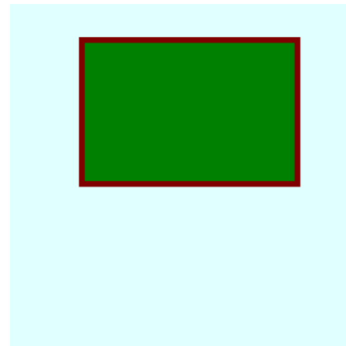


Обратите внимание на разделение обязанностей между *Geometry* и *Path*: *Geometry* обеспечивает координаты; и *Path* обеспечивает все данные для формирования визуального представления.

Когда мы знаем о существовании элементов *Line* и *Polyline*, *LineGeometry* может казаться лишним, но в отличие от *Line* и *Polyline* у *LineGeometry* есть два свойства-зависимости типа *Point*. Эти свойства могут быть очень полезными в качестве целей анимаций в некоторых сценариях.

RectangleGeometry (Геометрический элемент прямоугольник) описывает свойство *Rect* типа *Rect*. Это структура, определяющая прямоугольник с помощью четырех чисел: два числа задают координаты верхнего левого угла, и два других числа – размеры прямоугольника. В XAML эти четыре числа задаются в такой последовательности: координаты *x* и *y* верхнего левого угла и за ними ширина и высота прямоугольника:

```
<Grid Background="LightCyan">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
        <Path.Data>
            <RectangleGeometry
                Rect="100 50 300 200" />
        </Path.Data>
    </Path>
</Grid>
```

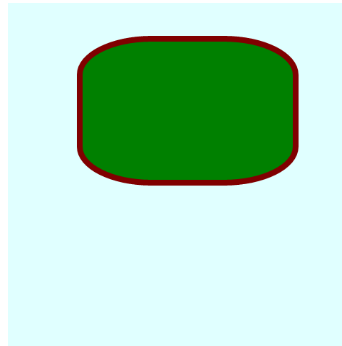


В данном примере координаты нижнего правого угла прямоугольника – (400, 250). В коде структура *Rect* имеет три конструктора, которые позволяют нам задать прямоугольник либо через *Point* и *Size*, либо посредством двух объектов *Point*, либо используя строку из четырех чисел, как в XAML: (*x*, *y*, *ширина*, *высота*).

Свойство *Bounds* класса *Geometry* также типа *Rect*. Для приведенного выше *RectangleGeometry* свойство *Bounds* будет возвращать те же значения: (100, 50, 300, 200). Для *LineGeometry* из предыдущего примера *Bounds* возвратит (100, 50, 200, 100).

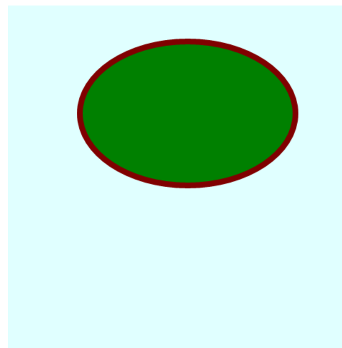
RectangleGeometry также определяет свойства *RadiusX* и *RadiusY* для скругления углов:

```
<Grid Background="LightCyan">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
        <Path.Data>
            <RectangleGeometry
                Rect="100 50 300 200"
                RadiusX="100"
                RadiusY="50" />
        </Path.Data>
    </Path>
</Grid>
```



Класс *EllipseGeometry* (Геометрический элемент эллипс) также имеет свойства *RadiusX* и *RadiusY*, но они используются для задания длин осей эллипса. Центр эллипса в *EllipseGeometry* задается посредством свойства *Center* типа *Point*:

```
<Grid Background="LightCyan">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
        <Path.Data>
            <EllipseGeometry
                Center="250 150"
                RadiusX="150"
                RadiusY="100" />
        </Path.Data>
    </Path>
</Grid>
```



Для определения местоположения окружности или эллипса часто удобнее задавать их центр, а не верхний левый угол (как с элементом *Ellipse*), особенно если учесть, что эти фигуры не имеют углов!

Давайте немного поупражняемся в интерактивном рисовании с помощью проекта TouchAndDrawCircles (Прикоснись и нарисуй круги). Когда пользователь касается экрана, это приложение создает новый круг, используя классы *Path* и *EllipseGeometry*. По мере того как пользователь ведет пальцем по экрану, круг увеличивается. Как только пользователь снимает палец с экрана, круг закрашивается случайным цветом. Уже существующий круг можно перемещать по экрану посредством касания.

В файле MainPage.xaml сетка для содержимого изначально пуста. Единственное, я задал ей отличный от нуля *Background*, чтобы она могла формировать события манипуляций:

Проект Silverlight: TouchAndDrawCircles Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      Background="Transparent" />
```

В файле выделенного кода всего несколько полей для отслеживания того, что происходит:

Проект Silverlight: TouchAndDrawCircles Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    Random rand = new Random();
    bool isDrawing, isDragging;
    Path path;
    EllipseGeometry ellipseGeo;
```

```
    …
}
```

Два поля типа Boolean указывают на текущее действие. Поле *Path* имеет действительное значение только во время отрисовки нового круга. Поле *EllipseGeometry* действительно при отрисовке нового круга или перемещении существующего.

Перегрузка метода *OnManipulationStarted* инициирует операцию рисования или перетягивания, но не допускает одновременного выполнения обеих операций. Значением свойства *OriginalSource* аргументов события может быть либо элемент *Path* – это означает, что пользователь коснулся одного из существующих кругов и хочет переместить его – либо *ContentPanel*, что инициирует новую операцию рисования:

Проект Silverlight: TouchAndDrawCircles Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    if (isDrawing || isDragging)
        return;

    if (args.OriginalSource is Path)
    {
        ellipseGeo = (args.OriginalSource as Path).Data as EllipseGeometry;

        isDragging = true;
        args.ManipulationContainer = ContentPanel;
        args.Handled = true;
    }
    else if (args.OriginalSource == ContentPanel)
    {
        ellipseGeo = new EllipseGeometry();
        ellipseGeo.Center = args.ManipulationOrigin;
        path = new Path();
        path.Stroke = this.Resources["PhoneForegroundBrush"] as Brush;
        path.Data = ellipseGeo;
        ContentPanel.Children.Add(path);

        isDrawing = true;
        args.Handled = true;
    }

    base.OnManipulationStarted(args);
}
```

В XAML-файле я задал свойству *Background* объекта *ContentPanel* значение *Transparent*, чтобы обеспечить возможность формирования событий *Manipulation*. Если значением свойства *OriginalSource* является *Grid*, то этот *Grid* будет и *ManipulationContainer*, и *ManipulationOrigin*. Это точка, которая необходима для задания значения свойства *Center* этого нового *EllipseGeometry*.

Для реализации операции перетягивания перегруженный метод *OnManipulationDelta* использует свойство *DeltaManipulation* аргументов события, с помощью которого изменяет свойство *Center* объекта *EllipseGeometry*:

Проект Silverlight: TouchAndDrawCircles Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationDelta(ManipulationDeltaEventArgs args)
{
    if (isDragging)
    {
```

```
        Point center = ellipseGeo.Center;
        center.X += args.DeltaManipulation.Translation.X;
        center.Y += args.DeltaManipulation.Translation.Y;
        ellipseGeo.Center = center;

        args.Handled = true;
    }
    else if (isDrawing)
    {
        Point translation = args.CumulativeManipulation.Translation;
        double radius = Math.Max(Math.Abs(translation.X),
                                 Math.Abs(translation.Y));
        ellipseGeo.RadiusX = radius;
        ellipseGeo.RadiusY = radius;

        args.Handled = true;
    }

    base.OnManipulationDelta(args);
}
```

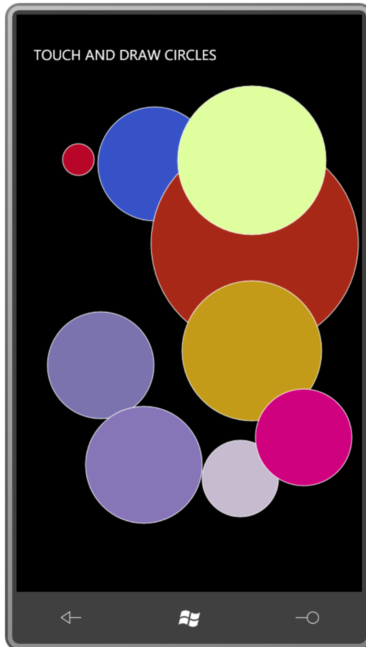
Для сравнения, при реализации операции рисования этот метод изменяет значения свойств *RadiusX* и *RadiusY* объекта *EllipseGeometry*. Для этого он использует свойство *CumulativeManipulation*, которое хранит всю манипуляцию, начиная с момента формирования события *ManipulationStarted*. Причина применения другого свойство проста: если пользователь инициирует операцию рисования и затем перемещает палец влево или вверх, коэффициенты переноса будут отрицательными. Но эти отрицательные числа должны быть преобразованы в положительное значение радиуса окружности. Оказывается проще взять абсолютное значение суммы всех коэффициентов переноса, чем менять существующие размеры.

Когда пользователь снимает палец с экрана, формируется событие *OnManipulationCompleted* (По завершении манипуляции), обеспечивающее очистку:

Проект Silverlight: TouchAndDrawCircles Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationCompleted(ManipulationCompletedEventArgs args)
{
    if (isDragging)
    {
        isDragging = false;
        args.Handled = true;
    }
    else if (isDrawing)
    {
        Color clr = Color.FromArgb(255, (byte)rand.Next(256),
                                        (byte)rand.Next(256),
                                        (byte)rand.Next(256));
        path.Fill = new SolidColorBrush(clr);

        isDrawing = false;
        args.Handled = true;
    }

    base.OnManipulationCompleted(args);
}
```

Очистка для операции переноса проста. Но операция рисования должна быть завершена заданием элементу *Path* случайной кисти *Fill*.

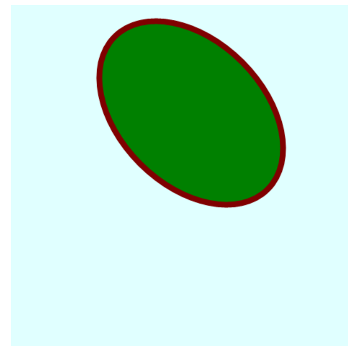


При работе с приложением можно заметить небольшую задержку между моментом, когда пользователь касается экрана, чтобы начать отрисовку или перетягивание круга, и фактическим началом выполнения этих операций. Это особенность событий Manipulation.

Геометрические элементы и трансформации

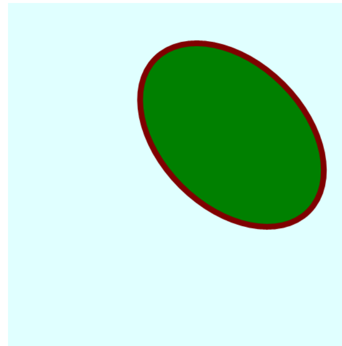
Чтобы с помощью *EllipseGeometry* отрисовать эллипс с наклонными осями, применяем *RotateTransform*. И у нас есть выбор. Поскольку *Path* наследуется от *UIElement*, мы можем применить *RotateTransform* к свойству *RenderTransform* элемента *Path*:

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="8"
        Fill="Green">
    <Path.Data>
      <EllipseGeometry Center="250 150"
                       RadiusX="150"
                       RadiusY="100" />
    </Path.Data>
    <Path.RenderTransform>
      <RotateTransform Angle="45"
                       CenterX="250"
                       CenterY="150" />
    </Path.RenderTransform>
  </Path>
</Grid>
```



Обратите внимание, что свойствам *CenterX* и *CenterY* объекта *RotateTransform* заданы те же значения, что и свойству *Center* самого *EllipseGeometry*, т.е. эллипс будет поворачиваться вокруг своего центра. При работе с объектами *Path* и *Geometry* обычно проще задать фактический центр трансформации, чем использовать *RenderTransformOrigin*. Обычно для *RenderTransformOrigin* задаются относительные координаты, например, (0.5, 0.5) определяет центр, но посмотрим, к чему это приводит в данном случае:

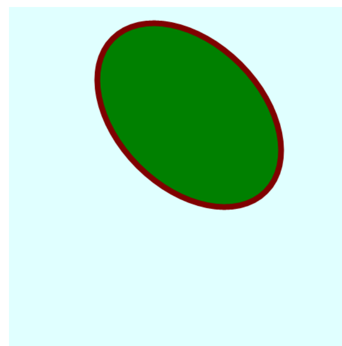
```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="8"
        Fill="Green"
        RenderTransformOrigin="0.5 0.5">
    <Path.Data>
      <EllipseGeometry Center="250 150"
                       RadiusX="150"
                       RadiusY="100" />
    </Path.Data>

    <Path.RenderTransform>
      <RotateTransform Angle="45" />
    </Path.RenderTransform>
  </Path>
</Grid>
```



В данном случае проблема в том, что элемент *Path* достаточно велик, чтобы вместить *EllipseGeometry* с центром в точке (250, 150) и *RadiusX* и *RadiusY* равными 150 и 100, соответственно. При таких параметрах элемент *Path* должен быть как минимум 400 пикселов шириной и 250 пикселов высотой. (На самом деле он немного больше из-за ненулевого *StrokeThickness*.) Центр этого *Path* находится примерно в точке (200, 125). Кроме того, как и у остальных элементов, свойства *HorizontalAlignment* и *VerticalAlignment* элемента *Path* по умолчанию имеют значение *Stretch*, так что он действительно заполняет свой контейнер, каковым в данном случае является квадрат со стороной 480 пикселов, и вращение выполняется вокруг точки (240, 240).

Также можно применить трансформацию к самому объекту *Geometry*:

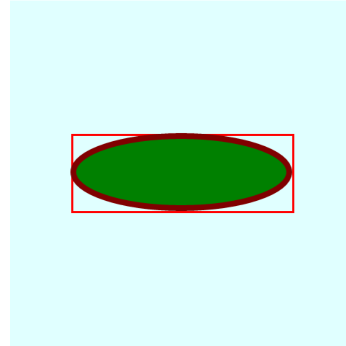
```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="8"
        Fill="Green">
    <Path.Data>
      <EllipseGeometry Center="250 150"
                       RadiusX="150"
                       RadiusY="100">
        <EllipseGeometry.Transform>
          <RotateTransform Angle="45"
                           CenterX="250"
                           CenterY="150"
/>
        </EllipseGeometry.Transform>
      </EllipseGeometry>
    </Path.Data>
  </Path>
</Grid>
```



Получаемый результат выглядит абсолютно точно так же, как и в предыдущем примере, в котором *CenterX* и *CenterY* для *RotateTransform* задавались явно. Но результаты трансформаций могут сильно отличаться в зависимости от того, к свойству *RenderTransform* какого объекта, *Path* или *Geometry*, они применяются.

Свойство *RenderTransform* не оказывает влияния на то, как элемент воспринимается системой компоновки, а вот свойство *Transform* класса меняет воспринимаемые размеры элемента. Чтобы увидеть эту разницу, заключим *Path* с *EllipseGeometry* в центрированный *Border*:

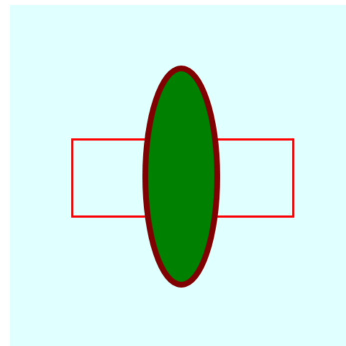
```
<Grid Background="LightCyan">
  <Border BorderBrush="Red"
          BorderThickness="3"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
      <Path.Data>
        <EllipseGeometry Center="150 50"
                         RadiusX="150"
                         RadiusY="50" />
      </Path.Data>
    </Path>
  </Border>
</Grid>
```



Я намеренно задал свойству *Center* объекта *EllipseGeometry* те же два значения, что и свойствам *RadiusX* и *RadiusY*, так *Path* будет занимать именно столько места, сколько необходимо для визуального представления эллипса.

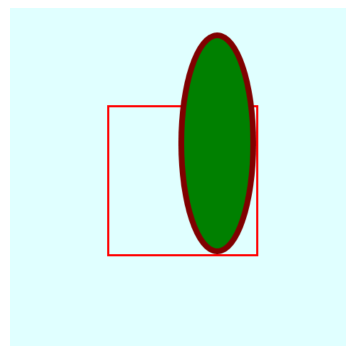
Теперь зададим *RenderTransform* элемента *Path* для реализации трансформации вращением:

```
<Grid Background="LightCyan">
  <Border BorderBrush="Red"
          BorderThickness="3"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
      <Path.Data>
        <EllipseGeometry Center="150 50"
                         RadiusX="150"
                         RadiusY="50" />
      </Path.Data>
      <Path.RenderTransform>
        <RotateTransform Angle="90"
                         CenterX="150"
                         CenterY="50" />
      </Path.RenderTransform>
    </Path>
  </Border>
</Grid>
```



Как было четко показано ранее в главе 8, *RenderTransform* не влияет на то, как элемент воспринимается системой компоновки. *Border* по-прежнему определяет собственные размеры на основании исходного, не развернутого, *Path*. Применение трансформации к *EllipseGeometry* приводит совершенно к другому результату:

```
<Grid Background="LightCyan">
  <Border BorderBrush="Red"
          BorderThickness="3"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
      <Path.Data>
        <EllipseGeometry Center="150 50"
                         RadiusX="150"
                         RadiusY="50">
          <EllipseGeometry.Transform>
            <RotateTransform Angle="90"

CenterX="150"
```



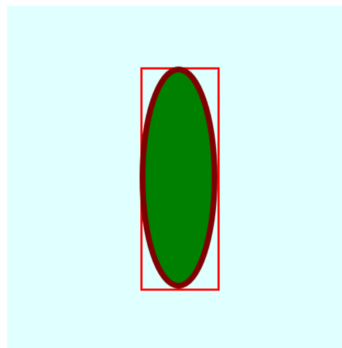
```
                              CenterY="50"
/>
          </EllipseGeometry.Transform>
        </EllipseGeometry>
      </Path.Data>
    </Path>
  </Border>
</Grid>
```

Но это тоже не выглядит правильным! Что произошло?

EllipseGeometry определяет эллипс, верхний левый угол ограничивающего прямоугольника которого располагается в точке (0, 0), а нижний правый угол – в точке (300, 100). Этот эллипс поворачивается на 90° вокруг точки (150, 50). Верхний левый угол ограничивающего прямоугольника развернутого эллипса располагается в точке (100, –100), и нижний правый угол – в точке (200, 200). Элемент *Border* занимает квадрат со стороной 200 пикселов, чтобы вместить нижний правый угол, но точки с отрицательными координатами оказываются вне *Border*.

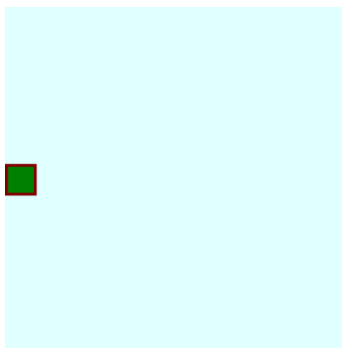
Чтобы все выполнялось «правильно», центр вращения необходимо задать в точке (50, 50):

```
<Grid Background="LightCyan">
  <Border BorderBrush="Red"
          BorderThickness="3"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">
    <Path Stroke="Maroon"
          StrokeThickness="8"
          Fill="Green">
      <Path.Data>
        <EllipseGeometry Center="150 50"
                         RadiusX="150"
                         RadiusY="50">
          <EllipseGeometry.Transform>
            <RotateTransform Angle="90"
                              CenterX="50"
                              CenterY="50"
/>
          </EllipseGeometry.Transform>
        </EllipseGeometry>
      </Path.Data>
    </Path>
  </Border>
</Grid>
```



Еще одно отличие между свойством *RenderTransform* элемента *Path* и свойством *Transform* объекта *Geometry* выявляется при использовании *ScaleTransform*. Начнем с небольшого прямоугольника, выровненного по левому краю:

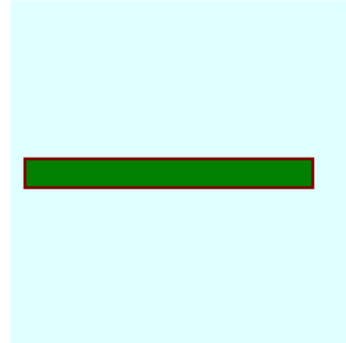
```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
          StrokeThickness="4"
          Fill="Green">
    <Path.Data>
      <RectangleGeometry
          Rect="2 220 40 40" />
    </Path.Data>
  </Path>
</Grid>
```



На самом деле я поместил *RectangleGeometry* в точке (2, 220), учитывая значение *StrokeThickness* элемента *Path*, чтобы не создавалось впечатление, что отображаемый объект выходит за рамки своего контейнера.

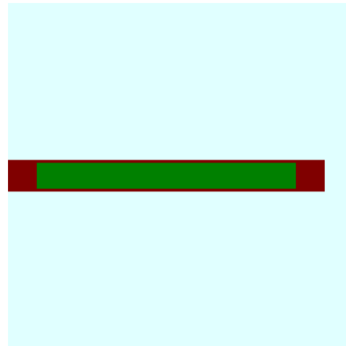
Теперь увеличим ширину элемента в 10 раз, применив *ScaleTransform* к *RectangleGeometry*:

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="4"
        Fill="Green">
    <Path.Data>
      <RectangleGeometry
          Rect="2 220 40 40">
        <RectangleGeometry.Transform>
          <ScaleTransform ScaleX="10" />
        </RectangleGeometry.Transform>
      </RectangleGeometry>
    </Path.Data>
  </Path>
</Grid>
```



Теперь ширина фигуры увеличилась в 10 раз, и объект *RectangleGeometry* выровнен по точке (20, 220). А вот если применить эту трансформацию к элементу *Path*, получаем совершенно другой эффект:

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="4"
        Fill="Green">
    <Path.Data>
      <RectangleGeometry
          Rect="2 220 40 40" />
    </Path.Data>
    <Path.RenderTransform>
      <ScaleTransform ScaleX="10" />
    </Path.RenderTransform>
  </Path>
</Grid>
```

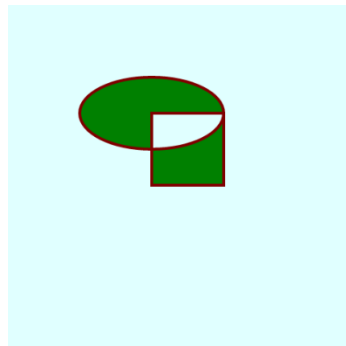


Теперь в 10 раз увеличилась толщина обводки справа и слева!

Группировка геометрических элементов

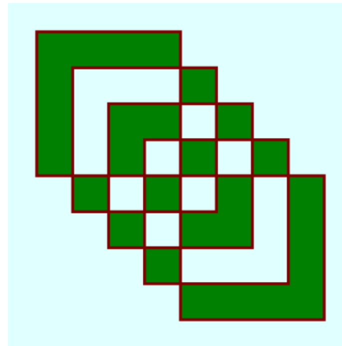
Среди классов, наследуемых от *Geometry*, есть класс *GeometryGroup* (Группа геометрических элементов). С его помощью мы получаем возможность комбинировать объекты *Geometry*.

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="4"
        Fill="Green">
    <Path.Data>
      <GeometryGroup>
        <EllipseGeometry Center="200 150"
                         RadiusX="100"
                         RadiusY="50" />
        <RectangleGeometry
            Rect="200 150 100 100" />
      </GeometryGroup>
    </Path.Data>
  </Path>
</Grid>
```



Обратите внимание, как применяется *FillRule* к этой комбинированной фигуре. И вот еще один пример:

```
<Grid Background="LightCyan">
  <Path Stroke="Maroon"
        StrokeThickness="4"
        Fill="Green">
    <Path.Data>
      <GeometryGroup>
        <RectangleGeometry
            Rect=" 40  40 200 200" />
        <RectangleGeometry
            Rect=" 90  90 200 200" />
        <RectangleGeometry
            Rect="140 140 200 200" />
        <RectangleGeometry
            Rect="190 190 200 200" />
        <RectangleGeometry
            Rect="240 240 200 200" />
      </GeometryGroup>
    </Path.Data>
  </Path>
</Grid>
```



Универсальный *PathGeometry*

LineGeometry, *RectangleGeometry*, *EllipseGeometry*, *GeometryGroup* – все это удобные особые случаи *PathGeometry* (Геометрический элемент контур), безусловно, самого универсального из всех производных класса *Geometry*. Классы *Path* и *PathGeometry* позволяют реализовывать любые задачи векторной графики, допустимые в Silverlight.

Сам *PathGeometry* определяет всего два свойства: уже знакомое нам *FillRule* и свойство *Figures* (Формы) типа *PathFigureCollection* (Коллекция контуров различной формы), который является коллекцией объектов *PathFigure* (Форма контура).

По существу *PathFigure* – это наборы соединенных прямых линий и кривых. Ключевое слово здесь *соединенных*. Объект *PathFigure* начинается в определенной точке, обозначенной свойством *StartPoint*, и продолжается серией соединенных сегментов.

Для этих соединенных сегментов класс *PathFigure* описывает свойство *Segments* (Сегменты) типа *PathSegmentCollection* (Коллекция сегментов контура), который является коллекцией объектов *PathSegment* (Сегмент контура). Как видно из иерархии классов, *PathSegment* – это абстрактный класс:

Object
- *DependencyObject* (абстрактный)
  - *PathSegment* (абстрактный)
    - *LineSegment* (запечатанный)
    - *PolyLineSegment* (запечатанный)
    - *ArcSegment* (запечатанный)
    - *QuadraticBezierSegment* (запечатанный)
    - *PolyQuadraticBezierSegment* (запечатанный)
    - *BezierSegment* (запечатанный)
    - *PolyQuadraticBezierSegment* (запечатанный)

PathFigure задает *StartPoint*. Первый объект *PathSegment* в коллекции *Segments* начинается в этой точке. Следующий объект *PathSegment* начинается в точке, в которой завершился первый *PathSegment*, и т.д.

Последняя точка последнего *PathSegment* в коллекции *Segments* может совпадать с начальной точкой (*StartPoint*) объекта *PathFigure*, а может и не совпадать. Чтобы *PathFigure*

гарантированно был замкнутым, можно задать свойство *IsClosed* (Замкнутый). В случае необходимости это обеспечит отрисовку прямой линии из последней точки последнего *PathSegment* в *StartPoint* объекта *PathFigure*.

PathFigure также определяет свойство *IsFilled*, имеющее значение *true* по умолчанию. Это свойство не зависит ни от одной кисти *Fill*, задаваемой для *Path*. Оно используется для вырезания (clipping) и проверки совпадения (hit-testing). В некоторых случаях Silverlight вопреки пожеланиям разработчика может предполагать, что область закрашена для целей вырезания и проверки совпадения. В этом случае задавайте свойству *IsFilled* значение *false*.

Итак, объект *PathGeometry* – это коллекция объектов *PathFigure*. Каждый объект *PathFigure* является серией соединенных прямых линий или кривых, определяемых коллекцией объектов *PathSegment*.

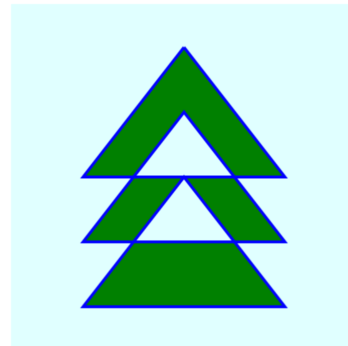
Остановимся на производных от *PathSegment* более подробно.

Класс *LineSegment* (Сегмент линия) описывает всего одно собственное свойство: *Point* типа *Point*. Объекту *LineSegment* необходим всего один объект *Point*, потому что он отрисовывает линию из точки, заданной свойством *StartPoint* объекта *PathFigure* (если *LineSegment* является первым сегментом в коллекции), или из конечной точки предыдущего сегмента.

Класс *PolyLineSegment* (Сегмент полилиния) определяет свойство *Points* типа *PointCollection* для отрисовки ряда соединенных прямых линий.

Рассмотрим *PathGeometry* с тремя объектами *PathFigure*, включающими 3, 2 и 1 объект *PathSegment*, соответственно:

```
<Grid Background="LightCyan">
  <Path Stroke="Blue"
        StrokeThickness="4"
        Fill="Green">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="240 60">
          <LineSegment Point="380 240" />
          <LineSegment Point="100 240" />
          <LineSegment Point="240 60" />
        </PathFigure>

        <PathFigure StartPoint="240 150"
                    IsClosed="True">
          <LineSegment Point="380 330" />
          <LineSegment Point="100 330" />
        </PathFigure>

        <PathFigure StartPoint="240 240"
                    IsClosed="True">
          <PolyLineSegment
              Points="380 420, 100 420"
/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```



Вторая и третья фигуры замкнуты явно посредством свойства *IsClosed*, но закрашены все три коллекции *PathFigure*.

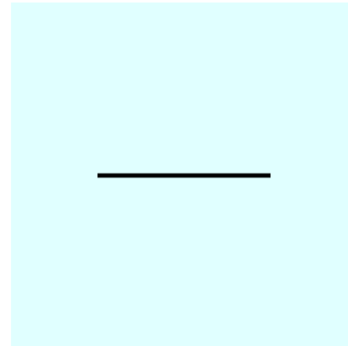
Класс *ArcSegment*

Сложности возникают с классом *ArcSegment* (Сегмент дуга). Дуга – это лишь часть эллипса, но поскольку *ArcSegment* должен соответствовать парадигме начальных и конечных точек, он должен определяться двумя точками, лежащими на контуре некоторого эллипса. Но если эллипс задается посредством центра и радиуса, как найти точные координаты точки его контура без тригонометрических вычислений?

Выход – задавать только *размер* эллипса, без его позиционирования. Фактическое местоположение эллипса определяется двумя точками.

Полагаю, необходимо привести конкретный пример. Проведем линию из точки (120, 240) в точку (360, 240).

```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />
</Grid>
```

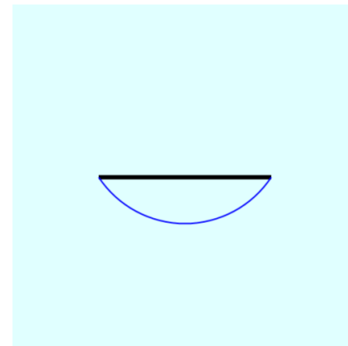


Я взял эту линию просто для наглядности и хочу нарисовать дугу между этими же двумя точками. Очевидно, что между этими точками можно построить бесконечное число дуг, но для любого конкретного эллипса существует только четыре возможных варианта.

Продемонстрируем это.

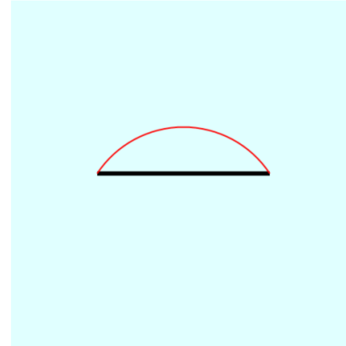
Предположим, мы хотим соединить эти две точки дугой, являющейся часть окружности с радиусом 144 пиксела. В данном фрагменте показано, как задать *ArcSegment* такого размера между точками (120, 240) и (360, 240):

```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Blue"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120
240">
          <ArcSegment Point="360 240"
                      Size="144 144" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```



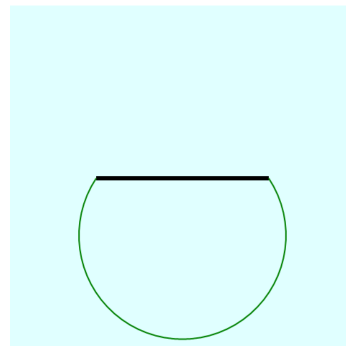
По умолчанию дуга отрисовывается из начальной точки в конечную точку в направлении против часовой стрелки. Это поведение можно переопределить, задав свойству *SweepDirection* (Направление развертывания) значение *Clockwise* (По часовой стрелке):

```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Red"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              SweepDirection="Clockwise"
/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```



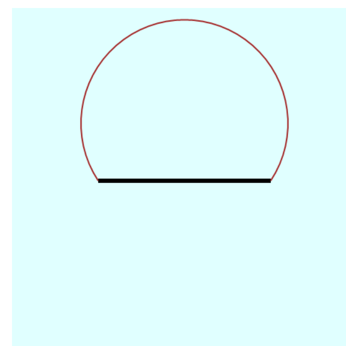
В обоих случаях между двумя точками отрисовывается меньшая из двух возможных дуг. На рисунке проиллюстрирован результат, получаемый, когда свойство *SweepDirection* имеет значение по умолчанию *CounterClockwise*, а свойству *IsLargeArc* (Большая дуга) задано значение *true*:

```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Green"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              IsLargeArc="True" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```



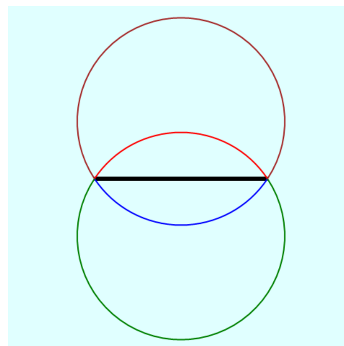
Наконец, зададим свойству *IsLargeArc* значение *true* и *SweepDirection* значение *Clockwise*:

```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Brown"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              IsLargeArc="True"

SweepDirection="Clockwise"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
```



```
  </Path>
</Grid>
```

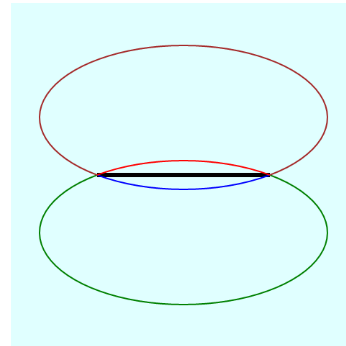
Чтобы лучше понять, что происходит (по крайней мере, на концептуальном уровне), отрисуем окружность целиком. Это будет окружность определенного размера, позиционированная так, что она проходит по двум заданным точкам. Существует два варианта позиционирования окружности по двум точкам, и описана она может быть по и против часовой стрелки:



```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Blue"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Red"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              SweepDirection="Clockwise"
/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Green"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              IsLargeArc="True" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Brown"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="144 144"
              IsLargeArc="True"
              SweepDirection="Clockwise"
/>
```

```
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```

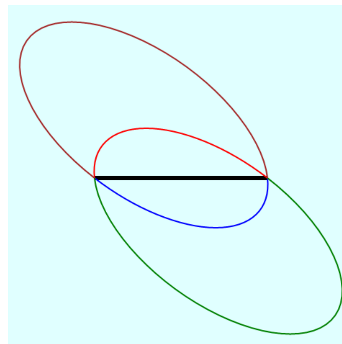
Это справедливо и для эллипса. Следующая разметка аналогична предыдущему примеру, только свойству *Size* объекта *ArcSegment* задано значение не (144, 144), а (200, 100):



```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Blue"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Red"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              SweepDirection="Clockwise"
/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Green"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              IsLargeArc="True" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Brown"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              IsLargeArc="True"
              SweepDirection="Clockwise"
/>
        </PathFigure>
```

```
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```

Если требуется построит дугу на основании эллипса, оси которого не лежат на вертикали и горизонтали, можно задать еще одно, последнее, свойство *ArcSegment* – *RotationAngle* (Угол поворота).



```
<Grid Background="LightCyan">
  <Polyline Points="120 240, 360 240"
            Stroke="Black"
            StrokeThickness="6" />

  <Path Stroke="Blue"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              RotationAngle="36" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Red"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              SweepDirection="Clockwise"
              RotationAngle="36" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Green"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              IsLargeArc="True"
              RotationAngle="36" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>

  <Path Stroke="Brown"
        StrokeThickness="2">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint="120 240">
          <ArcSegment
              Point="360 240"
              Size="200 100"
              IsLargeArc="True"
              SweepDirection="Clockwise"
```

```
                RotationAngle="36" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
  </Path>
</Grid>
```

Точки окружности, ограничивающие дугу, можно определить в коде алгоритмически. Эта идея положена в основу приложения SunnyDay (Солнечный день), в котором путем сочетания объектов *LineSegment* и *ArcSegment* построено сложное изображение. Экземпляр элемента *Path* создается в файле MainPage.xaml, и задаются значения всех его свойств, кроме самих сегментов:

Проект Silverlight: SunnyDay Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path Fill="Yellow">
        <Path.Data>
            <PathGeometry>
                <PathFigure x:Name="pathFigure"
                            IsClosed="True" />
            </PathGeometry>
        </Path.Data>
    </Path>
</Grid>
```

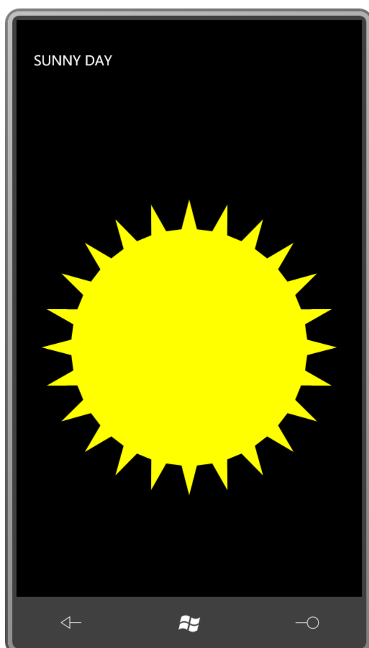
После этого обработчик события *Loaded* должен получить размер области содержимого и вычислить все координаты всех сегментов контура:

Проект Silverlight: SunnyDay Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    const int BEAMCOUNT = 24;
    const double INCREMENT = Math.PI / BEAMCOUNT;

    public MainPage()
    {
        InitializeComponent();
        Loaded += OnLoaded;
    }

    void OnLoaded(object sender, RoutedEventArgs args)
    {
        Point center = new Point(ContentPanel.ActualWidth / 2,
                                 ContentPanel.ActualHeight / 2);

        double radius = 0.45 * Math.Min(ContentPanel.ActualWidth,
                                        ContentPanel.ActualHeight);
        double innerRadius = 0.8 * radius;

        for (int i = 0; i < BEAMCOUNT; i++)
        {
            double radians = 2 * Math.PI * i / BEAMCOUNT;

            if (i == 0)
            {
                pathFigure.StartPoint = new Point(center.X, center.Y - radius);
            }

            LineSegment lineSeg = new LineSegment();
            lineSeg.Point = new Point(
                center.X + innerRadius * Math.Sin(radians + INCREMENT / 2),
```

```
                center.Y - innerRadius * Math.Cos(radians + INCREMENT / 2));
            pathFigure.Segments.Add(lineSeg);

            ArcSegment arcSeg = new ArcSegment();
            arcSeg.Point = new Point(
                center.X + innerRadius * Math.Sin(radians + 3 * INCREMENT / 2),
                center.Y - innerRadius * Math.Cos(radians + 3 * INCREMENT / 2));
            pathFigure.Segments.Add(arcSeg);

            lineSeg = new LineSegment();
            lineSeg.Point = new Point(
                center.X + radius * Math.Sin(radians + 2 * INCREMENT),
                center.Y - radius * Math.Cos(radians + 2 * INCREMENT));
            pathFigure.Segments.Add(lineSeg);
        }
    }
}
```

В результате получаем вот такое очень мультипликационное солнышко:



Кривые Безье

Пьер Этьен Безье (1910–1999) работал инженером французской автомобилестроительной компании Рено с 1933 по 1975 год. В 1960-е годы компания начала переход от создания кузовов автомобилей с использованием глиняных моделей к компьютеризированным средствам проектирования. Для этого потребовались математические описания кривых, с которыми инженеры могли бы работать, не вникая в математические дебри. Результатом этих исканий стали кривые, которые теперь носят имя Пьера Безье.

Кривая Безье – это сплин, т.е. графическое представление гладкой непрерывной функции, являющейся аппроксимацией дискретных данных. Silverlight поддерживает не только стандартную двухмерную форму кубической кривой Безье, но также и квадратичные кривые Безье, которые проще и быстрее обрабатываются, поэтому начнем разговор с них.

Кривая Безье второго порядка (или квадратичная) определяется тремя точками, которые обычно обозначают p_0, p_1 и p_2. Кривая начинается в точке p_0 и заканчивается в точке p_2. Точку p_1 называют *опорной точкой*. Кривая обычно не проходит через p_1, эта точка играет

роль своеобразного магнита, который притягивает кривую к себе. Из p_0 кривая выходит по касательной и в направлении прямой, проведенной из p_0 в p_1; в точке p_2 кривая проходит по касательной и в направлении прямой, проведенной из p_1 в p_2.

Лучший способ понять кривые Безье – поэкспериментировать с ними. Приложение QuadraticBezier (Квадратичная кривая Безье) отрисовывает всего одну кривую Безье, но позволяет задавать разные три точки и наблюдать, к чему это приводит.

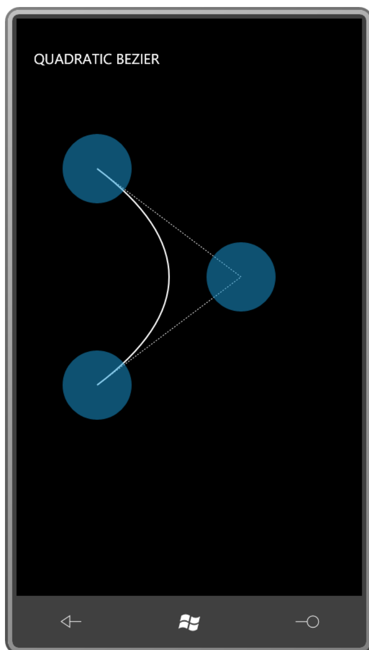
XAML-файл включает четыре элемента *Path* и *Polyline*, которые располагаются в *Grid* с одной ячейкой. Первый *Path* представляет квадратичную кривую Безье. Обратите внимание, что p_0 обеспечивается свойством *StartPoint* объекта *PathFigure*, тогда как p_1 и p_2 соответствуют свойствам *Point1* и *Point2* объекта *QuadraticBezierSegment* (Сегмент-квадратичная кривая Безье):

Проект Silverlight: QuadraticBezier Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path Stroke="{StaticResource PhoneForegroundBrush}"
          StrokeThickness="2">
        <Path.Data>
            <PathGeometry>
                <PathFigure x:Name="pathFig"
                            StartPoint="100 100">
                    <QuadraticBezierSegment x:Name="pathSeg"
                                            Point1="300 250"
                                            Point2="100 400" />
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>

    <Polyline Name="ctrlLine"
              Stroke="{StaticResource PhoneForegroundBrush}"
              StrokeDashArray="2 2"
              Points="100 100, 300 250, 100 400" />

    <Path Name="pt0Dragger"
          Fill="{StaticResource PhoneAccentBrush}"
          Opacity="0.5">
        <Path.Data>
            <EllipseGeometry x:Name="pt0Ellipse"
                             Center="100 100"
                             RadiusX="48"
                             RadiusY="48" />
        </Path.Data>
    </Path>

    <Path Name="pt1Dragger"
          Fill="{StaticResource PhoneAccentBrush}"
          Opacity="0.5">
        <Path.Data>
            <EllipseGeometry x:Name="pt1Ellipse"
                             Center="300 250"
                             RadiusX="48"
                             RadiusY="48" />
        </Path.Data>
    </Path>

    <Path Name="pt2Dragger"
          Fill="{StaticResource PhoneAccentBrush}"
          Opacity="0.5">
        <Path.Data>
            <EllipseGeometry x:Name="pt2Ellipse"
                             Center="100 400"
```

```
                                RadiusX="48"
                                RadiusY="48" />
        </Path.Data>
    </Path>
</Grid>
```

Элемент *Polyline* отрисовывает пунктирную линию, соединяющую две конечные точки и опорную точку. Остальные три элемента можно перетягивать по экрану, т.е. они позволяют пользователю перемещать все три точки. Изначально на экран выводится следующее:



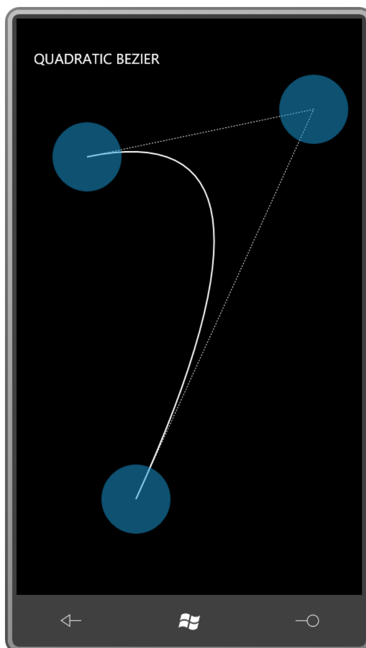
Всю логику перетягивания обеспечивает файл выделенного кода. Поскольку Silverlight for Windows Phone поддерживает привязки только для свойств, которые описаны классами, наследуемыми от *FrameworkElement*, я не смог связать соответствующие точки в XAML-файле. Каждую из них приходится задавать по отдельности в перегрузках *Manipulation*:

Проект Silverlight: QuadraticBezier Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    if (args.OriginalSource == pt0Dragger ||
        args.OriginalSource == pt1Dragger ||
        args.OriginalSource == pt2Dragger)
    {
        args.ManipulationContainer = ContentPanel;
        args.Handled = true;
    }
    base.OnManipulationStarted(args);
}

protected override void OnManipulationDelta(ManipulationDeltaEventArgs args)
{
    Point translate = args.DeltaManipulation.Translation;

    if (args.OriginalSource == pt0Dragger)
    {
        pathFig.StartPoint = Move(pathFig.StartPoint, translate);
        ctrlLine.Points[0] = Move(ctrlLine.Points[0], translate);
        pt0Ellipse.Center = Move(pt0Ellipse.Center, translate);
        args.Handled = true;
```

```
    }
    else if (args.OriginalSource == pt1Dragger)
    {
        pathSeg.Point1 = Move(pathSeg.Point1, translate);
        ctrlLine.Points[1] = Move(ctrlLine.Points[1], translate);
        pt1Ellipse.Center = Move(pt1Ellipse.Center, translate);
        args.Handled = true;
     }
     else if (args.OriginalSource == pt2Dragger)
     {
        pathSeg.Point2 = Move(pathSeg.Point2, translate);
        ctrlLine.Points[2] = Move(ctrlLine.Points[2], translate);
        pt2Ellipse.Center = Move(pt2Ellipse.Center, translate);
        args.Handled = true;
     }
     base.OnManipulationDelta(args);
}

Point Move(Point point, Point translate)
{
    return new Point(point.X + translate.X, point.Y + translate.Y);
}
```

Являясь квадратичной, эта версия кривой Безье имеет только один изгиб и отличается регулярностью поведения:



Для построения квадратичных кривых Безье используются следующие параметрические уравнения:

$$x(t) = (1-t)^2 x_0 + 2t(1-t)x_1 + t^2 x_2$$

$$y(t) = (1-t)^2 y_0 + 2t(1-t)y_1 + t^2 y_2$$

для t = 0 до 1, где $p_0 = (x_0, y_0)$ и т.д.

Кубический сплайн Безье считается более стандартным и имеет две опорные точку, а не одну. Кривая определяется четырьмя точками, которые обычно называют p_0, p_1, p_2 и p_3. Кривая начинается в точке p_0 и заканчивается в p_3. Из p_0 кривая выходит по касательной и в направлении прямой, проведенной из p_0 в p_1; в точке p_3 кривая проходит по касательной к

прямой, проведенной из p_3 в p_2. Эту кривую описывают следующие параметрические уравнения:

$$x(t) = (1-t)^3 x_0 + 3t(1-t)^2 x_1 + 3t^2(1-t) x_2 + t^3 x_3$$

$$y(t) = (1-t)^3 y_0 + 3t(1-t)^2 y_1 + 3t^2(1-t) y_2 + t^3 y_3$$

В приложении CubicBezier (Кубическая кривая Безье) мною был использован несколько другой подход. Чтобы все упростить, я описал производный от *UserControl* класс *PointDragger* (Модуль перетягивания точки). Файл PointDragger.xaml определяет дерево визуальных элементов. В него входит лишь *Grid*, имеющий *Path* с непрозрачностью (*Opacity*) 0,5, и *EllipseGeometry* без точки *Center*:

Проект Silverlight: CubicBezier Файл: PointDragger.xaml

```
<UserControl
    x:Class="CubicBezier.PointDragger"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Grid x:Name="LayoutRoot">
        <Path Fill="{StaticResource PhoneAccentBrush}"
              Opacity="0.5">
            <Path.Data>
                <EllipseGeometry x:Name="ellipseGeometry"
                                 RadiusX="48"
                                 RadiusY="48" />
            </Path.Data>
        </Path>
    </Grid>
</UserControl>
```

В файле выделенного кода описывается свойство-зависимость *Point* типа *Point* и событие *PointChanged* (Точка изменилась), которое формируется при изменении значения этого свойства. Обработчик события изменения свойства также задает значение *EllipseGeometry*, описанному в XAML-файле:

Проект Silverlight: CubicBezier Файл: PointDragger.xaml.cs (фрагмент)

```
public partial class PointDragger : UserControl
{
    public static readonly DependencyProperty PointProperty =
        DependencyProperty.Register("Point",
            typeof(Point),
            typeof(PointDragger),
            new PropertyMetadata(OnPointChanged));
     public event RoutedPropertyChangedEventHandler<Point> PointChanged;

    public PointDragger()
    {
        InitializeComponent();
    }

    public Point Point
    {
        set { SetValue(PointProperty, value); }
        get { return (Point)GetValue(PointProperty); }
    }

    …

    static void OnPointChanged(DependencyObject obj,
```

```
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as PointDragger).OnPointChanged((Point)args.OldValue,
                                              (Point)args.NewValue);
    }

    protected virtual void OnPointChanged(Point oldValue, Point newValue)
    {
        ellipseGeometry.Center = newValue;

        if (PointChanged != null)
            PointChanged(this,
                new RoutedPropertyChangedEventArgs<Point>(oldValue, newValue));
    }
}
```

Класс *PointDragger* также обрабатывает собственные события *Manipulation*, которые по сравнению с описываемыми в QuadraticBezier очень просты:

Проект Silverlight: CubicBezier Файл: PointDragger.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    args.ManipulationContainer = VisualTreeHelper.GetParent(this) as UIElement;
    args.Handled = true;
    base.OnManipulationStarted(args);
}

protected override void OnManipulationDelta(ManipulationDeltaEventArgs args)
{
    Point translate = args.DeltaManipulation.Translation;
    this.Point = new Point(this.Point.X + translate.X, this.Point.Y + translate.Y);
    args.Handled = true;
    base.OnManipulationDelta(args);
}
```

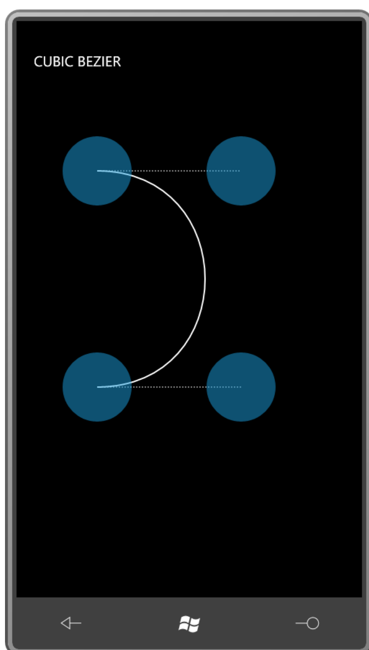
В файле MainPage.xaml описан *Path* с *BezierSegment* (Сегмент кривая Безье), два элемента *Polyline* для отрисовки касательных и четыре экземпляра *PointDragger*. Класс *BezierSegment* определяет свойства *Point1*, *Point2* и *Point3* для задания двух опорных точек и конечной точки.

Проект Silverlight: CubicBezier Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path Stroke="{StaticResource PhoneForegroundBrush}"
          StrokeThickness="2">
        <Path.Data>
            <PathGeometry>
                <PathFigure x:Name="pathFig"
                            StartPoint="100 100">
                    <BezierSegment x:Name="pathSeg"
                                   Point1="300 100"
                                   Point2="300 400"
                                   Point3="100 400" />
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>

    <Polyline Name="ctrl1Line"
              Stroke="{StaticResource PhoneForegroundBrush}"
              StrokeDashArray="2 2"
              Points="100 100, 300 100" />
```

```
    <Polyline Name="ctrl2Line"
              Stroke="{StaticResource PhoneForegroundBrush}"
              StrokeDashArray="2 2"
              Points="300 400, 100 400" />

    <local:PointDragger x:Name="pt0Dragger"
                        Point="100 100"
                        PointChanged="OnPointDraggerPointChanged" />

    <local:PointDragger x:Name="pt1Dragger"
                        Point="300 100"
                        PointChanged="OnPointDraggerPointChanged" />

    <local:PointDragger x:Name="pt2Dragger"
                        Point="300 400"
                        PointChanged="OnPointDraggerPointChanged" />

    <local:PointDragger x:Name="pt3Dragger"
                        Point="100 400"
                        PointChanged="OnPointDraggerPointChanged" />
</Grid>
```

Изначально на экран выводится следующее:



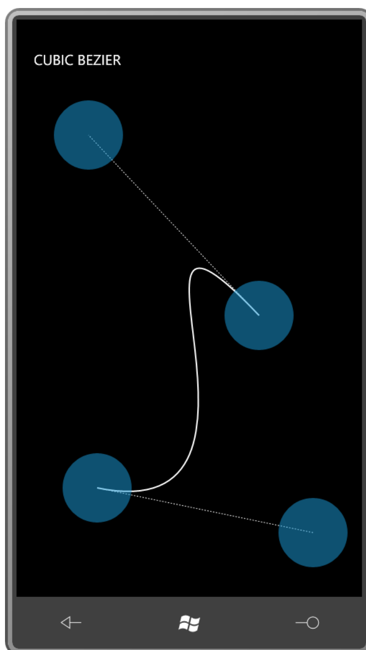
Обратите внимание на обработчики событий *PointChanged* элементов управления *PointDragger*. Реализация этих обработчиков – это практически все, что осталось сделать в MainPage.xaml.cs:

Проект Silverlight: CubicBezier Файл: MainPage.xaml.cs (фрагмент)

```
void OnPointDraggerPointChanged(object sender,
                                RoutedPropertyChangedEventArgs<Point> args)
{
    Point translate = new Point(args.NewValue.X - args.OldValue.X,
                                args.NewValue.Y - args.OldValue.Y);

    if (sender == pt0Dragger)
    {
        pathFig.StartPoint = Move(pathFig.StartPoint, translate);
```

```
        ctrl1Line.Points[0] = Move(ctrl1Line.Points[0], translate);
    }
    else if (sender == pt1Dragger)
    {
        pathSeg.Point1 = Move(pathSeg.Point1, translate);
        ctrl1Line.Points[1] = Move(ctrl1Line.Points[1], translate);
    }
    else if (sender == pt2Dragger)
    {
        pathSeg.Point2 = Move(pathSeg.Point2, translate);
        ctrl2Line.Points[0] = Move(ctrl2Line.Points[0], translate);
    }
    else if (sender == pt3Dragger)
    {
        pathSeg.Point3 = Move(pathSeg.Point3, translate);
        ctrl2Line.Points[1] = Move(ctrl2Line.Points[1], translate);
    }
}

Point Move(Point point, Point translate)
{
    return new Point(point.X + translate.X, point.Y + translate.Y);
}
```

В ходе экспериментов с данным приложением можно заметить, что кривые никогда не выходят за рамки четырехугольника, определенного двумя конечными и двумя опорными точками. (Его называют «выпуклой оболочкой» окружностей Безье.) В данном случае используется кубическая кривая Безье, поэтому она может изгибаться дважды:



Кроме классов *QuadraticBezierSegment* и *BezierSegment* для описания одиночных кривых Безье могут также использоваться *PolyQuadraticBezierSegment* (Сегмент множество квадратичных кривых Безье) и *PolyBezierSegment* (Сегмент множество кривых Безье), описывающие множества кривых Безье. Каждая новая кривая берет начало в точке окончания предыдущей кривой. В обоих классах есть свойство *Points* типа *PointCollection*.

Для PolyQuadraticBezierSegment количество объектов *Point* в коллекции *Points* должно быть кратно 2. Первый, третий, пятый и все последующие нечетные члены коллекции являются опорным точками. Для *PolyBezierSegment* количество точек должно быть кратно 3.

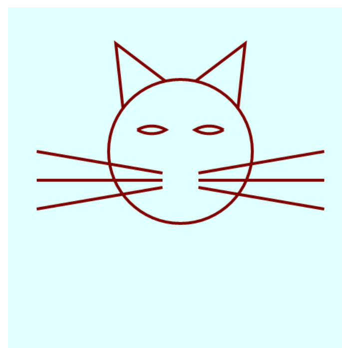
При соединении множества кривых Безье конечная точка одной кривой становится начальной точкой последующей кривой. Результирующая кривая будет гладкой в точке соединения только в случае, если две опорные точки обеих соединяемых кривых являются коллинеарными, т.е. лежат на одной прямой.

Синтаксис разметки контура

Silverlight поддерживает некоторого рода «мини-язык», который позволяет описывать весь *PathGeometry* в строке. В этом языке объекты *PathFigure* и *PathSegment* замещены буквами (такие как M, что означает Move (Перемещение), L – Line (Линия), A – Arc (Дуга), и C – Cubic Bézier (Кубическая кривая Безье)). Каждый новый *PathFigure* начинается с команды Move. Этот синтаксис описан в разделе Graphics (Графические элементы) в онлайн-документации по Silverlight.

Рассмотрим пример:

```
<Grid Background="LightCyan">
    <Path Stroke="Maroon"
          StrokeThickness="4"
          Data="M 160 140 L 150 50 220
103
                M 320 140 L 330 50 260
103
                M 215 230 L 40 200
                M 215 240 L 40 240
                M 215 250 L 40 280
                M 265 230 L 440 200
                M 265 240 L 440 240
                M 265 250 L 440 280
                M 240 100
                A 100 100 0 0 1 240 300
                A 100 100 0 0 1 240 100
                M 180 170
                A 40 40 0 0 1 220 170
                A 40 40 0 0 1 180 170
                M 300 170
                A 40 40 0 0 1 260 170
                A 40 40 0 0 1 300 170" />
</Grid>
```



Самым сложным является синтаксис описания дуги. Он начинается с задания размера эллипса, за которым следуют угол поворота и два флага: 1 для *IsLargeArc* и 1 для *Clockwise*. В конце задаются координаты точки. Для отрисовки замкнутой окружности ее разделяют на две половины и используют две команды Arc (или два объекта *ArcSegment*).

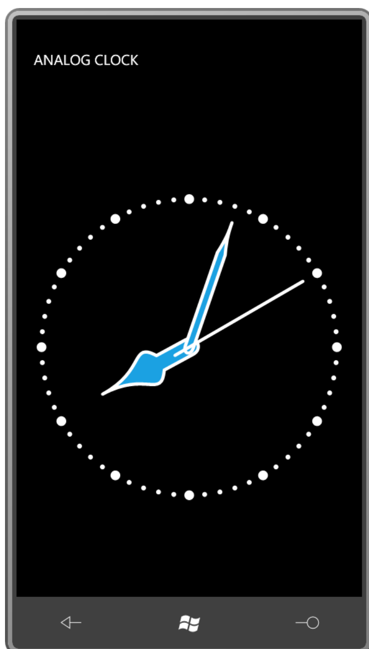
Геометрические элементы могут использоваться не только для рисования, но и для вырезания. Рассмотрим знаменитое изображение KeyholeOnTheMoon (Взгляд на Луну через замочную скважину):

В данном приложении используется свойство *Clip* (Вырезанное изображение) типа *Geometry*. Свойство *Clip* описано классом *FrameworkElement* и позволяет сделать визуально непрямоугольным любой элемент или элемент управления, а благодаря Path Markup Syntax (Синтаксис разметки контура) эта задача становится совершенно тривиальной:

Проект Silverlight: KeyholeOnTheMoon Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      Background="{StaticResource PhoneAccentBrush}">
    <Image Source="Images/BuzzAldrinOnTheMoon.png"
           Stretch="None"
           Clip="M 120 95 L 90 265 L 220 265 L 190 95
                          A 50 50 0 1 0 120 95" />
</Grid>
```

Я также использовал Silverlight Path Markup Syntax в приложении Analog Clock (Аналоговые часы). Вот как они выглядят:



Визуальное представление образовано пятью элементами *Path*. Кривые на часовой и минутной стрелках – это сплайны Безье. Штриховые метки циферблата – это отрисованные пунктиром дуговые сегменты.

XAML-файл определяет *Style*, применяемый ко всем пяти элементам *Path*:

Проект Silverlight: AnalogClock Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="pathStyle"
           TargetType="Path">
        <Setter Property="Fill" Value="{StaticResource PhoneAccentColor}" />
        <Setter Property="Stroke" Value="{StaticResource PhoneForegroundColor}" />
        <Setter Property="StrokeThickness" Value="2" />
        <Setter Property="StrokeStartLineCap" Value="Round" />
        <Setter Property="StrokeEndLineCap" Value="Round" />
        <Setter Property="StrokeLineJoin" Value="Round" />
        <Setter Property="StrokeDashCap" Value="Round" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Чтобы не усложнять, я выбрал произвольную систему координат. Графические элементы часов отрисовываются, исходя из предположения о том, что ширина и высота циферблата составляет 200 пикселов, и центр располагается в точке (0, 0). Таким образом, мы имеем диапазон допустимых значений слева направо (по оси X) от -100 до 100 и сверху вниз (по оси Y) – от -100 до 100.

Частично эти произвольные координаты часов определены явно заданными *Width* и *Height* вложенного *Grid*:

Проект Silverlight: AnalogClock Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      SizeChanged="OnContentPanelSizeChanged">

    <!-- Отрисовываем часы на Grid с центром в точке (0, 0) -->
    <Grid Width="200" Height="200">
```

```
        <Grid.RenderTransform>
            <TransformGroup>
                <ScaleTransform x:Name="scaleClock" />
                <TranslateTransform X="100" Y="100" />
            </TransformGroup>
        </Grid.RenderTransform>

        …

    </Grid>
</Grid>
```

Трансформация *TranslateTransform* смещает весь *Grid* вправо и вниз. Например, верхний левый угол, который располагался в точке (–100, –100), смещается в точку с координатами (0, 0); а точка с координатами (100, 100) перемещается в точку с координатами (200, 200).

Обратите внимание на обработчик событий *SizeChanged*, заданный в обычной сетке для содержимого. В коде для *ScaleTransform*, который применятся к вложенному *Grid*, используется фактический размер области содержимого. Это приводит произвольно выбранный размер в 200 пикселов к фактическому размеру:

Проект Silverlight: AnalogClock Файл: MainPage.xaml.cs (фрагмент)

```
void OnContentPanelSizeChanged(object sender, SizeChangedEventArgs args)
{
    double scale = Math.Min(args.NewSize.Width, args.NewSize.Height) / 200;
    scaleClock.ScaleX = scale;
    scaleClock.ScaleY = scale;
}
```

Рассмотрим пять контуров:

Проект Silverlight: AnalogClock Файл: MainPage.xaml (фрагмент)

```
<!-- Деления на циферблате (маленькие и большие). -->
<Path Data="M 0 -90 A 90 90 0 1 1 0 90
                    A 90 90 0 1 1 0 -90"
      Style="{StaticResource pathStyle}"
      Fill="{x:Null}"
      StrokeDashArray="0 3.14159"
      StrokeThickness="3" />

<Path Data="M 0 -90 A 90 90 0 1 1 0 90
                    A 90 90 0 1 1 0 -90"
      Style="{StaticResource pathStyle}"
      Fill="{x:Null}"
      StrokeDashArray="0 7.854"
      StrokeThickness="6" />

<!-- Часовая стрелка, указывающая на 12 часов. -->
<Path Data="M 0 -60 C 0 -30, 20 -30, 5 -20 L 5 0
                    C 5 7.5, -5 7.5, -5 0 L -5 -20
                    C -20 -30, 0 -30 0 -60"
      Style="{StaticResource pathStyle}">
    <Path.RenderTransform>
        <RotateTransform x:Name="rotateHour" />
    </Path.RenderTransform>
</Path>

<!-- Минутная стрелка, указывающая на 12 часов. -->
<Path Data="M 0 -80 C 0 -75, 0 -70, 2.5 -60 L 2.5 0
```

```
                    C 2.5 5, -2.5 5, -2.5 0 L -2.5 -60
                    C 0 -70, 0 -75, 0 -80"
      Style="{StaticResource pathStyle}">
    <Path.RenderTransform>
        <RotateTransform x:Name="rotateMinute"  />
    </Path.RenderTransform>
</Path>

<!-- Секундная стрелка, указывающая на 12 часов. -->
<Path Data="M 0 10 L 0 -80"
      Style="{StaticResource pathStyle}">
    <Path.RenderTransform>
        <RotateTransform x:Name="rotateSecond"  />
    </Path.RenderTransform>
</Path>
```

Параметры *StrokeDashArray* первых двух элементов *Path* тщательно просчитаны для создания шаблона делений на циферблате для 1 и 5 секунд. В качестве значений свойств *RenderTransform* остальных трех элементов *Path* заданы объекты *RotateTransform*. Эти объекты *RotateTransform* задаются повторно каждую секунду из файла выделенного кода:

Проект Silverlight: AnalogClock Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();

        DispatcherTimer tmr = new DispatcherTimer();
        tmr.Interval = TimeSpan.FromSeconds(1);
        tmr.Tick += new EventHandler(OnTimerTick);
        tmr.Start();
    }

    void OnTimerTick(object sender, EventArgs args)
    {
        DateTime dt = DateTime.Now;

        rotateSecond.Angle = 6 * dt.Second;
        rotateMinute.Angle = 6 * dt.Minute + rotateSecond.Angle / 60;
        rotateHour.Angle = 30 * (dt.Hour % 12) + rotateMinute.Angle / 12;
    }
    …
}
```

Как создавалась данная глава

Небольшие фрагменты XAML и сопровождающие их иллюстрации своим появлением обязаны концепции, которая возникла на истоке зарождения Windows Presentation Foundation. Хотя XAML разрабатывался преимущественно на основании предположения, что он будет компилироваться вместе с остальным исходным кодом, справедливым было допустить, что у разработчиков может возникнуть желание конвертировать XAML в объекты (и объекты в XAML) во время выполнения. Для этой цели в пространстве имен *System.Windows.Markup* имеется статический метод *XamlReader.Load* для конвертирования XAML в объекты и статический метод *XamlWriter.Save* для обратной операции.

Только первый из этих двух статических методов перешел в Silverlight и Silverlight for Windows Phone. Но он на самом деле очень полезен. Передайте в *XamlReader.Load* строку, содержащую действительный XAML (вместе с объявлениями необходимых пространств

имен, но без задания обработчиков событий), и метод возвратит объект, соответствующий корневому элементу, который будет также включать все остальные объекты дерева визуальных элементов.

Одним из применений *XamlReader.Load* в WPF был интерактивный инструмент разработки. Он включал *TextBox*, который позволял вводить и редактировать XAML, и выводил на экран результирующий объект. Конечно, большую часть времени в ходе редактирования XAML был недействительным, поэтому инструменту приходилось перехватывать возникающие ошибки и реагировать на них соответствующим образом.

Вышло несколько версий этого инструмента разработки. В состав WPF Software Development Kit входила версия под именем XamlPad. Для моей книги «*Applications = Code + Markup*»[1] (Майкрософт Пресс, 2006) я разработал версию XamlCruncher.

Позднее я усовершенствовал XamlCruncher, чтобы он мог представлять своего рода слайд-шоу небольших XAML-файлов и соответствующих им объектов. Я использовал это приложение в своей презентации о программировании на WPF и затем переписал его для Silverlight.

Библиотека Petzold.Phone.Silverlight включает «сущность» версии XamlCruncher для Windows Phone 7 в классе, производном от *TextBox*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: XamlCruncherTextBox.cs (фрагмент)

```
public class XamlCruncherTextBox : TextBox
{
    public event EventHandler<XamlCruncherEventArgs> XamlResult;

    public XamlCruncherTextBox()
    {
        this.AcceptsReturn = true;
        this.TextWrapping = TextWrapping.NoWrap;
        this.HorizontalScrollBarVisibility = ScrollBarVisibility.Auto;
        this.VerticalScrollBarVisibility = ScrollBarVisibility.Auto;

        TextChanged += OnTextBoxTextChanged;
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        string xaml =
            "<UserControl " +
            " xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'\r" +
            " xmlns:phone='clr-namespace:Microsoft.Phone.Controls;" +
            "assembly=Microsoft.Phone'\r" +
            " xmlns:shell='clr-namespace:Microsoft.Phone.Shell;" +
            "assembly=Microsoft.Phone'\r" +
            " xmlns:system='clr-namespace:System;assembly=mscorlib'\r" +
            " xmlns:petzold='clr-namespace:Petzold.Phone.Silverlight;" +
            "assembly=Petzold.Phone.Silverlight'>\r" +
            "     " + this.Text + "\r" +
            "</UserControl>";

        UserControl ctrl = null;

        try
        {
            ctrl = XamlReader.Load(xaml) as UserControl;
        }
        catch (Exception exc)
```

[1] Приложения = код + разметка (*прим. переводчика*).

```
        {
            OnXamlResult(new XamlCruncherEventArgs(exc.Message));
            return;
        }

        if (ctrl == null)
        {
            OnXamlResult(new XamlCruncherEventArgs("null result"));
            return;
        }

        OnXamlResult(new XamlCruncherEventArgs(ctrl));
    }

    protected virtual void OnXamlResult(XamlCruncherEventArgs args)
    {
        if (XamlResult != null)
                XamlResult(this, args);
    }
}
```

Обработчик *TextChanged* предполагает, что *TextBox* содержит фрагмент XAML, предназначенный как содержимое *UserControl*. Он заключает этот текст в теги *UserControl* и добавляет массу объявлений пространств имен – в том числе стандартных (и не вполне) *phone*, *shell*, *system* и *petzold* – и передает его в метод *XamlReader.Load*, который формирует исключение, если XAML недействительный.

В любом случае класс формирует событие *XamlResult* (Результат обработки XAML), предоставляя в аргументах события либо результирующий *UserControl*, либо сообщение об ошибке:

Проект Silverlight: Petzold.Phone.Silverlight Файл: XamlCruncherEventArgs.cs

```
using System;
using System.Windows;

namespace Petzold.Phone.Silverlight
{
    public class XamlCruncherEventArgs : EventArgs
    {
        public XamlCruncherEventArgs(UIElement element)
        {
            Element = element;
            Error = null;
        }

        public XamlCruncherEventArgs(string error)
        {
            Error = error;
            Element = null;
        }

        public UIElement Element { set; get; }

        public string Error { set; get; }
    }
}
```

Я использовал класс *XamlCruncherTextBox* в двух приложениях. Первое называлось просто XamlCruncher. Те кто отличается прилежанием и усердием, могут ввести XAML в это приложение на своем телефоне и увидеть результаты.

Прилежание и усердие потребуются для работы с клавиатурой. Независимо от того какая клавиатура используется, экранная или аппаратная, при вводе придется очень много переключаться между раскладками букв, цифр и символов. Например, на аппаратной клавиатуре, которую я использую при написании данной книги, нет угловых скобок и знака «равно», что абсолютно необходимо для XML, или фигурных скобок, которые чрезвычайно нужны для XAML. Чтобы ввести эти символы, необходимо использовать клавишу Sym, по нажатию которой выводится специальная дополнительная программная клавиатура, включающая эти три символа.

Область содержимого XamlCruncher разделена на две части с помощью *UniformStack*. Одна половина включает *XamlCruncherTextBox* с *TextBlock* для вывода сообщений об ошибках, и во второй половине располагается *ScrollViewer* с *Border* для размещения результирующего содержимого:

Проект Silverlight: XamlCruncher   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <petzold:UniformStack Name="uniformStack">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <petzold:XamlCruncherTextBox
                        x:Name="txtbox"
                        Grid.Row="0"
                        FontSize="{StaticResource PhoneFontSizeSmall}"
                        FontFamily="Courier New"
                        TextChanged="OnTextBoxTextChanged"
                        XamlResult="OnXamlCruncherTextBoxXamlResult" />

            <TextBlock Name="statusText"
                       Grid.Row="1"
                       TextWrapping="Wrap" />
        </Grid>

        <ScrollViewer HorizontalScrollBarVisibility="Auto"
                      VerticalScrollBarVisibility="Auto">
            <Border Name="container" />
        </ScrollViewer>

    </petzold:UniformStack>
</Grid>
```

Файл кода выполняет несколько задач. При каждом изменении текста он обеспечивает его сохранение в изолированном хранилище. Это позволяет возвращаться к работе с XAML снова и снова в течение нескольких дней или недель, если вдруг у вас не хватает терпения ввести все сразу.

Проект Silverlight: XamlCruncher   Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

    public MainPage()
    {
        InitializeComponent();
        Application.Current.UnhandledException += OnUnhandledException;
```

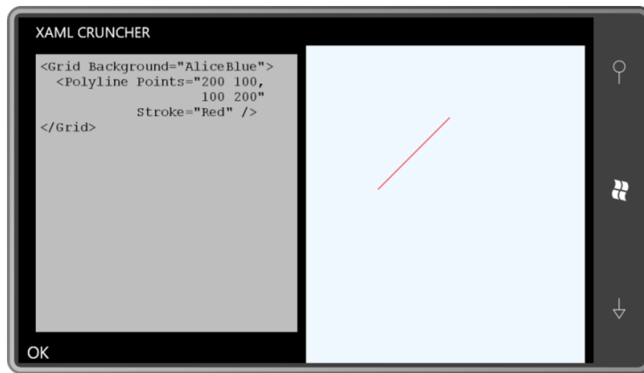
```
        string text;

        if (!settings.TryGetValue<string>("text", out text))
            text = "<Grid Background=\"AliceBlue\">\r    \r</Grid>";

        txtbox.Text = text;
    }

    protected override void OnOrientationChanged(OrientationChangedEventArgs args)
    {
        uniformStack.Orientation =
                ((args.Orientation & PageOrientation.Portrait) == 0) ?
                    System.Windows.Controls.Orientation.Horizontal :
                    System.Windows.Controls.Orientation.Vertical;

        base.OnOrientationChanged(args);
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        settings["text"] = txtbox.Text;
    }

    void OnUnhandledException(object sender,
                              ApplicationUnhandledExceptionEventArgs args)
    {
        statusText.Text = args.ExceptionObject.Message;
        args.Handled = true;
    }

    void OnXamlCruncherTextBoxXamlResult(object sender, XamlCruncherEventArgs args)
    {
        if (args.Error != null)
        {
            statusText.Text = args.Error;
        }
        else
        {
            container.Child = args.Element;
            statusText.Text = "OK";
        }
    }
}
```

Метод *OnOrientationChanged* меняет ориентацию *UniformStack* при изменении ориентации экрана. Поскольку с приложением XamlCruncher проще (незначительно) работать посредством аппаратной клавиатуры, его можно использовать при любой ориентации экрана.

Приложение также пытается обрабатывать необработанные исключения. Особенно часто это происходит при работе с анимациями, когда фрагмент XAML может нормально передаваться в *XamlReader.Load*, но позднее формировать исключение.

Это приложение следует выполнять с выключенным отладчиком Visual Studio, в противном случае Visual Studio будет прерывать выполнение каждый раз при формировании исключения.

Рассмотрим небольшой пример:



Приложение VectorGraphicsDemos (которое входит в исходный код для данной главы, но не представляет особого интереса, чтобы уделять ему особое внимание) включает элемент управления *XamlCruncherTextBox* и файл со всеми небольшими фрагментами XAML, представленными в ходе данной главы. Вы можете пролистать эти файлы, увидеть получаемые изображения и поредактировать их, если есть желание.

Глава 14
Растровая графика

В главе 4 я продемонстрировал, как приложение для Windows Phone 7 может получать растровые изображения. Они могут храниться в самом приложении, загружаться из Интернета либо поступать с камеры или из библиотеки изображений телефона. В данной главе мы выйдем за рамки задачи *загрузки* растровых изображений и обратимся к вопросам их *сохранения*. Растровые изображения могут сохраняться в изолированное хранилище или специальный альбом библиотеки изображений под именем «Saved Pictures» (Сохраненные изображения).

Если приложению необходимо сохранять растровое изображение, вероятно, для этого есть веские основания! Возможно, приложение создает растровое изображение с нуля или некоторым образом изменяет существующее. Для реализации этих задач используется замечательный и обладающий огромными возможностями класс *WriteableBitmap*.

Иерархия класса Bitmap

Как помните, существует два способа вывести растровое изображение на экран: с помощью элемента *Image* или создав *ImageBrush*. И свойство *Source* элемента *Image*, и свойство *ImageSource* объекта *ImageBrush* типа *ImageSource*. Класс *ImageSource* занимает очень важное место в разделе иерархии классов Silverlight, посвященном растровым изображениям:

Object
    DependencyObject (абстрактный)
        ImageSource (абстрактный)
            BitmapSource (абстрактный)
                BitmapImage
                WriteableBitmap

ImageSource имеет всего один класс-потомок и ни одного самостоятельно описанного открытого свойства, что наводит на мысль о его ненужности. Но такова ситуация только в Silverlight. В Windows Presentation Foundation от класса *ImageSource* наследуются классы, описывающие изображения, полученные средствами как векторной, так и растровой графики.

Остальные три класса описаны в пространстве имен *System.Windows.Media.Imaging*.

BitmapSource определяет два открытых свойства только для чтения и один метод:

- Свойство *PixelWidth* (Ширина в пикселах) типа *int*.
- Свойство *PixelHeight* (Высота в пикселах) типа *int*.
- Метод *SetSource*, имеющий один аргумент типа *Stream*.

Аргумент *Stream* может быть файловым потоком, сетевым потоком или потоком в памяти некоторого вида. Но *Stream* должен обеспечивать данные растрового изображения либо в формате JPEG, либо PNG. Создаваемое растровое изображение имеет фиксированный размер, который не может быть изменен.

Класс *BitmapImage* расширяет функциональность *BitmapSource* возможностью ссылаться на растровое изображение посредством URI. *BitmapImage* предоставляет следующие члены:

- Конструктор, принимающий аргумент типа *Uri*.
- Свойство *UriSource* типа *Uri*.
- Свойство *CreateOptions* (Параметры инициализации).
- Три события, которые позволяют отслеживать процесс загрузки и сообщать об его успешном завершении или сбое.

Свойство *CreateOptions* типа *CreateOptions* – это перечисление, включающее три члена: *None*, *DelayCreation* (Отложить инициализацию) и *IgnoreImageCache* (Игнорировать кэш изображения). Значение по умолчанию – *DelayCreation*, что не дает начать загрузку изображения до тех пор, пока оно действительно не потребуется для формирования визуального представления. Значение *IgnoreImageCache* используется, если приложение знает, что ранее загруженное изображение стало недействительным. *DelayCreation* и *IgnoreImageCache* можно использовать в сочетании с оператором C# побитовое ИЛИ.

Сочетая в себе функциональность *BitmapSource* и *BitmapImage*, класс *BitmapImage* позволяет загружать растровые изображения в формате JPEG или PNG, используя либо объект *Stream*, либо объект *Uri*. Он не предоставляет возможности сохранять растровые изображения.

Класс *WriteableBitmap* тоже продолжает эту традицию. Сам по себе он не имеет никакой функциональности для сохранения растровых изображений, но предоставляет доступ ко всем пикселам, образующим растровое изображение. Поддерживается только один формат пикселов, при котором каждый пиксел представлен 32-разрядным значением. Мы можем получить значения пикселов существующего растрового изображения или задать новые значения пикселов для *WriteableBitmap* и получить новое изображение. Доступ к значениям пикселов обеспечивает достаточно большую гибкость для сохранения или загрузки растровых изображений. Можно создать собственный «кодер» растровых изображений и сохранять значения пикселов в определенном растровом формате или собственный «декодер» для доступа к файлам определенного формата и преобразования их в значения пикселов без сжатия.

WriteableBitmap также предоставляет возможность для «рисования» изображений на битовой матрице, используя элементы Silverlight. Так можно нанести на битовую матрицу элементы *Button* и *Slider*, но чаще всего для этого используются элементы, наследуемые от *Shape*. Иначе говоря, *WriteableBitmap* позволяет преобразовывать векторный рисунок в растровое изображение.

Класс *WriteableBitmap* описывает следующие конструкторы, методы и свойства:

- Конструктор, принимающий *UIElement* и объект трансформации.
- Конструктор, принимающий ширину и высоту изображения в пикселах.
- Конструктор, принимающий объект *BitmapSource*.
- Метод *Render* (Формировать визуальное представление) принимает *UIElement* и объект трансформации.
- Метод *Invalidate* (Аннулировать) для обновления визуальных элементов растрового изображения.
- Свойство *Pixels* (Пикселы) – массив элементов типа *int*.

Не забывайте, что *WriteableBitmap* наследуется от класса *BitmapSource*, а не *BitmapImage*, поэтому не имеет возможности загрузки растрового изображения по URI. Но мы можем

загрузить так, с помощью URI, объект *BitmapImage* и затем создать из него *WriteableBitmap*, используя третий из перечисленных выше конструктор.

WriteableBitmap позволяет размещать изображения в растровой матрице, используя две техники:

- Путем формирования визуального представления объекта*UIElement* в растровой матрице.
- Прямой обработкой значений пикселов.

Эти техники могут сочетаться любым удобным способом.

Кроме того, Windows Phone 7 предоставляет несколько вспомогательных методов, которые обеспечивают альтернативные способы загрузки и сохранения файлов в формате JPEG:

- Статический метод *PictureDecoder.DecodeJpeg*, описанный в пространстве имен *Microsoft.Phone*, позволяет загружать файлы в формате JPEG из *Stream* с ограничением максимальных значений *Width* и *Height*. Это необходимо, если известно, что определенное изображение в формате JPEG может намного превышать имеющуюся область отображения. Метод возвращает объект *WritableBitmap*.
- Класс *Extensions* (Расширения) из пространства имен *System.Windows.Media.Imaging* включает два метода расширения для *WriteableBitmap*: *LoadJpeg* (Загрузить JPEG) (который не обеспечивает никакой дополнительной функциональности, кроме метода *SetSource*, определенного *BitmapSource*) и метод *SaveJpeg* (Сохранить JPEG), который позволяет менять высоту и ширину изображения и задавать качество сжатия.
- Метод *SavePicture* (Сохранить изображение) XNA-класса *MediaLibrary* позволяет сохранять растровое изображение в формате JPEG в библиотеку изображений телефона из объекта *Stream* или *байтового* массива. Этот метод лучше использовать в сочетании с методом расширения *SaveJpeg* с объектом-посредником *MemoryStream*, как я продемонстрирую ближе к концу данной главы.

WriteableBitmap и *UIElement*

Класс *WriteableBitmap* может получить визуальные элементы объекта *UIElement* двумя способами. В первом случае используется один из конструкторов:

```
WriteableBitmap writeableBitmap = new WriteableBitmap(element, transform);
```

Аргумент *element* (элемент) типа *UIElement*, и аргумент *transform* типа *Transform*. Этот конструктор создает растровое изображение на основании размера аргумента *element*, с учетом изменений, обусловливаемых аргументом *transform* (для которого можно задать значение *null*).

На растровой матрице формируется визуальное представление элемента и всех его дочерних визуальных элементов. Но любой *RenderTransform*, применяемый к элементу, игнорируется. Предоставление возможности применения этой трансформации по желанию является причиной введения второго аргумента. Результирующее растровое изображение берет за основу максимальные значения координат трансформированного элемента по вертикали и по горизонтали. Любая часть элемента, которая в результате трансформации оказалась в отрицательной области (слева или выше исходного элемента), отсекается.

Рассмотрим простой пример. В качестве фона сетки для содержимого используется текущий контрастный цвет. Сетка включает *TextBlock* и элемент *Image*:

Проект Silverlight: RecursivePageCaptures Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      Background="{StaticResource PhoneAccentBrush}">

    <TextBlock Text="Tap anywhere to capture page"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <Image Name="img"
           Stretch="Fill" />
</Grid>
```

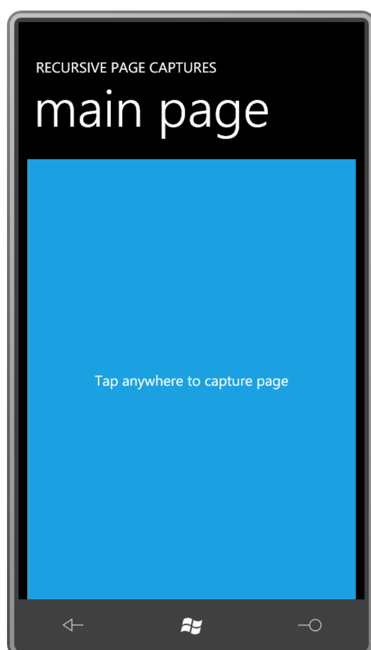
В элементе *Image* пока нет растрового изображения для отображения, но когда оно появится, соотношение его размеров будет проигнорировано, и изображение заполнит всю сетку для содержимого, перекрывая *TextBlock*.

По касанию экрана в файле выделенного кода объекту-источнику элемента *Image* задается новый *WriteableBitmap*, первым аргументом которого является сама страница:

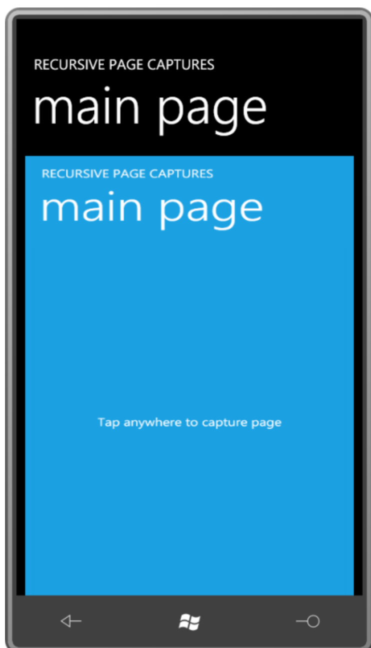
Проект Silverlight: RecursivePageCaptures Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        img.Source = new WriteableBitmap(this, null);

        args.Complete();
        args.Handled = true;
        base.OnManipulationStarted(args);
    }
}
```

При первом запуске приложения экран выглядит следующим образом:



После первого касания вся страница превращается в растровое изображение, отображаемое элементом *Image*:



Не забываем, что свойству *Background* перехваченного объекта *PhoneApplicationPage* задано значение по умолчанию *null*. Именно поэтому мы видим исходный фон панели содержимого за перехваченными заголовками. При последующих касаниях экрана происходит повторное сохранение содержимого страницы, включая сохраненный ранее элемент *Image*:

Эти элементы могут быть «сохранены» растровым изображением, только став его частью.

У класса *WriteableBitmap* также имеется метод *Render* с такими же двумя аргументами, как и в только что продемонстрированном конструкторе:

```
writeableBitmap.Render(element, transform);
```

Чтобы получить фактическое растровое изображение, содержащее все визуальные элементы, переданные в аргументе *element*, за вызовом *Render* должен следовать вызов *Invalidate*:

```
writeableBitmap.Invalidate();
```

Очевидно что на момент вызова этих методов *WriteableBitmap* уже создан, т.е. имеет фиксированный размер. На основании размера элемента и трансформации могут быть обрезаны некоторые (или все) элементы.

Если вызвать метод *Render* для вновь созданного элемента *Button*, к примеру, обнаружится, что он не работает. У этого нового *Button* нулевой размер. Чтобы задать элементу отличный от нулевого размер, необходимо вызвать методы *Measure* и *Arrange*. Однако в большинстве случаев мне не удавалось задать некоторый размер элементу даже путем вызова этих методов. Кажется, эта техника обеспечивает намного лучшие результаты, если элемент уже является частью дерева визуальных элементов. Лучше всего она работает с элементами *Image* и производными от *Shape*.

Рассмотрим приложение, которое получает растровое изображение из библиотеки изображений телефона и затем разрезает его на четыре части, ширина и высота каждой из которых соответствует половине ширины и высоты исходного изображения.

Область содержимого в приложении SubdivideBitmap (Разделение растрового изображения) включает *TextBlock* и *Grid* с двумя строками и двумя столбцами одинакового размера. В каждой из четырех ячеек *Grid* располагается элемент *Image*, имя которого соответствует его местоположению в сетке. Например, изображение с именем *imgUL* располагается в верхнем левом углу, а изображение с именем *imgLR* – в нижнем правом[1].

Проект Silverlight: SubdivideBitmap Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Name="txtblk"
               Text="Touch to choose image"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <Grid HorizontalAlignment="Center"
          VerticalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Image Name="imgUL" Grid.Row="0" Grid.Column="0" Margin="2" />
        <Image Name="imgUR" Grid.Row="0" Grid.Column="1" Margin="2" />
        <Image Name="imgLL" Grid.Row="1" Grid.Column="0" Margin="2" />
        <Image Name="imgLR" Grid.Row="1" Grid.Column="1" Margin="2" />
    </Grid>
</Grid>
```

[1] Аббревиатура *UL* соответствует английскому «Upper-Left», что в переводе означает верхний левый угол; и аббревиатура *LR* соответствует английскому «Lower-Right», что в переводе означает нижний правый угол (*прим. переводчика*).

Файл выделенного кода класса *MainPage* настроен для реализации задачи *PhotoChooserTask*: объект *PhotoChooserTask* определен как поле, и в конце конструктора прикреплен обработчик событий *Completed*.

Проект Silverlight: SubdivideBitmap Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    PhotoChooserTask photoChooser = new PhotoChooserTask();

    public MainPage()
    {
        InitializeComponent();
        photoChooser.Completed += OnPhotoChooserCompleted;
    }
    protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
    {
        int dimension = (int)Math.Min(ContentPanel.ActualWidth,
                                      ContentPanel.ActualHeight) - 8;

        photoChooser.PixelHeight = dimension;
        photoChooser.PixelWidth = dimension;
        photoChooser.Show();

        args.Complete();
        args.Handled = true;
        base.OnManipulationStarted(args);
    }
    …
}
```

После этого перегруженный *OnManipulationStarted* вызывает метод *Show* объекта *PhotoChooserTask*, запрашивая квадратную растровую матрицу, размеры которой определяются размером панели для содержимого. Из этого размера вычитаются 8 пикселов в пользу свойства *Margin*, заданного для каждого элемента *Image* в XAML-файле.

Когда *PhotoChooserTask* формирует событие *Completed*, вызывается обработчик этого события. Он начинает выполнение с создания объекта *BitmapImage* из потока, который ссылается на выбранное растровое изображение. Затем создается элемент *Image* (под именем *imgBase*) для отображения этого растрового изображения. Заметьте, элемент *Image* не является частью дерева визуальных элементов. Он существует исключительно для исполнения роли объекта-источника для вызовов *Render*.

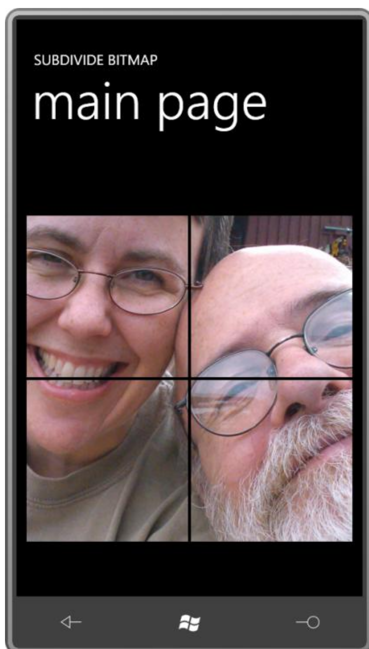
Проект Silverlight: SubdivideBitmap Файл: MainPage.xaml.cs (фрагмент)

```
void OnPhotoChooserCompleted(object sender, PhotoResult args)
{
    if (args.Error != null || args.ChosenPhoto == null)
        return;

    BitmapImage bitmapImage = new BitmapImage();
    bitmapImage.SetSource(args.ChosenPhoto);

    Image imgBase = new Image();
    imgBase.Source = bitmapImage;
    imgBase.Stretch = Stretch.None;

    // Верхний левый
    WriteableBitmap writeableBitmap = new WriteableBitmap(bitmapImage.PixelWidth /
2,
                                                          bitmapImage.PixelHeight /
```

```
2);
    writeableBitmap.Render(imgBase, null);
    writeableBitmap.Invalidate();
    imgUL.Source = writeableBitmap;

    // Верхний правый
    writeableBitmap = new WriteableBitmap(bitmapImage.PixelWidth / 2,
                                          bitmapImage.PixelHeight / 2);
    TranslateTransform translate = new TranslateTransform();
    translate.X = -bitmapImage.PixelWidth / 2;
    writeableBitmap.Render(imgBase, translate);
    writeableBitmap.Invalidate();
    imgUR.Source = writeableBitmap;

    // Нижний левый
    writeableBitmap = new WriteableBitmap(bitmapImage.PixelWidth / 2,
                                          bitmapImage.PixelHeight / 2);
    translate.X = 0;
    translate.Y = -bitmapImage.PixelHeight / 2;
    writeableBitmap.Render(imgBase, translate);
    writeableBitmap.Invalidate();
    imgLL.Source = writeableBitmap;

    // Нижний правый
    writeableBitmap = new WriteableBitmap(bitmapImage.PixelWidth / 2,
                                          bitmapImage.PixelHeight / 2);
    translate.X = -bitmapImage.PixelWidth / 2;
    writeableBitmap.Render(imgBase, translate);
    writeableBitmap.Invalidate();
    imgLR.Source = writeableBitmap;

    txtblk.Visibility = Visibility.Collapsed;
}
```

Далее обработчик события *Completed* создает четыре объекта *WriteableBitmap*, размеры каждого из которых соответствуют половине размеров исходного изображения. (Вычисления выполняются на основании размеров *BitmapImage*, а не размеров *Image*, которые на данный момент равны нулю.)

Также для всех вызовов *Render*, кроме первого, задан *TranslateTransform*, который обеспечивает смещение влево или вверх (или в обоих направлениях) на половину размера растрового изображения. За каждым вызовом *Render* следует вызов *Invalidate*. После этого каждый *WriteableBitmap* задается как значение свойства *Source* соответствующего элемента *Image* из XAML-файла. Задание свойства *Margin* для этих элементов *Image* обеспечивает их визуальное разделение, позволяя отчетливо видеть, что теперь мы работаем с четырьмя отдельными элементами *Image*:



Обратите внимание, что в коде используется всего один объект *TranslateTransform*. Как правило, для каждого элемента задается собственная трансформация, но если вы хотите применить одну и ту же трансформацию ко всем элементам, можно организовать ее совместное использование. В данном случае *TranslateTransform* используется временно исключительно в целях формирования визуального представления.

Далее в данной главе на примере небольшой игры я покажу другой подход реализации разделения растрового изображения на части.

Работа с пикселами

Свойство *Pixels* класса *WritableBitmap* – это массив объектов типа *int*, т.е. каждый пиксел включает 32 бита. Само свойство *Pixels* является свойством только для чтения, таким образом, мы не можем заменить массив целиком, но можем задавать и возвращать элементы этого массива.

Растровое изображение – это двухмерный массив пикселов. Свойство *Pixels* класса *WriteableBitmap* – это одномерный массив значений типа *int*. В массиве *Pixels* пикселы растрового изображения хранятся последовательно, начиная с верхней строки и вниз, двигаясь по строкам слева направо. Количество элементов в массиве равно произведению ширины и высоты растрового изображения в пикселах.

Если *bm* – объект *WriteableBitmap*, тогда количество элементов в свойстве *Pixels* равно *bm.PixelWidth * bm.PixelHeight*. Предположим, требуется выполнить доступ к пикселу в столбце *x*, где значения *x* лежат в диапазоне от 0 до *bm.PixelWidth* – 1, и строке *y*, где значения *y* лежат в диапазоне от 0 до *bm. PixelHeight* – 1. Свойство *Pixels* индексируется следующим образом:

```
bm.Pixels[y * bm.PixelWidth + x]
```

Silverlight for Windows Phone поддерживает только один формат пикселов, который иногда обозначают как PARGB32. Позвольте мне расшифровать это кодовое название. Начнем с конца.

«32» в конце названия формата означает 32 бита или 4 байта. Это размер одного пиксела. Буквы ARGB указывают на то, что байт Альфа-канала (непрозрачность) расположен в старших 8 битах 32-разрядного целого. За ним следует байт красного канала (Red), байт зеленого (Green) и байт синего (Blue), который занимает младшие 8 битов целого.

Если *A*, *R*, *G* и *B* типа *byte*, 32-разрядное целое значение пиксела можно составить следующим образом:

```
int pixel = A << 24 | R << 16 | G << 8 | B
```

Смещенные значения, неявно преобразованные в значения типа *int*, объединяются посредством оператора C# побитовое ИЛИ. Получить компоненты фактического значения пиксела можно следующим образом:

```
byte A = (byte)(pixel & 0xFF000000 >> 24);
byte R = (byte)(pixel & 0x00FF0000 >> 16);
byte G = (byte)(pixel & 0x0000FF00 >> 8);
byte B = (byte)(pixel & 0x000000FF);
```

Если байт альфа-канала содержит 255, пиксел непрозрачный. Значение 0 соответствует полностью прозрачному пикселу. Промежуточные значения указывают на промежуточные уровни прозрачности.

В формате пикселов PARGB32 P означает «premultiplied», т.е. «предварительно умножены». Это говорит о том, что если значение альфа-канала отлично от 255, значения красного, зеленого и синего уже приведены в соответствие заданной прозрачности.

Попробуем разобраться с этой концепцией на примере одного пиксела. Предположим, мы решили задать пикселу следующий цвет:

```
Color.FromArgb(128, 0, 0, 255)
```

Это соответствует синему с 50% прозрачностью. При формировании визуального представления этого пиксела на поверхности, имеющей определенный фон, его цвет должен комбинироваться с имеющимися цветами поверхности. При отрисовке на черном фоне результирующим RGB-цветом будет (0, 0, 128). Это что-то среднее между синим и черным. При отрисовке на белом фоне результирующим цветом будет (127, 127, 255). Каждый из трех компонентов результирующего цвета является средним арифметическим между значениями пиксела и поверхности.

При любом другом значении прозрачности, отличном от 50%, результирующий цвет является средневзвешенным значением от значений исходного пиксела и поверхности. Подстрочные индексы переменных в следующих формулах указывают на «результат» (result) формирования визуального представления частично прозрачного пиксела «источника» (source) на существующей «поверхности» (surface):

$$R_{result} = \left[(255 - A_{source}) \times R_{surface} + A_{source} \times R_{surface}\right] \div 255$$

$$G_{result} = \left[(255 - A_{source}) \times G_{surface} + A_{source} \times G_{surface}\right] \div 255$$

$$B_{result} = \left[(255 - A_{source}) \times B_{surface} + A_{source} \times B_{surface}\right] \div 255$$

Если визуальное представление растрового изображения формируется на произвольной поверхности, эти вычисления должны производиться для каждого пиксела.

Очень часто требуется формировать визуальное представление одного и того же растрового изображения многократно на разных поверхностях. Представленные выше вычисления можно несколько ускорить, если заранее умножить компоненты красного, зеленого и синего

цветов на значение альфа-канала. Это предварительное умножение для каждого компонента выполняется следующим образом:

$$PR_{source} = (A_{source} \times R_{source}) \div 255$$

И аналогично для зеленого и синего. Это позволяет вдвое сократить количество операций умножения в результирующих уравнениях для формирования визуального представления растрового изображения:

$$R_{result} = \left[(255 - A_{source}) \times R_{surface}\right] \div 255 + PR_{source}$$

Когда бы мы ни работали со свойством *Pixels* объекта *WriteableBitmap*, мы всегда имеем дело с предварительно умноженными значениями. Например, пикселу растрового изображения необходимо задать RGB-значение цвета (40, 60, 255), но при этом значение альфа-канала равно 192. ARGB-значение в растровом изображении будет (192, 30, 45, 192). Каждое из составляющих значений цвета было умножено на 192/255 или примерно на 0,75.

В любом предварительно умноженном значении цвета значения составляющих красного, зеленого и синего должны быть меньше или равны значению альфа-канала. Ничего страшного не случится, если одно из составляющих значений будет больше значения альфа-канала, но тогда не будет обеспечен заданный уровень прозрачности.

При работе с ARGB-значениями цвета *без* предварительного умножения на альфа-канал различают «прозрачный черный», которому соответствует ARGB-значение (0, 0, 0, 0), и «прозрачный белый», которому соответствует ARGB-значение (0, 255, 255, 255). С предварительным умножением это различие исчезает, потому что прозрачный белый тоже будет соответствовать (0, 0, 0, 0).

Когда *WriteableBitmap* создается впервые, все пикселы имеют нулевое значение, что можно трактовать как «прозрачный черный», или «прозрачный белый», или «прозрачный серо-буро-малиновый».

Прямая запись в массив *Pixels* объекта *WriteableBitmap* позволяет создавать изображения любого типа.

С помощью сравнительно простых алгоритмов можно создавать стили кистей, которые не обеспечиваются стандартными производными от *Brush*. Область содержимого проекта CircularGradient (Круговой градиент) включает только один элемент *Image*, предназначенный для размещения в нем растрового изображения:

Проект Silverlight: CircularGradient Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img"
           HorizontalAlignment="Center"
           VerticalAlignment="Center" />
</Grid>
```

В файле выделенного кода *MainPage* значение радиуса выбрано довольно произвольно, и для *WriteableBitmap* задан квадрат со стороной, равной двум радиусам. Два цикла *for* для *x* и *y* обеспечивают перебор всех пикселов этой растровой матрицы:

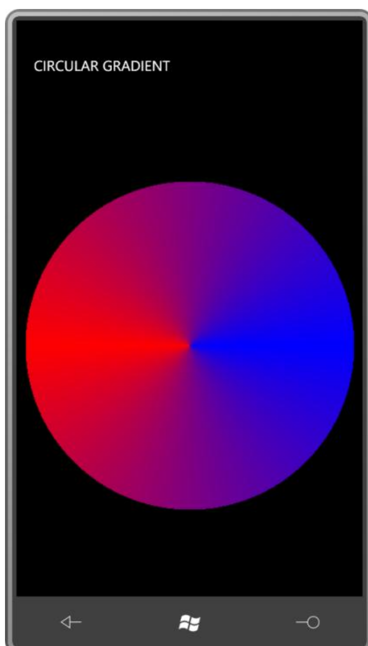
Проект Silverlight: CircularGradient Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
```

```
    const int RADIUS = 200;

    public MainPage()
    {
        InitializeComponent();

        WriteableBitmap writeableBitmap = new WriteableBitmap(2 * RADIUS, 2 *
RADIUS);

        for (int y = 0; y < writeableBitmap.PixelWidth; y++)
            for (int x = 0; x < writeableBitmap.PixelHeight; x++)
            {
                if (Math.Sqrt(Math.Pow(x - RADIUS, 2) + Math.Pow(y - RADIUS, 2)) <
RADIUS)
                {
                    double angle = Math.Atan2(y - RADIUS, x - RADIUS);
                    byte R = (byte)(255 * Math.Abs(angle) / Math.PI);
                    byte B = (byte)(255 - R);
                    int color = 255 << 24 | R << 16 | B;
                    writeableBitmap.Pixels[y * writeableBitmap.PixelWidth + x] =
color;
                }
            }

        writeableBitmap.Invalidate();
        img.Source = writeableBitmap;
    }
}
```

Центром *WriteableBitmap* является точка с координатами (200, 200). Вложенные циклы *for* начинаются с пропуска каждого пиксела, который отстоит от центра более чем на 200 пикселов. В этом квадратном растровом изображении непрозрачными будут только пикселы, принадлежащие кругу.

Линия, соединяющая центральную точку с любым пикселом растрового изображения, образует угол с горизонтальной осью. Этот угол вычисляется методом *Math.Atan2*. Затем на основании значения этого угла задаются значения переменным *R* и *B*, значение цвета формируется и сохраняется в массиве *Pixels*. Вызов *Invalidate* сопоставляет фактическое растровое изображение с этими пикселами. После этого растровое изображение задается как значение свойства *Source* элемента *Image*:



Векторная графика в растровой матрице

WriteableBitmap позволяет сочетать два подхода создания изображений. В следующем примере *WriteableBitmap* обеспечивает отображение *Path* поверх градиента, использующего прозрачность, так что мы можем увидеть предварительно умноженные альфа в действии.

Несомненно, вы помните этот элемент *Path* из предыдущей главы, который задавался строкой Silverlight Path Markup Syntax и обеспечивал отрисовку кота. Задача приложения VectorToRaster (Вектор в растр) – создать растровую матрицу точно такого же размера, как этот кот, и затем поместить в нее изображение кота.

Изображение кота описывается с помощью Path Markup Syntax в элементе *Path* в разделе *Resources* файла MainPage.xaml:

Проект Silverlight: VectorToRaster Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Path x:Key="catPath"
          Data="M 160 140 L 150 50 220 103
                M 320 140 L 330 50 260 103
                M 215 230 L 40 200
                M 215 240 L 40 240
                M 215 250 L 40 280
                M 265 230 L 440 200
                M 265 240 L 440 240
                M 265 250 L 440 280
                M 240 100 A 100 100 0 0 1 240 300
                          A 100 100 0 0 1 240 100
                M 180 170 A 40 40 0 0 1 220 170
                          A 40 40 0 0 1 180 170
                M 300 170 A 40 40 0 0 1 260 170
                          A 40 40 0 0 1 300 170" />
</phone:PhoneApplicationPage.Resources>
```

PathGeometry описывается в XAML в коллекции *Resources*. Я хотел определить *PathGeometry* напрямую, без применения *Path*, но как я ни старался и что бы я ни делал – задавал строку синтаксиса Path Markup Syntax как свойство *Figures* объекта *PathGeometry* либо помещал строку между открывающим и закрывающим тегами *PathGeometry* – у меня ничего не получилось.

Я использую элемент *Path* исключительно, чтобы обозначить эту строку как Path Markup Syntax для синтаксического анализатора XAML. Элемент *Path* не будет использоваться для каких-либо иных целей в приложении.

Область содержимого включает только пустой элемент *Image* для размещения растрового изображения:

Проект Silverlight: VectorToRaster Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img"
           HorizontalAlignment="Center"
           VerticalAlignment="Center" />
</Grid>
```

Все остальное происходит в конструкторе класса *MainPage*. Он немного длинноват, но снабжен подробными комментариями. Мы поэтапно рассмотрим всю логику:

Проект Silverlight: VectorToRaster Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();

    // Получаем PathGeometry из ресурса
    Path catPath = this.Resources["catPath"] as Path;
    PathGeometry pathGeometry = catPath.Data as PathGeometry;
    catPath.Data = null;

    // Получаем границы геометрического элемента
    Rect bounds = pathGeometry.Bounds;

    // Создаем новый контур, визуальное представление которого
    // будет сформировано в растровой матрице
    Path newPath = new Path
    {
        Stroke = this.Resources["PhoneForegroundBrush"] as Brush,
        StrokeThickness = 5,
        Data = pathGeometry
    };

    // Создаем WriteableBitmap
    WriteableBitmap writeableBitmap =
        new WriteableBitmap((int)(bounds.Width + newPath.StrokeThickness),
                            (int)(bounds.Height + newPath.StrokeThickness));

    // Задаем цвет фона растрового изображения
    Color baseColor = (Color)this.Resources["PhoneAccentColor"];

    // Рассматриваем растровое изображение как эллипс:
    // radiusX и radiusY – тоже центры!
    double radiusX = writeableBitmap.PixelWidth / 2.0;
    double radiusY = writeableBitmap.PixelHeight / 2.0;

    for (int y = 0; y < writeableBitmap.PixelHeight; y++)
        for (int x = 0; x < writeableBitmap.PixelWidth; x++)
        {
            double angle = Math.Atan2(y - radiusY, x - radiusX);
            double ellipseX = radiusX * (1 + Math.Cos(angle));
            double ellipseY = radiusY * (1 + Math.Sin(angle));

            double ellipseToCenter =
                Math.Sqrt(Math.Pow(ellipseX - radiusX, 2) +
                          Math.Pow(ellipseY - radiusY, 2));

            double pointToCenter =
                Math.Sqrt(Math.Pow(x - radiusX, 2) + Math.Pow(y - radiusY, 2));

            double opacity = Math.Min(1, pointToCenter / ellipseToCenter);

            byte A = (byte)(opacity * 255);
            byte R = (byte)(opacity * baseColor.R);
            byte G = (byte)(opacity * baseColor.G);
            byte B = (byte)(opacity * baseColor.B);

            int color = A << 24 | R << 16 | G << 8 | B;

            writeableBitmap.Pixels[y * writeableBitmap.PixelWidth + x] = color;
        }

    writeableBitmap.Invalidate();

    // Находим трансформацию для перемещения Path к краям
    TranslateTransform translate = new TranslateTransform
    {
        X = -bounds.X + newPath.StrokeThickness / 2,
```

```
        Y = -bounds.Y + newPath.StrokeThickness / 2
    };

    writeableBitmap.Render(newPath, translate);
    writeableBitmap.Invalidate();

    // Задаем растровое изображение как значение элемента Image
    img.Source = writeableBitmap;
}
```

Код начинается с извлечения *PathGeometry* из коллекции *Resources*. Поскольку *PathGeometry* прикрепляется к элементу *Path*, он не сможет использоваться для каких-либо иных целей. Поэтому свойству *Data* элемента *Path* задается значение *null*. На этом роль элемента *Path* заканчивается, и он больше не используется в данном приложении.

Свойство *Bounds*, определенное классом *Geometry*, возвращает объект *Rect* с координатами верхнего левого угла объекта *PathGeometry*, в данном случае это точка (40, 50), и его шириной и высотой, которые в данном случае равны 400 и 250, соответственно. Обратите внимание, что данные значения строго геометрические и не учитывают ширину обводки, которая может присутствовать при формировании визуального представления этого элемента.

Далее для этого геометрического элемента создается элемент *Path*. В отличие от элемента *Path* в коллекции *Resources* XAML-файла, для данного *Path* задана кисть *Stroke* толщиной (*StrokeThickness*) 5.

Каков будет фактический размер сформированного графического элемента? Нам известно, что он будет как минимум 400 пикселов шириной и 250 пикселов высотой. Провести *точные* вычисления сложно, но *оценить* размеры не составит большого труда: если все линии графического элемента обводятся кистью толщиной 5 пикселов, к его визуальному представлению слева, сверху, справа и снизу будет добавлено по 2,5 пиксела, т.е. ширина и высота элемента увеличатся на 5 пикселов. Такие вычисления позволяют создать *WriteableBitmap* соответствующего размера. (Эти вычисления не учитывают вынос конусов в точках соединения линий, но они просты и обычно обеспечивают приемлемые результаты.)

Прежде чем формировать визуальное представление *Path* в *WriteableBitmap*, я хочу задать для растровой матрицы градиентную кисть, прозрачную в центре и переходящую в контрастный цвет по краям:

```
Color baseColor = (Color)this.Resources["PhoneAccentColor"];
```

Вообще-то обратный градиент (т.е. прозрачный по краям) выглядел бы более привлекательным, но я хочу продемонстрировать, насколько точно совпадают размеры растровой матрицы и визуального представления геометрического элемента.

Здесь два вложенных цикла *for* для *x* и *y* обеспечивают перебор всех пикселов растровой матрицы. Для каждого пиксела вычисляется значение *opacity*, диапазон допустимых значений которого от 0 (прозрачный) до 1 (непрозрачный):

```
double opacity = Math.Min(1, pointToCenter / ellipseToCenter);
```

Это значение *opacity* используется не только для вычисления байта альфа-канала, но также как коэффициент предварительного умножения для значений красного, зеленого и синего каналов:

```
byte A = (byte)(opacity * 255);
byte R = (byte)(opacity * baseColor.R);
byte G = (byte)(opacity * baseColor.G);
byte B = (byte)(opacity * baseColor.B);
```

Осталось лишь свести вместе все компоненты цвета и проиндексировать массив *Pixels*:

```
int color = A << 24 | R << 16 | G << 8 | B;
writeableBitmap.Pixels[y * writeableBitmap.PixelWidth + x] = color;
```

На данный момент приложение обработало все элементы массива *Pixels*, поэтому пора обновить изображение:

```
writeableBitmap.Invalidate();
```

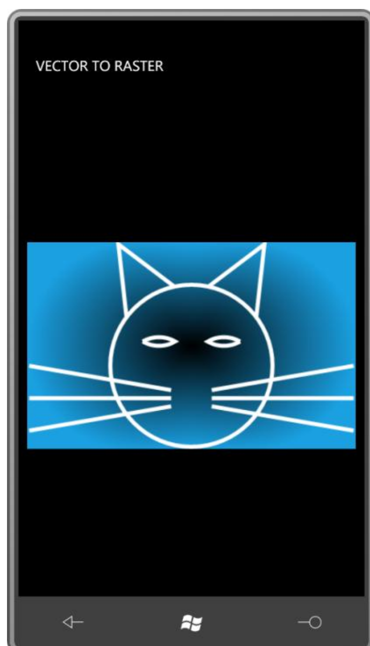
Теперь в растровой матрице должно быть сформировано визуальное представление элемента *Path* под именем *newPath* (Новый контур). Для элемента *Path* предусмотрен объект *PathGeometry*, верхний левый угол которого располагается в точке (40, 50), но при задании размеров *WriteableBitmap* использовались только ширина и высота геометрического элемента без учета толщины обводки. При формировании визуального представления *Path* во *WriteableBitmap* трансформация *TranslateTransform* должна обеспечить смещение верхнего левого угла прямоугольника на значения *X* и *Y*, полученные из свойства *Bounds* объекта *PathGeometry*. Но после этого необходимо также сместить *Path* немного вправо и вниз, чтобы вместить толщину обводки:

```
TranslateTransform translate = new TranslateTransform
{
    X = -bounds.X + newPath.StrokeThickness / 2,
    Y = -bounds.Y + newPath.StrokeThickness / 2
};
```

Теперь визуальное представление элемента *Path* во *WriteableBitmap* может быть сформировано:

```
writeableBitmap.Render(newPath, translate);
writeableBitmap.Invalidate();
```

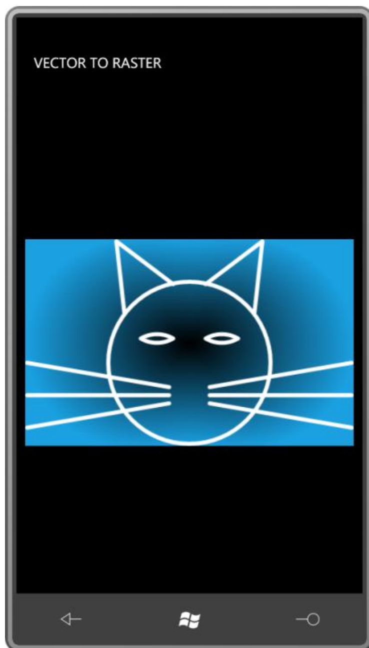
И вот результат:



Растровое изображение внизу точно соответствует геометрическому элементу, но немного шире необходимого размера по бокам. (Задайте для этих усов скругление на концах, и они будут точно касаться края.) Сверху высоты растрового изображения не достаточно, чтобы вместить конусы соединений линий ушей. Скруглите соединения, и картинка будет выглядеть лучше. Добавьте в описание *newPath* следующие три присваивания:

```
StrokeStartLineCap = PenLineCap.Round,
StrokeEndLineCap = PenLineCap.Round,
StrokeLineJoin = PenLineJoin.Round,
```

Теперь растровое изображение выглядит абсолютно правильно:



Изображения и захоронение

В 1890-х годах американский головоломщик Сэм Ллойд популяризировал головоломку, которая была изобретена пару десятилетий до этого и с тех пор известна как «пятнашки» (или по-французски «JeuDeTaquin»). В классическом виде этот пазл состоит из 15 фрагментов, каждый из которых обозначен цифрами от 1 до 15, случайным образом расположенных в сетке размером 4×4 (т.е. одна ячейка остается свободной). Цель – перемещая фрагменты, расставить их в правильном числовом порядке.

Эта головоломка была положена в основу первых игровых приложений, созданных для Apple Macintosh, где ее назвали PUZZLE. Вариант для Windows появилась в ранних версиях Microsoft Windows Software Development Kit (SDK) под именем MUZZLE. Это был единственный пример в SDK, написанный на Microsoft Pascal, а не на C.

Я хочу продемонстрировать вариант этой головоломки, в котором используются не нумерованные фрагменты, а разделенные на фрагменты фотографии из библиотеки изображений телефона. В результате игра становится несколько сложнее, но как вознаграждение это приложение демонстрирует сохранение изображений при захоронении приложения.

Область содержимого приложения включает *Grid* под именем *playGrid* (Игровое поле) для размещения фрагментов и две кнопки:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
```

```
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Grid Name="playGrid"
          Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
          HorizontalAlignment="Center"
          VerticalAlignment="Center" />

    <Button Content="load"
            Grid.Row="1" Grid.Column="0"
            Click="OnLoadClick" />

    <Button Name="scrambleButton"
            Content="scramble"
            Grid.Row="2" Grid.Column="1"
            IsEnabled="False"
            Click="OnScrambleClick" />
</Grid>
```

XAML-файл также включает две кнопки в *ApplicationBar*, которые также подписаны «load» (загрузить) и «scramble» (перемешать), что кажется совершенно излишним:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="False">
        <shell:ApplicationBarIconButton x:Name="appbarLoadButton"
                                        IconUri="/Images/appbar.folder.rest.png"
                                        Text="load"
                                        Click="OnLoadClick" />

        <shell:ApplicationBarIconButton x:Name="appbarScrambleButton"
                                        IconUri="/Images/appbar.refresh.rest.png"
                                        Text="scramble"
                                        IsEnabled="False"
                                        Click="OnScrambleClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

У меня не получилось реализовать функциональность рандомизации из *ApplicationBar*, но я оставил этот элемент в разметке (и коде) и задал его свойству *IsVisible* значение *false*. Может быть, однажды *ApplicationBar* обретет более регулярное поведение.

Класс *MainPage* в коде начинается с объявления некоторых констант. Приложение настроено на размещение 4 фрагментов изображения по вертикали и 4 по горизонтали, но это можно изменить. Очевидно, что в портретном режиме будет лучше, если VERT_TILES (Фрагментов по вертикали) будет больше HORZ_TILES (Фрагментов по горизонтали). Другие поля предусмотрены для сохранения данных состояния в объекте *PhoneApplicationService* при захоронении и использования *PhotoChooserTask* для выбора фотографий.

Исключительно важное значение имеет массив *tileImages* (Фрагменты изображения). В нем хранятся все элементы *Image* фрагментов изображения. В любой момент времени один из членов этого массива будет иметь значение *null*, представляя свободную ячейку. Эта свободная ячейка также обозначается индексами *emptyRow* (Пустая строка) и *emptyCol* (Пустой столбец).

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    const int HORZ_TILES = 4;
    const int VERT_TILES = 4;
    const int MARGIN = 2;

    PhoneApplicationService appService = PhoneApplicationService.Current;
    PhotoChooserTask photoChooser = new PhotoChooserTask();
    Random rand = new Random();

    Image[,] tileImages = new Image[VERT_TILES, HORZ_TILES];
    bool haveValidTileImages;
    int emptyRow, emptyCol;
    int scrambleCountdown;

    public MainPage()
    {
        InitializeComponent();

        for (int col = 0; col < HORZ_TILES; col++)
        {
            ColumnDefinition coldef = new ColumnDefinition();
            coldef.Width = new GridLength(1, GridUnitType.Star);
            playGrid.ColumnDefinitions.Add(coldef);
        }

        for (int row = 0; row < VERT_TILES; row++)
        {
            RowDefinition rowdef = new RowDefinition();
            rowdef.Height = new GridLength(1, GridUnitType.Star);
            playGrid.RowDefinitions.Add(rowdef);
        }

        appbarScrambleButton = this.ApplicationBar.Buttons[1] as
ApplicationBarIconButton;

        photoChooser.Completed += OnPhotoChooserCompleted;
    }
    …
}
```

В конструкторе приложение инициализирует коллекции *ColumnDefinition* и *RowDefinition* объекта *Grid*, в котором располагаются фрагменты, и (как обычно) задается обработчик события *Completed*, формируемого *PhotoChooserTask*.

Когда пользователь щелкает кнопку «load», приложение определяет размеры каждого фрагмента на основании заданных ширины и высоты области содержимого, количества фрагментов и размера поля. Полученные значения сохраняются в свойствах *PixelWidth* и *PixelHeight* объекта *PhotoChooserTask*:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

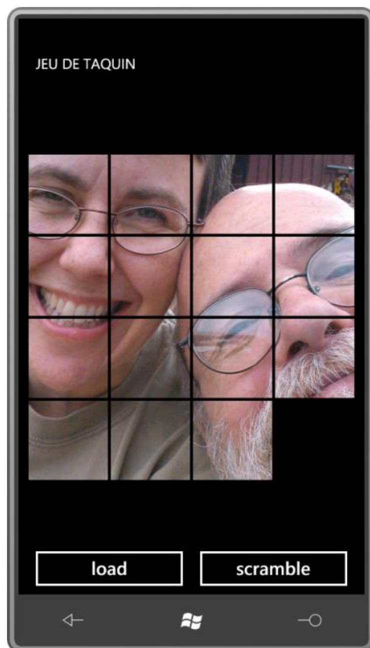
```
void OnLoadClick(object sender, EventArgs args)
{
    int tileSize = (int)Math.Min(ContentPanel.ActualWidth / HORZ_TILES,
                                 ContentPanel.ActualHeight / VERT_TILES)
                   - 2 * MARGIN;

    photoChooser.PixelWidth = tileSize * HORZ_TILES;
    photoChooser.PixelHeight = tileSize * VERT_TILES;
    photoChooser.Show();
}
```

По завершении *PhotoChooserTask* обработчик события *Completed* разделяет растровое изображение на небольшие квадратные фрагменты и создает элемент *Image* для каждого из них. В рассматриваемом ранее в данной главе приложении SubdivideBitmap была продемонстрирована реализация разделения растрового изображения на квадратные фрагменты с использованием метода *Render* объекта *WriteableBitmap*. В данном приложении сделано наоборот: объекты *WriteableBitmap* создаются размером с фрагмент изображения, и затем в их массивы *Pixels* копируются соответствующие пикселы полного растрового изображения:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

```
void OnPhotoChooserCompleted(object sender, PhotoResult args)
{
    if (args.Error == null && args.ChosenPhoto != null)
    {
        BitmapImage bitmapImage = new BitmapImage();
        bitmapImage.SetSource(args.ChosenPhoto);
        WriteableBitmap writeableBitmap = new WriteableBitmap(bitmapImage);
        int tileSize = writeableBitmap.PixelWidth / HORZ_TILES;

        emptyCol = HORZ_TILES - 1;
        emptyRow = VERT_TILES - 1;

        for (int row = 0; row < VERT_TILES; row++)
            for (int col = 0; col < HORZ_TILES; col++)
                if (row != emptyRow || col != emptyCol)
                {
                    WriteableBitmap tile = new WriteableBitmap(tileSize, tileSize);

                    for (int y = 0; y < tileSize; y++)
                        for (int x = 0; x < tileSize; x++)
                        {
                            int yBit = row * tileSize + y;
                            int xBit = col * tileSize + x;

                            tile.Pixels[y * tileSize + x] =
                                writeableBitmap.Pixels[yBit *
                                            writeableBitmap.PixelWidth + xBit];
                        }
                    GenerateImageTile(tile, row, col);
                }

        haveValidTileImages = true;
        scrambleButton.IsEnabled = true;
        appbarScrambleButton.IsEnabled = true;
    }
}

void GenerateImageTile(BitmapSource tile, int row, int col)
{
    Image img = new Image();
    img.Stretch = Stretch.None;
    img.Source = tile;
    img.Margin = new Thickness(MARGIN);
    tileImages[row, col] = img;

    Grid.SetRow(img, row);
    Grid.SetColumn(img, col);
    playGrid.Children.Add(img);
}
```

За фактическое создание элементов *Image* и их добавление в *Grid* отвечает метод *GenerateImageTile* (Создать фрагмент изображения). Этот метод также сохраняет элементы *Image* в массив *tileImages*.



На этом этапе фрагменты еще расположены в правильном порядке, но уже есть возможность перемещать их по полю. Когда мы обратимся к реализации перемещений фрагментов, обнаружится, чтобы алгоритмически это намного проще, чем можно было себе представить. Рассмотрим пустой квадрат. Какие фрагменты могут быть перемещены в этот квадрат? Только те, которые находятся слева, сверху, справа и снизу пустого квадрата, и эти фрагменты могут перемещаться только в одном направлении. Это означает, что пользовательский интерфейс должен реагировать только на касания и не обращать внимания на перемещения других фрагментов.

Если поразмыслить еще глубже, можно увидеть, что должно быть реализовано перемещение сразу всего ряда фрагментов по касанию любого фрагмента строки или столбца с пустым квадратом. В этом нет абсолютно никакой двусмысленности.

Рассмотрим полную реализацию логики перемещения:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    if (args.OriginalSource is Image)
    {
        Image img = args.OriginalSource as Image;
        MoveTile(img);
        args.Complete();
        args.Handled = true;
    }
    base.OnManipulationStarted(args);
}

void MoveTile(Image img)
{
    int touchedRow = -1, touchedCol = -1;

    for (int y = 0; y < VERT_TILES; y++)
```

```
        for (int x = 0; x < HORZ_TILES; x++)
            if (tileImages[y, x] == img)
            {
                touchedRow = y;
                touchedCol = x;
            }

    if (touchedRow == emptyRow)
    {
        int sign = Math.Sign(touchedCol - emptyCol);

        for (int x = emptyCol; x != touchedCol; x += sign)
        {
            tileImages[touchedRow, x] = tileImages[touchedRow, x + sign];
            Grid.SetColumn(tileImages[touchedRow, x], x);
        }
        tileImages[touchedRow, touchedCol] = null;
        emptyCol = touchedCol;
    }
    else if (touchedCol == emptyCol)
    {
        int sign = Math.Sign(touchedRow - emptyRow);

        for (int y = emptyRow; y != touchedRow; y += sign)
        {
            tileImages[y, touchedCol] = tileImages[y + sign, touchedCol];
            Grid.SetRow(tileImages[y, touchedCol], y);
        }
        tileImages[touchedRow, touchedCol] = null;
        emptyRow = touchedRow;
    }
}
```

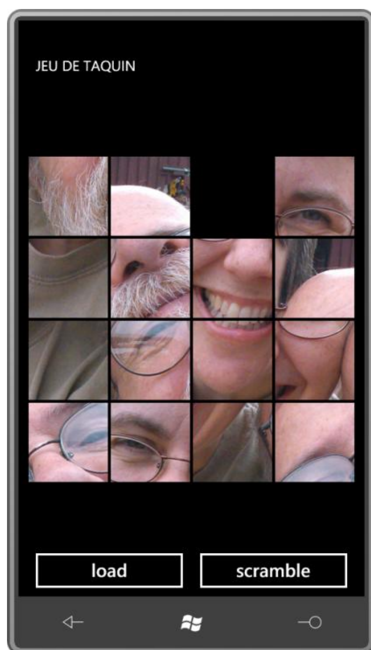
Метод *MoveTile* (Переместить фрагмент) сначала определяет строку и столбец фрагмента, которого коснулся пользователь. Чтобы перемещение состоялось, эта строка должна быть строкой или столбцом с пустым квадратом (она не может быть одновременно и столбцом, и строкой с пустым квадратом). Довольно универсальные циклы *for* обеспечивают перемещение нескольких фрагментов вверх, вниз, влево или вправо.

Логика рандомизации является дополнением к логике перемещения. По нажатию кнопки «scramble» приложение подключает обработчик события *CompositionTarget.Rendering*:

Проект Silverlight: JeuDeTaquin   Файл: MainPage.xaml.cs (фрагмент)

```
void OnScrambleClick(object sender, EventArgs args)
{
    scrambleCountdown = 10 * VERT_TILES * HORZ_TILES;
    scrambleButton.IsEnabled = false;
    appbarScrambleButton.IsEnabled = false;
    CompositionTarget.Rendering += OnCompositionTargetRendering;
}

void OnCompositionTargetRendering(object sender, EventArgs args)
{
    MoveTile(tileImages[emptyRow, rand.Next(HORZ_TILES)]);
    MoveTile(tileImages[rand.Next(VERT_TILES), emptyCol]);

    if (--scrambleCountdown == 0)
    {
        CompositionTarget.Rendering -= OnCompositionTargetRendering;
        scrambleButton.IsEnabled = true;
        appbarScrambleButton.IsEnabled = true;
    }
}
```

Обработчик события вызывает *MoveTile* дважды: один раз для перемещения фрагмента из строки с пустым квадратом и второй раз для перемещения фрагмента из столбца с пустым квадратом.



Данное приложение также обрабатывает захоронение, т.е. оно полностью сохраняет состояние игры при переходе пользователя на другую страницу и восстанавливает это состояние при повторной активации игры.

Мне удалось реализовать сохранение состояние игры с помощью всего нескольких полей. Поле *haveValidTileImages* (Имеются действительные фрагменты) получает значение *true*, если массив *tileImages* содержит действительные элементы *Image*; в противном случае с игрой ничего не происходит. Поля *emptyRow* и *emptyCol* также имеют больше значение. Но важнее всего, конечно же, растровые изображения, которые образуют фрагменты. Вместо того чтобы сохранять весь массив *Pixels* каждого *WriteableBitmap* целиком, я решил сэкономить память, сохраняя эти изображения в формате JPEG со сжатием:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    appService.State["haveValidTileImages"] = haveValidTileImages;

    if (haveValidTileImages)
    {
        appService.State["emptyRow"] = emptyRow;
        appService.State["emptyCol"] = emptyCol;

        for (int row = 0; row < VERT_TILES; row++)
            for (int col = 0; col < HORZ_TILES; col++)
                if (col != emptyCol || row != emptyRow)
                {
                    WriteableBitmap tile = tileImages[row, col].Source as
WriteableBitmap;
                    MemoryStream stream = new MemoryStream();
                    tile.SaveJpeg(stream, tile.PixelWidth, tile.PixelHeight, 0, 75);
                    appService.State[TileKey(row, col)] = stream.GetBuffer();
                }
    }
    base.OnNavigatedFrom(args);
```

```
}

…

string TileKey(int row, int col)
{
    return String.Format("tile {0} {1}", row, col);
}
```

Для каждого элемента *Image* в массиве *tileImages* приложение получает соответствующий *WriteableBitmap* и создает новый *MemoryStream*. Метод расширения *SaveJpeg* позволяет сохранять *WriteableBitmap* в формате JPEG в этот поток. Метод *GetBuffer* (Получить буфер) объекта *MemoryStream* получает массив *byte*, который просто сохраняется с остальными данными состояния.

Когда приложение возвращается из состояния захоронения, этот процесс выполняется в обратной последовательности:

Проект Silverlight: JeuDeTaquin Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    object objHaveValidTileImages;

    if (appService.State.TryGetValue("haveValidTileImages", out
objHaveValidTileImages) &&
        (bool)objHaveValidTileImages)
    {
        emptyRow = (int)appService.State["emptyRow"];
        emptyCol = (int)appService.State["emptyCol"];

        for (int row = 0; row < VERT_TILES; row++)
            for (int col = 0; col < HORZ_TILES; col++)
                if (col != emptyCol || row != emptyRow)
                {
                    byte[] buffer = (byte[])appService.State[TileKey(row, col)];
                    MemoryStream stream = new MemoryStream(buffer);
                    BitmapImage bitmapImage = new BitmapImage();
                    bitmapImage.SetSource(stream);
                    WriteableBitmap tile = new WriteableBitmap(bitmapImage);
                    GenerateImageTile(tile, row, col);
                }

        haveValidTileImages = true;
        appbarScrambleButton.IsEnabled = true;
    }

    base.OnNavigatedTo(args);
}
```

Этот метод читает буфер *byte* и преобразовывает его в *MemoryStream*, из которого создается *BitmapImage* и затем *WriteableBitmap*. После этого с помощью метода *GenerateImageTile* все элементы *Image* создаются и добавляются в *Grid*.

Важно помнить, что этот массив *byte*, используемый для сохранения и восстановления растрового изображения, очень отличается от массива *int*, доступного из свойства *Pixels* объекта *WriteableBitmap*. В массиве *Pixels* хранятся значения каждого пиксела растрового изображения, а массив *byte* – это растровое изображение в формате JPEG со сжатием, включающий все данные и заголовки файла JPEG и прочие подобные данные.

Сохранение в библиотеку изображений

В данной главе мы рассмотрим еще два приложения. Они создают изображения и позволяют сохранять их для потомков, например, в изолированном хранилище, благодаря чему пользователь также может возвращаться к работе над изображением снова и снова.

Но наиболее ценной возможностью для пользователя является сохранение растрового изображения в библиотеке изображений телефона. Специально для этой цели предусмотрена отдельная папка (или «альбом», как принято говорить) под именем «Saved Pictures». В библиотеке изображений пользователь может просматривать получившиеся растровые изображения, отсылать их по электронной почте или в составе текстовых сообщений. Также изображения из библиотеки переносятся на компьютер при обычной синхронизации, после чего могут быть распечатаны.

Доступ к библиотеке изображений обеспечивается посредством классов библиотек XNA, но они могут использоваться из приложений на Silverlight. Для этого потребуется лишь указать ссылку на библиотеку *Microsoft.Xna.Framework* и с помощью директивы *using* подключить пространство имен *Microsoft.Xna.Framework.Media*.

В приложении создается экземпляр класса *MediaLibrary*. Его свойство *SavedPictures* (Сохраненные изображения) возвращает коллекцию *PictureCollection*. Для каждого элемента, хранящегося в настоящее время в альбоме Saved Pictures, в коллекции *PictureCollection* имеется соответствующий объект *Picture*. Пользователь может работать с ними посредством их имен.

Класс *MediaLibrary* также включает метод *SavePicture*, который принимает два обязательных параметра: имя файла и объект *Stream*, ссылающийся на растровое изображение в формате JPEG. Как правило, объект *Stream* типа *MemoryStream*, содержимое которого создано путем вызова метода расширения *SaveJpeg* объекта *WriteableBitmap*.

Приложение Monochromize (Создание черно-белого изображения) позволяет пользователю выбирать изображение из библиотеки изображений. Когда приложение получает фотографию в виде объекта *WriteableBitmap*, оно выполняет доступ к его свойству *Pixels* и преобразовывает это изображение в монохромное изображение. По нажатию кнопки Save выполняется переход к странице, на которой пользователь может ввести имя файла и нажать кнопку OK. При возвращении назад к приложению черно-белое растровое изображение сохраняется в библиотеку изображений под заданным именем.

Страница приложения Monochromize, на которой пользователь может ввести имя файла – это эквивалент Windows Phone 7 традиционному диалоговому окну для сохранения файла, поэтому я назвал ее *SaveFileDialog* (Диалоговое окно для сохранения файла). Этот класс является производным от *PhoneApplicationPage* и входит в состав библиотеки Petzold.Phone.Silverlight.

Для возвращения имени файла в конкретное приложение, использующее страницу *SaveFileDialog*, я применил несколько иную стратегию. Когда пользователь нажимает кнопку «save» или «cancel», *SaveFileDialog* вызывает метод *GoBack* объекта *NavigationService*, как обычно, но при последующем выполнении перегруженного метода *OnNavigatedFrom* он делает попытку вызвать метод главной страницы приложения *SaveFileDialogCompleted* (Работа с диалоговым окном для сохранения файла завершена). Поэтому любая страница, выполняющая переход к *SaveFileDialog*, должна также реализовывать следующий интерфейс:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ISaveFileDialogCompleted.cs

```
namespace Petzold.Phone.Silverlight
{
    public interface ISaveFileDialogCompleted
    {
        void SaveFileDialogCompleted(bool okPressed, string filename);
    }
}
```

Область содержимого страницы *SaveFileDialog* включает традиционный *TextBox* с двумя кнопками, обозначенными надписями «save» и «cancel»:

Проект Silverlight: Petzold.Phone.Silverlight Файл: SaveFileDialog.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <TextBlock Text="file name" />
        <TextBox Name="txtbox"
                 TextChanged="OnTextBoxTextChanged" />
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Button Name="saveButton"
                    Content="save"
                    Grid.Column="0"
                    IsEnabled="False"
                    Click="OnSaveButtonClick" />

            <Button Content="cancel"
                    Grid.Column="2"
                    Click="OnCancelButtonClick" />
        </Grid>
    </StackPanel>
</Grid>
```

В файле выделенного кода также определен открытый метод *SetTitle* (Задать заголовок). Приложение, использующее *SaveFileDialog*, может вызвать этот метод и задать заголовок страницы соответственно имени приложения:

Проект Silverlight: Petzold.Phone.Silverlight Файл: SaveFileDialog.xaml.cs (фрагмент)

```
public partial class SaveFileDialog : PhoneApplicationPage
{
    PhoneApplicationService appService = PhoneApplicationService.Current;
    bool okPressed;
    string filename;

    public SaveFileDialog()
    {
        InitializeComponent();
    }

    public void SetTitle(string appTitle)
    {
        ApplicationTitle.Text = appTitle;
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        saveButton.IsEnabled = txtbox.Text.Length > 0;
    }
```

```
    void OnSaveButtonClick(object sender, RoutedEventArgs args)
    {
        okPressed = true;
        filename = txtbox.Text;
        this.NavigationService.GoBack();
    }

    void OnCancelButtonClick(object sender, RoutedEventArgs args)
    {
        okPressed = false;
        this.NavigationService.GoBack();
    }
    …
}
```

Также обратите внимание, что кнопка «save» неактивна до тех пор, пока в *TextBox* не будет введен хотя бы один символ.

Перегруженные методы навигации выполняют несколько задач. Метод *OnNavigatedTo* проверяет, содержит ли строка запроса имя исходного файла. (Приложение Monochromize не обеспечивает этой проверки, но она реализована в приложении, которое будет рассмотрено далее в данной главе.) Эти методы также обрабатывают захоронение, сохраняя заголовок приложения и любое имя файла, введенное пользователем:

Проект Silverlight: Petzold.Phone.Silverlight Файл: SaveFileDialog.xaml.cs (фрагмент)

```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    if (appService.State.ContainsKey("filename"))
        txtbox.Text = appService.State["filename"] as string;

    if (appService.State.ContainsKey("apptitle"))
        ApplicationTitle.Text = appService.State["apptitle"] as string;

    if (this.NavigationContext.QueryString.ContainsKey("FileName"))
        txtbox.Text = this.NavigationContext.QueryString["FileName"];

    base.OnNavigatedTo(args);
}

protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    if (!String.IsNullOrEmpty(txtbox.Text))
        appService.State["filename"] = txtbox.Text;

    appService.State["apptitle"] = ApplicationTitle.Text;

    if (args.Content is ISaveFileDialogCompleted)
        (args.Content as ISaveFileDialogCompleted).
                                SaveFileDialogCompleted(okPressed, filename);

    base.OnNavigatedFrom(args);
}
```

Самые важные операции описаны в конце метода *OnNavigatedFrom*. Здесь он проверяет, реализует ли страница, к которой выполняется переход, интерфейс *ISaveFileDialogCompleted*; и если да, вызывает метод *SaveFileDialogCompleted* этой страницы.

Описанная в XAML-файле область содержимого приложения Monochromize включает только пустой элемент *Image*:

Проект Silverlight: Monochromize Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img" />
</Grid>
```

ApplicationBar включает две кнопки для загрузки и сохранения:

Проект Silverlight: Monochromize Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton x:Name="appbarLoadButton"
                                       IconUri="/Images/appbar.folder.rest.png"
                                       Text="load"
                                       Click="OnAppbarLoadClick" />

        <shell:ApplicationBarIconButton x:Name="appbarSaveButton"
                                       IconUri="/Images/appbar.save.rest.png"
                                       Text="save"
                                       IsEnabled="False"
                                       Click="OnAppbarSaveClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

В файле выделенного кода всего несколько полей, и только одно из них действительно необходимо. Это *PhotoChooserTask*. Поле *PhoneApplicationService* используется исключительно для удобства. После того как приложение создает объект *WriteableBitmap*, он также сохраняется *Source* как свойство элемента *Image*.

Проект Silverlight: Monochromize Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage, ISaveFileDialogCompleted
{
    PhoneApplicationService appService = PhoneApplicationService.Current;
    PhotoChooserTask photoChooser = new PhotoChooserTask();
    WriteableBitmap writeableBitmap;

    public MainPage()
    {
        InitializeComponent();

        appbarLoadButton = this.ApplicationBar.Buttons[0] as
ApplicationBarIconButton;
        appbarSaveButton = this.ApplicationBar.Buttons[1] as
ApplicationBarIconButton;

        photoChooser.Completed += OnPhotoChooserCompleted;
    }
    …
}
```

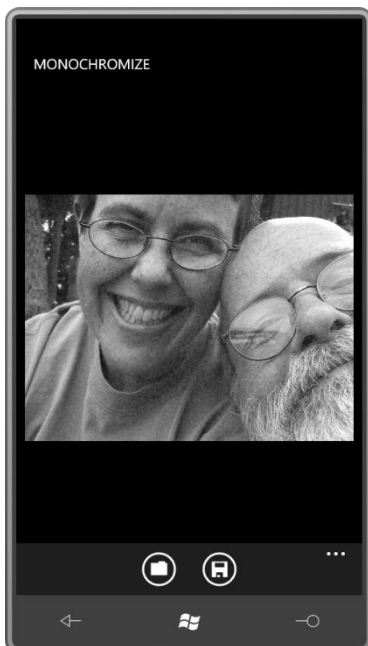
Обратите внимание, что этот класс реализует интерфейс *ISaveFileDialogCompleted*.

По щелчку кнопки «load» запускается *PhotoChooserTask*. По завершению его выполнения обработчик события *Completed* создает объект *WriteableBitmap* и изменяет все члены его массива *Pixels*, применяя стандартные взвешенные значения к значениям красного, зеленого и синего каналов.

Проект Silverlight: Monochromize Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarLoadClick(object sender, EventArgs args)
{
    appbarSaveButton.IsEnabled = false;
    photoChooser.Show();
}

void OnPhotoChooserCompleted(object sender, PhotoResult args)
{
    if (args.Error == null && args.ChosenPhoto != null)
    {
        BitmapImage bitmapImage = new BitmapImage();
        bitmapImage.SetSource(args.ChosenPhoto);
        writeableBitmap = new WriteableBitmap(bitmapImage);

        // Преобразование в черно-белое изображение
        for (int pixel = 0; pixel < writeableBitmap.Pixels.Length; pixel++)
        {
            int color = writeableBitmap.Pixels[pixel];
            byte A = (byte)(color & 0xFF000000 >> 24);
            byte R = (byte)(color & 0x00FF0000 >> 16);
            byte G = (byte)(color & 0x0000FF00 >> 8);
            byte B = (byte)(color & 0x000000FF);
            byte gray = (byte)(0.30 * R + 0.59 * G + 0.11 * B);

            color = (A << 24) | (gray << 16) | (gray << 8) | gray;
            writeableBitmap.Pixels[pixel] = color;
        }
        img.Source = writeableBitmap;
        appbarSaveButton.IsEnabled = true;
    }
}
```

Черно-белый *WriteableBitmap* задается как значение свойства *Source* элемента *Image*, после чего активируется кнопка для сохранения этого изображения.



По нажатию кнопки сохранить выполняется переход на страницу SaveFileDialog.xaml из библиотеки Petzold.Phone.Silverlight. Как мы только что видели, класс *SaveFileDialog*

обрабатывает свой перегруженный *OnNavigatedFrom*, вызывая метод *SaveFileDialogCompleted* класса, к которому выполняется переход:

Проект Silverlight: Monochromize Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarSaveClick(object sender, EventArgs args)
{
    this.NavigationService.Navigate(
            new Uri("/Petzold.Phone.Silverlight;component/SaveFileDialog.xaml",
                UriKind.Relative));
}

public void SaveFileDialogCompleted(bool okPressed, string filename)
{
    if (okPressed)
    {
        MemoryStream memoryStream = new MemoryStream();
        writeableBitmap.SaveJpeg(memoryStream, writeableBitmap.PixelWidth,
                                               writeableBitmap.PixelHeight, 0, 75);
        memoryStream.Position = 0;

        MediaLibrary mediaLib = new MediaLibrary();
        mediaLib.SavePicture(filename, memoryStream);
    }
}
```

При записи растрового изображения в библиотеку изображений метод *SaveFileDialogCompleted* использует имя файла, веденное пользователем. Сохранение изображения выполняется в два этапа: сначала метод *SaveJpeg* записывает *WriteableBitmap* в *MemoryStream* в формате JPEG; после этого свойство *Position* объекта *MemoryStream* сбрасывается в исходное значение, и поток сохраняется в библиотеку изображений.

Приложение Monochromize также обрабатывает захоронение. Метод *OnNavigatedFrom* использует метод расширения *SaveJpeg* для записи в объект *MemoryStream* и затем сохраняет массив *byte*. Этот метод также отвечает за вызов метода *SetTitle* страницы *SaveFileDialog* в случае перехода на нее:

Проект Silverlight: Monochromize Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    if (writeableBitmap != null)
    {
        MemoryStream stream = new MemoryStream();
        writeableBitmap.SaveJpeg(stream, writeableBitmap.PixelWidth,
                                         writeableBitmap.PixelHeight, 0, 75);
        appService.State["jpegBits"] = stream.GetBuffer();
    }

    if (args.Content is SaveFileDialog)
    {
        SaveFileDialog page = args.Content as SaveFileDialog;
        page.SetTitle(ApplicationTitle.Text);
    }

    base.OnNavigatedFrom(args);
}
```

Метод *OnNavigatedTo* отвечает за реактивацию элементов после захоронения. Массив *byte* преобразовывается в *WriteableBitmap*, и активируется кнопка для сохранения:

Проект Silverlight: Monochromize Файл: MainPage.xaml.cs (фрагмент)

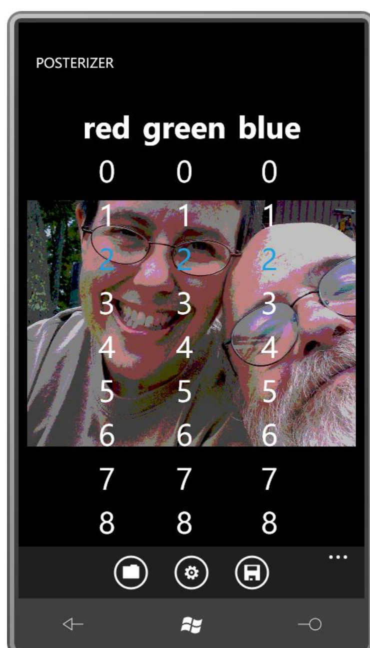
```
protected override void OnNavigatedTo(NavigationEventArgs args)
{
    if (appService.State.ContainsKey("jpegBits"))
    {
        byte[] bitmapBits = (byte[])appService.State["jpegBits"];
        MemoryStream stream = new MemoryStream(bitmapBits);
        BitmapImage bitmapImage = new BitmapImage();
        bitmapImage.SetSource(stream);
        writeableBitmap = new WriteableBitmap(bitmapImage);
        img.Source = writeableBitmap;
        appbarSaveButton.IsEnabled = true;
    }
    base.OnNavigatedTo(args);
}
```

Приложение расширений для обработки фотографий

Архитектурно и функционально приложение Posterizer (Постеризатор), завершающее эту главу, аналогично приложению Monochromize. Оно позволяет пользователю выбирать фотографию из библиотеки изображений и опять сохранять ее в альбом Saved Pictures. Но также с помощью этого приложения пользователь может сокращать битовое разрешение каждого цвета в отдельности (создание эффекта плаката). Для этого в нем предусмотрена строка элементов *RadioButton*. Данное приложение также должно сохранять исходный массив пикселов, чтобы обеспечить возможность восстанавливать изображение в полном цветовом разрешении.

Кроме того, Posterizer регистрирует себя как приложений «расширений для обработки фотографий». Это означает, что пользователь может вызвать его прямо из библиотеки изображений.

Для максимального удобства я решил реализовать элементы управления для выбора битового разрешения, наложив их поверх изображения:



Видимость этого наложения переключается средней кнопкой *ApplicationBar*. Выбранное значение выделяется контрастным цветом.

Наложение реализовано с помощью производного от *UserControl* класса *BitSelectDialog* (Диалоговое окно выбора битового разрешения). Начнем обсуждение с этого элемента управления. Дерево визуальных элементов включает лишь *Grid* с тремя столбцами и девятью строками:

Проект Silverlight: Posterizer   Файл: BitSelectDialog.xaml (фрагмент)

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
</Grid>
```

Каждая ячейка – это *TextBlock*. Логика работы с этими элементами управления в файле выделенного кода реализована таким образом, чтобы они вели себя как переключатели. Открытый интерфейс класса включает событие и открытое свойство, в котором сохраняются три текущие настройки:

Проект Silverlight: Posterizer   Файл: BitSelectDialog.xaml.cs (фрагмент)

```
public event EventHandler ColorBitsChanged;
…
public int[] ColorBits { protected set; get; }
```

В этом уже можно заметить небольшой недостаток. Несмотря на то что метод доступа для задания значения массива *ColorBits* (Биты цвета) является защищенным, и вносить изменения в массив из внешнего класса невозможно, отдельные члены этого массива могут быть заданы. При этом нет никакой возможности проинформировать класс об этом, кроме как сформировать событие *ColorBitsChanged* (Биты цвета изменены). Но я решил не усложнять код класса и не устранять этот недостаток.

Все элементы *TextBlock* создаются в конструкторе класса. Обратите внимание, что изначально в массиве *ColorBits* хранится три числа 2.

Проект Silverlight: Posterizer   Файл: BitSelectDialog.xaml.cs (фрагмент)

```
public partial class BitSelectDialog : UserControl
{
    Brush selectedBrush;
    Brush normalBrush;
    TextBlock[,] txtblks = new TextBlock[3, 9];

    …
```

```
    public BitSelectDialog()
    {
        InitializeComponent();

        ColorBits = new int[3];
        ColorBits[0] = 2;
        ColorBits[1] = 2;
        ColorBits[2] = 2;

        selectedBrush = this.Resources["PhoneAccentBrush"] as Brush;
        normalBrush = this.Resources["PhoneForegroundBrush"] as Brush;
        string[] colors = { "red", "green", "blue" };

        for (int col = 0; col < 3; col++)
        {
            TextBlock txtblk = new TextBlock
            {
                Text = colors[col],
                FontWeight = FontWeights.Bold,
                TextAlignment = TextAlignment.Center,
                Margin = new Thickness(8, 2, 8, 2)
            };

            Grid.SetRow(txtblk, 0);
            Grid.SetColumn(txtblk, col);
            LayoutRoot.Children.Add(txtblk);

            for (int bit = 0; bit < 9; bit++)
            {
                txtblk = new TextBlock
                {
                    Text = bit.ToString(),
                    Foreground = bit == ColorBits[col] ? selectedBrush :
normalBrush,
                    TextAlignment = TextAlignment.Center,
                    Padding = new Thickness(2),
                    Tag = col.ToString() + bit
                };

                Grid.SetRow(txtblk, bit + 1);
                Grid.SetColumn(txtblk, col);
                LayoutRoot.Children.Add(txtblk);

                txtblks[col, bit] = txtblk;
            }
        }
    }
}
```

В качестве значения свойства *Tag* каждого *TextBlock* задана строка из двух символов, которые обозначают цвет и количество бит, ассоциированных с этим элементом.

Я также описал открытый метод, с помощью которого приложение инициирует три значения *ColorBits*. При этом цвета *TextBlock* меняются, но события *ColorBitsChanged* не формируются. Это пригодится при повторной активации приложения после захоронения.

Проект Silverlight: Posterizer Файл: BitSelectDialog.xaml.cs (фрагмент)

```
public void Initialize(int[] colorBits)
{
    for (int clr = 0; clr < 3; clr++)
    {
        txtblks[clr, ColorBits[clr]].Foreground = normalBrush;
        ColorBits[clr] = colorBits[clr];
```

```
        txtblks[clr, ColorBits[clr]].Foreground = selectedBrush;
    }
}
```

Перегруженный метод *OnManipulationStarted* декодирует значение свойства *Tag* элемента управления *TextBlock*, которого коснулся пользователь, и определяет выбор пользователя:

Проект Silverlight: Posterizer Файл: BitSelectDialog.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    if (args.OriginalSource is TextBlock)
    {
        TextBlock txtblk = args.OriginalSource as TextBlock;
        string tag = txtblk.Tag as string;

        if (tag != null && tag.Length == 2)
        {
            int clr = Int32.Parse(tag[0].ToString());
            int bits = Int32.Parse(tag[1].ToString());

            if (ColorBits[clr] != bits)
            {
                txtblks[clr, ColorBits[clr]].Foreground = normalBrush;
                ColorBits[clr] = bits;
                txtblks[clr, ColorBits[clr]].Foreground = selectedBrush;

                if (ColorBitsChanged != null)
                    ColorBitsChanged(this, EventArgs.Empty);
            }

            args.Complete();
            args.Handled = true;
        }
    }
    base.OnManipulationStarted(args);
}
```

На основании декодированных данных свойства *Tag* элемента управления *TextBlock*, которого коснулся пользователь, метод может изменить цвет этого *TextBlock* (при этом выбор с этого элемента снимается), сохранить новое значение в массиве *ColorBits* и сформировать событие *ColorBitsChanged*.

Область содержимого класса *MainPage* включает (уже традиционно) пустой элемент *Image* и вот такой элемент управления *BitSelectDialog*:

Проект Silverlight: Posterizer Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

    <Image Name="img" />

    <local:BitSelectDialog x:Name="bitSelectDialog"
                           Visibility="Collapsed"
                           FontSize="{StaticResource PhoneFontSizeExtraLarge}"
                           HorizontalAlignment="Center"
                           VerticalAlignment="Center"
                           ColorBitsChanged="OnBitSelectDialogColorBitsChanged" />
</Grid>
```

Обратите внимание, что элемент управления *BitSelectDialog* имеет собственное свойство *Visibility*, которому задано значение *Collapsed*.

ApplicationBar в XAML-файле включает три кнопки:

Проект Silverlight: Posterizer Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton x:Name="appbarLoadButton"
                                        IconUri="/Images/appbar.folder.rest.png"
                                        Text="load"
                                        Click="OnAppbarLoadClick" />

        <shell:ApplicationBarIconButton x:Name="appbarSetBitsButton"

IconUri="/Images/appbar.feature.settings.rest.png"
                                        Text="set bits"
                                        IsEnabled="False"
                                        Click="OnAppbarSetBitsClick" />

        <shell:ApplicationBarIconButton x:Name="appbarSaveButton"
                                        IconUri="/Images/appbar.save.rest.png"
                                        Text="save"
                                        IsEnabled="False"
                                        Click="OnAppbarSaveClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

В файле выделенного кода поля класса *MainPage* включают три переменные, представляющие растровые изображения: переменную типа *WriteableBitmap*, массив *byte* и массив *int*.

Проект Silverlight: Posterizer Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage, ISaveFileDialogCompleted
{
    PhoneApplicationService appService = PhoneApplicationService.Current;
    PhotoChooserTask photoChooser = new PhotoChooserTask();
    WriteableBitmap writeableBitmap;
    byte[] jpegBits;
    int[] pixels;

    public MainPage()
    {
        InitializeComponent();

        appbarLoadButton = this.ApplicationBar.Buttons[0] as
ApplicationBarIconButton;
        appbarSetBitsButton = this.ApplicationBar.Buttons[1] as
ApplicationBarIconButton;
        appbarSaveButton = this.ApplicationBar.Buttons[2] as
ApplicationBarIconButton;

        photoChooser.Completed += OnPhotoChooserCompleted;
    }
}
```

Поле *WriteableBitmap* – это растровое изображение, заданное как значение элемента *Image* и отображаемое на экране. Этот то растровое изображение, пикселы которого были изменены для понижения цветового разрешения. Массив *jpegBits* – это исходный файл, загружаемый пользователем из библиотеки изображений. Массив *jpegBits* удобно сохранять в целях

захоронения, это гарантирует, что фотография, восстановленная после захоронения, полностью аналогична изначально загруженной. В массиве *pixels* хранятся неизмененные пикселы загруженного растрового изображения, но этот массив *не* сохраняется при захоронении. Для сохранения массива *jpegBits* по сравнению с *pixels* требуется намного меньше памяти.

Когда пользователь нажимает кнопку «load», происходит инициация класса *PhotoChooserTask*. В ходе обработки события *Completed* приложение задает *jpegBits* из потока *ChosenPhoto* и затем вызывает *LoadBitmap* (Загрузить растровое изображение).

Проект Silverlight: Posterizer Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarLoadClick(object sender, EventArgs args)
{
    bitSelectDialog.Visibility = Visibility.Collapsed;
    appbarSetBitsButton.IsEnabled = false;
    appbarSaveButton.IsEnabled = false;

    photoChooser.Show();
}

void OnPhotoChooserCompleted(object sender, PhotoResult args)
{
    if (args.Error == null && args.ChosenPhoto != null)
    {
        jpegBits = new byte[args.ChosenPhoto.Length];
        args.ChosenPhoto.Read(jpegBits, 0, jpegBits.Length);
        LoadBitmap(jpegBits);
    }
}

void LoadBitmap(byte[] jpegBits)
{
    // Создаем WriteableBitmap из массива jpegBits
    MemoryStream memoryStream = new MemoryStream(jpegBits);
    BitmapImage bitmapImage = new BitmapImage();
    bitmapImage.SetSource(memoryStream);
    writeableBitmap = new WriteableBitmap(bitmapImage);
    img.Source = writeableBitmap;

    // Копируем пикселы в массив pixels
    pixels = new int[writeableBitmap.PixelWidth * writeableBitmap.PixelHeight];

    for (int i = 0; i < pixels.Length; i++)
        pixels[i] = writeableBitmap.Pixels[i];

    appbarSetBitsButton.IsEnabled = true;
    appbarSaveButton.IsEnabled = true;
    ApplyBitSettingsToBitmap();
}
```

После этого метод *LoadBitmap* преобразует этот массив *byte* назад в *MemoryStream* для создания *BitmapImage* и *WriteableBitmap*. Может показаться, что так мы создаем растровое изображение окольными путями, но это имеет смысл с точки зрения захоронения.

После этого метод *LoadBitmap* создает копию массива *Pixels* объекта *WriteableBitmap* в виде поля *pixels*. Это поле *pixels* будет оставаться неизменным, в то время как массив *Pixels* объекта *WriteableBitmap* меняется, исходя из выбираемого пользователем битового разрешения. Наша задача – отсутствие необратимых операций. Пользователь всегда должен иметь возможность выбрать более высокое цветовое разрешение после применения более низкого.

Метод *ApplyBitSettingsToBitmap* (Применить настройки битового разрешения к растровому изображению), вызываемый в конце выполнения *LoadBitmap*, также вызывается каждый раз, когда *BitSelectDialog* формирует событие *ColorBitsChanged*. Видимость этого диалогового окна переключается по нажатию средней кнопки *ApplicationBar*:

Проект Silverlight: Posterizer Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarSetBitsClick(object sender, EventArgs args)
{
    bitSelectDialog.Visibility =
        bitSelectDialog.Visibility == Visibility.Collapsed ?
            Visibility.Visible : Visibility.Collapsed;
}

void OnBitSelectDialogColorBitsChanged(object sender, EventArgs args)
{
    ApplyBitSettingsToBitmap();
}

void ApplyBitSettingsToBitmap()
{
    if (pixels == null || writeableBitmap == null)
        return;

    int mask = -16777216;   // например, FF000000

    for (int clr = 0; clr < 3; clr++)
        mask |= (byte)(0xFF << (8 - bitSelectDialog.ColorBits[clr]))
                                << (16 - 8 * clr);

    for (int i = 0; i < pixels.Length; i++)
        writeableBitmap.Pixels[i] = mask & pixels[i];

    writeableBitmap.Invalidate();
}
```

Переменная *mask* (маска) образуется из трех значений битового разрешения и затем применяется ко всем значениям поля *pixels*, что обеспечивает задание всех значений массива *Pixels* объекта *WriteableBitmap*.

Когда пользователь нажимает кнопку, чтобы сохранить файл в библиотеку изображений, приложение Posterizer предлагает пользователю имя файла, образованное именем данного приложения (Posterizer), за которым следует трехзначное число, превышающее номера всех имеющихся в библиотеке изображений. Для этого приложение выполняет доступ к альбому сохраненных изображений посредством свойства *SavedPictures* объекта *MediaLibrary* и выполняет поиск соответствующих имен файлов:

Проект Silverlight: Posterizer Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarSaveClick(object sender, EventArgs args)
{
    int fileNameNumber = 0;
    MediaLibrary mediaLib = new MediaLibrary();
    PictureCollection savedPictures = mediaLib.SavedPictures;

    foreach (Picture picture in savedPictures)
    {
        string filename = Path.GetFileNameWithoutExtension(picture.Name);
        int num;

        if (filename.StartsWith("Posterizer"))
```

```
        {
            if (Int32.TryParse(filename.Substring(10), out num))
                fileNameNumber = Math.Max(fileNameNumber, num);
        }
    }

    string saveFileName = String.Format("Posterizer{0:D3}", fileNameNumber + 1);

    string uri = "/Petzold.Phone.Silverlight;component/SaveFileDialog.xaml" +
                 "?FileName=" + saveFileName;

    this.NavigationService.Navigate(new Uri(uri, UriKind.Relative));

}

public void SaveFileDialogCompleted(bool okPressed, string filename)
{
    if (okPressed)
    {
        MemoryStream memoryStream = new MemoryStream();
        writeableBitmap.SaveJpeg(memoryStream, writeableBitmap.PixelWidth,
                                               writeableBitmap.PixelHeight, 0, 75);
        memoryStream.Position = 0;

        MediaLibrary mediaLib = new MediaLibrary();
        mediaLib.SavePicture(filename, memoryStream);
    }
}
```

Приложение, обрабатывающее фотографии из библиотеки изображений, имеет опцию, позволяющую сделать его приложением «расширений для обработки фотографий». В этом случае при выборе фотографии в библиотеке изображений пользователь будет видеть меню, включающее пункт «extras» (расширения). По нажатию этого пункта пользователь получит список всех приложений, зарегистрировавшихся как приложения «расширений для обработки фотографий». Выбранное из данного списка приложение будет запускаться уже с загруженной в него фотографией.

К файлу проекта приложения следует добавить еще такой файл:

Проект Silverlight: Posterizer Файл: Extras.xml

```
<Extras>
    <PhotosExtrasApplication>
        <Enabled>true</Enabled>
    </PhotosExtrasApplication>
</Extras>
```

В разделе Properties этого файла для Build Action должно быть выбрано Content и для Copy to Output Directory (Копировать в выходной каталог) – Copy Always (Копировать всегда).

Приложение также должно быть готово обрабатывать специальный вызов *OnNavigatedTo* для отображения выбранной фотографии.

Рассмотрим оба перегруженных метода навигации. Метод *OnNavigatedFrom* определяет, выполняется ли захоронение приложения или переход к объекту *SaveFileDialog*. В случае захоронения приложение должно сохранить и выбранное в настоящий момент цветовое разрешение, и растровое изображение (если таковое существует). При переходе к *SaveFileDialog* метод задает заголовок страницы.

Проект Silverlight: Posterizer Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    appService.State["colorBits"] = bitSelectDialog.ColorBits;

    if (jpegBits != null)
    {
        appService.State["jpegBits"] = jpegBits;
    }

    if (args.Content is SaveFileDialog)
    {
        SaveFileDialog page = args.Content as SaveFileDialog;
        page.SetTitle(ApplicationTitle.Text);
    }

    base.OnNavigatedFrom(args);
}

protected override void OnNavigatedTo(NavigationEventArgs args)
{
    if (this.NavigationContext.QueryString.ContainsKey("token"))
    {
        string token = this.NavigationContext.QueryString["token"];

        MediaLibrary mediaLib = new MediaLibrary();
        Picture picture = mediaLib.GetPictureFromToken(token);
        Stream stream = picture.GetImage();
        jpegBits = new byte[stream.Length];
        stream.Read(jpegBits, 0, jpegBits.Length);
        LoadBitmap(jpegBits);
    }
    else if (appService.State.ContainsKey("colorBits"))
    {
        int[] colorBits = (int[])appService.State["colorBits"];
        bitSelectDialog.Initialize(colorBits);
    }

    if (appService.State.ContainsKey("jpegBits"))
    {
        jpegBits = (byte[])appService.State["jpegBits"];
        LoadBitmap(jpegBits);
    }
    base.OnNavigatedTo(args);
}
```

Метод *OnNavigatedTo* может показать, что приложение вызвано из библиотеки изображений. В этом случае словарь *QueryString* объекта *NavigationContext* будет содержать специальный строковый ключ «token». Элемент, соответствующий этой строке, передается в специальный метод *GetPictureFromToken* (Получить изображение по маркеру) объекта *MediaLibrary* для получения потока в памяти, из которого можно осуществить доступ к JPEG-файлу.

Как я говорил ранее, метод *LoadBitmap* удобно использовать для повторной активации приложения после захоронения, и логика, реализованная в конце этого метода, подтверждает это.

Глава 15
Анимации

Anima в переводе с латыни примерно означает «*жизненная сила*», очень близкое к греческому *psyche* – «дух». Таким образом, введение анимации в приложения можно рассматривать как процесс «оживления» неживых (или *неодушевленных*) объектов.

В предыдущих главах мы рассмотрели, как изменять местоположение элементов на экране посредством касания или на основании периодически формируемых объектом *DispatcherTimer* событий *Tick*. Также было показано событие *CompositionTarget.Rendering*, с помощью которого приложение может создавать анимации через изменение визуальных элементов синхронно с обновлением экрана.

И *DispatcherTimer*, и *CompositionTarget.Rendering* могут быть очень полезны, но для большинства задач анимации проще и лучше использовать встроенную поддержку анимации Silverlight. Это более 50 классов, структур и перечислений пространства имен *System.Windows.Media.Animation*.

С библиотекой классов Silverlight для создания анимаций работать проще, чем с ее альтернативами, отчасти потому что она позволяет описывать анимации в XAML. Также эти анимации предпочтительнее, чем *CompositionTarget.Rendering*, потому что некоторые основные типы анимаций используют графический процессор (Graphics Processing Unit, GPU) телефона. Эти анимации выполняются не в потоке пользовательского интерфейса, а в отдельном потоке, называемом потоком *компоновщика* (*compositor thread*) или *обрабатывающим* потоком (*render thread*).

Анимации играют важную роль в шаблонах элементов управления, которые используются для переопределения визуальных составляющих элементов управления. Элементы управления имеют *состояния*, например, ассоциированное с *Button* состояние *Pressed*, и все смены состояний реализуются с помощью анимаций. Шаблоны элементов управления будут рассмотрены в следующей главе.

Со временем многие разработчики приходят к использованию Expression Blend для описания анимаций и шаблонов элементов управления. Это нормально, но в данной главе я собираюсь показать, как создавать анимации вручную. Также будет продемонстрирована ценная (но часто игнорируемая) техника определения анимаций в коде, а не в XAML.

Сравнение анимации, основанной на кадрах, и анимации, использующей временную шкалу

Предположим, нам надо написать небольшое приложение, в котором вращение текста реализовано посредством события *CompositionTarget.Rendering*. Синхронизировать анимацию можно либо по частоте обновления экрана, либо по времени. Поскольку результат каждого отдельно взятого обновления экрана называется *кадром*, такой метод синхронизации анимации называют *синхронизацией по кадрам* (*frame-based*). Также анимация может *синхронизироваться по времени* (*time-based*).

Рассмотрим небольшое приложение, которое демонстрирует, в чем разница. Область содержимого XAML-файла включает два элемента *TextBlock*, в качестве значений свойств *RenderTransform* которых заданы объекты *RotateTransform*, и *Button*:

Проект Silverlight: FrameBasedVsTimeBased Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0"
               Text="Frame-Based"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate1" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Grid.Row="1"
               Text="Time-Based"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate2" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <Button Grid.Row="2"
            Content="Hang for 5 seconds"
            HorizontalAlignment="Center"
            Click="OnButtonClick" />
</Grid>
```

Файл выделенного кода сохраняет текущее время в специальном поле и затем задает обработчик события *CompositionTarget.Rendering*. После этого данный обработчик вызывается синхронно со сменой кадров.

Проект Silverlight: FrameBasedVsTimeBased Файл: MainPage.xaml.cs (фрагмент)

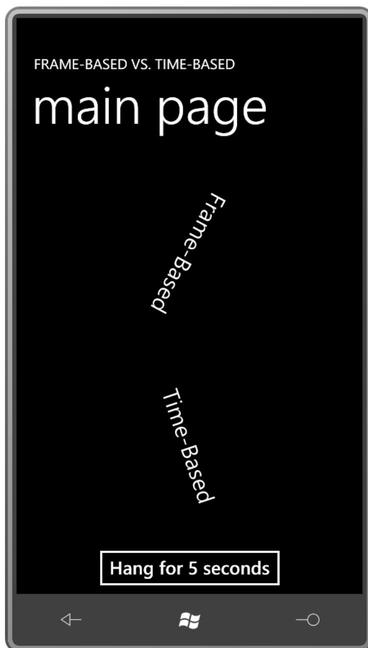
```
public partial class MainPage : PhoneApplicationPage
{
    DateTime startTime;

    public MainPage()
    {
        InitializeComponent();

        startTime = DateTime.Now;
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, EventArgs args)
    {
        // С синхронизацией по кадрам
        rotate1.Angle = (rotate1.Angle + 0.2) % 360;

        // С синхронизацией по времени
        TimeSpan elapsedTime = DateTime.Now - startTime;
        rotate2.Angle = (elapsedTime.TotalMinutes * 360) % 360;
    }
```

```
    void OnButtonClick(object sender, RoutedEventArgs args)
    {
        Thread.Sleep(5000);
    }
}
```

Угол поворота первого *TextBlock* увеличивается на 0,2° при каждой смене кадров. Я вычислил это, исходя из того что экран телефона обновляется с частотой 30 кадров в секунду. Умножаем 30 кадров в секунду на 60 секунд и на 0,2° и получаем 360° в минуту.

Угол поворота второго *TextBlock* вычисляется на основании истекшего времени. Структура *TimeSpan* имеет очень удобное свойство *TotalMinutes* (Общее количество минут). Его значение умножается на 360, что позволяет получить угол поворота текста в градусах.

Оба метода работают и демонстрируют даже примерно одинаковую производительность:



Но если вы используете телефон, подобный моему, со временем вы заметите, что анимация с синхронизацией по кадрам немного отстает от анимации с синхронизацией по времени, и это отставание растет. Почему?

В вычислениях для анимации с синхронизацией по кадрам предполагается, что обработчик события *CompositionTarget.Rendering* вызывается с частотой 30 раз в секунду. Но в используемом для данного примера телефоне частота обновления экрана около 27 кадров в секунду (около 27,35, если быть абсолютно точным).

В этом основная проблема анимации с синхронизацией по кадрам – зависимость от оборудования. Ее скорость может меняться.

Вот еще одно отличие. Предположим, приложению неожиданно требуется выполнить какую-то задачу, которая очень сильно загружает процессор. В приложении FrameBasedVsTimeBased такая задача моделируется с помощью кнопки *Button*, которая приостанавливает поток на 5 секунд. Обе анимации приостановятся на это время, потому что вызовы обработчика *CompositionTarget.Rendering* не являются асинхронными. Когда выполнение возобновится, анимация с синхронизацией по кадрам продолжится с момента, на котором приостановилась, а анимация с синхронизацией по времени перепрыгнет в то

положение, в котором она должна была бы находиться, если бы этой задержки не произошло.

То какой подход будет выбран, полностью зависит от приложения. Иногда действительно необходимо использовать анимацию с синхронизацией по кадрам. Но для реализации движения стрелок часов или чего-либо подобного подходит только анимация с синхронизацией по времени; анимация с синхронизацией по кадрам слишком непредсказуема, чтобы использоваться для таких задач.

Все классы библиотеки классов Silverlight для создания анимаций являются синхронизируемыми по времени. Разработчик задает, какое изменение должно быть выполнено через заданный период времени, все остальное делают классы анимаций Silverlight.

Цели анимации

Анимации в Silverlight реализовываются путем изменения определенного свойства определенного объекта, например, свойства *Opacity* объекта *Image*. Изменение значения свойства *Opacity* во времени приводит к созданию эффекта постепенного проявления элемента *Image*, или постепенного его исчезновения, или проявления и исчезновения, в зависимости от предъявляемых требований.

Цель анимации должна быть свойством-зависимостью! Очевидно что свойство-зависимость должно быть описано классом, наследуемым от *DependencyObject*.

Классы анимаций различают по *muny* анимируемого свойства. Целями анимаций Silverlight могут быть свойства типа *double*, *Color*, *Point* и *Object*. (Можно подумать, что вошедший в этот список *Object* объединяет в себе все остальные типы, но вскоре мы увидим, что анимации *Object* крайне ограничены в своей функциональности.)

Свойства типа *double* очень широко используются в Silverlight. К ним относятся свойства *Opacity*, *Canvas.Left* и *Canvas.Top*, *Height* и *Width*, а также все свойства классов трансформаций: свойства *X* и *Y* класса *TranslateTransform*, свойства *ScaleX* и *ScaleY* класса *ScaleTransform* и свойство *Angle* класса *RotateTransform*. Анимация трансформаций – самый распространенный и наиболее эффективный способ применения анимаций.

Также в данной главе я покажу, как использовать свойство *Projection* (Проекция), описанное элементом *UIElement*, для создания подобных трехмерной графике эффектов перспективы.

Включите фантазию и, встречая свойство-зависимость типа *double*, *Color* или *Point*, подумайте, как можно было бы анимировать его и создать интересный визуальный эффект. Например, анимацией свойства *Offset* объектов *GradientStop* можно обеспечить изменение градиентных кистей во времени, или анимация свойства *StrokeDashOffset* объекта *Shape* обеспечит перемещение точек и штрихов вдоль линий. (Напомните мне продемонстрировать пример этого!)

Анимировать свойство типа *Color* можно только для кистей *SolidColorBrush*, *LinearGradientBrush* и *RadialGradientBrush*.

Свойства типа *Point* довольно редко используются в Silverlight, только для объектов *Geometry*. Далее в данной главе будет приведена пара примеров анимации этих свойств.

Классы *DoubleAnimation* (Анимация свойств типа Double), *ColorAnimation* (Анимация свойств типа Color) и *PointAnimation* (Анимация свойств типа Point) позволяют анимировать свойства типа *double*, *Color* или *Point* непрерывно из одного значения в другое и, возможно, в обратном направлении один или много раз. (Даже после многих лет разработки приложений

на WPF и Silverlight мне по-прежнему хочется перевести *DoubleAnimation* как «двойная анимация». Но нет! Это анимация, целью которой являются свойства типа *double*, т.е. с плавающей точкой двойной точности.)

У классов *DoubleAnimation*, *ColorAnimation* и *PointAnimation* есть свойство *Easing* (Сглаживание). В качестве его значения может быть задан экземпляр одного из множества классов, которые позволяют менять скорость анимации в начале или в конце (или и там, и там) или даже кратковременно отклоняться от целевого значения анимации, делая движение более естественным.

Классы *DoubleAnimationUsingKeyFrames* (Анимация свойств типа Double с использованием ключевых кадров),*ColorAnimationUsingKeyFrames* (Анимация свойств типа Color с использованием ключевых кадров) и *PointAnimationUsingKeyFrames* (Анимация свойств типа Point с использованием ключевых кадров) позволяют объединять и создавать более сложные анимации. У этих классов есть свойство *KeyFrames* (Ключевые кадры), которое является коллекцией объектов ключевых кадров, указывающих на то, каким должно быть значение целевого свойства по истечении определенного времени.

Например, объект *DoubleAnimationUsingKeyFrames* может включать отдельные ключевые кадры типа *DiscreteDoubleKeyFrame* (Дискретный ключевой кадр типа Double) (для скачкообразного перехода к определенному значению в определенное время), *LinearDoubleKeyFrame* (Линейный ключевой кадр типа Double) (для перехода с постоянной скоростью, обеспечивающей достижение свойством определенного значения в определенное время), *SplineDoubleKeyFrame* (Сплайновый ключевой кадр типа Double) (позволяющий задавать анимацию, которая ускоряется или замедляется по сплайну Безье) и *EasingDoubleKeyFrame* (Сглаживающий ключевой кадр типа Double) (позволяющий применять одну из функций сглаживания). Аналогичные классы существуют для *ColorAnimationUsingKeyFrames* и *PointAnimationUsingKeyFrames*.

Теоретически мы можем анимировать свойства типа *Object*, но для этого в нашем распоряжении есть только классы *ObjectAnimationUsingKeyFrames* и *DiscreteObjectKeyFrame*. Это означает, что мы ограничены только анимациями по дискретным значениям. Практически всегда эта возможность используется для анимации свойств, являющихся перечислениями, таких как *Visibility*.

Щелчок и разворот

Предположим, требуется расширить функциональность кнопки и обеспечить дополнительную визуальную обратную связь пользователю. Решено сделать обратную связь *очень эффектной*, чтобы она могла разбудить даже сонного пользователя, и поэтому кнопка будет вращаться по кругу при каждом нажатии.

XAML-файл описывает несколько кнопок:

Проект Silverlight: ClickAndSpin Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Button Content="Button No. 1"
            Grid.Row="0"
            HorizontalAlignment="Center"
```

```
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform />
        </Button.RenderTransform>
    </Button>

    <Button Content="Button No. 2"
            Grid.Row="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform />
        </Button.RenderTransform>
    </Button>

    <Button Content="Button No. 3"
            Grid.Row="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform />
        </Button.RenderTransform>
    </Button>
</Grid>
```

Свойству *RenderTransform* каждой из кнопок задан *RotateTransform*, и в качестве *RenderTransformOrigin* задан центр элемента.

Обработчик события *Click* отвечает за определение и запуск анимации вращения нажатой кнопки. (Конечно, в реальном приложении обработчик *Click* также осуществлял бы какие-то важные операции!) Вначале обработчик получает объект *Button*, которого коснулся пользователь, и *RotateTransform*, ассоциированный с этим конкретным *Button*:

Проект Silverlight: ClickAndSpin Файл: MainPage.xaml.cs (фрагмент)

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
    Button btn = sender as Button;
    RotateTransform rotateTransform = btn.RenderTransform as RotateTransform;

    // Создаем и описываем анимацию
    DoubleAnimation anima = new DoubleAnimation();
    anima.From = 0;
    anima.To = 360;
    anima.Duration = new Duration(TimeSpan.FromSeconds(0.5));

    // Задаем присоединенные свойства
    Storyboard.SetTarget(anima, rotateTransform);
    Storyboard.SetTargetProperty(anima, new
PropertyPath(RotateTransform.AngleProperty));

    // Создаем раскадровку, добавляем анимацию и запускаем ее!
    Storyboard storyboard = new Storyboard();
    storyboard.Children.Add(anima);
    storyboard.Begin();
}
```

Реализация анимации в приложении включает три этапа:

1. Определение самой анимации. Целью необходимой здесь анимации будет свойство *Angle* объекта *RotateTransform*. Поскольку свойство *Angle* типа *double*, используем *DoubleAnimation*:

   ```
   DoubleAnimation anima = new DoubleAnimation();
   anima.From = 0;
   anima.To = 360;
   anima.Duration = new Duration(TimeSpan.FromSeconds(0.5));
   ```

   DoubleAnimation будет изменять свойство типа от значения 0 до значения 360 за ½ секунды. Свойство *Duration* (Продолжительность) объекта *DoubleAnimation* типа *Duration*. В коде его принято задавать на основании объекта *TimeSpan*, но свойство *Duration* не типа *TimeSpan* (главным образом, так сложилось исторически). В качестве альтернативы свойству *Duration* можно задать статическое значение *Duration.Automatic*, что равноценно незаданию этого свойства (или заданию значения *null*) и обеспечивает создание анимации продолжительностью 1 секунда.

2. Задание присоединенных свойств. *DoubleAnimation* должен быть ассоциирован с конкретным объектом и свойством этого объекта. Для этого используются два присоединенных свойства, определенных классом *Storyboard* (Раскадровка):

   ```
   Storyboard.SetTarget(anima, rotateTransform);
   Storyboard.SetTargetProperty(anima, new
   PropertyPath(RotateTransform.AngleProperty));
   ```

   Этими присоединенными свойствами являются *Target* и *TargetProperty* (Целевое свойство). Как помните, при задании присоединенных свойств в коде используются статические методы, начинающиеся со слова *Set*.

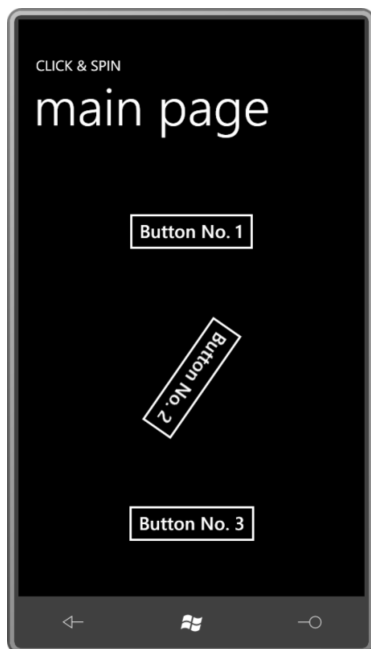
   В обоих случаях первым аргументом является только что созданный *DoubleAnimation*. Вызов *SetTarget* (Задать цель) указывает на объект, к которому будет применяться анимация (в данном случае это *RotateTransform*), и вызов *SetTargetProperty* (Задать целевое свойство) указывает на свойство этого объекта. Второй аргумент метода *SetTargetProperty* типа *PropertyPath* (Путь к свойству). Как видите, я указал полностью определенное свойство-зависимость для свойства *Angle* объекта *RotateTransform*.

3. Определяем, задаем и запускаем *Storyboard*.

   На данный момент, кажется, все готово, но необходимо сделать еще один шаг. В Silverlight анимации всегда заключены в объекты *Storyboard*. У отдельного *Storyboard* может быть множество потомков, поэтому с его помощью очень удобно синхронизировать множество анимаций. Но даже если требуется реализовать всего одну независимо выполняющуюся анимацию, все равно понадобится *Storyboard*:

   ```
   Storyboard storyboard = new Storyboard();
   storyboard.Children.Add(anima);
   storyboard.Begin();
   ```

   Вызов метода *Begin* объекта *Storyboard* выполнен, и теперь любая кнопка после нажатия делает полный оборот за полсекунды, что обеспечивает, вероятно, слишком *интенсивную* визуальную обратную связь для пользователя.



Некоторые вариации

Я задал целевое свойство анимации, используя полное имя свойства-зависимости:

```
Storyboard.SetTargetProperty(anima, new PropertyPath(RotateTransform.AngleProperty));
```

Альтернативный вариант – использование строки:

```
Storyboard.SetTargetProperty(anima, new PropertyPath("Angle"));
```

Этот синтаксис может казаться более привлекательным, поскольку он короче, однако, в этом случае велика вероятность ошибочного написания имени свойства.

Преимущество применения строк для задания свойств в том, что так мы можем задавать несколько имен свойств подряд. Это позволяет применять анимацию к свойству объекта без ссылки на сам объект. Например, в приведенном выше обработчике событий *Click* целью анимации можно задать *Button*, а не *RotateTransform*:

```
Storyboard.SetTarget(anima, btn);
```

RotateTransform по-прежнему должен присутствовать, и мы по-прежнему должны указывать, что целью является свойство *Angle* этого объекта, но посмотрите, как это делается:

```
Storyboard.SetTargetProperty(anima,
        new PropertyPath("(Button.RenderTransform).(RotateTransform.Angle)"));
```

Этот синтаксис (несомненно более типичный для XAML, чем для кода) показывает, что *RenderTransform* является свойством *Button*, и *Angle* – свойство *RotateTransform*, и *RotateTransform* задан как значение свойства *RenderTransform*. Если объект *RotateTransform* не задан как значение свойства *RenderTransform*, ничего не получится.

Этот синтаксис можно немного упростить, удалив развернутое описание свойства *Angle*:

```
Storyboard.SetTargetProperty(anima,
        new PropertyPath("(Button.RenderTransform).Angle"));
```

Но такой синтаксис может быть непонятным. Кажется, что *Angle* является свойством *RenderTransform*, а это на самом деле не так. *Angle* – это свойство объекта *RotateTransform*, который задан как значение свойства *RenderTransform*.

Независимо от того как оно задается, анимированное свойство должно быть свойством-зависимостью.

Классы *Storyboard* и *DoubleAnimation* на самом деле являются классами одного уровня, как видно из следующей иерархии классов:

Object
  DependencyObject (абстрактный)
    Timeline (абстрактный)
      DoubleAnimation
      DoubleAnimationUsingKeyFrames
      ColorAnimation
      ColorAnimationUsingKeyFrames
      PointAnimation
      PointAnimationUsingKeyFrames
      ObjectAnimationUsingKeyFrames
      Storyboard

Storyboard определяет свойство *Children* типа *TimelineCollection* (Коллекция временной шкалы). Это означает, что *Storyboard* может включать не только объекты анимации, но и другие объекты *Storyboard* для управления сложными коллекциями анимаций. *Storyboard* также описывает присоединенные свойства, используемые для связывания анимации с определенным объектом и свойством-зависимостью.

Класс *Timeline* (Временная шкала) определяет свойство *Duration*, для которого задано значение 0,5 секунды:

```
anima.Duration = new Duration(TimeSpan.FromSeconds(0.5));
```

При этом для раскадровки также можно задать меньшую продолжительность:

```
storyboard.Duration = new Duration(TimeSpan.FromSeconds(0.25));
```

Тогда анимация будет обрезана на 0,25 секунде. По умолчанию за продолжительность раскадровки берется самая большая продолжительность дочерних *Timeline* (в данном случае это 0,5 секунды), и в большинстве случаев это значение не переопределяется.

Timeline также описывает свойство *BeginTime* (Момент начала), которое можно задать либо для *Storyboard*, либо для *DoubleAnimation*:

```
anima.BeginTime = TimeSpan.FromSeconds(1);
```

Теперь запуск анимации будет отложен на 1 секунду.

Свойство *AutoReverse* (Автоматический возврат) типа Boolean со значением по умолчанию *false*. Зададим ему значение *true*:

```
anima.AutoReverse = true;
```

Теперь кнопка поворачивается на 360° по часовой стрелке и затем на 360° в обратном направлении. Общая продолжительность анимации составляет 1 секунду.

Свойство *RepeatBehavior* (Поведение повтора) указывает на то, сколько раз должна повториться анимация:

```
anima.RepeatBehavior = new RepeatBehavior(3);
```

Теперь кнопка делает три оборота общей продолжительностью 1,5 секунды. *RepeatBehavior* можно комбинировать с *AutoReverse*:

```
anima.RepeatBehavior = new RepeatBehavior(3);
anima.AutoReverse = true;
```

Теперь кнопка делает один поворот по часовой стрелке, затем против, затем опять по часовой стрелке и назад, и еще один раз по часовой стрелке и в обратном направлении. Общая продолжительность анимации – 3 секунды.

А если мы хотим, чтобы кнопка сделала три оборота по часовой стрелке и затем три оборота против? Нет ничего проще. Просто зададим *RepeatBehavior* для анимации:

```
anima.RepeatBehavior = new RepeatBehavior(3);
```

И *AutoReverse* для *Storyboard*:

```
storyboard.AutoReverse = true;
```

Значением *RepeatBehavior* может быть не только количество повторов, но единицы времени:

```
anima.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(0.75));
```

Теперь анимация кнопки будет повторяться в течение заданного для *RepeatBehavior* времени. В данном случае это составит 1½ разворота, и кнопка остановится в перевернутом положении (чтобы вернуть ее в нормальное состояние, *AutoReverse* должно быть задано *true*).

Свойству *RepeatBehavior* также может быть задано статическое значение *RepeatBehavior.Forever* (Постоянно), но, вероятно, для данного конкретного примера применять его нежелательно!

Для следующих нескольких экспериментов удалим все внесенные до сих пор изменения, приводя приложение в исходное состояние, только зададим продолжительность анимации равной 5, чтобы более ясно видеть, что происходит:

```
anima.Duration = new Duration(TimeSpan.FromSeconds(5));
```

Можно последовательно нажать все три кнопки, и анимации будут выполняться независимо. Такое поведение вполне ожидаемо, поскольку анимации являются независимыми объектами. Но что произойдет, если щелкнуть кнопку, для которой уже выполняется анимация? Как выясняется, она начнет движение заново. По сути, в этом случае мы применяем новую анимацию, которая замещает предыдущую, и новая анимация всегда начинается с угла 0° и выполняется до угла 360°. Рассмотрим, как описаны эти свойства:

```
anima.From = 0;
anima.To = 360;
```

Свойства-зависимости, такие как свойство *Angle* объекта *RotateTransform*, имеют *базовое значение*. Это значение свойства, когда анимация неактивна. Класс *DependencyObject* определяет метод *GetAnimationBaseValue* (Получить базовое значение анимации), с помощью которого можно получить это значение. *GetAnimationBaseValue* может быть вызван для любого производного от *DependencyObject*. В качестве аргумента в него передается *DependencyProperty*, например, *RotateTransform.AngleProperty*.

Если вызвать *GetAnimationBaseValue* для данного свойства *Angle*, будет возвращено значение нуль. Закомментируем свойство *From*, оставляя только *To*:

```
// anima.From = 0;
anima.To = 360;
```

И все работает! Выполняется анимация свойства *Angle* от ее базового значения нуль до заданного 360. Но если многократно щелкнуть *Button* во время вращения, происходит нечто странное. Возвращения кнопки в исходное положение к 0° не происходит, потому что

свойство *From* не задано, но скорость вращения кнопки с каждым нажатием уменьшается. Объясняется это тем, что каждая новая анимация начинается с текущего положения кнопки, поэтому до конечных 360° ей остается преодолеть меньшее расстояние, но отведенное на это время не меняется.

На самом ли деле такой сценарий работоспособен? Нажмем на кнопку еще раз после того, как анимация будет завершена. Ничего не происходит! После выполнения анимации свойство *Angle* остается с заданным значением 360, поэтому последующим анимациям просто нечего делать!

Теперь попробуем следующее:

```
anima.From = -360;
anima.To = null;
```

Выражение для свойства *To* кажется странным, поскольку из всего увиденного выше можно предположить, что *From* и *To* типа *double*. Но на самом деле эти свойства являются допускающими пустое значение *double*, и их значение по умолчанию – *null*. Задание *null* равноценно незаданию свойства вообще. Поведение в этом случае во многом похоже на поведение при исходных значениях: при каждом щечке кнопка перескакивает к значению –360°, и затем анимация возвращает ее к базовому значению, каковым является 0.

Посмотрим на это еще раз:

```
// anima.From = 0;
anima.To = 360;
```

По завершении анимации свойство *Angle* сохраняет значение 360. Такое поведение обусловлено свойством *FillBehavior* (Поведение завершения), определенным *Timeline*. Значением этого свойства по умолчанию является член перечисления *FillBehavior.HoldEnd* (Сохранять конечное значение), при котором свойство будет сохранять конечное значение по завершении анимации. Попробуем такой альтернативный вариант:

```
// anima.From = 0;
anima.To = 360;
anima.FillBehavior = FillBehavior.Stop;
```

Эти настройки приводят к удалению из свойства последствий анимации. По завершении анимации свойству *Angle* будет возвращено исходное значение 0. Визуально изменения значения увидеть нельзя, потому что 0° аналогично 360°, но значение изменяется. Если задать свойству *To* значение 180, то возвращение свойства к исходному значению будет более наглядным.

Альтернативой свойствам *To* и *From* является *By* (На):

```
// anima.From = 0;
// anima.To = 360;
anima.By = 90;
anima.FillBehavior = FillBehavior.HoldEnd;
```

В данном случае *FillBehavior* задано значение по умолчанию *HoldEnd*. По каждому щелчку кнопка разворачивается на 90°. Но если щелкнуть кнопку во время ее движения, отсчет угла поворота для новой анимации начнется с текущего положения кнопки, т.е. в итоге кнопка будет развернута на какой-то непонятный угол. Значение *By* позволяет прогрессивно увеличивать свойство на определенную величину при каждом последующем применении анимации.

Анимации, описанные в XAML

Бытует мнение, что раскадровки и анимации проще описывать в XAML, чем в коде, поэтому в большинстве случаев анимации Silverlight описываются в XAML. Но тогда возникают некоторые трудности в связи с совместным использованием ресурсов.

Перепишем приложение ClickAndSpin, определяя раскадровки и анимации в XAML. Рассмотрим область содержимого приложения XamlClickAndSpin:

Проект Silverlight: XamlClickAndSpin Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Button Name="btn1"
            Content="Button No. 1"
            Grid.Row="0"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform x:Name="rotate1" />
        </Button.RenderTransform>
    </Button>

    <Button Name="btn2"
            Content="Button No. 2"
            Grid.Row="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform x:Name="rotate2" />
        </Button.RenderTransform>
    </Button>

    <Button Name="btn3"
            Content="Button No. 3"
            Grid.Row="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <RotateTransform x:Name="rotate3" />
        </Button.RenderTransform>
    </Button>
</Grid>
```

Данная разметка практически аналогична предыдущей версии за исключением того, что всем элементам *Button* и всем объектам *RotateTransform* были присвоены имена для упрощения ссылки на них.

Раскадровки и анимации определены в коллекции *Resources* страницы:

Проект Silverlight: XamlClickAndSpin Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="storyboard1">
        <DoubleAnimation Storyboard.TargetName="rotate1"
                         Storyboard.TargetProperty="Angle"
                         From="0" To="360" Duration="0:0:0.5" />
    </Storyboard>

    <Storyboard x:Name="storyboard2">
        <DoubleAnimation Storyboard.TargetName="rotate2"
                         Storyboard.TargetProperty="Angle"
                         From="0" To="360" Duration="0:0:0.5" />
    </Storyboard>

    <Storyboard x:Name="storyboard3">
        <DoubleAnimation Storyboard.TargetName="rotate3"
                         Storyboard.TargetProperty="Angle"
                         From="0" To="360" Duration="0:0:0.5" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Три кнопки – три раскадровки. Обратите внимание на присоединенные свойства:

```
<DoubleAnimation Storyboard.TargetName="rotate1"
                 Storyboard.TargetProperty="Angle"
                 From="0" To="360" Duration="0:0:0.5" />
```

Для задания присоединенных свойств в коде вызываются статические методы *Storyboard.SetTarget* и *Storyboard.SetTargetProperty*. В XAML мы задаем присоединенные свойства *Storyboard.TargetName* (Имя цели) и *Storyboard.TargetProperty*. Заметьте разницу: в разметке ссылка на целевой объект происходит по имени, тогда как доступ в коде выполняется непосредственно к самому объекту.

Как альтернативный вариант ссылка на свойство *Angle* может быть сделана через объект *Button*:

```
<DoubleAnimation Storyboard.TargetName="btn1"

Storyboard.TargetProperty="(Button.RenderTransform).(RotateTransform.Angle)"
                 From="0" To="360" Duration="0:0:0.5" />
```

Или:

```
<DoubleAnimation Storyboard.TargetName="btn1"
                 Storyboard.TargetProperty="(Button.RenderTransform).Angle"
                 From="0" To="360" Duration="0:0:0.5" />
```

При таком синтаксисе объектам *RotateTransform* имена не требуются.

Обратите внимание на задание продолжительности. Требуется по крайней мере три числа, определяющих часы, минуты и секунды, разделенные двоеточиями. У секунд может быть дробная часть. Перед значением часов может быть указано количество дней; значения дней и часов разделяются точкой.

Чтобы было проще ссылаться на ресурсы *Storyboard* из кода, я использовал *x:Name*, а не *x:Key*. Обработчик события *Click* кнопки просто вызывает *Begin* соответствующего объекта:

Проект Silverlight: XamlClickAndSpin   Файл: MainPage.xaml.cs (фрагмент)

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
    if (sender == btn1)
        storyboard1.Begin();

    else if (sender == btn2)
```

```
        storyboard2.Begin();

    else if (sender == btn3)
        storyboard3.Begin();
}
```

Все достаточно просто. Но, вероятно, я не одинок в желании иметь возможность описывать в XAML всего одну раскадровку и анимацию, а не три. Кажется возможным опустить присвоение *Storyboard.TargetName* в XAML и вызывать метод *Storyboard.SetTarget* в коде, как только становится известным, какая кнопка нажата. Но мы не можем обеспечить совместного использования ресурсов, и если конкретные *Storyboard* и *DoubleAnimation* ассоциированы с одним *Button*, они не могут использоваться с другим *Button*. Имея один ресурс *Storyboard* и *DoubleAnimation*, мы не можем заставить вращаться две кнопки одновременно.

Даже если предположить, что одна кнопка будет останавливаться до начала вращения второй, мы должны гарантированно обеспечить остановку выполнения раскадровки, т.е. обязательно вызвать метод *Stop* объекта *Storyboard*. (Кроме методов *Begin* и *Stop* класс *Storyboard* описывает также методы *Pause* (Приостановить) и *Resume* (Возобновить), но они используются нечасто.)

Поучительная история

В предыдущих главах я показал, как с помощью метода *CompositionTarget.Rendering* перемещать и изменять визуальные объекты синхронно с частотой обновления экрана. Эта методика Silverlight хорошо подходит для некоторых сценариев, но пользоваться ею надо с осторожностью. Если вы действительно хотите применять *CompositionTarget.Rendering* в полноценных игровых циклах, например, пора подумать о XNA.

Большая проблема в том, что иногда *CompositionTarget.Rendering* работает не так хорошо, как ожидается. Например, вспомним приложение Spiral из главы 13 и рассмотрим приложение RotatedSpiral (Вращающаяся спираль), в котором делается попытка использовать *CompositionTarget.Rendering* для вращения спирали.

Проект Silverlight: RotatedSpiral Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    RotateTransform rotateTransform = new RotateTransform();

    public MainPage()
    {
        InitializeComponent();
        Loaded += OnLoaded;
    }

    void OnLoaded(object sender, RoutedEventArgs args)
    {
        Point center = new Point(ContentPanel.ActualWidth / 2 - 1,
                                 ContentPanel.ActualHeight / 2 - 1);
        double radius = Math.Min(center.X, center.Y);

        Polyline polyline = new Polyline();
        polyline.Stroke = this.Resources["PhoneForegroundBrush"] as Brush;
        polyline.StrokeThickness = 3;

        for (double angle = 0; angle < 3600; angle += 0.25)
        {
            double scaledRadius = radius * angle / 3600;
```

```
            double radians = Math.PI * angle / 180;
            double x = center.X + scaledRadius * Math.Cos(radians);
            double y = center.Y + scaledRadius * Math.Sin(radians);
            polyline.Points.Add(new Point(x, y));
        }
        ContentPanel.Children.Add(polyline);

        rotateTransform.CenterX = center.X;
        rotateTransform.CenterY = center.Y;
        polyline.RenderTransform = rotateTransform;

        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, EventArgs args)
    {
        TimeSpan elapsedTime = (args as RenderingEventArgs).RenderingTime;
        rotateTransform.Angle = 360 * elapsedTime.TotalSeconds / 3 % 360;
    }
}
```

Код здесь практически аналогичен приложению Spiral, но обратите внимание на поле *RotateTransform*. В конце обработчика события *Loaded* этот *RotateTransform* задается как значение свойства *RenderTransform* объекта *Polyline*, определяющего спираль, и прикрепляется обработчик события *CompositionTarget.Rendering*. Этот обработчик события меняет свойство *Angle* объекта *RotateTransform*, обеспечивая один поворот спирали каждые 3 секунды.

В этом коде нет никаких ошибок, но его производительность просто *ужасная*. Экран обновляется лишь один или два раза в секунду, и получаемая анимация очень прерывистая.

Мы рассмотрим три пути решения этой проблемы и повышения производительности. К счастью все они достаточно универсальны и могут применяться не только в данном конкретном приложении.

Решение 1:**Упрощение графических элементов**. Спираль реализована как полилиния, включающая 14400 точек. Это *много* больше, чем достаточно. Если в цикле *for* задать приращение 5, а не 0,25, спираль останется такой же гладкой, но и анимация станет более сглаженной. Вывод: чем меньше визуальных объектов, тем лучше производительность. Упростите свои графические элементы и деревья визуальных элементов.

Решение 2: **Кэширование визуальных элементов**. Silverlight пытается вращать *Polyline*, состоящий из множества отдельных точек. Эта задача была бы намного проще, если бы спираль была не сложным *Polyline*, а обычным растровым изображением. Можно создать *WriteableBitmap* этого графического объекта и вращать его. Или можно позволить Silverlight самостоятельно выполнить эквивалентную оптимизацию, просто задав для *Polyline* следующее свойство:

```
polyline.CacheMode = new BitmapCache();
```

Таким образом, Silverlight получает команду создать растровое изображение заданного элемента и использовать это растровое изображение для формирования визуального представления. Не следует применять данную методику к динамически изменяющимся векторным графическим элементам. Однако сложные графические элементы, в целом статические, но к которым может применяться анимация, являются замечательными кандидатурами для растрового кэширования. В XAML это выглядит следующим образом:

```
CacheMode="BitmapCache"
```

Решение 3: **Использование анимаций Silverlight вместо *CompositionTarget.Rendering***.

Перепишем приложение RotatedSpiral, сохранив число точек в *Polyline* и не прибегая к растровому кэшированию, но заменив *CompositionTarget.Rendering* объектом *DoubleAnimation*:

Проект Silverlight: AnimatedSpiral Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        Loaded += OnLoaded;
    }

    void OnLoaded(object sender, RoutedEventArgs args)
    {
        Point center = new Point(ContentPanel.ActualWidth / 2 - 1,
                                 ContentPanel.ActualHeight / 2 - 1);
        double radius = Math.Min(center.X, center.Y);

        Polyline polyline = new Polyline();
        polyline.Stroke = this.Resources["PhoneForegroundBrush"] as Brush;
        polyline.StrokeThickness = 3;

        for (double angle = 0; angle < 3600; angle += 0.25)
        {
            double scaledRadius = radius * angle / 3600;
            double radians = Math.PI * angle / 180;
            double x = center.X + scaledRadius * Math.Cos(radians);
            double y = center.Y + scaledRadius * Math.Sin(radians);
            polyline.Points.Add(new Point(x, y));
        }
        ContentPanel.Children.Add(polyline);

        RotateTransform rotateTransform = new RotateTransform();
        rotateTransform.CenterX = center.X;
        rotateTransform.CenterY = center.Y;
        polyline.RenderTransform = rotateTransform;

        DoubleAnimation anima = new DoubleAnimation
        {
            From = 0,
            To = 360,
            Duration = new Duration(TimeSpan.FromSeconds(3)),
            RepeatBehavior = RepeatBehavior.Forever
        };

        Storyboard.SetTarget(anima, rotateTransform);
        Storyboard.SetTargetProperty(anima,
                                     new
PropertyPath(RotateTransform.AngleProperty));

        Storyboard storyboard = new Storyboard();
        storyboard.Children.Add(anima);
        storyboard.Begin();
    }
}
```

И получаем намного более сглаженную анимацию, чем в предыдущем случае.

Чем обусловлена такая большая разница? Ведь анимации Silverlight на некотором уровне используют некоторый эквивалент *CompositionTarget.Rendering*, правда?

По правде говоря, это не так. Это во многом справедливо для настольной версии Silverlight, но Silverlight for Windows Phone доработан, чтобы обеспечить более активное применение графического процессора (graphics processing unit, GPU). GPU обычно ассоциируют с аппаратными ускорениями обработки сложных текстур и другими алгоритмами 3D графики, но в Silverlight GPU применяется для простых 2D анимаций.

Большинство приложений на Silverlight выполняются в одном потоке, который называют *потоком пользовательского интерфейса* или *UI-потоком*. UI-поток обрабатывает сенсорный ввод, компоновку и событие *CompositionTarget.Rendering*. Также используются некоторые рабочие потоки для таких задач, как растеризация, декодирование мультимедиа, датчики и асинхронный веб-доступ.

Silverlight for Windows Phone поддерживает *поток компоновщика* или *обрабатывающий поток*, в котором используется GPU.Обрабатывающий поток используется для нескольких типов анимаций свойств типа *double*, в частности:

- Трансформаций, задаваемых для свойства *RenderTransform*.
- Трансформаций перспективы, задаваемых для свойства *Projection*.
- Присоединенных свойств *Canvas.Left* и *Canvas.Top*.
- Свойства *Opacity*.
- Всех видов прямоугольных вырезаний.

Анимации, целевыми свойствами которых являются свойства типа *Color* или *Point*, по-прежнему выполняются в UI-потоке. Непрямоугольное вырезание или применение *OpacityMask* также осуществляются в UI-потоке и могут приводить к снижению производительности.

В качестве небольшой демонстрации рассмотрим проект UIThreadVsRenderThread (Сравнение UI-потока с обрабатывающим потоком), в котором вращение текста реализовано двумя разными способами:

Проект Silverlight: UIThreadVsRenderThread Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0"
               Text="UI Thread"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate1" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Grid.Row="1"
               Text="Render Thread"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               RenderTransformOrigin="0.5 0.5">
```

```
        <TextBlock.RenderTransform>
            <RotateTransform x:Name="rotate2" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <Button Grid.Row="2"
            Content="Hang for 5 seconds"
            HorizontalAlignment="Center"
            Click="OnButtonClick" />
</Grid>
```

Вращение первого *TextBlock* описывается в коде с использованием событий *CompositionTarget.Rendering*. Для второго *TextBlock* создается анимация с помощью следующего *Storyboard*, описанного в коллекции *Resources* страницы:

Проект Silverlight: UIThreadVsRenderThread Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="storyboard">
        <DoubleAnimation Storyboard.TargetName="rotate2"
                         Storyboard.TargetProperty="Angle"
                         From="0" To="360" Duration="0:1:0"
                         RepeatBehavior="Forever" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Конструктор *MainPage* запускает анимацию и подключает обработчик события *CompositionTarget.Rendering*.

Проект Silverlight: UIThreadVsRenderThread Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    DateTime startTime;

    public MainPage()
    {
        InitializeComponent();

        storyboard.Begin();
        startTime = DateTime.Now;
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, EventArgs args)
    {
        TimeSpan elapsedTime = DateTime.Now - startTime;
        rotate1.Angle = (elapsedTime.TotalMinutes * 360) % 360;
    }

    void OnButtonClick(object sender, RoutedEventArgs args)
    {
        Thread.Sleep(5000);
    }
}
```

Здесь для *CompositionTarget.Rendering* применяется логика синхронизации по времени, что обеспечивает одинаковую скорость перемещения обеих анимаций. Но нажмите эту кнопку, и произойдет нечто удивительное: *TextBlock*, вращение которого обеспечивается событиями *CompositionTarget.Rendering*, полностью замирает на пять секунд, тогда как *TextBlock*,

приводимый в движение *DoubleAnimation*, продолжает перемещаться! Это продолжает выполнение обрабатывающий поток даже несмотря на то, что UI-блокирован.

В случае применения к тексту вращения и масштабирования полезно знать другую настройку. Это присоединенное свойство, которое задается в XAML следующим образом:

```
TextOptions.TextHintingMode="Animated"
```

Альтернативой ему является свойство *Fixed* (Фиксированный), которое используется по умолчанию. Никакой оптимизации для повышения четкости и разборчивости текста, обозначенного как Animated, не производится.

Silverlight имеет три встроенные функции для визуализации проблем с производительностью. Все они могут использоваться на эмуляторе телефона, но приложения на эмуляторе обычно выполняются быстрее, чем на телефоне, поэтому мы все равно не сможем получить реальной картины производительности.

Более подробно вопросы производительности рассматриваются в документе «Creating High Performance Silverlight Applications for Windows Phone[1]», который доступен онлайн. Класс *Settings* пространства имен *System.Windows.Interop* включает три свойства типа Boolean, которые помогут визуализировать производительность. Доступ к этим трем свойствам осуществляется посредством объекта *SilverlightHost* (Хост Silverlight), который доступен как свойство *Host* (Хост) текущего объекта *Application*. Этим свойствам придается настолько важное значение, что они уже заданы – и все, кроме одного, закомментированы – в конструкторе класса *App* в стандартном файле App.xaml.cs.

Вот первое из них:

```
Application.Current.Host.Settings.EnableFrameRateCounter = true;
```

Данный флаг активирует небольшое окно сбоку экрана телефона, в котором отображается несколько элементов:

- Частота кадров (в кадрах в секунду) обрабатывающего потока (GPU).
- Частота кадров (в кадрах в секунду) UI-потока (CPU).
- Объем используемой видео-памяти в килобайтах.
- Число текстур, хранящихся в GPU.
- Число промежуточных объектов, созданных для сложных графических элементов.
- Доля закрашенных пикселов экрана в каждом кадре.

Самыми важными являются первые два показателя. При выполнении приложения на телефоне они должны составлять около 30 кадров в секунду. Вероятно, лучше всего использовать эти числа (и другие) для сравнения: запомните, чему они равны для относительно простых приложений, и затем отслеживайте изменения по мере того, как приложения становятся более сложными.

Второй используемый для диагностики флаг:

```
Application.Current.Host.Settings.EnableRedrawRegions = true;
```

Этот флаг достаточно любопытен. Каждый раз, когда UI-потоку требуется растеризировать графические элементы определенной области экрана, эта область выделяется другим цветом. (Иногда эти области называют «грязными», потому что они требуют обновления новыми

[1] Создание высокопроизводительных приложений на Silverlight for Windows Phone (*прим. переводчика*).

визуальными элементами.) Если мерцание наблюдается на больших областях экрана, это значит, что UI-поток не справляется с его обновлением. В идеале мы должны видеть лишь редкие вспышки цвета, которые возникают, когда анимации выполняются обрабатывающим потоком, а не UI-потоком.

И наконец, третий флаг:

```
Application.Current.Host.Settings.EnableCacheVisualization = true;
```

Данный флаг использует наложение цвета для выделения областей экрана, кэшированных в растровые изображения. Этот флаг можно применить в приложении RotatedSpiral (версией, использующей *CompositionTarget.Rendering*). При запуске этого приложения в том виде, в котором оно приведено выше, подцвечен весь экран. Это означает, что весь экран нуждается в растеризации при каждом изменении местоположения *Polyline*. На это уходит некоторое время, и поэтому производительность так низка. Теперь зададим следующее:

```
polyline.CacheMode = new BitmapCache();
```

На этот раз мы видим, что спираль находится в подцвеченном прямоугольнике, и вращается именно этот прямоугольник. Это и есть кэшированное растровое изображение.

Анимация по ключевым кадрам

Идея предоставления пользователю визуальной обратной связи при нажатии кнопки хороша, но вращать для этого кнопку на 360° – это уж слишком. Небольшого подрагивания будет вполне достаточно. Итак, открываем новый проект JiggleButtonTryout (Эксперимент с подрагивающей кнопкой) и начинаем экспериментировать.

Начнем с одного *Button* и зададим в качестве значения его свойства *RenderTransform* объект *TranslateTransform*:

Проект Silverlight: JiggleButtonTryout Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Content="Jiggle Button"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick">
        <Button.RenderTransform>
            <TranslateTransform x:Name="translate" />
        </Button.RenderTransform>
    </Button>
</Grid>
```

В коллекции *Resources* потребуется определить объект *Storyboard*:

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="jiggleStoryboard">

    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Файл выделенного кода обеспечивает запуск анимации по нажатию кнопки:

Проект Silverlight: JiggleButtonTryout Файл: MainPage.xaml.cs (фрагмент)

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
```

```
    jiggleStoryboard.Begin();
}
```

Наверное, начнем наши эксперименты с анимации свойства *X* объекта *TranslateTransform* с помощью *DoubleAnimation*:

```
<Storyboard x:Name="jiggleStoryboard">
    <DoubleAnimation Storyboard.TargetName="translate"
                     Storyboard.TargetProperty="X"
                     From="-10" To="10" Duration="0:0:0.05"
                     AutoReverse="True"
                     RepeatBehavior="3x" />
</Storyboard>
```

Получаемый результат можно назвать приемлемым, но это не вполне то, что хотелось бы: кнопка изначально отскакивает на 10 пикселов влево, затем возвращается назад, и так повторяется три раза. (Троекратный повтор анимации описан синтаксисом XAML "3x".) После этого кнопка замирает, оставаясь смещенной на 10 пикселов влево относительно ее исходного положения. Проблема станет более очевидной, если задать смещения от -100 до 100 и продолжительность анимации увеличить до ½ секунды.

Одним из способов решения этой проблемы является задание *FillBehavior* значения *Stop*, что обеспечит возвращение свойства к исходному значению по завершении анимации, обусловливая возвращение кнопки в исходное положение. Но это создает еще один дискретный «прыжок» в конце анимации вдобавок к прыжку в начале.

Чтобы все было реализовано правильно, нам понадобится две разных анимации. Сначала реализуем перемещение от 0 до -10, потом от -10 до 10 и обратно несколько раз, и, наконец, назад к 0. К счастью, в Silverlight есть возможность задания анимаций последовательно в строке. Это называют анимацией по ключевым кадрам. Первым делом для ее использования заменим *DoubleAnimation* на *DoubleAnimationUsingKeyFrames*. В этом новом подходе используются только *TargetName* и *TargetProperty*:

```
<Storyboard x:Name="jiggleStoryboard">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                   Storyboard.TargetProperty="X">

    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

При анимации свойств типа double класс *DoubleAnimationUsingKeyFrames* является единственной альтернативой *DoubleAnimation*, но при этом обеспечивает намного большую гибкость. У класса *DoubleAnimationUsingKeyFrames* есть дочерний член типа *DoubleKeyFrame* (Ключевой кадр типа double), предоставляющий четыре варианта выбора:

- *DiscreteDoubleKeyFrame* обеспечивает скачкообразный переход в определенное положение.
- *LinearDoubleKeyFame* выполняет линейную анимацию.
- *SplineDoubleKeyFrame* может ускоряться и замедляться.
- *EasingDoubleKeyFrame* (Ключевой кадр типа Double со сглаживанием) выполняет анимации с применением функций сглаживания.

Сейчас предлагаю применить *DiscreteDoubleKeyFrame* и *LinearDoubleKeyFrame*. Для каждого объекта keyframe должно быть задано два свойства: *KeyTime* (Опорный момент) и *Value* (Значение). *KeyTime* – это время, прошедшее с начала анимации; *Value* – значение, которое должно иметь свойство в этот момент.

Чтобы отчетливее увидеть, что происходит, будем раскачивать кнопку очень широко и будем делать это медленно. В начальный момент времени мы хотим, чтобы значение свойства было равно нулю:

```
<DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
```

Задание начального нулевого значения не является обязательным, потому что целевое свойство в начале анимации и так равно 0, но это не повредит.

Зададим, что в конце первой секунды значение должно быть равным –100:

```
<LinearDoubleKeyFrame KeyTime="0:0:01" Value="-100" />
```

Применение здесь *LinearDoubleKeyFrame* означает, что в течение промежутка времени от нуля до 1 секунды свойство *X* объекта *TranslateTransform* будет изменяться линейно от 0 до –100. Скорость анимации составляет 100 единиц в секунду, поэтому для перемещения объекта в положение 100 с той же скоростью потребуется три секунды:

```
<LinearDoubleKeyFrame KeyTime="0:0:03" Value="100" />
```

Это означает, что за период истекшего времени от 1 секунды до 3 секунд значение изменится от –100 до 100. Наконец, еще через секунду кнопка возвращается в исходное положение:

```
<LinearDoubleKeyFrame KeyTime="0:0:04" Value="0" />
```

Запустим эту анимацию и увидим, что кнопка перемещается влево, затем вправо, затем назад в центр без всяких скачков, и все движение длится 4 секунды. Общая продолжительность анимации по ключевым кадрам соответствует максимальному значению *KeyTime* среди всех объектов ключевых кадров.

Теперь повторим весь этот маневр три раза:

```
<Storyboard x:Name="jiggleStoryboard">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                   Storyboard.TargetProperty="X"
                                   RepeatBehavior="3x">
        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
        <LinearDoubleKeyFrame KeyTime="0:0:01" Value="-100" />
        <LinearDoubleKeyFrame KeyTime="0:0:03" Value="100" />
        <LinearDoubleKeyFrame KeyTime="0:0:04" Value="0" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Все шоу продолжается 12 секунд, и при этом проходит гладко, без каких либо прерываний.

Теперь когда мы обеспечили желаемое поведение кнопки, можно уменьшить смещения со 100 до 10:

```
<Storyboard x:Name="jiggleStoryboard">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                   Storyboard.TargetProperty="X"
                                   RepeatBehavior="3x">
        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
        <LinearDoubleKeyFrame KeyTime="0:0:01" Value="-10" />
        <LinearDoubleKeyFrame KeyTime="0:0:03" Value="10" />
        <LinearDoubleKeyFrame KeyTime="0:0:04" Value="0" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Чтобы привести значения времени к разумным величинам, я продемонстрирую небольшой трюк. Часто при создании анимаций мы сначала выполняем их очень медленно, чтобы убедиться в правильности отработки, и затем в окончательной версии задаем большую скорость. Конечно, можно пройтись по коду и изменить все значения *KeyTime*, или можно

просто задать для анимации *SpeedRatio* (Коэффициент скорости), как это делается в данной версии анимации в проекте JiggleButtonTryout:

Проект Silverlight: JiggleButtonTryout Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="jiggleStoryboard">
        <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                       Storyboard.TargetProperty="X"
                                       RepeatBehavior="3x"
                                       SpeedRatio="40">
            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <LinearDoubleKeyFrame KeyTime="0:0:01" Value="-10" />
            <LinearDoubleKeyFrame KeyTime="0:0:03" Value="10" />
            <LinearDoubleKeyFrame KeyTime="0:0:04" Value="0" />
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Каждый цикл переходов по ключевым кадрам занимает 4 секунды и повторяется 3 раза, что в общей сложности составляет 12 секунд. *SpeedRatio*, равный 40, эффективно ускоряет эту анимацию в 40 раз, т.е. ее продолжительность теперь составляет всего 0,3 секунды.

Этот эффект можно увековечить в пользовательском элементе управления *JiggleButton* (Подрагивающая кнопка), обеспечив возможность его многократного использования. Для этого есть несколько вариантов, но ни один из них не отвечает всем требованиям полностью.

Можно было бы наследоваться от *UserControl* и включить в этот элемент управления *Button* и трансформацию. Но чтобы сделать это правильно, пришлось бы воспроизвести все свойства и события *Button* как свойства и события *UserControl*. Другой подход предполагает применение шаблона. Вероятно, самым простым вариантом будет создать класс, производный от *Button*. Однако тогда придется настроить свойство *RenderTransform* соответственно этому конкретному случаю, и оно не сможет использоваться для других целей.

Триггер по событию Loaded

Windows Presentation Foundation обеспечивает большую гибкость в описании и использовании анимаций, чем Silverlight. WPF включает объекты, называемые *триггерами*, которые отвечают на события или на изменения свойств, и которые могут реализовывать запуск анимаций полностью в XAML, устраняя необходимость запуска *Storyboard* в файле выделенного кода. В Silverlight триггеры практически не используются, их почти полностью заменил Visual State Manager (Диспетчер визуальных состояний), который мы обсудим в следующей главе.

Но все-таки один триггер в Silverlight остался. Это триггер, отвечающий на событие *Loaded*. Он позволяет полностью описывать анимацию в XAML, обеспечивая ее запуск сразу после загрузки страницы (или другого элемента).

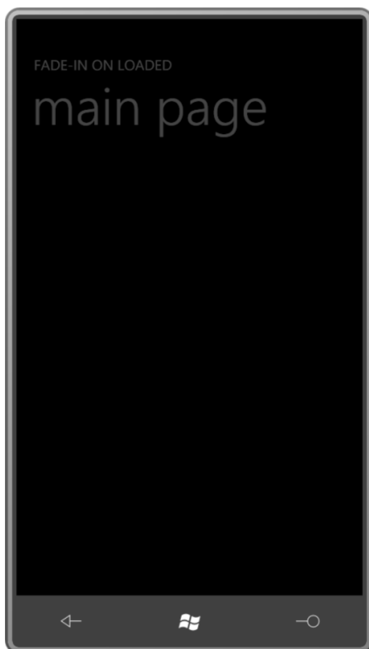
Проект FadeInOnLoaded (Проявление при загрузке) включает следующий XAML, который располагается ближе к концу кода страницы, непосредственно над закрывающим тегом *PhoneApplicationPage*. Это традиционное место размещения триггеров, срабатывающих по событию:

Проект Silverlight: FadeInOnLoaded Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="TitlePanel"
                                 Storyboard.TargetProperty="Opacity"
                                 From="0" To="1" Duration="0:0:10" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</phone:PhoneApplicationPage.Triggers>
```

Разметка начинается с тега свойства-элемента для свойства *Triggers* (Триггеры), описанного *FrameworkElement*. Свойство *Triggers* типа *TriggerCollection* (Коллекция триггеров), что кажется довольно всеобъемлющим и универсальным, но в Silverlight в эту коллекцию можно поместить лишь тег *EventTrigger* (Триггер, срабатывающий по событию), который всегда ассоциирован с событием *Loaded*. Далее следует тег *BeginStoryboard* (Начать раскадровку). Это единственное место в Silverlight, где можно увидеть тег *BeginStoryboard*. И наконец, что-то знакомое: *Storyboard* с одной или более анимациями, целью которых может быть любой объект-зависимость любого объекта страницы.

Целью данной анимации является свойство *Opacity* объекта *TitlePanel*. *TitlePanel* – это *TitlePanel*, включающий два заголовка вверху страницы. Я задал для анимации продолжительность 10 секунд, чтобы мы ничего не упустили. Как только страница загружается, плавно проявляется ее заголовок:



Документация Silverlight не рекомендует выполнять запуск анимаций таким образом. Безусловно, эта методика нашла свое узкое применение в реальных приложениях, и остается очень популярной в демонстрационных приложениях на базе XAML с анимациями, выполняющимися «вечно». Заставить анимацию выполняться вечно (или непрерывно в рамках вашего терпения) можно следующим выражением:

```
RepeatBehavior="Forever"
```

В документации сказано, что разметку со свойством-элементом *Triggers* можно размещать только в корневом элементе страницы, но на самом деле она может находиться и несколько ближе к объектам, к которым применяется анимация. Я продемонстрирую это в нескольких

следующих приложениях данной главы, в которых дерево визуальных элементов области содержимого и анимация являются одной счастливой семьей.

Все визуальные элементы следующего приложения располагаются в центрированном квадратном *Grid* с фиксированной стороной 400 пикселов. *Grid* включает пять концентрических окружностей. Все они являются элементами *Path*, в качестве значений свойств *Data* которых заданы объекты *EllipseGeometry*. Значения свойств *RadiusX* и *RadiusY* каждого *EllipseGeometry* на 25 пикселов превышают длину следующего наименьшего объекта.

Проект Silverlight: ExpandingCircles Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid Width="400" Height="400"
          HorizontalAlignment="Center"
          VerticalAlignment="Center" >

        <!-- Внутренняя окружность. -->
        <Path Name="pathInner"
              Stroke="{StaticResource PhoneAccentBrush}"
              StrokeThickness="12.5">
            <Path.Data>
                <EllipseGeometry x:Name="ellipse1"
                                 Center="200 200"
                                 RadiusX="0" RadiusY="0" />
            </Path.Data>
        </Path>

        <!-- Все окружности, кроме внутренней и внешней. -->
        <Path Stroke="{StaticResource PhoneAccentBrush}"
              StrokeThickness="12.5">
            <Path.Data>
                <GeometryGroup>
                    <EllipseGeometry x:Name="ellipse2"
                                     Center="200 200"
                                     RadiusX="25" RadiusY="25" />
                    <EllipseGeometry x:Name="ellipse3"
                                     Center="200 200"
                                     RadiusX="50" RadiusY="50" />
                    <EllipseGeometry x:Name="ellipse4"
                                     Center="200 200"
                                     RadiusX="75" RadiusY="75" />
                </GeometryGroup>
            </Path.Data>
        </Path>

        <!-- Внешняя окружность. -->
        <Path Name="pathOuter"
              Stroke="{StaticResource PhoneAccentBrush}"
              StrokeThickness="12.5">
            <Path.Data>
                <EllipseGeometry x:Name="ellipse5"
                                 Center="200 200"
                                 RadiusX="100" RadiusY="100" />
            </Path.Data>
        </Path>

        <Grid.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard RepeatBehavior="Forever">
                        <DoubleAnimation Storyboard.TargetName="pathInner"
                                         Storyboard.TargetProperty="StrokeThickness"
                                         From="0" Duration="0:0:5" />
```

```
                        <DoubleAnimation Storyboard.TargetName="ellipse1"
                                         Storyboard.TargetProperty="RadiusX"
                                         From="0" To="25" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse1"
                                         Storyboard.TargetProperty="RadiusY"
                                         From="0" To="25" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse2"
                                         Storyboard.TargetProperty="RadiusX"
                                         From="25" To="50" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse2"
                                         Storyboard.TargetProperty="RadiusY"
                                         From="25" To="50" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse3"
                                         Storyboard.TargetProperty="RadiusX"
                                         From="50" To="75" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse3"
                                         Storyboard.TargetProperty="RadiusY"
                                         From="50" To="75" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse4"
                                         Storyboard.TargetProperty="RadiusX"
                                         From="75" To="100" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse4"
                                         Storyboard.TargetProperty="RadiusY"
                                         From="75" To="100" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse5"
                                         Storyboard.TargetProperty="RadiusX"
                                         From="100" To="125" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="ellipse5"
                                         Storyboard.TargetProperty="RadiusY"
                                         From="100" To="125" Duration="0:0:5" />

                        <DoubleAnimation Storyboard.TargetName="pathOuter"
                                         Storyboard.TargetProperty="Opacity"
                                         From="1" To="0" Duration="0:0:4.9" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Grid.Triggers>
    </Grid>
</Grid>
```

Этот центрированный *Path* также является элементом, к которому присоединен *EventTrigger*. *Storyboard* содержит 12 объектов *DoubleAnimation*, которые выполняются все одновременно в течение 5 секунд. (На самом деле, последний длится 4,9 секунды, но я внес эту поправку, чтобы избежать случайных визуальных помех.) И весь *Storyboard* повторяется непрерывно. Целевыми свойствами всех объектов *DoubleAnimation* (кроме двух) являются свойства *RadiusX* и *RadiusY* пяти объектов *EllipseGeometry*. *DoubleAnimation* обеспечивают увеличение целевых свойств на 25 пикселов, т.е. делают их равными базовому значению следующего по размеру круга.

В то же время анимация свойства *Opacity* внешней окружности обеспечивает ее постепенное исчезновение, а анимация свойства *StrokeThickness* внутренней окружности создает впечатление, что она возникает ниоткуда. Тем самым получаем общий визуальный эффект расхождения концентрических кругов из центра и их исчезновения по достижении максимального размера:

Следующее приложение называется DashOffsetAnimation (Анимация смещения обводки пунктиром) и использует *Path* для отрисовки знака бесконечности при альбомном расположении экрана. знак бесконечности включает два полукруга (справа и слева), каждый из которых отрисовывается с помощью сплайнов Безье на основании всем известного приближения.

Кривая Безье обеспечивает замечательную аппроксимацию четверти окружности. Для окружности с центром в точке (0, 0) и радиусом 100 дуга, образующая нижнюю правую четверть, начинается в точке (100, 0) и переходит в направлении по часовой стрелке в точку (0, 100). Воспроизвести эту дугу можно с помощью кривой Безье, которая начинается в точке (100, 0), заканчивается в точке (0, 100) и имеет две опорные точки: (100, 55) и (55, 100). Продолжая по этой схеме – я называю ее правилом «Безье 55» – можно отрисовать полную окружность, состоящую из четырех соединенных кривых Безье. Эта аппроксимация настолько хороша, что в некоторых графических системах окружности реализованы именно с применением этой техники.

Для описания *Data*, которое показано ниже, используется Path Markup Syntax, начинающийся с M («move»). Далее следует C («cubic Bézier») с двумя опорными точками и конечной точкой. Но затем выполняется переход к S («smooth Bézier» [1]), для которой требуется задать лишь вторую опорную точку и конечную точку. Для определения первой опорной точки S, которая лежит на одной прямой с начальной точкой и предыдущей опорной точкой, автоматически используется предыдущая кривая Безье.

Массив *StrokeDashArray* задан двумя значениями: 0 и 1,5. Они указывают на то, что длина штриха равна 0, а расстояние между штрихами – 1,5. Но свойству *StrokeDashCap* задано значение *Round*, поэтому штрихи будут вырождены в круглые точки, отстоящие друг от друга на половину толщины линии.

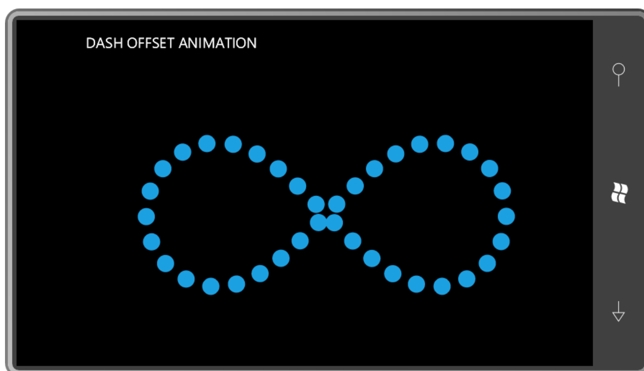
Проект Silverlight: DashOffsetAnimation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path Name="path"
          HorizontalAlignment="Center"
```

[1] сглаженная кривая Безье (*прим. переводчика*).

```
          VerticalAlignment="Center"
          Stroke="{StaticResource PhoneAccentBrush}"
          StrokeThickness="23.98"
          StrokeDashArray="0 1.5"
          StrokeDashCap="Round"
          Data="M 100    0
                C  45    0,    0   45, 0 100
                S  45  200,  100  200
                S 200  150,  250  100
                S 345    0,  400    0
                S 500   45,  500  100
                S 455  200,  400  200
                S 300  150,  250  100
                S 155    0,  100    0">

        <Path.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="path"

Storyboard.TargetProperty="StrokeDashOffset"
                                         From="0" To="1.5" Duration="0:0:1"
                                         RepeatBehavior="Forever" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Path.Triggers>
    </Path>
</Grid>
```

Целевым свойством этого *DoubleAnimation* является свойство *StrokeDashOffset* объекта *Path*, которое обычно равно нулю. Это свойство определяет местоположение точек, штрихов и пробелов соответственно началу линии. В результате анимации получаем эффект непрерывного перемещения точек по замкнутому контуру.



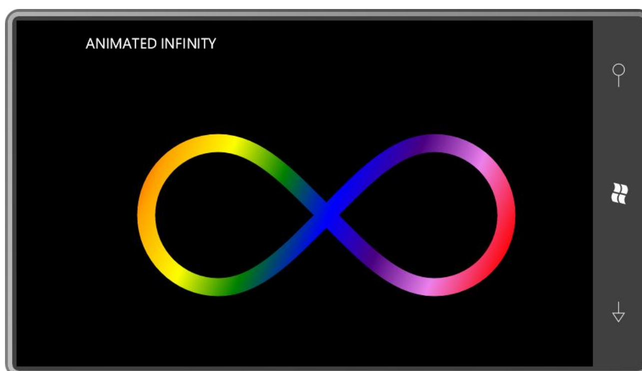
Критически важную роль в обеспечении сглаженности анимации *StrokeDashOffset* для замкнутого контура играет значение *StrokeThickness*. Общая длина линии должна быть кратной целое число раз произведению *StrokeDashArray* и *StrokeThickness*. Если значение *StrokeThickness* далеко от правильного, мы будем видеть эффект «пузырения», создаваемый в результате частичной отрисовки точек. Если значение очень близко, но все равно не соответствует необходимому, могут возникать мерцания.

AnimatedInfinity (Анимированная бесконечность) – еще одно приложение с альбомной ориентацией экрана. В нем используется все тот же Path Markup Syntax, но знак бесконечности закрашивается в традиционные цвета радуги с помощью *LinearGradientBrush*:

Проект Silverlight: AnimatedInfinity Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path HorizontalAlignment="Center"
          VerticalAlignment="Center"
          StrokeThickness="25"
          Data="M 100    0
                C  45    0,    0   45, 0 100
                S  45  200,  100  200
                S 200  150,  250  100
                S 345    0,  400    0
                S 500   45,  500  100
                S 455  200,  400  200
                S 300  150,  250  100
                S 155    0,  100    0">

        <Path.Stroke>
            <LinearGradientBrush SpreadMethod="Repeat">
                <LinearGradientBrush.Transform>
                    <TranslateTransform x:Name="translate" />
                </LinearGradientBrush.Transform>
                <LinearGradientBrush.GradientStops>
                    <GradientStop Offset="0.00" Color="Red" />
                    <GradientStop Offset="0.14" Color="Orange" />
                    <GradientStop Offset="0.28" Color="Yellow" />
                    <GradientStop Offset="0.42" Color="Green" />
                    <GradientStop Offset="0.56" Color="Blue" />
                    <GradientStop Offset="0.70" Color="Indigo" />
                    <GradientStop Offset="0.85" Color="Violet" />
                    <GradientStop Offset="1.00" Color="Red" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Path.Stroke>
        <Path.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="translate"
                                         Storyboard.TargetProperty="X"
                                         From="0" To="625" Duration="0:0:2"
                                         RepeatBehavior="Forever" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Path.Triggers>
    </Path>
</Grid>
```

Класс *Brush* определяет свойство *Transform*, но оно редко используется. В данном приложении в качестве значения этого свойства задан *TranslateTransform*, последующая анимация которого обеспечивает плавное перетекание цветов по фигуре:



Также можно применять анимации к свойствам типа *Color*, т.е. можно анимировать цвета кисти. Рассмотрим приложение, в котором выполняется анимация свойств *Color* двух объектов *GradientStop* в *LinearGradientBrush*:

Проект Silverlight: GradientAnimation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid HorizontalAlignment="Center"
          VerticalAlignment="Center">
        <TextBlock Text="GRADIENT"
                   FontSize="96"
                   FontWeight="Bold">
            <TextBlock.Foreground>
                <LinearGradientBrush>
                    <GradientStop x:Name="gradientStop1"
                                  Offset="0" Color="Red" />
                    <GradientStop x:Name="gradientStop2"
                                  Offset="1" Color="Blue" />
                </LinearGradientBrush>
            </TextBlock.Foreground>
        </TextBlock>

        <Grid.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard>
                        <ColorAnimation Storyboard.TargetName="gradientStop1"
                                        Storyboard.TargetProperty="Color"
                                        From="Red" To="Blue" Duration="0:0:11"
                                        AutoReverse="True"
                                        RepeatBehavior="Forever" />

                        <ColorAnimation Storyboard.TargetName="gradientStop2"
                                        Storyboard.TargetProperty="Color"
                                        From="Blue" To="Red" Duration="0:0:13"
                                        AutoReverse="True"
                                        RepeatBehavior="Forever" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Grid.Triggers>
    </Grid>
</Grid>
```

Для этих двух анимаций продолжительность задана простыми числами 11 и 13, так что весь цикл, включая *AutoReverse*, будет длиться почти 5 минут и затем повторяться снова.

Анимация присоединенных свойств (или нет)

Существует несколько разных способов применения анимаций Silverlight для перемещения элемента по экрану. Один из них – использовать в качестве цели анимации объект *TranslateTransform*, заданный как значение свойства *RenderTransform* элемента. Но вероятно, разработчики, которым более привычно работать с *Canvas*, захотят применить анимацию к присоединенным свойствам *Canvas.Left* и *Canvas.Top*. Для анимации присоединенных свойств используется специальный синтаксис, но он довольно прост.

В данном приложении используется квадратный *Canvas* со стороной 450 пикселов, который центрируется относительно области содержимого. Затем создается объект *Ellipse* с высотой и шириной по 50 пикселов. После этого организовывается перемещение *Ellipse* по периметру *Canvas*. Один цикл перемещения длится 4 секунды, циклы повторяются непрерывно.

Проект Silverlight: MoveOnCanvas Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Canvas Width="450" Height="450"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">

        <Ellipse Name="ball"
                 Fill="{StaticResource PhoneAccentBrush}"
                 Width="50" Height="50" />

        <Canvas.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard RepeatBehavior="Forever">
                        <DoubleAnimationUsingKeyFrames
                                        Storyboard.TargetName="ball"
                                        Storyboard.TargetProperty="(Canvas.Left)">
                            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
                            <LinearDoubleKeyFrame   KeyTime="0:0:1" Value="400" />
                            <DiscreteDoubleKeyFrame KeyTime="0:0:2" Value="400" />
                            <LinearDoubleKeyFrame   KeyTime="0:0:3" Value="0" />
                            <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="0" />
                        </DoubleAnimationUsingKeyFrames>

                        <DoubleAnimationUsingKeyFrames
                                        Storyboard.TargetName="ball"
                                        Storyboard.TargetProperty="(Canvas.Top)">
                            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
                            <DiscreteDoubleKeyFrame KeyTime="0:0:1" Value="0" />
                            <LinearDoubleKeyFrame   KeyTime="0:0:2" Value="400" />
                            <DiscreteDoubleKeyFrame KeyTime="0:0:3" Value="400" />
                            <LinearDoubleKeyFrame   KeyTime="0:0:4" Value="0" />
                        </DoubleAnimationUsingKeyFrames>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Canvas.Triggers>
    </Canvas>
</Grid>
```

Обратите внимание, что *Storyboard.TargetName* ссылается на элемент *Ellipse*, и в качестве значений атрибутов *Storyboard.TargetProperty* заданы строки "(Canvas.Left)" и "(Canvas.Top)". Когда целями анимации являются присоединенные свойства, в скобках следует указывать полные имена свойств. Все просто.

Не так просто – и та же проблема возникает при использовании *TranslateTransform* в качестве цели анимации– перемещать объект в нескольких направлениях. Для этого приходится отдельно обрабатывать координаты *X* и *Y*, а это часто вызывает затруднения. Я использовал подход с применением ключевых кадров. В обоих случаях первым указывается *DiscreteDoubleKeyFrame*, присваивающий свойству значение нуль, и затем чередуются объекты *DiscreteDoubleKeyFrame* и *LinearDoubleKeyFrame*, обеспечивающие перемещение *Ellipse* по периметру *Canvas*.

Обычно намного проще обрабатывать одновременно обе координаты, X и Y, с помощью *PointAnimation* или *PointAnimationUsingKeyFrames*. Конечно, в Silverlight очень немного классов, определяющих свойства-зависимости типа *Point*, но те которые делают это – в частности, производные *Geometry* – являются основными классами векторной графики.

Перепишем это приложение с использованием *DoubleAnimationUsingKeyFrames*, целью которого является свойство *Center* объекта *EllipseGeometry*:

Проект Silverlight: MoveInGrid Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid Width="450" Height="450"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">

        <Path Fill="{StaticResource PhoneAccentBrush}">
            <Path.Data>
                <EllipseGeometry x:Name="ballGeometry"
                                 RadiusX="25" RadiusY="25" />
            </Path.Data>
        </Path>

        <Grid.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard RepeatBehavior="Forever">
                        <PointAnimationUsingKeyFrames
                                Storyboard.TargetName="ballGeometry"
                                Storyboard.TargetProperty="Center">
                            <DiscretePointKeyFrame KeyTime="0:0:0" Value=" 25  25"
/>
                            <LinearPointKeyFrame   KeyTime="0:0:1" Value="425  25"
/>
                            <LinearPointKeyFrame   KeyTime="0:0:2" Value="425 425"
/>
                            <LinearPointKeyFrame   KeyTime="0:0:3" Value=" 25 425"
/>
                            <LinearPointKeyFrame   KeyTime="0:0:4" Value=" 25  25"
/>
                        </PointAnimationUsingKeyFrames>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Grid.Triggers>
    </Grid>
</Grid>
```

Координаты необходимо немного настроить, потому что теперь позиционирование шара выполняет по его центру, а не по верхнему левому углу. Но последовательность анимаций более четкая, и используется одна анимация, а не две.

Вот теперь оборотная сторона: анимации с целевыми свойствами типа *Point* обрабатываются *не* в обрабатывающем потоке GPU. Если это имеет значение, используйте анимации свойств типа *double*.

Если для вас развлечение имеет большее значение, чем производительность, можно создать *PathGeometry*, явно применяя объекты *PathFigure*, *LineSegment*, *ArcSegment*, *BezierSegment* и *QuadraticBezierSegment*. Тогда каждое свойство типа *Point* может быть целью анимации.

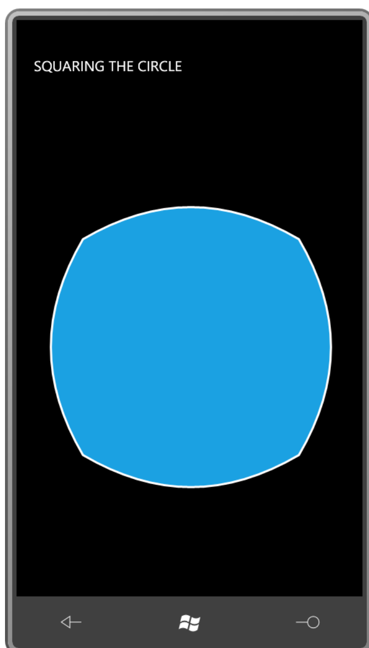
Рассмотрим приложение, которое доводит эту идею до крайности. В нем с помощью четырех сплайнов Безье создается окружность и затем выполняется анимация различных свойств *Point*, что обеспечивает превращение круга в квадрат и решает геометрическую проблему, которая не давала покоя математикам еще со времен Эвклида:

Проект Silverlight: SquaringTheCircle Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Fill="{StaticResource PhoneAccentBrush}"
          Stroke="{StaticResource PhoneForegroundBrush}"
          StrokeThickness="3" >
        <Path.Data>
            <PathGeometry>
                <PathFigure x:Name="bezier1" IsClosed="True">
                    <BezierSegment x:Name="bezier2" />
                    <BezierSegment x:Name="bezier3" />
                    <BezierSegment x:Name="bezier4" />
                    <BezierSegment x:Name="bezier5" />
                </PathFigure>
                <PathGeometry.Transform>
                    <TransformGroup>
                        <ScaleTransform ScaleX="2" ScaleY="2" />
                        <RotateTransform Angle="45" />
                        <TranslateTransform X="200" Y="200" />
                    </TransformGroup>
                </PathGeometry.Transform>
            </PathGeometry>
        </Path.Data>

        <Path.Triggers>
            <EventTrigger>
                <BeginStoryboard>
                    <Storyboard RepeatBehavior="Forever"
                                AutoReverse="True" >
                        <PointAnimation Storyboard.TargetName="bezier1"
                                        Storyboard.TargetProperty="StartPoint"
                                        From="0 100" To="0 125" />

                        <PointAnimation Storyboard.TargetName="bezier2"
                                        Storyboard.TargetProperty="Point1"
                                        From="55 100" To="62.5 62.5" />

                        <PointAnimation Storyboard.TargetName="bezier2"
                                        Storyboard.TargetProperty="Point2"
                                        From="100 55" To="62.5 62.5" />

                        <PointAnimation Storyboard.TargetName="bezier2"
                                        Storyboard.TargetProperty="Point3"
                                        From="100 0" To="125 0" />

                        <PointAnimation Storyboard.TargetName="bezier3"
                                        Storyboard.TargetProperty="Point1"
                                        From="100 -55" To="62.5 -62.5" />

                        <PointAnimation Storyboard.TargetName="bezier3"
                                        Storyboard.TargetProperty="Point2"
                                        From="55 -100" To="62.5 -62.5" />

                        <PointAnimation Storyboard.TargetName="bezier3"
                                        Storyboard.TargetProperty="Point3"
                                        From="0 -100" To="0 -125" />

                        <PointAnimation Storyboard.TargetName="bezier4"
                                        Storyboard.TargetProperty="Point1"
                                        From="-55 -100" To="-62.5 -62.5" />

                        <PointAnimation Storyboard.TargetName="bezier4"
                                        Storyboard.TargetProperty="Point2"
                                        From="-100 -55" To="-62.5 -62.5" />

                        <PointAnimation Storyboard.TargetName="bezier4"
                                        Storyboard.TargetProperty="Point3"
                                        From="-100 0" To="-125 0" />
```

```
                        <PointAnimation Storyboard.TargetName="bezier5"
                                        Storyboard.TargetProperty="Point1"
                                        From="-100 55" To="-62.5 62.5" />

                        <PointAnimation Storyboard.TargetName="bezier5"
                                        Storyboard.TargetProperty="Point2"
                                        From="-55 100" To="-62.5 62.5" />

                        <PointAnimation Storyboard.TargetName="bezier5"
                                        Storyboard.TargetProperty="Point3"
                                        From="0 100" To="0 125" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Path.Triggers>
    </Path>
</Grid>
```

Вот что мы имеем на полпути от квадрата к кругу:



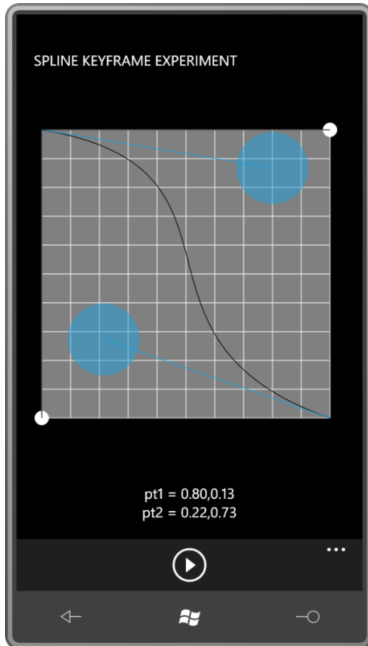
Сплайны и ключевые кадры

Три класса ключевых кадров начинаются со слова *Spline*: *SplineDoubleKeyFrame*, *SplinePointKeyFrame* (Сплайновый ключевой кадр типа Point) и *SplineColorKeyFrame* (Сплайновый ключевой кадр типа Color). Эти классы имеют свойства *KeyTime* и *Value*, как и дискретные и линейные ключевые кадры, но также они определяют свойство *KeySpline* (Ключевой сплайн). Это свойство позволяет создавать ключевой кадр, ускоряющийся или замедляющийся (или и то, и другое) в ходе выполнения, но при этом завершающийся заданным значением в заданное *KeyTime* время. Изменением скорости управляет сплайн Безье.

KeySpline – это структура с двумя свойствами типа *Point*: *ControlPoint1* (Опорная точка 1) и *ControlPoint2*. Координаты *X* и *Y* каждой из этих точек должны лежать в диапазоне от 0 до 1. Один объект *KeySpline* эффективно описывает кривую Безье, которая начинается в точке (0, 0) и заканчивается в точке (1, 1) с заданными двумя контрольными точками. Эти четыре точки полностью определяют кривую Безье, и создать произвольную кривую, петлю например, невозможно. Но мы увидим, что при этом обеспечивается достаточная гибкость.

Концептуально в ходе выполнения ключевого кадра координата X этого сплайна представляет нормализованное время, изменяющееся линейно в диапазоне от 0 до 1. Координата Y – это нормализованное значение анимации, также изменяющееся в диапазоне от 0 до 1, но нелинейным образом.

Безусловно, почувствовать сплайновые ключевые кадры можно, только поэкспериментировав с ними. Специально для этого у меня припасена программка, которая даже называется SplineKeyFrameExperiment (Эксперимент со сплайновым ключевым кадром):



Перемещать опорные точки можно с помощью голубых полупрозрачных кругов. На *ApplicationBar* имеется только одна кнопка с подписью «animate» (анимация):

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.transport.play.rest.png"
                                        Text="animate"
                                        Click="OnAppbarAnimateButtonClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

При нажатии кнопки белый шарик внизу сетки начинает перемещаться линейно слева направо, представляя линейное увеличение времени. Белый шарик, расположенный справа сетки, перемещается нелинейно сверху вниз соответственно форме сплайна.

Для простоты компоновка приложения строится на базе сетки с фиксированной шириной и высотой в 400 пикселов, поэтому для меньшего экрана это приложение придется немного подкорректировать.

Область содержимого представляет собой серый квадрат со стороной 400 пикселов с горизонтальными и вертикальными линиями, равномерно проведенными через каждые 40 пикселов. Каждая линия сетки представляет 0,1 единицы для отображения сплайна.

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid Name="graphGrid"
          Grid.Row="0"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">

        <!-- Фон -->
        <Path Fill="#808080"
              Data="M 0 0 L 400 0, 400 400, 0 400 Z" />

        <!-- Горизонтальные линии -->
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 0 400 0" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 40 400 40" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 80 400 80" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 120 400 120" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 160 400 160" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 200 400 200" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 240 400 240" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 280 400 280" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 320 400 320" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 360 400 360" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 400 400 400" />

        <!-- Вертикальные линии -->
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="0 0 0 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="40 0 40 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="80 0 80 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="120 0 120 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="160 0 160 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="200 0 200 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="240 0 240 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="280 0 280 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="320 0 320 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="360 0 360 400" />
        <Polyline Stroke="{StaticResource PhoneForegroundBrush}"
                  Points="400 0 400 400" />
        …
    </Grid>

    <TextBlock Name="txtblk"
               Grid.Row="1"
               TextAlignment="Center"
```

```
                Margin="0, 24" />
</Grid>
```

TextBlock внизу экрана используется для отображения значений двух опорных точек.

Ниже приведена разметка для кривой Безье, которая всегда начинается в верхнем левом углу сетки, который представляет точку (0, 0), и заканчивается в нижнем правом углу сетки, который представляет точку (1, 1). Две опорные точки (*Point1* и *Point2* объекта *BezierSegment*) задаются пользователем.

Данный фрагмент XAML также включает две касательные линии, соединяющие конечные точки с опорными. Я бы предпочел использовать привязку данных для связывания различных свойств этих элементов друг с другом, но в Silverlight 3 целями привязки могут быть только свойства элементов, а в данном случае свойства являются производными от *PathSegment*.

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<!-- Bezier curve -->
<Path Stroke="{StaticResource PhoneBackgroundBrush}">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="0 0">
                <BezierSegment x:Name="bezierSegment"
                               Point1="200 80"
                               Point2="200 320"
                               Point3="400 400" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>

<!-- Tangent lines -->
<Path Stroke="{StaticResource PhoneAccentBrush}">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="0 0">
                <LineSegment x:Name="tangentLine1"
                             Point="200 80" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>

<Path Stroke="{StaticResource PhoneAccentBrush}">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="400 400">
                <LineSegment x:Name="tangentLine2"
                             Point="200 320" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
```

В данном фрагменте описываются два небольших белых шарика, которые отображаются внизу и справа сетки. Один из них представляет время, и другой – анимируемый объект:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<!-- Balls -->
<Path Fill="{StaticResource PhoneForegroundBrush}">
    <Path.Data>
        <EllipseGeometry x:Name="timeBall"
                         RadiusX="10"
                         RadiusY="10"
                         Center="0 400" />
    </Path.Data>
</Path>

<Path Fill="{StaticResource PhoneForegroundBrush}">
    <Path.Data>
        <EllipseGeometry x:Name="animaBall"
                         RadiusX="10"
                         RadiusY="10"
                         Center="400 0" />
    </Path.Data>
</Path>
```

Этого не видно на статической иллюстрации, но небольшие шарики соединены со сплайном двумя линиями, горизонтальной и вертикальной. Эти линии отслеживают кривую сплайна при перемещении шариков:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<!-- Tracking lines -->
<Line x:Name="timeTrackLine"
    Stroke="{StaticResource PhoneBackgroundBrush}"
    Y2="400" />

<Line x:Name="animaTrackLine"
    Stroke="{StaticResource PhoneBackgroundBrush}"
    X2="400" />
```

Наконец, два полупрозрачных круга распознают касание и используются для перемещения опорных точек по сетке:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<!-- Draggers -->
<Path Name="dragger1"
      Fill="{StaticResource PhoneAccentBrush}"
      Opacity="0.5">
    <Path.Data>
        <EllipseGeometry x:Name="dragger1Geometry"
                         RadiusX="50"
                         RadiusY="50"
                         Center="200 80" />
    </Path.Data>
</Path>

<Path Name="dragger2"
      Fill="{StaticResource PhoneAccentBrush}"
      Opacity="0.5">
    <Path.Data>
        <EllipseGeometry x:Name="dragger2Geometry"
                         RadiusX="50"
                         RadiusY="50"
                         Center="200 320" />
    </Path.Data>
</Path>
```

Центры этих двух объектов *EllipseGeometry* обеспечивают две опорные точки для объекта *KeySpline*. В файле выделенного кода конструктор инициализирует располагающийся внизу экрана *TextBlock* значениями, нормализованными в диапазоне от 0 до 1:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        UpdateTextBlock();
    }

    void UpdateTextBlock()
    {
        txtblk.Text = String.Format("pt1 = {0:F2}\npt2 = {1:F2}",
                                    NormalizePoint(dragger1Geometry.Center),
                                    NormalizePoint(dragger2Geometry.Center));
    }

    Point NormalizePoint(Point pt)
    {
        return new Point(pt.X / 400, pt.Y / 400);
    }
    …
}
```

В виду отсутствия привязок данных в XAML перегруженный метод *OnManipulationDelta* должен изменять два дополнительных свойства типа *Point* (кроме *TextBlock*) при каждом перемещении прозрачных кругов:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationDelta(ManipulationDeltaEventArgs args)
{
    Point translation = args.DeltaManipulation.Translation;

    if (args.ManipulationContainer == dragger1)
    {
        Point pt = new Point(Clamp(dragger1Geometry.Center.X + translation.X),
                             Clamp(dragger1Geometry.Center.Y + translation.Y));

        dragger1Geometry.Center = pt;
        bezierSegment.Point1 = pt;
        tangentLine1.Point = pt;
        UpdateTextBlock();
    }
    if (args.ManipulationContainer == dragger2)
    {
        Point pt = new Point(Clamp(dragger2Geometry.Center.X + translation.X),
                             Clamp(dragger2Geometry.Center.Y + translation.Y));

        dragger2Geometry.Center = pt;
        bezierSegment.Point2 = pt;
        tangentLine2.Point = pt;
        UpdateTextBlock();
    }

    base.OnManipulationDelta(args);
}

double Clamp(double input)
```

```
{
    return Math.Max(0, Math.Min(400, input));
}
```

При нажатии кнопки *ApplicationBar* приложение должно задать четырем разным анимациям одинаковые объекты *KeySpline* и затем запустить выполнение *Storyboard*:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarAnimateButtonClick(object sender, EventArgs args)
{
    Point controlPoint1 = NormalizePoint(dragger1Geometry.Center);
    Point controlPoint2 = NormalizePoint(dragger2Geometry.Center);

    splineKeyFrame1.KeySpline = new KeySpline();
    splineKeyFrame1.KeySpline.ControlPoint1 = controlPoint1;
    splineKeyFrame1.KeySpline.ControlPoint2 = controlPoint2;

    splineKeyFrame2.KeySpline = new KeySpline();
    splineKeyFrame2.KeySpline.ControlPoint1 = controlPoint1;
    splineKeyFrame2.KeySpline.ControlPoint2 = controlPoint2;

    splineKeyFrame3.KeySpline = new KeySpline();
    splineKeyFrame3.KeySpline.ControlPoint1 = controlPoint1;
    splineKeyFrame3.KeySpline.ControlPoint2 = controlPoint2;

    splineKeyFrame4.KeySpline = new KeySpline();
    splineKeyFrame4.KeySpline.ControlPoint1 = controlPoint1;
    splineKeyFrame4.KeySpline.ControlPoint2 = controlPoint2;

    storyboard.Begin();
}
```

Эта раскадровка определена в коллекции *Resources* страницы:

Проект Silverlight: SplineKeyFrameExperiment Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="storyboard"
                SpeedRatio="0.25">
        <PointAnimation Storyboard.TargetName="timeBall"
                        Storyboard.TargetProperty="Center"
                        From="0 400" To="400 400" Duration="0:0:1" />

        <DoubleAnimation Storyboard.TargetName="timeTrackLine"
                         Storyboard.TargetProperty="X1"
                         From="0" To="400" Duration="0:0:1" />

        <DoubleAnimation Storyboard.TargetName="timeTrackLine"
                         Storyboard.TargetProperty="X2"
                         From="0" To="400" Duration="0:0:1" />

        <DoubleAnimation Storyboard.TargetName="animaTrackLine"
                         Storyboard.TargetProperty="X1"
                         From="0" To="400" Duration="0:0:1" />

        <PointAnimationUsingKeyFrames Storyboard.TargetName="animaBall"
                                      Storyboard.TargetProperty="Center">
            <DiscretePointKeyFrame KeyTime="0:0:0" Value="400 0" />
            <SplinePointKeyFrame x:Name="splineKeyFrame1"
                                 KeyTime="0:0:1" Value="400 400" />
        </PointAnimationUsingKeyFrames>

        <DoubleAnimationUsingKeyFrames Storyboard.TargetName="timeTrackLine"
```

```
                               Storyboard.TargetProperty="Y1">
            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <SplineDoubleKeyFrame x:Name="splineKeyFrame2"
                                 KeyTime="0:0:1" Value="400" />
        </DoubleAnimationUsingKeyFrames>

        <DoubleAnimationUsingKeyFrames Storyboard.TargetName="animaTrackLine"
                                       Storyboard.TargetProperty="Y1">
            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <SplineDoubleKeyFrame x:Name="splineKeyFrame3"
                                 KeyTime="0:0:1" Value="400" />
        </DoubleAnimationUsingKeyFrames>

        <DoubleAnimationUsingKeyFrames Storyboard.TargetName="animaTrackLine"
                                       Storyboard.TargetProperty="Y2">
            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <SplineDoubleKeyFrame x:Name="splineKeyFrame4"
                                 KeyTime="0:0:1" Value="400" />
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Итак, попробуем. Если поместить обе опорные точки в точку (1, 0), получим анимацию, стартующую очень медленно и ускоряющуюся к ее окончанию. Задание обеим опорным точкам координат (0, 1) обеспечивает обратный эффект. Зададим для первой опорной точки координаты (1, 0) и для второй – (0, 1), и получим анимацию, медленную в начале, затем ускоряющуюся и затем опять замедляющуюся в конце. Меняем местами значения координат опорных точек – и получаем обратный эффект.

Конечно, не обязательно использовать предельные значения. Задавая промежуточные величины, можно обеспечить более интересные эффекты. Например, значения (0.25, 0) и (0.6, 0.2) позволят смоделировать свободное падение. Для моделирования перемещения объекта вверх с последующим падением вследствие земного притяжения, вычтите из 1 каждую из этих координат.

Конечно же, у меня есть пример.

Проблема прыгающего мяча

Рассмотрим XAML, созданный для телефона с большим экраном и обеспечивающий перемещение шара вверх и вниз:

```
<Grid x:Name="ContentPanel" Grid.Row="1">
    <Path Fill="Red">
        <Path.Data>
            <EllipseGeometry RadiusX="25" RadiusY="25" />
        </Path.Data>

        <Path.RenderTransform>
            <TranslateTransform x:Name="translate" X="240" />
        </Path.RenderTransform>
    </Path>

    <Path Fill="{StaticResource PhoneAccentBrush}"
          Data="M 100 625 L 380 625, 380 640, 100 640 Z" />

    <Grid.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard RepeatBehavior="Forever">
                    <DoubleAnimation Storyboard.TargetName="translate"
                                     Storyboard.TargetProperty="Y"
                                     From="50" To="600" Duration="0:0:1"
```

```
                                        AutoReverse="True"
                                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Grid.Triggers>
</Grid>
```

В нем используется два элемента *Path*. Первый описывает красный шар (будем называть его «мячом») второй – поверхность, по которой этот шар будет «прыгать».

К мячу применяется *TranslateTransform*: значение свойства *X* фиксировано, что обеспечивает горизонтальное центрирование мяча; к свойству *Y* применяется анимация в диапазоне значений от 50 до 600 и обратно. Но получаемое перемещение не выглядит как реальное подпрыгивание мяча, потому что выполняется постоянно с одинаковой скоростью, т.е. не подчиняется законам физики. В реальности физические законы действуют по умолчанию, но в компьютерной графике для их реализации часто приходится потрудиться.

Получить более реалистичную модель подпрыгивания мяча поможет объект *DoubleAnimationUsingKeyFrames* с *SplineDoubleKeyFrame*, который ускоряет мяч при падении и замедляет при его движении вверх. Для создания эффекта свободного падения используются опорные точки сплайна:

```
<Storyboard RepeatBehavior="Forever">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                   Storyboard.TargetProperty="Y">
        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="50" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1" Value="600"
                                KeySpline="0.25 0, 0.6 0.2" />
        <SplineDoubleKeyFrame   KeyTime="0:0:2" Value="50"
                                KeySpline="0.75 1, 0.4 0.8" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Вот теперь намного лучше, но по-прежнему не вполне правильно. Проблема в моделировании момента удара мяча о землю. Когда мяч ударяется о землю, он имеет максимальную скорость и сразу же с максимальной скоростью начинает движение в противоположном направлении.

В реальности все происходит не так. Когда мяч ударяется о землю, он замедляется и немного деформируется, сжимаясь. Обратная деформация приводит к тому, что мяч опять ускоряется. Можно ли это смоделировать? Почему нет?

Для реализации приостановки мяча в момент соударения с землей необходим еще один ключевой кадр. Я решил, что мяч будет сжиматься и разжиматься в течение одной десятой секунды. Вероятно, этого больше чем достаточно, поэтому я немного подкорректировал хронометраж:

```
<Storyboard RepeatBehavior="Forever">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="translate"
                                   Storyboard.TargetProperty="Y">
        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="50" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1" Value="600"
                                KeySpline="0.25 0, 0.6 0.2" />
        <DiscreteDoubleKeyFrame KeyTime="0:0:1.1" Value="600" />
        <SplineDoubleKeyFrame   KeyTime="0:0:2.1" Value="50"
                                KeySpline="0.75 1, 0.4 0.8" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Трансформация *TranslateTransform* начинается в нулевой момент времени со значения 50. В течение следующей секунды ее значение меняется до 600 (мяч ускоряется). Следующую

десятую секунды значение остается равным 600 и затем возвращается к 50 в течение следующей секунды. Теперь анимация длится не 2, а 2,1 секунды.

Конечно, само по себе это выглядит даже еще хуже. Но давайте добавим в *Path*, описывающий мяч, один *ScaleTransform*:

```
<Path.RenderTransform>
    <TransformGroup>
        <ScaleTransform x:Name="scale" CenterY="25" />
        <TranslateTransform x:Name="translate" X="240" />
    </TransformGroup>
</Path.RenderTransform>
```

Не подверженный трансформациям центр мяча находится в точке (0, 0), радиус мяча равен 25 пикселам, поэтому нижняя точка мяча имеет координаты (0, 25). Это точка, которая касается пола, и она должна оставаться неподвижной во время *ScaleTransform*, поэтому мы и задали *CenterY* равным 25. *CenterX* равен 0 по умолчанию.

Вот еще две анимации для моментального распрямления мяча после деформации:

```
<Storyboard RepeatBehavior="Forever">
    …
    <DoubleAnimationUsingKeyFrames
                    Storyboard.TargetName="scale"
                    Storyboard.TargetProperty="ScaleX">
        <DiscreteDoubleKeyFrame KeyTime="0:0:1"    Value="1" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1.05" Value="1.5"
                                KeySpline="0.75 1, 0.4 0.8" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1.1"  Value="1"
                                KeySpline="0.25 0, 0.6 0.2" />
    </DoubleAnimationUsingKeyFrames>

    <DoubleAnimationUsingKeyFrames
                    Storyboard.TargetName="scale"
                    Storyboard.TargetProperty="ScaleY">
        <DiscreteDoubleKeyFrame KeyTime="0:0:1"    Value="1" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1.05" Value="0.66"
                                KeySpline="0.75 1, 0.4 0.8" />
        <SplineDoubleKeyFrame   KeyTime="0:0:1.1"  Value="1"
                                KeySpline="0.25 0, 0.6 0.2" />
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

В промежутке времени между 1 секундой и 1,05 секунды ширина мяча увеличивается на 50% и высота уменьшается на треть. Ситуация меняется на обратную в течение следующих 0,05 секунд, после чего мяч приобретает нормальную форму и начинает движение вверх.

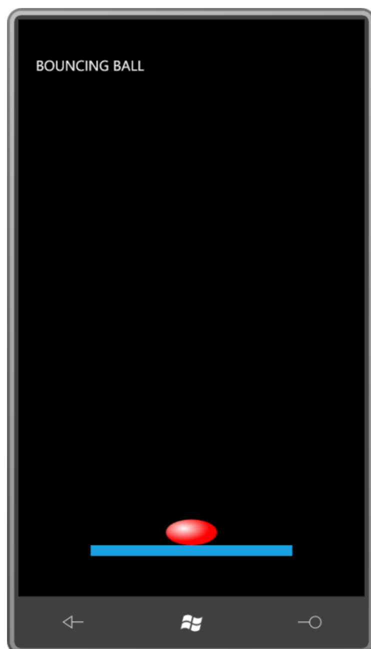
В окончательной версии приложения BouncingBall (Прыгающий мяч) к мячу также применяется *RadialGradientBrush*:

Проект Silverlight: BouncingBall Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1">
    <Path>
        <Path.Data>
            <EllipseGeometry RadiusX="25" RadiusY="25" />
        </Path.Data>

        <Path.Fill>
            <RadialGradientBrush GradientOrigin="0.35 0.35"
                                 Center="0.35 0.35">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="Red" />
            </RadialGradientBrush>
```

```
        </Path.Fill>

        <Path.RenderTransform>
            <TransformGroup>
                <ScaleTransform x:Name="scale" CenterY="25" />
                <TranslateTransform x:Name="translate" X="240" />
            </TransformGroup>
        </Path.RenderTransform>
    </Path>

    <Path Fill="{StaticResource PhoneAccentBrush}"
          Data="M 100 625 L 380 625, 380 640, 100 640 Z" />

    <Grid.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard RepeatBehavior="Forever">
                    <DoubleAnimationUsingKeyFrames
                                    Storyboard.TargetName="translate"
                                    Storyboard.TargetProperty="Y">
                        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="50" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:1" Value="600"
                                                KeySpline="0.25 0, 0.6 0.2" />
                        <DiscreteDoubleKeyFrame KeyTime="0:0:1.1" Value="600" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:2.1" Value="50"
                                                KeySpline="0.75 1, 0.4 0.8" />
                    </DoubleAnimationUsingKeyFrames>

                    <DoubleAnimationUsingKeyFrames
                                    Storyboard.TargetName="scale"
                                    Storyboard.TargetProperty="ScaleX">
                        <DiscreteDoubleKeyFrame KeyTime="0:0:1"    Value="1" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:1.05" Value="1.5"
                                                KeySpline="0.75 1, 0.4 0.8" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:1.1"  Value="1"
                                                KeySpline="0.25 0, 0.6 0.2" />
                    </DoubleAnimationUsingKeyFrames>

                    <DoubleAnimationUsingKeyFrames
                                    Storyboard.TargetName="scale"
                                    Storyboard.TargetProperty="ScaleY">
                        <DiscreteDoubleKeyFrame KeyTime="0:0:1"    Value="1" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:1.05" Value="0.66"
                                                KeySpline="0.75 1, 0.4 0.8" />
                        <SplineDoubleKeyFrame   KeyTime="0:0:1.1"  Value="1"
                                                KeySpline="0.25 0, 0.6 0.2" />
                    </DoubleAnimationUsingKeyFrames>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Grid.Triggers>
</Grid>
```

И вот что мы получаем:



Функции сглаживания

Описывать ключевые кадры с помощью сплайнов, в определенном смысле, просто, потому что участвуют лишь четыре числа, но и не так легко. Для создания необходимого эффекта необходимо использовать сплайн Безье, но не всегда очевидно, как именно реализовать этот эффект.

Намного удобнее было бы использовать готовые функции, которые обеспечивали бы общее впечатление выполнения физических законов и не требовали сложных умозаключений. Такими функциями являются *функции сглаживания*. Это классы, наследуемые от *EasingFunctionBase* (Базовый класс функций сглаживания) и включающие типовые переходы, которые можно добавлять в начало или конец (или и туда, и туда) анимаций. Во всех классах анимаций (*DoubleAnimation*, *PointAnimation* и *ColorAnimation*) есть свойство *EasingFunction* типа *EasingFunctionBase*. Также в нашем распоряжении имеются классы *EasingDoubleKeyFrame*, *EasingColorKeyFrame* и *EasingPointKeyFrame*.

Класс *EasingFunctionBase* определяет только одно свойство: *EasingMode* (Режим сглаживания). Его значениями могут быть члены перечисления *EasingMode*: *EaseOut* (Сглаживание в конце)(значение по умолчанию, которое обеспечивает применение перехода только в конце анимации), *EaseIn* (Сглаживание в начале) или *EaseInOut* (Сглаживание в начале и в конце). От *EasingFunctionBase* наследуются одиннадцать классов. От него можно унаследовать и собственный класс, если требуется обеспечить большую управляемость.

Проект TheEasingLife (Упрощение жизни) позволяет выбирать из одиннадцати производных от *EasingFunctionBase* и наблюдать их эффект на примере простой анимации *PointAnimation* шарообразного объекта. В области содержимого располагаются два элемента *Polyline* и один *Path*, но координаты не заданы, это делается в коде.

Проект Silverlight: TheEasingLife   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Polyline Name="polyline1"
              Stroke="{StaticResource PhoneForegroundBrush}" />
```

```
    <Polyline Name="polyline2"
              Stroke="{StaticResource PhoneForegroundBrush}" />

    <Path Fill="{StaticResource PhoneAccentBrush}">
        <Path.Data>
            <EllipseGeometry x:Name="ballGeometry"
                             RadiusX="25"
                             RadiusY="25" />
        </Path.Data>
    </Path>
</Grid>
```

Коллекция *Resources* включает *Storyboard* с анимацией *PointAnimation*, целевым свойством которой является свойство *Center* объекта *EllipseGeometry*. Для *PointAnimation* задано только свойство *Duration*:

Проект Silverlight: TheEasingLife Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="storyboard"
                Completed="OnStoryboardCompleted">
        <PointAnimation x:Name="pointAnimation"
                        Storyboard.TargetName="ballGeometry"
                        Storyboard.TargetProperty="Center"
                        Duration="0:0:2" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Обратите внимание, что задан обработчик события *Completed* объекта *Storyboard*. Это событие *Completed* описано классом *Timeline*, его удобно использовать как средство уведомления приложения о завершении анимации.

ApplicationBar включает две кнопки: «animate» и «settings»:

Проект Silverlight: TheEasingLife Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.transport.play.rest.png"
                                        Text="animate"
                                        Click="OnAppbarPlayButtonClick" />
        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.feature.settings.rest.png"
                                        Text="settings"
                                        Click="OnAppbarSettingsButtonClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Координаты двух элементов *Polyline* и *EllipseGeometry* задаются в обработчике события *Loaded* на основании размера панели содержимого. Предполагается, что шар будет перемещаться между верхним *Polyline* и нижним *Polyline*; фактические координаты хранятся в массиве *ballPoints* (Координаты шара). Направлением движения (вверх или вниз) управляет поле *isForward* (Вперед).

Проект Silverlight: TheEasingLife Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
```

```
    PointCollection ballPoints = new PointCollection();
    bool isForward = true;

    public MainPage()
    {
        InitializeComponent();
        Loaded += OnMainPageLoaded;
    }

    public EasingFunctionBase EasingFunction { get; set; }

    void OnMainPageLoaded(object sender, RoutedEventArgs args)
    {
        double left = 100;
        double right = ContentPanel.ActualWidth - 100;
        double center = ContentPanel.ActualWidth / 2;
        double top = 100;
        double bottom = ContentPanel.ActualHeight - 100;

        polyline1.Points.Add(new Point(left, top));
        polyline1.Points.Add(new Point(right, top));

        polyline2.Points.Add(new Point(left, bottom));
        polyline2.Points.Add(new Point(right, bottom));

        ballPoints.Add(new Point(center, top));
        ballPoints.Add(new Point(center, bottom));

        ballGeometry.Center = ballPoints[1 - Convert.ToInt32(isForward)];
    }
    …
}
```

Также обратите внимание на открытое свойство *EasingFunction*. При нажатии кнопки «animate» обработчик *Click* заполняет недостающие параметры *PointAnimation* (включая свойство *EasingFunction*) и запускает эту анимацию:

Проект Silverlight: TheEasingLife Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarPlayButtonClick(object sender, EventArgs args)
{
    pointAnimation.From = ballPoints[1 - Convert.ToInt32(isForward)];
    pointAnimation.To = ballPoints[Convert.ToInt32(isForward)];
    pointAnimation.EasingFunction = EasingFunction;

    storyboard.Begin();
}

void OnStoryboardCompleted(object sender, EventArgs args)
{
    isForward ^= true;
}
```

Обработчик события *Completed* меняет значение *isForward*, готовя объект к следующей анимации.

При нажатии кнопки «settings» приложение переходит к странице *EasingFunctionDialog* (Диалоговое окно выбора функции сглаживания), на которой можно выбрать необходимую функцию сглаживания:

Проект Silverlight: TheEasingLife Файл: MainPage.xaml.cs (фрагмент)

```
void OnAppbarSettingsButtonClick(object sender, EventArgs args)
{
    NavigationService.Navigate(new Uri("/EasingFunctionDialog.xaml",
UriKind.Relative));
}

protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    if (args.Content is EasingFunctionDialog)
    {
        (args.Content as EasingFunctionDialog).EasingFunction = EasingFunction;
    }
    base.OnNavigatedTo(args);
}
```

Когда перегруженный метод *OnNavigatedFrom* узнает о том, что выполняется переход от *MainPage* к странице *EasingFunctionDialog*, он передает содержимое свойства *EasingFunction* в *EasingFunctionDialog*, который также имеет открытое свойство *EasingFunction*.

Область содержимого файла EasingFunctionDialog.xaml file включает только *ScrollViewer* со *StackPanel* в нем:

Проект Silverlight: TheEasingLife Файл: EasingFunctionDialog.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ScrollViewer>
        <StackPanel Name="stack" />
    </ScrollViewer>
</Grid>
```

В своем перегруженном методе диалог заполняет *StackPanel* элементами *RadioButton*, применяя технологию отражения. На момент вызова *OnNavigatedTo* свойству *EasingFunction* уже задано действительное значение перегруженным методом *OnNavigatedFrom* класса *MainPage*:

Проект Silverlight: TheEasingLife Файл: EasingFunctionDialog.xaml.cs (фрагмент)

```
public partial class EasingFunctionDialog : PhoneApplicationPage
{
    public EasingFunctionDialog()
    {
        InitializeComponent();
    }

    public EasingFunctionBase EasingFunction { get; set; }

    …

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Создаем RadioButton с подписью "None"
        RadioButton radio = new RadioButton();
        radio.Content = "None";
        radio.IsChecked = (EasingFunction == null);
        radio.Checked += OnRadioButtonChecked;
        stack.Children.Add(radio);

        Assembly assembly = Assembly.Load("System.Windows");

        // Создаем RadioButton для каждой функции сглаживания
        foreach (Type type in assembly.GetTypes())
            if (type.IsPublic && type.IsSubclassOf(typeof(EasingFunctionBase)))
```

```
            {
                radio = new RadioButton();
                radio.Content = type.Name;
                radio.Tag = type;
                radio.IsChecked = (EasingFunction != null &&
                                    EasingFunction.GetType() == type);
                radio.Checked += OnRadioButtonChecked;
                stack.Children.Add(radio);
            }
            base.OnNavigatedTo(args);
    }
    …
}
```

Обратите внимание, что свойство *Tag* каждого *RadioButton* – это объект *Type* (Тип), обозначающий производный от *EasingFunctionBase* класс, ассоциированный с этой кнопкой. Когда пользователь нажимает один из элементов *RadioButton*, это свойство *Tag* используется для создания нового объекта соответствующего типа:

Проект Silverlight: TheEasingLife Файл: EasingFunctionDialog.xaml.cs (фрагмент)

```
void OnRadioButtonChecked(object sender, RoutedEventArgs args)
{
    Type type = (sender as RadioButton).Tag as Type;

    if (type == null)
    {
        EasingFunction = null;
    }
    else
    {
        ConstructorInfo constructor = type.GetConstructor(Type.EmptyTypes);
        EasingFunction = constructor.Invoke(null) as EasingFunctionBase;
    }
}
```

Наконец, когда необходимая функция сглаживания выбрана, пользователь нажимает кнопку Back, и вызывается перегруженный метод *OnNavigatedFrom* диалогового окна. Он сохраняет выбранную функцию в *MainPage*:

Проект Silverlight: TheEasingLife Файл: EasingFunctionDialog.xaml.cs (фрагмент)

```
protected override void OnNavigatedFrom(NavigationEventArgs args)
{
    if (args.Content is MainPage)
    {
        (args.Content as MainPage).EasingFunction = EasingFunction;
    }
    base.OnNavigatedFrom(args);
}
```

Не забывайте, что во всех производных от *EasingFunctionBase* все свойства имеют значения по умолчанию, в том числе и свойство *EasingMode*, которое ограничивает действие функции только к концу анимации. Можно заметить, что некоторые из этих эффектов – в частности, *BackEase* (Сглаживание с откатом) и *ElasticEase* (Эластичное сглаживание) – приводят к отклонению от заданного конечного значения. В преимущественном большинстве случаев это не важно, но иногда это может приводить к недопустимым значениям свойств. Мы не хотим задавать свойству *Opacity* значения, вне диапазона от 0 до 1, например.

Анимация трансформации перспективы

Все трансформации, используемые в *RenderTransform*, являются примерами двухмерных *афинных* преобразований. Афинные преобразования отличаются очень регулярным поведением и даже немного скучны: прямые линии всегда трансформируются в прямые, эллипсы – в эллипсы, и квадраты – в параллерограммы. Две линии, бывшие параллельными до трансформации, остаются параллельными и после нее.

Silverlight 3 вводит в *UIElement* новое свойство *Projection*, которое позволяет задавать *неафинные* преобразования для графических объектов, текста, элементов управления и мультимедиа. Неафинные преобразования не сохраняют параллелизма.

Используемый в Silverlight 3 тип афинного преобразования по-прежнему реализован посредством умножения матриц, и по-прежнему его возможности ограничены. Прямые линии всегда трансформируются в прямые, и квадрат всегда будет превращен в простой выпуклый четырехугольник. Под «четырехугольником» я понимаю фигуру с четырьмя сторонами (еще их называют тетрагонами); говоря «простой» я имею в виду, что стороны не пересекаются друг с другом, кроме как в вершинах; «выпуклый» означает, что внутренние углы в каждой вершине не превышают 180 градусов.

Этот тип неафинного преобразования очень пригодится для создания трансформаций *сужения*, при которых противоположные стороны квадрата или прямоугольника немного сужаются в одном направлении. Объект приобретает вид трехмерного, потому что его часть визуально располагается дальше. Этот эффект называют *перспективой*.

В некотором смысле свойство *Projection* обеспечивает Silverlight псевдо 3D-эффект. Это не настоящая трехмерная система, потому что мы не можем определять объекты в трехмерном пространстве, нет понятия камер, освещения или теней, и, что возможно самое важное, нет вырезания объектов на основании их размещения в трехмерном пространстве.

Но при этом использование трансформации *Projection* требует от разработчика мыслить в категориях трехмерного пространства, особенно это касается объемного вращения. К счастью создатели Silverlight обеспечили простоту использования свойства *Projection*.

В качестве значения свойства *Projection* может использоваться один из двух объектов: *Matrix3DProjection*, обеспечивающий возможность применения математических вычислений и гибкость, или *PlaneProjection* (Планарная проекция), обеспечивающий упрощенный подход. *PlaneProjection* определяет двенадцать задаваемых свойств, но шести из них будет вполне достаточно.

Самыми важными свойствами *PlaneProjection* являются *RotationX* (Вращение вокруг оси X), *RotationY* и *RotationZ*. Задавая этим свойствам значения угла, мы обеспечиваем вращение объекта вокруг оси X (которая располагается горизонтально слева направо),Y (которая располагается вертикально сверху вниз) и Z (которая располагается перпендикулярно к экрану в направлении к пользователю).

Направление вращения определяется по правилу правой руки: поместите свой большой палец по оси в положительном направлении (для оси X это будет вправо, для Y – вниз, для Z – к себе). Расположение остальных пальцев руки указывает на направление вращения при положительных углах вращения. Отрицательные углы обеспечивают вращение в противоположном направлении.

Сложное вращение зависит от порядка применения составляющих его вращений. Применяя *PlaneProjection*, мы отказываемся от некоторой гибкости этих вращений. *PlaneProjection* всегда применяет сначала *RotationX*, затем *RotationY* и, наконец, *RotationZ*, но в большинстве

случаев требуется задать только одно из этих свойств. Как и *RenderTransform*, *Projection* не влияет на компоновку. Система компоновки всегда «видит» исходный элемент, без учета применяемых к нему трансформаций и проекций.

RotationX, *RotationY* и *RotationZ* продублированы свойствами-зависимостями, поэтому могут использоваться как цели анимаций, что демонстрирует приложение PerspectiveRotation (Вращение перспективы). В области содержимого располагается *TextBlock*, в качестве значения свойства *Projection* которого задан объект *PlaneProjection*, и три кнопки:

Проект Silverlight: PerspectiveRotation Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <TextBlock Name="txtblk"
               Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3"
               Text="ROTATE"
               FontSize="{StaticResource PhoneFontSizeHuge}"
               Foreground="{StaticResource PhoneAccentBrush}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.Projection>
            <PlaneProjection x:Name="planeProjection" />
        </TextBlock.Projection>
    </TextBlock>

    <Button Grid.Row="1" Grid.Column="0"
            Content="Rotate X"
            Click="RotateXClick" />

    <Button Grid.Row="1" Grid.Column="1"
            Content="Rotate Y"
            Click="RotateYClick" />

    <Button Grid.Row="1" Grid.Column="2"
            Content="Rotate Z"
            Click="RotateZClick" />
</Grid>
```

В коллекции *Resources* описаны три раскадровки для анимации свойств *RotationX*, *RotationY* и *RotationZ* объекта *PlaneProjection*:

Проект Silverlight: PerspectiveRotation Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="rotateX">
        <DoubleAnimation Storyboard.TargetName="planeProjection"
                         Storyboard.TargetProperty="RotationX"
                         From="0" To="360" Duration="0:0:5" />
    </Storyboard>

    <Storyboard x:Name="rotateY">
        <DoubleAnimation Storyboard.TargetName="planeProjection"
                         Storyboard.TargetProperty="RotationY"
```

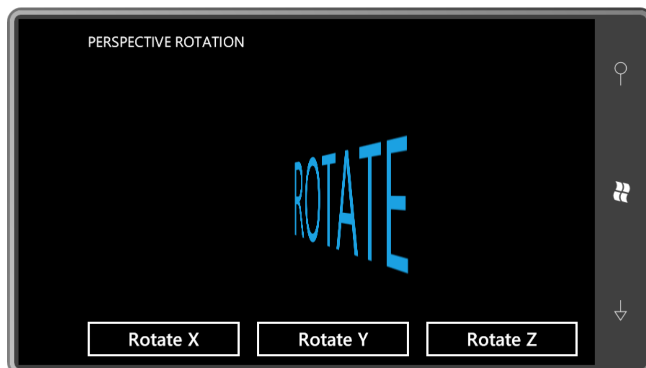
```
                            From="0" To="360" Duration="0:0:5" />
    </Storyboard>

    <Storyboard x:Name="rotateZ">
        <DoubleAnimation Storyboard.TargetName="planeProjection"
                         Storyboard.TargetProperty="RotationZ"
                         From="0" To="360" Duration="0:0:5" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Кнопки просто обеспечивают запуск соответствующей раскадровки:

Проект Silverlight: PerspectiveRotation Файл: MainPage.xaml.cs (фрагмент)

```
void RotateXClick(object sender, RoutedEventArgs args)
{
    rotateX.Begin();
}

void RotateYClick(object sender, RoutedEventArgs args)
{
    rotateY.Begin();
}

void RotateZClick(object sender, RoutedEventArgs args)
{
    rotateZ.Begin();
}
```

На иллюстрации представлено вращение вокруг оси Y:



Анимации выполняются довольно медленно, что позволяет нажать несколько кнопок и увидеть, как они взаимодействуют. Практически создается эффект вращения текста в воздухе в разных направлениях.

В двухмерном пространстве вращение выполняется относительно точки. В трехмерном пространстве вращение выполняется относительно линии, которую обычно называют осью вращения. Класс *PlaneProjection* предпочитает определять центр вращения тремя числами. Для этого используются свойства *CenterOfRotationX* (Координата X центра вращения), *CenterOfRotationY* и *CenterOfRotationZ*. В результате эти три числа определяют точку в трехмерном пространстве, координаты которой остаются неизменными в ходе вращения. *CenterOfRotationX* не оказывает влияния на вращение вокруг оси X, аналогично остальные два свойства.

Свойства *CenterOfRotationX* и *CenterOfRotationY* задаются в относительных координатах на основании размера вращаемого элемента, где точка (0, 0) соответствует верхнему левому углу. По умолчанию используются значения 0,5, что соответствует центру элемента.

Если задать *CenterOfRotationX* значение 0, свойство *RotationY* заставит элемент вращаться вокруг его левого края. Если задать *CenterOfRotationY* значение 1, свойство *RotationX* обусловит вращение элемента вокруг его нижнего края.

Свойство *CenterOfRotationZ* задается в абсолютных координатах, т.е. пикселах, где 0 соответствует плоскости экрана и положительные значения увеличиваются по направлению от экрана к пользователю. Здесь для внутренних вычислений принято, что зритель (пользователь) находится на расстоянии 1000 пикселов от экрана. В PerspectiveRotation попробуем задать свойству *CenterOfRotationZ* объекта *PlaneProjection* значение 200:

```
<TextBlock.Projection>
    <PlaneProjection x:Name="planeProjection"
                     CenterOfRotationZ="200" />
</TextBlock.Projection>
```

Теперь понажимаем на кнопки «Rotate X» (Вращать относительно оси X) и «Rotate Y». Как видим, текст как будто покидает границы экрана (где координата Z равна 0) и вращается вокруг точки с координатой Z равной 200, разворачиваясь перед пользователем в точке с координатой Z=200. Значения больше 500 для *CenterOfRotationZ* будут приводить к некорректной работе проекций. Проецируемый объект будет достигать значения Z=1000 и «бить» пользователя по носу.

Остальные свойства *PlaneProjection* обеспечивают перенос объекта в направлениях по осям X, Y и Z. Концептуально сначала применяются свойства *LocalOffsetX* (Локальное смещение по оси X) *LocalOffsetY* и *LocalOffsetZ*, затем элемент вращается, и после этого применяются свойства *GlobalOffsetX* (Общее смещение по оси X), *GlobalOffsetY* и *GlobalOffsetZ*.

Зададим значение 200 свойству *LocalOffsetX* или *GlobalOffsetX*. В любом случае исходный текст смещается вправо на 200 пикселов. Но в случае использования *GlobalOffsetX* создается впечатление, что вправо смещается весь экран. Зададим *LocalOffsetX* и повернем текст вокруг оси Y. Текст сначала сместится вправо, переместится влево и затем опять вернется вправо.

С помощью анимированных трансформаций проекций можно создавать как небольшие, так и крупные эффекты. Примером крупного эффекта является изменением способа вывода новой страницы приложения на экран. Файл MainPage.xaml приложения SweepIntoView (Дверь между страницами) включает только короткий текст:

Проект Silverlight: SweepIntoView Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock Text="Touch to go to second page[1]"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Файл выделенного кода использует касание для перехода к Page2.xaml:

Проект Silverlight: SweepIntoView Файл: MainPage.xaml.cs (фрагмент)

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    this.NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.Relative));

    args.Complete();
    args.Handled = true;
```

[1] Коснитесь, чтобы перейти на вторую страницу (*прим. переводчика*).

```
    base.OnManipulationStarted(args);
}
```

Для разнообразия (и чтобы более ясно видеть, что происходит) область содержимого Page2.xaml закрашивается контрастным фоном:

Проект Silverlight: SweepIntoView Файл: Page2.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      Background="{StaticResource PhoneAccentBrush}">
    <TextBlock Text="Touch to go back"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

В файле выделенного кода также есть перегруженный метод *OnManipulationStarted*:

Проект Silverlight: SweepIntoView Файл: Page2.xaml.cs (фрагмент)

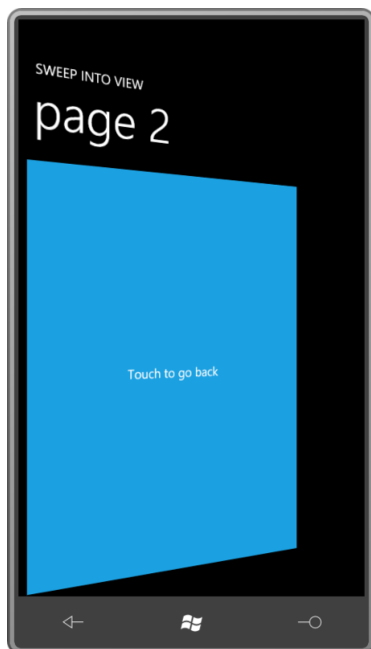
```
protected override void OnManipulationStarted(ManipulationStartedEventArgs args)
{
    this.NavigationService.GoBack();

    args.Complete();
    args.Handled = true;
    base.OnManipulationStarted(args);
}
```

Но отличает это приложение дополнительная разметка в файле Page2.xaml. Благодаря ей страница не просто неожиданно возникает на экране, а плавно «выплывает»:

Проект Silverlight: SweepIntoView Файл: Page2.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Projection>
    <PlaneProjection x:Name="planeProjection"
                     CenterOfRotationX="0" />
</phone:PhoneApplicationPage.Projection>

<phone:PhoneApplicationPage.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="planeProjection"
                                 Storyboard.TargetProperty="RotationY"
                                 From="-90" To="0" Duration="0:0:01" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</phone:PhoneApplicationPage.Triggers>
```

PlaneProjection задан для свойства *Projection* всего элемента *PhoneApplicationPage*, и анимация запускается при первой загрузке страницы. Анимация заставляет свойство *RotationY* изменяться от –90 градусов до нуля, при этом *CenterOfRotationX* задано равным нулю. Благодаря этому страница появляется на экране движением, напоминающим закрытие двери:



Анимации и приоритетность свойств

Примеры данной главы включают небольшое приложение ButtonSetAndAnimate (Задание и анимация кнопки), которое не делает ничего особенно полезного, кроме как иллюстрирует место анимации в иерархии приоритетности свойств-зависимостей.

XAML-файл описывает *Slider* с диапазоном от 0 до 100, *TextBlock*, отображающий значение *Slider*, и четыре кнопки:

Проект Silverlight: ButtonSetAndAnimate Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
               Text="{Binding ElementName=slider, Path=Value}"
               HorizontalAlignment="Center"
               Margin="24" />

    <Slider Name="slider"
            Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2"
            Minimum="0" Maximum="100"
            Orientation="Horizontal"
            VerticalAlignment="Center" />

    <Button Grid.Row="2" Grid.Column="0"
            Content="Set to 0"
            Click="OnSetToZeroClick" />
```

```
    <Button Grid.Row="2" Grid.Column="1"
            Content="Set to 100"
            Click="OnSetToOneHundredClick" />

    <Button Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2"
            Content="Animate to 50"
            HorizontalAlignment="Center"
            Click="OnAnimateTo50Click" />

    <Button Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
            Content="Set Maximum to 25"
            HorizontalAlignment="Center"
            Click="OnSetMaxTo40Click" />
</Grid>
```

Также в XAML-файле имеется анимация, целью которой является свойство *Value* объекта *Slider*.

Проект Silverlight: ButtonSetAndAnimate Файл: MainPage.xaml (фрагмент)

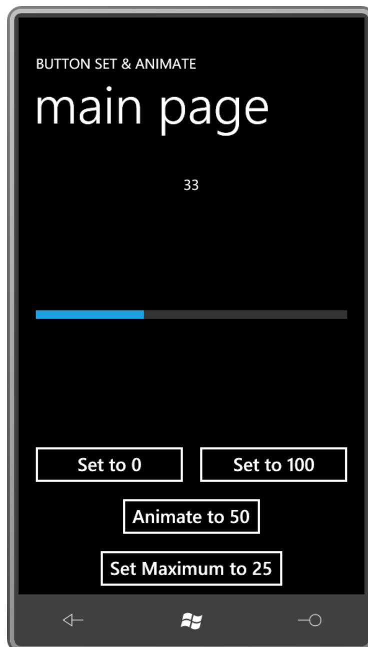
```
<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="storyboard">
        <DoubleAnimation Storyboard.TargetName="slider"
                         Storyboard.TargetProperty="Value"
                         To="50" Duration="0:0:5" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>
```

Обработчики кнопок описаны в файле выделенного кода:

Проект Silverlight: ButtonSetAndAnimate Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    void OnSetToZeroClick(object sender, RoutedEventArgs args)
    {
        slider.Value = 0;
    }

    void OnSetToOneHundredClick(object sender, RoutedEventArgs args)
    {
        slider.Value = 100;
    }

    void OnAnimateTo50Click(object sender, RoutedEventArgs args)
    {
        storyboard.Begin();
    }

    void OnSetMaxTo40Click(object sender, RoutedEventArgs e)
    {
        slider.Maximum = 25;
    }
}
```

Вот что мы получаем на экране:



Slider можно перемещать, проводя по нему пальцем или задавая его предельно допустимые значения посредством двух верхних кнопок. Так себе. Теперь щелкнем кнопку «Animate to 50» (Переместить с помощью анимации в значение 50).

В процессе анимации *Slider* и его перемещения в центральное положение попробуйте изменить это поведение, проводя по элементу пальцем или нажимая кнопки «Set to 0» (Задать равным 0) или «Set to 100». Никакого эффекта. Анимация имеет приоритет над локальными параметрами. Это означает, что схема приоритетности свойств-зависимостей (в последний раз упоминаемая нами в главе 11) должна быть дополнена анимациями, и они должны располагаться в ней в самом верху:

Анимации имеют приоритет над

Локальными параметрами, которые имеют приоритет над

Настройками стиля, которые имеют приоритет над

Стилем темы, который имеет приоритет над

Унаследованными свойствами, которые являются более приоритетными, чем

Значения по умолчанию

Так и должно быть. Анимации должны иметь приоритет над локальными параметрами или они не смогут использоваться для свойств, которые просто инициализированы некоторым значением.

По завершении анимации *Slider* доступен для манипуляций как посредством сенсорного ввода, так и с помощью двух верхних кнопок. Это поведение не является корректным и не соответствует документации. Используемое для *FillBehavior* значение по умолчанию *HoldEnd* должно приводить к «замораживанию» ползунка после окончания анимации. *Slider* должен сохранять, и его положение должно отражать конечное значение анимации.

Существует ли что-либо более приоритетное, чем анимации? Да, существует, но вряд ли это сразу придет вам в голову, и вряд ли вы сможете найти примеры этому вне *Slider* и *ScrollBar*.

Зададим *Slider* его максимальное значение и снова нажмем кнопку «Animate to 50». В момент, когда *Slider* почти достиг значения 50, щелкнем кнопку «Set Maximum to 25». Тем самым свойству *Maximum* объекта *Slider* будет задано значение 25, и анимация будет немедленно остановлена. И опять же все выглядит логичным. Независимо от того что делает анимация, *Slider* нет никакого смысла, чтобы значение его свойства *Value* выходило за рамки диапазона допустимых значений, определяемого свойствами *Minimum* и *Maximum*. Это пример принудительного задания свойства:

Принудительное задание свойства имеет приоритет над

Анимациями, которые являются более приоритетными, чем

Локальные параметры, которые имеют приоритет над

Настройками стиля, которые имеют приоритет над

Стилем темы, который имеет приоритет над

Унаследованными свойствами, которые являются более приоритетными, чем

Значения по умолчанию

Теоретически значения шаблонных свойств также должны быть включены в эту схему между локальными параметрами и настройками стиля. Но поскольку их тяжело дифференцировать, в данной книге остановимся на таком виде этой схемы.

Глава 16
Два шаблона

Шаблоны в Silverlight – это описанные в XAML деревья визуальных элементов и элементов управления. Особыми эти деревья делает то, что они используются как шаблоны или трафареты для создания идентичных деревьев визуальных элементов. Шаблоны практически всегда определяются как ресурсы, поэтому они используются совместно, и практически всегда включают привязки, поэтому могут быть ассоциированы с разными объектами и предполагают разное представление.

Один тип шаблонов (*DataTemplate*) используется для формирования визуального представления объектов, которые в противном случае его не имеют. Другой тип шаблонов (*ControlTemplate*) используется для настройки визуального представления элементов управления. На самом деле есть еще один, третий тип шаблонов (*ItemsPanelTemplate* (Шаблон панели элементов)). Он очень прост и имеет специальное применение, которое мы обсудим в следующей главе.

Несомненно, шаблон является одной из самых мощных возможностей Silverlight и, вероятно, одной из самых сложных. Поэтому многие разработчики полагаются в этом вопросе исключительно на Expression Blend. В данной главе будет продемонстрировано, как создавать шаблоны вручную, что позволит вам лучше понимать шаблоны, формируемые Expression Blend, если в дальнейшем вы решите пойти по этому пути.

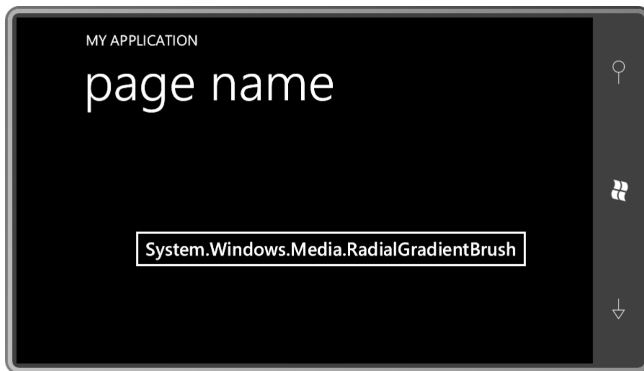
ContentControl и *DataTemplate*

В главе 10 мною было продемонстрировано, как в качестве значения свойства *Content* производного от *ContentControl* класса (например, *Button*) можно задавать практически любой объект. Если класс этого объекта наследуется от *FrameworkElement* (например, *TextBlock* или *Image*), элемент отображается внутри *ContentControl*. Но также свойству *Content* можно присвоить объект, класс которого не является производным от *FrameworkElement*. Рассмотрим *Button*, в качестве значения свойства *Content* которого задан *RadialGradientBrush*:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">

    <RadialGradientBrush>
        <GradientStop Offset="0" Color="Blue" />
        <GradientStop Offset="1" Color="AliceBlue" />
    </RadialGradientBrush>

</Button>
```

Обычно в *Button* кисть используется для свойства *Foreground*, или *Background*, или даже свойства *BorderBrush*, но никак не для *Content*. В чем здесь смысл?

Если объект, заданный как значение свойства *Content* объекта *ContentControl*, не наследуется от *FrameworkElement*, формирование его визуального представления осуществляется его методом *ToString*. А если класс не имеет перегрузки для метода *ToString*, на экран выводится полное имя класса. В этом случае данная конкретная кнопка выглядит следующим образом:



Это не совсем то, чем можно похвастать перед своими друзьями, демонстрируя свои навыки в программировании.

Вставим в *Button* класс *Clock* из главы 12:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">

    <petzold:Clock />

</Button>
```

И в этом случае в *Button* не отображается ничего ценного:

Но, несомненно, существует способ разумного представления такого объекта. Для этого в качестве значения свойства *ContentTemplate* нашего *Button* зададим объект типа *DataTemplate*. Рассмотрим синтаксис с пустым *DataTemplate*:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">

    <petzold:Clock />

    <Button.ContentTemplate>
        <DataTemplate>

        </DataTemplate>
    </Button.ContentTemplate>
</Button>
```

ContentTemplate – это одно из двух свойств, описываемых *ContentControl* и наследуемых *Button*; вторым свойством является сам *Content*.

И теперь осталось лишь разместить между тегами *DataTemplate* дерево визуальных элементов, включающее привязки к свойствам класса *Clock*:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">
```

```
    <petzold:Clock />

    <Button.ContentTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="The time is:"
                           TextAlignment="Center" />
                <StackPanel Orientation="Horizontal"
                            HorizontalAlignment="Center">
                    <TextBlock Text="{Binding Hour}" />
                    <TextBlock Text=":" />
                    <TextBlock Text="{Binding Minute}" />
                    <TextBlock Text=":" />
                    <TextBlock Text="{Binding Second}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </Button.ContentTemplate>
</Button>
```

Объект *Button* использует дерево визуальных элементов для отображения объекта *Content*. Привязки в этом дереве обычно довольно простые. Этим привязкам данных не нужно свойство *Source*, потому что ассоциированный с этим деревом визуальных элементов *DataContext* – это и есть объект, заданный как значение свойства *Content*. Для представленных здесь привязок требуется задать лишь свойства *Path*, и часть «Path=» расширения разметки *Binding* можно опустить.

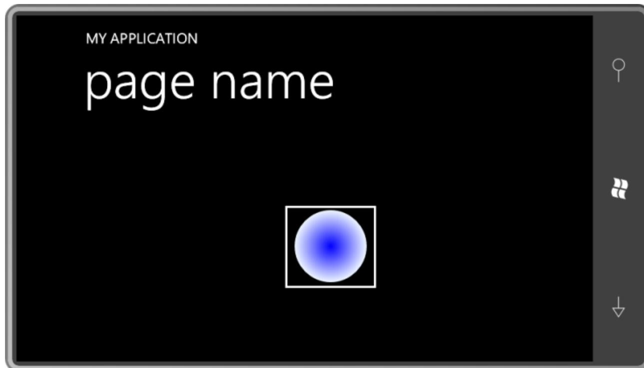
Теперь время отображается и динамически обновляется. Конечно, чтобы минуты и секунды всегда были представлены двумя разрядами, привязки для свойств *Minute* и *Second* должны ссылаться на конвертер, форматирующий строковые значения.

Существование *DataTemplate* означает, что теперь мы действительно *можем* задавать в качестве содержимого *Button* объект *RadialGradientBrush*. Для этого надо просто описать дерево визуальных элементов, использующее эту кисть в *DataTemplate*:

```
<Button HorizontalAlignment="Center"
        VerticalAlignment="Center">

    <RadialGradientBrush>
        <GradientStop Offset="0" Color="Blue" />
        <GradientStop Offset="1" Color="AliceBlue" />
    </RadialGradientBrush>

    <Button.ContentTemplate>
        <DataTemplate>
            <Ellipse Width="100"
                     Height="100"
                     Fill="{Binding}" />
        </DataTemplate>
```

```
    </Button.ContentTemplate>
</Button>
```

Обратите внимание на значение свойства *Fill* объекта *Ellipse*. Для него просто задано расширение разметки *Binding* без задания параметра *Path*. Свойству *Fill* не нужно какое-то отдельное свойство *RadialGradientBrush*. Ему нужен объект в целом. И вот какую кнопку мы получаем:



Эта техника может использоваться с любым производным от *ContentControl* классом или даже с самим *ContentControl*.

Опишем *DataTemplate* в коллекции *Resources* файла MainPage.xaml:

Проект Silverlight: ContentControlWithDataTemplates Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <DataTemplate x:Key="brushTemplate">
        <Ellipse Width="100"
                 Height="100"
                 Fill="{Binding}" />
    </DataTemplate>
</phone:PhoneApplicationPage.Resources>
```

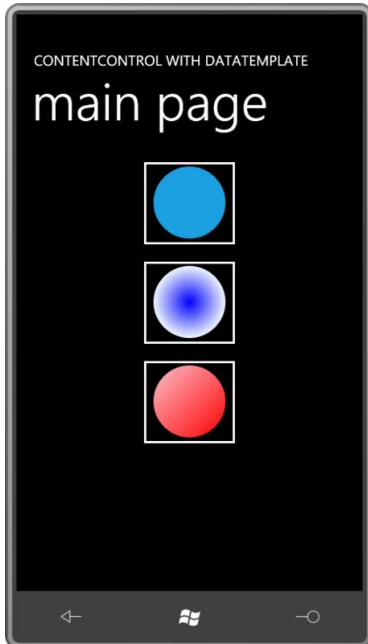
Разместим на панели содержимого этой страницы три экземпляра *Button*. В качестве значения свойства *ContentTemplate* всех трех экземпляров зададим этот ресурс, но в каждом случае для свойства *Content* будем использовать разные типы объектов *Brush*:

Проект Silverlight: ContentControlWithDataTemplates Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <Button HorizontalAlignment="Center"
                ContentTemplate="{StaticResource brushTemplate}">
            <SolidColorBrush Color="{StaticResource PhoneAccentColor}" />
        </Button>

        <Button HorizontalAlignment="Center"
                ContentTemplate="{StaticResource brushTemplate}">
            <RadialGradientBrush>
                <GradientStop Offset="0" Color="Blue" />
                <GradientStop Offset="1" Color="AliceBlue" />
            </RadialGradientBrush>
       </Button>

       <Button HorizontalAlignment="Center"
                ContentTemplate="{StaticResource brushTemplate}">
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="Pink" />
```

```
                <GradientStop Offset="1" Color="Red" />
            </LinearGradientBrush>
        </Button>
    </StackPanel>
</Grid>
```

В результате получаем следующее:



DataTemplate определен как ресурс, поэтому он может использоваться совместно всеми элементами управления *Button*. Но на основании этого шаблона для каждого *Button* а строится собственное дерево визуальных элементов. Где-то в дереве визуальных элементов для каждого *Button* определен *Ellipse*, в качестве значения свойства *Content* которого используется привязка к *SolidColorBrush*.

Разделение *DataTemplate* между несколькими элементами управления *Button* удобно, но не демонстрирует всей мощи этой техники. Только представьте, что можно описать *DataTemplate* для отображения элементов *ListBox* или *ComboBox*. Как это делать, будет показано в следующей главе.

Анализ дерева визуальных элементов

В предыдущем разделе были упомянуты деревья визуальных элементов. Рассмотрим несколько примеров.

В приложении ButtonTree (Дерево кнопки) создается дерево визуальных элементов для довольно стандартного *Button* (свойству *Content* которого задан просто текст); *Button*, значением свойства *Content* которого задан элемент *Image*; и еще двух кнопок, значениями свойств *Content* которых заданы объекты *RadialGradientBrush* и *Clock* (как показано в примере выше) посредством *ContentTemplate*. Каждый *Button* отображается в отдельной ячейке *Grid*:

Проект Silverlight: ButtonTree   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
```

```
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
       <ColumnDefinition Width="*" />
       <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Grid.Row="0" Grid.Column="0"
            Content="Click to Dump"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick" />

    <Button Grid.Row="0" Grid.Column="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick">
        <Image Source="ApplicationIcon.png"
               Stretch="None" />
    </Button>

    <Button Grid.Row="1" Grid.Column="0"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick">
        <Button.Content>
            <RadialGradientBrush>
                <GradientStop Offset="0" Color="Blue" />
                <GradientStop Offset="1" Color="AliceBlue" />
            </RadialGradientBrush>
        </Button.Content>

        <Button.ContentTemplate>
            <DataTemplate>
                <Ellipse Width="100"
                         Height="100"
                         Fill="{Binding}" />
            </DataTemplate>
        </Button.ContentTemplate>
    </Button>

    <Button Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick">
        <Button.Content>
            <petzold:Clock />
        </Button.Content>

        <Button.ContentTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="The time is:"
                               TextAlignment="Center" />
                    <StackPanel Orientation="Horizontal"
                                HorizontalAlignment="Center">
                        <TextBlock Text="{Binding Hour}" />
                        <TextBlock Text=":" />
                        <TextBlock Text="{Binding Minute}" />
                        <TextBlock Text=":" />
                        <TextBlock Text="{Binding Second}" />
                    </StackPanel>
                </StackPanel>
            </DataTemplate>
        </Button.ContentTemplate>
    </Button>

    <ScrollViewer Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
```

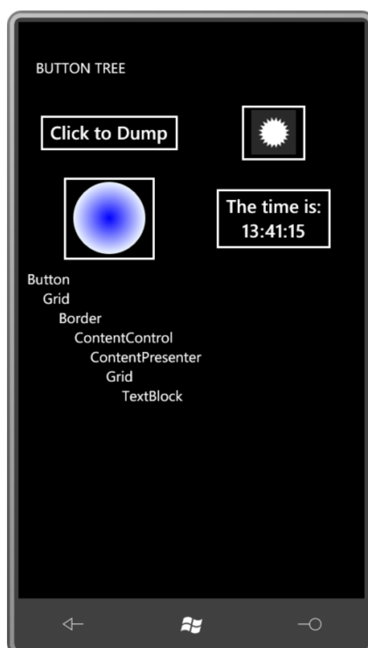
```
                    HorizontalScrollBarVisibility="Auto">
        <StackPanel Name="stackPanel" />
    </ScrollViewer>
</Grid>
```

В конце этого фрагмента в *ScrollViewer* описан *StackPanel* для отображения этого визуального дерева. В файле выделенного кода с помощью рекурсивного метода статического класса *VisualTreeHelper* (Вспомогательный класс для работы с визуальным деревом) выполняется перечисление дочерних объектов элемента, и затем их имена отображаются в виде списка иерархии:

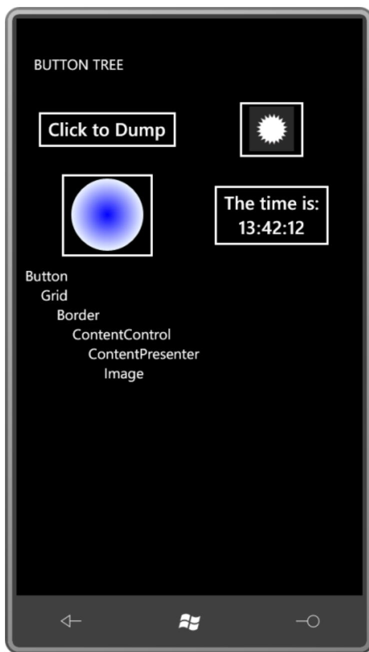
Проект Silverlight: ButtonTree Файл: MainPage.xaml.cs (фрагмент)

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
    Button btn = sender as Button;
    stackPanel.Children.Clear();
    DumpVisualTree(btn, 0);
}

void DumpVisualTree(DependencyObject parent, int indent)
{
    TextBlock txtblk = new TextBlock();
    txtblk.Text = String.Format("{0}{1}", new string(' ', 4 * indent),
                                      parent.GetType().Name);
    stackPanel.Children.Add(txtblk);

    int numChildren = VisualTreeHelper.GetChildrenCount(parent);

    for (int childIndex = 0; childIndex < numChildren; childIndex++)
    {
        DependencyObject child = VisualTreeHelper.GetChild(parent, childIndex);
        DumpVisualTree(child, indent + 1);
    }
}
```

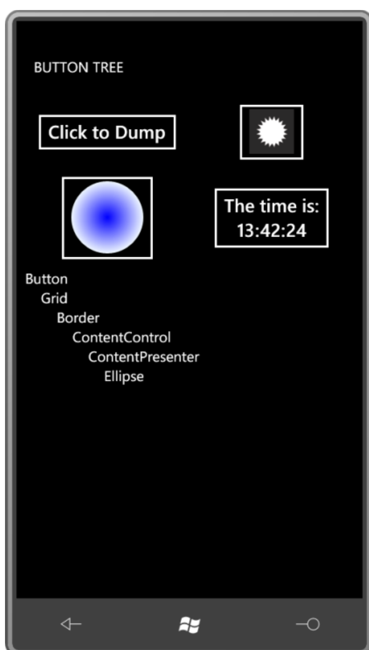
Щелкните кнопку в правом верхнем углу, свойству *Content* которого задан текст, и приложение выведет на экран дерево визуальных элементов объекта *Button*:



Нет ничего удивительного в присутствии элемента *Border*, он ясно виден на *Button* с *TextBlock*, используемом для отображения текста. Очевидно, что первый *Grid*, в котором размещается *Border*, является *Grid* с одной ячейкой; позднее в этой главе я покажу предназначение этого *Grid*. Цель использования *Grid* для размещения *TextBlock* не так очевидна. Если в качестве значения свойства *Content* элемента *Button* явно задать *TextBlock*, второй *Grid* исчезает, и дерево становится больше похожим на дерево для *Button*, в качестве значения свойства *Content* которого задан элемент *Image*:

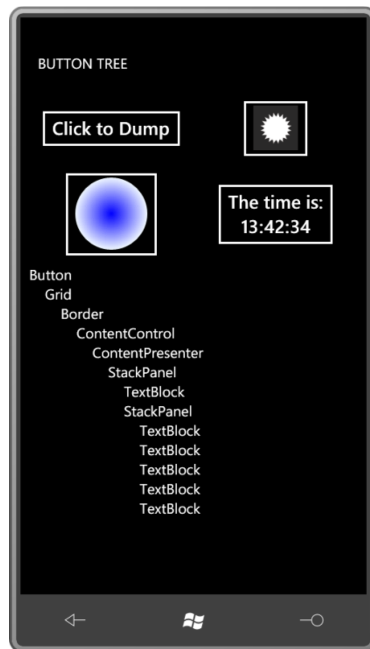


Часть дерева визуальных элементов вплоть до *ContentPresenter* (Средство отображения содержимого) определяет представление стандартного *Button*. (Вскоре мы увидим, как эту часть можно заменить, если задать свойству *Template* элемента управления объект типа *ControlTemplate*.) Все что располагается ниже *ContentPresenter*, используется для отображения содержимого *Button*. Рассмотрим, как выглядит дерево, если свойству *Content* задан *RadialGradientBrush*, но *ContentTemplate* задан *Ellipse*, ссылающийся на эту кисть:



Те кто знает шаблоны элементов управления по Windows Presentation Foundation или веб-версии Silverlight, несомненно, ожидают увидеть *ContentPresenter*. Это производный от *FrameworkElement* класс, специально предназначенный для размещения содержимого. Именно *ContentPresenter* форматирует некоторые объекты как текст в отсутствие *DataTemplate* или фактически применяет *DataTemplate*. Но *ContentControl* может привести в замешательство. Я тоже был сбит им с толку сначала. *Button* наследуется от *ContentControl*, но это совсем не означает, что дерево визуальных элементов должно включать еще один *ContentControl*! Я еще вернусь к странному представлению *ContentControl* позже в этой главе.

Наконец, вот такое дерево визуальных элементов мы получаем при использовании более развернутого *DataTemplate* для отображения объекта *Clock*:



Теперь мы знаем, как описывать часть дерева визуальных элементов после *ContentPresenter* для любого элемента управления, наследуемого от *ContentControl*. Далее рассмотрим, как можно переопределить верхнюю часть этого дерева визуальных элементов.

Основы *ControlTemplate*

DataTemplate позволяет настраивать представление содержимого *ContentControl*. *ControlTemplate*, который можно задать как значение свойства *Template* любого *Control*, обеспечивает возможность настраивать представление самого элемента управления, что часто называют визуальным стилем элемента управления. Эти два разных назначения отражены в следующей таблице:

Свойство	Тип свойства	Назначение
Template	*ControlTemplate*	настраивает визуальный стиль элемента управления
ContentTemplate	*DataTemplate*	настраивает представление содержимого

Не забывайте, что свойство *ContentTemplate* описано классом *ContentControl*, и его можно найти только в классах, наследуемых от *ContentControl*. Но свойство *Template* определено классом *Control*, и его присутствие является, наверное, основным отличием элементов управления от производных от *FrameworkElement*, таких как *TextBlock* и *Image*.

Когда у разработчика возникает мысль о том, что ему нужен пользовательский элемент управления, он должен прежде всего задать себе вопрос, а действительно ли ему нужен

новый элемент управления, или достаточно будет изменить внешний вид уже существующего. Например, требуется элемент управления определенного вида, который должен будет менять свой внешний вид по касанию пользователя, а при повторном касании возвращаться к исходному представлению. Но это просто *ToggleButton* с другими визуальными элементами, т.е. другим *ControlTemplate*.

Как и стили, очень часто шаблоны определяются как ресурсы. Как и *Style*, *ControlTemplate* требует задания *TargetType*:

```
<ControlTemplate x:Key="btnTemplate" TargetType="Button">
    …
</ControlTemplate>
```

Очень часто *Template* описывается как часть *Style*:

```
<Style x:Key="btnStyle" TargetType="Button">
    <Setter Property="Margin" Value="6" />
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                …
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Обратите внимание, что для описания метода *Setter*, который задает в качестве значения свойства *Template* объект типа *ControlTemplate*, используется синтаксис свойство-элемент. Определение шаблона как части стиля – очень распространенный подход, потому что обычно требуется задать некоторые свойства элемента управления, чтобы сделать их более соответствующими создаваемому шаблону. Эти теги *Setter* эффективно переопределяют значения по умолчанию свойств элемента управления, к которому применяется стиль и шаблон, но они могут быть переопределены локальными параметрами конкретного элемента управления.

Создадим собственный пользовательский *Button*. Этот новый *Button* полностью сохранит функциональность обычного *Button*, но при этом разработчик будет иметь полный контроль за его представлением. Конечно, чтобы не вводить дополнительной сложности, новый *Button* на вид не будет отличаться от обычного *Button*, но при этом будет демонстрировать все используемые концепции!

Начнем со стандартного *Button* с текстовым содержимым, выравнивание которого задано таким образом, что он занимает лишь столько места, сколько ему необходимо для отображения содержимого:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
</Button>
```

Чтобы более свободно экспериментировать с *ControlTemplate*, не будем описывать его как ресурс, но вынесем его свойство *Template* как свойство-элемент объекта *Button* и зададим *ControlTemplate* в качестве его значения:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Template>
        <ControlTemplate TargetType="Button">

        </ControlTemplate>
```

```
    </Button.Template>
</Button>
```

Как только свойству *Template* задается пустой *ControlTemplate*, кнопка исчезает. Дерева визуальных элементов, которое определяет внешний вид элемента управления, больше не существует. Именно это дерево мы и будем помещать в *ControlTemplate*. Чтобы гарантированно ничего не повредить, вставим *TextBlock* в *ControlTemplate*:

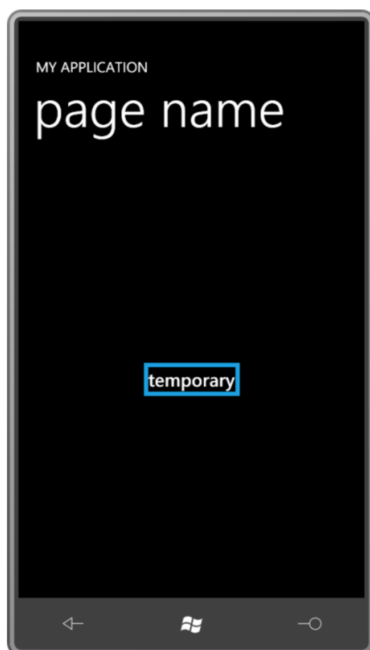
```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <TextBlock Text="temporary" />
        </ControlTemplate>
    </Button.Template>
</Button>
```

Теперь *Button* описывается одним словом «temporary» (временно). Он не обеспечивает никакой обратной связи при касании, но во всем остальном это полнофункциональная кнопка. Несомненно, в ней есть серьезные недостатки, потому что на самом деле на *Button* должна отображаться надпись «Click me!», но это мы скоро исправим.

Зададим рамку вокруг *TextBlock*:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <Border BorderBrush="{StaticResource PhoneAccentBrush}"
                    BorderThickness="6">
                <TextBlock Text="temporary" />
            </Border>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Вот как это выглядит теперь:



На самом деле, явно задавать значения свойств в коде, как это сделано здесь в шаблоне, не очень хорошая идея, особенно если этот шаблон предполагается для совместного использования многими элементами управления. Вообще нет смысла задавать в шаблоне *BorderBrush* и *BorderThickness*, потому что эти свойства определяет сам класс *Control*. Если мы действительно хотим задать рамку вокруг кнопки, мы должны задавать эти свойства в *Button*, а не в шаблоне, потому что этот шаблон может использоваться совместно несколькими кнопками, для которых потребуются рамки разной толщины и отрисованные разными кистями.

Итак, перенесем эти свойства из шаблона в саму кнопку:

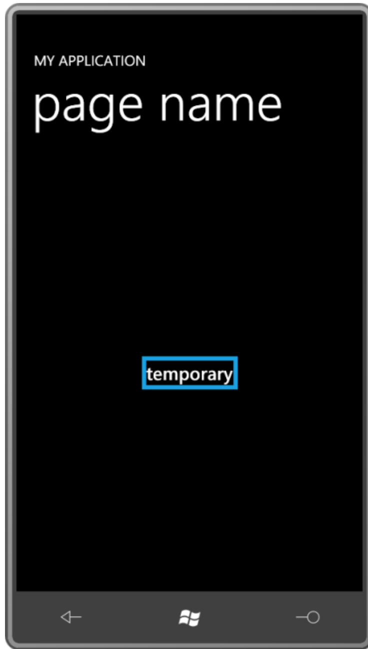
```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        BorderBrush="{StaticResource PhoneAccentBrush}"
        BorderThickness="6">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <Border>
                <TextBlock Text="temporary" />
            </Border>
        </ControlTemplate>
    </Button.Template>
</Button>
```

К сожалению, теперь этих свойств нет в дереве визуальных элементов шаблона, поэтому рамка исчезла, и особого улучшения не видно. Рамка, описанная в шаблоне, не наследует автоматически свойства *BorderBrush* и *BorderThickness*, заданные для кнопки. Это не наследуемые свойства.

Чтобы свойства рамки в шаблоне получали такие же значения, что и свойства в *Button*, необходимо использовать привязку. Это особый тип привязки, который имеет собственное расширение разметки. Она называется *TemplateBinding*:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        BorderBrush="{StaticResource PhoneAccentBrush}"
        BorderThickness="6">
    <Button.Template>
        <ControlTemplate TargetType="Button">
            <Border BorderBrush="{TemplateBinding BorderBrush}"
                    BorderThickness="{TemplateBinding BorderThickness}">
                <TextBlock Text="temporary" />
            </Border>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Применение *TemplateBinding* означает, что свойства этого конкретного элемента в дереве визуальных элементов шаблона – в частности, свойства *BorderBrush* и *BorderThickness* элемента *Border* – будут иметь значения, такие же как заданы аналогичным свойствам в самом элементе управления. Теперь у *Button* появилась рамка контрастного цвета:



Синтаксически привязка *TemplateBinding* очень проста. Ее целью всегда является свойство зависимости из дерева визуальных элементов шаблона. Она всегда ссылается на свойство элемента управления, к которому применяется шаблон. Больше в расширении разметки *TemplateBinding* ничего не может быть. *TemplateBinding* может использоваться только в деревьях визуальных элементов, определенных в *ControlTemplate*.

С другой стороны, в *TemplateBinding* нет ничего особенного. Фактически, это сокращенная форма, и когда вы увидите ее полную версию, вы будете счастливы, что такая сокращенная форма существует. Задание атрибута

```
BorderBrush="{TemplateBinding BorderBrush}"
```

является сокращенной записью такого выражения:

```
BorderBrush="{Binding RelativeSource={RelativeSource TemplatedParent},
                      Path=BorderBrush}"
```

Это привязка к объекту *Border*, и синтаксис *RelativeSource* ссылается на еще один элемент дерева, имеющий отношение к этому *Border*. *TemplatedParent* (Родитель для применения шаблона) – это *Button*, к которому применяется этот шаблон, так что привязка указывает на *BorderBrush* этого *Button*. (Понятно? Нет?) Эта альтернатива *TemplateBinding* пригодится, когда требуется установить двунаправленную привязку к свойству шаблона, потому что *TemplateBinding* обеспечивает только однонаправленную привязку и не предоставляет свойства *Mode*.

Вернемся к рассматриваемому шаблону. Теперь, когда мы описали *TemplateBinding* для *BorderBrush* и *BorderThickness*, возник другой вопрос. Пусть решено, что толщина рамки данной кнопки должна составлять 6 пикселов и быть закрашенной контрастным цветом, но такие значения можно обеспечить, только явно задав свойства *BorderBrush* и *BorderThickness* в *Button*. Было бы здорово, если бы эти свойства не надо было задавать в *Button*. Иначе говоря, мы хотим, чтобы эти свойства в *Button* имели значения по умолчанию, которые могут быть переопределены локальными параметрами.

Это можно реализовать, задав желаемые значения по умолчанию в *Style*. Для удобства я определил такой *Style* прямо в *Button*:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
        </Style>
    </Button.Style>

    <Button.Template>
        <ControlTemplate TargetType="Button">
            <Border BorderBrush="{TemplateBinding BorderBrush}"
                    BorderThickness="{TemplateBinding BorderThickness}">
                <TextBlock Text="temporary" />
            </Border>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Теперь шаблон получает некоторые значения по умолчанию от *Style*, но эти настройки могут быть переопределены локально в кнопке. (Если вы не хотите, чтобы эти свойства переопределялись локальными параметрами и всегда имели точно определенные значения, задавайте их явно прямо в коде шаблона.)

Очень часто свойство *Template* описывается как часть *Style* следующим образом:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Border BorderBrush="{TemplateBinding BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}">
                            <TextBlock Text="temporary" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Button.Style>
</Button>
```

Теперь *Style* задает значения по умолчанию для свойств, которые также используются шаблоном. Добавим свойство *Background* в *Border* и тоже зададим для него значение по умолчанию:

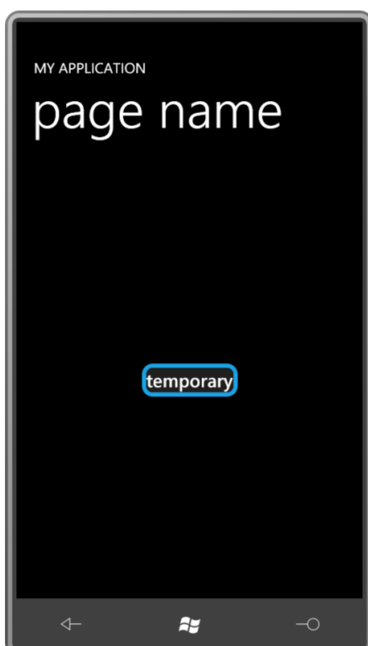
```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
            <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}"
/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
```

```
                        <Border BorderBrush="{TemplateBinding BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}"
                                Background="{TemplateBinding Background}">
                            <TextBlock Text="temporary" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Button.Style>
</Button>
```

Но возможно, мы хотим, чтобы наша новая кнопка имела скругленные углы. Нам известно, что *Button* не описывает свойства *CornerRadius*, поэтому оно может быть задано явно прямо в шаблоне:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
            <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}"
/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Border BorderBrush="{TemplateBinding BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}"
                                Background="{TemplateBinding Background}"
                                CornerRadius="12">
                            <TextBlock Text="temporary" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Button.Style>
</Button>
```

Вот что мы имеем на данный момент:



На кнопке по-прежнему отображается надпись «temporary», хотя на самом деле на ней должно быть написано «Click me!» (Щелкни меня!). Велик соблазн вставить здесь *TextBlock* и задать его свойство *Text* в *TemplateBinding* свойства *Content* элемента управления *Button*:

```
<TextBlock Text="{TemplateBinding Content}" />
```

Такая схема будет вполне работоспособной в данном примере, но это очень и очень неправильно. Проблема в том, что свойство *Content* класса *Button* типа *object*. В качестве его значения может быть задано все, что угодно – *Image*, *Panel*, *Shape*, *RadialGradientBrush* – и это может обусловить небольшие неприятности для *TextBlock*.

К счастью в Silverlight есть класс, который существует специально для отображения содержимого в производном от *ContentControl* классе. Этот класс носит имя *ContentPresenter*. У него есть свойство *Content* типа *object*, и *ContentPresenter* выводит этот объект на экран независимо от того, является ли он просто строкой или каким-либо иным элементом:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
            <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}"
/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Border BorderBrush="{TemplateBinding BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}"
                                Background="{TemplateBinding Background}"
                                CornerRadius="12">
                            <ContentPresenter Content="{TemplateBinding Content}" />
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Button.Style>
</Button>
```

Обратите внимание, как свойство *Content* класса *ContentPresenter* связано со свойством *Content* класса *Button*. *ContentPresenter* обладает отличительным преимуществом возможности работы с любыми видами объектов. *ContentPresenter* может создавать собственное дерево визуальных элементов. Например, если *Content* строкового типа, *ContentPresenter* создает *TextBlock* для отображения этой строки. *ContentPresenter* также доверено создавать дерево визуальных элементов для отображения содержимого на основании *DataTemplate*, заданного для *Control*. Для этой цели у *ContentPresenter* есть собственное свойство *ContentTemplate*, которое можно связать посредством привязки с *ContentTemplate* элемента управления:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}"
/>
            <Setter Property="BorderThickness" Value="6" />
            <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}"
/>
            <Setter Property="Template">
                <Setter.Value>
```

```
                    <ControlTemplate TargetType="Button">
                        <Border BorderBrush="{TemplateBinding BorderBrush}"
                                BorderThickness="{TemplateBinding BorderThickness}"
                                Background="{TemplateBinding Background}"
                                CornerRadius="12">
                            <ContentPresenter
                                Content="{TemplateBinding Content}"
                                ContentTemplate="{TemplateBinding ContentTemplate}"
/>
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Button.Style>
</Button>
```

Эти два присваивания *TemplateBinding* для *ContentPresenter* настолько стандартные, что они не требуют явного задания! Все будет сделано автоматически. Но я чувствую себя комфортнее, когда вижу их заданными явно.

Давайте вспомним, что класс *Control* описывает свойство *Padding*, предназначенное для создания небольшого отступа вокруг содержимого элемента управления. Зададим свойство *Padding* для нашего *Button*:

```
<Button Content="Click me!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Padding="24">
    …
</Button>
```

Ничего не происходит. Дерево визуальных элементов должно включить это свойство *Padding*. Оно должно обеспечить небольшой зазор между *Border* и *ContentPresenter*. Как это сделать? Одним из решений будет применить *TemplateBinding* к свойству *Padding* объекта *Border*. Но если в *Border* будет располагаться еще какое-то содержимое, кроме *ContentPresenter*, ничего не получится. Стандартный подход в данном случае – задать *TemplateBinding* для свойства *Margin* объекта *ContentPresenter*:

```
<ContentPresenter
        Content="{TemplateBinding Content}"
        ContentTemplate="{TemplateBinding ContentTemplate}"
        Margin="{TemplateBinding Padding}" />
```

Чтобы получить желаемый эффект, значение *Padding* для *Button* задавать не требуется. *Style* темы для *Button* определяет значение *Padding*, которое хорошо подходит для данного *Button* даже со скругленными углами *Border*.

Теперь зададим свойствам *HorizontalAlignment* и *VerticalAlignment* нашего *Button* значение *Stretch*. Все работает нормально, поэтому в шаблоне об этом можно не беспокоиться. Аналогично можно задать для *Button* свойство *Margin*, и оно тоже будет распознаваться системой компоновки.

Но после того, как свойствам *HorizontalAlignment* и *VerticalAlignment* кнопки задано значение *Stretch*, все содержимое *Button* остается в его верхнем левом углу:

Класс *Control* определяет два свойства, *HorizontalContentAlignment* и *VerticalContentAlignment*, посредством которых можно управлять выравниванием содержимого в *ContentControl*. Но если задать эти свойства для нашей кнопки, обнаружится, что они не работают.

Это свидетельствует о том, что в шаблон требуется что-то добавить для обработки этих свойств. Мы должны выровнять *ContentPresenter* в *Border* с помощью свойств *HorizontalContentAlignment* и *VerticalContentAlignment*. Осуществляется это путем предоставления разметки *TemplateBinding*, целевыми свойствами которой задаются свойства *HorizontalAlignment* и *VerticalAlignment* объекта *ContentPresenter*:

```
<ContentPresenter
        Content="{TemplateBinding Content}"
        ContentTemplate="{TemplateBinding ContentTemplate}"
        Margin="{TemplateBinding Padding}"
        HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
        VerticalAlignment="{TemplateBinding VerticalContentAlignment}" />
```

Опять же, это очень стандартная разметка для *ContentPresenter*, которая обычно копируется из приложения в приложение.

Задание свойств шрифта или свойства *Foreground* в *Button* приводит к изменению текста кнопки соответствующим образом. Эти свойства наследуются по дереву визуальных элементов шаблона, и разработчику не надо что-либо добавлять в шаблон, чтобы применить их. (Но в стиле темы для *Button* явно заданы свойства *Foreground*, *FontFamily* и *FontSize*, поэтому сам *Button* не может унаследовать эти свойства через дерево визуальных элементов, и очевидно, что мы ничего не можем сделать в пользовательском *Style* для изменения этого поведения.)

Диспетчер визуальных состояний

Все это время, пока мы изменяли внешний вид *Button* с помощью шаблона, кнопка оставалась полностью функциональной и формировала события *Click* при каждом нажатии. Большая проблема в том, что *Button* не предоставляет визуальной обратной связи пользователю. Внешний вид кнопки настроен, но не меняется при манипуляциях с кнопкой.

На самом деле, чтобы сделать этот шаблон функционально и визуально завершенным, необходимо добавить в него всего две возможности:

- *Button* должен обеспечивать визуальную обратную связь при нажатии кнопки пользователем.
- Находясь в неактивном состоянии, *Button* должен визуально отображать это.

Эти две возможности взаимосвязаны, потому что обе касаются изменения визуальных элементов элемента управления в определенных условиях. Также их связывает используемый для их реализации инструмент Silverlight, называемый Visual State Manager (Диспетчер визуальных состояний).

Visual State Manager помогает разработчику работать с *визуальными состояниями*. Визуальные состояния – это изменения визуальных элементов элемента управления, являющиеся результатом изменений свойств (или состояний) элемента управления. Все важные визуальные состояния класса *Button* в Windows Phone 7 связаны со свойствами *IsPressed* (Нажат) и *IsEnabled* (Включен).

Все визуальные состояния, поддерживаемые конкретным элементом управления, описаны в его документации. На первой странице документации класса *Button* можно увидеть класс, определенный шестью атрибутами типа *TemplateVisualStateAttribute* (Атрибут визуального состояния шаблона):

```
[TemplateVisualStateAttribute(Name = "Disabled", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "Normal", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "MouseOver", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "Pressed", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "Unfocused", GroupName = "FocusStates")]
[TemplateVisualStateAttribute(Name = "Focused", GroupName = "FocusStates")]
public class Button : ButtonBase
```

Класс *Button* имеет шесть визуальных состояний. У каждого из этих состояний есть имя, но также обратите внимание, что для каждого из них указано и имя группы: CommonStates (Общие состояния) или FocusStates (Состояния фокуса).

В рамках группы визуальные состояния являются взаимоисключающими, т.е. к *Button* одновременно может применяться только одно состояние. Соответственно состояниями группы CommonStates кнопка может быть либо в обычном состоянии (Normal), либо неактивной (Disabled), либо с указателем мыши над ней, либо нажатой. Нет необходимости беспокоиться о сочетаниях этих состояний и создавать особое состояние для случая, когда указатель мыши проходит над неактивной кнопкой, например, потому что эти два состояния никогда не будут иметь место в один и тот же момент времени.

Реализация перехода кнопки в определенное состояние осуществляется в коде класса *Button* посредством вызовов статического метода *VisualStateManager.GoToState* (Перейти в состояние). Шаблон отвечает за визуальные изменения на основании этих состояний.

В шаблонах для Windows Phone 7 все намного проще, чем в Silverlight для Веб, потому что здесь нам не надо беспокоиться о двух состояниях группы FocusStates или о состоянии MouseOver (Наведение указателя мыши). Таким образом, остаются только состояния Normal, Disabled и Pressed.

Очень часто в шаблон включают дополнительные элементы специально для реализации этих визуальных состояний. Неактивное состояние элемента управления обычно обозначается через более тусклое отображение содержимого элемента независимо от природы этого содержимого: будь то текст, растровое изображение или что-то еще. Таким образом,

неактивное состояние может быть реализовано путем помещения полупрозрачного *Rectangle* поверх всего элемента управления.

Итак, давайте поместим все дерево визуальных элементов в *Grid* с одной ячейкой и добавим *Rectangle* в конце, чтобы он располагался поверх всех элементов:

```
<ControlTemplate TargetType="Button">
    <Grid>
        <Border BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}"
                Background="{TemplateBinding Background}"
                CornerRadius="24">
            <ContentPresenter
                    Content="{TemplateBinding Content}"
                    ContentTemplate="{TemplateBinding ContentTemplate}"
                    Margin="{TemplateBinding Padding}"
                    HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}"
                    VerticalAlignment="{TemplateBinding VerticalContentAlignment}" />
        </Border>

        <Rectangle Name="disableRect"
                   Fill="{StaticResource PhoneBackgroundBrush}"
                   Opacity="0" />
    </Grid>
</ControlTemplate>
```

К счастью, свойство *Opacity* нового *Rectangle* имеет значение 0. В противном случае этот *Rectangle* загораживал бы весь элемент управления! Но если задать *Opacity* значение 0,6, например, мы получим необходимый эффект затемнения, не зависящий от содержимого элемента управления.

Обратите внимание, при задании цвета *Rectangle* используется ресурс PhoneBackgroundBrush. Углы нашего *Button* скруглены, и мы совсем не хотим, чтобы *Rectangle* искажал цвет элементов, находящихся за *Button* и видимых из-за этих скруглений. Также можно задать для *Rectangle* такое же скругление углов, как и для *Border*, что обеспечит большую гибкость при выборе цвета для *Rectangle*.

Теперь, когда *Rectangle* на месте, нам осталось лишь найти способ изменять значение *Opacity* с 0 на 0,6 при переходе кнопки в состояние Disabled.

Разметка для Visual State Manager всегда располагается после открывающего тега элемента верхнего уровня шаблона, в данном случае это *Grid*. Эта разметка начинается с тега *VisualStateManager.VisualStateGroups* (Группы визуальных состояний), в рамках которого может быть множество разделов *VisualStateGroups*. Я не буду включать группу FocusStates:

```
<ControlTemplate TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">

            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        …
    </Grid>
</ControlTemplate>
```

Теги *VisualStateGroup* включают наборы тегов *VisualState* (Визуальное состояние) для каждого визуального состояния этой группы:

```
<ControlTemplate TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
```

```
                <VisualState x:Name="Normal" />
                <VisualState x:Name="MouseOver" />

                <VisualState x:Name="Pressed">

                </VisualState>

                <VisualState x:Name="Disabled">

                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        …
    </Grid>
</ControlTemplate>
```

Тег *VisualState* для состояния Normal пуст, потому что шаблон изначально создается для кнопки в обычном состоянии. Однако этот тег нельзя опустить, потому что в этом случае элемент управления не сможет вернуться в состояние Normal после пребывания в другом состоянии. Состояние MouseOver не используется, поэтому тоже остается пустым.

В тегах *VisualState* мы указываем, что должно происходить, когда элемент управления находится в данном состоянии. Как это делается? Можно предположить, что для этого используется тег *Setting*, как в *Style*, и такой подход прекрасно работал бы. Но Visual State Manager обеспечивает намного большую гибкость и позволяет использовать анимации. А поскольку синтаксис анимаций не намного сложнее синтаксиса *Setting*, Visual State Manager буквально требует применения анимаций. В теги *VisualState* мы помещаем *Storyboard*, включающий одну или более анимаций, целевыми свойствами которых являются свойства именованных элементов шаблона. В большинстве случаем для этих анимаций будет задана продолжительность (*Duration*), равная 0, что обеспечивает мгновенное изменение визуального состояния. Но по желанию можно сделать анимации смены состояний более плавными. Рассмотрим анимацию свойства *Opacity* объекта *Rectangle* под именем *disableRect*:

```
<ControlTemplate TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal" />
                <VisualState x:Name="MouseOver" />

                <VisualState x:Name="Pressed">

                </VisualState>

                <VisualState x:Name="Disabled">
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="disableRect"
                                         Storyboard.TargetProperty="Opacity"
                                         To="0.6" Duration="0:0:0" />
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        …

    </Grid>
</ControlTemplate>
```

Как правило, анимациям в Visual State Manager не задано значение *From*, поэтому они просто стартуют от существующего значения. Пустой тег *VisualState* для состояния Normal при переходе элемента управления в это состояние эффективно восстанавливает для *Opacity* значение, которое это свойство имело до анимации.

Реализация состояния Pressed представляет некоторые сложности. Как правило, состояние Pressed визуализируется в форме обратного видео. В Веб-версии Silverlight прямо в коде шаблона *Button* в качестве фона задан *LinearGradientBrush*, и для состояния Pressed изменяются значения свойств этой кисти. Поскольку шаблон управляет кистью для состояния Normal, он без труда может изменить эту кисть для отображения состояния Pressed.

В создаваемом здесь шаблоне *Button* цвет *Foreground* по умолчанию задан в стиле темы для *Button*, а цвет *Background* по умолчанию определен в *Style*, частью которого является наш шаблон. Если эти свойства не меняются, этими цветами будут белый на черном (для «темной» темы) или черным на белом. Но свойства могут переопределяться локальными параметрами *Button*.

Было бы замечательно, если бы имелся некоторый графический эффект для инвертирования цветов, но такого эффекта нет. Для состояния Pressed нам приходится задавать новые цвета фона и переднего плана анимаций, чтобы создать видимость инверсии цветов. То есть если в качестве переднего плана задан ресурс PhoneForegroundBrush и в качестве фона – ресурс PhoneBackgroundBrush, для состояния Pressed в качестве *Foreground* можно задать PhoneBackgroundBrush и в качестве *Background* – PhoneForegroundBrush.

А можем ли мы использовать *ColorAnimation* для этого? Это было бы возможным, если бы нам точно было известно, что кисти для *Foreground* и *Background* являются объектами *SolidColorBrush*. Но мы этого не знаем. Поэтому приходится использовать объекты *ObjectAnimationUsingKeyFrames* (Анимация свойств типа Object с использованием ключевых кадров) для применения анимаций непосредственно к свойствам *Foreground* и *Background*. Дочерними элементами *ObjectAnimationUsingKeyFrames* могут быть только объекты типа *DiscreteObjectKeyFrame* (Дискретный ключевой кадр типа Object).

Начнем со свойства *Background* и зададим имя объекту *Border*:

```
<Border Name="border"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}"
        Background="{TemplateBinding Background}"
        CornerRadius="12">
```

Посредством этого имени анимация может применяться к свойству *Background* этого *Border*:

```
<VisualState x:Name="Pressed">
    <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="border"
                                       Storyboard.TargetProperty="Background">
            <DiscreteObjectKeyFrame KeyTime="0:0:0"
                                    Value="{StaticResource PhoneForegroundBrush}" />
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

Для состояния Pressed анимация меняет свойство *Background* элемента *Border* и задает для него кисть, описанную как ресурс PhoneForegroundBrush. Превосходно!

Теперь добавим такую же анимацию для свойства *Foreground* элемента ... какого элемента? В дереве визуальных элементов этого шаблона нет элемента со свойством *Foreground*!

Было бы просто идеально, если бы *ContentPresenter* имел свойство *Foreground*, но этого свойства у него нет.

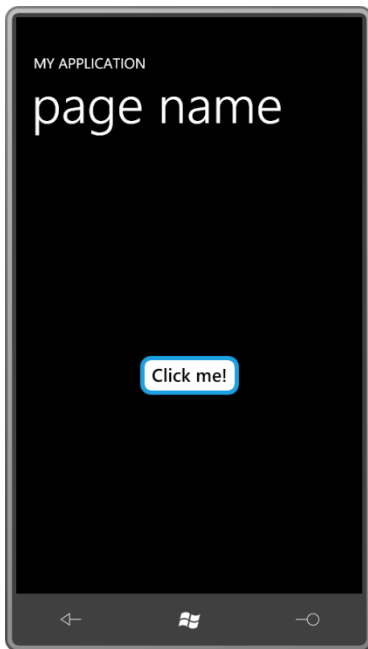
Но, минутку. А что же *ContentControl*? *ContentControl* – это по сути *ContentPresenter*, но у *ContentControl* есть свойство *Foreground*. Поэтому заменим *ContentPresenter* на *ControlControl* и зададим для него имя:

```
<ContentControl Name="contentControl"
                Content="{TemplateBinding Content}"
                ContentTemplate="{TemplateBinding ContentTemplate}"
                Margin="{TemplateBinding Padding}"
                HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
                VerticalAlignment="{TemplateBinding VerticalContentAlignment}" />
```

Теперь мы можем определить вторую анимацию для состояния Pressed:

```
<VisualState x:Name="Pressed">
    <Storyboard>
        …
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="contentControl"
                                       Storyboard.TargetProperty="Foreground">
            <DiscreteObjectKeyFrame KeyTime="0:0:0"
                                    Value="{StaticResource PhoneBackgroundBrush}" />
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

И так наша кнопка выглядит при нажатии:



Теперь я объявляю, что этот шаблон полностью завершен! (И теперь абсолютно ясно, почему шаблон *Button* по умолчанию включает *ContentControl*.)

Рассмотрим полный текст *Style* и *ControlTemplate* в контексте страницы. В приложении CustomButtonTemplate (Пользовательский шаблон кнопки) *Style* описан в коллекции *Resources* страницы. Главным образом чтобы сократить длину строк и вместить их в ширину страницы без переносов, *ControlTemplate* определен как отдельный ресурс, на который затем ссылается *Style*. Привожу *ControlTemplate*, сразу за которым следует *Style*, использующий этот шаблон:

Проект Silverlight: CustomButtonTemplate   Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <ControlTemplate x:Key="buttonTemplate" TargetType="Button">
        <Grid>
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CommonStates">
                    <VisualState x:Name="Normal" />
```

```
                    <VisualState x:Name="MouseOver" />

                    <VisualState x:Name="Pressed">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames
                                        Storyboard.TargetName="border"
                                        Storyboard.TargetProperty="Background">
                                <DiscreteObjectKeyFrame KeyTime="0:0:0"
                                    Value="{StaticResource PhoneForegroundBrush}" />
                            </ObjectAnimationUsingKeyFrames>

                            <ObjectAnimationUsingKeyFrames
                                        Storyboard.TargetName="contentControl"
                                        Storyboard.TargetProperty="Foreground">
                                <DiscreteObjectKeyFrame KeyTime="0:0:0"
                                    Value="{StaticResource PhoneBackgroundBrush}" />
                            </ObjectAnimationUsingKeyFrames>
                        </Storyboard>
                    </VisualState>

                    <VisualState x:Name="Disabled">
                        <Storyboard>
                            <DoubleAnimation Storyboard.TargetName="disableRect"
                                             Storyboard.TargetProperty="Opacity"
                                             To="0.6" Duration="0:0:0" />
                        </Storyboard>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

            <Border Name="border"
                    BorderBrush="{TemplateBinding BorderBrush}"
                    BorderThickness="{TemplateBinding BorderThickness}"
                    Background="{TemplateBinding Background}"
                    CornerRadius="12">

                <ContentControl Name="contentControl"
                                Content="{TemplateBinding Content}"
                                ContentTemplate="{TemplateBinding ContentTemplate}"
                                Margin="{TemplateBinding Padding}"
                                HorizontalAlignment="{TemplateBinding
                                                 HorizontalContentAlignment}"
                                VerticalAlignment="{TemplateBinding
                                                 VerticalContentAlignment}" />
            </Border>

            <Rectangle Name="disableRect"
                       Fill="{StaticResource PhoneBackgroundBrush}"
                       Opacity="0" />
        </Grid>
    </ControlTemplate>

    <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="BorderBrush" Value="{StaticResource PhoneAccentBrush}" />
        <Setter Property="BorderThickness" Value="6" />
        <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}" />
        <Setter Property="Template" Value="{StaticResource buttonTemplate}" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Область содержимого включает *Button*, использующий этот *Style*, конечно. Но я хотел протестировать активацию и деактивацию *Button* интерактивным путем, поэтому добавил на страницу *ToggleButton* и задал для его свойства *IsChecked* привязку к свойству *IsEnabled* объекта *Button*, к которому применяются стиль и шаблон.

Но кажется не вполне правильным включать (т.е. выделять цветом) *ToggleButton*, когда *Button* находится в своем обычном состоянии (т.е. активный). Я хотел, чтобы *ToggleButton* выводил «Button Enabled» (Кнопка активна), когда *ToggleButton* включен и *Button* активен, и «Button Disabled» (Кнопка неактивна), когда *ToggleButton* выключен и *Button* неактивен.

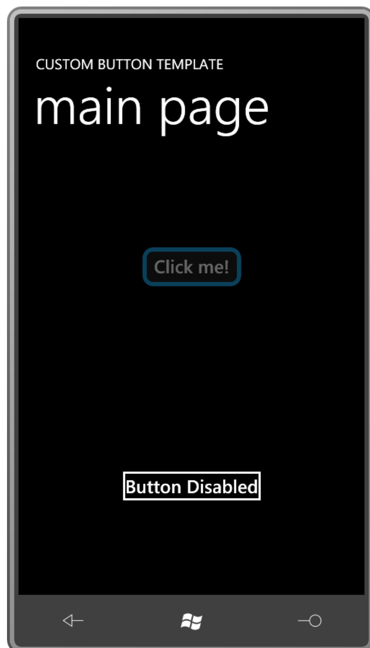
В этом прелесть шаблонов: вы можете сделать все это прямо в XAML без лишнего шума и дополнительных инструментов, таких как Expression Blend.

Проект Silverlight: CustomButtonTemplate   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Button Grid.Row="0"
            Content="Click me!"
            Style="{StaticResource buttonStyle}"
            IsEnabled="{Binding ElementName=toggleButton, Path=IsChecked}"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />

    <ToggleButton Name="toggleButton"
                  Grid.Row="1"
                  IsChecked="true"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center">
        <ToggleButton.Template>
            <ControlTemplate TargetType="ToggleButton">
                <Border BorderBrush="{StaticResource PhoneForegroundBrush}"
                        BorderThickness="{StaticResource PhoneBorderThickness}">

                    <VisualStateManager.VisualStateGroups>
                        <VisualStateGroup x:Name="CheckStates">
                            <VisualState x:Name="Checked">
                                <Storyboard>
                                    <ObjectAnimationUsingKeyFrames
                                            Storyboard.TargetName="txtblk"
                                            Storyboard.TargetProperty="Text">
                                        <DiscreteObjectKeyFrame KeyTime="0:0:0"
                                                                Value="Button Enabled" />
                                    </ObjectAnimationUsingKeyFrames>
                                </Storyboard>
                            </VisualState>

                            <VisualState x:Name="Unchecked" />

                        </VisualStateGroup>
                    </VisualStateManager.VisualStateGroups>

                    <TextBlock Name="txtblk"
                               Text="Button Disabled"/>
                </Border>
            </ControlTemplate>
       </ToggleButton.Template>
    </ToggleButton>
</Grid>
```

Данный *ToggleButton* имеет, что называется, специальный шаблон узкого назначения *ControlTemplate*, поэтому в нем нет никаких излишеств. Дерево визуальных элементов включает лишь *Border* и *TextBlock*. Свойство *Content* проигнорировано, и свойство *Text* объекта *TextBlock* инициализируется значением «Button Disabled». Все остальное делают

визуальные состояния. Кроме обычных визуальных состояний *Button*, *ToggleButton* также определяет группу CheckStates (Состояния переключателя), включающую состояния Checked (Установлен) и Unchecked (Снят). Данный шаблон обрабатывает только эти два состояния, и анимация состояния Checked задает свойству *Text* объекта *TextBlock* значение «Button Enabled». Вот как это выглядит при неактивном *Button*:



Я не описывал состояние Disabled для *ToggleButton*, потому что этот шаблон предполагается использовать только для данного приложения, и я знаю, что *ToggleButton* никогда не будет неактивным.

Совместное и повторное использование стилей шаблонов

Как известно, можно наследовать один *Style* от другого, при этом наследуются все объекты *Setter*. Новый *Style* может добавлять новые *Setter* или переопределять имеющиеся.

А вот от *ControlTemplate* наследоваться нельзя. Нельзя сослаться на существующий *ControlTemplate* и заменить часть его дерева визуальных элементов или задать что-то дополнительно. (Довольно сложно представить механизм или синтаксис такого процесса.)

Как правило, если требуется внести какие-либо изменения в существующий *ControlTemplate*, необходимо получить копию всего шаблона и редактировать ее. Шаблоны по умолчанию обычно включены в документацию Silverlight. (Но как я уже упоминал в этой главе, документация Silverlight включает только шаблоны для Веб-версии Silverlight, в ней нет шаблонов для Silverlight for Windows Phone.) Expression Blend также имеет доступ к стандартным шаблонам по умолчанию.

Если требуется использовать совместно *Style* или *ControlTemplate* (или *Style*, включающий *ControlTemplate*) для нескольких элементов страницы, просто поместите его в коллекцию *Resources* этой страницы. Если требуется использовать *Style* или *ControlTemplate* для нескольких страниц, поместите его в коллекцию *Resources* файла App.xaml.

Также ресурсы могут использоваться совместно несколькими приложениями. Для этого ресурсы описываются в XAML-файле, корневым элементом которого является *ResourceDictionary*. Рассмотрим такой файл, который называется SharedResources.xaml:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <SolidColorBrush x:Key="brush" Color="Blue" />

    …

</ResourceDictionary>
```

Файл может иметь не только *SolidColorBrush*, а намного больше ресурсов. У каждого ресурса, конечно же, должен быть атрибут *x:Key*. Этот файл можно создавать как часть проекта или добавлять в проект уже существующий файл. В любом случае для Build Action в окне свойств должно быть задано Page.

Теперь мы можем сослаться на этот файл в коллекции *Resources* в App.xaml:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="SharedResources.xaml" />

                …

        </ResourceDictionary.MergedDictionaries>

        …

    </ResourceDictionary>
</Application.Resources>
```

Обратите внимание, что свойство-элемент *ResourceDictionary.MergedDictionary* (Объединенный словарь) используется для ссылки на внешние объекты *ResourceDictionary*.

Рассмотрим другой подход. Предлагаю определить пользовательский стиль и шаблон для существующего элемента управления и затем вместо того, чтобы ссылаться на стиль, ссылаться на этот элемент управления, используя новое имя. Такое тоже возможно.

Рассмотрим приложение FlipToggleDemo (Демонстрация переключателя с переворачиванием), включающее пользовательский класс *FlipToggleButton* (Переключатель с переворачиванием), наследуемый от *ToggleButton*. Но *FlipToggleButton* не вносит никакого нового кода в *ToggleButton*, добавляются только *Style* и *ControlTemplate*.

В проекте FlipToggleDemo я добавил новый элемент типа Windows Phone User Control (Пользовательский элемент управления для Windows Phone) и назвал его FlipToggleButton.xaml. В результате этого были созданы файл FlipToggleButton.xaml и файл FlipToggleButton.xaml.cs для класса, который наследуется от *UserControl*. После этого в обоих файлах я изменил *UserControl* на *ToggleButton*, так чтобы *FlipToggleButton* наследовался от *ToggleButton*.

Чтобы не усложнять, я решил не реализовывать никакие смены состояний для неактивной кнопки, но переворачивать ее для состояния Unchecked. Рассмотрим полный XAML-файл для пользовательской кнопки. (Отступы уменьшены на два пробела, чтобы строки не превышали ширины страницы книги.):

Проект Silverlight: FlipToggleDemo Файл: FlipToggleButton.xaml

```
<ToggleButton x:Class="FlipToggleDemo.FlipToggleButton"
              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ToggleButton.Style>
    <Style TargetType="ToggleButton">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="ToggleButton">
            <Border BorderBrush="{StaticResource PhoneForegroundBrush}"
                    BorderThickness="{StaticResource PhoneBorderThickness}"
                    Background="{TemplateBinding Background}"
                    RenderTransformOrigin="0.5 0.5">
              <Border.RenderTransform>
                <RotateTransform x:Name="rotate" />
              </Border.RenderTransform>

              <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CheckStates">
                  <VisualState x:Name="Checked">
                    <Storyboard>
                      <DoubleAnimation Storyboard.TargetName="rotate"
                                       Storyboard.TargetProperty="Angle"
                                       To="180" Duration="0:0:0.5" />
                    </Storyboard>
                  </VisualState>

                  <VisualState x:Name="Unchecked">
                    <Storyboard>
                      <DoubleAnimation Storyboard.TargetName="rotate"
                                       Storyboard.TargetProperty="Angle"
                                       Duration="0:0:0.5" />
                    </Storyboard>
                  </VisualState>
                </VisualStateGroup>
              </VisualStateManager.VisualStateGroups>

              <ContentPresenter Content="{TemplateBinding Content}"
                                ContentTemplate="{TemplateBinding ContentTemplate}"
                                Margin="{TemplateBinding Padding}"
                                HorizontalAlignment="{TemplateBinding
                                             HorizontalContentAlignment}"
                                VerticalAlignment="{TemplateBinding
                                             VerticalContentAlignment}" />
            </Border>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </ToggleButton.Style>
</ToggleButton>
```

Обычно основную часть XAML-файла занимает свойство *Content* корневого элемента. В данном случае это свойство *Style* корневого элемента. Обратите внимание, что и для объекта *Style*, и для объекта *ControlTemplate* в качестве значения свойства *TargetType* задан *ToggleButton*, а не *FlipToggleButton*. В этом нет никакой ошибки, потому что ни один из них не использует никаких свойств, специально определенных *FlipToggleButton*, потому что *FlipToggleButton* не определяет никаких новых свойств *FlipToggleButton*.

Сам шаблон довольно прост и включает только *Border* и *ContentPresenter* со всеми привязками, определенными в стандартном шаблоне. Но *Border* также описывает свойство *RenderTransformOrigin*, и значением его свойства *RenderTransform* задан объект *RotateTransform*.

Для двух анимаций, обеспечивающих переворот кнопки (для состояния Checked) и возвращение ее в нормальное положение (для состояния Unchecked), задана отличная от нуля продолжительность. Для *DoubleAnimation*, реализующего состояние Checked, не задано значение *From*; используется базовое значение этого свойства, которое равно нулю. Для *DoubleAnimation*, реализующего состояние Unchecked, не задано ни *To*, ни *From*! Анимация начинается со значения свойства *Angle* объекта *RotateTransform* – вероятнее всего это будет 180 градусов, но может быть и меньше, если предыдущая анимация не была полностью завершена к моменту перехода кнопки в состояние Unchecked – и завершается в базовом значении, которым является нуль.

Рассмотрим файл выделенного кода для пользовательского элемента управления полностью:

Проект Silverlight: FlipToggleDemo Файл: FlipToggleButton.xaml.cs

```
using System.Windows.Controls.Primitives;

namespace FlipToggleDemo
{
    public partial class FlipToggleButton : ToggleButton
    {
        public FlipToggleButton()
        {
            InitializeComponent();
        }
    }
}
```

Экземпляр пользовательской кнопки создается в файле MainPage.xaml проекта:

Проект Silverlight: FlipToggleDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <local:FlipToggleButton Content="Flip Toggle"
                            HorizontalAlignment="Center"
                            VerticalAlignment="Center" />
</Grid>
```

Библиотека пользовательских элементов управления

Как правило, при создании пользовательского элемента управления для него описываются некоторые новые свойства, а также *Style* и *ControlTemplate* по умолчанию, и этот новый элемент управления помещается в DLL для совместного использования множеством приложений. Можно объединять код элемента управления и *Style*, как было показано в примере *FlipToggleButton*, но более стандартным подходом для библиотек Silverlight является описание *Style* в специальном файле generic.xaml, который располагается в папке Themes. Корневым элементом этого файла generic.xaml является *ResourceDictionary*.

Рассмотрим пример.

Предположим, мы хотим создать шаблон *ToggleButton*, подобный реализованному в проекте CustomButtonTemplate, но сделать его более обобщенным. Вместо того чтобы переключаться между двумя текстовыми строками, заданными в коде, мы хотим переключаться между двумя объектами любого типа. И не просто переключаться, мы хотим, чтобы объект, ассоциированный с состоянием Checked, и объект, ассоциированный с состоянием Unchecked, плавно перетекали друг в друга. Назовем эту новую кнопку *FadableToggleButton* (Переключатель с плавным переходом).

Проанализировав все это, мы понимаем, что элементу управления необходимо новое свойство *CheckedContent* (Содержимое в состоянии Checked), подобное обычному свойству *Content*. Свойство *Content* – это объект, отображаемый, когда кнопка находится в состоянии Unchecked, а *CheckedContent* – объект, отображаемый для состояния Checked.

Я описал этот класс в библиотеке Petzold.Phone.Silverlight. Привожу здесь код *FadableToggleButton* полностью:

Проект Silverlight: Petzold.Phone.Silverlight Файл: FadableToggleButton.cs

```
using System.Windows;
using System.Windows.Controls.Primitives;

namespace Petzold.Phone.Silverlight
{
    public class FadableToggleButton : ToggleButton
    {
        public static readonly DependencyProperty CheckedContentProperty =
            DependencyProperty.Register("CheckedContent",
            typeof(object),
            typeof(FadableToggleButton),
            new PropertyMetadata(null));

        public FadableToggleButton()
        {
            this.DefaultStyleKey = typeof(FadableToggleButton);
        }

        public object CheckedContent
        {
            set { SetValue(CheckedContentProperty, value); }
            get { return (object)GetValue(CheckedContentProperty); }
        }
    }
}
```

И это весь код на C#, необходимый для реализации данного элемента управления! В нем даже нет обработчика события изменения значения для нового свойства *CheckedContent*. Только определение *DependencyProperty* и свойства CLR. Все остальное – это XAML.

Но обратите внимание на конструктор. Если бы этот файл кода был *частичным* описанием класса, которое сопровождается XAML-файлом, мы бы увидели в конструкторе вызов метода *InitializeComponent*. Но вместо этого мы обнаруживаем там следующее:

```
this.DefaultStyleKey = typeof(FadableToggleButton);
```

Данное выражение указывает на то, что этот класс имеет определение *Style* по умолчанию, и значением *TargetType* в этом *Style* является *FadableToggleButton*. Чтобы применить *Style* по умолчанию к экземплярам класса, Silverlight должен найти описание этого *Style*. Где его искать?

Silverlight ведет поиск в особом XAML-файле библиотеки. Этот XAML-файл всегда называется generic.xaml и всегда располагается в папке Themes проекта DLL. Так элемент управления получает стиль темы и шаблон по умолчанию.

Корневым элементом файла generic.xaml file является *ResourceDictionary*. Но этот файл является особым по другой причине: его содержимое рассматривается как ресурсы, но элементы *Style* не требуют атрибутов *x:Key* или *x:Name*, потому что ссылка на них осуществляется через *TargetType*.

Рассмотрим часть файла generic.xaml из папки Themes библиотеки Petzold.Phone.Silverlight, которая включает описание *Style* по умолчанию класса *FadableToggleButton*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: Themes/generic.xaml (фрагмент)

```
<ResourceDictionary
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Petzold.Phone.Silverlight">

  <Style TargetType="local:FadableToggleButton">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="local:FadableToggleButton">
          <Grid>
            <VisualStateManager.VisualStateGroups>
              <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal" />
                <VisualState x:Name="MouseOver" />
                <VisualState x:Name="Pressed" />

                <VisualState x:Name="Disabled">
                  <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="disableRect"
                                     Storyboard.TargetProperty="Opacity"
                                     To="0.6" Duration="0:0:0" />
                  </Storyboard>
                </VisualState>
              </VisualStateGroup>

              <VisualStateGroup x:Name="CheckStates">
                <VisualState x:Name="Checked">
                  <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="uncheckedContent"
                                     Storyboard.TargetProperty="Opacity"
                                     To="0" Duration="0:0:0.5" />
                    <DoubleAnimation Storyboard.TargetName="checkedContent"
                                     Storyboard.TargetProperty="Opacity"
                                     To="1" Duration="0:0:0.5" />
                  </Storyboard>
                </VisualState>

                <VisualState x:Name="Unchecked">
                  <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="uncheckedContent"
                                     Storyboard.TargetProperty="Opacity"
                                     Duration="0:0:0.5" />
                    <DoubleAnimation Storyboard.TargetName="checkedContent"
                                     Storyboard.TargetProperty="Opacity"
                                     Duration="0:0:0.5" />
                  </Storyboard>
                </VisualState>
              </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

            <Border BorderBrush="{StaticResource PhoneForegroundBrush}"
                    BorderThickness="{StaticResource PhoneBorderThickness}"
                    Background="{TemplateBinding Background}">

              <Grid Margin="{TemplateBinding Padding}">
                <ContentPresenter
                        Name="uncheckedContent"
                        Content="{TemplateBinding Content}"
                        ContentTemplate="{TemplateBinding ContentTemplate}"
                        HorizontalAlignment="{TemplateBinding
                                                  HorizontalContentAlignment}"
                        VerticalAlignment="{TemplateBinding
```

```
                                                    VerticalContentAlignment}" />

                <ContentPresenter
                        Name="checkedContent"
                        Opacity="0"
                        Content="{TemplateBinding CheckedContent}"
                        ContentTemplate="{TemplateBinding ContentTemplate}"
                        HorizontalAlignment="{TemplateBinding
                                                    HorizontalContentAlignment}"
                        VerticalAlignment="{TemplateBinding
                                                    VerticalContentAlignment}" />
              </Grid>
            </Border>

            <Rectangle Name="disableRect"
                       Fill="{StaticResource PhoneBackgroundBrush}"
                       Opacity="0" />
          </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>

  …

</ResourceDictionary>
```

Значением *TargetType* для *Style* является *FadableToggleButton*, и этого достаточно, чтобы Silverlight мог найти определение *Style*, который становится темой по умолчанию для *FadableToggleButton*. В *Border* располагается *Grid* с одной ячейкой. *Grid* включает два элемента *ContentPresenter*: свойство *TemplateBinding* одного из них ссылается на обычное свойство *Content*, и другого – на свойство *CheckedContent*. Свойству *Opacity* объекта *ContentPresenter*, ссылающего на свойство *CheckedContent*, задано начальное значение нуль. Целью анимаций для состояний Checked и Unchecked является свойство *Opacity* объекта *ContentPresenter*, что обеспечивает постепенное появление одного объекта и исчезновение другого.

Несмотря на то что свойства *Content* этих двух элементов *ContentPresenter* связаны с двумя разными свойствами *FadableToggleButton*, свойства *ContentTemplate* обоих связаны с тем же свойством *ContentTemplate*, которое изначально определено *ContentControl*. Если значением *DataTemplate* свойства *ContentTemplate* задать *FadableToggleButton*, тот же *DataTemplate* должен применяться к обоим свойствам *Content* и свойству *CheckedContent*. Иначе говоря, этот шаблон неявно предполагает, что свойства *Content* и *CheckedContent* одного типа.

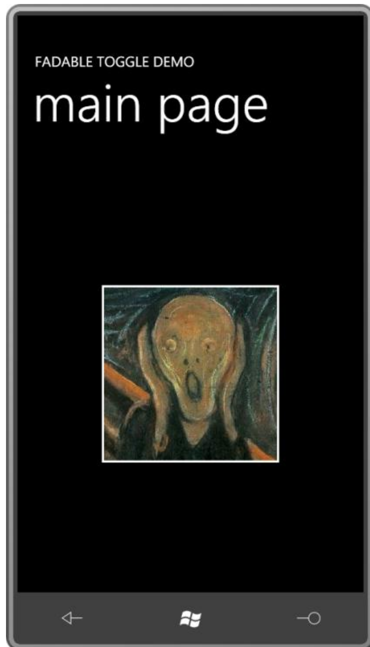
Чтобы протестировать этот новый элемент управления, я создал приложение FadableToggleDemo (Демонстрация переключателя с плавным переходом). Этот проект включает ссылку на библиотеку Petzold.Phone.Silverlight и объявление пространства имен XML для библиотеки в MainPage.xaml. В папку *Images* этого проекта я добавил два растровых изображения одного размера. Эти изображения используются элементами *Image*, заданными как значения свойств *Content* и *CheckedContent* кнопки:

Проект Silverlight: FadableToggleDemo Файл: MainPage.xaml (фрагмент)

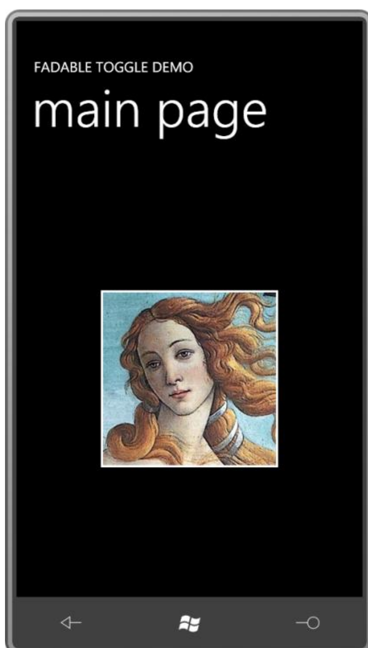
```
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

            <petzold:FadableToggleButton HorizontalAlignment="Center"
                                         VerticalAlignment="Center">

                <petzold:FadableToggleButton.Content>
                    <Image Source="Images/MunchScream.jpg"
```

```
                        Stretch="None" />
                </petzold:FadableToggleButton.Content>

                <petzold:FadableToggleButton.CheckedContent>
                    <Image Source="Images/BotticelliVenus.jpg"
                           Stretch="None" />
                </petzold:FadableToggleButton.CheckedContent>

            </petzold:FadableToggleButton>

        </Grid>
```

Для свойства *Content* property используется картинаЭдварда Мунка *Крик*:



Для свойства *CheckedContent* используется фрагмент картины Боттичелли *Рождение Венеры*:



Вариации на тему *Slider*

Как можно ожидать, один из самых сложных стандартных шаблонов Silverlight – это шаблон для *Slider*. Поэтому важно изучить его, особенно если вам не очень нравится шаблон *Slider* по умолчанию, реализованный в Windows Phone 7.

На первый взгляд *Slider*, кажется, не подходит под схему шаблонов, главным образом потому что он включает движущиеся части. Как же именно он реализуется?

Если посмотреть на документацию *Slider*, мы найдем в ней привычные теги *TemplateVisualStateAttribute*, а также коллекцию тегов *TemplatePartAttribute* (Атрибут части шаблона) (здесь я привожу их немного в другом порядке, чем они представлены в документации):

```
[TemplateVisualStateAttribute(Name = "Normal", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "MouseOver", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "Disabled", GroupName = "CommonStates")]
[TemplateVisualStateAttribute(Name = "Focused", GroupName = "FocusStates")]
[TemplateVisualStateAttribute(Name = "Unfocused", GroupName = "FocusStates")]
[TemplatePartAttribute(Name = "HorizontalTemplate", Type = typeof(FrameworkElement))]
[TemplatePartAttribute(Name = "HorizontalTrackLargeChangeDecreaseRepeatButton",
                       Type = typeof(RepeatButton))]
[TemplatePartAttribute(Name = "HorizontalTrackLargeChangeIncreaseRepeatButton",
                       Type = typeof(RepeatButton))]
[TemplatePartAttribute(Name = "HorizontalThumb", Type = typeof(Thumb))]
[TemplatePartAttribute(Name = "VerticalTemplate", Type = typeof(FrameworkElement))]
[TemplatePartAttribute(Name = "VerticalTrackLargeChangeDecreaseRepeatButton",
                       Type = typeof(RepeatButton))]
[TemplatePartAttribute(Name = "VerticalTrackLargeChangeIncreaseRepeatButton",
                       Type = typeof(RepeatButton))]
[TemplatePartAttribute(Name = "VerticalThumb", Type = typeof(Thumb))]
public class Slider : RangeBase
```

Это означает, что шаблон для *Slider* должен включать восемь элементов с именами «HorizontalTemplate» (Горизонтальный шаблон) и т.д. Эти элементы называют «частями» шаблона. Части «HorizontalTemplate» и «VerticalTemplate» (Вертикальный шаблон) могут быть только типа *FrameworkElement* (или производным от *FrameworkElement*), но остальные части должны быть типа *RepeatButton* или *Thumb* (Бегунок).

RepeatButton и *Thumb* – это пара элементов управления, использовать которые в данной книге до сих пор не было повода. (Оба этих класса располагаются в пространстве имен *System.Windows.Controls.Primitives* – тонкий намек на то, что данные элементы управления предполагается использовать для построения других элементов управления.) *RepeatButton* аналогичен обычному *Button*, за исключением того что при удержании пальца на нем он формирует повторяющиеся события *Click*. Он идеально подходит для *ScrollBar* или *Slider* и, вероятно, создавался именно для этих целей.

Thumb – довольно специализированный элемент управления, который сообщает о том, каким образом пользователь перемещает его. Но *Thumb* сложно найти в стандартном *Slider* в Windows Phone 7, потому что он довольно хорошо спрятан в шаблоне темы. Одна из моих задач здесь – вернуть *Thumb* в *Slider*.

Элемент управления с частями (такой как *Slider*) перегружает метод *OnApplyTemplate* (При применении шаблона), чтобы получать уведомления при задании шаблона его свойству *Template*. После этого с помощью метода *GetTemplateChild* (Получить дочерний элемент шаблона) он находит элементы с заданными именами. Он может задавать обработчики событий для этих элементов и манипулировать ими в ходе работы. (Реализация этого процесса в коде будет показана ближе к концу данной главы.)

Стандартный *Slider* поддерживает горизонтальную и вертикальную ориентации, и шаблон включает два отдельных (и довольно независимых) шаблона для обеспечения этих ориентаций. Эти два шаблона представлены элементами «HorizontalTemplate» и «VerticalTemplate». Если свойство *Orientation* объекта *Slider* имеет значение *Horizontal*, *Slider* задает свойству *Visibility* элемента «HorizontalTemplate» значение *Visible* и свойству *Visibility* элемента «VerticalTemplate» значение *Collapsed*; и противоположные значения для вертикальной ориентации.

Самый простой подход при проектировании нового шаблона для *Slider* – использовать *Grid* с одной ячейкой, который будет включать оба этих шаблона. Вложенный *Grid* под именем «HorizontalTemplate» имеет три столбца с двумя элементами управления *RepeatButton* и одним *Thumb*. У другого вложенного *Grid* под именем «VerticalTemplate» – три строки.

Рассмотри то, что я называю «скелетом» шаблона для *Slider*, который определен как ресурс:

Проект Silverlight: BareBonesSlider Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <ControlTemplate x:Key="bareBonesSliderTemplate"
                     TargetType="Slider">
        <Grid>
            <Grid Name="HorizontalTemplate">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>

                <RepeatButton Name="HorizontalTrackLargeChangeDecreaseRepeatButton"
                              Grid.Column="0"
                              Content="-" />

                <Thumb Name="HorizontalThumb"
                       Grid.Column="1" />

                <RepeatButton Name="HorizontalTrackLargeChangeIncreaseRepeatButton"
                              Grid.Column="2"
                              Content="+" />
            </Grid>

            <Grid Name="VerticalTemplate">
                <Grid.RowDefinitions>
                    <RowDefinition Height="*" />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>

                <RepeatButton Name="VerticalTrackLargeChangeDecreaseRepeatButton"
                              Grid.Row="0"
                              Content="-" />

                <Thumb Name="VerticalThumb"
                       Grid.Row="1" />

                <RepeatButton Name="VerticalTrackLargeChangeIncreaseRepeatButton"
                              Grid.Row="2"
                              Content="+" />
            </Grid>
        </Grid>
    </ControlTemplate>
</phone:PhoneApplicationPage.Resources>
```

Вовсе не обязательно, чтобы в шаблоне *Slider* элементы управления *RepeatButton* и *Thumb* располагались в *Grid* с тремя строками или тремя столбцами, но, безусловно, это самое простое решение. Обратите внимание, я присвоил свойствам *Content* элементов управления *RepeatButton* знаки плюс и минус в зависимости от их роли, что выглядит довольно необычным.

Сосредоточимся на ориентации *Horizontal*. *RepeatButton* для уменьшения значений располагается в первой ячейке *Grid* с шириной *Auto*, *Thumb* помещен во вторую ячейку *Grid*, для *Width* которой также задано значение *Auto*. Сам *Thumb* имеет фиксированную ширину, а вот ширина уменьшающегося *RepeatButton* меняется напрямую логикой *Slider* для отражения значения его свойства *Value*. Когда *Value* получает значение *Minimum*, *RepeatButton* достигает нулевой ширины. Если *Value* задано значение *Maximum*, ширина *RepeatButton* соответствует разности между шириной всего элемента управления и шириной *Thumb*.

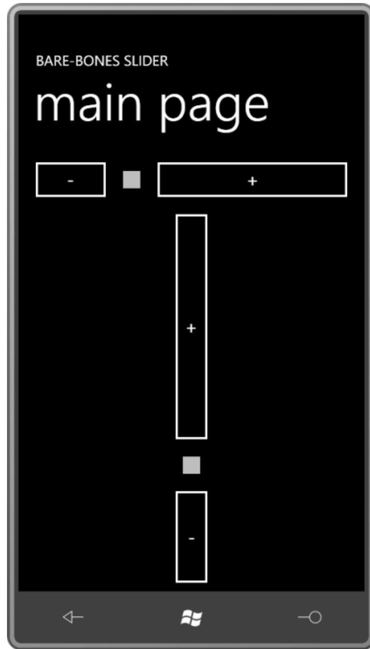
Slider меняет значение свойства *Value* (и, следовательно, относительный размер двух элементов управления *RepeatButton*), когда пользователь касается *RepeatButton* или физически перемещает *Thumb*. (Более подробно элемент управления *Thumb* мы рассмотрим несколько позже.)

Далее в проекте BareBonesSlider (Простой ползунок) в области содержимого создаются два элемента управления *Slider* и применяется шаблон:

Проект Silverlight: BareBonesSlider Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Slider Grid.Row="0"
            Orientation="Horizontal"
            Template="{StaticResource bareBonesSliderTemplate}" />

    <Slider Grid.Row="1"
            Orientation="Vertical"
            Template="{StaticResource bareBonesSliderTemplate}"
            HorizontalAlignment="Center" />
</Grid>
```

Вот как выглядят все эти элементы управления при небольшом смещении ползунка относительно его исходного положения:



RepeatButton выглядит, как обычный *Button*, и *Thumb* – как прямоугольник, окруженный прозрачной областью.

Теперь, когда мы знаем, как создать пользовательский шаблон для *Slider*, можем ли мы придать ему немного более привлекательный вид? Да, и ключом к реализации этого является осознание того, что *RepeatButton* и *Thumb* наследуются от *Control*, т.е. у обоих классов есть свойство *Template*, и в рамках шаблона *Slider* мы можем описывать новые шаблоны для *RepeatButton* и *Thumb* специально для использования в шаблоне *Slider*.

Рассмотрим более замысловатый *Slider*, который также включает привязки шаблонов для свойств *Background* и *Foreground*. Для этих свойств в *Style* заданы значения по умолчанию. Также этот *Style* включает *ControlTemplate*. Здесь я привожу только внешний *Grid* объекта *ControlTemplate*, у которого есть собственный раздел *Resources* для описания очень простого *ControlTemplate* для *RepeatButton* и довольно развернутых шаблонов для горизонтального и вертикального *Thumb*:

Проект Silverlight: AlternativeSlider   Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="alternativeSliderStyle" TargetType="Slider">
        <Setter Property="Background"
                Value="{StaticResource PhoneBackgroundBrush}" />
        <Setter Property="Foreground"
                Value="{StaticResource PhoneForegroundBrush}" />
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="Slider">
                    <Grid Background="{TemplateBinding Background}">

                        <Grid.Resources>
                            <ControlTemplate x:Key="repeatButtonTemplate"
                                             TargetType="RepeatButton">
                                <Rectangle Fill="Transparent" />
                            </ControlTemplate>

                            <Style x:Key="horizontalThumbStyle"
                                   TargetType="Thumb">
                                <Setter Property="Width" Value="72" />
                                <Setter Property="Height" Value="72" />
```

```
                            <Setter Property="Template">
                                <Setter.Value>
                                    <ControlTemplate TargetType="Thumb">
                                        <Border Background="Transparent">
                                            <Rectangle Margin="18 0"
                                                       RadiusX="6"
                                                       RadiusY="6"
                                                       Stroke="{StaticResource
                                                            PhoneAccentBrush}"
                                                       Fill="{TemplateBinding
                                                          Foreground}" />
                                        </Border>
                                    </ControlTemplate>
                                </Setter.Value>
                            </Setter>
                        </Style>

                        <Style x:Key="verticalThumbStyle"
                               TargetType="Thumb">
                            <Setter Property="Width" Value="72" />
                            <Setter Property="Height" Value="72" />
                            <Setter Property="Template">
                                <Setter.Value>
                                    <ControlTemplate TargetType="Thumb">
                                        <Border Background="Transparent">
                                            <Rectangle Margin="0 18"
                                                       RadiusX="6"
                                                       RadiusY="6"
                                                       Stroke="{StaticResource
                                                            PhoneAccentBrush}"
                                                       Fill="{TemplateBinding
                                                              Foreground}" />
                                        </Border>
                                    </ControlTemplate>
                                </Setter.Value>
                            </Setter>
                        </Style>
                    </Grid.Resources>

                    …

                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</phone:PhoneApplicationPage.Resources>
```

Шаблон *RepeatButton* – это всего лишь прозрачный *Rectangle*. (*Rectangle* должен иметь прозрачный, а не нулевой *Fill*, тогда он сможет принимать сенсорный ввод.) Однако для стилей *Thumb* мне пришлось переопределить свойства *Width* и *Height*. В стиле темы им задано значение 48, что мне показалось слишком большим. Я задал *Border* с прозрачным фоном для создания большей мишени касания, но видимую часть сделал несколько поменьше, чтобы она больше была похожей на обычный бегунок *Slider*.

Описания обоих элементов *Grid* для горизонтальной и вертикальной ориентации начинаются с *Rectangle*, который обеспечивает своего рода визуальную обратную связь. Каждый *RepeatButton* и *Thumb* используют *ControlTemplate* либо *Style*, описанный для этого элемента управления ранее:

Проект Silverlight: AlternativeSlider Файл: MainPage.xaml (фрагмент)

```
<Grid Name="HorizontalTemplate">
    <Grid.ColumnDefinitions>
```

```
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Rectangle Grid.Column="0" Grid.ColumnSpan="3"
               Height="8"
               Margin="12 0"
               Stroke="{TemplateBinding Foreground}"
               Fill="{StaticResource PhoneAccentBrush}" />

    <RepeatButton Name="HorizontalTrackLargeChangeDecreaseRepeatButton"
                  Grid.Column="0"
                  Template="{StaticResource repeatButtonTemplate}" />

    <Thumb Name="HorizontalThumb"
           Grid.Column="1"
           Style="{StaticResource horizontalThumbStyle}" />

    <RepeatButton Name="HorizontalTrackLargeChangeIncreaseRepeatButton"
                  Grid.Column="2"
                  Template="{StaticResource repeatButtonTemplate}" />
</Grid>

<Grid Name="VerticalTemplate">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Rectangle Grid.Row="0" Grid.RowSpan="3"
               Width="8"
               Margin="0 12"
               Stroke="{TemplateBinding Foreground}"
               Fill="{StaticResource PhoneAccentBrush}" />

    <RepeatButton Name="VerticalTrackLargeChangeDecreaseRepeatButton"
                  Grid.Row="0"
                  Template="{StaticResource repeatButtonTemplate}" />

    <Thumb Name="VerticalThumb"
           Grid.Row="1"
           Style="{StaticResource verticalThumbStyle}" />

    <RepeatButton Name="VerticalTrackLargeChangeIncreaseRepeatButton"
                  Grid.Row="2"
                  Template="{StaticResource repeatButtonTemplate}" />
</Grid>
```

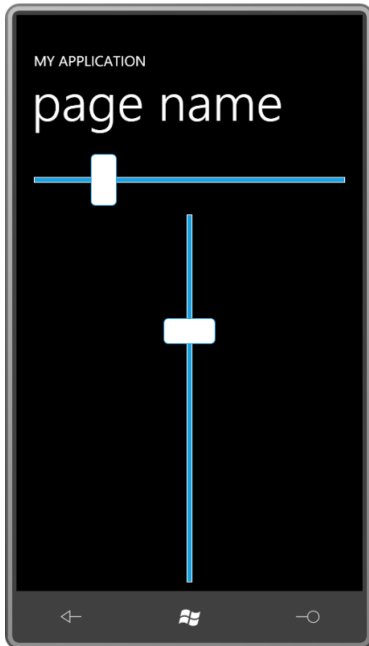
Область содержимого в данном приложении очень похожа на предыдущее, только элементы управления *Slider* используют этот новый стиль:

Проект Silverlight: AlternativeSlider Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Slider Grid.Row="0"
            Orientation="Horizontal"
            Style="{StaticResource alternativeSliderStyle}" />

    <Slider Grid.Row="1"
```

```
            Orientation="Vertical"
            Style="{StaticResource alternativeSliderStyle}"
            HorizontalAlignment="Center" />
</Grid>
```

И вот что мы получаем:



Я решил перенести эти *Style* с *ControlTemplate* в библиотеку Petzold.Phone.Silverlight как *Style* по умолчанию для класса *AltSlider*. У этого класса нет дополнительных свойств, поэтому в файле кода необходимо лишь указать класс, от которого наследуется *AltSlider*, и класс, используемый для обнаружения местонахождения *Style* по умолчанию:

Проект Silverlight: Petzold.Phone.Silverlight Файл: AltSlider.cs

```
using System.Windows.Controls;

namespace Petzold.Phone.Silverlight
{
    public class AltSlider : Slider
    {
        public AltSlider()
        {
            this.DefaultStyleKey = typeof(AltSlider);
        }
    }
}
```

Этот *Style* по умолчанию (включающий *ControlTemplate*) описывается в файле generic.xaml. Я не буду приводить здесь этот файл полностью, потому что в нем преимущественно повторяется определение *Style* из проекта AlternativeSlider (Альтернативный слайдер):

Проект Silverlight: Petzold.Phone.Silverlight Файл: Themes/generic.xaml (фрагмент)

```
<ResourceDictionary
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Petzold.Phone.Silverlight">
```

```
    …

    <Style TargetType="local:AltSlider">
        <Setter Property="Background"
                Value="{StaticResource PhoneBackgroundBrush}" />
        <Setter Property="Foreground"
                Value="{StaticResource PhoneForegroundBrush}" />
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="local:AltSlider">
                    …
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

    …

</ResourceDictionary>
```

Безусловно, теперь необходимо выполнить тестирование. В проекте AltSliderDemo имеется ссылка на проект Petzold.Phone.Silverlight и объявление пространства имен XML для него. Область содержимого такая же, как и в двух предыдущих приложениях:

Проект Silverlight: AltSliderDemo Файл: MainPage.xaml (фрагмент)

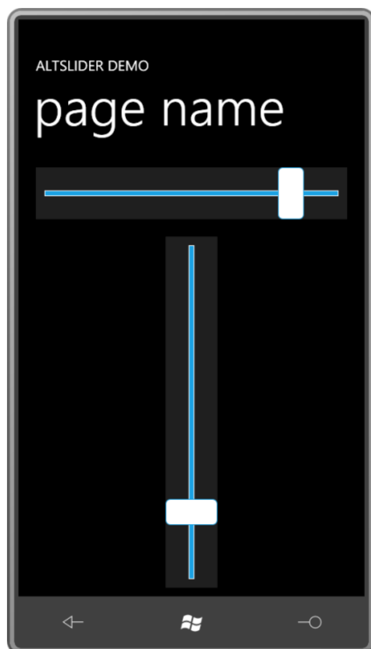
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <petzold:AltSlider Grid.Row="0"
                       Orientation="Horizontal"
                       Style="{StaticResource altSliderStyle}"/>

    <petzold:AltSlider Grid.Row="1"
                       Orientation="Vertical"
                       HorizontalAlignment="Center"
                       Style="{StaticResource altSliderStyle}" />
</Grid>
```

Как видим, свойствам *Style* этих элементов управления *AltSlider* заданы некоторые значения. Что это? Это ссылки на *Style*, определенный в коллекции *Resources* страницы:

Проект Silverlight: AltSliderDemo Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="altSliderStyle"
           TargetType="petzold:AltSlider">
        <Setter Property="Margin" Value="12" />
        <Setter Property="Background" Value="{StaticResource PhoneChromeBrush}" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Единственным назначением этого *Style* является демонстрация того, что определение *Style* по умолчанию для класса *AltSlider* не устраняет возможности задания впоследствии другого *Style*, даже такого, который будет переопределять одно из свойств исходного *Style*. Вот что мы видим на экране при выполнении данного приложения:



Такой необходимый *Thumb*

Уже несколько раз в данной книге у меня возникало желание применить элемент управления *Thumb*. В последний раз это было в проекте SplineKeyFrameExperiment из предыдущей главы. Две главы ранее я даже создал элемент управления *PointDragger* в приложении CubicBezier, чтобы компенсировать отсутствие элемента управления *Thumb*.

Thumb не только компонент шаблона *Slider*, он также может использоваться как элемент управления общего назначения для манипуляций. Проблема в том, что с шаблоном по умолчанию *Thumb* настолько уродлив, что его вполне можно зачислить в ряды «непригодных к использованию». Безусловно, для него необходимо создать пользовательский шаблон.

Thumb наследуется от *Control*, определяет метод *IsDragging* (Перетягивается) и три события: *DragStarted* (Перетягивание началось), *DragDelta* (Приращение в ходе перетягивания) и *DragCompleted* (Перетягивание завершено). Метод *CancelDrag* (Отменить перетягивание) позволяет отменить процесс на полпути.

Самым важным событием является *DragDelta*, имеющее два аргумента: *HorizontalChange* (Изменение по горизонтали) и *VerticalChange* (Изменение по вертикали). *Thumb* можно рассматривать как высокоуровневый интерфейс событий *Manipulation*, по крайней мере при работе с переносом.

Рассмотрим область содержимого приложения ThumbBezier (Кривая Безье с бегунком), которое аналогично приложению CubicBezier из главы 13 за исключением того, что в нем *TranslateTransform* используется для позиционирования четырех элементов *Thumb* в четырех точках.

Проект Silverlight: ThumbBezier Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Path Stroke="{StaticResource PhoneForegroundBrush}"
          StrokeThickness="2">
        <Path.Data>
            <PathGeometry>
```

```
                <PathFigure x:Name="pathFig"
                            StartPoint="100 100">
                    <BezierSegment x:Name="pathSeg"
                                   Point1="300 100"
                                   Point2="300 400"
                                   Point3="100 400" />
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>

    <Polyline Name="ctrl1Line"
              Stroke="{StaticResource PhoneForegroundBrush}"
              StrokeDashArray="2 2"
              Points="100 100, 300 100" />

    <Polyline Name="ctrl2Line"
              Stroke="{StaticResource PhoneForegroundBrush}"
              StrokeDashArray="2 2"
              Points="300 400, 100 400" />

    <Thumb Name="pt0Thumb"
           Style="{StaticResource thumbStyle}"
           DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TranslateTransform X="100" Y="100" />
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb Name="pt1Thumb"
           Style="{StaticResource thumbStyle}"
           DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TranslateTransform X="300" Y="100" />
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb Name="pt2Thumb"
           Style="{StaticResource thumbStyle}"
           DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TranslateTransform X="300" Y="400" />
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb Name="pt3Thumb"
           Style="{StaticResource thumbStyle}"
           DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TranslateTransform X="100" Y="400" />
        </Thumb.RenderTransform>
    </Thumb>
</Grid>
```

Все четыре элемента управления *Thumb* используют совместно один и тот же обработчик событий *DragDelta*, описание которого является практически единственной задачей файла выделенного кода в данном случае. Для работы с *Thumb* и аргументами его событий понадобится посредством директивы *using* подключить пространство имен *System.Windows.Control.Primitives*.

Проект Silverlight: ThumbBezier   Файл: MainPage.xaml.cs (фрагмент)

```
void OnThumbDragDelta(object sender, DragDeltaEventArgs args)
{
    Thumb thumb = sender as Thumb;
```

```
    TranslateTransform translate = thumb.RenderTransform as TranslateTransform;
    translate.X += args.HorizontalChange;
    translate.Y += args.VerticalChange;

    if (thumb == pt0Thumb)
    {
        pathFig.StartPoint =
            Move(pathFig.StartPoint, args.HorizontalChange, args.VerticalChange);
        ctrl1Line.Points[0] =
            Move(ctrl1Line.Points[0], args.HorizontalChange, args.VerticalChange);
    }
    else if (thumb == pt1Thumb)
    {
        pathSeg.Point1 =
            Move(pathSeg.Point1, args.HorizontalChange, args.VerticalChange);
        ctrl1Line.Points[1] =
            Move(ctrl1Line.Points[1], args.HorizontalChange, args.VerticalChange);
    }
    else if (thumb == pt2Thumb)
    {
        pathSeg.Point2 =
            Move(pathSeg.Point2, args.HorizontalChange, args.VerticalChange);
        ctrl2Line.Points[0] =
            Move(ctrl2Line.Points[0], args.HorizontalChange, args.VerticalChange);
    }
    else if (thumb == pt3Thumb)
    {
        pathSeg.Point3 =
            Move(pathSeg.Point3, args.HorizontalChange, args.VerticalChange);
        ctrl2Line.Points[1] =
            Move(ctrl2Line.Points[1], args.HorizontalChange, args.VerticalChange);
    }
}

Point Move(Point point, double horzChange, double vertChange)
{
    return new Point(point.X + horzChange, point.Y + vertChange);
}
```

Style и *ControlTemplate*, описанные в коллекции *Resources*, делают *Thumb* на вид очень похожим на полупрозрачные круглые элементы, используемые мною в предыдущей версии приложения. Единственным визуальным элементом *ControlTemplate* является *Path* с *EllipseGeometry*:

Проект Silverlight: ThumbBezier Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="thumbStyle" TargetType="Thumb">
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="VerticalAlignment" Value="Top" />
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="Thumb">

                    <Path Name="path"
                          Fill="{StaticResource PhoneAccentBrush}"
                          Opacity="0.5">
                        <Path.RenderTransform>
                            <ScaleTransform x:Name="scale" />
                        </Path.RenderTransform>

                        <Path.Data>
                            <EllipseGeometry x:Name="ellipseGeometry"
                                             RadiusX="48" RadiusY="48" />
                        </Path.Data>
```

```
                        <VisualStateManager.VisualStateGroups>
                            <VisualStateGroup x:Name="CommonStates">
                                <VisualState x:Name="Normal">
                                    <Storyboard>
                                        <DoubleAnimation
                                            Storyboard.TargetName="path"
                                            Storyboard.TargetProperty="Opacity"
                                            Duration="0:0:0.25" />
                                        <DoubleAnimation
                                            Storyboard.TargetName="scale"
                                            Storyboard.TargetProperty="ScaleX"
                                            Duration="0:0:0.25" />
                                        <DoubleAnimation
                                            Storyboard.TargetName="scale"
                                            Storyboard.TargetProperty="ScaleY"
                                            Duration="0:0:0.25" />

                                    </Storyboard>
                                </VisualState>

                                <VisualState x:Name="MouseOver" />
                                <VisualState x:Name="Disabled" />

                                <VisualState x:Name="Pressed">
                                    <Storyboard>
                                        <DoubleAnimation
                                            Storyboard.TargetName="path"
                                            Storyboard.TargetProperty="Opacity"
                                            To="0.75" Duration="0:0:0.25" />
                                        <DoubleAnimation
                                            Storyboard.TargetName="scale"
                                            Storyboard.TargetProperty="ScaleX"
                                            To="1.25" Duration="0:0:0.25" />
                                        <DoubleAnimation
                                            Storyboard.TargetName="scale"
                                            Storyboard.TargetProperty="ScaleY"
                                            To="1.25" Duration="0:0:0.25" />
                                    </Storyboard>
                                </VisualState>
                            </VisualStateGroup>
                        </VisualStateManager.VisualStateGroups>

                    </Path>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</phone:PhoneApplicationPage.Resources>
```

У *Thumb* есть визуальное состояние Pressed, т.е. может быть введена возможность, которой не было в предыдущих приложениях: *Thumb* будет показывать, что он нажат и перетягивается, немного увеличиваясь в размере и изменяя цвет. Несколько анимаций – это все что требуется для добавления этой возможности в шаблон.

Пользовательские элементы управления

Как известно, элементы управления, которые предполагается использовать только для специальных целей в собственных приложениях, проще всего создавать, наследуясь от *UserControl*. Для этого просто в XAML-файле определяем дерево визуальных элементов для элемента управления.

Можно применить аналогичный подход с *ContentControl*, но в этом случае XAML-файл будет включать определения *Style* и *ControlTemplate*. Преимущество такого подхода в использовании свойства *Content* для собственных целей элемента управления.

Также можно наследоваться от *Control*. Такой подход имеет смысл, если производный класс располагается в библиотеке, и вы хотите, чтобы элемент управления имел заменяемый шаблон. *Style* и *ControlTemplate* для темы по умолчанию располагаются в файле generic.xaml библиотеки.

В библиотеке Petzold.Phone.Silverlight есть пример такого элемента управления: *XYSlider*. Этот элемент управления позволяет пользователю перемещать *Thumb* по двухмерной поверхности, сохраняя его местоположение в свойстве *Value* типа *Point*. Но координаты нормализованы в диапазоне от 0 до 1 относительно верхнего левого угла. Такая нормализация устраняет необходимость задания значений *Minimum* и *Maximum*, как в обычном *Slider*.

Кроме свойства *Value*, класс *XYSlider* также определяет свойство *PlaneBackground* (Плоский фон) типа *Brush*. Это поверхность, по которой перемещается *Thumb*, и вскоре мы увидим, почему ее надо выносить отдельно от обычного свойства *Background* объекта *Control*.

Как видно из его атрибутов, в шаблоне для этого класса должны присутствовать два элемента: *Canvas* под именем «PlanePart» (Часть-описание плоскости) и *Thumb* под именем «ThumbPart» (Часть-описание бегунка):

Проект Silverlight: Petzold.Phone.Silverlight Файл: XYSlider.cs (фрагмент)

```
[TemplatePartAttribute(Name = "PlanePart", Type = typeof(Canvas))]
[TemplatePartAttribute(Name = "ThumbPart", Type = typeof(Thumb))]
public class XYSlider : Control
{
    Canvas planePart;
    Thumb thumbPart;
    Point absoluteThumbPoint;

    public event RoutedPropertyChangedEventHandler<Point> ValueChanged;

    public static readonly DependencyProperty PlaneBackgroundProperty =
        DependencyProperty.Register("PlaneBackground",
            typeof(Brush),
            typeof(XYSlider),
            new PropertyMetadata(new SolidColorBrush(Colors.Gray)));

    public static readonly DependencyProperty ValueProperty =
        DependencyProperty.Register("Value",
            typeof(Point),
            typeof(XYSlider),
            new PropertyMetadata(new Point(0.5, 0.5), OnValueChanged));

    public XYSlider()
    {
        this.DefaultStyleKey = typeof(XYSlider);
    }

    public Brush PlaneBackground
    {
        set { SetValue(PlaneBackgroundProperty, value); }
        get { return (Brush)GetValue(PlaneBackgroundProperty); }
    }

    public Point Value
    {
        set { SetValue(ValueProperty, value); }
```

```
        get { return (Point)GetValue(ValueProperty); }
    }

    …
}
```

Производный от *Control* объект получает уведомление о том, что его шаблон готов, посредством вызова метода *OnApplyTemplate* (При применении шаблона). Это подходящий момент для класса вызвать метод *GetTemplateChild*, передав в него имена, указанные в атрибутах. Правильным поведением класса будет допускать возможность отсутствия некоторых частей, даже частей, играющих ключевую роль в правильном функционировании элемента управления:

Проект Silverlight: Petzold.Phone.Silverlight Файл: XYSlider.cs (фрагмент)

```
public override void OnApplyTemplate()
{
    if (planePart != null)
    {
        planePart.SizeChanged -= OnPlaneSizeChanged;
    }

    if (thumbPart != null)
    {
        thumbPart.DragDelta -= OnThumbDragDelta;
    }

    planePart = GetTemplateChild("PlanePart") as Canvas;
    thumbPart = GetTemplateChild("ThumbPart") as Thumb;

    if (planePart != null && thumbPart != null)
    {
        planePart.SizeChanged += OnPlaneSizeChanged;
        thumbPart.DragStarted += OnThumbDragStarted;
        thumbPart.DragDelta += OnThumbDragDelta;
        ScaleValueToPlane(this.Value);
    }

    base.OnApplyTemplate();
}
```

В случае наличия *Canvas* и *Thumb* определяется обработчик событий *SizeChanged* объекта *Canvas* и событий *DragStarted* и *DragDelta* объекта *Thumb*.

Обработчик событий *SizeChanged* обновляет местоположение *Thumb* относительно *Canvas*; обработчик *DragDelta* обновляет значение свойства *Value* элемента управления *XYSlider*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: XYSlider.cs (фрагмент)

```
void OnPlaneSizeChanged(object sender, SizeChangedEventArgs args)
{
    ScaleValueToPlane(this.Value);
}

void OnThumbDragStarted(object sender, DragStartedEventArgs args)
{
    absoluteThumbPoint = new Point(Canvas.GetLeft(thumbPart),
                                   Canvas.GetTop(thumbPart));
}

void OnThumbDragDelta(object sender, DragDeltaEventArgs args)
{
```

```
    absoluteThumbPoint.X += args.HorizontalChange;
    absoluteThumbPoint.Y += args.VerticalChange;

    Value = new Point(Math.Max(0,
                      Math.Min(1, absoluteThumbPoint.X / planePart.ActualWidth)),
                      Math.Max(0,
                      Math.Min(1, absoluteThumbPoint.Y / planePart.ActualHeight)));
}

void ScaleValueToPlane(Point point)
{
    if (planePart != null && thumbPart != null)
    {
        Canvas.SetLeft(thumbPart, planePart.ActualWidth * point.X);
        Canvas.SetTop(thumbPart, planePart.ActualHeight * point.Y);
    }
}
```

Обработчик событий изменения значения свойства *Value* также обновляет координаты *Thumb* и формирует событие *ValueChanged*:

Проект Silverlight: Petzold.Phone.Silverlight Файл: XYSlider.cs (фрагмент)

```
static void OnValueChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
    (obj as XYSlider).OnValueChanged((Point)args.OldValue, (Point)args.NewValue);
}

protected virtual void OnValueChanged(Point oldValue, Point newValue)
{
    if (newValue.X < 0 || newValue.X > 1 || newValue.Y < 0 || newValue.Y > 1)
        throw new ArgumentOutOfRangeException("Value",
            "Value property must be Point with coordinates between 0 and 1");

    ScaleValueToPlane(newValue);

    if (ValueChanged != null)
        ValueChanged(this,
            new RoutedPropertyChangedEventArgs<Point>(oldValue, newValue));
}
```

Style и *ControlTemplate* по умолчанию располагаются в файле generic.xaml:

Проект Silverlight: Petzold.Phone.Sivlerlight Файл: Themes/generic.xaml (фрагмент)

```
<Style TargetType="local:XYSlider">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="local:XYSlider">
        <Border Background="{TemplateBinding Background}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}">

          <Canvas Name="PlanePart"
                  Background="{TemplateBinding PlaneBackground}"
                  Margin="48">
            <Thumb Name="ThumbPart">
              <Thumb.Style>
                <Style TargetType="Thumb">
                  <Setter Property="Width" Value="96" />
                  <Setter Property="Height" Value="96" />
                  <Setter Property="Template">
                    <Setter.Value>
```

```
                      <ControlTemplate TargetType="Thumb">
                        <Path Name="path"
                              Stroke="{StaticResource PhoneForegroundBrush}"
                              StrokeThickness="{StaticResource
                                                       PhoneStrokeThickness}"
                              Fill="Transparent">
                          <Path.Data>
                            <GeometryGroup FillRule="Nonzero">
                              <EllipseGeometry RadiusX="48" RadiusY="48" />
                              <EllipseGeometry RadiusX="6" RadiusY="6" />
                              <LineGeometry StartPoint="-48 0" EndPoint="-6 0" />
                              <LineGeometry StartPoint="48 0" EndPoint="6 0" />
                              <LineGeometry StartPoint="0 -48" EndPoint="0 -6" />
                              <LineGeometry StartPoint="0 48" EndPoint="0 6" />
                            </GeometryGroup>
                          </Path.Data>
                        </Path>
                      </ControlTemplate>
                    </Setter.Value>
                  </Setter>
                </Style>
              </Thumb.Style>
            </Thumb>
          </Canvas>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Border окружает весь элемент управления. *Canvas* под именем «PlanePart» задано свойство *Margin*, величина которого соответствует половине размера *Thumb*. Это позволяет центрировать *Thumb* для обозначения точки на плоскости, оставаясь при этом полностью в рамках элемента управления. В *ControlTemplate* для *Control* имеется другой *ControlTemplate* для *Thumb*, который формирует своего рода шаблон «мишени».

Протестируем приложение в проекте WorldMap. Область содержимого включает *XYSlider*, значением свойства *PlaneBackground* которого задан *ImageBrush*, использующий карту мира:

Проект Silverlight: WorldMap   Файл: MainPage.xaml (фрагмент)

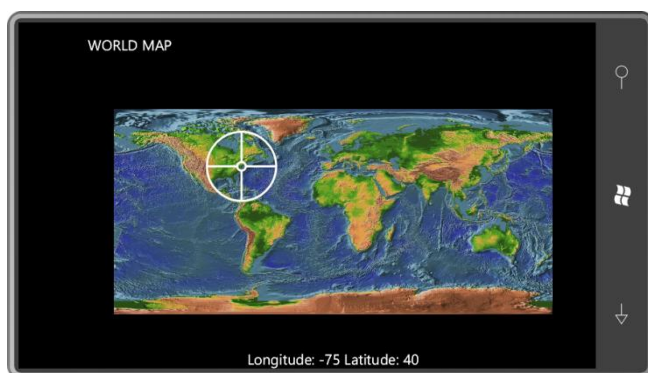
```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <petzold:XYSlider Name="xySlider"
                      Grid.Row="0"
                      ValueChanged="OnXYSliderValueChanged">
        <petzold:XYSlider.PlaneBackground>
            <!-- Изображение любезно предоставлено NASA/JPL-Caltech
(http://maps.jpl.nasa.gov). -->
            <ImageBrush ImageSource="Images/ear0xuu2.jpg" />
        </petzold:XYSlider.PlaneBackground>
    </petzold:XYSlider>

    <TextBlock Name="txtblk"
               Grid.Row="1"
               HorizontalAlignment="Center" />
</Grid>
```

Файл выделенного кода посвящен обработке события *ValueChanged*, формируемого *XYSlider*, и преобразованию нормализованных координат *Point* в широту и долготу:

Проект Silverlight: WorldMap Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        DisplayCoordinates(xySlider.Value);
    }

    void OnXYSliderValueChanged(object sender,
                                RoutedPropertyChangedEventArgs<Point> args)
    {
        DisplayCoordinates(args.NewValue);
    }

    void DisplayCoordinates(Point point)
    {
        double longitude = 360 * point.X - 180;
        double latitude = 90 - 180 * point.Y;
        txtblk.Text = String.Format("Longitude: {0:F0} Latitude: {1:F0}",
                                    longitude, latitude);
    }
}
```

И вот что мы имеем:



Проводя пальцем по экрану, мы перемещаем *Thumb*. При этом внизу экрана выводятся значения долготы и широты соответствующей точки на карте мира. Не сложно представить, как можно расширить приложение WorldMap, чтобы получать координаты телефона и использовать их при инициализации *Thumb*.

Глава 17
Элементы управления списками

Остается еще одна базовая категория элементов управления, которую мы до сих пор не обсудили. Это ветка класса *ItemsControl*, производного от *Control*. Привожу полную иерархию классов этой ветки:

Object
- *DependencyObject* (абстрактный)
  - *UIElement* (абстрактный)
    - *FrameworkElement* (абстрактный)
      - *Control* (абстрактный)
        - *ItemsControl*
          - *Selector* (абстрактный)
            - *ListBox*
            - *ComboBox*
          - *TemplatedItemsControl* (универсальный)
            - *Panorama*
            - *Pivot*
            - *PivotHeadersControl*
          - *MapItemsControl*

ItemsControl и его производные обеспечивают отображение коллекций элементов. Кроме того, класс *Selector* (Селектор) и его производные реализуют свойства и логику, позволяющие пользователю выбирать один или более элементов коллекции. (Классы, наследуемые от *TemplatedItemsControl* (Шаблонный элемент управления списками), будут обсуждаться в следующей главе.)

Наверное, самым известным из элементов управления этого типа является *ListBox*, который присутствует в средах Windows уже на протяжении 25 лет. Исходный *ListBox* – это вертикальный список элементов с возможностью прокрутки, по которым можно перемещаться с помощью клавиатуры или мыши. (В Windows Phone 7 это делается посредством сенсорного ввода.) Может быть выбран один или (такая возможность также может быть реализована) несколько элементов. Элемент управления выделяет выбранный элемент и делает его доступным для обработки. *ComboBox* (Поле со списком) появился несколько позднее, чем *ListBox*, и назван так, потому что сочетает поле с возможностью редактирования текста и выпадающий *ListBox*.

ItemsControl не так привычен ветеранам разработки на платформе Windows. Часто он очень похож на *ListBox*, но не реализует никакой логики выбора. (Он даже не реализует прокрутки, но ее не сложно добавить.) *ItemsControl* предназначен просто для целей представления. И хотя *ItemsControl* это не *ListBox*, он все равно имеет огромное значение в разработке на Silverlight и также пригодится для реализации пользовательской логики выбора.

ItemsControl и все его производные обобщенно называют *элементами управления списками* – три слова, которые вынесены в название этой главы. Я откладывал обсуждение этого семейства элементов управления до сих пор, потому что практически всегда для формирования визуального представления элементов в элементах управления *ItemsControl* используется *DataTemplate*. *ItemsControl* и *DataTemplate* буквально созданы друг для друга.

Элементы управления списками и деревья визуальных элементов

Существует три основных способа заполнения элемента управления списками: с помощью кода, XAML и привязки данных.

Метод с применением кода продемонстрирован в проекте ItemsControlsFromCode (Заполнение элементов управления списками из кода). Данное приложение предназначено для отображения в альбомном режиме. В каждом из трех столбцов сетки для содержимого создается по экземпляру *ItemsControl*, *ListBox* и *ComboBox*:

Проект Silverlight: ItemsControlsFromCode Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />п
    </Grid.ColumnDefinitions>

    <ItemsControl Name="itemsControl" Grid.Column="0" />

    <ListBox Name="listBox" Grid.Column="1" />

    <ComboBox Name="comboBox" Grid.Column="2"
              VerticalAlignment="Top"
              Foreground="Black" />
</Grid>
```

Я задал для *ComboBox* пару свойств. При реализации возможности выпадающего списка лучше выравнивать элемент управления по верху ячейки. Также я обнаружил, что шаблон по умолчанию для *ComboBox* не был скорректирован для телефона, поэтому для отображения элементов было необходимо задать свойство *Foreground*.

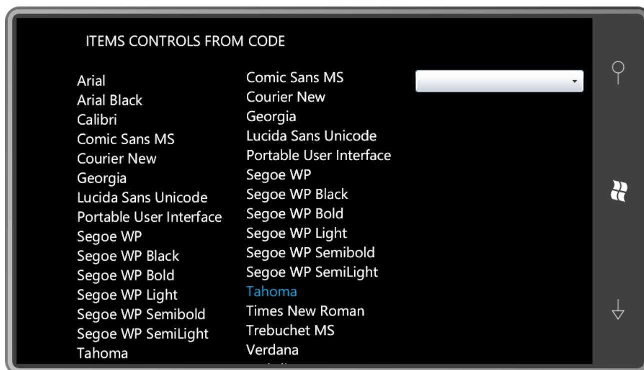
Файл выделенного кода заполняет каждый из этих элементов управления объектами *FontFamily*:

Проект Silverlight: ItemsControlsFromCode Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();

        FillItUp(itemsControl);
        FillItUp(listBox);
        FillItUp(comboBox);
    }

    void FillItUp(ItemsControl itemsControl)
    {
        string[] fontFamilies =
        {
            "Arial", "Arial Black", "Calibri", "Comic Sans MS",
            "Courier New", "Georgia", "Lucida Sans Unicode",
            "Portable User Interface", "Segoe WP", "Segoe WP Black",
            "Segoe WP Bold", "Segoe WP Light", "Segoe WP Semibold",
            "Segoe WP SemiLight", "Tahoma", "Times New Roman",
            "Trebuchet MS", "Verdana", "Webdings"
        };
```

```
        foreach (string fontFamily in fontFamilies)
            itemsControl.Items.Add(new FontFamily(fontFamily));
    }
}
```

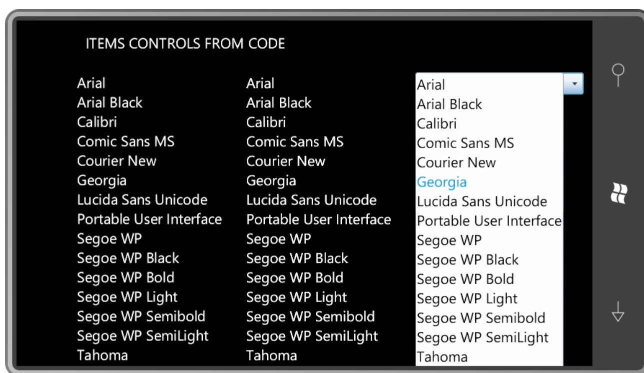
Свойство *Items* (Элементы), определенное *ItemsControl*, типа *ItemCollection* (Коллекция элементов), и в него можно поместить практически все что угодно. Если объект, помещаемый в коллекцию, наследуется от *FrameworkElement* (как *Button*, например), элемент будет самостоятельно реализовывать свое отображение. В противном случае применяется метод *ToString* элемента. Очень удобно, что в классе *FontFamily* есть метод *ToString*, который обеспечивает вывод на экран имени *FontFamily*:



Возможно, первое, на что обращаешь внимание в данном приложении – это отсутствие прокрутки в *ItemsControl*. Если хотите реализовать прокрутку *ItemsControl*, поместите его в *ScrollViewer*.

ListBox имеет собственный *ScrollViewer*. И прокрутка, и выбор элементов осуществляется посредством касания. При выборе элемента он выделяется контрастным цветом, как выделен шрифт Tahoma в данном примере.

ComboBox не раскрывается до тех пор, пока его не коснуться. В случае любого касания этого элемента управления, раскрывается список:



Теперь понятно, почему я задал свойству *Foreground* значение Black. Сначала я задал в качестве его значения ресурс *PhoneBackgroundBrush*, но затем обнаружил, что *ComboBox* использует те же цвета и со светлой темой.

Чтобы соответствовать эстетике Windows Phone 7, *ComboBox* необходим *ControlTemplate*. Поэтому я не буду описывать этот элемент управления в данной книге.

Большая часть этой главы посвящена определению шаблонов для элементов управления списками, поэтому полезным будет рассмотреть деревья визуальных элементов этих трех элементов управления, чтобы получить представление об их внутренней архитектуре.

Проект ItemsControlsVisualTrees (Деревья визуальных элементов для элементов управления списками) очень похож на проект ItemsControlsFromCode за исключением того, что в нем вместо *ComboBox* используется еще один *ItemsControl* (но этот уже в *ScrollViewer*), а также имеется несколько кнопок. Этот второй *ItemsControl* применяется для отображения деревьев визуальных элементов, ассоциированных с первым *ItemsControl* и *ListBox*.

Рассмотрим область содержимого, описанную в XAML-файле. Чтобы предоставить *ItemsControl*, используемому для отображения деревьев визуальных элементов, достаточно пространства по горизонтали, я уменьшил ширину первых двух столбцов:

Проект Silverlight: ItemsControlsVisualTrees Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ItemsControl Name="itemsControl"
                  Grid.Column="0" Grid.Row="0" />

    <Button Content="Dump"
            Grid.Column="0" Grid.Row="1"
            Click="OnItemsControlDumpClick" />

    <ListBox Name="listBox"
             Grid.Column="1" Grid.Row="0" />

    <Button Content="Dump"
            Grid.Column="1" Grid.Row="1"
            Click="OnListBoxDumpClick" />

    <ScrollViewer Grid.Column="2" Grid.Row="0" Grid.RowSpan="2">
        <ItemsControl Name="dumpTreeItemsControl" />
    </ScrollViewer>
</Grid>
```

Файл выделенного кода сначала заполняет первые два элемента управления объектами *FontFamily*, как это было в предыдущем приложении:

Проект Silverlight: ItemsControlsVisualTrees Файл: MainPage.xaml.cs (фрагмент)

```
public MainPage()
{
    InitializeComponent();

    FillItUp(itemsControl);
    FillItUp(listBox);
}

void FillItUp(ItemsControl itemsControl)
{
```

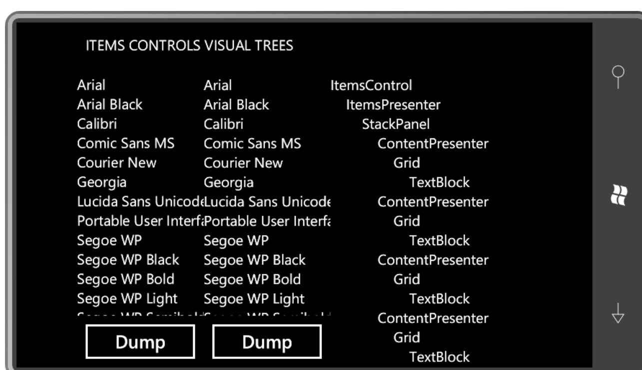
```
    string[] fontFamilies =
    {
        "Arial", "Arial Black", "Calibri", "Comic Sans MS",
        "Courier New", "Georgia", "Lucida Sans Unicode",
        "Portable User Interface", "Segoe WP", "Segoe WP Black",
        "Segoe WP Bold", "Segoe WP Light", "Segoe WP Semibold",
        "Segoe WP SemiLight", "Tahoma", "Times New Roman",
        "Trebuchet MS", "Verdana", "Webdings"
    };

    foreach (string fontFamily in fontFamilies)
        itemsControl.Items.Add(new FontFamily(fontFamily));
}
```

Этот класс также включает обработчики событий *Click* для двух кнопок и реагирует на эти события, отображая дерево визуальных элементов соответствующего элемента управления списками:

Проект Silverlight: ItemsControlsVisualTrees Файл: MainPage.xaml.cs (фрагмент)

```
void OnItemsControlDumpClick(object sender, RoutedEventArgs args)
{
    dumpTreeItemsControl.Items.Clear();
    DumpVisualTree(itemsControl, 0);
}

void OnListBoxDumpClick(object sender, RoutedEventArgs args)
{
    dumpTreeItemsControl.Items.Clear();
    DumpVisualTree(listBox, 0);
}

void DumpVisualTree(DependencyObject parent, int indent)
{
    TextBlock txtblk = new TextBlock();
    txtblk.Text = String.Format("{0}{1}", new string(' ', 4 * indent),
                                parent.GetType().Name);
    dumpTreeItemsControl.Items.Add(txtblk);

    int numChildren = VisualTreeHelper.GetChildrenCount(parent);

    for (int childIndex = 0; childIndex < numChildren; childIndex++)
    {
        DependencyObject child = VisualTreeHelper.GetChild(parent, childIndex);
        DumpVisualTree(child, indent + 1);
    }
}
```

На иллюстрации приложение отображает дерево визуальных элементов *ItemsControl*:



Полное дерево визуальных элементов потенциально включает несколько шаблонов. Это может обусловливать некоторые сложности, поэтому сделаем небольшой обзор.

В некоторых элементах управления, таких как *Slider*, *ControlTemplate* определяет представление всего элемента управления. Стиль темы для *Slider* определяет *ControlTemplate* по умолчанию и также можно определить собственный *ControlTemplate*. Этот *ControlTemplate* может включать описания других *ControlTemplate* для *ToggleButton* и *Thumb*, образующих *Slider*.

В производных *ContentControl*, таких как *Button*, потенциально имеется два шаблона: *ControlTemplate*, определяющий визуальный стиль элемента управления, и *DataTemplate*, описывающий, как в элементе управления формируется визуальное представление объекта, заданного в качестве значения свойства *Content*.

В производных *ItemsControl* используются три типа шаблонов: *ControlTemplate* для визуального стиля элемента управления, *ItemsPanelTemplate* для панели, которая используется для размещения элементов, и *DataTemplate*, применяемый к каждому элементу.

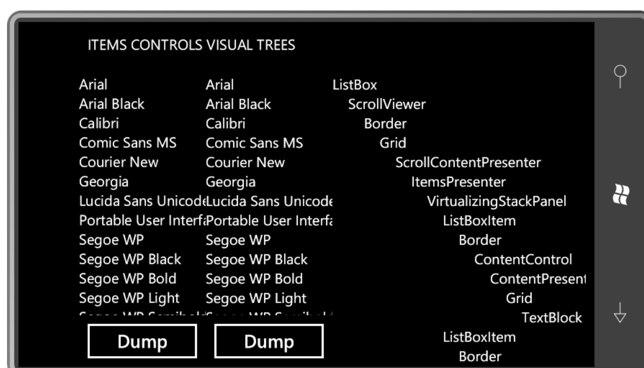
Дерево визуальных элементов *ItemsControl* начинается с *ControlTemplate* для элемента управления, и *ControlTemplate* по умолчанию для *ItemsControl* - просто *ItemsPresenter*. Это довольно загадочный класс, не определяющий не одного открытого свойства. *ItemsPresenter* иногда используется как «местозаполнитель» для элемента управления списками в пользовательском *ControlTemplate*.

Для отображения элементов в элементе управления списками *ItemsPresenter* всегда использует производный от *Panel*. В стандартном *ItemsControl* это *StackPanel* с вертикальной ориентацией. Эту панель можно заменить, задав свойству *ItemsPanel*, определенному *ItemsControl*, значение *ItemsPanelTemplate*. Как это делается, рассматривается далее в этой главе.

За этим следуют совершенно идентичные фрагменты деревьев визуальных элементов для каждого элемента коллекции, начиная со знакомого элемента под именем *ContentPresenter*. Если вспомнить предыдущую главу, *ContentPresenter* является ядром *ContentControl*. Этот элемент отвечает за размещение производных *FrameworkElement*; или преобразование объектов, не являющихся производными *FrameworkElement*, в текст с помощью метода *TextString*; или за размещение объекта с использованием дерева визуальных элементов, построенного на основании *DataTemplate*, который задан как значение свойства *ContentTemplate*. *ContentPresenter* играет такую же роль в данном случае, но для каждого элемента в отдельности.

В *ItemsControl*, рассматриваемом в данном приложении, каждый элемент отображается посредством *TextBlock*, помещенного в *Grid* с одной ячейкой.

Дерево визуальных элементов *ListBox* намного более запутанное:



Дерево визуальных элементов начинается с *ControlTemplate* по умолчанию для *ListBox*. Данное дерево начинается с *ScrollViewer*, который является элементом управления, поэтому имеет собственный *ControlTemplate* и собственное дерево визуальных элементов, начинающееся с *Border* и заканчивающееся *ScrollContentPresenter* (Средство отображения содержимого с прокруткой), который функционирует как механизм *ScrollViewer*. *ScrollViewer* наследуется от *ContentControl*, и в рамках *ControlTemplate* для *ListBox* значением свойства *Content* объекта *ScrollViewer* задается *ItemsPresenter* – тот же класс, который полностью формирует *ControlTemplate* по умолчанию для *ItemsControl*.

В дереве визуальных элементов *ItemsControl* в *ItemsPresenter* размещается *StackPanel*; а в *ListBox ItemsPresenter* размещается *VirtualizingStackPanel* (Виртуализирующая стек-панель). Позвольте мне вернуться к этому позднее.

В дереве визуальных элементов *ItemsControl* каждый элемент – это *ContentPresenter* (мы познакомились с этим классом в предыдущей главе). В данном случае каждый элемент – это *ListBoxItem* (Элемент окна списка), который сам по себе наследуется от *ContentControl* и имеет собственный шаблон и собственный *ContentPresenter*.

Почему такая разница? Почему у *ListBox* есть специальный класс под именем *ListBoxItem* для размещения каждого элемента, а у *ItemsControl* такого класса нет?

Ответ прост: выбор. Кто-то должен обрабатывать особое отображение выбранного элемента в *ListBox* и *ComboBox*, и именно для этой цели существуют классы *ListBoxItem* и *ComboBoxItem* (Элемент поля со списком) (который наследуется от *ListBoxItem*). *ListBoxItem* наследуется от *ContentControl* – из дерева визуальных элементов можно видеть, что в своем шаблоне он включает *ContentControl*, как и *Button* – но также определяет свойство *IsSelected*. *ListBox* знает, что его элементы располагаются в элементах управления *ListBoxItem*, поэтому он может задать свойство *IsSelected* для выбранного элемента, что используется шаблоном *ListBoxItem* для выделения этого элемента.

В системе понятий элементов управления списками *ListBoxItem* играет роль контейнера для элементов, располагающихся в *ListBox*. Эти контейнеры *ListBoxItem* создаются автоматически при добавлении новых элементов в *ListBox*. Открытый интерфейс для создания и управления этими контейнерами описывается классом *ItemsControl*. К нему относятся свойство *ItemContainerGenerator* (Генератор контейнера элементов) и несколько перегружаемых методов для определения альтернативного класса контейнера. Но обсуждение контейнеров выходит за рамки данной книги.

Может возникнуть желание определить для *ListBoxItem* другой *ControlTemplate*, например, чтобы изменить способ выделения выбранных элементов. Небольшое беспокойство вызывает то, что созданием и работой с экземплярами *ListBoxItem* занимается логика контейнера, но, к счастью, создать пользовательский *ControlTemplate* для *ListBoxItem* проще, чем кажется. И *ListBox*, и *ComboBox* определяют свойство *ItemContainerStyle* (Стиль контейнера элементов), в качестве значения которого можно задать объект *Style*, который *ListBox* применяет к каждому экземпляру *ListBoxItem*. Безусловно, *Style* может включать метод *Setter* для свойства *Template*. Это простой подход. А если требуется, чтобы *ListBox* применял пользовательский контейнер класса, унаследованный от *ListBoxItem*, придется обратиться к логике контейнера-генератора.

Если внимательно посмотреть на эти деревья визуальных элементов – не забывая о том, что в общем случае каждый элемент получит собственное дерево визуальных элементов, определенное *DataTemplate* – могут возникнуть опасения по поводу производительности. Не допускайте, чтобы *DataTemplate* был слишком сложным. *VirtualizingStackPanel* – это еще один класс, который способствует лучшей производительности. Он создает дерево визуальных

элементов для объекта только в момент, когда этот объект должен отображаться. От *VirtualizingPanel* можно наследовать собственные виртуализирующие панели, но, боюсь, эта тема также выходит за рамки данной книги.

Настройка представления элементов

Второй из трех подходов к заполнению элемента управления списками требует явного определения содержимого в XAML. Этот подход используется в проекте ItemsControlsFromXaml (Элементы управления списками из XAML) для заполнения *ItemsControl* и двух элементов управления *ListBox*. Определенное *ItemsControl* свойство *Items* – это свойство содержимого элемента управления, поэтому в XAML требуется лишь поместить все объекты между открывающим и закрывающим тегами заданного элемента управления списками.

Ожидается, что нам потребуется форматировать некоторые строки в привязках данных. Для этого включаем *StringFormatConverter* в коллекцию *Resources* в файле MainPage.xaml приложения:

Проект Silverlight: ItemsControlsFromXaml Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <petzold:StringFormatConverter x:Name="stringFormat" />
</phone:PhoneApplicationPage.Resources>
```

В области содержимого располагается *Grid* с тремя столбцами:

Проект Silverlight: ItemsControlsFromXaml Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    …

</Grid>
```

Первая ячейка *Grid* включает *ScrollViewer*, в котором располагается *ItemsControl*, в котором находятся объекты *Color* для всех цветов, определенных в Silverlight:

Проект Silverlight: ItemsControlsFromXaml Файл: MainPage.xaml (фрагмент)

```
<ScrollViewer Grid.Column="0">
    <ItemsControl>
        <Color>AliceBlue</Color>
        <Color>AntiqueWhite</Color>
        <Color>Aqua</Color>
        <Color>Aquamarine</Color>
        <Color>Azure</Color>
        …
        <Color>Wheat</Color>
        <Color>White</Color>
        <Color>WhiteSmoke</Color>
        <Color>Yellow</Color>
        <Color>YellowGreen</Color>
```

```
    </ItemsControl>
</ScrollViewer>
```

Исходя из нашего опыта по размещению объекта *Color* в *Button*, можно догадаться, что в результате мы получим список из 141 шестнадцатеричного значения цвета. Но по крайней мере, его можно будет прокручивать.

Для разнообразия во втором столбце *Grid* разместим *ListBox*, включающий 141 объект *SolidColorBrush*:

Проект Silverlight: ItemsControlsFromXaml Файл: MainPage.xaml (фрагмент)

```
<ListBox Grid.Column="1"
         DisplayMemberPath="Color">
    <SolidColorBrush Color="AliceBlue" />
    <SolidColorBrush Color="AntiqueWhite" />
    <SolidColorBrush Color="Aqua" />
    <SolidColorBrush Color="Aquamarine" />
    <SolidColorBrush Color="Azure" />
    …
    <SolidColorBrush Color="Wheat" />
    <SolidColorBrush Color="White" />
    <SolidColorBrush Color="WhiteSmoke" />
    <SolidColorBrush Color="Yellow" />
    <SolidColorBrush Color="YellowGreen" />
</ListBox>
```

Опять же из опыта размещения объекта *SolidColorBrush* в *Button* нам известно, что это приведет даже к еще более печальным результатам: мы получим 141 экземпляр текстовой строки с полным именем класса «System.Windows.Media.SolidColorBrush».

Но посмотрим на задание следующего свойства *ListBox*:

```
DisplayMemberPath="Color"
```

Это свойство определяется классом *ItemsControl* и позволяет задавать свойство элементов, входящих в элемент управления списками, которое будет использоваться для целей отображения. (Конечно, это целесообразно, только если все элементы одного типа, что не является обязательным требованием.) При таком значении данного свойства *ListBox* будет отображать не объект *SolidColorBrush*, а свойство *Color* каждого *SolidColorBrush* и те же шестнадцатеричные значения, что отображаются в *ItemsControl*.

Третий столбец *ListBox* описан правильно. Он включает все тот же 141 элемент *SolidColorBrush*, как и первый *ListBox*, но в нем также есть *DataTemplate*, заданный как значение свойства *ItemTemplate*, который позволяет определять формат отображения элементов:

Проект Silverlight: ItemsControlsFromXaml Файл: MainPage.xaml (фрагмент)

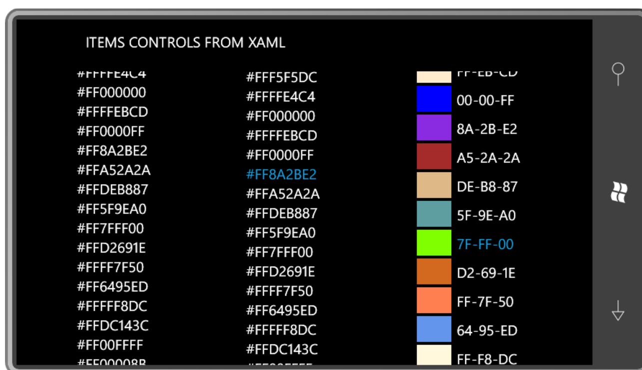
```
<ListBox Grid.Column="2">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Rectangle Width="48" Height="36"
                           Margin="2"
                           Fill="{Binding}" />

                <StackPanel Orientation="Horizontal"
                            VerticalAlignment="Center">
                    <TextBlock Text="{Binding Color.R,
```

```
                                    Converter={StaticResource stringFormat},
                                    ConverterParameter=' {0:X2}'}" />
                    <TextBlock Text="{Binding Color.G,
                                    Converter={StaticResource stringFormat},
                                    ConverterParameter='-{0:X2}'}" />
                    <TextBlock Text="{Binding Color.B,
                                    Converter={StaticResource stringFormat},
                                    ConverterParameter='-{0:X2}'}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>

    <SolidColorBrush Color="AliceBlue" />
    <SolidColorBrush Color="AntiqueWhite" />
    <SolidColorBrush Color="Aqua" />
    <SolidColorBrush Color="Aquamarine" />
    <SolidColorBrush Color="Azure" />
    …
    <SolidColorBrush Color="Wheat" />
    <SolidColorBrush Color="White" />
    <SolidColorBrush Color="WhiteSmoke" />
    <SolidColorBrush Color="Yellow" />
    <SolidColorBrush Color="YellowGreen" />
</ListBox>
```

В этом *DataTemplate* значением свойства *Fill* объекта *Rectangle* задан пустой *Binding*:

```
Fill="{Binding}"
```

Это означает, что *Fill* задан конкретный элемент *ListBox* типа *SolidColorBrush*. Привязки трех элементов *TextBlock* ссылаются на свойства *R*, *G* и *B* свойства *Color* кисти. И хотя этот *ListBox* по-прежнему выводит на экран шестнадцатеричные числа, по крайне мере он делает это стильно:



В XAML можно описывать список элементов, включающий небольшое количество неизменных элементов. Я использовал эту технику для отображения 141 значения *Color* лишь потому, что они не могут быть сгенерированы в коде на Silverlight посредством применения технологии отражения к классу *Colors*. (Класс *Colors* в Silverlight определяет лишь 15 из этих цветов, поэтому мне пришлось написать приложение на WPF, оно сформировало разметку, которую я затем скопировал и вставил в XAML-файл Silverlight.)

Если элементы, которые требуется поместить в *ItemsControl* или *ListBox* в XAML, являются просто текстовыми строками, синтаксический анализатор XAML должен как-то различать их. Возможно, самым простым решением будет задать пространство имен XML для пространства имен *System*:

```
xmlns:system="clr-namespace:System;assembly=mscorlib"
```

После этого можно явно разделять элементы тегами *String*:

```
<ItemsControl>
    <system:String>Item Number 1</system:String>
    <system:String>Item Number 2</system:String>
    <system:String>Item Number 3</system:String>
    <system:String>Item Number 4</system:String>
    …
</ItemsControl>
```

Аналогичным образом с помощью тегов *system:Double* можно явно заполнить *ItemsControl* числами. При использовании *ListBox*, а не *ItemsControl*, строковые элементы можно разделять тегами *ListBoxItem*:

```
<ListBox>
    <ListBoxItem>Item Number 1</ListBoxItem>
    <ListBoxItem>Item Number 1</ListBoxItem>
    <ListBoxItem>Item Number 1</ListBoxItem>
</ListBox>
```

Ранее я говорил, что *ListBox* автоматически формирует объекты *ListBoxItem* как контейнеры. Не приведет ли такая разметка к тому, что *ListBox* поместит эти объекты *ListBoxItem* в дополнительные объекты *ListBoxItem*? На самом деле, нет. Специально для предотвращения этой проблемы *ItemsControl* определяет виртуальный метод *IsItemItsOwnContainerOverride* (Элемент переопределяет собственный контейнер).

В предыдущей главе я привел небольшую схему, которая, надеюсь, помогла понять различия между двумя шаблонами, которые могут применяться к производному *ContentControl*:

Свойство	Тип свойства	Назначение
Template	*ControlTemplate*	настраивает визуальный стиль элемента управления
ContentTemplate	*DataTemplate*	настраивает представление содержимого

Существует три типа шаблонов, которые могут применяться к элементу управления списками, и еще один косвенно доступен для *ListBox* и *ComboBox*. Привожу их в таблице в порядке, в котором они могут встречаться в дереве визуальных элементов, сверху вниз.

Свойство	Тип свойства	Назначение
Template	*ControlTemplate*	настраивает визуальный стиль элемента управления
ItemsPanel	*ItemsPanelTemplate*	задает *Panel*, используемый для размещение элементов списка
ItemContainerStyle	*Style*	стиль *ListBoxItem* или *ComboBoxItem*
ItemTemplate	*DataTemplate*	настраивает представление самого элемента

Выбор в *ListBox*

Selector (класс, от которого наследуются *ListBox* и *ComboBox*) определяет свойство *SelectedIndex* (Индекс выбранного элемента), значением которого является индекс выбранного элемента или -1, если в данный момент не выбран ни один элемент. В *Selector* также описывается свойство *SelectedItem* (Выбранный элемент), значением которого является выбранный элемент или *null*, если ни один элемент не выбран. Если *SelectedIndex* не равен -1, значением *SelectedItem* является тот же объект, что возвращает свойство *Items* при передаче в него *SelectedIndex*.

Событие *SelectionChanged* формируется при выборе другого элемента. Это событие подразумевает, что *SelectedItem* замечательно подходит для использования в качестве источника привязки. *SelectedItem* дублируется свойством-зависимостью, так что он также может быть целью привязки.

Если свойства *SelectedIndex* или *SelectedItem* объекта *ListBox* не заданы явно, и пользователь еще не касался *ListBox*, *SelectedIndex* будет равен –1, и *SelectedItem* будет иметь значение *null*. Не лишним будет подготовиться к таким ситуациям.

Приложение ListBoxSelection (Выбор элемента в окне списка) позволяет пользователю выбирать *Color* и *FontFamily* из двух элементов управления *ListBox* и выводить на экран некоторый текст на основании сделанного выбора. Коллекция *Resources* включает стандартный конвертер привязок и *Style* для *ListBox*:

Проект Silverlight: ListBoxSelection Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <petzold:StringFormatConverter x:Name="stringFormat" />

    <Style x:Key="listBoxStyle"
           TargetType="ListBox">
        <Setter Property="BorderBrush"
                Value="{StaticResource PhoneForegroundBrush}" />
        <Setter Property="BorderThickness"
                Value="{StaticResource PhoneBorderThickness}" />
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="Margin" Value="3" />
        <Setter Property="Padding" Value="3" />
    </Style>
</phone:PhoneApplicationPage.Resources>
```

Все три элемента располагаются в *Grid* с тремя строками:

Проект Silverlight: ListBoxSelection Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    …

</Grid>
```

ListBox включает список объектов *SolidColorBrush* с тем же шаблоном *DataTemplate*, что использовался в предыдущем приложении для форматирования элементов:

Проект Silverlight: ListBoxSelection Файл: MainPage.xaml (фрагмент)

```
<ListBox Name="brushListBox"
         Grid.Row="0"
         SelectedIndex="0"
         Style="{StaticResource listBoxStyle}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Rectangle Width="48" Height="36"
                           Margin="2"
                           Fill="{Binding}" />

                <StackPanel Orientation="Horizontal"
```

```
                            VerticalAlignment="Center">
                    <TextBlock Text="{Binding Color.R,
                                      Converter={StaticResource stringFormat},
                                      ConverterParameter=' {0:X2}'}" />
                    <TextBlock Text="{Binding Color.G,
                                      Converter={StaticResource stringFormat},
                                      ConverterParameter='-{0:X2}'}" />
                    <TextBlock Text="{Binding Color.B,
                                      Converter={StaticResource stringFormat},
                                      ConverterParameter='-{0:X2}'}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>

    <SolidColorBrush Color="AliceBlue" />
    <SolidColorBrush Color="AntiqueWhite" />
    <SolidColorBrush Color="Aqua" />
    <SolidColorBrush Color="Aquamarine" />
    <SolidColorBrush Color="Azure" />
    …
    <SolidColorBrush Color="Wheat" />
    <SolidColorBrush Color="White" />
    <SolidColorBrush Color="WhiteSmoke" />
    <SolidColorBrush Color="Yellow" />
    <SolidColorBrush Color="YellowGreen" />
</ListBox>
```

Обратите внимание, что *SelectedIndex* явно задано значение 0. Благодаря этому *ListBox* будет иметь действительный *SelectedItem* при запуске приложения.

Во втором *ListBox* отображаются семейства шрифтов. Я бы предпочел использовать реальные объекты *FontFamily*, но они не могут быть созданы в XAML, потому что у *FontFamily* нет конструктора без параметров. Поэтому мне пришлось сохранять имена в виде строк. Начальным значением *SelectedIndex* задано 5, я выбрал это число произвольно.

Было бы вполне логичным, если бы имена семейств шрифтов в *ListBox* выводились шрифтом, который они обозначают? Это легко реализовать с помощью *DataTemplate*. Просто с помощью привязки свяжем свойства *Text* и *FontFamily* объекта *TextBlock* с элементами *ListBox*:

Проект Silverlight: ListBoxSelection Файл: MainPage.xaml (фрагмент)

```
<ListBox Name="fontFamilyListBox"
         Grid.Row="1"
         SelectedIndex="5"
         Style="{StaticResource listBoxStyle}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"
                       FontFamily="{Binding}" />
        </DataTemplate>
    </ListBox.ItemTemplate>

    <system:String>Arial</system:String>
    <system:String>Arial Black</system:String>
    <system:String>Calibri</system:String>
    <system:String>Comic Sans MS</system:String>
    <system:String>Courier New</system:String>
    <system:String>Georgia</system:String>
    <system:String>Lucida Sans Unicode</system:String>
    <system:String>Portable User Interface</system:String>
    <system:String>Segoe WP</system:String>
    <system:String>Segoe WP Black</system:String>
    <system:String>Segoe WP Bold</system:String>
```

```
    <system:String>Segoe WP Light</system:String>
    <system:String>Segoe WP Semibold</system:String>
    <system:String>Segoe WP SemiLight</system:String>
    <system:String>Tahoma</system:String>
    <system:String>Times New Roman</system:String>
    <system:String>Trebuchet MS</system:String>
    <system:String>Verdana</system:String>
    <system:String>Webdings</system:String>
</ListBox>
```

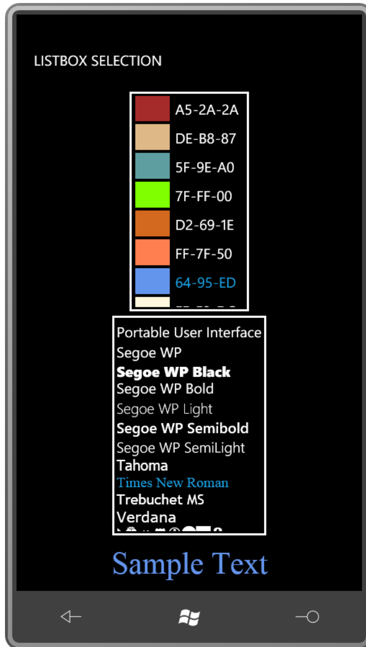
Поскольку элементы в *ListBox* являются строками, а не объектами *FontFamily*, я не был уверен, что привязка к *FontFamily* в шаблоне будет работать, но все получилось.

XAML-файл завершается описанием *TextBlock*, который не относится ни к одному шаблону. Два его свойства являются целями привязок, которые ссылаются на два элемента управления *ListBox*:

Проект Silverlight: ListBoxSelection   Файл: MainPage.xaml (фрагмент)

```
<TextBlock Grid.Row="2"
           Text="Sample Text"
           FontSize="{StaticResource PhoneFontSizeExtraLarge}"
           HorizontalAlignment="Center"
           Margin="12"

           Foreground="{Binding ElementName=brushListBox,
                                Path=SelectedItem}"

           FontFamily="{Binding ElementName=fontFamilyListBox,
                                Path=SelectedItem}" />
```

Когда я впервые создал это приложение, казалось, что привязка *FontFamily* в *DataTemplate* работала нормально, а вот привязка *FontFamily* в конце описания *TextBlock* приводила к исключению времени выполнения. Я написал *StringToFontFamilyConverter* (Конвертер строки в семейство шрифтов) (который до сих пор можно найти в библиотеке Petzold.Phone.Silverlight), но, казалось, проблема касалась значения *null* для *SelectedItem* из *ListBox*. Избавиться от этой проблемы с привязкой помогло явное задание начального *SelectedIndex*.



Если немного поиграть с этим приложением, можно заметить, что *TextBlock* немного меняет высоту при изменении *FontFamily*. Это создает эффект домино, приводя к изменениям высот двух *ListBox*. *ListBox* также может менять ширину. Если *HorizontalAlignment* не *Stretch*, ширина *ListBox* будет как раз такой, какая необходима для отображения содержимого. Но поскольку *ListBox* по умолчанию использует *VirtualizingStackPanel*, деревья визуальных элементов для элементов списка создаются только тогда, когда они должны отображаться. Таким образом, *ListBox* не всегда знает ширину своего самого широкого элемента. Очень странно видеть изменение ширины *ListBox* при прокрутке списка элементов!

По этим причинам ширину и высоту *ListBox* часто задают явно или посредством *Grid*.

Привязка к *ItemsSource*

Мы рассмотрели, как заполнять элемент управления списками посредством кода или с помощью списка, описанного в XAML. Также можно задать элементы с помощью свойства *ItemsSource*, определенного *ItemsControl*. Свойство *ItemsSource* типа *IEnumerable* (Перечислимый), поэтому в качестве его значения можно использовать практически любой тип коллекции, включая простой массив. Однако для работы с коллекцией, элементы которой могут добавляться и удаляться динамически, очень часто применяется класс *ObservableCollection* (Коллекция поддающаяся наблюдению), который реализует интерфейс *INotifyCollectionChanged* (Уведомление об изменении коллекции). Элемент управления списками устанавливает обработчик этого события, что позволяет ему получать уведомления об изменениях коллекции и обновляться соответственно.

При работе с данными часто требуется обеспечить программный интерфейс между объектами, которые мы хотим выводить на экран, и элементами пользовательского интерфейса, которые отображают их. Погружение в дебри архитектур модель-представление выходит далеко за рамки обсуждения данной книги. Вместо этого я выбрал более простой подход и предлагаю обсудить простые промежуточные классы, которые иногда называют *презентаторами*.

Создадим класс *ColorPresenter* (Презентатор цвета), который может заполнять *ListBox* 140 стандартными цветами (исключая *Transparent*) посредством единственной привязки к

ItemsSource и в то же время предоставляет свойства, позволяющие отображать эти цвета в более удобном для восприятия пользователем виде.

Остается загадкой, почему класс *Colors* в Silverlight определяет только 15 статических свойств типа *Color*, вместо 141. Это делает класс *ColorPresenter* довольно неуклюжим. У меня уже было приложение на WPF, в котором применялась технология отражения к WPF-классу *Colors*, так что я адаптировал его для формирования названий и значений цветов для передачи в этот класс. Они располагаются в статических массивах класса *ColorPresenter* в библиотеке Petzold.Phone.Silverlight:

Проект Silverlight: Petzold.Phone.Silverlight Файл: ColorPresenter.cs

```
using System;
using System.Text;
using System.Windows.Media;

namespace Petzold.Phone.Silverlight
{
    public class ColorPresenter
    {
        static string[] colorNames =
        {
            "AliceBlue", "AntiqueWhite", "Aqua", "Aquamarine", "Azure",
            "Beige", "Bisque", "Black", "BlanchedAlmond", "Blue", "BlueViolet",
            "Brown", "BurlyWood", "CadetBlue", "Chartreuse", "Chocolate",
            "Coral", "CornflowerBlue", "Cornsilk", "Crimson", "Cyan",
            "DarkBlue", "DarkCyan", "DarkGoldenrod", "DarkGray", "DarkGreen",
            "DarkKhaki", "DarkMagenta", "DarkOliveGreen", "DarkOrange",
            "DarkOrchid", "DarkRed", "DarkSalmon", "DarkSeaGreen",
            "DarkSlateBlue", "DarkSlateGray", "DarkTurquoise", "DarkViolet",
            "DeepPink", "DeepSkyBlue", "DimGray", "DodgerBlue", "Firebrick",
            "FloralWhite", "ForestGreen", "Fuchsia", "Gainsboro", "GhostWhite",
            "Gold", "Goldenrod", "Gray", "Green", "GreenYellow", "Honeydew",
            "HotPink", "IndianRed", "Indigo", "Ivory", "Khaki", "Lavender",
            "LavenderBlush", "LawnGreen", "LemonChiffon", "LightBlue",
            "LightCoral", "LightCyan", "LightGoldenrodYellow", "LightGray",
            "LightGreen", "LightPink", "LightSalmon", "LightSeaGreen",
            "LightSkyBlue", "LightSlateGray", "LightSteelBlue", "LightYellow",
            "Lime", "LimeGreen", "Linen", "Magenta", "Maroon",
            "MediumAquamarine", "MediumBlue", "MediumOrchid", "MediumPurple",
            "MediumSeaGreen", "MediumSlateBlue", "MediumSpringGreen",
            "MediumTurquoise", "MediumVioletRed", "MidnightBlue", "MintCream",
            "MistyRose", "Moccasin", "NavajoWhite", "Navy", "OldLace", "Olive",
            "OliveDrab", "Orange", "OrangeRed", "Orchid", "PaleGoldenrod",
            "PaleGreen", "PaleTurquoise", "PaleVioletRed", "PapayaWhip",
            "PeachPuff", "Peru", "Pink", "Plum", "PowderBlue", "Purple", "Red",
            "RosyBrown", "RoyalBlue", "SaddleBrown", "Salmon", "SandyBrown",
            "SeaGreen", "SeaShell", "Sienna", "Silver", "SkyBlue", "SlateBlue",
            "SlateGray", "Snow", "SpringGreen", "SteelBlue", "Tan", "Teal",
            "Thistle", "Tomato", "Turquoise", "Violet", "Wheat", "White",
            "WhiteSmoke", "Yellow", "YellowGreen"
        };

        static uint[] uintColors =
        {
            0xFFF0F8FF, 0xFFFAEBD7, 0xFF00FFFF, 0xFF7FFFD4, 0xFFF0FFFF,
            0xFFF5F5DC, 0xFFFFE4C4, 0xFF000000, 0xFFFFEBCD, 0xFF0000FF,
            0xFF8A2BE2, 0xFFA52A2A, 0xFFDEB887, 0xFF5F9EA0, 0xFF7FFF00,
            0xFFD2691E, 0xFFFF7F50, 0xFF6495ED, 0xFFFFF8DC, 0xFFDC143C,
            0xFF00FFFF, 0xFF00008B, 0xFF008B8B, 0xFFB8860B, 0xFFA9A9A9,
            0xFF006400, 0xFFBDB76B, 0xFF8B008B, 0xFF556B2F, 0xFFFF8C00,
            0xFF9932CC, 0xFF8B0000, 0xFFE9967A, 0xFF8FBC8F, 0xFF483D8B,
            0xFF2F4F4F, 0xFF00CED1, 0xFF9400D3, 0xFFFF1493, 0xFF00BFFF,
            0xFF696969, 0xFF1E90FF, 0xFFB22222, 0xFFFFFAF0, 0xFF228B22,
            0xFFFF00FF, 0xFFDCDCDC, 0xFFF8F8FF, 0xFFFFD700, 0xFFDAA520,
```

```
        0xFF808080, 0xFF008000, 0xFFADFF2F, 0xFFF0FFF0, 0xFFFF69B4,
        0xFFCD5C5C, 0xFF4B0082, 0xFFFFFFF0, 0xFFF0E68C, 0xFFE6E6FA,
        0xFFFFF0F5, 0xFF7CFC00, 0xFFFFFACD, 0xFFADD8E6, 0xFFF08080,
        0xFFE0FFFF, 0xFFFAFAD2, 0xFFD3D3D3, 0xFF90EE90, 0xFFFFB6C1,
        0xFFFFA07A, 0xFF20B2AA, 0xFF87CEFA, 0xFF778899, 0xFFB0C4DE,
        0xFFFFFFE0, 0xFF00FF00, 0xFF32CD32, 0xFFFAF0E6, 0xFFFF00FF,
        0xFF800000, 0xFF66CDAA, 0xFF0000CD, 0xFFBA55D3, 0xFF9370DB,
        0xFF3CB371, 0xFF7B68EE, 0xFF00FA9A, 0xFF48D1CC, 0xFFC71585,
        0xFF191970, 0xFFF5FFFA, 0xFFFFE4E1, 0xFFFFE4B5, 0xFFFFDEAD,
        0xFF000080, 0xFFFDF5E6, 0xFF808000, 0xFF6B8E23, 0xFFFFA500,
        0xFFFF4500, 0xFFDA70D6, 0xFFEEE8AA, 0xFF98FB98, 0xFFAFEEEE,
        0xFFDB7093, 0xFFFFEFD5, 0xFFFFDAB9, 0xFFCD853F, 0xFFFFC0CB,
        0xFFDDA0DD, 0xFFB0E0E6, 0xFF800080, 0xFFFF0000, 0xFFBC8F8F,
        0xFF4169E1, 0xFF8B4513, 0xFFFA8072, 0xFFF4A460, 0xFF2E8B57,
        0xFFFFF5EE, 0xFFA0522D, 0xFFC0C0C0, 0xFF87CEEB, 0xFF6A5ACD,
        0xFF708090, 0xFFFFFAFA, 0xFF00FF7F, 0xFF4682B4, 0xFFD2B48C,
        0xFF008080, 0xFFD8BFD8, 0xFFFF6347, 0xFF40E0D0, 0xFFEE82EE,
        0xFFF5DEB3, 0xFFFFFFFF, 0xFFF5F5F5, 0xFFFFFF00, 0xFF9ACD32
    };

    // Статический конструктор
    static ColorPresenter()
    {
        Colors = new ColorPresenter[140];

        for (int i = 0; i < 140; i++)
        {
            // Раскладываем значение цвета на компоненты
            byte A = (byte)((uintColors[i] & 0xFF000000) >> 24);
            byte R = (byte)((uintColors[i] & 0x00FF0000) >> 16);
            byte G = (byte)((uintColors[i] & 0x0000FF00) >> 8);
            byte B = (byte)((uintColors[i] & 0x000000FF) >> 0);

            // Создаем отображаемое имя для цвета
            StringBuilder builder = new StringBuilder();

            foreach (char ch in colorNames[i])
            {
                if (builder.Length == 0 || Char.IsLower(ch))
                    builder.Append(ch);
                else
                {
                    builder.Append(' ');
                    builder.Append(ch);
                }
            }

            // Создаем ColorPresenter для каждого цвета
            ColorPresenter clrPresenter = new ColorPresenter();
            clrPresenter.Color = Color.FromArgb(A, R, G, B);
            clrPresenter.Name = colorNames[i];
            clrPresenter.DisplayName = builder.ToString();
            clrPresenter.Brush = new SolidColorBrush(clrPresenter.Color);

            // Добавляем его в статический массив
            Colors[i] = clrPresenter;
        }
    }

    public static ColorPresenter[] Colors { protected set; get; }

    public Color Color { protected set; get; }

    public string Name { protected set; get; }

    public string DisplayName { protected set; get; }

    public Brush Brush { protected set; get; }
```

```
        public override string ToString()
        {
            return Name;
        }
    }
}
```

В конце этого фрагмента мы видим открытые свойства экземпляров, предоставляемые *ColorPresenter*: *Color* типа *Color*, *Brush* типа *Brush*, а также *Name* типа *string* и *DisplayName* (Отображаемое имя). Свойство *DisplayName* преобразовывает стандартные имена, записанные одним словом с применением нотации «camel», в несколько слов. Например, «AliceBlue» становится «Alice Blue».

ColorPresenter также предоставляет открытое статическое свойство *Colors* (Цвета). Это массив, включающий все 140 объектов *ColorPresenter*. Этот массив и все его содержимое создается в статическом конструкторе класса.

Если бы *ColorPresenter* использовался исключительно в коде, нам не пришлось бы создавать никаких дополнительных экземпляров этого класса. Для получения всех 140 объектов *ColorPresenter* можно было бы просто выполнять доступ к статическому свойству *ColorPresenter.Colors*.

Но Silverlight не обеспечивает возможности доступа к статическому свойству в XAML без создания экземпляра класса, включающего это свойство, поэтому в проекте ColorPresenterDemo (Демонстрация презентатора цвета) класс *ColorPresenter* включен в коллекцию *Resources*:

Проект Silverlight: ColorPresenterDemo Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <petzold:ColorPresenter x:Key="colorPresenter" />
    <petzold:StringFormatConverter x:Key="stringFormat" />
</phone:PhoneApplicationPage.Resources>
```

У экземпляра *ColorPresenter*, созданного в XAML-файле, не будет никаких полезных свойств экземпляра, но приложению необходимо только его статическое свойство *Colors*.

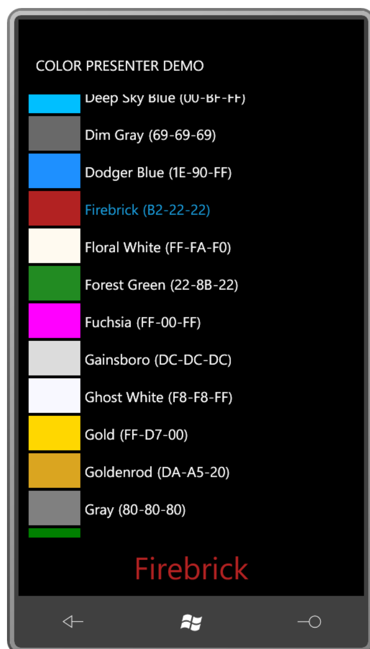
В *Grid* для содержимого только две строки: одна для размещения *ListBox*, и другая – для *TextBlock* с привязками к *ListBox*. Обратите внимание, что свойство *ItemsSource* класса *ListBox* связано посредством привязки со свойством *Colors* ресурса *ColorPresenter*. Благодаря этой привязке *ListBox* заполняется 140 объектами типа *ColorPresenter*, благодаря чему *DataTemplate* может иметь привязки к свойствам *DisplayName* и *Color* этого класса:

Проект Silverlight: ColorPresenterDemo Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListBox Grid.Row="0"
             Name="listBox"
             ItemsSource="{Binding Source={StaticResource colorPresenter},
                                   Path=Colors}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Grid>
```

```
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>

                    <Rectangle Grid.Column="0"
                               Fill="{Binding Brush}"
                               Width="72" Height="48"
                               Margin="2 2 6 2" />

                    <StackPanel Grid.Column="1"
                                Orientation="Horizontal"
                                VerticalAlignment="Center">

                        <TextBlock Text="{Binding DisplayName}" />

                        <TextBlock Text="{Binding Color.R,
                                       Converter={StaticResource stringFormat},
                                        ConverterParameter=' ({0:X2}'}" />

                        <TextBlock Text="{Binding Color.G,
                                       Converter={StaticResource stringFormat},
                                       ConverterParameter='-{0:X2}'}" />

                        <TextBlock Text="{Binding Color.B,
                                       Converter={StaticResource stringFormat},
                                       ConverterParameter='-{0:X2})'}" />
                    </StackPanel>
                </Grid>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>

    <TextBlock Grid.Row="1"
               FontSize="{StaticResource PhoneFontSizeExtraLarge}"
               HorizontalAlignment="Center"
               Margin="12"

               Text="{Binding ElementName=listBox,
                              Path=SelectedItem.DisplayName}"

               Foreground="{Binding ElementName=listBox,
                                    Path=SelectedItem.Brush}" />
</Grid>
```

Свойство *SelectedItem* также типа *ColorPresenter*, так что *TextBlock* может ссылаться на свойства *ColorPresenter* для привязок к *Text* и *Foreground*:



Отображение имен цветов в пользовательском интерфейсе выбора цвета – довольно полезная функция, реализацией которой не стоит пренебрегать. Пользователи, имеющие опыт работы с цветами в HTML, будут очень признательны за это.

Базы данных и бизнес-объекты

Использовать *ListBox* для отображения объектов *Color* или *FontFamily* можно в каких-то специализированных приложениях, но что вы собираетесь поместить в *свой* элемент управления списками?

Как правило, *ItemsControl* или *ListBox* заполняются этими непонятными, но при этом повсеместно применяющимися сущностями, которые называют *бизнес-объектами*.

Например, в приложении для выбора гостиницы, скорее всего, будет использоваться класс *Hotel* (Гостиница), и *ListBox* будет заполняться объектами *Hotel*. Являясь бизнес-объектом, *Hotel* не будет наследоваться от *FrameworkElement*. Но весьма вероятно, что *Hotel* будет реализовывать интерфейс *INotifyPropertyChanged*, чтобы иметь возможность динамически обновлять стоимость номера. Другой бизнес-объект будет сохранять коллекцию объектов *Hotel*, возможно, используя *ObservableCollection* и реализуя *INotifyCollectionChanged* для динамического отображения изменений при вводе в эксплуатацию новой гостиницы.

Чтобы создать пример, немного более приближенный к реальному приложению, я собираюсь посвятить остаток этой главы приложениям, использующим базу данных учеников школы. В этих примерах база данных загружается из папки на моем Веб-сайте. Но поскольку в этой главе я хочу сосредоточиться исключительно на представлении этих данных, изменения свойств класса *Student* (Учащийся) будут моделироваться локально.

В папке *http://www.charlespetzold.com/Students* моего Веб-сайта располагается файл students.xml, включающий данные 69 учащихся. В этой папке также хранятся черно-белые фотографии всех этих учеников. Фотографии я взял из школьных альбомов города Эль-Пасо, штат Техас, за 1912 – 1914 годы. Эти школьные альбомы хранятся в оцифрованном виде в Публичной библиотеке города Эль-Пасо и доступны на их сайте по адресу *http://www.elpasotexas.gov/library/ourlibraries/main_library/yearbooks/yearbooks.asp*.

Среди исходного кода для главы 17 можно найти проект библиотеки ElPasoHighSchool, которая включает несколько классов для чтения XML-файла с моего Веб-сайта и его десериализации в .NET-объекты.

Рассмотрим класс *Student*. Он реализует *INotifyPropertyChanged* и имеет несколько свойств, относящихся к учащемуся, включая имя, пол, имя файла с фотографией и средний балл:

Проект Silverlight: ElPasoHighSchool Файл: Student.cs

```
using System;
using System.ComponentModel;

namespace ElPasoHighSchool
{
    public class Student : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        string fullName;
        string firstName;
        string middleName;
        string lastName;
        string sex;
        string photoFilename;
        decimal gradePointAverage;

        public string FullName
        {
            set
            {
                if (fullName != value)
                {
                    fullName = value;
                    OnPropertyChanged("FullName");
                }
            }
            get
            {
                return fullName;
            }
        }

        public string FirstName
        {
            set
            {
                if (firstName != value)
                {
                    firstName = value;
                    OnPropertyChanged("FirstName");
                }
            }
            get
            {
                return firstName;
            }
        }

        public string MiddleName
        {
            set
            {
                if (middleName != value)
                {
                    middleName = value;
                    OnPropertyChanged("MiddleName");
```

```
                }
            }
            get
            {
                return middleName;
            }
        }

        public string LastName
        {
            set
            {
                if (lastName != value)
                {
                    lastName = value;
                    OnPropertyChanged("LastName");
                }
            }
            get
            {
                return lastName;
            }
        }

        public string Sex
        {
            set
            {
                if (sex != value)
                {
                    sex = value;
                    OnPropertyChanged("Sex");
                }
            }
            get
            {
                return sex;
            }
        }

        public string PhotoFilename
        {
            set
            {
                if (photoFilename != value)
                {
                    photoFilename = value;
                    OnPropertyChanged("PhotoFilename");
                }
            }
            get
            {
                return photoFilename;
            }
        }

        public decimal GradePointAverage
        {
            set
            {
                if (gradePointAverage != value)
                {
                    gradePointAverage = value;
                    OnPropertyChanged("GradePointAverage");
                }
            }
            get
            {
                return gradePointAverage;
```

```
            }
        }

        protected virtual void OnPropertyChanged(string propChanged)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propChanged));
        }
    }
}
```

Для каждого учащегося будет создан один экземпляр класса *Student*. Изменение любого из перечисленных свойств приводит к формированию события *PropertyChanged*. Таким образом, этот класс подходит для использования в качестве источника для привязок данных.

Класс *StudentBody* (Основные сведения об учащемся) также реализует *INotifyPropertyChanged*:

Проект Silverlight: ElPasoHighSchool Файл: StudentBody.cs

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Xml.Serialization;

namespace ElPasoHighSchool
{
    public class StudentBody : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        string school;
        ObservableCollection<Student> students =
                                    new ObservableCollection<Student>();

        public string School
        {
            set
            {
                if (school != value)
                {
                    school = value;
                    OnPropertyChanged("School");
                }
            }
            get
            {
                return school;
            }
        }

        public ObservableCollection<Student> Students
        {
            set
            {
                if (students != value)
                {
                    students = value;
                    OnPropertyChanged("Students");

                }
            }
            get
            {
                return students;
            }
        }
```

```
        protected virtual void OnPropertyChanged(string propChanged)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propChanged));
        }
    }
}
```

Этот класс включает свойство для обозначения названия школы и *ObservableCollection* типа *Student* для хранения всех объектов *Student*. *ObservableCollection* очень популярен в Silverlight, потому что реализует интерфейс *INotifyCollectionChanged*, т.е. формирует события *CollectionChanged* при каждом добавлении или удалении элемента из коллекции.

Прежде чем продолжить, рассмотрим фрагмент файла student.xml, который хранится на моем Веб-сайте:

Файл: http://www.charlespetzold.com/Students/students.xml (фрагмент)

```
<?xml version="1.0" encoding="utf-8"?>
<StudentBody xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <School>El Paso High School</School>
  <Students>
    <Student>
      <FullName>Adkins Bowden</FullName>
      <FirstName>Adkins</FirstName>
      <MiddleName />
      <LastName>Bowden</LastName>
      <Sex>Male</Sex>
      <PhotoFilename>
          http://www.charlespetzold.com/Students/AdkinsBowden.png
      </PhotoFilename>
      <GradePointAverage>2.71</GradePointAverage>
    </Student>
    <Student>
      <FullName>Alfred Black</FullName>
      <FirstName>Alfred</FirstName>
      <MiddleName />
      <LastName>Black</LastName>
      <Sex>Male</Sex>
      <PhotoFilename>
          http://www.charlespetzold.com/Students/AlfredBlack.png
      </PhotoFilename>
      <GradePointAverage>2.87</GradePointAverage>
    </Student>
    …

    <Student>
      <FullName>William Sheley Warnock</FullName>
      <FirstName>William</FirstName>
      <MiddleName>Sheley</MiddleName>
      <LastName>Warnock</LastName>
      <Sex>Male</Sex>
      <PhotoFilename>
          http://www.charlespetzold.com/Students/WilliamSheleyWarnock.png
      </PhotoFilename>
      <GradePointAverage>1.82</GradePointAverage>
    </Student>
  </Students>
</StudentBody>
```

Как видите, теги элементов соответствуют свойствам классов *Student* и *StudentBody*. Я создал этот файл, применяя сериализацию XML посредством класса *XmlSerializer*. С помощью

десериализации XML этот файл может быть преобразован назад в объекты *Student* и *StudentBody*. Это задача класса *StudentBodyPresenter*, который снова реализует *INotifyPropertyChanged*:

Проект Silverlight: ElPasoHighSchool Файл: StudentBodyPresenter.cs

```
using System;
using System.ComponentModel;
using System.IO;
using System.Net;
using System.Windows.Threading;
using System.Xml.Serialization;

namespace ElPasoHighSchool
{
    public class StudentBodyPresenter : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        StudentBody studentBody;
        Random rand = new Random();

        public StudentBodyPresenter()
        {
            Uri uri =
                new Uri("http://www.charlespetzold.com/Students/students.xml");

            WebClient webClient = new WebClient();
            webClient.DownloadStringCompleted += OnDownloadStringCompleted;
            webClient.DownloadStringAsync(uri);
        }

        void OnDownloadStringCompleted(object sender,
                                       DownloadStringCompletedEventArgs args)
        {
            StringReader reader = new StringReader(args.Result);
            XmlSerializer xml = new XmlSerializer(typeof(StudentBody));
            StudentBody = xml.Deserialize(reader) as StudentBody;

            DispatcherTimer tmr = new DispatcherTimer();
            tmr.Tick += TimerOnTick;
            tmr.Interval = TimeSpan.FromMilliseconds(100);
            tmr.Start();
        }

        public StudentBody StudentBody
        {
            protected set
            {
                if (studentBody != value)
                {
                    studentBody = value;
                    OnPropertyChanged("StudentBody");
                }
            }
            get
            {
                return studentBody;
            }
        }

        protected virtual void OnPropertyChanged(string propChanged)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propChanged));
        }
```

```
        void TimerOnTick(object sender, EventArgs args)
        {
            int index = rand.Next(studentBody.Students.Count);
            Student student = studentBody.Students[index];

            double factor = 1 + (rand.NextDouble() - 0.5) / 5;

            student.GradePointAverage =
                    Math.Max(0, Math.Min(5, Decimal.Round((decimal)factor *
                                            student.GradePointAverage, 2)));
        }
    }
}
```

Класс *StudentBodyPresenter* (Презентатор основных сведений об учащемся) реализует доступ к файлу students.xml с помощью класса *WebClient*. Как мы помним, *WebClient* осуществляет асинхронный Веб-доступ, поэтому нуждается в методе обратного вызова для сообщения приложению о завершении своего выполнения. После этого метод *Deserialize* класса *XmlSerializer* преобразует текстовый XML-файл в объект *StudentBody*, который доступен как открытое свойство этого класса. Когда метод обратного вызова *OnDownloadStringCompleted* (По завершении загрузки строки) задает это свойство, класс формирует свое первое и единственное событие *PropertyChanged*.

Обратный вызов *OnDownloadStringCompleted* также запускает *DispatcherTimer*, который моделирует изменение данных. Свойство *GradePointAverage* одного из студентов меняется десять раз в секунду, что приводит к формированию события *PropertyChanged* соответствующим классом *Student*. Я очень надеюсь увидеть эти динамические изменения на экране.

Эксперименты с базой данных можно начать с открытия нового проекта на Silverlight, включения в него ссылки на библиотеку ElPasoHighSchool.dll и объявления пространства имен XML в файле MainPage.xaml:

```
xmlns:elpaso="clr-namespace:ElPasoHighSchool;assembly=ElPasoHighSchool"
```

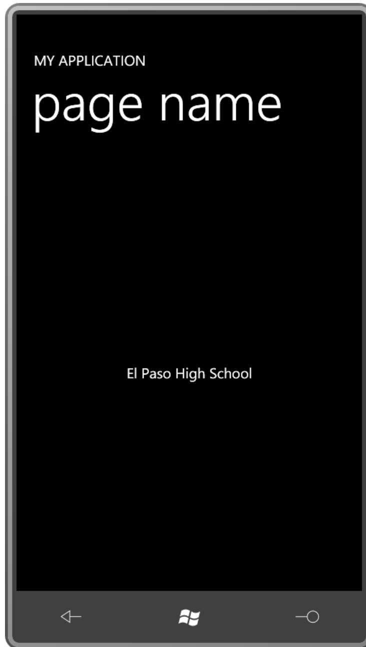
После этого создаем экземпляр класса *StudentBodyPresenter* в коллекции *Resources*:

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />
</phone:PhoneApplicationPage.Resources>
```

Теперь можно поместить в область содержимого *TextBlock* с привязкой к этому ресурсу:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Text="{Binding Source={StaticResource studentBodyPresenter},
                              Path=StudentBody.School}" />
</Grid>
```

Такой экран свидетельствует об успешной загрузке и десериализации файла students.xml нашим приложением:



Если изменить путь привязки и задать StudentBody.Students вместо StudentBody.School, на экран будет выведен *ObservableCollection*:

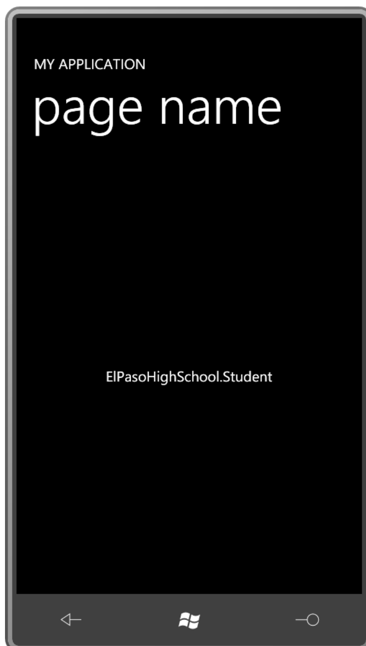
Можно выполнить доступ к свойству *Count* класса *ObservableCollection*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Text="{Binding Source={StaticResource studentBodyPresenter},
                              Path=StudentBody.Students.Count}" />
</Grid>
```

И коллекция *Students* может быть проиндексирована:

```
<TextBlock HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Text="{Binding Source={StaticResource studentBodyPresenter},
                          Path=StudentBody.Students[23]}" />
```

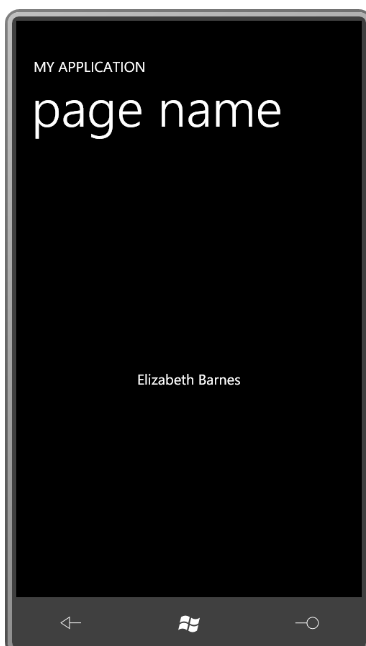
Здесь мы видим, что коллекция *Students* включает объекты типа *Student*:



Чтобы не удлинять эту привязку, разделим ее, задав *DataContext* в *Grid* для содержимого. *DataContext* наследуется по дереву визуальных элементов и упрощает описание привязки в *TextBlock*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Text="{Binding Path=Students[23].FullName}" />
</Grid>
```

Данная привязка указывает на имя конкретного учащегося:



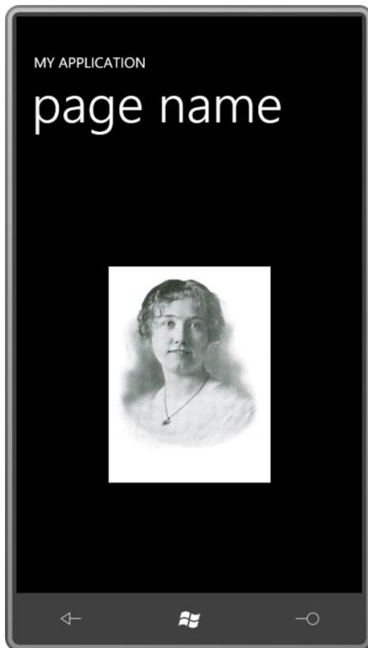
Привязку можно еще более упростить, убрав из нее часть «Path=»:

```
<TextBlock HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Text="{Binding Students[23].FullName}" />
```

Теперь заменим *TextBlock* элементом *Image*, который будет ссылаться на свойство *PhotoFilename* класса *Student*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <Image HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Stretch="None"
           Source="{Binding Students[23].PhotoFilename}" />

</Grid>
```

И фотография учащегося успешно загружается и выводится на экран:

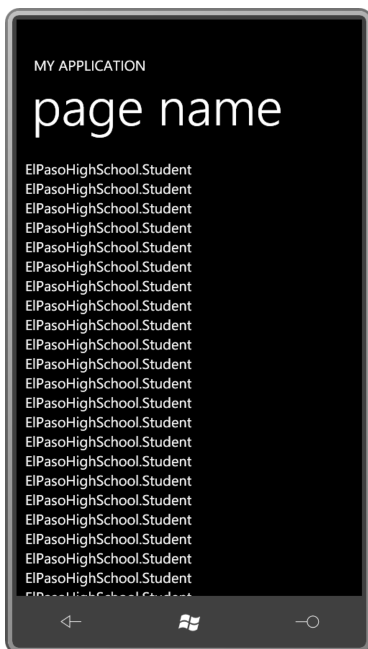


Ну, а теперь пора заканчивать заниматься глупостями и вставить реальный *ListBox*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <ListBox ItemsSource="{Binding Students}" />

</Grid>
```

Свойство *Students* типа *ObservableCollection*, который, безусловно, реализует *IEnumerable*, а это на самом деле все что необходимо *ListBox* для его *ItemsSource*. Но также *ListBox* проверяет, способен ли объект, связанный с *ItemsSource* посредством привязки, на что-то большее. Например, реализует ли он *INotifyCollectionChanged*, что делает *ObservableCollection*. Благодаря реализации *INotifyCollectionChanged* при добавлении нового *Student* в коллекцию или удалении из коллекции объектов учащихся, окончивших школу, *ListBox* будет знать об этом и менять отображаемые элементы соответствующим образом.

Но пока что, кажется, *ListBox* не очень доволен своими данными:



При виде *ListBox* или *ItemsControl*, отображающего огромный список идентичных имен классов, не впадайте в отчаяние. Наоборот, надо радоваться! Такой вывод свидетельствует о том, что *ListBox* успешно заполнен элементами одного типа. Теперь для отображения чего-то более значащего ему необходим лишь *DataTemplate* или (для ленивых) параметр *DisplayMemberPath*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <ListBox ItemsSource="{Binding Students}"
             DisplayMemberPath="FullName" />

</Grid>
```

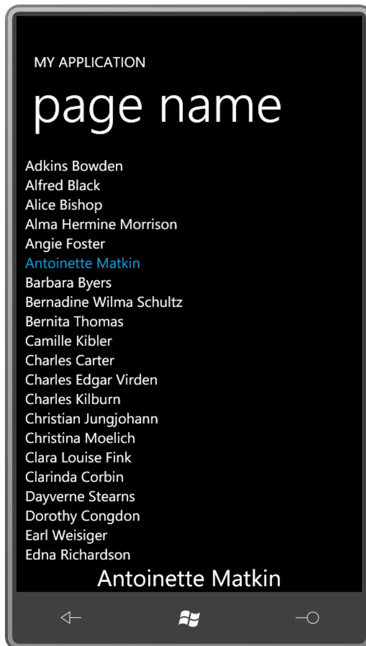
Вот что получается:

Давайте пока оставим *ListBox* как есть и сосредоточимся на отображении выделенного элемента *ListBox*.

Добавив еще одну строку в *Grid*, разместим внизу экрана еще один *TextBlock*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListBox Grid.Row="0"
             Name="listBox"
             ItemsSource="{Binding Students}"
             DisplayMemberPath="FullName" />

    <TextBlock Grid.Row="1"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               Text="{Binding ElementName=listBox,
                              Path=SelectedItem.FullName}" />
</Grid>
```

Обратите внимание на привязку, заданную для *TextBlock*. Свойство *SelectedItem* класса *ListBox* типа *Student*, поэтому путь привязки может ссылаться на одно из свойств *Student*, например *FullName* (Полное имя). Теперь при выборе элемента *ListBox* в *TextBlock* отображается значение свойства *FullName* этого элемента:

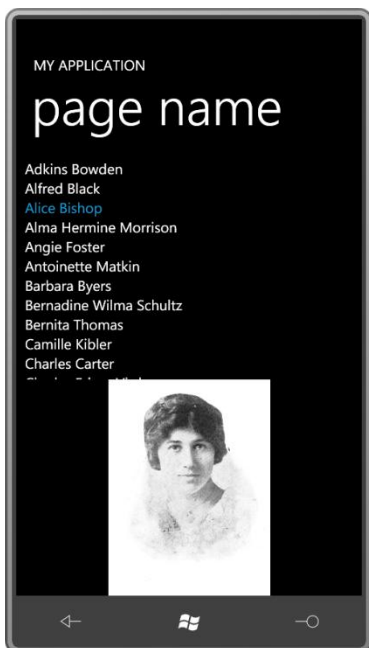


Или заменим *TextBlock* элементом *Image*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListBox Grid.Row="0"
             Name="listBox"
```

```
             ItemsSource="{Binding Students}"
             DisplayMemberPath="FullName" />

    <Image Grid.Row="1"
           HorizontalAlignment="Center"
           Stretch="None"
           Source="{Binding ElementName=listBox,
                            Path=SelectedItem.PhotoFilename}" />
</Grid>
```

Можете теперь выбрать в *ListBox* любой элемент и увидеть фотографию соответствующего учащегося:



Чтобы обеспечить возможность просмотра множества свойств выбранного элемента, потребуется поместить в *Border* еще одно описание *DataContext*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListBox Grid.Row="0"
             Name="listBox"
             ItemsSource="{Binding Students}"
             DisplayMemberPath="FullName" />

    <Border Grid.Row="1"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="{StaticResource PhoneBorderThickness}"
            HorizontalAlignment="Center"
            DataContext="{Binding ElementName=listBox,
                                  Path=SelectedItem}">


    </Border>
</Grid>
```

В этом *Border* могут располагаться панель и элементы с привязками к свойствам класса *Student*. Именно это и было сделано мною в приложении StudentBodyListBox. XAML-файл включает объявление пространства имен XML для библиотеки ElPasoHighSchool:

```
xmlns:elpaso="clr-namespace:ElPasoHighSchool;assembly=ElPasoHighSchool"
```

Экземпляр класса *StudentBodyPresenter* создается в коллекции *Resources*:

Проект Silverlight: StudentBodyListBox   Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />
</phone:PhoneApplicationPage.Resources>
```

Рассмотрим область содержимого:

Проект Silverlight: StudentBodyListBox   Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0"
               Text="{Binding School}"
               FontSize="{StaticResource PhoneFontSizeLarge}"
               HorizontalAlignment="Center"
               TextDecorations="Underline" />

    <ListBox Grid.Row="1"
             Name="listBox"
             ItemsSource="{Binding Students}"
             DisplayMemberPath="FullName" />

    <Border Grid.Row="2"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="{StaticResource PhoneBorderThickness}"
            HorizontalAlignment="Center"
            DataContext="{Binding ElementName=listBox,
                                  Path=SelectedItem}">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <TextBlock Grid.Row="0"
                       Text="{Binding FullName}"
                       TextAlignment="Center" />

            <Image Grid.Row="1"
                   Width="225"
                   Height="300"
                   Margin="24 6"
                   Source="{Binding PhotoFilename}" />

            <StackPanel Grid.Row="2"
                        Orientation="Horizontal"
                        HorizontalAlignment="Center">
```

```
                <TextBlock Text="GPA=" />
                <TextBlock Text="{Binding GradePointAverage}" />
            </StackPanel>
        </Grid>

    </Border>
</Grid>
```

В *Border* располагается *Grid* с тремя строками, в котором размещен *TextBlock* с привязкой к свойству *FullName*, элементу *Image* и *StackPanel* для отображения среднего балла. Обратите внимание, что я задал элементу *Image* конкретные размеры, исходя из известных размеров предоставляемых изображений. Это обеспечит неизменность размера элемента *Image*.

Теперь мы можем прокручивать список *ListBox* и просматривать сведения каждого учащегося:

Немного терпения и вскоре мы сможем видеть изменения среднего балла, что будет обеспечено мощью событий *INotifyPropertyChanged* и свойств-зависимостей в действии.

Эти замечательные шаблоны данных

Оставшуюся часть данной главы я хочу посвятить *ItemsControl* и сосредоточиться исключительно на представлении и навигации, отложив в сторону выбор. Для работы нам понадобится новый проект. Включим в него ссылку на библиотеку ElPasoHighSchool и в XAML-файле добавим объявление пространства имен XML для этой библиотеки, и также создадим экземпляр класса *StudentBodyPresenter* в коллекции *Resources*, как в предыдущем приложении. Рассмотрим *ItemsControl*, который помещен в *ScrollViewer*, заполняющий весь *Grid* для содержимого:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <ScrollViewer>
        <ItemsControl ItemsSource="{Binding Students}"
                      DisplayMemberPath="FullName" />
    </ScrollViewer>
```

```
</Grid>
```

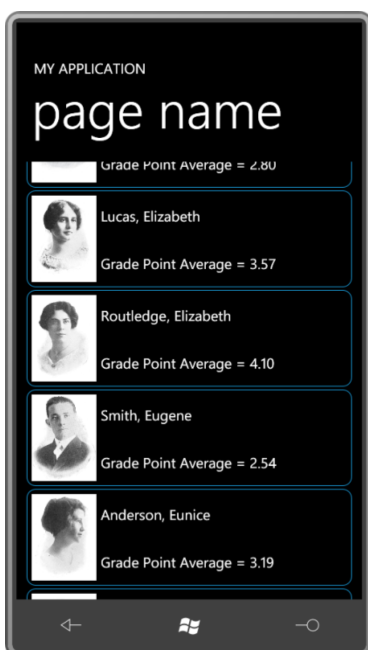
ScrollViewer обеспечивает возможность прокручивать содержимое:

Замена *DisplayMemberPath* на *DataTemplate* позволит обеспечить расширенное и лучше отформатированное отображние сведений:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ScrollViewer>
        <ItemsControl ItemsSource="{Binding Students}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="{StaticResource PhoneAccentBrush}"
                            BorderThickness="1"
                            CornerRadius="12"
                            Margin="2">
                        <Grid>
                            <Grid.RowDefinitions>
                                <RowDefinition Height="*" />
                                <RowDefinition Height="*" />
                            </Grid.RowDefinitions>

                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="Auto" />
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>

                            <Image Grid.Row="0" Grid.Column="0" Grid.RowSpan="2"
                                   Source="{Binding PhotoFilename}"
                                   Height="120"
                                   Width="90"
                                   Margin="6" />

                            <StackPanel Grid.Row="0" Grid.Column="1"
                                        Orientation="Horizontal"
                                        VerticalAlignment="Center">
                                <TextBlock Text="{Binding LastName}" />
                                <TextBlock Text=", " />
                                <TextBlock Text="{Binding FirstName}" />
                                <TextBlock Text=", " />
                                <TextBlock Text="{Binding MiddleName}" />
                            </StackPanel>
```

```
                        <StackPanel Grid.Row="1" Grid.Column="1"
                                    Orientation="Horizontal"
                                    VerticalAlignment="Center">
                            <TextBlock Text="Grade Point Average = " />
                            <TextBlock Text="{Binding GradePointAverage}" />
                        </StackPanel>
                    </Grid>
                </Border>
            </DataTemplate>
        </ItemsControl.ItemTemplate>
    </ItemsControl>
</ScrollViewer>
</Grid>
```

В данном шаблоне высота отдельных элементов зависит от явно заданного параметра *Height* элемента *Image*. Чтобы предотвратить перемещение текста вправо при загрузке изображения, явно задается и параметр *Width*. В результате получаем следующее:



Сортировка

Ранее при выводе на экран данных об учащихся для отображения имени учащегося использовалось свойство *FullName* класса *Student*. Можно заметить, что файл students.xml сортирован по этому свойству, и именно в этом порядке данные учащихся выводятся на экран. Все популярные приложения для электронной почты сортируют контакты по имени, поэтому такой вариант мне показался вполне подходящим.

Но в только что рассмотренном нами *DataTemplate* используются свойства *LastName* (Фамилия), *FirstName* и *MiddleName* (Второе имя). Это приводит к выводу несортированных данных, что выглядит странным и просто неправильным.

Как это исправить?

Один из подходов – посредством кода. Класс *StudentBodyPresenter* может выполнять повторную сортировку данных после их загрузки. Но хотелось бы использовать более гибкое решение, позволяющее применять к данным разные параметры сортировки.

Это можно сделать – причем, полностью в XAML – используя класс *CollectionViewSource* (Источник представления коллекции) из пространства имен *System.Windows.Data*. Этот класс используется в сочетании с классом *SortDescription* (Описание сортировки), описанным в пространстве имен *System.ComponentModel*. Кроме ссылки и объявления пространства имен XML для библиотеки ElPasoHighSchool, нам понадобится объявление пространства имен XML для *System.ComponentModel*:

```
xmlns:componentmodel="clr-namespace:System.ComponentModel;assembly=System.Windows"
```

Весь *CollectionViewSource* может описываться в коллекции *Resources*:

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />

    <CollectionViewSource x:Key="sortedStudents"
                          Source="{Binding Source={StaticResource
studentBodyPresenter},
                                           Path=StudentBody.Students}">
        <CollectionViewSource.SortDescriptions>
            <componentmodel:SortDescription PropertyName="LastName"
                                            Direction="Ascending" />
        </CollectionViewSource.SortDescriptions>
    </CollectionViewSource>

</phone:PhoneApplicationPage.Resources>
```

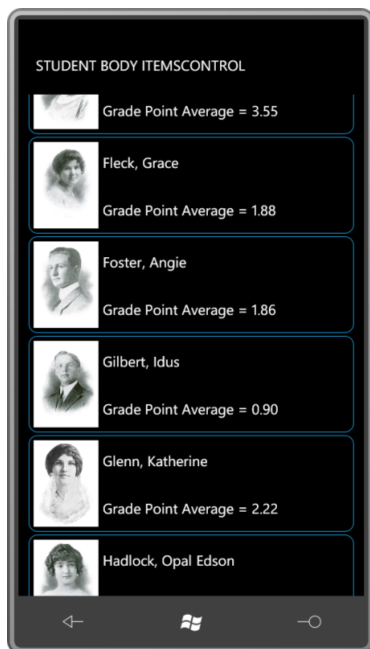
Обратите внимание, теперь свойство *Source* класса *CollectionViewSource* ссылается на свойство *Students* свойства *StudentBody* класса *StudentBodyPresenter*. Свойство *Students* типа *ObservableCollection<Student>*. Свойство *Source* класса *CollectionViewSource* должно быть коллекцией.

Объект *SortDescription* указывает на то, что мы хотим выполнять сортировку по свойству *LastName* в порядке по возрастанию. Поскольку свойство *LastName* типа *string*, никакого дополнительного кода для описания сортировки не требуется.

Теперь можно убрать *Binding* из *DataContext* класса *Grid*, и свойство *Source* класса *ItemsControl* может ссылаться на ресурс *CollectionViewSource*:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ScrollViewer>
        <ItemsControl ItemsSource="{Binding Source={StaticResource sortedStudents}}">
        …
        </ItemsControl>
    </ScrollViewer>
</Grid>
```

В результате данные выводятся на экран в алфавитном порядке, что выглядит более привлекательно:



CollectionViewSource может включать несколько объектов *SortDescription*. Попробуем сделать следующее:

```
<CollectionViewSource x:Key="sortedStudents"
                      Source="{Binding Source={StaticResource studentBodyPresenter},
                                       Path=StudentBody.Students}">
    <CollectionViewSource.SortDescriptions>
        <componentmodel:SortDescription PropertyName="Sex"
                                        Direction="Ascending" />
        <componentmodel:SortDescription PropertyName="LastName"
                                        Direction="Ascending" />
    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

Теперь сначала выводятся данные всех женщин, а затем всех мужчин.

Или можно выводить имена учащихся мужского пола цветом *PowderBlue* (Зеленовато-голубой) и имена учащихся женского пола – цветом *Pink* (Розовый). Это довольно старомодная условность, но все-таки мы имеем дело с данными учащихся, которые посещали школу почти 100 лет назад! В общем, совершенно не важно, какие цвета использовать, но вот как реализовать это?

К счастью, у класса *Student* есть свойство *Sex* (Пол), значением которого является текстовая строка, либо «Male» (Мужской), либо «Female» (Женский). Поскольку в *DataTemplate* мы имеем дело с привязками данных, очевидным решением является применение конвертера данных. В библиотеке Petzold.Phone.Silverlight как раз есть такой конвертер, который идеально подходит для этого случая:

Проект Silverlight: Petzold.Phone.Silverlight Файл: SexToBrushConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;
using System.Windows.Media;

namespace Petzold.Phone.Silverlight
{
    public class SexToBrushConverter : IValueConverter
    {
```

```
        public Brush MaleBrush { get; set; }
        public Brush FemaleBrush { get; set; }

        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            string sex = value as string;

            switch (sex)
            {
                case "Male": return MaleBrush;
                case "Female": return FemaleBrush;
            }

            return null;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return null;
        }
    }
}
```

Как все конвертеры данных, наш конвертер наследуется от *IValueConverter* и имеет два метода: *Convert* и *ConvertBack*. Этот конвертер также определяет два свойства: *MaleBrush* (Кисть для Male) и *FemaleBrush* (Кисть для Female). Эти свойства позволяют нам избежать задания кистей в коде. Здесь реализован только метод *Convert*: при получении значения «Male» он возвращает *MaleBrush* и при получении значения «Female» – *FemaleBrush*.

Теперь сведем все в один проект. Кроме библиотеки ElPasoHighSchool, проект StudentBodyItemsControl также включает ссылку на библиотеку Petzold.Phone.Silverlight. В разделе *Resources* создаются экземпляры классов *StudentBodyPresenter*, *CollectionViewSource* для сортировки и *SexToBrushConverter* (Конвертер пола в кисть):

Проект Silverlight: StudentBodyItemsControl Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />

    <CollectionViewSource x:Key="sortedStudents"
                          Source="{Binding Source={StaticResource
studentBodyPresenter},
                                           Path=StudentBody.Students}">
        <CollectionViewSource.SortDescriptions>
            <componentmodel:SortDescription PropertyName="LastName"
                                            Direction="Ascending" />
        </CollectionViewSource.SortDescriptions>
    </CollectionViewSource>

    <petzold:SexToBrushConverter x:Key="sexToBrushConverter"
                                 FemaleBrush="Pink"
                                 MaleBrush="PowderBlue" />
</phone:PhoneApplicationPage.Resources>
```

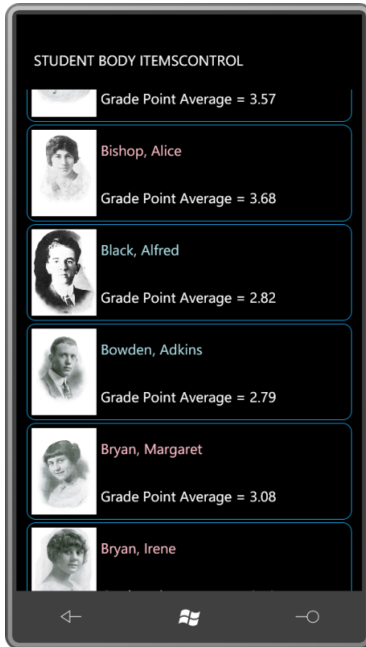
В приведенной ниже разметке я использовал пять элементов *TextBlock* для отображения полного имени учащегося – *LastName*, *FirstName* и *MiddleName* с запятой и пробелом – и по крайней мере четырем из них необходимы привязки, связывающие свойство *Foreground* со свойством *Sex* объекта *Student*, в которых используется *SexToBrushConverter*. Одну и ту же привязку необходимо применить четыре раза.

Или, вероятно, мы можем немного упростить разметку, заключив все пять элементов *TextBlock* в *ContentControl*. Если свойство *Foreground* класса *ContentControl* задается одной привязкой, это же свойство наследуется всеми *TextBlock*. Это и реализуется в следующем *DataTemplate*, который во всем остальном полностью повторяет предыдущий *DataTemplate*.

Проект Silverlight: StudentBodyItemsControl Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ScrollViewer>
        <ItemsControl ItemsSource="{Binding
                                    Source={StaticResource sortedStudents}}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="{StaticResource PhoneAccentBrush}"
                            BorderThickness="1"
                            CornerRadius="12"
                            Margin="2">
                        <Grid>
                            <Grid.RowDefinitions>
                                <RowDefinition Height="*" />
                                <RowDefinition Height="*" />
                            </Grid.RowDefinitions>

                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="Auto" />
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>

                            <Image Grid.Row="0" Grid.Column="0" Grid.RowSpan="2"
                                   Source="{Binding PhotoFilename}"
                                   Height="120"
                                   Width="90"
                                   Margin="6" />

                            <ContentControl Grid.Row="0" Grid.Column="1"
                                            HorizontalAlignment="Left"
                                            VerticalAlignment="Center"
                              Foreground="{Binding Sex,
                                Converter={StaticResource sexToBrushConverter}}">

                                <StackPanel Orientation="Horizontal">
                                    <TextBlock Text="{Binding LastName}" />
                                    <TextBlock Text=", " />
                                    <TextBlock Text="{Binding FirstName}" />
                                    <TextBlock Text=", " />
                                    <TextBlock Text="{Binding MiddleName}" />
                                </StackPanel>
                            </ContentControl>

                            <StackPanel Grid.Row="1" Grid.Column="1"
                                        Orientation="Horizontal"
                                        VerticalAlignment="Center">
                                <TextBlock Text="Grade Point Average = " />
                                <TextBlock Text="{Binding GradePointAverage}" />
                            </StackPanel>
                        </Grid>
                    </Border>
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
    </ScrollViewer>
</Grid>
```

Полагаю, эффект от добавления цвета стоит затраченных на его реализацию усилий:



Замена панели

Как было показано ранее в этой главе при рассмотрении деревьев визуальных элементов, для отображения всех элементов *ItemsControl* использует *ItemsPresenter*. Одним из основных элементов отображения списков является панель. По умолчанию используется *StackPanel* (или *VirtualizingStackPanel* для *ListBox*) с вертикальной ориентацией. Вертикальная стек-панель является таким естественным выбором для реализации этой задачи, что менять ее на что-то другое просто не приходит в голову.

Но заменить ее можно. Для этого в качестве значения свойства *ItemsPanel* класса *ItemsControl* задается другой шаблон. Обычно он предельно прост.

Проект HorizontalItemsControl (Горизонтальный элемент управления списками) во многом повторяет предыдущий проект. В нем также есть ссылки и объявления пространств имен для Petzold.Silverlight.Phone и ElPasoHighSchool и абсолютно идентичная коллекция *Resources*. Отличие в применение *StackPanel* с горизонтальной ориентацией в *ItemsControl*. *DataTemplate* для учащихся в данном приложении также довольно сильно отличается, в нем за базовую взята альбомная ориентация телефона.

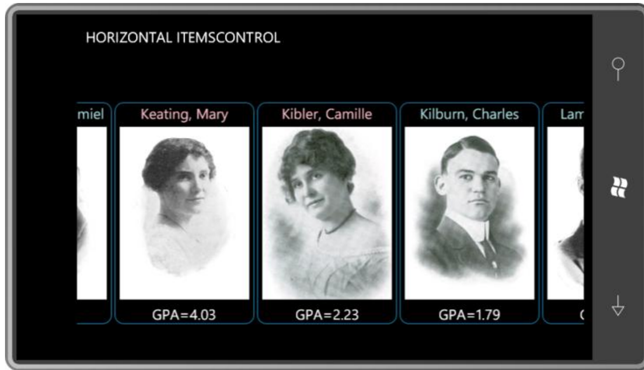
Поскольку *ItemsControl* теперь отображает элементы горизонтально, а не вертикально, поведение *ScrollViewer* по умолчанию необходимо полностью изменить. *ScrollViewer* должен обеспечивать прокрутку в горизонтальном направлении:

Проект Silverlight: HorizontalItemsControl Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

    <ScrollViewer VerticalAlignment="Center"
                  HorizontalScrollBarVisibility="Auto"
                  VerticalScrollBarVisibility="Disabled">
        <ItemsControl ItemsSource="{Binding
                                    Source={StaticResource sortedStudents}}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="{StaticResource PhoneAccentBrush}"
                            BorderThickness="1"
```

```
                            CornerRadius="12"
                            Margin="2">
                        <Grid>
                            <Grid.RowDefinitions>
                                <RowDefinition Height="Auto" />
                                <RowDefinition Height="Auto" />
                                <RowDefinition Height="Auto" />
                            </Grid.RowDefinitions>

                            <ContentControl Grid.Row="0"
                                            HorizontalAlignment="Center"
                              Foreground="{Binding Sex,
                                Converter={StaticResource sexToBrushConverter}}">

                                <StackPanel Orientation="Horizontal">
                                    <TextBlock Text="{Binding LastName}" />
                                    <TextBlock Text=", " />
                                    <TextBlock Text="{Binding FirstName}" />
                                    <TextBlock Text=", " />
                                    <TextBlock Text="{Binding MiddleName}" />
                                </StackPanel>
                            </ContentControl>

                            <Image Grid.Row="1"
                                   Source="{Binding PhotoFilename}"
                                   Height="240"
                                   Width="180"
                                   Margin="6" />

                            <StackPanel Grid.Row="2"
                                        Orientation="Horizontal"
                                        HorizontalAlignment="Center">
                                <TextBlock Text="GPA=" />
                                <TextBlock Text="{Binding GradePointAverage}" />
                            </StackPanel>
                        </Grid>
                    </Border>
                </DataTemplate>
            </ItemsControl.ItemTemplate>

            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <StackPanel Orientation="Horizontal" />
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>

        </ItemsControl>
    </ScrollViewer>
</Grid>
```

В конце разметки *ItemsControl* можно увидеть свойство *ItemsPanel*. В качестве его значения задан объект типа *ItemsPanelTemplate*, включающий производный от *Panel*, который мы хотим использовать.



Не все типы панелей подходят для элементов управления списками. Как правило, используется панель, дочерние элементы которой организовываться на основании порядка их расположения в коллекции *Children*, а не на основании присоединенных свойств.

Часто разработчики создают пользовательские панели для *ListBox* или *ItemsControl*. Иногда эти панели приобретают даже форму каруселей. В конце этой главы я приведу пример пользовательской панели для отображения учащихся.

Построение гистограммы при помощи *DataTemplate*

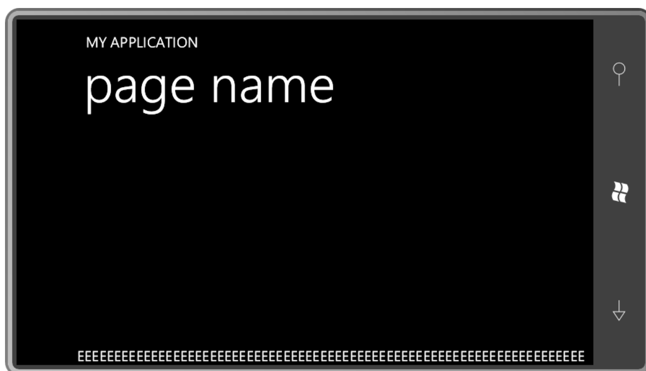
Сочетая *DataTemplate* и *ItemsPanelTemplate*, можно получить *ListBox* или *ItemsControl*, не похожий ни на что ранее виденное.

Создадим новый проект и включим в него ссылки и объявления пространств имен XML для библиотек Petzold.Phone.Silverlight и ElPasoHighSchool. В корневом теге файла MainPage.xaml зададим свойства для обеспечения альбомной ориентации. Поместим *StudentBodyPresenter* в коллекцию *Resources*.

Это *ItemsControl* без *ScrollViewer*. *ItemsSource* – это свойство *Students* экземпляра *StudentBodyPresenter*. Для *ItemsPanelTemplate* задан *UniformStack* с горизонтальной ориентацией:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <ItemsControl ItemsSource="{Binding Students}"
                  VerticalAlignment="Bottom">

        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <petzold:UniformStack Orientation="Horizontal" />
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
    </ItemsControl>
</Grid>
```

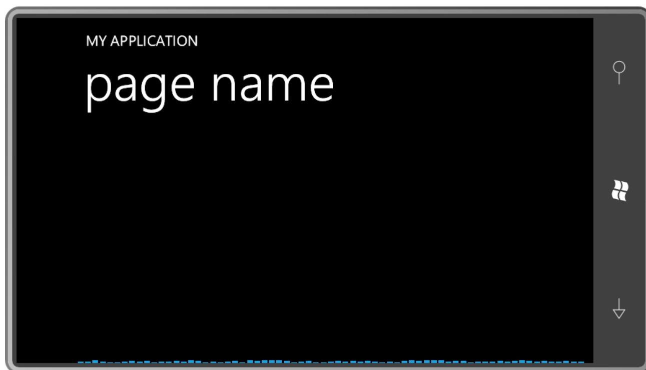
Без *DataTemplate* *ItemsControl* выводит на экран полное имя класса в виде строки «ElPasoHighSchool.Student». Но в панели *UniformStack* для каждого элемента выделяется одинаковое пространство, поэтому видимым остается только первая «E»:



Выглядит малообещающим, но для *DataTemplate* зададим *Rectangle*, свойство *Height* которого связано посредством привязки со свойством *GradePointAverage*:

```
<ItemsControl ItemsSource="{Binding Students}"
              VerticalAlignment="Bottom">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Rectangle Fill="{StaticResource PhoneAccentBrush}"
                       Height="{Binding GradePointAverage}"
                       VerticalAlignment="Bottom"
                       Margin="1 0" />
        </DataTemplate>
    </ItemsControl.ItemTemplate>

    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <petzold:UniformStack Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
</ItemsControl>
```

ItemsControl выравнивается по низу экрана и каждый *Rectangle* выравнивается по низу *ItemsControl*. В результате получаем гистограмму:



Безусловно, значения свойства *GradePointAverage* лежат в диапазоне от 0 до 5, поэтому столбики диаграммы очень малы. Как решить эту проблему?

Первым в голову приходит применить к *Rectangle* трансформацию *ScaleTransform* с постоянным коэффициентом масштабирования по вертикали равным, скажем, 50. Я сначала так и сделал, но не был доволен результатом. Кажется, перед масштабированием высоты прямоугольников были округлены до ближайшего пиксела. Поэтому я отказался от такого подхода и написал новый конвертер данных:

Проект Silverlight: Petzold.Phone.Silverlight Файл: MultiplyConverter.cs

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Petzold.Phone.Silverlight
{
    public class MultiplyConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            double multiplier;

            if (value is IConvertible &&
                    parameter is string &&
                        Double.TryParse(parameter as string, out multiplier))
            {
                return (value as IConvertible).ToDouble(culture) * multiplier;
            }
            return value;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            double divider;

            if (value is IConvertible &&
                    parameter is string &&
                        Double.TryParse(parameter as string, out divider))
            {
                return (value as IConvertible).ToDouble(culture) / divider;
            }
            return value;
        }
    }
}
```

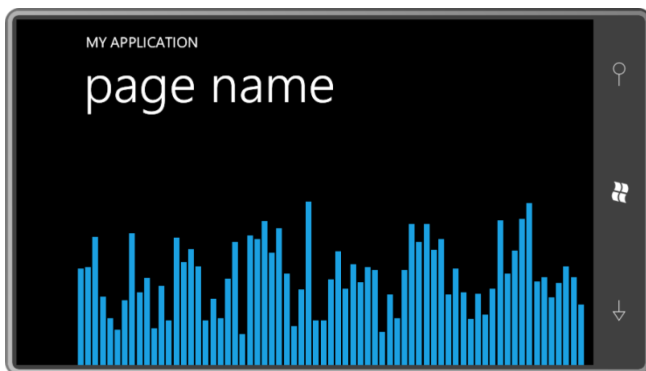
Этот конвертер умножает значение источника привязки на коэффициент, заданный как параметр конвертера. Определим один из них в коллекции *Resources*:

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />
    <petzold:MultiplyConverter x:Key="multiply" />
</phone:PhoneApplicationPage.Resources>
```

Теперь дадим ссылку на конвертер в привязке, чтобы обеспечить умножение каждого значения на 50:

```
<DataTemplate>
    <Rectangle Fill="{StaticResource PhoneAccentBrush}"
               Height="{Binding GradePointAverage,
                                Converter={StaticResource multiply},
                                ConverterParameter=50}"
               VerticalAlignment="Bottom"
               Margin="1 0" />
</DataTemplate>
```

И теперь все выглядит, как настоящая гистограмма:



Более того, поскольку значения *GradePointAverage* меняются динамически, столбики гистограммы меняют свою высоту.

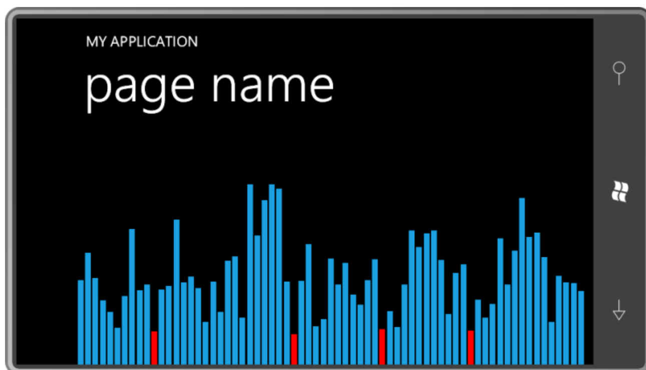
Помните конвертер *ValueToBrushConverter* из библиотеки Petzold.Phone.Silverlight? С его помощью мы можем задать цветовой код для гистограммы и получать уведомления цветом, когда средний балл любого из учащихся опускается ниже 1, к примеру. Так конвертер выглядел бы в коллекции *Resources*:

```
<petzold:ValueToBrushConverter x:Key="valueToBrush"
                               Criterion="1"
                               GreaterThanBrush="{StaticResource PhoneAccentBrush}"
                               EqualToBrush="{StaticResource PhoneAccentBrush}"
                               LessThanBrush="Red" />
```

Вот новый *DataTemplate*:

```
<DataTemplate>
    <Rectangle Fill="{Binding GradePointAverage,
                              Converter={StaticResource valueToBrush}}"

               Height="{Binding GradePointAverage,
                                Converter={StaticResource multiply},
                                ConverterParameter=50}"
               VerticalAlignment="Bottom"
               Margin="1 0" />
</DataTemplate>
```

Классный руководитель этих учащихся был бы рад данной возможности, поскольку с ее помощью мог бы без труда видеть, кто из учеников нуждается в особом внимании:



Можно ли определить, кому из учащихся соответствует тот или иной столбик гистограммы?

Один из возможных подходов реализован в проекте GpaBarChart. Он включает класс *StudentBodyPresenter* и два упоминаемых мною конвертера, которые определены как ресурсы:

Проект Silverlight: GpiBarChart Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />
    <petzold:MultiplyConverter x:Key="multiply" />
    <petzold:ValueToBrushConverter x:Key="valueToBrush"
                                   Criterion="1"
                                   GreaterThanBrush="{StaticResource PhoneAccentBrush}"
                                   EqualToBrush="{StaticResource PhoneAccentBrush}"
                                   LessThanBrush="Red" />
</phone:PhoneApplicationPage.Resources>
```

Область содержимого, преимущественно, осталась прежней, я лишь добавил *Border* с именем «studentDisplay» (Монитор учащихся), который будет размещаться вверху экрана. Этот *Border* включает пару элементов *TextBlock*, свойства *Text* которых связаны посредством привязки данных со свойствами *FullName* и *GradePointAverage*, исходя из предположения о том, что *DataContext* этого *Border* является объектом типа *Student*. Как правило, это не так, поэтому свойству *Visibility* нашего *Border* задано начальное значение *Collapsed*:

Проект Silverlight: GpiBarChart Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">

    <Border x:Name="studentDisplay"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="{StaticResource PhoneBorderThickness}"
            HorizontalAlignment="Center"
            VerticalAlignment="Top"
            Margin="24"
            Padding="12"
            CornerRadius="24"
            Visibility="Collapsed">
        <StackPanel>
            <TextBlock Text="{Binding FullName}"
                       HorizontalAlignment="Center" />
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="GPA = " />
                <TextBlock Text="{Binding GradePointAverage}" />
            </StackPanel>
        </StackPanel>
    </Border>

    <ItemsControl ItemsSource="{Binding Students}"
                  VerticalAlignment="Bottom">
        <ItemsControl.ItemTemplate>
            <DataTemplate>
                <Rectangle Fill="{Binding GradePointAverage,
                                  Converter={StaticResource valueToBrush}}"

                           Height="{Binding GradePointAverage,
                                    Converter={StaticResource multiply},
                                    ConverterParameter=50}"
                           VerticalAlignment="Bottom"
                           Margin="1 0" />
            </DataTemplate>
        </ItemsControl.ItemTemplate>

        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <petzold:UniformStack Orientation="Horizontal" />
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
```

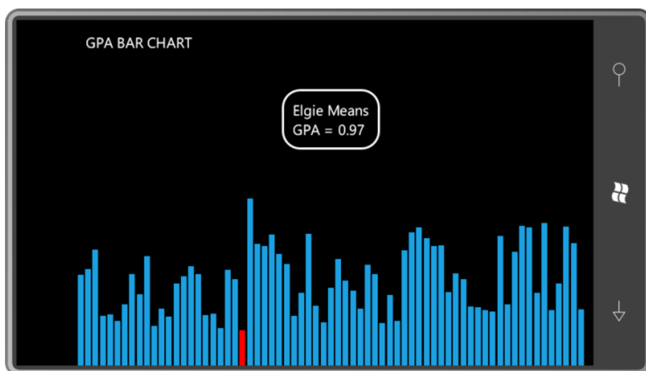
```
    </ItemsControl>
</Grid>
```

Файл выделенного кода восполняет недостающую логику. Страница обрабатывает событие *Touch.FrameReported*. Если *Rectangle* находится непосредственно под главной точкой касания, обработчик события получает *DataContext* этого *Rectangle*. Это объект типа *Student*. Далее этот объект задается как значение *DataContext* объекта *Border*. Свойство *TouchAction* используется для переключения значения *Visibility*:

Проект Silverlight: GpiBarChart Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        Touch.FrameReported += OnTouchFrameReported;
    }

    void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
    {
        TouchPoint touchPoint = args.GetPrimaryTouchPoint(this);

        if (touchPoint != null && touchPoint.Action == TouchAction.Down)
            args.SuspendMousePromotionUntilTouchUp();

        if (touchPoint != null && touchPoint.TouchDevice.DirectlyOver is Rectangle)
        {
            Rectangle rectangle =
                    (touchPoint.TouchDevice.DirectlyOver as Rectangle);

            // Этот DataContext является объектом типа Student
            object dataContext = rectangle.DataContext;
            studentDisplay.DataContext = dataContext;

            if (touchPoint.Action == TouchAction.Down)
                studentDisplay.Visibility = Visibility.Visible;

            else if (touchPoint.Action == TouchAction.Up)
                studentDisplay.Visibility = Visibility.Collapsed;
        }
    }
}
```

Проводя пальцем по столбикам, можно видеть, какого учащегося каждый из них представляет:



Картотека

В рассмотренном нами приложении GpiBarChart нам, в некотором смысле, удалось вместить данные всех студентов в один экран, но при этом, конечно же, предоставляемые сведения ограничены. Можно ли выводить больше информации на экран? Популярной метафорой для отображения данных является картотека. Изначально все карточки отображаются лишь частично, но при этом приложение обеспечивает возможность просматривать каждую из карточек полностью.

На этапе подготовки к реализации этой задачи, я добавил в библиотеку Petzold.Phone.Silverlight новую панель. Отчасти эта панель аналогична панели *UniformStack*, обсуждаемой в главе 9. Как и *UniformStack*, эта новая панель выделяет всем своим дочерним элементам равное пространство. Но в отличие от *UniformStack* дочерние элементы этой панели перекрывают друг друга, если пространства панели не достаточно для отображения их полностью. Поэтому я назвал эту панель *OverlapPanel* (Панель с перекрытием).

OverlapPanel определяет свойство *Orientation* и расставляет свои дочерние элементы по вертикали или по горизонтали. Если *OverlapPanel* расставляет дочерние элементы по горизонтали, каждый последующий дочерний элемент размещается поверх и несколько правее предыдущего, оставляя видимым лишь узкую полоску с левого края. Для вертикальной ориентации видимой остается верхушка каждого дочернего элемента.

Если дочерних элементов очень много, видимая полоска становится очень тонкой. Чтобы сделать *OverlapPanel* более полезной, необходимо обеспечить возможность задания минимальной высоты или ширины видимой части перекрываемого элемента, даже если это будет приводить к выходу панели за границы выделенного для нее пространства. В случае выхода за допустимые границы поведение *OverlapPanel* во многом аналогично обычному *StackPanel*:для просмотра всех элементов понадобится *ScrollViewer*.

OverlapPanel определяет два свойства: *Orientation* и *MinimumOverlap* (Минимальное наложение):

Проект Silverlight: Petzold.Phone.Silverlight   Файл: OverlapPanel.cs (фрагмент)

```
public class OverlapPanel : Panel
{
    Size maxChildSize = new Size();

    public static readonly DependencyProperty OrientationProperty =
        DependencyProperty.Register("Orientation",
            typeof(Orientation),
            typeof(OverlapPanel),
            new PropertyMetadata(Orientation.Horizontal, OnAffectsMeasure));

    public static readonly DependencyProperty MinimumOverlapProperty =
        DependencyProperty.Register("MinimumOverlap",
            typeof(double),
            typeof(OverlapPanel),
            new PropertyMetadata(0.0, OnAffectsMeasure));

    public Orientation Orientation
    {
        set { SetValue(OrientationProperty, value); }
        get { return (Orientation)GetValue(OrientationProperty); }
    }

    public double MinimumOverlap
    {
        set { SetValue(MinimumOverlapProperty, value); }
        get { return (double)GetValue(MinimumOverlapProperty); }
```

```
    }

    static void OnAffectsMeasure(DependencyObject obj,
                                 DependencyPropertyChangedEventArgs args)
    {
        (obj as OverlapPanel).InvalidateMeasure();
    }
    …
}
```

Изменение значения любого из этих двух свойств приводит к вызову метода *InvalidateMeasure*, который инициирует новый пересчет размеров элементов компоновки.

Метод *MeasureOverride* сначала проходит по всем дочерним элементам для получения их максимального размера. Для *OverlapPanel* с *ItemsControl* или *ListBox* все дочерние элементы будут, вероятнее всего, одного размера.

Проект Silverlight: Petzold.Phone.Silverlight Файл: OverlapPanel.cs (фрагмент)

```
protected override Size MeasureOverride(Size availableSize)
{
    if (Children.Count == 0)
        return new Size(0, 0);

    maxChildSize = new Size();

    foreach (UIElement child in Children)
    {
        if (Orientation == Orientation.Horizontal)
            child.Measure(new Size(Double.PositiveInfinity, availableSize.Height));
        else
            child.Measure(new Size(availableSize.Width, Double.PositiveInfinity));

        maxChildSize.Width = Math.Max(maxChildSize.Width,
                                      child.DesiredSize.Width);

        maxChildSize.Height = Math.Max(maxChildSize.Height,
                                       child.DesiredSize.Height);
    }

    if (Orientation == Orientation.Horizontal)
    {
        double maxTotalWidth = maxChildSize.Width * Children.Count;
        double minTotalWidth = maxChildSize.Width +
                               MinimumOverlap * (Children.Count - 1);

        if (Double.IsPositiveInfinity(availableSize.Width))
            return new Size(minTotalWidth, maxChildSize.Height);

        if (maxTotalWidth < availableSize.Width)
            return new Size(maxTotalWidth, maxChildSize.Height);

        else if (minTotalWidth < availableSize.Width)
            return new Size(availableSize.Width, maxChildSize.Height);

        return new Size(minTotalWidth, maxChildSize.Height);
    }
    // Orientation = Vertical
    double maxTotalHeight = maxChildSize.Height * Children.Count;
    double minTotalHeight = maxChildSize.Height +
                            MinimumOverlap * (Children.Count - 1);

    if (Double.IsPositiveInfinity(availableSize.Height))
        return new Size(maxChildSize.Width, minTotalHeight);
```

```
    if (maxTotalHeight < availableSize.Height)
        return new Size(maxChildSize.Width, maxTotalHeight);

    else if (minTotalHeight < availableSize.Height)
        return new Size(maxChildSize.Width, availableSize.Height);

    return new Size(maxChildSize.Width, minTotalHeight);
}
```

После этого метод разделяется на две ветви в зависимости от значения свойства *Orientation*. Например, для вертикальной ориентации (которая будет использоваться в примере ниже) метод вычисляет *maxTotalHeight* (Максимальная общая высота), которая соответствует сумме высот всех дочерних элементов без наложения, и *minTotalHeight* (Минимальная общая высота), которая соответствует суммарной высоте всех элементов при максимально допустимом наложении. Если доступная высота не бесконечна (эта возможность обрабатывается отдельно), она либо больше *maxTotalHeight*, либо находится в диапазоне между *minTotalHeight* и *maxTotalHeight*, либо меньше *minTotalHeight*. Если все дочерние элементы могут поместиться в предоставляемом пространстве, располагаясь вплотную друг к другу без наложения, запрашивается такой размер. Но метод никогда не запрашивает высоту, меньшую, чем требуется для отображения всех дочерних элементов.

Метод *ArrangeOverride* несколько проще. Значение *приращения* соответствует минимально допустимой ширине или высоте видимой части дочернего элемента:

Проект Silverlight: Petzold.Phone.Silverlight Файл: OverlapPanel.cs (фрагмент)

```
protected override Size ArrangeOverride(Size finalSize)
{
    if (Children.Count == 0)
        return finalSize;

    double increment = 0;

    if (Orientation == Orientation.Horizontal)
        increment = Math.Max(MinimumOverlap,
            (finalSize.Width - maxChildSize.Width) / (Children.Count - 1));
    else
        increment = Math.Max(MinimumOverlap,
            (finalSize.Height - maxChildSize.Height) / (Children.Count - 1));

    Point ptChild = new Point();

    foreach (UIElement child in Children)
    {
        child.Arrange(new Rect(ptChild, maxChildSize));

        if (Orientation == Orientation.Horizontal)
            ptChild.X += increment;
        else
            ptChild.Y += increment;
    }

    return finalSize;
}
```

Проект StudentCardFile (Картотека учащихся) включает ссылки на библиотеки Petzold.Phone.Silverlight и ElPasoHighSchool. Файл MainPage.xaml включает класс *StudentBodyPresenter* в коллекции *Resources*:

Проект Silverlight: StudentCardFile Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <elpaso:StudentBodyPresenter x:Key="studentBodyPresenter" />
</phone:PhoneApplicationPage.Resources>
```

Область содержимого довольно проста и включает только *ScrollViewer* и *ItemsControl*. Свойство *ItemsPanel* класса *ItemsControl* ссылается на *OverlapPanel* с двумя заданными свойствами:

Проект Silverlight: StudentCardFile Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
      DataContext="{Binding Source={StaticResource studentBodyPresenter},
                            Path=StudentBody}">
    <ScrollViewer>
        <ItemsControl ItemsSource="{Binding Students}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <local:StudentCard />
                </DataTemplate>
            </ItemsControl.ItemTemplate>

            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <petzold:OverlapPanel Orientation="Vertical"
                                          MinimumOverlap="24" />
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>
        </ItemsControl>
    </ScrollViewer>
</Grid>
```

Простота разметки здесь является, главным образом, результатом задания элемента управления *StudentCard* в качестве значения свойства *DataTemplate* класса *ItemsControl*.

StudentCard наследуется от *UserControl*. Наследование от *UserControl* является традиционной методикой создания элемента управления, который будет использоваться как *DataTemplate*. Если в приведенном ниже фрагменте не обращать внимания на многоточия (…), мы имеем здесь довольно простой набор элементов *TextBlock* и *Image* со свернутым *Rectangle*, который используется как разделительная линия:

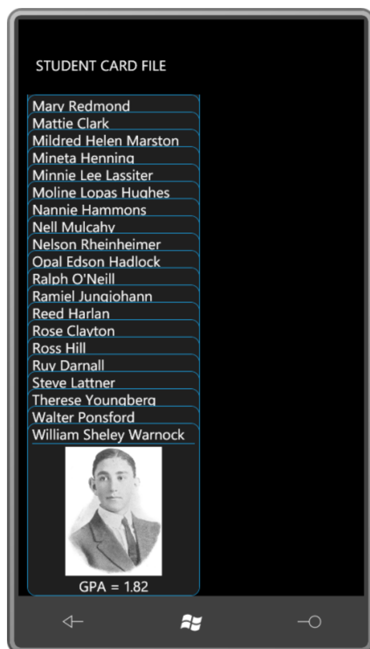
Проект Silverlight: StudentCardFile Файл: StudentCard.xaml (фрагмент)

```
<UserControl x:Class="StudentCardFile.StudentCard"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             FontFamily="{StaticResource PhoneFontFamilyNormal}"
             FontSize="{StaticResource PhoneFontSizeNormal}"
             Foreground="{StaticResource PhoneForegroundBrush}"
             Width="240" Height="240">

    …

    <Border BorderBrush="{StaticResource PhoneAccentBrush}"
            BorderThickness="1"
            Background="{StaticResource PhoneChromeBrush}"
            CornerRadius="12"
            Padding="6 0">

        …

        <Grid>
            <Grid.RowDefinitions>
```

```
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="*" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <TextBlock Grid.Row="0"
                       Text="{Binding FullName}" />

            <Rectangle Grid.Row="1"
                       Fill="{StaticResource PhoneAccentBrush}"
                       Height="1"
                       Margin="0 0 0 4" />

            <Image Grid.Row="2"
                   Source="{Binding PhotoFilename}" />

            <StackPanel Grid.Row="3"
                        Orientation="Horizontal"
                        HorizontalAlignment="Center">
                <TextBlock Text="GPA = " />
                <TextBlock Text="{Binding GradePointAverage}" />
            </StackPanel>
        </Grid>
    </Border>
</UserControl>
```

Карточки выложены сверху вниз в левой части экрана, но видна только верхняя часть каждой из них. Удобно, что у каждой карточки сверху располагается *TextBlock* с именем учащегося:

Я задал для *MinimumOverlap* значение, обеспечивающее видимость этого *TextBlock*. Если прокрутить весь список вниз, мы увидим, что самая нижняя карточка видна полностью:



Это было бы замечательно, если бы требовалось просматривать только последнюю карточку. Нам же необходимо выводить на экран любую карточку, выбираемую пользователем. Одним из подходов может быть изменение присоединенного свойства *Canvas.ZIndex* выбранной карточки. Или можно изменять порядок всех карточек в колоде, обеспечивая перемещение выбранной карточки и ее расположение поверх остальных.

Я решил, что выбранная карточка при касании будет выезжать из колоды и затем возвращаться на место при повторном касании или при выборе другой карточки.

При интеграции *ScrollViewer* с остальным кодом обнаруживается, что *ScrollViewer* перехватывает события *Manipulation*. Безусловно, *ScrollViewer* использует эти события *Manipulation* для собственной логики прокрутки, но это мешает дочерним визуальным элементам *ScrollViewer* (таким как элементы *StudentCard*) обрабатывать события *Manipulation* для реализации собственного перемещения при выборе.

Поэтому я решил добавить в *StudentCard* обработчик низкоуровневого события *Touch.FrameReported* и использовать его для переключения значения свойства-зависимости *IsOpen*. Рассмотрим это свойство в файле выделенного кода *StudentCard*:

Проект Silverlight: StudentCardFile   Файл: StudentCard.xaml.cs (фрагмент)

```
public partial class StudentCard : UserControl
{
    …
    public static readonly DependencyProperty IsOpenProperty =
        DependencyProperty.Register("IsOpen",
            typeof(bool),
            typeof(StudentCard),
            new PropertyMetadata(false, OnIsOpenChanged));
    …
    bool IsOpen
    {
        set { SetValue(IsOpenProperty, value); }
        get { return (bool)GetValue(IsOpenProperty); }
    }
    …
}
```

Чуть ниже я покажу обработчик события изменения значения свойства для *IsOpen*.

При касании одного из экземпляров *StudentCard* он должен выдвигаться из колоды карточек. Но если в этот момент раскрыта другая карточка, она должна быть задвинута назад в колоду. Если класс *CardFile* должен обрабатывать эту логику самостоятельно, каждому экземпляру *CardFile* необходим доступ ко всем остальным экземплярам. Поэтому для работы с этими экземплярами я определил статическое поле типа *List*:

Проект Silverlight: StudentCardFile Файл: StudentCard.xaml.cs (фрагмент)

```
public partial class StudentCard : UserControl
{
    static List<StudentCard> studentCards = new List<StudentCard>();
    …
    public StudentCard()
    {
        InitializeComponent();
        studentCards.Add(this);
    }
    …
}
```

Каждый новый экземпляр просто добавляет себя в коллекцию.

Я понял, что каждый отдельный экземпляр *StudentCard* не нуждается в собственном обработчике событий *Touch.FrameReported*. Все экземпляры могут использовать совместно один статический обработчик, заданный в статическом конструкторе и ссылающийся на статические поля:

Проект Silverlight: StudentCardFile Файл: StudentCard.xaml.cs (фрагмент)

```
public partial class StudentCard : UserControl
{
    …
    static int contactTime;
    static Point contactPoint;
    …
    static StudentCard()
    {
        Touch.FrameReported += OnTouchFrameReported;
    }
    …
    static void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
    {
        TouchPoint touchPoint = args.GetPrimaryTouchPoint(null);

        if (touchPoint != null && touchPoint.Action == TouchAction.Down)
        {
            contactPoint = touchPoint.Position;
            contactTime = args.Timestamp;
        }
        else if (touchPoint != null && touchPoint.Action == TouchAction.Up)
        {
            // Проверяем, приходится ли касание непосредственно
            // на StudentCard или дочерний элемент
            DependencyObject element = touchPoint.TouchDevice.DirectlyOver;

            while (element != null && !(element is StudentCard))
                element = VisualTreeHelper.GetParent(element);

            if (element == null)
                return;
```

```
            // Получаем точку снятия касания и вычисляем разницу
            Point liftPoint = touchPoint.Position;
            double distance = Math.Sqrt(Math.Pow(contactPoint.X - liftPoint.X, 2) +
                                        Math.Pow(contactPoint.Y - liftPoint.Y, 2));

            // Распознаем прикосновение как Tap, если
            // пройденное расстояние < 12 пикселов за 1/4 секунды
            if (distance < 12 && args.Timestamp - contactTime < 250)
            {
                // Выполняем перечисление объектов StudentCard и задаем свойство
IsOpen
                foreach (StudentCard studentCard in studentCards)
                    studentCard.IsOpen =
                            (element == studentCard && !studentCard.IsOpen);
            }
        }
    }
    …
}
```

Немного поэкспериментировав, я определился с тем, что буду квалифицировать как касание только прикосновение, длящееся не более ¼ секунды, при котором перемещение точки касания составляет не более 12 пикселов. Это позволит распознавать касания и отличать их от прокручивания.

В конце кода этого метода цикл *foreach* обеспечивает перечисление всех объектов *StudentCard* и задание значения свойства *IsOpen* для каждого из них. *IsOpen* всегда имеет значение *false*, если *StudentCard* не является выбранным элементом; и *IsOpen* всегда задается значение *false*, если в настоящее время это свойство имеет значение *true*. В противном случае, если объект *StudentCard* является выбранным элементом, текущее значение *IsOpen* *false* меняется на *true*. Конечно же, как для любого свойства-зависимости, обработчики событий изменения свойства *IsOpen* будут вызываться, только если значение свойства действительно изменилось.

Мы еще не рассматривали обработчик событий изменения свойства для свойства *IsOpen*. Как обычно, статическая версия этого метода вызывает версию экземпляра:

Проект Silverlight: StudentCardFile   Файл: StudentCard.xaml.cs (фрагмент)

```
public partial class StudentCard : UserControl
{
    …
    static void OnIsOpenChanged(DependencyObject obj,
                                DependencyPropertyChangedEventArgs args)
    {
        (obj as StudentCard).OnIsOpenChanged(args);
    }
    …
    void OnIsOpenChanged(DependencyPropertyChangedEventArgs args)
    {
        VisualStateManager.GoToState(this, IsOpen ? "Open" : "Normal", false);
    }
}
```

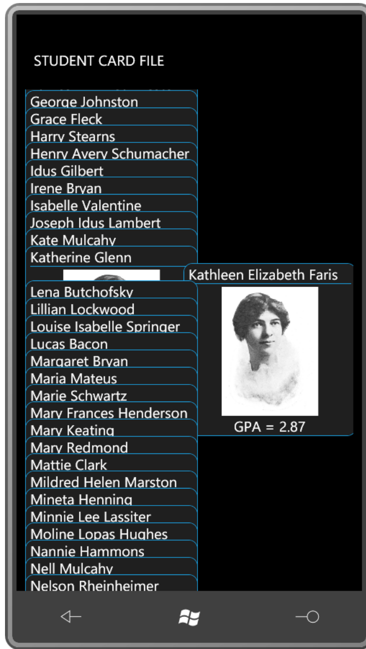
Версия экземпляра вызывает метод *VisualStateManager.GoToState*. Несмотря на то что Visual State Manger чаще всего используется в связи с элементами управления и шаблоном элементов управления, он может также использоваться с производными от *UserControl*, такими как *StudentCard*. Вызов *GoToState* обеспечивает переключение состояния из кода.

В XAML-файле разметка Visual State Manager должна располагаться сразу после самого верхнего элемента дерева визуальных элементов. В StudentCard.xaml это элемент *Border*. Рассмотрим остальной код StudentCard.xaml (с некоторым повтором предыдущего фрагмента), демонстрирующий разметку Visual State Manager для *TranslateTransform*, заданного для самого элемента управления:

Проект Silverlight: StudentCardFile Файл: StudentCard.xaml (фрагмент)

```
<UserControl x:Class="StudentCardFile.StudentCard"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             FontFamily="{StaticResource PhoneFontFamilyNormal}"
             FontSize="{StaticResource PhoneFontSizeNormal}"
             Foreground="{StaticResource PhoneForegroundBrush}"
             Width="240" Height="240">

    <UserControl.RenderTransform>
        <TranslateTransform x:Name="translate" />
    </UserControl.RenderTransform>

    <Border BorderBrush="{StaticResource PhoneAccentBrush}"
            BorderThickness="1"
            Background="{StaticResource PhoneChromeBrush}"
            CornerRadius="12"
            Padding="6 0">

        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Open">
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetName="translate"
                                 Storyboard.TargetProperty="X"
                                 To="220" Duration="0:0:1" />
                    </Storyboard>
                </VisualState>

                <VisualState x:Name="Normal">
                <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="translate"
                                 Storyboard.TargetProperty="X"
                                 Duration="0:0:1" />
                </Storyboard>
            </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>

        …

    </Border>
</UserControl>
```

Когда пользователь касается одного из элементов, он выдвигается в сторону, обеспечивая просмотр всей карточки:



В ходе данной главы я попытался реализовать полностью в XAML несколько совершенно разных типов операций. Это не всегда возможно; очень часто код просто необходим, в частности, для обработки сенсорного ввода.

Но код не может *заменить* XAML, код только *поддерживает* разметку. Эти классы принимают форму конвертеров привязок и пользовательских панелей, на которые ссылается XAML-файл. В общем, создавайте код *для* XAML, а не *вместо* XAML, как это делают все хорошие разработчики на Silverlight и Windows Phone 7.

Глава 18
Сводное представление и панорама

Приложения на Silverlight, которые должны представлять пользователю большие объемы информации, традиционно используют структуру постраничной навигации. Однако для приложения, выполняющегося на телефоне, разделение на страницы не самое лучшее решение. Портретная ориентация, простота мультисенсорного ввода и приобретающие все большую популярность пользовательские интерфейсы «fluid[1]» – все эти факторы предлагают другие типы компоновки. Такие альтернативы представлены в Windows Phone 7 двумя новыми элементами управления: *Pivot* (Сводное представление) и *Panorama* (Панорама).

И *Pivot*, и *Panorama* располагаются в библиотеке Microsoft.Phone.Controls, поэтому любое приложение, использующее эти элементы управления, должно ссылаться на эту DLL. Описаны *Pivot* и *Panorama* в пространстве имен *Microsoft.Phone.Controls*, вспомогательные компоненты – в пространстве имен*Microsoft.Phone.Controls.Primitives*, но эти классы пригодятся лишь для настройки данных элементов управления.

Концептуально *Pivot* и *Panorama* очень похожи. Оба элемента управления являются средством организации разрозненных компонентов приложения в виртуальном ориентированном горизонтально пространстве, которое может быть в несколько раз шире, чем фактическая ширина экрана телефона. Пользователь перемещается по элементу управления, просто проводя пальцем по экрану. Элементы управления *Pivot* и *Panorama* создавались преимущественно для компоновок с портретной ориентацией, но могут использоваться и в альбомном режиме.

Сходства и отличия

И *Pivot*, и *Panorama* наследуются от *ItemsControl* в виде класса с обобщенным параметром:

```
public class TemplatedItemsControl<T> : ItemsControl where T : new(),
FrameworkElement
```

Здесь описывается *ItemsControl*, который предполагается заполнять объектами типа *T*. *Pivot* и *Panorama* наследуются от *TemplatedItemsControl* с заданием в качестве параметра типа *PivotItem* (Элемент сводного представления) или *PanoramaItem* (Элементы панорамы), соответственно:

```
public class Pivot : TemplatedItemsControl<PivotItem>
public class Panorama : TemplatedItemsControl<PanoramaItem>
```

Предполагается, что элемент управления *Pivot* будет содержать элементы типа *PivotItem*, тогда как элемент управления *Panorama* будет включать элементы типа *PanoramaItem*. И *PivotItem*, и *PanoramaItem* наследуются от *ContentControl*. Если коллекция *Items* объекта *Pivot* или *Panorama* заполняется явно в XAML и коде, вы захотите заполнить ее элементами *PivotItem* или *PanoramaItem*; у этих элементов управления есть такое важное свойство, как *Header* (Верхний колонтитул), которое необходимо задать. Если же применяется привязка к свойству *ItemsSource*, определенному *ItemsControl*, эти объекты *PivotItem* и *PanoramaItem* создаются автоматически, а свойство *Header* задается через шаблон. (Не волнуйтесь, я приведу примеры.)

«Fluid» в переводе с английского означает «подвижный», «изменчивый». Интерфейсы такого типа предполагают плавное перетекание одного экрана или страницы в другую (*прим. переводчика*).

Для создания экземпляров этих элементов управления в XAML-файле необходимо объявить пространство имен XML для библиотеки и пространства имен Microsoft.Phone.Controls:

```
xmlns:controls="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
```

Вероятно, самый лучший способ познакомиться с этими классами – это поэкспериментировать с конкретным примером. Диалоговое окно New Project в Visual Studio предоставляет опции для создания проекта типа Windows Phone Pivot Application или Windows Phone Panorama Application. Безусловно, их можно использовать для своих экспериментов, но для демонстрационного приложения данной главы я выбрал другой подход.

Рассмотрим файл MainPage.xaml проекта PivotDemonstration (Демонстрация сводного представления). Я создал это приложение обычным путем, т.е. выбрав Windows Phone Application в диалоговом окне New Project. Но после этого я удалил большую часть содержимого MainPage.xaml, оставив только теги *PhoneApplicationPage*. Добавил объявление пространства имен XML для «controls» (самая длинная строка) и заменил содержимое страницы элементом управления *Pivot* с четырьмя дочерними *PivotItem*:

Проект Silverlight: PivotDemonstration Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage
    x:Class="PivotDemonstration.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
  xmlns:controls="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
    xmlns:system="clr-namespace:System;assembly=mscorlib"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <controls:Pivot Title="PIVOT DEMONSTRATION">
        <controls:PivotItem Header="ListBox">
            …
        </controls:PivotItem>

        <controls:PivotItem Header="Ellipse">
            …
        </controls:PivotItem>

        <controls:PivotItem Header="TextBlock">
            …
        </controls:PivotItem>

        <controls:PivotItem Header="Animation">
            …
        </controls:PivotItem>
    </controls:Pivot>
</phone:PhoneApplicationPage>
```

Свойству *Title* элемента управления *Pivot* задано «PIVOT DEMONSTRATION» (Демонстрация сводного представления). По умолчанию этот заголовок будет того же размера и располагаться там же, где и текст, который отображается вверху страницы Windows Phone.

(Обычно этот текст отображается элементом *TextBlock* под именем *ApplicationTitle*.) Для всех четырех элементов управления *PivotItem* задано свойство *Header*. Местоположение и размер этого текста аналогичны тексту, выводимому обычным *TextBlock* под именем *PageTitle*.

Элемент управления *PivotItem* наследуется от *ContentControl*, так что его возможности практически неограниченны. Для первого *PivotItem* я задал *ListBox*, включающий все доступные для приложений Windows Phone 7 шрифты, а также простой *DataTemplate*:

Проект Silverlight: PivotDemonstration Файл: MainPage.xaml (фрагмент)

```
<controls:PivotItem Header="ListBox">
    <ListBox FontSize="{StaticResource PhoneFontSizeLarge}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <TextBlock Text="{Binding}"
                           FontFamily="{Binding}" />
            </DataTemplate>
        </ListBox.ItemTemplate>

        <system:String>Arial</system:String>
        <system:String>Arial Black</system:String>
        <system:String>Calibri</system:String>
        <system:String>Comic Sans MS</system:String>
        <system:String>Courier New</system:String>
        <system:String>Georgia</system:String>
        <system:String>Lucida Sans Unicode</system:String>
        <system:String>Portable User Interface</system:String>
        <system:String>Segoe WP</system:String>
        <system:String>Segoe WP Black</system:String>
        <system:String>Segoe WP Bold</system:String>
        <system:String>Segoe WP Light</system:String>
        <system:String>Segoe WP Semibold</system:String>
        <system:String>Segoe WP SemiLight</system:String>
        <system:String>Tahoma</system:String>
        <system:String>Times New Roman</system:String>
        <system:String>Trebuchet MS</system:String>
        <system:String>Verdana</system:String>
        <system:String>Webdings</system:String>
    </ListBox>
</controls:PivotItem>
```

PivotItem предоставляет *ListBox* пространство, равное размеру страницы за вычетом области, занимаемой текстом *Title* и *Header*:

Конечно, есть возможность прокрутки *ListBox* в вертикальном направлении. Обратите внимание, что текст *Header* второго *PivotItem* рядом с заголовком первого затенен. В этом втором *PivotItem* отображается только *Ellipse*:

Проект Silverlight: PivotDemonstration   Файл: MainPage.xaml (фрагмент)

```
<controls:PivotItem Header="Ellipse">
    <Ellipse>
        <Ellipse.Fill>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="{StaticResource PhoneAccentColor}"
/>
                <GradientStop Offset="0.5" Color="{StaticResource
PhoneBackgroundColor}" />
                <GradientStop Offset="1" Color="{StaticResource
PhoneForegroundColor}" />
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
</controls:PivotItem>
```

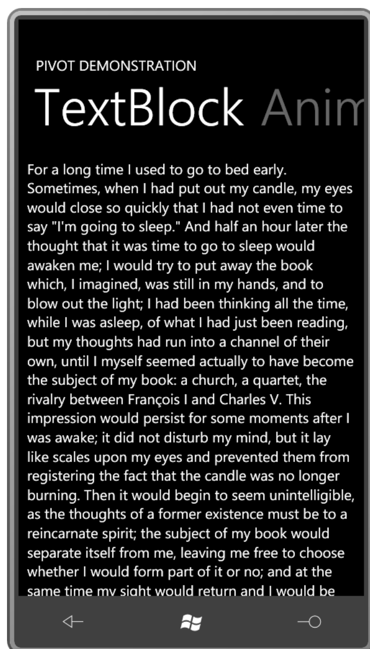
Здесь мы можем ясно видеть размер области, которую *PivotItem* предлагает своему содержимому:

Третий *PivotItem* включает *ScrollViewer* с большим *TextBlock*, содержимым которого является вступительный абзац известного романа:

Проект Silverlight: PivotDemonstration Файл: MainPage.xaml (фрагмент)

```
<controls:PivotItem Header="TextBlock">
    <ScrollViewer>
        <!-- from http://www.gutenberg.org/files/7178/7178-8.txt -->
        <TextBlock TextWrapping="Wrap">
            For a long time I used to go to bed early. Sometimes, when I had put out
            my candle, my eyes would close so quickly that I had not even time to
            say "I'm going to sleep." And half an hour later the thought that it was
            time to go to sleep would awaken me; I would try to put away the book
            which, I imagined, was still in my hands, and to blow out the light; I
            had been thinking all the time, while I was asleep, of what I had just
            been reading, but my thoughts had run into a channel of their own,
            until I myself seemed actually to have become the subject of my book:
            a church, a quartet, the rivalry between François I and Charles V. This
            impression would persist for some moments after I was awake; it did not
            disturb my mind, but it lay like scales upon my eyes and prevented them
            from registering the fact that the candle was no longer burning. Then
            it would begin to seem unintelligible, as the thoughts of a former
            existence must be to a reincarnate spirit; the subject of my book would
            separate itself from me, leaving me free to choose whether I would form
            part of it or no; and at the same time my sight would return and I
            would be astonished to find myself in a state of darkness, pleasant and
            restful enough for the eyes, and even more, perhaps, for my mind, to
            which it appeared incomprehensible, without a cause, a matter dark
            indeed.
        </TextBlock>
    </ScrollViewer>
</controls:PivotItem>
```

И здесь тоже никаких проблем с прокруткой:

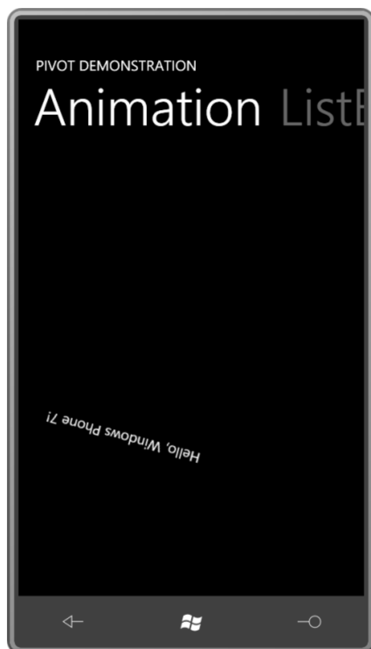


Последний *PivotItem* включает *TextBlock*, к которому применяются несколько анимаций:

Проект Silverlight: PivotDemonstration Файл: MainPage.xaml (фрагмент)

```
<controls:PivotItem Header="Animation">
    <TextBlock Text="Hello, Windows Phone 7!"
               HorizontalAlignment="Left"
               VerticalAlignment="Top"
               RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <CompositeTransform x:Name="xform" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <controls:PivotItem.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="xform"
                                     Storyboard.TargetProperty="Rotation"
                                     From="0" To="360" Duration="0:0:3"
                                     RepeatBehavior="Forever" />

                    <DoubleAnimation Storyboard.TargetName="xform"
                                     Storyboard.TargetProperty="TranslateX"
                                     From="0" To="300" Duration="0:0:5"
                                     AutoReverse="True"
                                     RepeatBehavior="Forever" />

                    <DoubleAnimation Storyboard.TargetName="xform"
                                     Storyboard.TargetProperty="TranslateY"
                                     From="0" To="600" Duration="0:0:7"
                                     AutoReverse="True"
                                     RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </controls:PivotItem.Triggers>
</controls:PivotItem>
```

Анимации обеспечивают перемещение *TextBlock* по экрану и вращение:



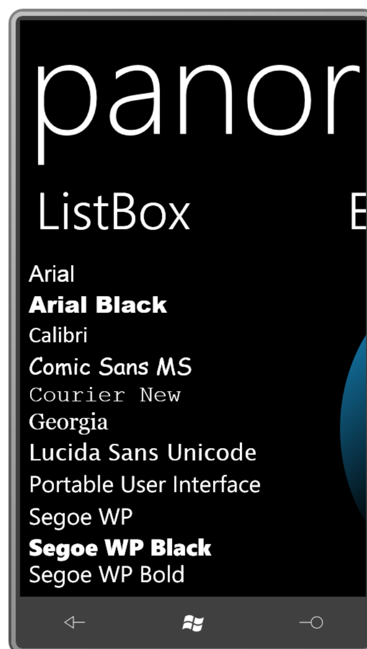
Обратите внимание на заголовок первого *PivotItem*, который расположен справа от активного *PivotItem*. Анимации рассчитаны соответственно приближенному размеру области содержимого *PivotItem* для большого экрана при портретном режиме отображения. Если повернуть телефон или эмулятор на бок, *TextBlock* будет выходить за границы экрана время от времени.

Приложение PanoramaDemonstration очень похоже на PivotDemonstration, просто в файле MainPage.xaml проекта PivotDemonstration слово «Pivot» везде заменяем на «Panorama». Единственное отличие – текст, являющийся значением свойства *Title*, написан строчными буквами:

Проект Silverlight: PanoramaDemostration Файл: MainPage.xaml (фрагмент)

```
<controls:Panorama Title="panorama demonstration">
    <controls:PanoramaItem Header="ListBox">
        …
    </controls:PanoramaItem>

    <controls:PanoramaItem Header="Ellipse">
        …
    </controls:PanoramaItem>

    <controls:PanoramaItem Header="TextBlock">
        …
    </controls:PanoramaItem>

    <controls:PanoramaItem Header="Animation">
        …
    </controls:PanoramaItem>
</controls:Panorama>
```

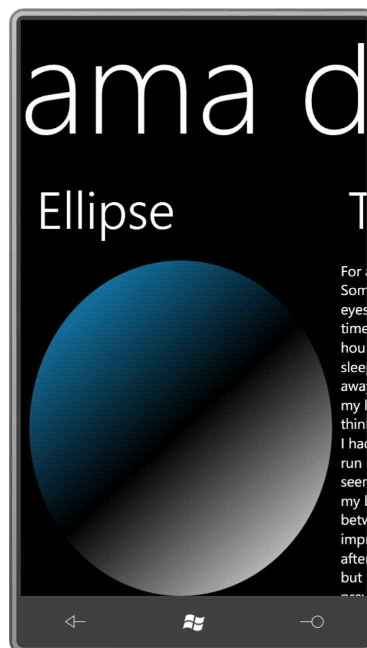
Несмотря на всю концептуальную схожесть *Pivot* и *Panorama*, эти элементы управления имеют совершенно разную эстетику. Следующие несколько снимков экрана позволяют сравнить эти два элемента управления: *Pivot* располагается слева, и *Panorama* – справа. Обратите внимание на то, как обрабатывается *Title* в *Panorama*: он намного большего размера и охватывает все остальные элементы:



Я не сделал этого в данном приложении, но обычно свойству *Background* элемента управления *Panorama* задается *ImageBrush* с протяженной в горизонтальном направлении растровой матрицей, которая выходит за границы экрана, охватывая весь элемент управления. (Это можно увидеть в приложениях Games (Игры), Marketplace (Магазин) и Pictures (Изображения) телефона.)

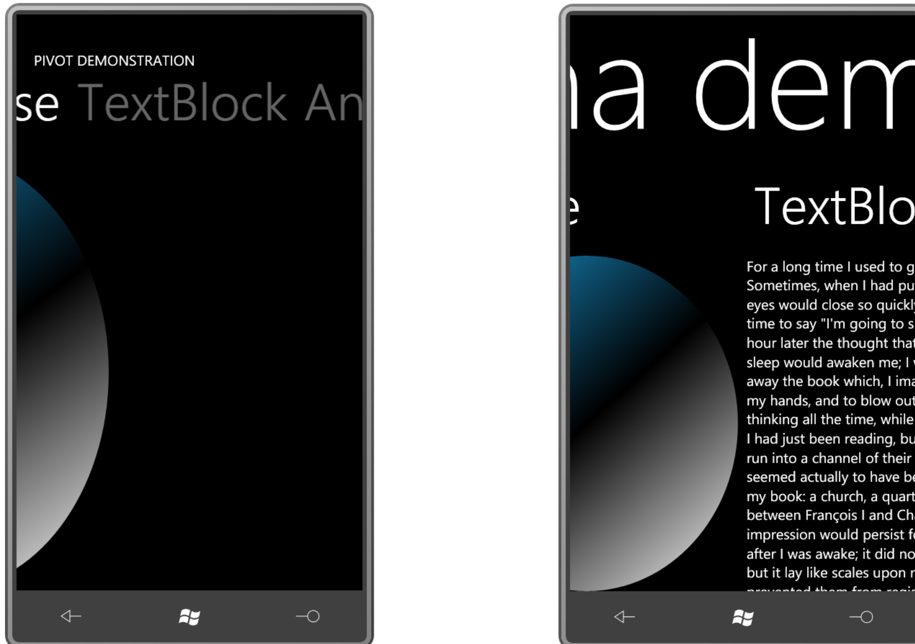
Из-за такого большого заголовка *Panorama* предоставляет меньше места для содержимого своих *PanoramaItem*. Доступное пространство сокращено и в горизонтальном направлении, потому что справа выглядывает часть следующего элемента.

Перемещаться вперед и назад по *Pivot* и *Panorama*, можно просто проводя пальцем по экрану слева направо или справа налево. В *Panorama* вполне естественным будет провести пальцем по тексту *Title*, чтобы «перелистнуть» элемент управления. В *Pivot* (но не в *Panorama*) перейти к одному из элементов, можно коснувшись текста его *Header*:



Обратите внимание также на смещение *Title* элемента управления *Panorama*, которое позволяет визуально определить текущее местонахождение относительно виртуальной ширины всего содержимого.

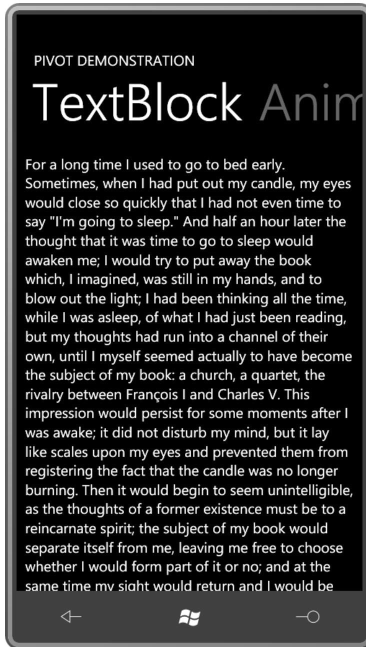
В ходе экспериментов с перемещениями между элементами *Pivot* и *Panorama* обнаруживается, что эти два элемента управления ведут себя очень по-разному. В обоих случаях тексты *Header* визуально несколько отделены от самих элементов управления. В *Pivot* один элемент сначала полностью исчезает с экрана, и только после этого выводится содержимое другого элемента. В *Panorama* мы можем видеть содержимое двух элементов одновременно. Вот как выглядит процесс перехода от элемента к элементу:

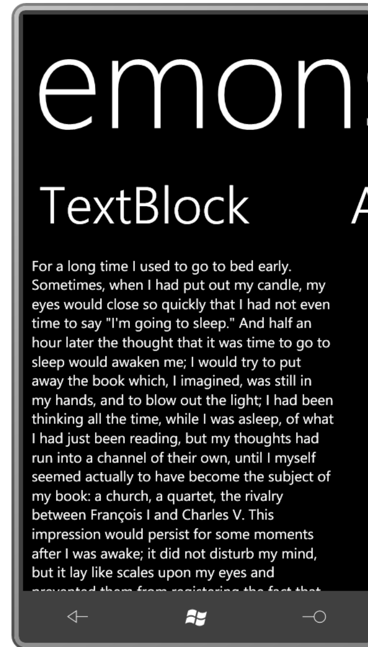


Panorama обеспечивает намного более реальное ощущение широкого виртуального экрана с ограниченным окном просмотра, особенно при использовании протяженного фонового изображения. В случае с *Pivot* кажется, что он занимает только область экрана и просто переносит отдельные элементы в и из области видимости.

Элемент управления *Pivot* определяет несколько событий, которых нет в *Panorama*. Это *LoadingPivotItem* (Загрузка элемента сводного представления), *LoadedPivotItem* (Элемент сводного представления загружен), *UnloadingPivotItem* (Выгрузка элемента сводного представления), *UnloadedPivotItem* (Элемент сводного представления выгружен). Данные события сигнализируют о выходе одного элемента из области просмотра и помещении в нее другого элемента; они неуместны в более неразрывной схеме смены элементов в *Panorama*.

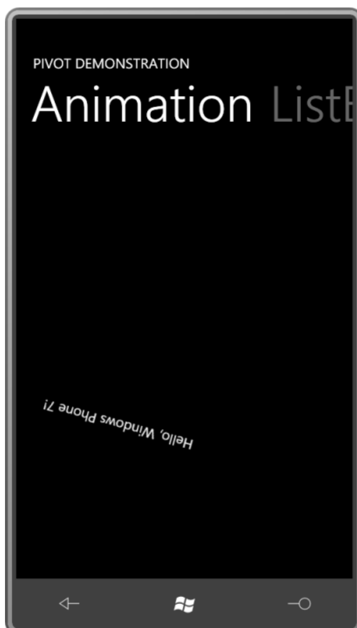
И *Pivot*, и *Panorama* определяют события*SelectionChanged*, а также свойства *SelectedIndex* и *SelectedItem*. Выбирается весь элемент *PivotItem* или *PanoramaItem* целиком, и событие формируется, только лишь когда элемент полностью занимает окно просмотра.





И *Pivot*, и *Panorama* определяют свойства *TitleTemplate* (Шаблон заголовка) и *HeaderTemplate* (Шаблон верхнего колонтитула) типа *DataTemplate*. Когда для задания содержимого элемента управления используются привязки, это позволяет определять дерево визуальных элементов для обозначения того, как свойства *Title* и *Header* применяют эти данные.

Особенно большое значение имеет свойство *HeaderTemplate* при связывании посредством привязки свойства *ItemsSource* элемента управления *Pivot* или *Panorama* с коллекцией. В этом случае объекты *PivotItem* или *PanoramaItem* не создаются явно. Этот *HeaderTemplate* понадобится в привязке для задания текста *Header*, но шаблон может состоять из одного только *TextBlock*. Позже в этой главе будет рассмотрен соответствующий пример.



Те кто жаждет приключений, может описать абсолютно новый *ControlTemplate* для *Pivot* или *Panorama*. Для шаблона *Pivot* необходим *PivotHeadersControl* (Элемент управления верхние колонтитулы сводного представления) (который наследуется от *TemplateItemsControl* (Элемент управления элементы шаблона) типа *PivotHeaderItem* (Элемент верхнего

колонтитула сводного представления)), и для шаблона *Panorama* необходимы три объекта *PanningLayer* (Слой панорамирования). *PanningLayer* наследуется от *ContentControl*, и пространство имен *Microsoft.Phone.Controls.Primitives* включает классы *PanningBackgroundLayer* (Слой фона панорамирования) и *PanningTitleLayer* (Слой заголовка панорамирования), которые наследуются от *PanningLayer*.

В конце концов мы возвращаемся к исходному представлению. Однако *Panorama* позволяет одновременно видеть три дочерних элемента: *ListBox* полностью, небольшой фрагмент *Ellipse* справа и кусочек вращающегося текста, пересекающий элемент «Comic Sans MS» – это анимация элемента, располагающегося слева.

Сортировка коллекции музыкальных произведений по композитору

Элемент управления *Pivot* идеально подходит для реализации приложения, которое я давно задумал. Это приложение должно компенсировать основной, по моему мнению, недостаток таких портативных музыкальных проигрывателей, как Zune и Windows Phone 7. Здесь требуется небольшое разъяснение.

Вероятно, вам известно, что все многообразие музыкальных произведений США и Европы можно разделить на две большие категории: ориентированные на исполнителя и ориентированные на композитора. Традиция ориентации на исполнителя тесно связана с появлением и эволюцией звукозаписывающих технологий и охватывает исполнителей, скажем, начиная от Роберта Джонсона (1911–1938), заканчивая Леди Гага (г.р. 1986). К этой категории относятся произведения, представленные музыкальной формой *песня*, которая обычно исполняется вокалистом под аккомпанемент и длится всего несколько минут.

Традиция ориентированности на композитора имеет намного более протяженную историю, начиная, скажем, от Клаудио Монтеверди (1567–1643), и до Дженифер Хигтон (г.р. 1962), и охватывает намного большее число музыкальных форм (например, струнный квартет, фортепьянный концерт, симфония, опера и песни в том числе), очень отличающихся по длительности, стилям и инструментальному составу.

Люди, слушающие музыку второго типа, предпочитают организовывать ее по композитору, затем по произведению и уже потом по исполнителю. (В ориентированной на исполнителя традиции для обозначения человека или группы людей, исполняющих музыку, достаточно свойства *artist* (*исполнитель*).) Настольное ПО Zune позволяет вводить данные о композиторе при загрузке произведения и копировании с CD, но эти сведения не передаются с файлами на портативные устройства, такие как телефон. Даже если данные о композиторе включены в музыкальные файлы, передаваемые на телефон, открытые свойства классов, используемых для доступа к музыке, не обеспечивают доступа к ним.

Чтобы компенсировать этот недостаток, люди, слушающие ориентированную на композитора музыку, часто включают имя композитора в название альбома, используя двоеточие в качестве разделителя, например так:

Малер: симфония No. 2

Многие CD копируются с использованием такого формата для названий альбомов. В альбомах, включающих музыку нескольких композиторов, я указываю имена всех композиторов через запятую:

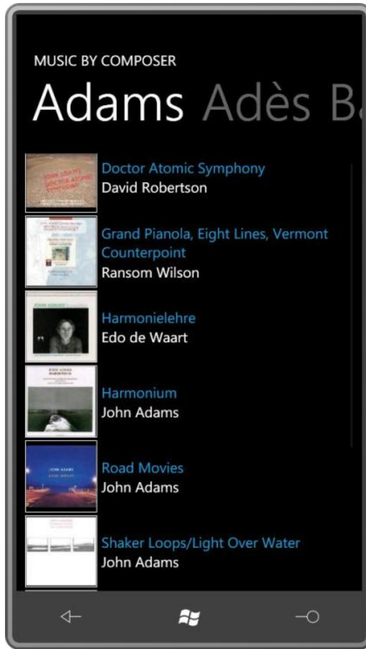
Адес, Шуберт: фортепьянные квинтеты

За все эти годы я скопировал на ПК около 600 CD и большинство из них идентифицировал именно таким образом. Проигрыватель выводит список альбомов в алфавитном порядке по названию альбома, т.е. получается, что при таком именовании альбомов музыка сортирована по композитору, что также очень удобно.

Но мне хотелось большего. Я хотел создать иерархическую структуру по именам композиторов и иметь возможность видеть и выбирать любого композитора, от Шуберта до Дебюсси или Мессиана.

Реализовать эту идею я решил в приложении для Windows Phone 7 под названием MusicByComposer (Музыка по композитору). Это приложение будет выполнять доступ к музыкальной библиотеке телефона и – исходя из предположения, что названия альбомов начинаются с имени или имен композиторов, за которыми следует двоеточие – извлекает имена композиторов из названий альбомов. После этого выполняется сортировка музыки по композитору, и каждый композитор становится *PivotItem*. Содержимым этого *PivotItem* является *ListBox* со списком всех альбомов музыки соответствующего композитора.

Стандартное окно приложения MusicByComposer выглядит следующим образом:



Как видим, это стандартный элемент управления *Pivot*, каждый *PivotItem* которого соответствует определенному композитору. В данном случае первый *PivotItem* представляет альбомы американского композитора Джона Адамса (г.р. 1947). Заголовки второго *PivotItem*, которые можно видеть на иллюстрации выше, говорят о том, что он включает произведения британского композитора Томаса Адеса (г.р. 1971) и немецкого композитора Иоганна Себастьяна Баха (1685–1750).

Все музыкальные файлы, не содержащие двоеточия в названии альбома, будут перечислены в одном *PivotItem* под заголовком «Other» (Прочие).

PivotItem каждого композитора включает *ListBox*, в который входит эскиз обложки альбома, название альбома (без имени композитора), отображаемое шрифтом контрастного цвета, и имя исполнителя, для шрифта которого используется цвет переднего плана.

По касанию названия любого альбома выполняется переход на страницу этого альбома:

Это стандартный *PhoneApplicationPage* с двумя стандартными элементами *TextBlock* для названия приложения и заголовка страницы, но, как видите, размер и местоположение этих заголовков аналогичны заголовкам элемента управления *Pivot* на главной странице. Немного крупнее отображается обложка альбома, полное название альбома и исполнитель. Ниже располагается *ScrollViewer* и *ItemsControl*, в котором перечислены все треки альбома. Эта страница не имеет интерфейса для обработки сенсорного ввода, кроме прокрутки, все управление осуществляется кнопками *ApplicationBar*: возвращение к предыдущему треку, воспроизведение и приостановка воспроизведения, переход к следующему треку. Трек, который воспроизводится в настоящее время, выделяется контрастным цветом, и для него выводятся данные о длительности и продолжительности воспроизведения.

Обычно в Windows Phone 7, если приложение начинает воспроизведение альбома, проигрывается весь альбом, даже если приложение завершается или гаснет экран телефона. Приложение MusicByComposer позволяет переходить к другим альбомам, но прекращает воспроизведение текущего альбома и начинает воспроизводить следующий только после нажатия средней кнопки *ApplicationBar* для приостановки текущего воспроизведения и повторного нажатия этой же кнопки для начала воспроизведения альбома текущей страницы.

Подключение XNA

Как можно вспомнить из глав 4 и 14, приложение на Silverlight может выполнять доступ к библиотеке фотографий телефона и сохранять изображения в ней. Это осуществляется с помощью класса XNA *MediaLibrary* из пространства имен *Microsoft.Xna.Framework.Media*. Для доступа и воспроизведения музыкальных файлов используется этот же класс и другие классы этого пространства имен.

В любом приложении, использующем *MediaLibrary*, должна присутствовать ссылка на библиотеку Microsoft.Xna.Framework. Приложению MusicByComposer для работы с элементом управления *Pivot* также необходима ссылка на Microsoft.Phone.Controls.

При использовании сервисов XNA для воспроизведения музыки из приложения на Silverlight возникают некоторые сложности. Как описывается в разделе документации по XNA «Enable XNA Framework Events in Windows Phone Applications» [1], для этого требуется класс, вызывающий статический метод XNA *FrameworkDispatcher.Update* с частотой, равной частоте обновления экрана, т.е. 30 раз в секунду. По сути, следующий класс проекта MusicByComposer – это класс, представленный в этом разделе документации:

Проект Silverlight: MusicByComposer Файл: XnaFrameworkDispatcherService.cs

```
using System;
using System.Windows;
using System.Windows.Threading;
using Microsoft.Xna.Framework;

namespace MusicByComposer
{
    public class XnaFrameworkDispatcherService : IApplicationService
    {
        DispatcherTimer timer;

        public XnaFrameworkDispatcherService()
        {
            timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromTicks(333333);
```

[1] Использование событий XNA Framework в приложениях Windows Phone (*прим. переводчика*).

```
            timer.Tick += OnTimerTick;
            FrameworkDispatcher.Update();
        }

        void OnTimerTick(object sender, EventArgs args)
        {
            FrameworkDispatcher.Update();
        }

        void IApplicationService.StartService(ApplicationServiceContext context)
        {
            timer.Start();
        }

        void IApplicationService.StopService()
        {
            timer.Stop();
        }
    }
}
```

Экземпляр этого класса должен быть создан в разделе *ApplicationLifetimeObjects* файла App.xaml. Обратите внимание на объявление пространства имен XML для «local»:

Проект Silverlight: MusicByComposer   Файл: App.xaml

```
<Application
    x:Class="MusicByComposer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:local="clr-namespace:MusicByComposer">

    <!--Ресурсы приложения-->
    <Application.Resources>
    </Application.Resources>

    <Application.ApplicationLifetimeObjects>

        <!-- Необходимо для воспроизведения музыки из приложения на Silverlight -->
        <local:XnaFrameworkDispatcherService />

        <!--Обязательный объект, обрабатывающий события времени жизни приложения-->
        <shell:PhoneApplicationService
            Launching="Application_Launching" Closing="Application_Closing"
            Activated="Application_Activated"
Deactivated="Application_Deactivated"/>
    </Application.ApplicationLifetimeObjects>
</Application>
```

Для целей тестирования эмулятор телефона поставляется с музыкальной библиотекой, включающей всего один альбом с тремя короткими песнями. С их помощью можно настроить базовую логику извлечения и воспроизведения альбома, но вряд ли получится протестировать приложение по-настоящему.

Для отладки из Visual Studio приложения, выполняющегося на реальном телефоне, понадобиться выйти из настольного приложения Zune (потому что оно требует эксклюзивного доступа к музыкальной библиотеке) и запустить инструмент Connect (Подключение): WPDTPTConnect32 для 32-разрядной Windows или WPDTPTConnect64 для 64-разрядной Windows.

Я также выявил еще одну проблему. Когда приложение развернуто на телефоне и выполняется вне Visual Studio, оно сообщает, что в музыкальной библиотеке телефона нет доступных файлов… Этого не происходит, если до выполнения данного приложения выполнялось приложение на XNA. Я выяснил, что это дефект первой версии Windows Phone 7. Чтобы обойти эту проблему, я решил выполнять доступ к приложению через хаб Games телефона. Для этого в теге *App* файла WMAppManifest.xml зададим следующий атрибут:

```
Genre="apps.games"
```

Также я снабдил приложение файлами Background.png и ApplicationIcon.png с портретами, наверное, самых известных композиторов.

Музыкальные классы XNA: MediaLibrary

Приложение, которое должно воспроизводить музыкальные файлы под управлением Windows Phone 7, использует классы пространства имен *Microsoft.Xna.Framework.Media*. Но сначала потребуется выполнить доступ к музыкальной библиотеке, и для этого необходим новый экземпляр *MediaLibrary* – этот же класс применялся для доступа к библиотеке фотографий.

Класс *MediaLibrary* определяет несколько свойств только для чтения, которые обеспечивают возможность доступа к музыкальной библиотеке несколькими стандартными способами. К этим свойствам относятся:

- *Albums* (Альбомы) типа *AlbumCollection* (Коллекция альбомов), коллекция объектов *Album*.
- *Songs* (Песни) типа *SongCollection* (Коллекция песен), коллекция объектов *Song* (Песня).
- *Artists* (Исполнители) типа *ArtistCollection* (Коллекция исполнителей), коллекция объектов *Artist* (Исполнитель).
- *Genres* (Жанры) типа *GenreCollection* (Коллекция жанров), коллекция объектов *Genre* (Жанр).

Каждая из этих коллекция включает всю музыку библиотеки, но организовывает ее по-разному. (Существенно упростило бы приложение наличие свойства *Composer* (Композитор) типа *ComposerCollection* (Коллекция композиторов).)

Для моих целей свойство *Albums* класса *MediaLibrary* кажется наиболее полезным. Класс *AlbumCollection* – это коллекция элементов типа *Album*, и *Album* имеет следующие свойства только для чтения (помимо всех остальных):

- *Name* типа *string*
- *Artist* типа *Artist*
- *Songs* типа *SongCollection*
- *HasArt* (Включает обложку) типа *bool*

Когда *HasArt* имеет значение *true*, мы можем вызывать два метода: *GetAlbumArt* (Получить обложку альбома) и *GetThumbnail* (Получить эскиз). Оба метода возвращают объекты *Stream* для доступа к растровому изображению обложки альбома. *GetAlbumArt* возвращает квадратное растровое изображение со стороной около 200 пикселов, и *GetThumbnail* возвращает квадратное растровое изображение со стороной около 100 пикселов.

SongCollection в экземпляре *Album* включает все треки альбома. (В ориентированной на композитора традиции не уместно использовать слово *песня* в применении к записям альбома, поскольку отдельный трек может быть частью симфонии, например, но под влиянием ориентированной на исполнителя традиции мы вынуждены мириться с таким именованием классов.) Объект *Song* включает несколько свойств только для чтения:

- *Name* типа *string*
- *Album* типа *Album*
- *Artist* типа *Artist*
- *Duration* типа *TimeSpan*.

Для организации музыкальной библиотеки по композитору и для целей привязки данных я ввел еще пару новых классов. Мой класс *AlbumInfo* (Сведения об альбоме) – это по сути класс-оболочка для XNA-класса *Album*:

Проект Silverlight: MusicByComposer Файл: AlbumInfo.cs

```
using System;
using System.Windows.Media.Imaging;
using Microsoft.Xna.Framework.Media;

namespace MusicByComposer
{
    public class AlbumInfo : IComparable<AlbumInfo>
    {
        BitmapImage albumArt;
        BitmapImage thumbnailArt;

        public AlbumInfo(string shortAlbumName, Album album)
        {
            this.ShortAlbumName = shortAlbumName;
            this.Album = album;
        }

        public string ShortAlbumName { protected set; get; }

        public Album Album { protected set; get; }

        public BitmapSource AlbumArt
        {
            get
            {
                if (albumArt == null && Album.HasArt)
                {
                    BitmapImage bitmapImage = new BitmapImage();
                    bitmapImage.SetSource(Album.GetAlbumArt());
                    albumArt = bitmapImage;
                }
                return albumArt;
            }
        }

        public BitmapSource ThumbnailArt
        {
            get
            {
                if (thumbnailArt == null && Album.HasArt)
                {
                    BitmapImage bitmapImage = new BitmapImage();
                    bitmapImage.SetSource(Album.GetThumbnail());
                    thumbnailArt = bitmapImage;
                }
```

```
                return thumbnailArt;
            }
        }

        public int CompareTo(AlbumInfo albumInfo)
        {
            return ShortAlbumName.CompareTo(albumInfo.ShortAlbumName);
        }
    }
}
```

В классе *AlbumInfo* имеется свойство типа *Album* и еще три новых свойства. *ShortAlbumName* (Краткое название альбома) – это название альбома без указания композитора или композиторов в начале (например, «Малер: Симфония No. 2» становится «Симфония No. 2»). Это свойство используется в методе *CompareTo* (Сравнить с) для целей сортировки. На первом из трех снимков экрана для приложения MusicByComposer можно заметить, что названия альбомов сортированы.

Методы *GetAlbumArt* и *GetThumbnail* класса *Album* возвращают объекты *Stream*. Для целей привязки я обеспечил два открытых свойства типа *BitmapImage*. Класс создает эти объекты лишь при первом доступе к ним и затем кэширует их для последующего использования.

Следующий класс, *ComposerInfo* (Сведения о композиторе), включает имя композитора и список всех объектов *AlbumInfo* с музыкой этого композитора:

Проект Silverlight: MusicByComposer Файл: ComposerInfo.cs

```
using System;
using System.Collections.Generic;

namespace MusicByComposer
{
    public class ComposerInfo
    {
        public ComposerInfo(string composer, List<AlbumInfo> albums)
        {
            Composer = composer;
            albums.Sort();
            Albums = albums;
        }

        public string Composer { protected set; get; }

        public IList<AlbumInfo> Albums { protected set; get; }
    }
}
```

Обратите внимание, что сортировка списка объектов *AlbumInfo* выполняется в конструкторе.

Класс *MusicPresenter* (Презентатор музыки) отвечает за доступ к музыкальной библиотеке телефона, получение всех альбомов, проверку названий альбомов на наличие имен композиторов и создание объектов типа *ComposerInfo* и *AlbumInfo*. Вся основная работа выполняется в конструкторе экземпляра: данные сохраняются в словаре, при этом имена композиторов используются как ключи, ссылающиеся на элементы типа *List<AlbumInfo>*:

Проект Silverlight: MusicByComposer Файл: MusicPresenter.cs

```
using System;
using System.Collections.Generic;
```

```
using Microsoft.Xna.Framework.Media;

namespace MusicByComposer
{
    public class MusicPresenter
    {
        // Статический конструктор
        static MusicPresenter()
        {
            if (Current == null)
                Current = new MusicPresenter();
        }

        // Конструктор экземпляра
        public MusicPresenter()
        {
            // Делаем этот класс синглтоном
            if (MusicPresenter.Current != null)
            {
                this.Composers = MusicPresenter.Current.Composers;
                return;
            }

            MediaLibrary mediaLib = new MediaLibrary();
            Dictionary<string, List<AlbumInfo>> albumsByComposer =
                                new Dictionary<string, List<AlbumInfo>>();

            foreach (Album album in mediaLib.Albums)
            {
                int indexOfColon = album.Name.IndexOf(':');

                // Проверка на ошибки
                if (indexOfColon != -1 &&
                    // Двоеточие в начале названия альбома
                    (indexOfColon == 0 ||
                    // Двоеточие в конце названия альбома
                    indexOfColon == album.Name.Length - 1 ||
                    // ничего перед двоеточием
                    album.Name.Substring(0, indexOfColon).Trim().Length == 0 ||
                    // ничего после двоеточия
                    album.Name.Substring(indexOfColon + 1).Trim().Length == 0))
                {
                    indexOfColon = -1;
                }

                // Основная логика обработки альбомов, включающих имя композитора
                if (indexOfColon != -1)
                {
                    string[] albumComposers =
                                album.Name.Substring(0, indexOfColon).Split(',');
                    string shortAlbumName = album.Name.Substring(indexOfColon +
1).Trim();
                    bool atLeastOneEntry = false;

                    foreach (string composer in albumComposers)
                    {
                        string trimmedComposer = composer.Trim();

                        if (trimmedComposer.Length > 0)
                        {
                            atLeastOneEntry = true;

                            if (!albumsByComposer.ContainsKey(trimmedComposer))
                                albumsByComposer.Add(trimmedComposer,
                                                new List<AlbumInfo>());

                            albumsByComposer[trimmedComposer].Add(
                                                new AlbumInfo(shortAlbumName,
album));
```

```
                    }
                }

                // Еще один вариант ошибки: только запятые перед двоеточием
                if (!atLeastOneEntry)
                {
                    indexOfColon = -1;
                }
            }

            // Категория "Other" для альбомов без указания имени композитора
            if (indexOfColon == -1)
            {
                if (!albumsByComposer.ContainsKey("Other"))
                    albumsByComposer.Add("Other", new List<AlbumInfo>());

                albumsByComposer["Other"].Add(new AlbumInfo(album.Name, album));
            }
        }

        mediaLib.Dispose();

        // Передаем ключи Dictionary в List для сортировки
        List<string> composerList = new List<string>();

        foreach (string composer in albumsByComposer.Keys)
            composerList.Add(composer);

        (composerList as List<string>).Sort();

        // Определяем свойство Composers
        Composers = new List<ComposerInfo>();

        foreach (string composer in composerList)
            Composers.Add(new ComposerInfo(composer,
albumsByComposer[composer]));

        Current = this;
    }

    public static MusicPresenter Current { protected set; get; }

    public IList<ComposerInfo> Composers { private set; get; }
  }
}
```

Приложению необходим только один экземпляр данного класса. Музыкальная библиотека не будет меняться в ходе выполнения приложения, поэтому нет основания для повторного выполнения этого конструктора экземпляра. По этой причине по завершении выполнения конструктор задает в качестве значения статического свойства *Current* (Текущий) экземпляр только что созданного *MusicPresenter*. Этот первый экземпляр фактически создается статическим конструктором в самом начале класса и завершается заданием свойства *Composers* (Композиторы) (ближе к концу описания класса), которое включает список объектов *ComposerInfo*. Если конструктор вызывается снова, он просто передает существующее свойство *Composers* в новый экземпляр.

Почему бы не сделать класс *MusicPresenter* статическим и не упростить весь этот процесс? Да потому что *MusicPresenter* используется в привязках данных в файлах XAML, и для этих привязок необходим фактический экземпляр класса. Однако доступ к этому классу должен осуществляться также из кода, для этого и пригодится свойство *MusicPresenter.Current*.

Этот статический конструктор выполняется, когда приложение впервые выполняет доступ к классу, конечно же, и также при повторном доступе к этому классу после захоронения. В

этом случае восстанавливать данные из *MediaLibrary*, безусловно, проще, чем сохранять их в изолированное хранилище.

Вывод альбомов на экран

Когда приложение начинает выполнение, на экран выводится *MainPage*. XAML-файл включает объявления пространств имен XML для «controls» (чтобы обеспечить доступ к элементу управления *Pivot*) и «local» (для *MusicPresenter*). В коллекции *Resources* создается экземпляр *MusicPresenter*:

Проект Silverlight: MusicByComposer Файл: MainPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.Resources>
    <local:MusicPresenter x:Key="musicPresenter" />
</phone:PhoneApplicationPage.Resources>
```

В дизайнере Visual Studio сообщит о том, что не может создать экземпляр *MusicPresenter*. Конечно же, не может, потому что для этого необходимо иметь доступ к музыкальной библиотеке телефона (или эмулятора телефона).

Практически все дерево визуальных элементов страницы – это элемент управления *Pivot*:

Проект Silverlight: MusicByComposer Файл: MainPage.xaml (фрагмент)

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <controls:Pivot Name="pivot"
                    Title="MUSIC BY COMPOSER"
                    ItemsSource="{Binding Source={StaticResource musicPresenter},
                                          Path=Composers}">
        <controls:Pivot.HeaderTemplate>
            <!-- Объекты типа ComposerInfo -->
            <DataTemplate>
                <TextBlock Text="{Binding Composer}" />
            </DataTemplate>
        </controls:Pivot.HeaderTemplate>

        <controls:Pivot.ItemTemplate>
            <!-- Объекты типа ComposerInfo -->
            <DataTemplate>
                <ListBox ItemsSource="{Binding Albums}"
                         SelectionChanged="OnListBoxSelectionChanged">
                    <ListBox.ItemTemplate>
                        <!-- Объекты типа AlbumInfo -->
                        <DataTemplate>
                            <Grid Background="Transparent">
                                <Grid.ColumnDefinitions>
                                    <ColumnDefinition Width="Auto" />
                                    <ColumnDefinition Width="*" />
                                </Grid.ColumnDefinitions>

                                <Border Grid.Column="0"
                                        BorderBrush="{StaticResource
PhoneForegroundBrush}"
                                        BorderThickness="1"
                                        Width="100" Height="100"
                                        Margin="0 2 6 2">
                                    <Image Source="{Binding ThumbnailArt}" />
                                </Border>

                                <StackPanel Grid.Column="1"
                                            VerticalAlignment="Center">
```

```
                                    <TextBlock
                                            Text="{Binding ShortAlbumName}"
                                            Foreground="{StaticResource 
PhoneAccentBrush}"
                                            TextWrapping="Wrap" />
                                    <TextBlock Text="{Binding Album.Artist.Name}"
                                               TextWrapping="Wrap" />
                                </StackPanel>
                            </Grid>
                        </DataTemplate>
                    </ListBox.ItemTemplate>
                </ListBox>
            </DataTemplate>
        </controls:Pivot.ItemTemplate>
    </controls:Pivot>
</Grid>
```

XAML-файл подчеркивает мощь шаблонов и привязки данных. Как мы помним, *Pivot* наследуется от *ItemsTemplate*, так что у него есть свойство *ItemsSource*, которое можно связать с коллекцией посредством привязки данных:

```
ItemsSource="{Binding Source={StaticResource musicPresenter},
                      Path=Composers}"
```

Это означает, что *Pivot* заполняется коллекцией объектов типа *ComposerInfo*. Внутри себя *Pivot* будет формировать объект типа *PivotInfo* для каждого элемента *ComposerInfo*. Свойство *Header* каждого *PivotItem* должно быть связано посредством привязки данных со свойством *Composer* соответствующего объекта *ComposerInfo*. Но фактический объект *PivotItem* создается автоматически! Именно для этого *Pivot* определяет свойство *HeaderTemplate*:

```
<controls:Pivot.HeaderTemplate>
    <!-- Объекты типа ComposerInfo -->
    <DataTemplate>
        <TextBlock Text="{Binding Composer}" />
    </DataTemplate>
</controls:Pivot.HeaderTemplate>
```

Не стоит беспокоиться о форматировании объекта *TextBlock* в этом шаблоне, он волшебным образом получает необходимое форматирование, вероятно, через наследование свойств.

Класс *Pivot* также определяет *ItemTemplate*. Это *DataTemplate*, используемый для формирования содержимого каждого *PivotItem*:

```
<controls:Pivot.ItemTemplate>
    <!-- Объекты типа ComposerInfo -->
    <DataTemplate>
        <ListBox ItemsSource="{Binding Albums}"
                 SelectionChanged="OnListBoxSelectionChanged">
        …
        </ListBox>
    </DataTemplate>
</controls:Pivot.ItemTemplate>
```

Этот *DataTemplate* состоит из *ListBox* со списком всех альбомов, ассоциированных с композитором, который представлен объектом *PivotItem*. Свойство *ItemsSource* этого *ListBox* связано посредством привязки данных со свойством *Albums* объекта *ComposerInfo*. Это означает, что *ListBox* заполняется коллекцией объектов типа *AlbumInfo*, т.е. *DataTemplate* объекта *ListBox* определяет, как будут отображаться эти элементы:

```
<ListBox.ItemTemplate>
    <!-- Объекты типа AlbumInfo -->
    <DataTemplate>
        …
```

```
    </DataTemplate>
</ListBox.ItemTemplate>
```

Этот *DataTemplate* использует свойства *ThumbnailArt*, *ShortAlbumName* и *Album* объекта *AlbumInfo*.

Первые два снимка экрана приложения MusicByComposer, показанные ранее, являются исключительно продуктом файла MainPage.xaml и объектов данных, выполняющих роль источников привязки. Файлу выделенного кода для *MainPage* осталось лишь обработать формируемое *ListBox* событие *SelectionChanged* для перехода к AlbumPage.xaml:

Проект Silverlight: MusicByComposer Файл: MainPage.xaml.cs (фрагмент)

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    void OnListBoxSelectionChanged(object sender, SelectionChangedEventArgs args)
    {
        ComposerInfo composerInfo = pivot.SelectedItem as ComposerInfo;
        int composerInfoIndex =
MusicPresenter.Current.Composers.IndexOf(composerInfo);

        AlbumInfo albumInfo = (sender as ListBox).SelectedItem as AlbumInfo;
        int albumInfoIndex = composerInfo.Albums.IndexOf(albumInfo);

        // Создаем URI с двумя индексами и выполняем переход
        string destinationUri =

String.Format("/AlbumPage.xaml?ComposerInfoIndex={0}&AlbumInfoIndex={1}",
                              composerInfoIndex, albumInfoIndex);

        this.NavigationService.Navigate(new Uri(destinationUri, UriKind.Relative));
    }
}
```

Строка запроса состоит из двух индексов: индекс в коллекции *Composers* объекта *MusicPresenter* для обозначения текущего объекта *ComposerInfo*, и индекс свойства *Albums* объекта *ComposerInfo*, указывающий на выбранный *AlbumInfo*.

Код получает отображаемый в настоящий момент объект *ComposerInfo* через свойство *SelectedItem* элемента управления *Pivot*. Изначально я намеревался сохранять *SelectedIndex* элемента управления *Pivot* при захоронении, чтобы иметь возможность восстановить представление *MainPage* при возвращении к приложению. Однако у меня возникли проблемы с заданием *SelectedIndex* для вновь создаваемого элемента управления *Pivot*, поэтому пока я отказался от этой идеи, т.е. после захоронения данное приложение всегда возвращается к отображению альбомов Джона Адамса на *MainPage*.

Вызов метода *Navigate* создает экземпляр *AlbumPage*, который обеспечивает отображение альбома. *AlbumPage* – это обычный производный от *PhoneApplicationPage* с двумя обычными заголовками. (В качестве заголовка страницы используется имя композитора из кода.) Область содержимого XAML-файла предполагает, что значением *DataContext* объекта *AlbumPage* задан экземпляр *AlbumInfo*. (Это также определено в коде.) В первой строке сетки для содержимого располагается изображение обложки альбома, название альбома и исполнитель. Вторая строка – это *ScrollViewer* с *ItemsControl* для отображения песен:

Проект Silverlight: MusicByComposer Файл: AlbumPage.xaml (фрагмент)

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Border Grid.Row="0" Grid.Column="0"
            BorderBrush="{StaticResource PhoneForegroundBrush}"
            BorderThickness="1"
            Height="200" Width="200"
            Margin="0 0 6 0">

       <Image Source="{Binding AlbumArt}" />

    </Border>

    <StackPanel Grid.Row="0" Grid.Column="1"
                VerticalAlignment="Center">

        <TextBlock Text="{Binding Album.Name}"
                   Foreground="{StaticResource PhoneAccentBrush}"
                   TextWrapping="Wrap" />

        <TextBlock Text=" " />

        <TextBlock Text="{Binding Album.Artist}"
                   TextWrapping="Wrap" />
    </StackPanel>

    <ScrollViewer Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2">
        <ItemsControl ItemsSource="{Binding Album.Songs}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <local:SongTitleControl Song="{Binding}" />
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
    </ScrollViewer>
</Grid>
```

Обратите внимание, что значением свойства *ItemsSource* объекта *ItemsControl*, отображающего песни, задана коллекция *Songs* свойства *Album* объекта *AlbumInfo*. Это свойство *Songs* типа *SongCollection*. Оно включает объекты XNA-класса *Song*. Каждый объект *Song* этой коллекции является источником привязки для класса *SongTitleControl* (Элемент управления для названия песни), который мы рассмотрим чуть ниже.

AlbumPage.xaml также включает *ApplicationBar* для управления проигрывателем:

Проект Silverlight: MusicByComposer Файл: AlbumPage.xaml (фрагмент)

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.transport.rew.rest.png"
                                        Text="previous"
                                        Click="OnAppbarPreviousButtonClick" />
```

```
        <shell:ApplicationBarIconButton x:Name="appbarPlayPauseButton"

IconUri="/Images/appbar.transport.play.rest.png"
                                        Text="play"
                                        Click="OnAppbarPlayButtonClick" />

        <shell:ApplicationBarIconButton
IconUri="/Images/appbar.transport.ff.rest.png"
                                        Text="next"
                                        Click="OnAppbarNextButtonClick" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Музыкальные классы XNA: *MediaPlayer*

Для отображения музыки, хранящейся в музыкальной библиотеке, мы используем XNA-класс *MediaLibrary* и связанные с ним классы. Воспроизведение же музыки осуществляется с помощью статического XNA-класса *MediaPlayer* (Проигрыватель мультимедиа).

Класс *MediaPlayer* воспроизводит либо объект *Song*, либо все песни коллекции *SongCollection*, либо все песни коллекции *SongCollection*, начиная с заданного индекса. Такая функциональность обеспечивается тремя разновидностями статического метода *MediaPlayer.Play*.

Создать объект *SongCollection* самостоятельно невозможно. Для этого обязательно необходимо получить неизменный *SongCollection* от одного из классов (например, *Album*). Это означает, что обеспечить пользователю возможность выбирать определенные песни альбома или менять последовательность треков каким-то образом не так просто. Это потребует от приложения сохранения собственного списка объектов *Song* с последующим их воспроизведением. Для данного относительно простого демонстрационного приложения я решил не реализовывать ничего подобного.

Кроме *Play*, *MediaPlayer* также определяет методы *Pause*, *Resume* и *Stop*, а также методы *MovePrevious* (Перейти к предыдущему) и *MoveNext* (Перейти к следующему) для перехода к предыдущему или следующему элементу коллекции *SongCollection*.

Все самые важные свойства *MediaPlayer* являются свойствами только для чтения:

- *State*, которое возвращает один из членов перечисления *MediaState* (Состояние мультимедиа): *Playing* (Воспроизводится), *Paused* (Воспроизведение приостановлено) или *Stopped* (Воспроизведение остановлено).
- *PlayPosition* (Положение головки воспроизведения) – это объект *TimeSpan*, указывающий на положение головки воспроизведения в рамках воспроизводимой в настоящий момент песни.
- *Queue* (Очередь) – это объект *MediaQueue* (Очередь мультимедиа), включающий коллекцию объектов *Song* воспроизводимой в настоящий момент коллекции, а также свойство *ActiveSong* (Активная песня).

  Из свойства *ActiveSong* можно получить объект *Album* и другие данные, касающиеся этой песни.

  MediaPlayer также определяет два события:

- *MediaStateChanged* (Состояние мультимедиа изменилось)
- *ActiveSongChanged* (Активная песня изменилась)

Файл выделенного кода для *AlbumPage* отвечает за фактическое воспроизведение альбома. Но сначала давайте рассмотрим части класса, осуществляющего основные рутинные операции:

Проект Silverlight: MusicByComposer Файл: AlbumPage.xaml.cs (фрагмент)

```
public partial class AlbumPage : PhoneApplicationPage
{
    // Используется для переключения значков воспроизводить и приостановить
    static Uri playButtonIconUri =
                new Uri("/Images/appbar.transport.play.rest.png", UriKind.Relative);
    static Uri pauseButtonIconUri =
                new Uri("/Images/appbar.transport.pause.rest.png",
UriKind.Relative);

    int composerInfoIndex;
    int albumInfoIndex;

    public AlbumPage()
    {
        InitializeComponent();
        appbarPlayPauseButton = this.ApplicationBar.Buttons[1] as
ApplicationBarIconButton;
    }

    protected override void OnNavigatedFrom(NavigationEventArgs args)
    {
        PhoneApplicationService.Current.State["ComposerInfoIndex"] =
composerInfoIndex;
        PhoneApplicationService.Current.State["AlbumInfoIndex"] = albumInfoIndex;

        base.OnNavigatedFrom(args);
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Переход с MainPage
        if (this.NavigationContext.QueryString.ContainsKey("ComposerInfoIndex"))
        {
            composerInfoIndex =

Int32.Parse(this.NavigationContext.QueryString["ComposerInfoIndex"]);
            albumInfoIndex =
                Int32.Parse(this.NavigationContext.QueryString["AlbumInfoIndex"]);
        }

        // Повторная активация после захоронения
        else if
(PhoneApplicationService.Current.State.ContainsKey("ComposerInfoIndex"))
        {
            composerInfoIndex =
                (int)PhoneApplicationService.Current.State["ComposerInfoIndex"];
            albumInfoIndex =
                (int)PhoneApplicationService.Current.State["AlbumInfoIndex"];
        }

        ComposerInfo composerInfo =
MusicPresenter.Current.Composers[composerInfoIndex];
        AlbumInfo albumInfo = composerInfo.Albums[albumInfoIndex];

        // Задаем заголовок страницы и DataContext
        PageTitle.Text = composerInfo.Composer;
        this.DataContext = albumInfo;

        // Получаем состояние мультимедиа при его изменении и также прямо сейчас
        MediaPlayer.MediaStateChanged += OnMediaPlayerMediaStateChanged;
```

```
        OnMediaPlayerMediaStateChanged(null, EventArgs.Empty);

        base.OnNavigatedTo(args);
    }
    …
}
```

При захоронении метод *OnNavigatedFrom* сохраняет два поля: *composerInfoIndex* (Индекс сведений о композиторе) и *albumInfoIndex* (Индекс сведений об альбоме). Эти же два значения *MainPage* передает в *AlbumPage* посредством строки запроса для навигации. Метод *OnNavigatedTo* получает эти значения либо из строки запроса, либо из свойства *State* объекта *PhoneApplicationService* и использует их для задания элемента *PageTitle* (для отображения имени композитора) и *DataContext* страницы (чтобы обеспечить работоспособность привязок в AlbumPage.xaml).

Метод *OnNavigatedTo* также задает обработчик события *MediaPlayer.MediaStateChanged* для обеспечения отображения соответствующего значка кнопки, сочетающей функции Play и Pause.

Обработчик событий для этой кнопки оказался самым сложным аспектом данного класса:

Проект Silverlight: MusicByComposer Файл: AlbumPage.xaml.cs (фрагмент)

```
void OnAppbarPlayButtonClick(object sender, EventArgs args)
{
    Album thisPagesAlbum = (this.DataContext as AlbumInfo).Album;

    switch (MediaPlayer.State)
    {
        // MediaPlayer в настоящий момент осуществляет воспроизведение,
        // поэтому приостановить.
        case MediaState.Playing:
            MediaPlayer.Pause();
            break;

        // MediaPlayer в настоящий момент приостановлен...
        case MediaState.Paused:
            MediaQueue queue = MediaPlayer.Queue;

            // поэтому если мы находимся на той же странице,
            // что и приостановленная песня, возобновить ее воспроизведение.
            if (queue.ActiveSong != null &&
                queue.ActiveSong.Album == thisPagesAlbum)
            {
                MediaPlayer.Resume();
            }
            // в противном случае начать воспроизведение альбома данной страницы.
            else
            {
                goto case MediaState.Stopped;
            }
            break;

        // Воспроизведение в MediaPlayer остановлено,
        // начать воспроизведение альбома данной страницы.
        case MediaState.Stopped:
            MediaPlayer.Play(thisPagesAlbum.Songs);
            break;
    }
}

void OnAppbarPreviousButtonClick(object sender, EventArgs args)
{
```

```
    MediaPlayer.MovePrevious();
}

void OnAppbarNextButtonClick(object sender, EventArgs args)
{
    MediaPlayer.MoveNext();
}
```

После того как приложение вызывает метод *MediaPlayer.Play* объекта *Song* или *SongCollection*, воспроизведение продолжается, даже если пользователь закрывает приложение, или телефон выключает и блокирует экран. Именно так и должно быть. Пользователь хочет слушать музыку несмотря ни на что до тех пор, пока не кончится заряд аккумулятора.

По этой причине с большой осторожностью следует относиться к вызову метода *MediaPlayer.Stop*, поскольку это приведет к остановке воспроизведения без возможности его возобновления. Я вообще не нашел оснований для вызова метода *MediaPlayer.Stop* в своем приложении.

Приложения, такие как MusicByComposer, должны позволять пользователю выходить из приложения и возвращаться в него, а также переходить к страницам разных альбомов, не оказывая влияния на воспроизведение музыки. При этом пользователь также должен иметь возможность переключения от воспроизводимой в настоящий момент музыки к текущему альбому. Я считаю, что все эти опции можно реализовать как четыре варианта поведения по нажатию кнопки play/pause:

- Если музыка воспроизводится, на кнопке play/pause отображается значок паузы, и по нажатию этой кнопки воспроизведение должно быть приостановлено.
- Если воспроизведения не осуществляется, кнопка play/pause отображает значок воспроизведения, и по ее нажатию должно начинаться воспроизведение просматриваемого альбома.
- Если воспроизведение приостановлено, на кнопке play/pause также отображается значок воспроизведения. Если пользователь находится на странице активного в настоящий момент альбома, нажатие кнопки воспроизведения должно просто обеспечить возобновление воспроизведения приостановленной до этого композиции.
- Однако если воспроизведение приостановлено, но пользователь находится на странице *другого* альбома, по нажатию кнопки воспроизведения должно начаться воспроизведение альбома текущей страницы.

На практике эта логика работает хорошо.

Вне нашего внимания остался лишь класс *SongTitleControl*, экземпляр которого используется для отображения каждой отдельной песни альбома. *SongTitleControl* также отвечает за выделение названия песни, которая воспроизводится в настоящий момент, и отображение истекшего времени и общей продолжительности этой песни.

SongTitleControl наследуется от *UserControl* и имеет простое дерево визуальных элементов:

Проект Silverlight: MusicByComposer Файл: SongTitleControl.xaml (фрагмент)

```
<Grid x:Name="LayoutRoot">
    <StackPanel Margin="0 3">
        <TextBlock Name="txtblkTitle"
                   Text="{Binding Name}"
                   TextWrapping="Wrap" />
```

```
        <TextBlock Name="txtblkTime"
                   Margin="24 6"
                   Visibility="Collapsed" />
    </StackPanel>
</Grid>
```

В файле AlbumPage.xaml класс *SongTitleControl* включает привязку к свойству *Song*. Это означает, что *SongTitleControl* должен определять свойство-зависимость *Song* XNA-типа *Song*. Рассмотрим описание этого свойства *Song* и обработчиков изменения его значения:

Проект Silverlight: MusicByComposer Файл: SongTitleControl.xaml.cs (фрагмент)

```
public static readonly DependencyProperty SongProperty =
    DependencyProperty.Register("Song",
        typeof(Song),
        typeof(SongTitleControl),
        new PropertyMetadata(OnSongChanged));

…

public Song Song
{
    set { SetValue(SongProperty, value); }
    get { return (Song)GetValue(SongProperty); }
}

static void OnSongChanged(DependencyObject obj, DependencyPropertyChangedEventArgs
args)
{
    (obj as SongTitleControl).OnSongChanged(args);
}

void OnSongChanged(DependencyPropertyChangedEventArgs args)
{
    if (Song != null)
        MediaPlayer.ActiveSongChanged += OnMediaPlayerActiveSongChanged;
    else
        MediaPlayer.ActiveSongChanged -= OnMediaPlayerActiveSongChanged;

    OnMediaPlayerActiveSongChanged(null, EventArgs.Empty);
}
```

Если *Song* задано значение, отличное от *null*, задается обработчик для событий *MediaPlayer.ActiveSongChanged*. Рассмотрим этот обработчик событий:

Проект Silverlight: MusicByComposer Файл: SongTitleControl.xaml.cs (фрагмент)

```
void OnMediaPlayerActiveSongChanged(object sender, EventArgs args)
{
    if (this.Song == MediaPlayer.Queue.ActiveSong)
    {
        txtblkTitle.FontWeight = FontWeights.Bold;
        txtblkTitle.Foreground = this.Resources["PhoneAccentBrush"] as Brush;
        txtblkTime.Visibility = Visibility.Visible;
        timer.Start();
    }
    else
    {
        txtblkTitle.FontWeight = FontWeights.Normal;
        txtblkTitle.Foreground = this.Resources["PhoneForegroundBrush"] as Brush;
        txtblkTime.Visibility = Visibility.Collapsed;
        timer.Stop();
```

```
    }
}
```

Свойство *Text* объекта *txtblkTitle* обрабатывается с помощью привязки в XAML-файле. Если активной песней является *Song*, ассоциированный с экземпляром *SongTitleControl*, этот *TextBlock* выделяется контрастным цветом, становится видимым другой *TextBlock* с информацией о времени воспроизведения, и запускается *DispatcherTimer*:

Проект Silverlight: MusicByComposer Файл: SongTitleControl.xaml.cs (фрагмент)

```
public partial class SongTitleControl : UserControl
{
    DispatcherTimer timer = new DispatcherTimer();
    …
    public SongTitleControl()
    {
        InitializeComponent();
        timer.Interval = TimeSpan.FromSeconds(0.25);
        timer.Tick += OnTimerTick;
    }
    …
    void OnTimerTick(object sender, EventArgs args)
    {
        TimeSpan dur = this.Song.Duration;
        TimeSpan pos = MediaPlayer.PlayPosition;

        txtblkTime.Text = String.Format("{0}:{1:D2} / {2}:{3:D2}",
                                        (int)pos.TotalMinutes, pos.Seconds,
                                        (int)dur.TotalMinutes, dur.Seconds);
    }
}
```

Обработчик событий *Tick* просто форматирует для целей отображения длительность песни и время, соответствующее текущему положению головки воспроизведения.

Я думал о переносе части этого кода в XAML, для чего потребовалось бы определить свойство для истекшего времени, а также использовать Visual State Manager для состояний ActiveSong и NotActiveSong (Неактивная песня), и затем ввести конвертер *StringFormatterConverter* для форматирования двух объектов *TimeSpan*. Но для данного конкретного приложения использование файла кода мне показалось более простым решением.

Мы видели множество примеров мощи XAML, но иногда код является по-настоящему правильным решением.

Часть III

XNA

Глава 19
Принципы движения

Приложение на XNA, в сущности, состоит в реализации перемещения спрайтов по экрану. Иногда перемещением этих спрайтов управляет пользователь, иногда они движутся самостоятельно, как будто под действием некоторой скрытой силы. Перемещаться могут не только спрайты, но и текст, и именно текстом мы и займемся в этой главе. Концепции и стратегии перемещения текста по экрану аналогичны перемещению спрайтов.

Эффект перемещения текста создается путем задания различных координат в методе *DrawString* в ходе последовательных вызовов метода *Draw* класса *Game*. В главе 1, как мы помним, переменной *textPosition* просто задавалось фиксированное значение в ходе выполнения метода *LoadContent*. Данный код обеспечивает размещение текста в центре экрана:

```
Vector2 textSize = segoe14.MeasureString(text);
Viewport viewport = this.GraphicsDevice.Viewport;
textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                           (viewport.Height - textSize.Y) / 2);
```

В большинстве приложений данной главы значение *textPosition* пересчитывается при каждом вызове *Update*. Благодаря этому метод *Draw* каждый раз отрисовывает текст в разных местах. Обычно ничего сверхъестественного не происходит, текст просто перемещается по экрану сверху вниз, затем снизу вверх и опять вниз. Как в известном анекдоте: «Нанести, смыть, повторить».

Я начну с самого простого подхода в реализации перемещения текста и затем буду его дорабатывать. Тем, кто не привык оперировать категориями векторных и параметрических уравнений, сначала может показаться, что мои дополнения только все усложняют, но в итоге вы убедитесь, что приложение стало более гибким и простым.

Простейший подход

Начнем с самого простого: просто переместим текст вверх и вниз по вертикали, реализовывая, таким образом, перемещение в одном измерении. Для этого достаточно последовательно увеличивать, а затем уменьшать координату *Y* значения *textPosition*.

Чтобы поэкспериментировать, создадим в Visual Studio проект под именем NaiveTextMovement (Простое перемещение текста) и добавим в папку Content шрифт Segoe UI Mono размером 14 пунктов. Поля класса *Game1* описываются следующим образом:

Проект XNA: NaiveTextMovement Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // пикселов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Viewport viewport;
    Vector2 textSize;
    Vector2 textPosition;
    bool isGoingUp = false;
        …
```

```
    }
}
```

Здесь ничего не должно вызывать удивления. И SPEED, и TEXT определены как константы. Для SPEED задано значение 240 пикселов в секунду. Поле *isGoingUp* (Перемещается вверх) типа Boolean указывает на направление перемещения текста в настоящий момент, вверх или вниз.

Метод *LoadContent* хорошо знаком нам из приложения главы 1. Единственное отличие в данном случае в том, что окно просмотра сохраняется как поле:

Проект XNA: NaiveTextMovement   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    textSize = segoe14.MeasureString(TEXT);
    textPosition = new Vector2(viewport.X + (viewport.Width - textSize.X) / 2, 0);
}
```

Обратите внимание, что значение *textPosition* обеспечивает центрирование текста по горизонтали, но размещает его вверху экрана. Все основные вычисления, как и в большинстве приложений на XNA, осуществляются в методе *Update*:

Проект XNA: NaiveTextMovement   Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (!isGoingUp)
    {
        textPosition.Y += SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
        if (textPosition.Y + textSize.Y > viewport.Height)
        {
            float excess = textPosition.Y + textSize.Y - viewport.Height;
            textPosition.Y -= 2 * excess;
            isGoingUp = true;
        }
    }
    else
    {
        textPosition.Y -= SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;

        if (textPosition.Y < 0)
        {
            float excess = - textPosition.Y;
            textPosition.Y += 2 * excess;
            isGoingUp = false;
        }
    }

    base.Update(gameTime);
}
```

Аргумент *GameTime* (Время игры) метода *Update* имеет два критически важных свойства типа *TimeSpan*: *TotalGameTime* (Общее время игры) и *ElapsedGameTime* (Истекшее время игры).

«Время игры» не всегда синхронизируется с реальным временем, существует несколько аппроксимаций, которые обеспечивают плавность анимаций, но оно близко к нему. *TotalGameTime* отражает время, прошедшее с начала игры. *ElapsedGameTime* – это время с момента предыдущего вызова метода *Update*. В общем случае *ElapsedGameTime* всегда будет иметь одно и то же значение – 33-1/3 миллисекунды, что соответствует частоте обновления экрана телефона 30 Гц.

Для задания темпа перемещения можно использовать *TotalGameTime* или *ElapsedGameTime*. В данном примере при первом вызове *Update* вычисляется *textPosition*, обеспечивая размещение текста у верхнего края экрана, и *isGoingUp* задается значение false. Приращение *textPosition.Y* рассчитывается на основании значения SPEED (которое выражается в пикселах в секунду) и общего времени в секундах, истекшего с момента последнего вызова *Update*, что будет составлять 1/30 секунды.

Такие вычисления могут приводить к перемещению текста на слишком большую дистанцию и выходу его за границы экрана. В этом случае сумма координаты текста по вертикали и его высоты будет больше значения свойства *Bottom* высоты окна просмотра. Чтобы предотвратить такую ситуацию, мною предусмотрено так называемое *превышение*. Это величина, на которую координата текста по вертикали превышает размер экрана. Для компенсации я умножаю величину превышения на два, чтобы создать эффект, как будто текст «отскакивает» вверх от нижнего края экрана на величину этого превышения. И в этой точке свойству *isGoingUp* присваивается значение *true*.

Логика перемещения вверх, как я люблю говорить, абсолютно такая же, но полностью противоположна. Перегруженный метод *Draw* прост:

Проект XNA: NaiveTextMovement Файл: Game1.cs (фрагмент)

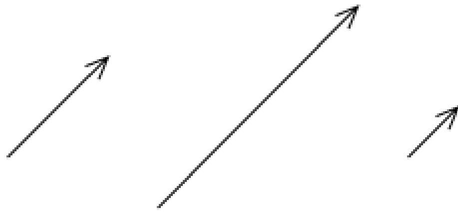
```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Большая проблема в том, что этот простой подход не включает никакого математического инструментария, который позволил бы нам сделать что-либо боле сложное, например, переместить текст по диагонали, а не просто в одном направлении.

В приложении NaiveTextMovement нет никакой концепции направления, которая позволила бы уйти от просто движения по горизонтали или вертикали. Для этого нам необходимы векторы.

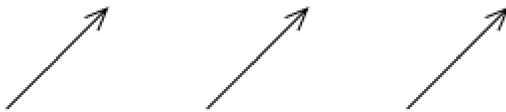
Краткий обзор векторов

Вектор – это математическая сущность, объединяющая в себе концепции как направления, так и величины. Обычно вектор представляют в виде линии со стрелкой. Данные три вектора имеют одинаковое направление, но различную величину (или модуль):



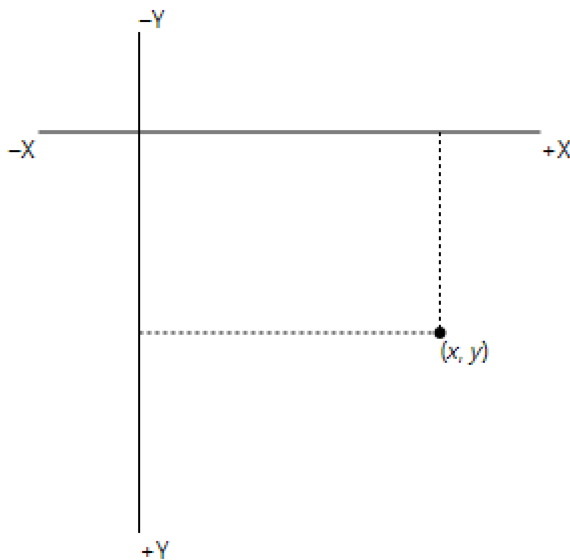
Эти три вектора одинаковые по модулю, но направлены по-разному:

Модуль и направление этих трех векторов одинаковые, т.е. данные векторы считаются тождественными:



У вектора нет месторасположения, поэтому даже если эти три вектора зрительно располагаются в разных местах, на самом деле у них нет никакого определенного местоположения.

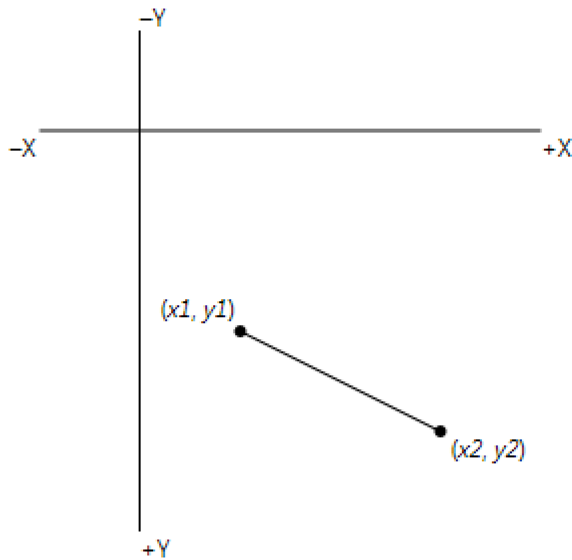
У точки нет величины и нет размера. Точка – это *просто* координата. В двухмерном пространстве точка описывается числовой парой (*x*, *y*), представляющей расстояние по вертикали и по горизонтали от начала координат (0, 0) до этой точки:



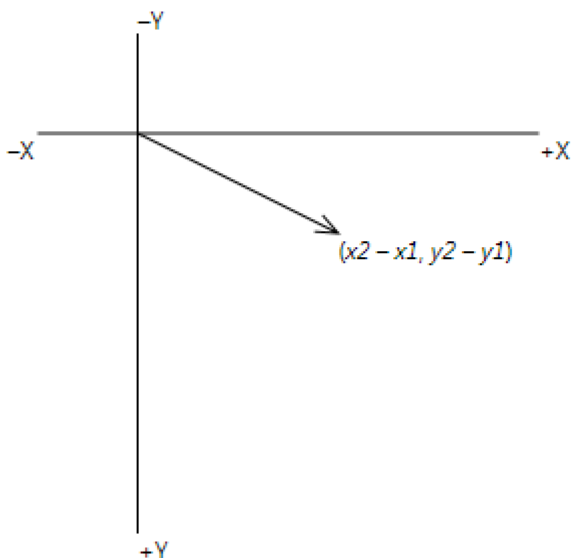
На рисунке значения по оси Y увеличиваются в направлении вниз, что соответствует принятой в XNA двухмерной системе координат. (В трехмерном XNA иначе.)

У вектора есть величина и размер, но нет местоположения. Но, как и точка, вектор представляется числовой парой (*x*, *y*), только обычно на письме вектор выделяют жирным шрифтом, ***(x, y)***, чтобы отличить его от точки.

Как так получилось, что и точка в двухмерном пространстве, и вектор обозначаются одинаково? Рассмотрим две точки, (*x1*, *y1*) и (*x2*, *y2*), и линию, проведенную из первой точки во вторую:



Длина и направление этой линии аналогичны длине и направлению линии, проведенной из начала координат в точку (*x2 – x1*, *y2 – y1*):



Эту величину и направление определяет вектор ***(x2 – x1, y2 – y1)***.

По этой причине для хранения двухмерных координатных точек и двухмерных векторов XNA использует одну и ту же структуру: *Vector2*. (В XNA также имеется структура *Point*, но поля *X* и *Y* в ней типа *integer*.)

Модуль вектора **(*x*, *y*)** – это длина линии, проведенной из точки (0, 0) в точку (*x*, *y*). Длину линии и вектор можно вычислить с помощью теоремы Пифагора, которая поистине является самым полезным инструментом в программировании компьютерной графики:

$$\text{длина} = \sqrt{x^2 + y^2}$$

Структура *Vector2* определяет метод *Distance* (Расстояние), который будет осуществлять это вычисление. *Vector2* также включает метод *DistanceSquared* (Расстояние в квадрате), в котором за более длинным именем кроется более простое вычисление. Вероятнее всего, структура *Vector2* будет реализовывать *DistanceSquared* следующим образом:

```
public float DistanceSquare()
{
    return x * x + y * y;
}
```

Тогда метод *Distance* использует *DistanceSquared*:

```
public float Distance()
{
    return (float)Math.Sqrt(DistanceSquare());
}
```

Если требуется лишь сравнить модули двух векторов, используйте *DistanceSquared*, потому что он более производительный. В контексте работы с объектами *Vector2* понятия «длина», «расстояние» и «модуль» могут использоваться взаимозаменяемо.

Поскольку *Vector2* предполагается использовать для представления точек, векторов и размеров, эта структура обеспечивает достаточную гибкость для осуществления разнообразных арифметических действий, но только от разработчика зависит их правильность и целесообразность. Например, предположим, что *point1* и *point2* являются объектами типа *Vector2*, но используются для представления точек. Нет никакого смысла складывать эти точки, несмотря на то что *Vector2* позволит сделать это. А вот если вычесть одну точку из другой, мы получим вектор:

```
Vector2 vector = point2 - point1;
```

Данная операция просто обеспечивает вычитание значений *X* и значений *Y*. Вектор направлен из точки *point1* в точку *point2*, и его модуль равен расстоянию между этими точками. Также часто складывают вектор и точку:

```
Vector2 point2 = point1 + vector;
```

Данная операция позволяет получить точку, располагающуюся от заданной на определенном расстоянии и в определенном направлении. Вектор можно умножить на число. Если *vector* является объектом типа *Vector2*, тогда:

```
vector *= 5;
```

что эквивалентно:

```
vector.X *= 5;
vector.Y *= 5;
```

Данная операция эффективно увеличивает модуль вектора в 5 раз. Аналогичным образом вектор можно разделить на число. Если разделить вектор на его длину, длина результирующего вектора будет равна 1. Это так называемый *нормализованный* вектор, и в структуре *Vector2* имеется метод *Normalize* (Нормализовать) специально для получения такого вектора. Выражение:

```
vector.Normalize();
```

эквивалентно

```
vector /= vector.Distance();
```

Часто удобнее с помощью статического метода *Vector.Normalize* создать нормализованный вектор из другого вектора:

```
Vector normalizedVector = Vector.Normalize(vector)
```

Нормализованный вектор представляет только направление без величины, но его можно умножить на число, чтобы задать ему соответствующую длину.

Если *vector* имеет определенную длину и направление, то –*vector* имеет такую же длину, но направлен в противоположную сторону. Я применю эту операцию в следующем приложении.

Вектор **(*x*, *y*)** направлен из точки (0, 0) в точку (*x*, *y*). Это направление можно преобразовать в угол с помощью второго полезного инструмента программирования компьютерной графики – метода *Math.Atan2*:

```
float angle = (float)Math.Atan2(vector.Y, vector.X);
```

Обратите внимание, что компонент *Y* задается первым. Угол измеряется в радианах (вспомним, что в 360 градусах 2π радиан) и отсчитывается по часовой стрелке от положительного направления оси X.

Имея угол в радианах, из него можно получить нормализованный вектор следующим образом:

```
Vector2 vector = new Vector2((float)Math.Cos(angle),
                             (float)Math.Sin(angle));
```

Структура *Vector2* имеет четыре статических свойства: *Vector2.Zero* возвращает объект *Vector2*, *X* и *Y* которого имеют нулевые значения. На самом деле, это недействительный вектор, поскольку не имеет направления, но он пригодится для представления точки начала координат. *Vector2.UnitX* – это нормализованный вектор **(1, 0)**, т.е. он определяет направление, прямо противоположное направлению оси X. И *Vector2.UnitY* – это вектор **(0, 1)**, направленный вверх. *Vector2.One* – это точка (1, 1) или вектор **(1, 1)**, который пригодится при использовании *Vector2* в качестве коэффициента масштабирования в горизонтальном и вертикальном направлениях (как я делаю далее в этой главе).

Перемещение спрайтов с помощью векторов

Небольшой курс повышения квалификации обеспечит достаточно знаний, чтобы доработать приложение по перемещению текста и использовать в нем векторы. Этот проект Visual Studio называется VectorTextMovement (Перемещение текста с помощью векторов). Рассмотрим поля, описываемые в нем:

Проект XNA: VectorTextMovement Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // пикселов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Vector2 midPoint;
    Vector2 pathVector;
```

```
    Vector2 pathDirection;
    Vector2 textPosition;
    …
}
```

Текст будет перемещаться между двумя точками (которые в методе *LoadContent* называются *position1* и *position2*). Поле *midPoint* (Средняя точка) сохраняет координаты точки, лежащей на средине пути между этими двумя точками. Поле *pathVector* (Вектор пути) – это вектор из *position1* в *position2*; и *pathDirection* (Направление пути) – это нормализованный *pathVector*.

Все эти поля вычисляются и инициализируются в методе *LoadContent*:

Проект XNA: VectorTextMovement   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Viewport viewport = this.GraphicsDevice.Viewport;

    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    Vector2 textSize = segoe14.MeasureString(TEXT);

    Vector2 position1 = new Vector2(viewport.Width - textSize.X, 0);
    Vector2 position2 = new Vector2(0, viewport.Height - textSize.Y);
    midPoint = Vector2.Lerp(position1, position2, 0.5f);

    pathVector = position2 - position1;
    pathDirection = Vector2.Normalize(pathVector);
    textPosition = position1;
}
```

Начальная точка – *position1*, т.е. текст помещается в верхний правый угол. Точка *position2* соответствует нижнему левому углу. Для вычисления *midPoint* используется статический метод *Vector2.Lerp*, имя которого расшифровывается как Linear interpolation (линейная интерполяция). Если третий аргумент равен 0, *Vector2.Lerp* возвращает свой первый аргумент; если третий аргумент равен 1, *Vector2.Lerp* возвращает свой второй аргумент; для всех промежуточных значений метод выполняет линейную интерполяцию. Наверное, использовать *Lerp* для вычисления средней точки это слишком, потому что требуется всего лишь найти среднее для двух значений *X* и двух значений *Y*.

Обратите внимание, что *pathVector* – это вектор из точки *position1* в точку *position2*, тогда как *pathDirection* – это тот же вектор, но нормализованный. Метод завершается инициализацией *textPosition* значением *position1*. Назначение этих полей должно стать очевидным в методе *Update*:

Проект XNA: VectorTextMovement   Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float pixelChange = SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
    textPosition += pixelChange * pathDirection;

    if ((textPosition - midPoint).LengthSquared() > (0.5f *
pathVector).LengthSquared())
    {
        float excess = (textPosition - midPoint).Length()- (0.5f *
```

```
pathVector).Length();
        pathDirection = -pathDirection;
        textPosition += 2 * excess * pathDirection;
    }

    base.Update(gameTime);
}
```

При первом вызове *Update* поле *textPosition* имеет значение *position1*, и *pathDirection* является нормализованным вектором из точки *position1* в точку *position2*. Самое главное вычисление:

```
textPosition += pixelChange * pathDirection;
```

Результатом умножения нормализованного *pathDirection* на *pixelChange* (Пройденное расстояние в пикселах) является вектор, направленный так же, как и *pathDirection*, но имеющий длину *pixelChange*. Значение *textPosition* увеличивается на эту величину.

Через несколько секунд приращений *textPosition* выйдет за границу, обозначенную *position2*. Это можно определить, когда длина вектора из *midPoint* в *textPosition* превысит длину половины *pathVector*. В этот момент направление движения необходимо изменить на обратное: знак значения *pathDirection* меняется на обратный, и *textPosition* настраивается для обеспечения отскока.

Обратите внимание, что больше нет необходимости определять, движется текст вверх или вниз. Вычисления с участием *textPosition* и *midPoint* обрабатывают оба случая. Также заметьте, что выражение *if* выполняет сравнение на основании *LengthSquared*, но для расчета *excess* требуется метод *Length*. Поскольку условие *if* проверяется для каждого вызова *Update*, код должен быть максимально эффективным. Длина половины *pathVector* неизменна, поэтому я мог бы обеспечить даже большую эффективность, если бы сохранял *Length* или *LengthSquared* (или оба) как поля.

Метод *Draw* ничем не отличается от предыдущего примера:

Проект XNA: VectorTextMovement Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Параметрические уравнения

Всем хорошо известно, что когда преподаватель математики или физики говорит: «А теперь, чтобы упростить, введем новую переменную,» – никто не верит в то, что действительно что-то упростится. Но очень часто именно так и происходит, и именно в этом ценность параметрических уравнений. В по внешнему виду сложную систему формул вводится новая переменная, которую часто называют просто *t*, как будто подразумевая *время* (*time*). Значение *t* обычно лежит в диапазоне от 0 до 1 (хотя это всего-лишь принятая условность), и другие переменные вычисляются на основании *t*. И что поразительно, это действительно все упрощает.

Давайте подойдем к решению проблемы перемещения текста по экрану в категориях «цикла». Один цикл включает перемещение текста из верхнего правого угла (*position1*) в нижний левый угол (*position2*) и назад в *position1*.

Сколько длится этот цикл? Продолжительность цикла можно легко подсчитать, зная постоянную скорость, которая выражена в пикселах в секунду, и дистанцию перемещения, которая равна удвоенному модулю вектора, называемого *pathVector* в предыдущем приложении. Этот вектор вычислялся как *position2 – position1*.

Когда известна скорость в циклах в секунду, можно без труда вычислить значение переменной *tLap*, лежащее в диапазоне от 0 до 1, где 0 соответствует началу цикла, и 1 – концу, под достижении которого цикл опять начинается с 0. Из *tLap* мы можем получить *pLap*, что является относительным положением в цикле. Диапазон его допустимых значений также от 0 до 1, где 0 – положение вверху экрана или *position1*, и 1 – положение внизу экрана или *position2*. Из *pLap* несложно вычислить *textPosition*. Следующая таблица представляет взаимоотношения между этими тремя переменными:

tLap:	0	0.5	1
pLap:	0	1	0
textPosition:	position1	position2	position1

Вероятно, сразу же можно увидеть, что

```
textPosition = position1 + pLap * pathVector;
```

где *pathVector* (как и в предыдущем приложение) равен разности между *position2* и *position1*. Единственным действительно сложным моментом является вычисление *pLap* на основании *tLap*.

Проект ParametricTextMovement (Параметрическое перемещение текста) включает следующие поля:

Проект XNA: ParametricTextMovement Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 240f;           // пикселов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Vector2 position1;
    Vector2 pathVector;
    Vector2 textPosition;
    float lapSpeed;                     // циклов в секунду
    float tLap;
    …
}
```

Единственными новыми переменными здесь являются *lapSpeed* (Скорость цикла) и *tLap*. Как уже стало привычным, большинство переменных задается в ходе выполнения метода *LoadContent*:

Проект XNA: ParametricTextMovement Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Viewport viewport = this.GraphicsDevice.Viewport;

    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    Vector2 textSize = segoe14.MeasureString(TEXT);
    position1 = new Vector2(viewport.Width - textSize.X, 0);
    Vector2 position2 = new Vector2(0, viewport.Height - textSize.Y);
    pathVector = position2 - position1;

    lapSpeed = SPEED / (2 * pathVector.Length());
}
```

В выражении для *lapSpeed* числитель выражается в пикселах в секунду. Знаменатель – это дистанция, преодолеваемая в ходе всего цикла, что составляет две длины *pathVector*. Таким образом, знаменатель выражается в пикселах в цикл. В результате деления пикселов в секунду на пикселы в цикл получаем циклы в секунду.

Одним из больших преимуществ данной методики с применением параметров – исключительная элегантность метода *Update*:

Проект XNA: ParametricTextMovement Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    tLap += lapSpeed * (float)gameTime.ElapsedGameTime.TotalSeconds;
    tLap %= 1;
    float pLap = tLap < 0.5f ? 2 * tLap : 2 - 2 * tLap;
    textPosition = position1 + pLap * pathVector;

    base.Update(gameTime);
}
```

Приращение значения поля *tLap* соответствует произведению *lapSpeed* на истекшее время в секундах. При втором вычислении от полученного значения отбрасывается целая часть, таким образом, если *tLap* увеличивается до 1,1, к примеру, то он возвращается к значению 0,1.

Соглашусь с тем, что вычисление *pLap* из *tLap*, которое является преобразованием, в некотором роде, на первый взгляд кажется абсолютно запутанным. Но если разложить все по полочкам, окажется, нет ничего сложного: если *tLap* меньше 0,5, тогда *pLap* равен двум *tLap*, т.е. для *tLap* от 0 до 0,5 значения *pLap* меняются от 0 до 1. Если *tLap* больше или равен 0,5, *tLap* удваивается и вычитается из 2, поэтому для *tLap* от 0,5 до 1 значения *pLap* меняются от 1 обратно к 0.

Метод *Draw* остается неизменным:

Проект XNA: ParametricTextMovement Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White);
    spriteBatch.End();
```

```
    base.Draw(gameTime);
}
```

Существуют различные эквивалентные способы выполнения этих вычислений. Вместо сохранения *pathVector* как поля, можно было бы сохранять *position2*. Тогда в ходе выполнения метода *Update* вычисление *textPosition* выполнялось бы с использованием метода *Vector2.Lerp*:

```
textPosition = Vector2.Lerp(position1, position2, pLap);
```

В методе *Update* вместо вычисления приращения *tLap* можно находить *tLap* прямо из *TotalGameState* (Общее состояние игры) аргумента *GameTime* и сохранять переменную локально:

```
float tLap = (lapSpeed * (float)gameTime.TotalGameTime.TotalSeconds) % 1;
```

Возможности, обеспечиваемые функцией преобразования

Теперь я хочу изменить одно выражение в приложении ParametricTextMovement и значительно улучшить его, сделав перемещение текста более естественным и плавным. Можно ли этого добиться? Безусловно!

Ранее я уже приводил данную таблицу:

tLap:	0	0.5	1
pLap:	0	1	0
textPosition:	position1	position2	position1

В проекте ParametricTextMovement я предположил, что функция преобразования между *tLap* и *pLap* будет линейной, т.е. такой:

```
float pLap = tLap < 0.5f ? 2 * tLap : 2 - 2 * tLap;
```

Но это не является обязательным условием. Проект VariableTextMovement (Разнообразное перемещение текста) аналогичен ParametricTextMovent за исключением того, как вычисляется *pLap*. В данном проекте это делается следующим образом:

```
float pLap = (1 - (float)Math.Cos(tLap * MathHelper.TwoPi)) / 2;
```

Когда *tLap* равен 0, косинус равен 1 и *pLap* равен 0. Когда *tLap* равен 0,5, аргументом функции косинуса является π радиан (180 градусов), т.е. косинус равен –1. Это значение вычитается из 1, и результат делится на 2, т.е. получаем 1. И так далее. Кажется, небольшое изменение, но разница огромна: теперь при приближении к углам текст замедляется и ускоряется, когда отдаляется от них.

Можно также попробовать несколько других функций. Эта обеспечивает замедление только при приближении к низу экрана:

```
float pLap = (float)Math.Sin(tLap * Math.PI);
```

Вверху же экрана текст перемещается с максимальной скоростью и, кажется, отскакивает рикошетом от края. Следующая функция обеспечивает обратное поведение и создает эффект того, что подпрыгивающий мяч замедляется вследствие действия гравитации при приближении к верхней границе экрана:

```
float pLap = 1 - Math.Abs((float)Math.Cos(tLap * Math.PI));
```

Как видите, это правда: использование параметрических уравнений не только упрощает код, но и намного упрощает его усовершенствование.

Изменение размера текста

В документации класса *SpriteBatch* можно найти еще пять версий метода *DrawString*. До сих пор я использовал такой вариант:

```
DrawString(spriteFont, text, position, color);
```

Есть еще два варианта этого метода:

```
DrawString(spriteFont, text, position, color, rotation, origin, uniformScale,
effects, layerDepth);

DrawString(spriteFont, text, position, color, rotation, origin, vectorScale, effects,
layerDepth);
```

Остальные три версии метода *DrawString* отличаются лишь вторым аргументом: используется *StringBuilder* вместо *string*. В случае необходимости отображения часто изменяющегося текста лучше использовать *StringBuilder*. Это позволит избежать частого распределения памяти из локальной кучи.

Дополнительные аргументы этих более длинных версий метода *DrawString* предусмотрены, главным образом, для обеспечения вращения, масштабирования и переворачивания текста. Исключением является последний аргумент: это значение с *плавающей точкой*, которое определяет порядок размещения спрайтов, начиная от самого верхнего (0) до самого нижнего (1). Я не буду использовать этот аргумент в *DrawString*.

Предпоследним аргументом является член перечисления *SpriteEffects* (Эффекты). По умолчанию используется *None*. Члены *FlipHorizontally* (Перевернуть по горизонтали) и *FlipVertically* (Перевернуть по вертикали) создают зеркальное отображение, но не меняют местоположение текста:

SpriteEffects.None

На самом деле, все это одна и та же строка, только она развернута на 180° в разных направлениях.

Аргумент *origin* (центр) – это точка, значением по умолчанию которой является (0, 0). Этот аргумент используется для трех взаимосвязанных целей:

- Для обозначения точки относительно строки текста, которая выровнена относительно экрана с помощью аргумента *position*.
- Для обозначения центра вращения. Аргумент *rotation* – это угол в радианах, отсчитываемый по часовой стрелке.
- Это центр масштабирования. Масштаб может быть задан одним числом, что обеспечивает одинаковое масштабирование в горизонтальном и вертикальном направлениях с сохранением пропорций; ила масштаб задается с помощью *Vector2*, что позволяет применять разные коэффициенты масштабирования в разных направлениях.

(Иногда эти два режима масштабирования называют изотропным – равный во всех направлениях – и анизотропным.)

Если используются развернутые версии *DrawString* и нет необходимости в масштабировании, не присваивайте этому аргументу значение нуль! Текст или спрайт с нулевым масштабом не будет виден на экране, и вам придется потратить множество часов, выясняя, что не так. (Я говорю, исходя из собственного опыта.) Если масштабирования не требуется, задайте в качестве значения этого аргумента 1 или статическое свойство *Vector2.One*.

В самом первом приложении на XNA этой книги выполнялось вычисление переменной *textPosition*, исходя из размеров экрана и размеров текста:

```
textPosition = new Vector2((viewport.Width - textSize.X) / 2,
                           (viewport.Height - textSize.Y) / 2);
```

textPosition – это точка экрана, в которой располагается верхний левый угол текста. Развернутые версии метода *DrawString* обеспечивают некоторые альтернативные варианты. Например:

```
textPosition = new Vector2(viewport.Width / 2, viewport.Height / 2);
origin = new Vector2(textSize.X / 2, textSize.Y / 2);
```

Теперь *textPosition* определяет центр экрана, и *origin* определяет центр текста. Данный вызов *DrawString* использует эти две переменные для размещения текста в центре экрана:

```
spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White,
                       0, origin, 1, SpriteEffects.None, 0);
```

В качестве значения *textPosition* может быть задан нижний правый угол экрана, и для *origin* может быть зада нижний правый угол текста:

```
textPosition = new Vector2(viedwport.Width, viewport.Height);
origin = new Vector2(textSize.X, textSize.Y);
```

Теперь текст будет размещаться в нижнем правом углу экрана.

Вращение и масштабирование всегда выполняются относительно точки. Это очевидно для вращения, что подтвердит каждый, кто смотрел мультфильм про Карлсона. Но и масштабирование выполняется относительно точки. Тогда как объект увеличивается или уменьшается в размерах, одна его точка остается неподвижной; эта точка задается аргументом *origin* метода *DrawString*. (И эта точка может не принадлежать масштабируемому объекту.)

Проект ScaleTextToViewport (Масштабирование текста до размеров окна просмотра) обеспечивает отображение строки текста в центре экрана и увеличение его размеров до заполнения окна просмотра. Как и другие приложения, данное включает шрифт. Рассмотрим поля:

Проект XNA: ScaleTextToViewport   Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.5f;           // циклов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Vector2 textPosition;
    Vector2 origin;
    Vector2 maxScale;
```

```
    Vector2 scale;
    float tLap;
    …
}
```

В данном приложении выполняется полный цикл увеличения текста и возвращения его к исходному размеру. В ходе этого цикла значение поля *scale* меняется между *Vector2.One* и *maxScale* (Максимальный масштаб).

Метод *LoadContent* задает значением поля *textPosition* центр экрана, значением поля *origin* – центр текста, и в качестве значения *maxScale* – максимальный коэффициент масштабирования, который обеспечит заполнение экрана текстом. Все выравнивание, вращение и масштабирование выполняются относительно и центра текста, и центра экрана.

Проект XNA: ScaleTextToViewport Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Viewport viewport = this.GraphicsDevice.Viewport;

    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    Vector2 textSize = segoe14.MeasureString(TEXT);
    textPosition = new Vector2(viewport.Width / 2, viewport.Height / 2);
    origin = new Vector2(textSize.X / 2, textSize.Y / 2);
    maxScale = new Vector2(viewport.Width / textSize.X, viewport.Height /
textSize.Y);
}
```

Как и в нескольких предыдущих приложениях, значение переменной *tLap* многократно меняется от 0 до 1 и обратно. В ходе одного цикла переменная *pLap* меняет значение от 0 до 1 и опять возвращается к 0, где 0 означает исходный размер без масштабирования, и 1 – максимальное масштабирование. Метод *Vector2.Lerp* вычисляет *scale* на основании значения *pLap*.

Проект XNA: ScaleTextToViewport Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    tLap = (SPEED * (float)gameTime.TotalGameTime.TotalSeconds) % 1;
    float pLap = (1 - (float)Math.Cos(tLap * MathHelper.TwoPi)) / 2;
    scale = Vector2.Lerp(Vector2.One, maxScale, pLap);

    base.Update(gameTime);
}
```

Метод *Draw* использует одну из развернутых версий *DrawString*, аргументы *textPosition*, *angle* и *origin* которого вычисляются в ходе выполнения *LoadContent*, а *scale* – в методе *Update*:

Проект XNA: ScaleTextToViewport Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);
```

```
    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White,
                           0, origin, scale, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

При выполнении приложения можно заметить, что масштабирование по вертикали не обеспечивает растяжение текста до самых краев экрана. Причина в том, что метод *MeasureString* возвращает вертикальный размер, исходя из максимальной высоты шрифта, куда входит пространство для подстрочных элементов, возможных диакритических знаков и небольшой отступ, конечно же.

Также должно быть очевидным, что здесь мы имеем дело с растровым шрифтом:

Механизм отображения пытается сгладить зазубрины, но это еще вопрос, что лучше, «ступеньки» или размытость. Масштабировать текст и обеспечивать гладкие векторные контуры поможет Silverlight. Или всегда можно использовать шрифт большого размера и просто работать с его уменьшенными версиями.

Два приложения, реализующие вращение текста

Завершим эту главу двумя приложениями, реализующими вращение текста.

Было бы довольно просто написать приложение, которое просто вращает текст вокруг центральной точки, но давайте попробуем сделать что-то более сложное. Давайте будем постепенно увеличивать скорость вращения и останавливать движение по касанию. После снятия касания вращение должно медленно возобновляться, опять же с постепенным ускорением. Когда количество оборотов в секунду достигает частоты обновления экрана (или какой-то целой его части), вращение текста должно замедляться, останавливаться и возобновляться в обратном направлении. Это будет выглядеть забавно.

Небольшое введение в работу с ускорением. Одним из самых часто встречающихся в повседневной жизни примеров ускорения движущихся объектов является свободное падение. В безвоздушном пространстве на поверхности Земли сила гравитации создает постоянное ускорение в 9,8 метра в секунду на секунду или, как чаще всего говорят, в секунду в квадрате:

$a = 9{,}8\ \text{м}/c^2$

Звучит странно: «метров в секунду на секунду». Но на самом деле это означает, что каждую секунду скорость увеличивается на 9,8 метра в секунду. В любой момент времени *t* скорость вычисляется по простой формуле:

$v(t) = at$

где *a* – это 9,8 метра на секунду в квадрате. Если умножить ускорение, выраженное в метрах на секунду в квадрате, на время в секундах, получим метры в секунду, что соответствует скорости. В 0 секунд скорость равна 0. В 1 секунду скорость составляет 9,8 метра в секунду. Через 2 секунды скорость уже будет 19,6 метров в секунду и т.д..

Расстояние, которое объект проходит в свободном падении, описывается уравнением:

$x(t) = \frac{at^2}{2}$

Элементарные вычисления делают семейство этих формул вполне понятным: скорость является производным от расстояния, ускорения является производным от скорости. В этом уравнении ускорение умножается на время в квадрате, таким образом, получаем метры. В конце первой секунды скорость объекта, находящегося в свободном падении, достигает 9,8 метров в секунду, но поскольку свободное падение началось с нулевой скорости, объект преодолел расстояние лишь в 4,9 метра. К концу второй секунды объект пройдет 19,6 метра.

В проекте TouchToStopRotation (Коснись, чтобы остановить вращение) скорость выражается в оборотах в секунду, и ускорение – в оборотах в секунду в квадрате. Данное приложение требует дополнительной директивы *using* для пространства имен *System.Text*.

Проект XNA: TouchToStopRevolution Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float ACCELERATION = 1;       // оборотов в секунду в квадрате
    const float MAXSPEED = 30;          // оборотов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Vector2 textPosition;
    Vector2 origin;
    Vector2 statusPosition;
    float speed;
    float angle;
    StringBuilder strBuilder = new StringBuilder();
    …

}
```

Константе MAXSPEED (Максимальная скорость) задано значение 30 оборотов в секунду, что соответствует частоте кадров. По достижении этой скорости вращающийся текст должен остановиться. Константа ACCELERATION (Ускорение) равна 1 обороту в секунду в квадрате. Это означает, что каждую секунду скорость увеличивается на 1 оборот в секунду. В конце первой секунды скорость составляет 1 оборот в секунду. В конце второй секунды скорость равна 2 оборотам в секунду. Скорость достигает значения MAXSPEED в конце 30 секунды.

Поля включают переменную *speed* и объект *StringBuilder*, который будет использоваться для отображения на экране текущей скорости в точке *statusPosition* (Местоположение отображения состояния). Большинство этих полей инициируется в методе *LoadContent*:

Проект XNA: TouchToStopRevolution Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
```

```
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Viewport viewport = this.GraphicsDevice.Viewport;
    textPosition = new Vector2(viewport.Width / 2, viewport.Height / 2);

    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    Vector2 textSize = segoe14.MeasureString(TEXT);
    origin = new Vector2(textSize.X / 2, textSize.Y / 2);
    statusPosition = new Vector2(viewport.Width - textSize.X,
                                 viewport.Height - textSize.Y);
}
```

Метод *Update* увеличивает значение переменной *speed*, исходя из ускорения, и затем увеличивает *angle* на основании нового значения *speed*.

Проект XNA: TouchToStopRevolution Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (TouchPanel.GetState().Count == 0)
    {
        speed += ACCELERATION * (float)gameTime.ElapsedGameTime.TotalSeconds;
        speed = Math.Min(MAXSPEED, speed);
        angle += MathHelper.TwoPi * speed *
(float)gameTime.ElapsedGameTime.TotalSeconds;
        angle %= MathHelper.TwoPi;
    }
    else
    {
        if (speed == 0)
            SuppressDraw();

        speed = 0;
    }
    strBuilder.Remove(0, strBuilder.Length);
    strBuilder.AppendFormat(" {0:F1} revolutions/second", speed);

    base.Update(gameTime);
}
```

Если метод *TouchPanel.GetState()* возвращает непустую коллекцию, т.е. если имеет место касание экрана, переменная *speed* возвращается к значению нуль. Более того, если при последующем вызове *Update* касание еще не завершено, вызывается метод *SuppressDraw*. Таким образом, касаясь экрана, пользователь не только останавливает вращение текста, но и обеспечивает сохранение энергии.

Также обратите внимание на использование *StringBuilder* для обновления поля состояния. Метод *Draw* аналогичен всем предыдущим приложениям, но включает два вызова *DrawString*:

Проект XNA: TouchToStopRevolution Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, strBuilder, statusPosition, Color.White);
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White,
                           angle, origin, 1, SpriteEffects.None, 0);
```

```
    spriteBatch.End();

    base.Draw(gameTime);
}
```

В заключительном приложении данной главы я вернулся к расположению по умолчанию верхнего левого угла текста. Но мне хотелось, чтобы верхний левый угол строки текста медленно передвигался по периметру экрана с внутренней стороны, и также чтобы текст оставался полностью видимым в любой момент времени. Такая постановка задачи подразумевает разворот текста на 90 градусов при прохождении каждого из углов. Данный снимок экрана иллюстрирует прохождение нижнего правого угла экрана:

Я назвал приложение TextCrawl (Ползущий текст). Поля нам уже знакомы:

Проект XNA: TextCrawl Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.1f;           // циклов в секунду
    const string TEXT = "Hello, Windows Phone 7!";

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont segoe14;
    Viewport viewport;
    Vector2 textSize;
    Vector2 textPosition;
    float tCorner;          // высота / периметр
    float tLap;
    float angle;
    …
}
```

При перемещении текста по периметру против часовой стрелки значение переменной *tLap* меняется от 0 до 1. Чтобы помочь в определении, на какой стороне находится текст в настоящий момент, я определил переменную *tCorner* (Угол). Если значение *tLap* меньше значения *tCorner*, текст располагается на левом краю экрана; если *tLap* больше *tCorner*, но меньше 0,5, текст внизу экрана, и т.д. В методе *LoadContent* нет ничего особенного:

Проект XNA: TextCrawl Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;
    tCorner = 0.5f * viewport.Height / (viewport.Width + viewport.Height);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
```

```
    textSize = segoe14.MeasureString(TEXT);
}
```

Метод *Update* просто кошмарный, я его боюсь. Его задача – вычислить *textPosition* и *angle* для последующего вызова *DrawString*.

Проект XNA: TextCrawl   Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    tLap = (tLap + SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds) % 1;

    if (tLap < tCorner)             // вниз вдоль левой стороны экрана
    {
        textPosition.X = 0;
        textPosition.Y = (tLap / tCorner) * viewport.Height;
        angle = -MathHelper.PiOver2;

        if (textPosition.Y < textSize.X)
            angle += (float)Math.Acos(textPosition.Y / textSize.X);
    }
    else if (tLap < 0.5f)           // вдоль нижнего края экрана
    {
        textPosition.X = ((tLap - tCorner) / (0.5f - tCorner)) * viewport.Width;
        textPosition.Y = viewport.Height;
        angle = MathHelper.Pi;

        if (textPosition.X < textSize.X)
            angle += (float)Math.Acos(textPosition.X / textSize.X);
    }
    else if (tLap < 0.5f + tCorner) // вверх вдоль правой стороны экрана
    {
        textPosition.X = viewport.Width;
        textPosition.Y = (1 - (tLap - 0.5f) / tCorner) * viewport.Height;
        angle = MathHelper.PiOver2;

        if (textPosition.Y + textSize.X > viewport.Height)
            angle += (float)Math.Acos((viewport.Height - textPosition.Y) /
textSize.X);
    }
    else                            // вдоль верхнего края экрана
    {
        textPosition.X = (1 - (tLap - 0.5f - tCorner) / (0.5f - tCorner)) *
viewport.Width;
        textPosition.Y = 0;
        angle = 0;

        if (textPosition.X + textSize.X > viewport.Width)
            angle += (float)Math.Acos((viewport.Width - textPosition.X) /
textSize.X);
    }

    base.Update(gameTime);
}
```

При написании этого кода, я сначала сосредоточился на обеспечении корректной работы первых трех выражений каждого блока *if* и *else*. Эти выражения просто обеспечивают перемещение верхнего левого угла строки текста против часовой стрелки по внутреннему периметру экрана. Исходное вычисление *angle* гарантирует, что верх текста вплотную прилегает к краю экрана. И только когда я добился того, чтобы все это работало, я был готов

приступить к созданию кода, который обеспечивает изменение *angle* при огибании углов. Несколько простых рисунков убедили меня, что для этой работы прекрасно подойдет арккосинус. Несмотря на всю сложность *Update*, метод *Draw* абсолютно тривиальный:

Проект XNA: TextCrawl Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, TEXT, textPosition, Color.White,
                           angle, Vector2.Zero, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

В следующей главе мы рассмотрим, как перемещать спрайты вдоль кривых.

Глава 20
Текстуры и спрайты

Я обещал, что навыки по использованию XNA для перемещения текста по экрану будут хорошим подспорьем в искусстве перемещения обычных растровых спрайтов. Это становится очевидным уже при первом ознакомлении с методами *Draw*, поддерживаемыми *SpriteBatch*. Методы *Draw* имеют практически такие же аргументы, что и *DrawString*, но работают с растровыми изображениями, а не текстом. В данной главе мы рассмотрим техники по перемещению и развороту спрайтов, уделив особое внимание перемещению вдоль кривых.

Разновидности метода *Draw*

Метод под именем *Draw* есть и у класса *Game*, и у класса *SpriteBatch*. Несмотря на идентичные наименования, эти два метода не связаны генеалогически через иерархию классов. В классе, наследуемом от *Game*, мы создаем перегрузку метода *Draw*, чтобы иметь возможность вызывать метод *Draw* класса *SpriteBatch*. Для этого последнего метода *Draw* предлагается семь разных вариантов. Начнем с самого простого:

```
Draw(Texture2D texture, Vector2 position, Color color)
```

По сути, первый аргумент, *Texture2D* – это растровое изображение. *Texture2D* потенциально несколько более сложен, чем обычное растровое изображение, потому что может включать множество текстур с варьируемым разрешением. (Они представляют одно и то же изображение, но с разным разрешением, благодаря чему изображение может отображаться с разным масштабом.) Объекты *Texture2D*, которые мы будем обсуждать, это старые добрые растровые изображения. Профессиональные разработчики игровых приложений обычно создают эти растровые изображения в специализированных инструментах, но я буду использовать обычный Paint, потому что он самый легкодоступный. После создания эти растровые изображения должны быть добавлены в содержимое проекта XNA и затем загружаться в приложение так же, как загружается шрифт.

Второй аргумент метода *Draw* показывает, где на экране будет отображаться заданное растровое изображение. По умолчанию аргумент *position* обозначает точку на экране, в которой должен располагаться верхний левый угол текстуры.

Аргумент *Color* используется несколько иначе, чем в *DrawString*, потому что сама текстура может содержать данные цвета. В документации этот аргумент называют «модуляцией цветового канала». Он выполняет роль фильтра при просмотре растрового изображения.

Концептуально каждый пиксел растрового изображения включает однобайтное значение красного, однобайтное значение зеленого и однобайтное значение синего (пока что забудем об альфа-канале). Когда *Draw* выводит на экран растровое изображение, эти значения красного, зеленого и синего цветов умножаются на однобайтные значения красного, зеленого и синего аргумента *Color* метода *Draw*, и результаты делятся на 255, чтобы вернуть их в диапазон от 0 до 255. Полученные таким образом значения и применяются для закрашивания пикселов.

Например, предположим, наша текстура включает большой объем сведений о цвете, и требуется, чтобы все эти цвета отображались. Для этого используйте в методе *Draw* значение *Color.White*.

Теперь, предположим, требуется отрисовать такую же текстуру, но сделать ее более темной (возможно, в нашем игровом мире наступает вечер). Применим в методе *Draw* немного серого. Чем темнее серый, тем темнее будет выглядеть текстура. Если использовать *Color.Black*, текстура будет отображаться как силуэт без цвета.

Предположим, мы имеем абсолютно белую текстуру, но хотим, чтобы на экране она отображалась синей. Используем для этого *Color.Blue* в методе *Draw*. Эту же абсолютно белую текстуру можно отображать любым цветом. (Именно это я и делаю в первом примере данной главы.)

Желтая текстура (сочетание красного и зеленого) при использовании в методе *Draw* значения *Color.Green* будет отображаться зеленой. Если использовать в методе *Draw* значение *Color.Red*, эта текстура будет отображаться красной. Значение *Color.Blue* сделает ее черной. Аргумент метода может только смягчить или приглушить цвет, с его помощью нельзя получить цвета, которых нет в текстуре.

Рассмотрим второй вариант метода *Draw*:

```
Draw(Texture2D texture, Rectangle destination, Color color)
```

Вместо *Vector2* для обозначения местоположения текстуры используется *Rectangle*, который сочетает в себе точку (верхний левый угол), ширину и высоту. Если ширина и высота *Rectangle* не совпадают с шириной и высотой текстуры, размеры текстуры будут приведены в соответствие размеру *Rectangle*. Исходные пропорции не принимаются во внимание.

Если требуется отображать прямоугольный фрагмент текстуры, можно использовать одну из двух несколько расширенных версий метода *Draw*:

```
Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color)
Draw(Texture2D texture, Rectangle destination, Rectangle? source, Color color)
```

Третьи аргументы – это допускающие пустое значение объекты *Rectangle*. Если в качестве этого аргумента задать *null*, результат будет аналогичен применению одной из первых двух версий *Draw*. Эти аргументы позволяют задать подмножество пикселов изображения.

В следующих двух версиях метода *Draw* имеется пять дополнительных аргументов, которые нам знакомы из методов *DrawString*:

```
Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color,
     float rotation, Vector2 origin, float scale, SpriteEffects effects, float depth)

Draw(Texture2D texture, Vector2 position, Rectangle? source, Color color,
     float rotation, Vector2 origin, Vector2 scale, SpriteEffects effects, float
depth)
```

Как и в *DrawString*, угол *rotation* задается в радианах и отсчитывается по часовой стрелке. *origin* – это точка текстуры, которая должна совпадать со значением аргумента *position*. Задание одного обеспечит *float* масштабирование текстуры одинаково во всех направлениях, *Vector2*обеспечит разное масштабирование в вертикальном и горизонтальном направлениях. Использование перечисления *SpriteEffects* позволяет переворачивать изображение в горизонтальном или вертикальном направлении, получая его зеркальное отображение. С последним аргументом мы получаем возможность переопределять настройки по умолчанию для компоновки множества текстур на экране.

Наконец, имеется немного сокращенная развернутая версия этого метода, где вторым аргументом является целевой прямоугольник:

```
spriteBatch.Draw(Texture2D texture, Rectangle destination, Rectangle? source, Color
color,
                  float rotation, Vector2 origin, SpriteEffects effects, float depth)
```

Обратите внимание, что в данном случае нет отдельного аргумента для задания масштаба, потому что масштабирование здесь обрабатывается через аргумент *destination* (место назначения).

В рамках метода *Draw* класса *Game* используется объект *SpriteBatch* следующим образом:

```
spriteBatch.Begin();
spriteBatch.Draw …
spriteBatch.End();
```

Между вызовами *Begin* и *End* может быть любое число вызовов *Draw* и *DrawString*. Вызовы *Draw* могут относиться к одной и той же структуре. Также может выполняться множество вызовов *Begin* с последующим вызовом *End* и вызовами *Draw* и *DrawString* между ними.

Еще одно приложение «Здравствуй, Мир»?

Если вы уже устали от приложений «здравствуй, Мир», у меня плохие новости. Но на этот раз мы создадим очень угловатое представление слова «HELLO», используя два разных растровых изображения – вертикальную черту и горизонтальную черту. Буква «H» будет образована двумя вертикальными и одной горизонтальной чертой. Буква «O» в конце будет выглядеть как прямоугольник.

И по касанию экрана все 15 элементов будут разлетаться в разные стороны и затем опять собираться в слово. Звучит интересно?

Если бы мы создавали приложение FlyAwayHello (Улетающее здравствуй) с нуля, первым шагом мы добавили бы содержимое в каталог Content. На этот раз это был бы не шрифт, а два растровых изображения: HorzBar.png и VertBar.png. Их можно создать прямо в Visual Studio или в Paint. По умолчанию в Paint создается абсолютно белое растровое изображение. Это просто идеальный вариант! Все что мне необходимо – это изменить размер. В Paint зайдем в меню Button (сверху слева под строкой заголовка) и выберем Properties. Зададим ширину 45 пикселов и высоту 5 пикселов. (На самом деле, конкретные значения не особенно важны, приложение обеспечивает некоторую гибкость с этой точки зрения.) Удобнее всего сохранить файл прями в папке Content проекта под именем HorzBar.png. Теперь изменим размер и зададим ширину 5 пикселов и высоту 75 пикселов. Сохраним это изображение в файле под именем VertBar.png.

Несмотря на то что файлы располагаются в соответствующей папке, проект XNA не знает об их существовании. В Visual Studio щелкнем правой кнопкой мыши папку Content и в выпадающем меню выберем Add Existing Item (Добавить существующий элемент). Можно выбрать оба файла PNG и добавить их в проект.

Я собираюсь использовать небольшой класс *SpriteInfo* (Сведения о спрайте) для отслеживания 15 текстур, необходимых для формирования текста. Если создаете проект с нуля, щелкните правой кнопкой мыши имя проекта, в появившемся меню выберите Add и затем New Item (или выберите Add New Item в главном меню Project). В диалоговом окне выберите Class и задайте для него имя SpriteInfo.cs.

Проект XNA: FlyAwayHello Файл: SpriteInfo.cs (полностью)

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace FlyAwayHello
{
    public class SpriteInfo
    {
```

```
        public static float InterpolationFactor { set; get; }

        public Texture2D Texture2D { protected set; get; }
        public Vector2 BasePosition { protected set; get; }
        public Vector2 PositionOffset { set; get; }
        public float MaximumRotation { set; get; }

        public SpriteInfo(Texture2D texture2D, int x, int y)
        {
            Texture2D = texture2D;
            BasePosition = new Vector2(x, y);
        }

        public Vector2 Position
        {
            get
            {
                return BasePosition + InterpolationFactor * PositionOffset;
            }
        }

        public float Rotation
        {
            get
            {
                return InterpolationFactor * MaximumRotation;
            }
        }
    }
}
```

Конструктор должен сохранять *Texture2D* вместе с данными позиционирования, которые используются для исходного размещения каждого спрайта и формирования слова «HELLO». Затем в анимации «разлета» приложение задает свойства *PositionOffset* (Смещение положения) и *MaximumRotation* (Максимальный разворот при вращении). Свойства *Position* и *Rotation* осуществляют вычисления на основании значения статического свойства *InterpolationFactor* (Коэффициент интерполяции), значения которого лежат в диапазоне от 0 до 1.

Рассмотрим поля класса *Game1*:

Проект XNA: FlyAwayHello Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    static readonly TimeSpan ANIMATION_DURATION = TimeSpan.FromSeconds(5);
    const int CHAR_SPACING = 5;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Viewport viewport;
    List<SpriteInfo> spriteInfos = new List<SpriteInfo>();
    Random rand = new Random();
    bool isAnimationGoing;
    TimeSpan animationStartTime;
    …
}
```

Данное приложение запускает анимацию только по касанию экрана пользователем, поэтому расчет времени в данном случае выполняется несколько иначе, чем в предыдущих приложениях, как будет видно в методе *Update*.

Метод *LoadContent* загружает два объекта *Texture2D* с помощью того же универсального метода *Load*, который использовался в предыдущих приложениях для загрузки *SpriteFont*. Теперь мы располагаем достаточным объемом сведений для создания и инициализации всех объектов *SpriteInfo*:

Проект XNA: FlyAwayHello Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;

    Texture2D horzBar = Content.Load<Texture2D>("HorzBar");
    Texture2D vertBar = Content.Load<Texture2D>("VertBar");

    int x = (viewport.Width - 5 * horzBar.Width - 4 * CHAR_SPACING) / 2;
    int y = (viewport.Height - vertBar.Height) / 2;
    int xRight = horzBar.Width - vertBar.Width;
    int yMiddle = (vertBar.Height - horzBar.Height) / 2;
    int yBottom = vertBar.Height - horzBar.Height;

    // H
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(vertBar, x + xRight, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yMiddle));

    // E
    x += horzBar.Width + CHAR_SPACING;
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yMiddle));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));

    // LL
    for (int i = 0; i < 2; i++)
    {
        x += horzBar.Width + CHAR_SPACING;
        spriteInfos.Add(new SpriteInfo(vertBar, x, y));
        spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));
    }

    // O
    x += horzBar.Width + CHAR_SPACING;
    spriteInfos.Add(new SpriteInfo(vertBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y));
    spriteInfos.Add(new SpriteInfo(horzBar, x, y + yBottom));
    spriteInfos.Add(new SpriteInfo(vertBar, x + xRight, y));
}
```

Метод *Update* отвечает за продолжение выполнения анимации. Если поле *isAnimationGoing* (Анимация продолжается) имеет значение *false*, он проводит проверку на наличие нового касания.

Проект XNA: FlyAwayHello Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (isAnimationGoing)
    {
        TimeSpan animationTime = gameTime.TotalGameTime - animationStartTime;
```

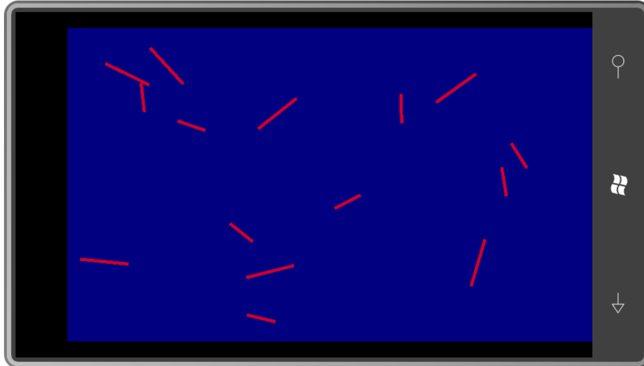
```
        double fractionTime = (double)animationTime.Ticks /
                                                    ANIMATION_DURATION.Ticks;

        if (fractionTime >= 1)
        {
            isAnimationGoing = false;
            fractionTime = 1;
        }

        SpriteInfo.InterpolationFactor = (float)Math.Sin(Math.PI * fractionTime);
    }
    else
    {
        TouchCollection touchCollection = TouchPanel.GetState();
        bool atLeastOneTouchPointPressed = false;

        foreach (TouchLocation touchLocation in touchCollection)
            atLeastOneTouchPointPressed |=
                touchLocation.State == TouchLocationState.Pressed;

        if (atLeastOneTouchPointPressed)
        {
            foreach (SpriteInfo spriteInfo in spriteInfos)
            {
                float r1 = (float)rand.NextDouble() - 0.5f;
                float r2 = (float)rand.NextDouble() - 0.5f;
                float r3 = (float)rand.NextDouble();

                spriteInfo.PositionOffset = new Vector2(r1 * viewport.Width,
                                                        r2 * viewport.Height);
                spriteInfo.MaximumRotation = 2 * (float)Math.PI * r3;
            }
            animationStartTime = gameTime.TotalGameTime;
            isAnimationGoing = true;
        }
    }
    base.Update(gameTime);
}
```

Когда анимация запускается, ее свойству *animationStartTime* (Время начала анимации) задается значение свойства *TotalGameTime* объекта *GameTime*. При последующих вызовах метод *Update* сравнивает это значение с новым *TotalGameTime* и вычисляет коэффициент интерполяции. Свойство *InterpolationFactor* класса *SpriteInfo* является статическим, поэтому задается только раз и распространяется на все экземпляры *SpriteInfo*. Метод *Draw* перебирает все объекты *SpriteInfo*, выполняя доступ к их свойствам *Position* и *Rotation*:

Проект XNA: FlyAwayHello Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);
    spriteBatch.Begin();

    foreach (SpriteInfo spriteInfo in spriteInfos)
    {
        spriteBatch.Draw(spriteInfo.Texture2D, spriteInfo.Position, null,
            Color.Lerp(Color.Blue, Color.Red, SpriteInfo.InterpolationFactor),
            spriteInfo.Rotation, Vector2.Zero, 1, SpriteEffects.None, 0);
    }

    spriteBatch.End();
    base.Draw(gameTime);
}
```

Вызов *Draw* также использует *SpriteInfo.InterpolationFactor* для интерполяции между синим и красным при определении цвета элементов слова. Обратите внимание, что структура *Color* также имеет метод *Lerp*. Слово отображается синим, но когда разлетается на отдельные элементы, они меняют цвет на красный.



Этот вызов *Draw*, на самом деле, мог бы быть частью *SpriteInfo*. *SpriteInfo* мог бы определять собственный метод *Draw* с аргументом типа *SpriteBatch* и затем передавать свои свойства *Texture2D*, *Position* и *Rotation* в метод *Draw* этого *SpriteBatch*:

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(Texture2D, Position, null,
                     Color.Lerp(Color.Blue, Color.Red, InterpolationFactor),
                     Rotation, Vector2.Zero, 1, SpriteEffects.None, 0);
}
```

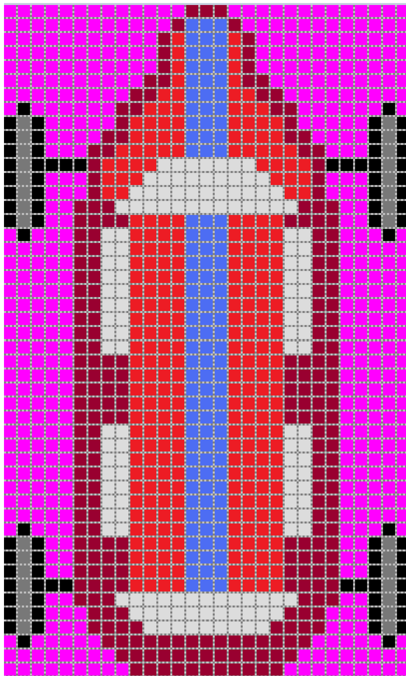
Тогда цикл в перегруженном *Draw* класса *Game1* выглядел бы следующим образом:

```
foreach (SpriteInfo spriteInfo in spriteInfos)
{
    spriteInfo.Draw(spriteBatch);
}
```

Это общепринятая методика, позволяющая сократить количество открытых свойств в *SpriteInfo*.

Перемещение по заданной траектории

Далее в данной главе я хочу сосредоточиться на методиках перемещения спрайтов по некоторой траектории. Чтобы добавить реализма, я поручил моей жене Дидре нарисовать в Paint небольшой гоночный автомобиль:



Размеры автомобиля составляют 48 пикселов в длину и 29 пикселов в ширину. Обратите внимание на пурпурный фон. Если требуется сделать часть изображения прозрачным в XNA, можно использовать 32-разрядный растровый формат, который поддерживает прозрачность, такой как PNG, например. Каждый пиксел в этом формате имеет 8-битовый красный, зеленый и синий компоненты, но также и 8-битовый альфа-канал для определения прозрачности. (Этот формат мы рассмотрим в следующей главе.) Приложение Paint в Windows не поддерживает прозрачности для растровых изображений, увы, но вместо этого может использоваться пурпурный цвет. В Paint для получения этого цвета задайте красному и синему каналам значение 255 и зеленому – 0.

Во все проекты данной главы файл этого изображения, car.png, включен как часть содержимого проекта. Первый проект, CarOnRectangularCourse (Объезд по прямоугольной траектории), демонстрирует достаточно неуклюжий подход к реализации перемещения автомобиля по периметру экрана. Он включает следующие поля:

Проект XNA: CarOnRectangularCourse Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 100;            // пикселов в секунду
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D car;
    Vector2 carCenter;
    Vector2[] turnPoints = new Vector2[4];
    int sideIndex = 0;
    Vector2 position;
    float rotation;
    …
}
```

Массив *turnPoints* (Точки разворота) включает четыре точки в углах экрана, в которых автомобиль делает резкий разворот. Вычисление координат этих точек – является одной из основных задач метода *LoadContent*, который также загружает *Texture2D* и инициализирует остальные поля:

Проект XNA: CarOnRectangularCourse Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    float margin = car.Width;
    Viewport viewport = this.GraphicsDevice.Viewport;
    turnPoints[0] = new Vector2(margin, margin);
    turnPoints[1] = new Vector2(viewport.Width - margin, margin);
    turnPoints[2] = new Vector2(viewport.Width - margin, viewport.Height - margin);
    turnPoints[3] = new Vector2(margin, viewport.Height - margin);
    position = turnPoints[0];
    rotation = MathHelper.PiOver2;
}
```

Я использую поле *carCenter* в качестве аргумента *origin* для метода *Draw*, т.е. эта та точка автомобиля, которая должна совпадать с точкой траектории, определенной одним из четырех членов массива *turnPoints*. Благодаря значению *margin* эта траектория располагается на расстоянии одной ширины автомобиля от края экрана, таким образом, автомобиль отстоит от края экрана на половину своей ширины.

Я назвал это приложение «неуклюжим», и метод *Update* является абсолютным подтверждением этому:

Проект XNA: CarOnRectangularCourse Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float pixels = SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;

    switch (sideIndex)
    {
         case 0:         // верх
            position.X += pixels;

            if (position.X > turnPoints[1].X)
            {
                position.X = turnPoints[1].X;
                position.Y = turnPoints[1].Y + (position.X - turnPoints[1].X);
                rotation = MathHelper.Pi;
                sideIndex = 1;
            }
            break;

        case 1:         // правый край
            position.Y += pixels;

            if (position.Y > turnPoints[2].Y)
            {
                position.Y = turnPoints[2].Y;
                position.X = turnPoints[2].X - (position.Y - turnPoints[2].Y);
                rotation = -MathHelper.PiOver2;
                sideIndex = 2;
            }
            break;

        case 2:         // низ
            position.X -= pixels;
```

```
            if (position.X < turnPoints[3].X)
            {
                position.X = turnPoints[3].X;
                position.Y = turnPoints[3].Y + (position.X - turnPoints[3].X);
                rotation = 0;
                sideIndex = 3;
            }
            break;

        case 3:          // левый край
            position.Y -= pixels;

            if (position.Y < turnPoints[0].Y)
            {
                position.Y = turnPoints[0].Y;
                position.X = turnPoints[0].X - (position.Y - turnPoints[0].Y);
                rotation = MathHelper.PiOver2;
                sideIndex = 0;
            }
            break;
    }
    base.Update(gameTime);
}
```

Этот тот тип кода, который просто кричит: «Есть лучший способ сделать это!» Он не элегантен и не универсален. Но прежде чем перейти к более гибкому подходу, рассмотрим абсолютно предсказуемый метод *Draw*, который включает обновленные значения *position* и *rotation*, вычисленные в ходе выполнения *Update*:

Проект XNA: CarOnRectangularCourse Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Перемещение вдоль полилинии

Код из предыдущего приложения может использоваться для любой прямоугольной траектории, координаты углов которой хранятся в массиве *turnPoints*. Но он не годится для произвольной коллекции из четырех точек или коллекции, включающей большее количество точек. В компьютерной графике коллекцию точек, описывающую последовательности прямых линий, часто называют *полилинией*. Было бы неплохо написать некоторый код, который описывал бы перемещение автомобиля по произвольной полилинии.

Следующий проект под именем CarOnPolylineCourse (Объезд по сложной траектории) включает класс *PolylineInterpolator* (Средство интерполяции полилинии), который делает это возможным. Но прежде чем перейти к классу *PolylineInterpolator*, давайте сначала рассмотрим класс *Game1*. Он включает такие поля:

Проект XNA: CarOnPolylineCourse Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.25f;          // циклов в секунду
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D car;
    Vector2 carCenter;
    PolylineInterpolator polylineInterpolator = new PolylineInterpolator();
    Vector2 position;
    float rotation;
    …
}
```

Как видим, скорость здесь представлена в циклах, и создается экземпляр этого загадочного класса *PolylineInterpolator*. Метод *LoadContent* практически аналогичен тому, каким он был в предыдущем проекте, за исключением того, что в данном случае точки добавляются не в массив *turnPoints*, а в свойство *Vertices* (Вершины) класса *PolylineInterpolator*:

Проект XNA: CarOnPolylineCourse   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("Car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    float margin = car.Width;
    Viewport viewport = this.GraphicsDevice.Viewport;

    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(viewport.Width - car.Width, car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, viewport.Height - car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(viewport.Width - car.Width, viewport.Height -
car.Width));
    polylineInterpolator.Vertices.Add(
                new Vector2(car.Width, car.Width));
}
```

Обратите внимание, этот метод в конце опять добавляет начальную точку, и задаваемые точки описывают не совсем такую же траекторию, что в предыдущем проекте. Ранее автомобиль перемещался из верхнего левого угла в верхний правый, затем вниз в нижний правый угол, по нижнему краю в нижний левый угол и вверх в верхний левый угол. Сейчас же порядок перемещения таков, что автомобиль из верхнего левого угла движется в верхний правый угол, затем по диагонали вниз в нижний левый угол, вдоль по нижнему краю в нижний правый угол и затем по диагонали возвращается в исходную точку. Именно такой маневренности не хватало в предыдущем приложении.

Как и в приложениях предыдущей главы, в которых использовались параметрические уравнения, метод *Update* прост до слез:

Проект XNA: CarOnPolylineCourse   Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
```

```
    float t = (SPEED * (float)gameTime.TotalGameTime.TotalSeconds) % 1;
    float angle;
    position = polylineInterpolator.GetValue(t, false, out angle);
    rotation = angle + MathHelper.PiOver2;

    base.Update(gameTime);
}
```

Как обычно, значения *t* лежат в диапазоне от 0 до 1, где 0 указывает на начало траектории в верхнем левом углу экрана. По мере движения автомобиля по траектории и приближения к исходной позиции снова, значение *t* стремится к 1. Это *t* передается непосредственно в метод *GetValue* класса *PolylineInterpolator*, который возвращает значение *Vector2* где-то на полилинии.

Как дополнительный приз последний аргумент метода *GetValue* позволяет получить значение *angle*, которое определяет касательную к полилинии в данной точке. Этот угол отсчитывается по часовой стрелке от положительного направления оси X. Например, когда автомобиль движется из верхнего левого в верхний правый угол, *angle* равен 0. Когда автомобиль находится на пути из верхнего правого в нижний левый угол, значение угла лежит в диапазоне от π/2 и π в зависимости от соотношения размеров экрана. Автомобиль на рисунке расположен вертикально вверх, поэтому в ходе перемещения его придется поворачивать на дополнительные π/2 радиан.

Метод *Draw* никак не изменился:

Проект XNA: CarOnPolylineCourse Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

На данном снимке экрана автомобиль направляется в нижний левый угол:

Для демонстрационных целей в классе *PolylineInterpolator* я пренебрег эффективностью во имя простоты. Привожу данный класс полностью:

Проект XNA: CarOnPolylineCourse Файл: PolylineInterpolator.cs (полностью)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace CarOnPolylineCourse
{
    public class PolylineInterpolator
    {
        public PolylineInterpolator()
        {
            Vertices = new List<Vector2>();
        }

        public List<Vector2> Vertices { protected set; get; }

        public float TotalLength()
        {
            float totalLength = 0;

            // Обратите внимание, объезд траектории начинается с индекса 1
            for (int i = 1; i < Vertices.Count; i++)
            {
                totalLength += (Vertices[i] - Vertices[i - 1]).Length();
            }
            return totalLength;
        }

        public Vector2 GetValue(float t, bool smooth, out float angle)
        {
            if (Vertices.Count == 0)
            {
                return GetValue(Vector2.Zero, Vector2.Zero, t, smooth, out angle);
            }

            else if (Vertices.Count == 1)
            {
                return GetValue(Vertices[0], Vertices[0], t, smooth, out angle);
            }

            if (Vertices.Count == 2)
            {
                return GetValue(Vertices[0], Vertices[1], t, smooth, out angle);
            }

            // Вычисляем общую протяженность маршрута
            float totalLength = TotalLength();
            float accumLength = 0;

            // Обратите внимание, объезд траектории начинается с индекса 1
            for (int i = 1; i < Vertices.Count; i++)
            {
                float prevLength = accumLength;
                accumLength += (Vertices[i] - Vertices[i - 1]).Length();

                if (t >= prevLength / totalLength && t <= accumLength / totalLength)
                {
                    float tPrev = prevLength / totalLength;
                    float tThis = accumLength / totalLength;
                    float tNew = (t - tPrev) / (tThis - tPrev);

                    return GetValue(Vertices[i - 1], Vertices[i],
                                    tNew, smooth, out angle);
                }
            }

            return GetValue(Vector2.Zero, Vector2.Zero, t, smooth, out angle);
        }

        Vector2 GetValue(Vector2 vertex1, Vector2 vertex2, float t,
```

```
                            bool smooth, out float angle)
        {
            angle = (float)Math.Atan2(vertex2.Y - vertex1.Y, vertex2.X - vertex1.X);

            return smooth ? Vector2.SmoothStep(vertex1, vertex2, t) :
                            Vector2.Lerp(vertex1, vertex2, t);
        }
    }
}
```

Всего одно свойство *Vertices* позволяет определить коллекцию объектов *Vector2*, которая описывает полилинию. Если требуется, чтобы полилиния заканчивалась в той же точке, в которой начинается, необходимо явно продублировать эту точку в коллекции. Всю работу выполняет метод *GetValue*. Сначала этот метод определяет общую длину полилинии. Затем проходит по всем вершинам и суммирует длины отрезков между ними, выявляя пару точек, для которых эта суммарная длина выходит за рамки диапазона допустимых значений *t*. Тогда эти точки передаются в закрытый метод *GetValue* для выполнения линейной интерполяции с помощью *Vector2.Lerp* и для вычисления угла между двумя касательными, используя второго лучшего друга разработчика графических приложений метод *Math.Atan2*.

Но постойте, в методе *GetValue* есть еще аргумент Boolean, который заставляет этот метод использовать *Vector2.SmoothStep*, а не *Vector2.Lerp*. Чтобы попробовать этот альтернативный вариант, замените данный вызов метода *Update* класса *Game1*:

```
position = polylineInterpolator.GetValue(t, false, out angle);
```

следующим:

```
position = polylineInterpolator.GetValue(t, true, out angle);
```

Более «сглаженную» интерполяцию обеспечит кубическое уравнение. Также оно обусловит замедление автомобиля при приближении к вершинам и ускорение после их прохождения. Такое изменение скорости весьма приятно, но движение по-прежнему будет неровным и далеким от реалистичного.

Мне не нравится неэффективность класса *PolylineInterpolator*. *GetValue* приходится несколько раз вызывать метод *Length* класса *Vector2*, а это, безусловно, включает вычисление квадратного корня. Было бы хорошо, если бы класс сохранял общую длину и суммарную длину для каждой вершины, обеспечивая возможность просто повторно использовать эти данные при последующих вызовах *GetValue*. Как обсуждалось, класс не может сделать этого, потому что не располагает сведениями о том, когда значения *Vector2* добавляются или удаляются из коллекции *Vertices*. Одно из решений – сделать эту коллекцию закрытой и разрешить передавать в конструктор класса только коллекцию точек. Другой подход – заменить класс *List* классом *ObservableCollection*, что обеспечит уведомление о добавлении или удалении объектов.

Эллиптическая траектория

Самым далеким от реалистичного движения в предыдущем приложении была реализация поворотов. В реальности, чтобы повернуть, автомобиль замедляется, но фактически он перемещается по искривленной траектории, изменяя направление движения. Чтобы добавить реализма в предыдущее приложение, заменим углы кривыми. Эти кривые могут быть аппроксимированы с помощью полилиний, но увеличивающееся число полилиний потребует реструктуризации класса *PolylineInterpolator* для обеспечения лучшей производительности.

Вместо этого я применю другой подход и буду проводить автомобиль по традиционной *овальной* траектории или, выражаясь более математическим языком, по *эллиптической* траектории.

Немного теории. Любая точка окружности с центром в точке (0, 0) и радиусом *R* может быть описана уравнением

$$x^2 + y^2 = R^2$$

где (*x*, *y*) – координаты этой точки.

Эллипс имеет два радиуса. Если они параллельны горизонтальной и вертикальной осям, их обозначают R_x и R_y, и уравнение эллипса выглядит следующим образом:

$$\left(\frac{x}{R_x}\right)^2 + \left(\frac{y}{R_y}\right)^2 = 1$$

Для наших целей удобнее представить эллипс в параметрической форме. В этих двух уравнениях *x* и *y* являются функциями угла *α*, допустимые значения которого лежат в диапазоне от 0 до 2π:

$$\begin{cases} x = R_x \cos \propto \\ y = R_y \sin \propto \end{cases}$$

Если центром эллипса является точка (*Cx*, *Cy*), уравнения принимают такой вид:

$$\begin{cases} x = C_x + \mathrm{R}_x \cos \propto \\ y = C_y + \mathrm{R}_y \sin \propto \end{cases}$$

Введем переменную *t*, допустимые значения которой лежат в диапазоне от 0 до 1, и уравнения тогда будут выглядеть следующим образом:

$$\begin{cases} x(t) = C_x + R_x \cos(2\pi t) \\ y(t) = C_y + R_y \sin(2\pi t) \end{cases}$$

И это будет идеальным решением для наших целей. Пока *t* изменяется от 0 до 1, машина проходит один полный круг. Но как обеспечивать разворот автомобиля, чтобы создать эффект его перемещения по касательной к этому эллипсу? В этом случае на помощь приходит дифференциальное исчисление. Сначала найдем производные от наших параметрических уравнений:

$$\begin{cases} x'(t) = -R_x \sin(2\pi t) \\ y'(t) = R_y \cos(2\pi t) \end{cases}$$

С физической точки зрения эти уравнения представляют мгновенное изменение направления движения относительно осей X и Y, соответственно. Чтобы перейти к углу между двумя касательными, просто применим такой незаменимый *Math.Atan2*.

И теперь мы готовы к написанию кода. Задаем следующие поля:

Проект XNA: CarOnOvalCourse Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.25f;           // циклов в секунду
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D car;
    Vector2 carCenter;
    Point ellipseCenter;
```

```
    float ellipseRadiusX, ellipseRadiusY;
    Vector2 position;
    float rotation;
    …
}
```

Среди этих полей имеется три элемента, которые используются в параметрических уравнениях эллипса: центр и два радиуса. Они определяются во время выполнения метода *LoadContent* на основании размеров доступной области экрана:

Проект XNA: CarOnOvalCourse Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    car = this.Content.Load<Texture2D>("car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
    Viewport viewport = this.GraphicsDevice.Viewport;
    ellipseCenter = viewport.Bounds.Center;
    ellipseRadiusX = viewport.Width / 2 - car.Width;
    ellipseRadiusY = viewport.Height / 2 - car.Width;
}
```

Обратите внимание, в приведенном ниже методе *Update* вычисляются значения двух углов. Первый под называнием *ellipseAngle* (Угол эллипса) зависит от *t* и определяет точку положения автомобиля на эллипсе. Этот угол передается в параметрические уравнения эллипса для получения местоположения автомобиля в виде координат *x* и *y*:

Проект XNA: CarOnOvalCourse Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float t = (SPEED * (float)gameTime.TotalGameTime.TotalSeconds) % 1;
    float ellipseAngle = MathHelper.TwoPi * t;
    float x = ellipseCenter.X + ellipseRadiusX * (float)Math.Cos(ellipseAngle);
    float y = ellipseCenter.Y + ellipseRadiusY * (float)Math.Sin(ellipseAngle);
    position = new Vector2(x, y);

    float dxdt = -ellipseRadiusX * (float)Math.Sin(ellipseAngle);
    float dydt = ellipseRadiusY * (float)Math.Cos(ellipseAngle);
    rotation = MathHelper.PiOver2 + (float)Math.Atan2(dydt, dxdt);

    base.Update(gameTime);
}
```

Второй угол, вычисляемый в *Update*, называется *rotation*. Это угол, определяющий ориентацию автомобиля. Переменные *dxdt* и *dydt* являются производными параметрических уравнений, рассмотренных нами ранее. Метод *Math.Atan2* определяет угол вращения относительно положительного направления оси X, и, учитывая исходную ориентацию автомобиля, к нему необходимо добавить еще 90 градусов.

К этому моменту метод *Draw* мы уже знаем наизусть:

Проект XNA: CarOnOvalCourse Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

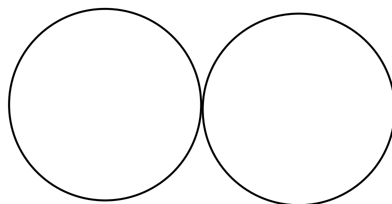
Обобщенное решение для перемещения по кривой

Для описания движения по кривым траекториям использовать параметрические уравнения не очень удобно, поэтому сам XNA предлагает обобщенное решение с использованием классов *Curve* (Кривая) и *CurveKey* (Ключ кривой), описанных в пространстве имен *Microsoft.Xna.Framework*.

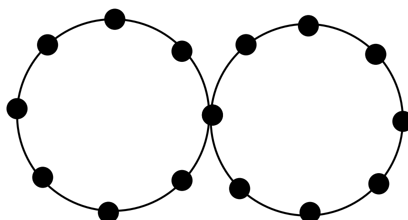
Класс *Curve* включает свойство *Keys* (Ключи) типа *CurveKeyCollection* (Коллекция ключей кривой). Это коллекция объектов *CurveKey*. Каждый объект *CurveKey* позволяет задавать числовую пару (*Position*, *Value*). И свойство *Position*, и свойство *Value* типа *float*. Затем местоположение передается в свойство *Curve* метода *Evaluate* (Вычислить), и он возвращает интерполированное значение.

Но здесь все довольно запутанно, потом что (как указывается в документации) свойство *Position* класса *CurveKey* – это практически всегда *время*, а свойство *Value* очень часто обозначает *местоположение* или, чтобы быть более точным, одна из *координат*. Чтобы использовать *Curve* для интерполяции между точками в двухмерном пространстве, потребуется два экземпляра *Curve*: один для координаты X, и второй для Y. Эти экземпляры *Curve* во многом описываются параметрическими уравнениями.

Предположим, требуется провести автомобиль по траектории, напоминающей знак бесконечности. Знак бесконечности будет реализован как две соприкасающихся окружности. (Методика, которую я собираюсь представить, позволит впоследствии разъединить эти окружности, если потребуется.)



На этих двух окружностях отрисуем точки через каждые 45 градусов:



Если радиус каждой окружности составляет 1 единицу, вся фигура занимает 4 единицы в ширину и 2 единицы в высоту. Координаты X этих точек (двигаясь слева направо) соответствуют значениям 0, 0.293, 1, 0.707, 2, 2.293, 3, 3.707 и 4; и координаты Y (двигаясь сверху вниз) соответствуют значениям 0, 0.293, 1, 1.707 и 2. Значение 0,707 – это просто синус и косинус угла 45 градусов, и 0,293 – значение, получаемое, если вычесть предыдущее значение из единицы.

Начнем с самой левой точки и будем двигаться по часовой стрелке вдоль первой окружности. В центре фигуры перейдем к движению против часовой стрелки по второй окружности, чтобы образовать траекторию в форме знака бесконечности, и завершим движение в точке, с которой начали. Значения X меняются следующим образом:

0, 0.293, 1, 1.707, 2, 2.293, 3, 3.707, 4, 3.707, 3, 2.293, 2, 1.707, 1, 0.293, 0

Если при обходе знака бесконечности используется параметр *t*, меняющийся в диапазоне от 0 до 1, первое значение будет соответствовать *t* = 0, и последнее (которое аналогично) соответствует *t* = 1. Для каждого последующего значения *t* увеличивается на 1/16 или 0,0625. Получаем ряд значений Y:

1, 0.293, 0, 0.293, 1, 1.707, 2, 1.707, 1, 0.293, 0, 0.293, 1, 1.707, 2, 1.707, 1

Теперь мы готовы к написанию кода. Определим поля проекта CarOnInfinityCourse (Автомобиль на бесконечной траектории):

Проект XNA: CarOnInfinityCourse Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float SPEED = 0.1f;           // циклов в секунду
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Viewport viewport;
    Texture2D car;
    Vector2 carCenter;
    Curve xCurve = new Curve();
    Curve yCurve = new Curve();
    Vector2 position;
    float rotation;
    …
}
```

Обратим внимание на два объекта *Curve*, один для описания координат X, и другой – для координат Y. Поскольку при инициализации этих объектов используются координаты, описанные выше, и они не требуют организации доступа к каким-либо ресурсам или содержимому приложения, я решил использовать для этой работы перегруженный метод *Initialize*.

Проект XNA: CarOnInfinityCourse Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    float[] xValues = { 0, 0.293f, 1, 1.707f, 2, 2.293f, 3, 3.707f,
                        4, 3.707f, 3, 2.293f, 2, 1.707f, 1, 0.293f };
    float[] yValues = { 1, 0.293f, 0, 0.293f, 1, 1.707f, 2, 1.707f,
                        1, 0.293f, 0, 0.293f, 1, 1.707f, 2, 1.707f };

    for (int i = -1; i < 18; i++)
    {
        int index = (i + 16) % 16;
```

```
        float t = 0.0625f * i;
        xCurve.Keys.Add(new CurveKey(t, xValues[index]));
        yCurve.Keys.Add(new CurveKey(t, yValues[index]));
    }
    xCurve.ComputeTangents(CurveTangent.Smooth);
    yCurve.ComputeTangents(CurveTangent.Smooth);
    base.Initialize();
}
```

Массивы *xValues* и *yValues* содержат только 16 значений; они не включают последнюю точку, которая дублирует первую. Может показаться довольно странным, но цикл *for* начинается с индекса –1 и до 17, но модуль 16 гарантирует, что массивы индексируются от 0 до 15. В итоге коллекции *Keys* классов *xCurve* и *yCurve* получают координаты, ассоциированные со значениями *t* –0.0625, 0, 0.0625, 0.0125, ..., 0.875, 0.9375, 1 и 1.0625. Очевидно, что здесь на две точки больше, чем требуется для правильного выполнения поставленной задачи.

Эти дополнительные точки используются для вызовов метода *ComputeTangents* (Вычислить тангенс) после цикла *for*. Класс *Curve* выполняет интерполяцию кубическим сплайном Эрмита, которую также называют *cspline*. Возьмем две точки: *pt1* и *pt2*. cspline выполняет интерполяцию между этими двумя точками на основании не только *pt1* и *pt2*, но также используя касательные к кривой в этих точках *pt1* и *pt2*. Эти касательные для объекта *Curve* можно задать как часть объектов *CurveKeys*, или их можно вычислять в объекте *Curve*, исходя из соприкасающихся точек. Такой подход я реализовал посредством двух вызовов *ComputeTangents*. Благодаря аргументу *CurveTangent.Smooth* метод *ComputeTangents* использует не только две соприкасающиеся точки, но и точки на противоположной стороне. На самом деле, это простое взвешенное среднее, но это вариант лучше, чем остальные имеющиеся альтернативы.

Классы *Curve* и *CurveKey* предлагают и другие варианты, но предпринятый мною подход обеспечивает наилучшие результаты с минимальными усилиями. Разве не в этом суть программирования?

Метод *LoadContent* должен загрузить изображение автомобиля и получить координаты его центра:

Проект XNA: CarOnInfinityCourse Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    viewport = this.GraphicsDevice.Viewport;
    car = this.Content.Load<Texture2D>("Car");
    carCenter = new Vector2(car.Width / 2, car.Height / 2);
}
```

Теперь займемся *Update*. Этот метод вычисляет *t* на основании *TotalGameTime*. Класс *Curve* определяет метод *Evaluate*, который может принимать это значение *t* напрямую; так приложение получает интерполированные координаты X и Y. Но все данные двух объектов *Curve* определяются, исходя из максимального значения координаты X = 4 и Y = 2. Поэтому *Update* вызывает небольшой метод *GetValue*, который масштабирует значения соответственно размеру экрана.

Проект XNA: CarOnInfinityCourse Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
```

```
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float t = (SPEED * (float)gameTime.TotalGameTime.TotalSeconds) % 1;
    float x = GetValue(t, true);
    float y = GetValue(t, false);
    position = new Vector2(x, y);

    rotation = MathHelper.PiOver2 + (float)
        Math.Atan2(GetValue(t + 0.001f, false) - GetValue(t - 0.001f, false),
                   GetValue(t + 0.001f, true) - GetValue(t - 0.001f, true));

    base.Update(gameTime);
}

float GetValue(float t, bool isX)
{
    if (isX)
        return xCurve.Evaluate(t) * (viewport.Width - 2 * car.Width) / 4 +
car.Width;

    return yCurve.Evaluate(t) * (viewport.Height - 2 * car.Width) / 2 + car.Width;
}
```

После вычисления поля *position* мы приходим к небольшой проблеме, потому что классу *Curve* не хватает очень важного метода: метода, который бы обеспечивал вычисление касательной сплайна. Класс *Curve* использует касательные для *вычисления* сплайна, но после того как сплайн рассчитан, класс не обеспечивает доступа к касательным самого сплайна!

Это назначение других четырех вызовов *GetValue*. Для приближенного вычисления производной к *t* прибавляются и вычитаются небольшие значения. Таким образом *Math.Atan2* вычисляет угол *rotation*.

Как обычно, стандартный *Draw*:

Проект XNA: CarOnInfinityCourse   Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Blue);

    spriteBatch.Begin();
    spriteBatch.Draw(car, position, null, Color.White, rotation,
                     carCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Если предполагается вычислять касательные, используемые для обсчета сплайна, с помощью класса *Curve* (как я сделал в данном приложении), классу необходимо предоставить достаточное количество точек. То есть не только сверх диапазона точек, которые подлежат интерполяции, но достаточно для того, чтобы обеспечить более или менее точные касательные. Изначально я попробовал определить траекторию в виде знака бесконечности, задавая точки через каждые 90 градусов, и это совершенно не сработало.

Глава 21
Динамические текстуры

Самый обычный способ получения объекта *Texture2D* для приложения на XNA – загрузить его как содержимое. В главе 4 мы также видели, как приложение создает *Texture2D* из объекта *Stream* с помощью статического метода *Texture2D.FromSteam*. Этот объект может ссылаться на растровое изображение, загруженное из Интернета, или изображение из библиотеки фотографий пользователя, или фотографию, только что снятую камерой телефона.

Также объект *Texture2D* может создаваться полностью в коде с использованием такого конструктора:

```
Texture2D texture = new Texture2D(this.GraphicsDevice, width, height);
```

Аргументы *width* и *height* – это величины типа *integer*, обозначающие желаемый размер *Texture2D* в пикселах. После того как *Texture2D* создан, этот размер не может быть изменен. Общее число пикселов в растровом изображении легко подсчитать, умножив *width* на *height*. Данный конструктор создает растровую матрицу, заполненную нулями. И теперь главный вопрос: как заполнить эту растровую матрицу реальным содержимым?

Есть два способа:

- Рисовать в растровой матрице точно так же, как мы рисуем на экране устройства.
- Алгоритмически обрабатывать значения пикселов, составляющих растровую матрицу.

Эти две техники могут комбинироваться друг с другом или использоваться отдельно. С их помощью можно изменять уже существующее изображение.

Целевой объект прорисовки

Строго говоря, первая из упомянутых техник *не может* использоваться с объектом *Texture2D*. Необходимо создать экземпляр класса, наследуемого от *Texture2D*, под именем *RenderTarget2D*:

```
RenderTarget2D renderTarget = new RenderTarget2D(this.GraphicsDevice, width, height);
```

Как всегда при использовании свойства *GraphicsDevice* класса *Game*, мы должны подождать, пока метод *LoadContent* создаст объекты *Texture2D* или *RenderTarget2D*, необходимые нашему приложению. Обычно эти объекты сохраняются в полях, что позволяет выводить их на экран позже, при выполнении перегруженного метода *Draw*.

Идея, стоящая за применением *RenderTarget2D*, довольно проста, но чтобы понять ее, необходимо подготовить почву:

Как известно, отрисовка образов на экране устройства имеет место во время выполнения перегруженного метода *Draw* класса *Game*. Закрасить весь экран определенным цветом, можно вызвав метод *Clear* (Очистить) объекта *GraphicsDevice*, ассоциированного с нашей игрой:

```
this.GraphicsDevice.Clear(Color.CornflowerBlue);
```

Отрисовку объектов *Texture2D* и текстовых строк на экране можно реализовать, используя объект *SpriteBatch*:

```
spriteBatch.Begin();
spriteBatch.Draw(…);
spriteBatch.DrawString(…);
spriteBatch.End();
```

Этот объект *SpriteBatch*, как обычно, создается в перегруженном методе *LoadContent*. Он ассоциирован с объектом *GraphicsDevice*, потому что *GraphicsDevice* необходим в его конструкторе:

```
spriteBatch = new SpriteBatch(this.GraphicsDevice);
```

Вызовы метода *Clear* объекта *GraphicsDevice* и методов *Draw* и *DrawString* объекта *SpriteBatch*, на самом деле, обеспечивают отрисовку в растровой матрице, называемой *задним буфером*, содержимое которой после этого передается на экран. Некоторые сведения о заднем буфере можно получить через свойство *PresentationParameters* (Параметры представления) объекта *GraphicsDevice*. Если приложение выполняется на телефоне с большим экраном, и для заднего буфера не обозначено то, что должны использоваться размеры, отличные от заданных по умолчанию, свойства *BackBufferWidth* (Ширина заднего буфера) и *BackBufferHeight* (Высота заднего буфера) объекта *PresentationParameters* будут возвращать значения 800 и 480, соответственно.

Класс *PresentationParameters* также определяет свойство *BackBufferFormat* (Формат заднего буфера), значением которого является член перечисления *SurfaceFormat* (Формат поверхности). Этот формат показывает и количество бит в каждом пикселе, и то, как эти биты представляют цвет. Для устройств, работающих под управлением Windows Phone 7, значением свойства *BackBufferFormat* является *SurfaceFormat.Bgr565*. Это означает, что каждый пиксел описывается 16 битами, из которых по 5 битов отведено для красного (R) и синего (B), и 6 – для зеленого (G) каналов в следующей конфигурации битов:

RRRRRGGGGGGBBBBB

Для зеленого отводится дополнительный бит, потому что он находится в центре спектра электромагнитного излучения, различимого человеческим глазом, т.е. это основной цвет, к которому наш глаз наиболее чувствителен.

Если посмотреть на градиенты цвета на экране устройства Windows Phone 7 – один из них вскоре будет представлен в данной главе – можно заметить, что они не настолько гладкие и равномерные, как выглядят градиенты на экранах настольных компьютеров. Чаще всего для настольных мониторов используются видеоадаптеры, в которых для каждого основного цвета отведено по 8 бит. Тех пяти или шести бит, которые отводятся в устройстве Windows Phone 7, совершенно не достаточно для представления градаций цвета, воспринимаемых большинством людей. Скорее всего, в будущем в устройствах Windows Phone мы выйдем за рамки 16-разрядного представления цвета.

В приложении на XNA обычный черный буфер в объекте *GraphicsDevice* временно может быть заменен объектом типа *RenderTarget2D*:

```
this.GraphicsDevice.SetRenderTarget(renderTarget);
```

После этого этот *RenderTarget2D* можно использовать для отрисовки так же, как используется черный буфер. По завершении рисования мы разрываем связь между *RenderTarget2D* и *GraphicsDevice* с помощью повторного вызова *SetRenderTarget* (Задать целевой объект прорисовки), передав в него аргумент *null*:

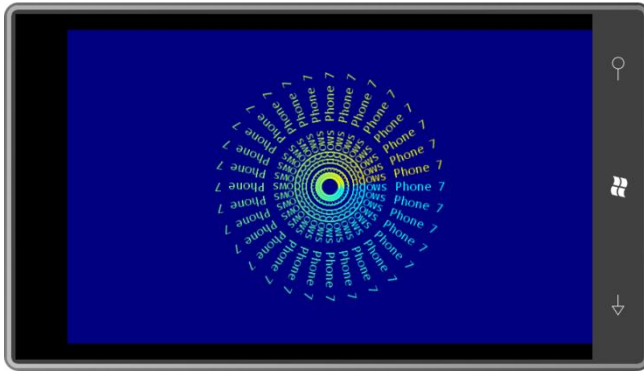
```
this.GraphicsDevice.SetRenderTarget(null);
```

Теперь *GraphicsDevice* возвращен в обычное состояние.

В случае создания *RenderTarget2D*, который остается неизменным в ходе всего выполнения приложения, вся эта операция обычно производится во время выполнения перегруженного метода *LoadContent*. Если *RenderTarget2D* должен меняться, рисование в растровой матрице может быть реализовано в ходе выполнения перегруженного *Update*. *RenderTarget2D* наследуется от *Texture2D*, поэтому *RenderTarget2D* может отображаться на экране во время выполнения перегруженного *Draw*, как любое другое изображение *Texture2D*.

Конечно же, мы не ограничены одним объектом *RenderTarget2D*. В случае наличия пакетов последовательных изображений, которые все вместе образуют некоторую анимацию, можно создавать множества объектов *RenderTarget2D* и затем отображать последовательно, как некоторого рода фильм.

Предположим, требуется выводить на экран нечто подобное:



Это набор текстовых строк «Windows Phone 7», выстроенных вокруг некоторой центральной точки, цвета которых изменяются от голубого к желтому. Конечно, мы можем включить в перегруженный *Draw* цикл, который будет делать 32 вызова метода *DrawString* объекта *SpriteBatch*. Но если объединить эти текстовые строки в одно растровое изображение, все эти вызовы в перегруженном *Draw* можно будет заменить всего одним вызовом метода *Draw* объекта *SpriteBatch*. Более того, в этом случае весь этот набор текстовых строк очень просто рассматривать как единую сущность и, например, вращать, как «волчок».

Эта идея легла в основу приложения PinwheelText (Волчок с текстом). Содержимое приложения включает *SpriteFont*, в качестве которого задан Segoe UI Mono размером 14 пунктов, но объекта *SpriteFont* нет среди полей приложения, как нет и самого текста:

Проект XNA: PinwheelText Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Vector2 screenCenter;
    RenderTarget2D renderTarget;
    Vector2 textureCenter;
    float rotationAngle;
    …
}
```

Метод *LoadContent* – самая сложная часть приложения, но результатом его выполнения является лишь задание полей *screenCenter*, *renderTarget* и *textureCenter* (Центр текстуры). Переменные *segoe14* и *textSize*, задаваемые в начале метода, как обычно, сохраняются как поля, но здесь они используются только локально:

Проект XNA: PinwheelText Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки
текстур.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Получаем сведения об окне просмотра
    Viewport viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);

    // Загружаем шрифт и получаем размер текста
    SpriteFont segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    string text = " Windows Phone 7";
    Vector2 textSize = segoe14.MeasureString(text);

    // Создаем RenderTarget2D
    renderTarget =
        new RenderTarget2D(this.GraphicsDevice, 2 * (int)textSize.X,
                                                2 * (int)textSize.X);

    // Находим центр
    textureCenter = new Vector2(renderTarget.Width / 2,
                                renderTarget.Height / 2);

    Vector2 textOrigin = new Vector2(0, textSize.Y / 2);

    // Задаем RenderTarget2D как GraphicsDevice
    this.GraphicsDevice.SetRenderTarget(renderTarget);

    // Очищаем RenderTarget2D и формируем визуальное представление текста
    this.GraphicsDevice.Clear(Color.Transparent);
    spriteBatch.Begin();

    for (float t = 0; t < 1; t += 1f / 32)
    {
        float angle = t * MathHelper.TwoPi;
        Color clr = Color.Lerp(Color.Cyan, Color.Yellow, t);
        spriteBatch.DrawString(segoe14, text, textureCenter, clr,
                               angle, textOrigin, 1, SpriteEffects.None, 0);
    }

    spriteBatch.End();

    // Возвращаем GraphicsDevice в обычное состояние
    this.GraphicsDevice.SetRenderTarget(null);
}
```

Ширина и высота создаваемого *RenderTarget2D* в два раза превышают ширину и высоту текстовой строки. *RenderTarget2D* задается в *GraphicsDevice* посредством вызова *SetRenderTarget* и затем очищается до прозрачного цвета методом *Clear*. В этот момент последовательность вызовов объекта *SpriteBatch* обеспечивает формирование 32 визуальных представлений текстовой строки на *RenderTarget2D*. Вызов *LoadContent* завершается возвращением *GraphicsDevice* к обычному заднему буферу.

Метод *Update* вычисляет угол поворота результирующего растрового изображения, обеспечивая его разворот на 360° каждые 8 секунд:

Проект XNA: PinwheelText Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
```

```
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    rotationAngle =
        (MathHelper.TwoPi * (float) gameTime.TotalGameTime.TotalSeconds / 8) %
                                            MathHelper.TwoPi;
    base.Update(gameTime);
}
```

Как и обещалось, перегруженный метод *Draw* может работать с этим *RenderTarget2D* как с обычным *Texture2D* с помощью единственного вызова *Draw* объекта *SpriteBatch*. Создается эффект синхронного вращения всех 32 текстовых строк:

Проект XNA: PinwheelText Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(renderTarget, screenCenter, null, Color.White,
                     rotationAngle, textureCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

В приложении FlyAwayHello из предыдущей главы в качестве содержимого загружались два белых растровых изображения. В этом не было особой необходимости, на самом деле. Приложение могло бы создавать их как объекты *RenderTarget2D* и затем просто закрашивать белым цветом с помощью несколько простых выражений. В методе *LoadContent* приложения FlyAwayHello эти два выражения:

```
Texture2D horzBar = Content.Load<Texture2D>("HorzBar");
Texture2D vertBar = Content.Load<Texture2D>("VertBar");
```

можно заменить следующими:

```
RenderTarget2D horzBar = new RenderTarget2D(this.GraphicsDevice, 45, 5);
this.GraphicsDevice.SetRenderTarget(horzBar);
this.GraphicsDevice.Clear(Color.White);
this.GraphicsDevice.SetRenderTarget(null);

RenderTarget2D vertBar = new RenderTarget2D(this.GraphicsDevice, 5, 75);
this.GraphicsDevice.SetRenderTarget(vertBar);
this.GraphicsDevice.Clear(Color.White);
this.GraphicsDevice.SetRenderTarget(null);
```

Да, кода стало больше, но теперь нам не нужны два файла растровых изображений в качестве содержимого приложения. Также если когда-либо нам понадобится изменить размеры этих растровых изображений, в коде это не составит никакого труда.

Приложение DragAndDraw (Перетягиваем и рисуем), которое мы рассмотрим следующим, позволяет отрисовывать посредством сенсорного ввода множество одноцветных прямоугольников. Каждый раз, когда пользователь касается и проводит пальцем по экрану, отрисовывается новый прямоугольник случайного цвета. И при этом в приложении используется всего один объект *RenderTarget2D*, содержащий всего один закрашенный белым пиксел!

Этот единственный объект *RenderTarget2D* хранится как поле вместе с коллекцией объектов *RectangleInfo*, описывающих каждый отрисованный прямоугольник:

Проект XNA: DragAndDraw Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    struct RectangleInfo
    {
        public Vector2 point1;
        public Vector2 point2;
        public Color color;
    }

    List<RectangleInfo> rectangles = new List<RectangleInfo>();
    Random rand = new Random();
    RenderTarget2D tinyTexture;
    bool isDragging;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Активируем жесты перетягивания
        TouchPanel.EnabledGestures = GestureType.FreeDrag |
                                     GestureType.DragComplete;
    }
    …
}
```

Также, обратите внимание, в конце конструктора *Game1* активируются два сенсорных жеста, *FreeDrag* и *DragComplete*. Эти жесты описывают касание экрана, проведение пальцем по экрану (в любом направлении) и снятие касания.

Метод *LoadContent* создает крошечный объект *RenderTarget2D* и закрашивает его белым цветом:

Проект XNA: DragAndDraw Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Создаем белое растровое изображение 1x1
    tinyTexture = new RenderTarget2D(this.GraphicsDevice, 1, 1);
    this.GraphicsDevice.SetRenderTarget(tinyTexture);
    this.GraphicsDevice.Clear(Color.White);
    this.GraphicsDevice.SetRenderTarget(null);
}
```

Метод *Update* обрабатывает жесты перетягивания. Как вы, возможно, помните из главы 3, статический класс *TouchPanel* распознает и простое касание, и сложные жесты. В данном приложении я использую поддержку жестов.

Если жесты активированы, необходимо задать *TouchPanel.IsGestureAvailable* (Жесты доступны) значение *true*, чтобы сделать их доступными. После этого можно вызывать

TouchPanel.ReadGesture, который возвратит объект типа *GestureSample*. *TouchPanel.IsGestureAvailable* возвращает *false*, если в этом конкретном вызове *Update* больше нет доступных жестов.

Для данного приложения значением свойства *GestureType* объекта *GestureSample* будет один из двух членов перечисления: *GestureType.FreeDrag* или *GestureType.DragComplete*. Тип *FreeDrag* указывает на то, что палец коснулся экрана или перемещается по нему. *DragComplete* указывает на то, что касание завершено.

Для жеста *FreeDrag* действительны два других свойства класса *GestureSample*: *Position* – объект *Vector2*, обозначающий текущее положение пальца относительно экрана; *Delta* – это также объект *Vector2*, который показывает разницу между текущим положением пальца и положением пальца в предыдущем *FreeDrag*. (Я не использую свойство *Delta* в данном приложении.) Эти свойства недействительны для жеста *DragComplete*.

В приложении предусмотрено поле *isDragging*, которое помогает различать первое касание экрана и проведение пальцем по нему, поскольку оба этих жеста описываются с помощью *FreeDrag*:

Проект XNA: DragAndDraw Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch (gesture.GestureType)
        {
            case GestureType.FreeDrag:
                if (!isDragging)
                {
                    RectangleInfo rectInfo = new RectangleInfo();
                    rectInfo.point1 = gesture.Position;
                    rectInfo.point2 = gesture.Position;
                    rectInfo.color = new Color(rand.Next(256),
                                               rand.Next(256),
                                               rand.Next(256));
                    rectangles.Add(rectInfo);
                    isDragging = true;
                }
                else
                {
                    RectangleInfo rectInfo = rectangles[rectangles.Count - 1];
                    rectInfo.point2 = gesture.Position;
                    rectangles[rectangles.Count - 1] = rectInfo;
                }
                break;

            case GestureType.DragComplete:
                if (isDragging)
                    isDragging = false;
                break;
        }
    }
    base.Update(gameTime);
}
```

Если *isDragging* имеет значение *false*, значит, палец впервые коснулся экрана. Приложение создает новый объект *RectangleInfo* и добавляет его в коллекцию. В этот момент значениями полей *point1* и *point2* объекта *RectangleInfo* задаются координаты точки касания экрана пальцем, и полю *color* – случайное значение *Color*.

С последующими жестами *FreeDrag* полю *point2* объекта *RectangleInfo*, добавленного в коллекцию самым последним на текущий момент, задаются координаты точки, соответствующей текущему положению пальца на экране. С формированием события *DragComplete* больше ничего не требуется делать, и полю *isDragging* присваивается значение *false*.

В перегруженном методе *Draw* (представленном ниже) приложение вызывает метод *Draw* объекта *SpriteBatch* по одному разу для каждого объекта *RectangleInfo* в коллекции. При этом каждый раз используется разновидность *Draw*, которая обеспечивает масштабирование *Texture2D* до размера целевого *Rectangle*:

```
Draw(Texture2D texture, Rectangle destination, Color color)
```

Первым аргументом всегда является белый *RenderTarget2D* размером 1×1, называемый *tinyTexture* (Крошечная текстура); и последний аргумент – это случайный цвет, хранящийся в объекте *RectangleInfo*.

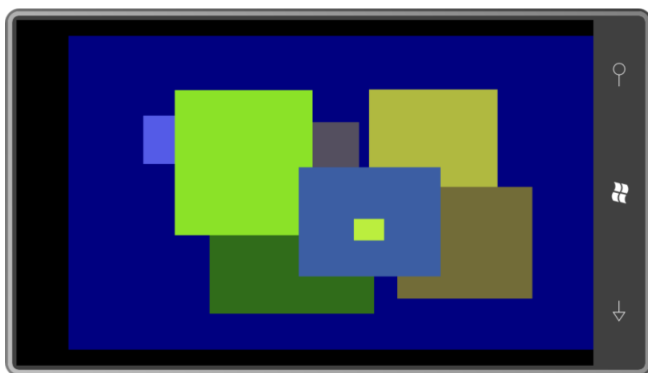
Тем не менее, аргумент *Rectangle* метода *Draw* требует некоторой обработки. Каждый объект *RectangleInfo* включает две точки, *point1* и *point2*, которые представляют противоположные углы прямоугольника, отрисовываемого пользователем. Но в зависимости от того, как палец перемещается по экрану, *point1* может быть верхним правым углом, и *point2* – нижним левым углом прямоугольника, либо *point1* может быть нижним правым углом, и *point2* – верхним левым углом прямоугольника, либо два других возможных варианта.

Передаваемый в *Draw* объект *Rectangle* должен иметь точку, обозначающую верхний левый угол прямоугольника, а также неотрицательные значения ширины и высоты. (На самом деле, *Rectangle* может также принимать точку, соответствующую нижнему правому углу, и отрицательные значения ширины и высоты, но этот небольшой факт не упрощает логики.) Для этой цели вызываются методы *Math.Min* и *Math.Abs*:

Проект XNA: DragAndDraw Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();

    foreach (RectangleInfo rectInfo in rectangles)
    {
        Rectangle rect =
            new Rectangle((int)Math.Min(rectInfo.point1.X, rectInfo.point2.X),
                          (int)Math.Min(rectInfo.point1.Y, rectInfo.point2.Y),
                          (int)Math.Abs(rectInfo.point2.X - rectInfo.point1.X),
                          (int)Math.Abs(rectInfo.point2.Y - rectInfo.point1.Y));

        spriteBatch.Draw(tinyTexture, rect, rectInfo.color);
    }

    spriteBatch.End();

    base.Draw(gameTime);
}
```

Вот как все это выглядит в действии:



Сохранение содержимого целевого объекта прорисовки

Ранее я упоминал, что для описания каждого пиксела заднего буфера Windows Phone 7 – и самого экрана – выделяется лишь 16 бит. Так какого же формата растровое изображение создает *RenderTarget2D*?

По умолчанию *RenderTarget2D* создается с разрешением 32 бита на пиксел – по 8 бит для красного, зеленого, синего и альфа каналов – соответственно члену перечисления *SurfaceFormat.Color*. Больше об этом формате я смогу сказать в конце данной главы, но в наши дни данный 32-разрядный формат считается достаточно стандартным. Это единственный формат цвета, поддерживаемый растровыми изображениями Silverlight, например.

Для повышения производительности можно создать объект *RenderTarget2D* или *Texture2D* с таким же форматом пикселов, что и в заднем буфере и экране устройства. Оба класса поддерживают конструкторы, включающие аргументы типа *SurfaceFormat* для обозначения формата цвета.

Использование объекта *RenderTarget2D* с форматом *SurfaceFormat.Bgr565* в приложении PinwheelText не приведет ни к чему хорошему. В этом формате не предусмотрен альфа-канал, поэтому фон *RenderTarget2D* не сможет быть прозрачным. Его придется специально закрашивать определенным цветом соответственно цвету фона заднего буфера.

В следующем приложении создается объект *RenderTarget2D*, который имеет не только такой же размер, что и задний буфер, но и такой же формат цвета. Однако это приложение довольно старомодно, и вот почему.

На заре Microsoft Windows на торговых выставках, где демонстрировалось множество компьютеров, очень часто можно было увидеть приложения, просто отображающие непрерывные последовательности прямоугольников разных размеров и цветов. Как реализовать такую программу на XNA? Стратегия ее написания не так очевидна. Имеет смысл добавлять новый прямоугольник в коллекцию в ходе выполнения метода *Update*, но мы не хотим создавать приложение, подобное DragAndDraw. Каждую секунду мы будем увеличивать коллекцию прямоугольников на 30 объектов. Через час перегруженный метод *Draw* уже будет пытаться сформировать визуальное представление более ста тысяч прямоугольников каждые 33 миллисекунды!

Вместо этого, вероятно, вы захотите создавать прямоугольники случайного размера и цвета на *RenderTarget2D*, размер которого соответствует размеру заднего буфера. В основе этих прямоугольников, которые будут успешно накладываться на *RenderTarget2D*, может лежать все то же белое растровое изображение 1×1, используемое в DragAndDraw.

Эти два растровых изображения хранятся как поля приложения RandomRectangles (Случайные прямоугольники) вместе с объектом *Random*:

Проект XNA: RandomRectangles Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Random rand = new Random();
    RenderTarget2D tinyTexture;
    RenderTarget2D renderTarget;
    …
}
```

Метод *LoadContent* создает два объекта *RenderTarget2D*. Для большого требуется развернутый конструктор, некоторые аргументы которого ссылаются на свойства, рассмотрение которых выходит за рамки данной книги:

Проект XNA: RandomRectangles Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    tinyTexture = new RenderTarget2D(this.GraphicsDevice, 1, 1);
    this.GraphicsDevice.SetRenderTarget(tinyTexture);
    this.GraphicsDevice.Clear(Color.White);
    this.GraphicsDevice.SetRenderTarget(null);

    renderTarget = new RenderTarget2D(
                this.GraphicsDevice,
                this.GraphicsDevice.PresentationParameters.BackBufferWidth,
                this.GraphicsDevice.PresentationParameters.BackBufferHeight,
                false,
                this.GraphicsDevice.PresentationParameters.BackBufferFormat,
                DepthFormat.None, 0, RenderTargetUsage.PreserveContents);
}
```

В конструкторе можно увидеть ссылку на *BackBufferFormat*. Но также стоит обратить внимание на последний аргумент: член перечисления *RenderTargetUsage.PreserveContents* (Сохранять содержимое). Это не опция по умолчанию. Обычно при задании *RenderTarget2D* в *GraphicsDevice* существующее содержимое растрового изображения игнорируется и, буквально, аннулируется. Опция *PreserveContents* обеспечивает сохранение существующих данных целевого объекта прорисовки и отображение каждого нового прямоугольника поверх всех предыдущих.

В методе *Update* определяются некоторые случайные значения координат и цвета, задается большой объект *RenderTarget2D* в *GraphicsDevice*, и поверх существующего содержимого отрисовывается крошечная структура с использованием этих случайных значений *Rectangle* и *Color*:

Проект XNA: RandomRectangles Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
```

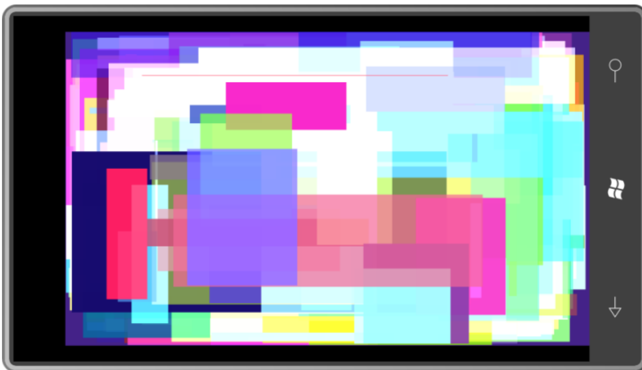
```
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    int x1 = rand.Next(renderTarget.Width);
    int x2 = rand.Next(renderTarget.Width);
    int y1 = rand.Next(renderTarget.Height);
    int y2 = rand.Next(renderTarget.Height);
    int r = rand.Next(256);
    int g = rand.Next(256);
    int b = rand.Next(256);
    int a = rand.Next(256);

    Rectangle rect = new Rectangle(Math.Min(x1, x2), Math.Min(y1, y2),
                                   Math.Abs(x2 - x1), Math.Abs(y2 - y1));
    Color clr = new Color(r, g, b, a);

    this.GraphicsDevice.SetRenderTarget(renderTarget);
    spriteBatch.Begin();
    spriteBatch.Draw(tinyTexture, rect, clr);
    spriteBatch.End();
    this.GraphicsDevice.SetRenderTarget(null);

    base.Update(gameTime);
}
```

Перегруженный *Draw* просто выводит на экран большой *RenderTarget2D* целиком:

Проект XNA: RandomRectangles Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(renderTarget, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Буквально через минуту экран уже выглядит следующим образом:



Цвета, используемые для прямоугольников, включают случайное значение альфа-канала, поэтому (как мы видим) прямоугольники частично прозрачные. Что любопытно, мы можем получить это значение прозрачности, даже несмотря на то что у создаваемого прямоугольника нет альфа-канала. Изменим конструктор для *tinyTexture* следующим образом:

```
tinyTexture = new RenderTarget2D(this.GraphicsDevice, 1, 1, false,
                                 SurfaceFormat.Bgr565, DepthFormat.None);
```

Теперь сам *tinyTexture* не имеет характеристики прозрачности, но по-прежнему его визуальное представление может быть сформировано на текстуре большего размера с частично прозрачным цветом посредством вызова метода *Draw* объекта *SpriteBatch*.

Отрисовка линий

Разработчиков, имеющих опыт работы в более популярных графических средах разработки, очень пугает отсутствие в XNA возможности формирования визуального представления простых линий и кривых в двухмерном пространстве. Я покажу два способа обойти это ограничение.

Предположим, требуется отрисовать красную линию между точками (x_1, y_1) и (x_2, y_2), и мы хотим, чтобы эта линия была толщиной 3 пиксела. Сначала создадим *RenderTarget2D*, высота которого равна 3 пикселам, и ширина определяется выражением:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Это длина линии, соединяющей две точки. Теперь определим *RenderTarget2D* как *GraphicsDevice*, очистим его *Color.Red* и вернем *GraphicsDevice* в нормальное состояние.

В ходе выполнения перегруженного метода *Draw* отрисуем это растровое изображение на экране, используя для позиционирования точку (x_1, y_1), и возьмем за центр точку (0, 1). Центр – это точка *RenderTarget2D*, координаты которой определены аргументом *position*. Создаваемая линия должна быть 3 пиксела толщиной, поэтому вертикальная ось растрового изображения должна проходить через точку (x_1, y_1). В данном вызове *Draw* также понадобится применить поворот на угол между векторами (x_1, y_1) и (x_2, y_2), который вычисляется с помощью *Math.Atan2*.

На самом деле вовсе не обязательно, чтобы размер растрового изображения соответствовал размеру линии. Мы можем использовать намного меньшее растровое изображение и применять к нему масштабирование. Самым удобным размером для этих целей является 2 пиксела в ширину и 3 пиксела в высоту. Это позволяет использовать точку (0, 1) в качестве центра в вызове *Draw*, т.е. точка (0, 1) растрового изображения остается фиксированной. Коэффициент масштабирования в горизонтальном направлении обеспечивает растяжение растрового изображения до длины линии, и коэффициент масштабирования в вертикальном направлении обеспечивает получение заданной толщины линии.

Подобный класс имеется в проекте библиотеки XNA Petzold.Phone.Xna. Я создал этот проект в Visual Studio, выбрав типом проекта Windows Phone Game Library (4.0). Рассмотрим полный код класса, который я назвал *LineRenderer* (Средство формирования визуального представления линии):

Проект XNA: Petzold.Phone.Xna Файл: LineRenderer.cs

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Petzold.Phone.Xna
{
    public class LineRenderer
    {
        RenderTarget2D lineTexture;

        public LineRenderer(GraphicsDevice graphicsDevice)
        {
            lineTexture = new RenderTarget2D(graphicsDevice, 2, 3);
```

```
            graphicsDevice.SetRenderTarget(lineTexture);
            graphicsDevice.Clear(Color.White);
            graphicsDevice.SetRenderTarget(null);
        }

        public void DrawLine(SpriteBatch spriteBatch,
                             Vector2 point1, Vector2 point2,
                             float thickness, Color color)
        {
            Vector2 difference = point2 - point1;
            float length = difference.Length();
            float angle = (float)Math.Atan2(difference.Y, difference.X);
            spriteBatch.Draw(lineTexture, point1, null, color, angle,
                             new Vector2(0, 1),
                             new Vector2(length / 2, thickness / 3),
                             SpriteEffects.None, 0);
        }
    }
}
```

Конструктор создает небольшой белый *RenderTarget2D*. Метод *DrawLine* (Отрисовать линию) принимает обязательный аргумент типа *SpriteBatch* и вызывает метод *Draw* этого объекта. Обратите внимание на коэффициент масштабирования, он является 7 аргументом в вызове метода *Draw*. Ширина *RenderTarget2D* составляет 2 пиксела, т.е. коэффициент горизонтального масштабирования равен половине длины линии. Высота растрового изображения – 3 пиксела, т.е. коэффициент вертикального масштабирования соответствует толщине линии, деленной на 3. Выбранная мною высота в 3 пиксела обеспечивает гарантированное прохождение линии через геометрическую точку независимо от ее толщины.

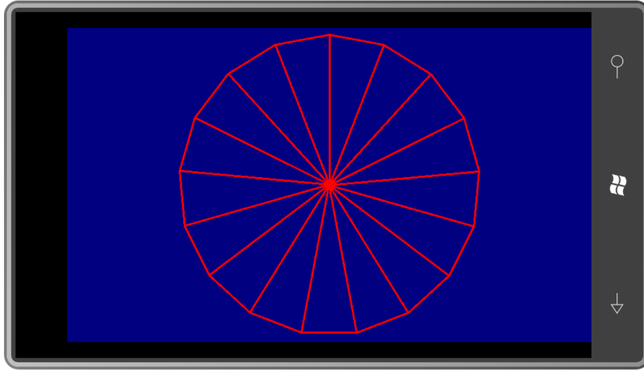
Чтобы использовать этот класс в своем приложении, сначала необходимо выполнить сборку этого проекта библиотеки. После этого в Solution Explorer любого обычного проекта XNA щелкните правой кнопкой мыши раздел References и выберите Add Reference. В появившемся диалоговом окне Add Reference выберите пункт Browse. Перейдите к папке с файлом Petzold.Phone.Xna.dll и выберите его.

В файл кода потребуется добавить директиву *using*:

```
using Petzold.Phone.Xna;
```

Я создам объект *LineRenderer* в перегруженном методе *LoadContent* и затем вызову *DrawLine* в перегруженном *Draw*, передавая в него объект *SpriteBatch*, используемый для отрисовки остальных двухмерных графических элементов.

Все это продемонстрировано в проекте TapForPolygon (Коснуться для создания многоугольника). Приложение начинает выполнение с отрисовки треугольника, образованного линиями, проведенными из центра к каждой из вершин. По касанию экрана треугольник становится квадратом, затем пятиугольником и т.д.:



Класс *Game1* включает поля для *LineRenderer*, а также пару полезных переменных.

Проект XNA: TapForPolygon Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    LineRenderer lineRenderer;
    Vector2 center;
    float radius;
    int vertexCount = 3;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Активируем касания
        TouchPanel.EnabledGestures = GestureType.Tap;
    }
    …
}
```

Обратите внимание, жест *Tap* активируется в конструкторе. *LineRenderer* создается в перегруженном *LoadContent*:

Проект XNA: TapForPolygon Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Viewport viewport= this.GraphicsDevice.Viewport;
    center = new Vector2(viewport.Width / 2, viewport.Height / 2);
    radius = Math.Min(center.X, center.Y) - 10;

    lineRenderer = new LineRenderer(this.GraphicsDevice);
}
```

Перегруженный метод *Update* отвечает за выявление факта касания. Если факт касания установлен, переменная *vertexCount* (Количество вершин) получает приращение,

обеспечивая переход от, скажем, шестнадцатиугольника к семнадцатиугольнику, как показано выше.

Проект XNA: TapForPolygon Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
        if (TouchPanel.ReadGesture().GestureType == GestureType.Tap)
            vertexCount++;

    base.Update(gameTime);
}
```

Визуальное представление линий, которые на самом деле являются всего одним объектом *RenderTarget2D*, вытянутым в виде линий, формируется в перегруженном методе *Draw*. В основе цикла *for* лежит переменная *vertexCount*. С каждой итерацией выполняется отрисовка двух линий: одна проводится из центра к новой вершине, и другая – из предыдущей вершины в новую вершину:

Проект XNA: TapForPolygon Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();

    Vector2 saved = new Vector2();

    for (int vertex = 0; vertex <= vertexCount; vertex++)
    {
        double angle = vertex * 2 * Math.PI / vertexCount;
        float x = center.X + radius * (float)Math.Sin(angle);
        float y = center.Y - radius * (float)Math.Cos(angle);
        Vector2 point = new Vector2(x, y);

        if (vertex != 0)
        {
            lineRenderer.DrawLine(spriteBatch, center, point, 3, Color.Red);
            lineRenderer.DrawLine(spriteBatch, saved, point, 3, Color.Red);
        }
        saved = point;
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```

LineRenderer может использоваться для отрисовки линий не только просто на экране, но и на других объектах *RenderTarget2D*. Одно из возможных подобных применений класса *LineRenderer* – приложение для «рисования пальцами», которое позволяет отрисовывать линии произвольной формы посредством сенсорного ввода, т.е. буквально проводя пальцем по экрану. Следующий проект является очень простым первым приближением такого

приложения. Отрисовываемые в нем линии всегда красного цвета толщиной 25 пикселов. Рассмотрим поля и конструктор (и пусть вас не пугает имя проекта[1]):

Проект XNA: FlawedFingerPaint Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    RenderTarget2D renderTarget;
    LineRenderer lineRenderer;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Активируем жесты
        TouchPanel.EnabledGestures = GestureType.FreeDrag;
    }
    …
}
```

Заметьте, что активирован только жест *FreeDrag*. Каждый жест обеспечивает отрисовку короткой линии, соединенной с предыдущей.

Объект *RenderTarget2D* под именем *renderTarget* используется как своего рода «холст», на котором можно рисовать посредством сенсорного ввода. Он создается в методе *LoadContent*, что обеспечивает соответствие его размеров размерам заднего буфера, использование одного и того же формата цвета и сохранение содержимого:

Проект XNA: FlawedFingerPaint Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    renderTarget = new RenderTarget2D(
                        this.GraphicsDevice,
                        this.GraphicsDevice.PresentationParameters.BackBufferWidth,
                        this.GraphicsDevice.PresentationParameters.BackBufferHeight,
                        false,
                        this.GraphicsDevice.PresentationParameters.BackBufferFormat,
                        DepthFormat.None, 0, RenderTargetUsage.PreserveContents);

    this.GraphicsDevice.SetRenderTarget(renderTarget);
    this.GraphicsDevice.Clear(Color.Navy);
    this.GraphicsDevice.SetRenderTarget(null);

    lineRenderer = new LineRenderer(this.GraphicsDevice);
}
```

В перегруженном *LoadContent* также создается объект *LineRenderer*.

[1] FlawedFingerPaint – несовершенное приложение для рисования посредством сенсорного ввода (*прим. переводчика*).

Как помним, жест *FreeDrag* сопровождается свойством *Position*, которое указывает на текущую точку касания, и свойством *Delta*, которое представляет разницу между текущей и предыдущей точками касания. Предыдущую точку касания можно вычислить, вычитая значение *Delta* из значения *Position*. Эти две точки используются для отрисовки короткой линии на холсте *RenderTarget2D*:

Проект XNA: FlawedFingerPaint Файл: Game1.cs (фрагмент)

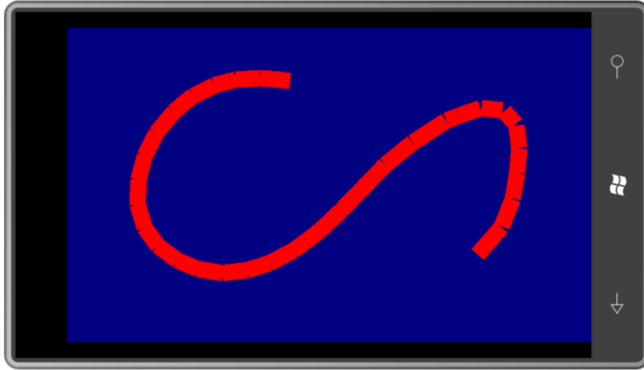
```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag &&
            gesture.Delta != Vector2.Zero)
        {
            this.GraphicsDevice.SetRenderTarget(renderTarget);
            spriteBatch.Begin();
            lineRenderer.DrawLine(spriteBatch,
                                  gesture.Position,
                                  gesture.Position - gesture.Delta,
                                  25, Color.Red);
            spriteBatch.End();
            this.GraphicsDevice.SetRenderTarget(null);
        }
    }
    base.Update(gameTime);
}
```

После этого перегруженному методу *Draw* остается лишь отрисовать холст на экране:

Проект XNA: FlawedFingerPaint Файл: Game1.cs (фрагмент)

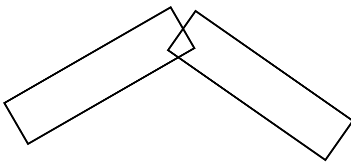
```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(renderTarget, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Это приложение действительно замечательно работает. Быстро проведите пальцем по экрану, и на нем появится изогнутая линия:



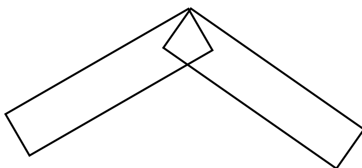
Единственная проблема в том, что линия выглядит несплошной, и это особенно заметно в местах перегибов кривой. Это вполне оправдывает имя данного проекта.

Если подумать о том, визуальное представление каких объектов формируется здесь на самом деле, станет абсолютно понятно, откуда все эти разрывы. Ведь фактически здесь отрисовываются прямоугольники между парами точек, и если эти прямоугольники располагаются под углом друг к другу, возникает «разлом» в кривой:



Это практически незаметно для тонких линий, но бросается в глаза при отрисовке более толстых кривых.

Что можно сделать? Если метод, отображающий эти прямоугольные текстуры, осведомлен о том, что выполняется отрисовка последовательности линий (что называется *полилинией* в кругах разработчиков графики), он может немного увеличивать коэффициент масштабирования растрового изображения в вертикальном направлении, обеспечивая соприкосновение прямоугольников в вершинах, а не по центральной оси:



Такое точное сопоставление углов обеспечат определенные вычисления с участием угла между этими двумя линиями. Данную методику придется немного скорректировать для приложения рисования посредством сенсорного ввода, потому что в момент формирования визуального представления текущей линии мы не можем знать, как пройдет следующая линия.

В средах, поддерживающих функции отрисовки линий (таких как Silverlight) такие проблемы тоже возникают при использовании значений свойств по умолчанию. Но в Silverlight можно задать скругленные наконечники линий, и все соединения будут выглядеть гладкими.

В XNA скругления наконечников, вероятно, лучше всего обрабатывать напрямую через изменение значений пикселов.

Работа с пикселами

Ранее в данной главе я продемонстрировал, как создавать пустой объект *Texture2D*, используя один из его конструкторов:

```
Texture2D texture = new Texture2D(this.GraphicsDevice, width, height);
```

Как и для заднего буфера, и для *RenderTarget2D*, формат описания пикселов определяется членом перечисления *SurfaceFormat*. Свойство *Format* (Формат) объекта *Texture2D*, созданного с помощью этого простого конструктора, будет иметь значение *SurfaceFormat.Color*. Это означает, что каждый пиксел описывается 4 байтами (или 32 битами) данных, по одному байту для значений красного, зеленого и синего и еще один байт для альфа-канала, который определяет непрозрачность этого пиксела.

Также возможно (и очень удобно) рассматривать каждый пиксел как 32-разрядное целое без знака, что в C#обозначают как *uint*. Цвета представляются 8-разрязными шестнадцатеричными значениями типа *uint*:

AABBGGRR

Каждая буква представляет четыре бита. Если имеется *Texture2D*, загружаемый как содержимое или создаваемый, как показано выше, и его свойство *Format* имеет значение *SurfaceFormat*, все значения битов растрового изображения можно получить, предварительно создав массив типа *uint*, достаточно большой для включения всех пикселов:

```
uint[] pixels = new uint[texture.width * texture.height];
```

После этого все пикселы *Texture2D* передаются в массив следующим образом:

```
texture.GetData<uint>(pixels);
```

GetData – обобщенный метод, в котором необходимо просто указать тип данных массива. Перегрузки *GetData* позволяют получать выборки пикселов, соответствующие прямоугольным подмножествам растрового изображения, или передавать пикселы растрового изображения в массив *pixels* с определенным смещением.

RenderTarget2D наследуется от *Texture2D*, поэтому эта методика может применяться и к объектам *RenderTarget2D*.

Также можно передать данные из массива *pixels* назад в растровое изображение:

```
texture.SetData<uint>(pixels);
```

Пикселы в массиве *pixels* организованы по строкам, начиная с самой верхней строки. Пикселы в каждой строке размещаются слева направо. Для конкретной строки *y* и столбца *x* растрового изображения массив *pixels* индексируется по следующей простой формуле:

```
pixels[y * texture.width + x]
```

Структура *Color* имеет одно исключительно удобное свойство – *PackedValue* (Упакованное значение). Оно обеспечивает преобразование объекта *Color* в *uint* конкретного формата, который необходим для этого массива, например:

```
pixels[y * texture.width + x] = Color.Fuchsia.PackedValue;
```

Кстати, *Color* и *uint* так тесно взаимосвязаны, что как альтернативный вариант можно создать массив *pixels* типа *Color*:

```
Color[] pixels = new Color[texture.Width * texture.Height];
```

После этого данный массив может использоваться в *GetData*

```
texture.GetData<Color>(pixels);
```

и *SetData*

```
texture.SetData<Color>(pixels);
```

и для задания отельным пикселам значений *Color*:

```
pixels[y * texture.width + x] = Color.AliceBlue;
```

Единственным обязательным условием является последовательность.

Можно создавать объекты *Texture2D* в других цветовых форматах, но массив пикселов должен включать члены соответствующего размера, например, *ushort* для формата *SurfaceFormat.Bgr565*. Следовательно, формат *SurfaceFormat.Color* использовать легче всего, поэтому именно его я придерживаюсь в данной главе.

Рассмотрим простой пример. Предположим, для игры требуется создать градиентный фон, меняющийся слева направо от синего к красному. Это реализовано в проекте GradientBackground (Градиентный фон). Рассмотрим поля:

Проект XNA: GradientBackground Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Rectangle viewportBounds;
    Texture2D background;
    …
}
```

Вся основная работа выполняется перегруженным методом *LoadContent*: создается растровая матрица на базе размера *Viewport* (здесь используется свойство *Bounds*, которое имеет удобные целочисленные размеры) и заполняется данными. Интерполяция для создания градиента выполняется методом *Color.Lerp* по значению *x*:

Проект XNA: GradientBackground Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    viewportBounds = this.GraphicsDevice.Viewport.Bounds;
    background = new Texture2D(this.GraphicsDevice, viewportBounds.Width,
                                                    viewportBounds.Height);

    Color[] pixels = new Color[background.Width * background.Height];

    for (int x = 0; x < background.Width; x++)
    {
        Color clr = Color.Lerp(Color.Blue, Color.Red,
                               (float)x / background.Width);

        for (int y = 0; y < background.Height; y++)
            pixels[y * background.Width + x] = clr;
    }
```

```
    background.SetData<Color>(pixels);
}
```

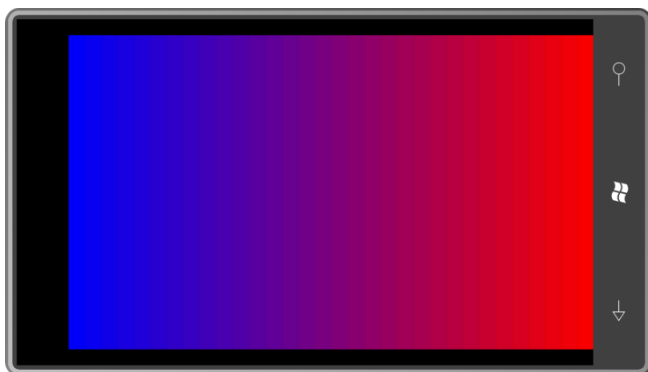
Не забывайте вызывать *SetData* после заполнения данными массива *pixels*! Приятно предполагать наличие некоторой неявной привязки между *Texture2D* и массивом, но на самом деле ее нет.

Метод *Draw* просто отрисовывает *Texture2D*, как обычно:

Проект XNA: GradientBackground   Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(background, viewportBounds, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Получаем такой градиент:



Хотя кажется, что код подразумевает сотни переходных оттенков между чисто-синим и чисто-красным, 16-битовое цветовое разрешение экрана устройства Windows Phone 7 позволяет четко отображать только 32 полосы[1].

Для данного конкретного примера, в котором *Texture2D* остается неизменным в направлении сверху вниз, нет необходимости в таком большом количестве строк. Кстати, объект *background* может быть создан всего в одну строку:

```
background = new Texture2D(this.GraphicsDevice, viewportBounds.Width, 1);
```

Весь остальной код *LoadContent* основывается на свойствах *background.Width* и *background.Height*, поэтому больше ничего менять не надо (хотя циклы, конечно, можно было бы упростить). В методе *Draw* растровые изображения растягиваются для заполнения *Rectangle*:

```
spriteBatch.Draw(background, viewportBounds, Color.White);
```

Ранее в данной главе я создавал белый *RenderTarget2D* размером 1×1, используя такой код:

```
tinyTexture = new RenderTarget2D(this.GraphicsDevice, 1, 1);
this.GraphicsDevice.SetRenderTarget(tinyTexture);
```

[1] На самом деле это не совсем так. Глубина цвета в Windows Phone 7 позволит отобразить более качественное изображение, чем описывает автор. Скорее всего, это связанно с тем, что книга писалась до появления финальной версии платформы (*прим. научного редактора*).

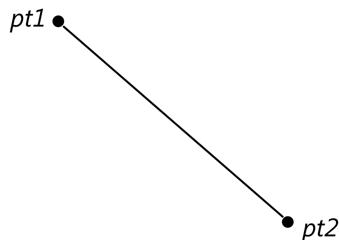
```
this.GraphicsDevice.Clear(Color.White);
this.GraphicsDevice.SetRenderTarget(null);
```

С *Texture2D* то же самое можно сделать всего двумя строками кода, используя инициализацию массива прямо в строке:

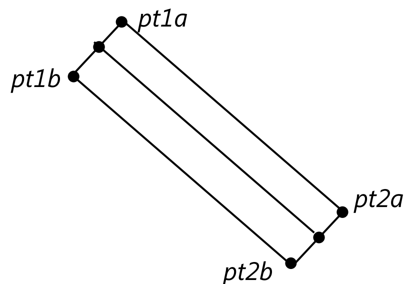
```
tinyTexture = new Texture2D(this.GraphicsDevice, 1, 1);
tinyTexture.SetData<Color>(new Color[] { Color.White });
```

Геометрия рисования прямых линий

При отрисовке линий в *Texture2D* было бы удобно напрямую задавать пикселы в растровой матрице для формирования визуального представления линии. Для анализа и иллюстрации предположим, что требуется провести линию между точками *pt1* и *pt2*:



Данная линия с точки зрения геометрии имеет нулевую толщину, но отрисованная линия не может быть нулевой толщины. Примем ее равной 2*R* пикселам. (*R* означает *радиус*, и вскоре станет понятно, почему я оперирую здесь этими понятиями.) На самом деле мы хотим нарисовать прямоугольник, отступая от *pt1* и *pt2* в стороны на *R* пикселов:



Как вычислить координаты этих углов? Это очень просто сделать с помощью векторов. Найдем ненормализованный вектор, направленный из *pt1* в *pt2*, и нормализуем его:

```
Vector2 vector = Vector2.Normalize(pt2 - pt1);
```

Этот вектор необходимо поворачивать с шагом в 90 градусов, и тут есть небольшая хитрость. Чтобы повернуть *vector* на 90 градусов по часовой стрелке, поменяйте координаты X и Y местами и умножьте координату Y на -1:

```
Vector2 vect90 = new Vector2(-vector.Y, vector.X)
```

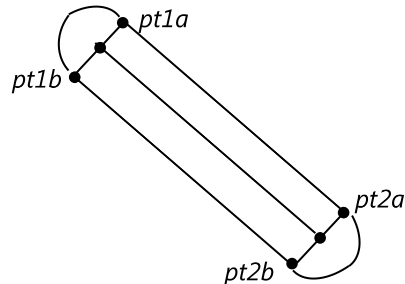
Вектор, развернутый на –90 градусов относительно *vector* – это *vect90*, направленный в противоположную сторону.

Если *vector* направлен из *pt1* в *pt2*, тогда вектор из *pt1* в *pt1a* (например) является этим же вектором, развернутым на –90 и длиной *R*. Тогда чтобы получить *pt1a*, добавим этот вектор к *pt1*.

```
Vector2 pt1a = pt1 - R * vect90;
```

Таким же образом можно найти *pt1b*, *pt2a* и *pt2b*.

Но как было продемонстрировано ранее, прямоугольники не обеспечивают должного эффекта при отрисовке толстых линий, соединяемых под углом. Чтобы избежать возникновения разломов, к этим прямоугольникам необходимо добавить скругленные наконечники:



Эти наконечники реализованы как полуокружности радиусом R с центром в точках *pt1* и *pt2*.

На данный момент мы получили полный контур линии между двумя точками: линия из *pt1a* в *pt2a*, полукруг из *pt2a* в *pt2b*, еще одна линия из *pt2b* в *pt1b* и еще один полукруг из *pt1b* в *pt1a*. Цель – найти все пикселы (x, y), попадающие в этот контур.

Для рисования векторных контуров идеально подходят параметрические уравнения. При закрашивании областей лучше вернуться к стандартным уравнениям, которые мы все изучали в старшей школе. Вспомним уравнения прямой с угловым коэффициентом:

$$y = mx + b$$

где *m* – это наклон линии, и *b* – значение *y* в точке пересечения линии с осью Y.

Но в компьютерной графике области традиционно заполняются на основании горизонтальных строк развертки, которые также называют строками растра. (Эти термины пришли из телевидения.) Данное уравнение прямой представляет *x* как функцию *y*:

$$x = ay + b$$

Для линии из *pt1* в *pt2*

$$a = \frac{pt2.X - pt1.X}{pt2.Y - pt1.Y}$$

$$b = pt1.X - a \cdot pt1.Y$$

Для любого *y* на линии имеется точка, соединяющая *pt1* и *pt2*, если *y* лежит между *pt1.Y* и *pt2.Y*. Значение *x* находится из уравнений линии.

Посмотрите на предыдущий рисунок и представьте горизонтальную строку развертки, которая пересекает эти две линии, проходящие из *pt1a* в *pt2a* и из *pt1b* в *pt2b*. Для любого *y* может быть найден *xa* на линии, соединяющей *pt1a* и *pt2a*, и *xb* на линии, соединяющей *pt1b* и *pt2b*. На этой строке развертки закрашиваемые пикселы лежат в диапазоне между (*xa*, *y*) и (*xb*, *y*). Эту же операцию можно повторить для всех *y*.

Для скругленных наконечников все сложнее, но не на много. Все точки (*x*, *y*) окружности радиусом *R* с центром в начале координат удовлетворяют уравнению:

$$x^2 + y^2 = R^2$$

Для окружности с центром в точке (*xc*, *yc*) это уравнение принимает вид:

$$(x - xc)^2 + (y - yc)^2 = R^2$$

Или для любого *y*:

$$x = xc \pm \sqrt{R^2 - (y - yc)^2}$$

Если выражение под корнем отрицательное, *y* лежит вне окружности. В противном случае для каждого *y* существует (как правило) два значения *x*. Единственным исключением является равенство подкоренного выражения нулю, т.е. когда *y* отстоит от *yc* ровно на *R* единиц, что соответствует верхней и нижней точкам окружности.

Мы имеем дело с половинами окружностей, поэтому все несколько сложнее. Рассмотрим верхний полукруг. Его центр лежит в точке *pt1*, и дуга, равная половине окружности, соединяет точки *pt1b* и *pt1a*. Линия между *pt1* и *pt1b* лежит под определенным углом *angle1*, который можно вычислить с помощью *Math.Atan2*. Аналогично линия, соединяющая точки *pt1* и *pt1a*, лежит под углом *angle2*. Если точка (*x*, *y*) лежит на окружности, как было вычислено выше, линия, соединяющая центр окружности и эту точку, тоже образует угол *angle*. Если этот угол лежит в диапазоне между *angle1* и *angle2*, данная точка принадлежит интересующей нас половине окружности. (Термин «между» несколько неточен в данном случае, потому что возвращаемые *Math.Atan2* значения углов лежат в диапазоне от π до –π.)

Теперь для любого *y* мы можем проверить обе линии и обе полуокружности и найти все точки (*x*, *y*), лежащие на этих четырех фигурах. Таких точек будет максимум две: одна там, где строка развертки входит в фигуру, и другая в месте ее выхода. Для этой конкретной строки развертки все пикселы между данными двумя точками могут закрашиваться.

Проект Petzold.Phone.Xna включает несколько структур, которые помогают в отрисовке линий в *Texture2D*. (Я сделал их структурами, а не классами, поскольку, возможно, их экземпляры будут часто создаваться в ходе вызовов *Update*.) Все эти структуры реализовывают следующий простой интерфейс:

Проект XNA: Petzold.Phone.Xna Файл: IGeometrySegment.cs

```
using System.Collections.Generic;

namespace Petzold.Phone.Xna
{
    public interface IGeometrySegment
    {
        void GetAllX(float y, IList<float> xCollection);
    }
}
```

Для любого значения *y* метод *GetAllX* добавляет элементы в коллекцию значений *x*. На практике, когда структуры находятся в библиотеке, эта коллекция часто будет возвращаться пустой. Иногда она будет содержать только одно значение, иногда два.

Рассмотрим структуру *LineSegment*:

Проект XNA: Petzold.Phone.Xna Файл: LineSegment.cs

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace Petzold.Phone.Xna
{
```

```
    public struct LineSegment : IGeometrySegment
    {
        readonly float a, b;         // как в x = ay + b

        public LineSegment(Vector2 point1, Vector2 point2) : this()
        {
            Point1 = point1;
            Point2 = point2;

            a = (Point2.X - Point1.X) / (Point2.Y - Point1.Y);
            b = Point1.X - a * Point1.Y;
        }

        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            if ((Point2.Y > Point1.Y && y >= Point1.Y && y < Point2.Y) ||
                (Point2.Y < Point1.Y && y <= Point1.Y && y > Point2.Y))
            {
                xCollection.Add(a * y + b);
            }
        }
    }
}
```

Обратите внимание, что выражение *if* в методе *GetAllX* проверяет, лежит ли *y* между *Point1.Y* и *Point2.Y*; оно допускает значения *y*, равные *Point1.Y*, но не допускает их равенство *Point2.Y*. Иначе говоря, оно определяет линию как множество точек от *Point1* (включая) до, но не включая *Point2*. Эта предосторожность относительно включения или невключения точек обретает смысл при соединении множества линий и дуг: она позволяет избежать возможности дублирования значений *x* в коллекции.

Также обратите внимание на отсутствие каких-либо специальных ограничений относительно горизонтальных линий, т.е. линий, для которых *Point1.Y* равна *Point2.Y*, и *a* равна бесконечности. В этом случае условие выражения *if* метода не выполняется, строка развертки никогда не пересекает горизонтальную граничную линию.

Следующая структура аналогична, но описывает обобщенную дугу окружности:

Проект XNA: Petzold.Phone.Xna Файл: ArcSegment.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace Petzold.Phone.Xna
{
    public struct ArcSegment : IGeometrySegment
    {
        readonly double angle1, angle2;

        public ArcSegment(Vector2 center, float radius,
                          Vector2 point1, Vector2 point2) :
            this()
        {
            Center = center;
            Radius = radius;
            Point1 = point1;
            Point2 = point2;
            angle1 = Math.Atan2(point1.Y - center.Y, point1.X - center.X);
            angle2 = Math.Atan2(point2.Y - center.Y, point2.X - center.X);
```

```
        }

        public Vector2 Center { private set; get; }
        public float Radius { private set; get; }
        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            double sqrtArg = Radius * Radius - Math.Pow(y - Center.Y, 2);

            if (sqrtArg >= 0)
            {
                double sqrt = Math.Sqrt(sqrtArg);
                TryY(y, Center.X + sqrt, xCollection);
                TryY(y, Center.X - sqrt, xCollection);
            }
        }

        public void TryY(double y, double x, IList<float> xCollection)
        {
            double angle = Math.Atan2(y - Center.Y, x - Center.X);

            if ((angle1 < angle2 && (angle1 <= angle && angle < angle2)) ||
                (angle1 > angle2 && (angle1 <= angle || angle < angle2)))
            {
                xCollection.Add((float)x);
            }
        }
    }
}
```

Довольно сложное (но симметричное) выражение *if* в методе *TryY* отвечает за ограничение значений угла диапазоном от π до –π. Также обратите внимание, что сравнение *angle* с *angle1* и *angle2* допускает варианты, когда *angle* равен *angle1*, но не допускает равенства *angle* и *angle2*. Таким образом, допускаются все углы от *angle1* (включая) до, но не включая *angle2*.

Итак, окончательная структура, используемая при отрисовке линий и представляющая прямую линию со скругленными наконечниками, выглядит следующим образом:

Проект XNA: Petzold.Phone.Xna Файл: RoundCappedLines.cs

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace Petzold.Phone.Xna
{
    public class RoundCappedLine : IGeometrySegment
    {
        LineSegment lineSegment1;
        ArcSegment arcSegment1;
        LineSegment lineSegment2;
        ArcSegment arcSegment2;

        public RoundCappedLine(Vector2 point1, Vector2 point2, float radius)
        {
            Point1 = point1;
            Point2 = point2;
            Radius = radius;

            Vector2 vector = point2 - point1;
            Vector2 normVect = vector;
            normVect.Normalize();
```

```
            Vector2 pt1a = Point1 + radius * new Vector2(normVect.Y, -normVect.X);
            Vector2 pt2a = pt1a + vector;
            Vector2 pt1b = Point1 + radius * new Vector2(-normVect.Y, normVect.X);
            Vector2 pt2b = pt1b + vector;

            lineSegment1 = new LineSegment(pt1a, pt2a);
            arcSegment1 = new ArcSegment(point2, radius, pt2a, pt2b);
            lineSegment2 = new LineSegment(pt2b, pt1b);
            arcSegment2 = new ArcSegment(point1, radius, pt1b, pt1a);
        }

        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }
        public float Radius { private set; get; }

        public void GetAllX(float y, IList<float> xCollection)
        {
            arcSegment1.GetAllX(y, xCollection);
            lineSegment1.GetAllX(y, xCollection);
            arcSegment2.GetAllX(y, xCollection);
            lineSegment2.GetAllX(y, xCollection);
        }
    }
}
```

Данная структура включает два объекта *LineSegment* и два объекта *ArcSegment* и определяет их все, используя аргументы, передаваемые в ее собственный конструктор. Реализация *GetAllX* заключается в вызове этого же метода для всех четырех компонентов. Код, вызывающий *GetAllX*, должен обеспечивать предварительную очистку коллекции. Для *RoundCappedLines* (Линии со скругленными наконечниками) этот метод будет возвращать коллекцию либо с одним значением *x* (для целей закрашивания этот случай может быть проигнорирован), либо с двумя значениями *x*, в этом случае пикселы, лежащие между этими двумя значениями, могут быть закрашены.

Использовать эту структуру в реальном приложении не так просто, как класс *LineRenderer*. Как это делается, продемонстрировано в проекте BetterFingerPaint (Более совершенное приложение для рисования посредством сенсорного ввода). Поля включают *Texture2D* для рисования, массив пикселов для этой текстуры и пригодную для повторного использования коллекцию объектов *float* для передачи в структуры рисования линий.

Проект XNA: BetterFingerPaint   Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D canvas;
    Color[] pixels;
    List<float> xCollection = new List<float>();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Активируем жесты FreeDrag
        TouchPanel.EnabledGestures = GestureType.FreeDrag;
    }
```

```
    …
}
```

Конструктор *Game1* активирует жест *FreeDrag* и, как обычно, эти жесты обрабатываются в перегруженном *Update*, показанном ниже.

Перегруженный метод *LoadContent* создает *Texture2D* размером соответственно размеру экрана и затем инициализирует его, присваивая пикселам *Color.Navy*:

Проект XNA: BetterFingerPaint Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Rectangle viewportBounds = this.GraphicsDevice.Viewport.Bounds;
    canvas = new Texture2D(this.GraphicsDevice, viewportBounds.Width,
                                               viewportBounds.Height);

    pixels = new Color[canvas.Width * canvas.Height];

    for (int i = 0; i < pixels.Length; i++)
        pixels[i] = Color.Navy;

    canvas.SetData<Color>(pixels);
}
```

Ключевой вызов в перегруженном *Update* – это конструктор *RoundCappedLine* с двумя точками и радиусом, который равен половине толщины линии. После этого цикл может перебирать все значения Y холста, вызывать метод *GetAllX* объекта *RoundCappedLine* и закрашивать область между значениями X, возвращенными в коллекции. Но подпрограмма пытается ограничить цикл и вызовы метода только теми значениями X и Y, которые могли быть затронуты конкретным жестом.

Проект XNA: BetterFingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    bool canvasNeedsUpdate = false;
    int yMinUpdate = Int32.MaxValue, yMaxUpdate = 0;

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag &&
            gesture.Delta != Vector2.Zero)
        {
            Vector2 point1 = gesture.Position - gesture.Delta;
            Vector2 point2 = gesture.Position;
            float radius = 12;

            RoundCappedLine line = new RoundCappedLine(point1, point2, radius);

            int yMin = (int)(Math.Min(point1.Y, point2.Y) - radius - 1);
            int yMax = (int)(Math.Max(point1.Y, point2.Y) + radius + 1);
```

```
            yMin = Math.Max(0, Math.Min(canvas.Height, yMin));
            yMax = Math.Max(0, Math.Min(canvas.Height, yMax));

            for (int y = yMin; y < yMax; y++)
            {
                xCollection.Clear();
                line.GetAllX(y, xCollection);

                if (xCollection.Count == 2)
                {
                    int xMin = (int)(Math.Min(xCollection[0],
                                              xCollection[1]) + 0.5f);
                    int xMax = (int)(Math.Max(xCollection[0],
                                              xCollection[1]) + 0.5f);

                    xMin = Math.Max(0, Math.Min(canvas.Width, xMin));
                    xMax = Math.Max(0, Math.Min(canvas.Width, xMax));

                    for (int x = xMin; x < xMax; x++)
                    {
                        pixels[y * canvas.Width + x] = Color.Red;
                    }
                    yMinUpdate = Math.Min(yMinUpdate, yMin);
                    yMaxUpdate = Math.Max(yMaxUpdate, yMax);
                    canvasNeedsUpdate = true;
                }
            }
        }
    }

    if (canvasNeedsUpdate)
    {
        this.GraphicsDevice.Textures[0] = null;

        int height = yMaxUpdate - yMinUpdate;
        Rectangle rect = new Rectangle(0, yMinUpdate, canvas.Width, height);
        canvas.SetData<Color>(0, rect, pixels,
                    yMinUpdate * canvas.Width, height * canvas.Width);
    }
    base.Update(gameTime);
}
```

После обработки всех жестов – а за один вызов *Update* может обрабатываться более одного жеста – метод имеет в своем распоряжении значения *yMinUpdate* и *yMaxUpdate*, указывающие на строки, которые были подвержены влиянию в ходе этих конкретных жестов. Эти значения используются для создания объекта *Rectangle*, обеспечивающего обновление холста *Texture2D* из массива *pixels* только там, где пикселы изменились.

Проще всего вызвать *SetData* следующим образом:

```
texture.SetData<Color>(pixels);
```

Существует такой альтернативный вариант:

```
texture.SetData<Color>(pixels, startIndex, count);
```

Этот вызов метода обеспечивает заполнение всего *Texture2D* значениями из массива *pixels*, начиная со значения с индексом *startIndex* (Начальный индекс). Аргумент *count* по-прежнему должен быть равен произведению ширины и высоты *Texture2D* в пикселах. Количество значений в массиве, начиная с индекса *startIndex*, должно соответствовать *count*. Этот вариант метода удобно использовать в случаях, когда один и тот же массив *pixels* применяется для заполнения нескольких небольших объектов *Texture2D*.

Третья разновидность метода:

```
texture.SetData<Color>(0, rectangle, pixels, startIndex, count);
```

Аргумент *rectangle* типа *Rectangle* определяет прямоугольник в рамках *Texture2D*, и только этот прямоугольник будет обновляться. *startIndex* по-прежнему задает индекс массива *pixels*, с которого будет производится обработка элементов этого массива. *count* должен быть равен произведению высоты и ширины *rectangle*. Данный метод предполагает, что для закрашивания заданной прямоугольной области используются пикселы массива *pixels* в количестве, равном *count*, начиная с пиксела с индексом *startIndex*.

Предположим, мы работаем с единственным массивом *pixels*, который соответствует полному *Texture2D*, и требуется ограничить изменения конкретной прямоугольной областью. Мы не располагаем возможностью задавать любой прямоугольник, потому что строки пикселов в массиве *pixels* по-прежнему опираются на полную ширину *Texture2D*. Это означает, что ширина прямоугольника должна соответствовать ширине *Texture2D*. Короче говоря, в вызове *SetData* могут участвовать только полные строки, одна или более. Именно поэтому в коде сохраняются только *yMinUpdate* и *yMaxUpdate*, и не сохраняются эквивалентные значения X.

В показанном выше методе *Update* этот вызов можно увидеть перед вызовом *SetData*:

```
this.GraphicsDevice.Textures[0] = null;
```

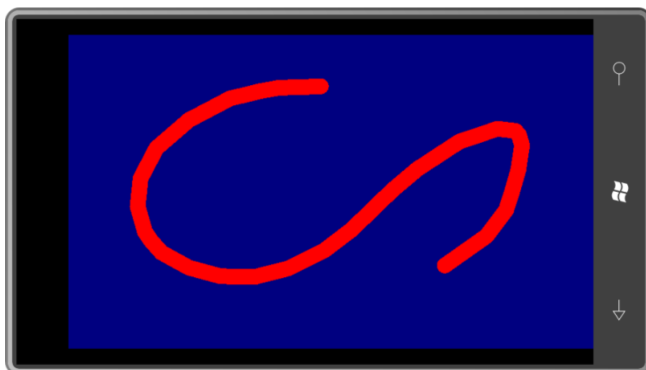
Иногда этот вызов необходим при вызове *SetData* из перегруженного *Update*, если определенный *Texture2D* отображался последним в методе *Draw* и по-прежнему задан в объекте *GraphicsDevice*.

Перегруженный *Draw* абсолютно тривиальный:

Проект XNA: BetterFingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(canvas, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

То что мы видим на экране, не может не радовать:



Отрисовываемые контуры сплошные, без разрывов. И обратите внимание на эти замечательные скругленные концы.

Приложение BetterFingerPaint не позволит рисовать двумя пальцами сразу. Обеспечить эту возможность с помощью жестов довольно тяжело. Одиночное касание формирует жесты

FreeDrag, тогда как касание в двух точках формирует жесты *Pinch*, которые включают действительные свойства *Position2* и *Delta2*. Но тогда приложение не сможет обрабатывать касание одновременно в трех точках.

Для обработки мультисенсорного ввода необходимо вернуться к интерфейсу простого касания, как показано в следующем проекте MultiFingerPaint (Рисование посредством мультисенсорного ввода). MultiFingerPaint в большей части идентичен BetterFingerPaint, но его конструктор *не* активирует жесты:

Проект XNA: MultiFingerPaint Файл: Game1.cs (фрагмент)

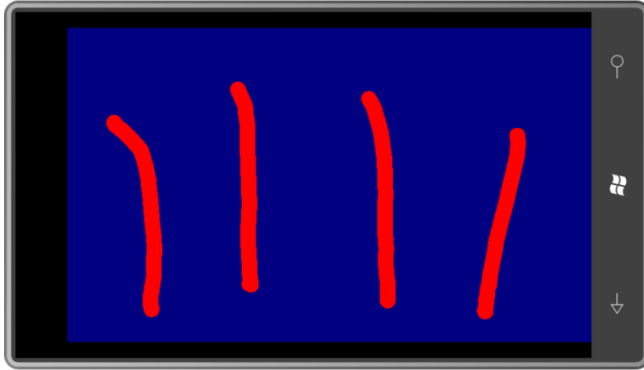
```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Частота кадров по умолчанию для Windows Phone - 30 кадров/с
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

Я продемонстрирую только часть перегруженного метода *Update*, потому что все остальное осталось неизменным. Данный метод, по сути, перебирает члены коллекции *TouchCollection*, полученной из вызова *TouchPanel.GetState* (Получить состояние). Эта коллекция включает объекты *TouchLocation* для множества касаний, перемещений по экрану и снятий касания, но наше приложение интересуют только перемещения. В данном случае даже не требуется отслеживать множество касаний. Необходимо лишь получить текущую точку касания, предыдущую точку этого же касания с помощью *TryGetPreviousLocation* (Попытаться получить предыдущее местоположение) и отрисовать линию между этими двумя точками:

Проект XNA: MultiFingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    bool canvasNeedsUpdate = false;
    int yMinUpdate = Int32.MaxValue, yMaxUpdate = 0;

    TouchCollection touches = TouchPanel.GetState();

    foreach (TouchLocation touch in touches)
    {
        if (touch.State == TouchLocationState.Moved)
        {
            TouchLocation previousTouch;
            touch.TryGetPreviousLocation(out previousTouch);

            Vector2 point1 = previousTouch.Position;
            Vector2 point2 = touch.Position;
            float radius = 12;

            RoundCappedLine line = new RoundCappedLine(point1, point2, radius);

    …
}
```

И вот что мы получаем, проводя по экрану одновременно четырьмя пальцами:



Изменение существующих изображений

Чтобы изменить существующее изображение, можно вызвать *GetData* «исходного» *Texture2D*, изменить полученные пикселы, применяя к ним некоторый алгоритм, и передать полученные значения пикселов в «результирующий» *Texture2D*, вызвав для него *SetData*. Это продемонстрировано в проекте RippleEffect (Эффект волны). Исходный *Texture2D* – это растровое изображение, которое я скопировал со своего сайта. Приложение изменяет его пикселы, создавая эффект прохождения по нему горизонтальных волн:

Исходный («src») и результирующий («dst») объекты *Texture2D*, а также соответствующие массивы пикселов сохраняются в полях приложения:

Проект XNA: RippleEffect Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    const int RIPPLE = 10;
    Texture2D srcTexture;
    Texture2D dstTexture;
    uint[] srcPixels;
    uint[] dstPixels;
    Vector2 position;

    …
}
```

Задается константа, которая показывает, что пикселы исходного растрового изображения будут перемещены вверх и вниз на 10 пикселов. Эта константа используется как в алгоритме,

обрабатывающем значения пикселов исходного изображения, так и для определения того, на сколько больше должно быть результирующие изображения по сравнению с исходным.

Метод *LoadContent* загружает *srcTexture* из содержимого приложения и копирует значения пикселов в массив *srcPixels*. Результирующий *dstTexture* на 20 пикселов выше исходного *srcTexture*. Для результирующих значений пикселов создан массив, но он до сих пор никак не использовался:

Проект XNA: RippleEffect   Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    srcTexture = this.Content.Load<Texture2D>("PetzoldTattoo");
    srcPixels = new uint[srcTexture.Width * srcTexture.Height];
    srcTexture.GetData<uint>(srcPixels);

    dstTexture = new Texture2D(this.GraphicsDevice,
                              srcTexture.Width,
                              srcTexture.Height + 2 * RIPPLE);
    dstPixels = new uint[dstTexture.Width * dstTexture.Height];

    Viewport viewport = this.GraphicsDevice.Viewport;
    position = new Vector2((viewport.Width - dstTexture.Width) / 2,
                           (viewport.Height - dstTexture.Height) / 2);
}
```

Целью метода *Update* является передача значений пикселов из *srcPixels* в *dstPixels* с использованием алгоритма, который включает анимацию. После этого массив *dstPixels* копируется в *dstTexture* с помощью метода *SetData*.

Для переноса значений пикселов из исходного изображения в результирующее может использоваться два разных подхода:

- Последовательный перебор всех строк и столбцов исходного изображения. Получение значений всех пикселов исходного изображения. Для каждого пиксела определение соответствующей строки и столбца в результирующем изображении и сохранение его значения.
- Последовательный перебор всех строк и столбцов результирующего изображения. Сопоставление строки и столбца с соответствующими строкой и столбцом исходного изображения, получение значения пиксела и сохранение его в результирующем изображении.

В общем случае второй подход несколько сложнее первого, но только он гарантирует задание каждого пиксела результирующего растрового изображения. Поэтому в циклах *for* следующего метода используются значения *xDst* and *yDst*, определяющие столбец и строку результирующего растрового изображения. На их основании вычисляются *xSrc* and *ySrc*. (В данном конкретном алгоритме *xSrc* всегда равен *xDst*.)

После этого два массива значений пикселов индексируются с использованием переменных *dstIndex* и *srcIndex*. Хотя *dstIndex* всегда будет действительным, потому что основывается на действительных значениях *xDst* и *yDst*, некоторые значения *srcIndex* могут оказаться недействительными. Пикселы, соответствующие недействительным *dstIndex*, я делаю прозрачными.

Проект XNA: RippleEffect Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    float phase =
        (MathHelper.TwoPi * (float)gameTime.TotalGameTime.TotalSeconds) %
                                                            MathHelper.TwoPi;

    for (int xDst = 0; xDst < dstTexture.Width; xDst++)
    {
        int xSrc = xDst;
        float angle = phase - xDst * MathHelper.TwoPi / 100;
        int offset = (int)(RIPPLE * Math.Sin(angle));

        for (int yDst = 0; yDst < dstTexture.Height; yDst++)
        {
            int dstIndex = yDst * dstTexture.Width + xDst;
            int ySrc = yDst - RIPPLE + offset;
            int srcIndex = ySrc * dstTexture.Width + xSrc;

            if (ySrc < 0 || ySrc >= srcTexture.Height)
                dstPixels[dstIndex] = Color.Transparent.PackedValue;
            else
                dstPixels[dstIndex] = srcPixels[srcIndex];
        }
    }
    this.GraphicsDevice.Textures[0] = null;
    dstTexture.SetData<uint>(dstPixels);

    base.Update(gameTime);
}
```

В этом перегруженном *Update* объект *srcTexture* используется исключительно для определения, не находится ли значение *yDst* ниже нижней строки растрового изображения. Несомненно, я мог бы сохранить это число строк и удалить само изображение *srcTexture*.

В результате выполнения перегруженного *Update* получаем *dstTexture*, обновленный значениями пикселов из массива *dstPixels*. Перегруженный *Draw* просто выводит на экран это изображение:

Проект XNA: RippleEffect Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(dstTexture, position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Данное приложение только меняет координаты пикселов, но подобные программы могут использоваться и для изменения значений цветов пикселов. Также можно вычислять значения пикселов на основании нескольких исходных изображений, создавая эффекты фильтрации.

Однако если значения пикселов вычисляются и передаются при каждом вызове *Update*, могут возникать проблемы с производительностью. И обработка значений каждого пиксела, и вызов *SetData* занимают значительное время. Первая версия данного приложения выполнялась нормально на эмуляторе телефона, но на реальном телефоне производительность падала до примерно двух обновлений в секунду. Я уменьшил размеры растрового изображения вдвое (т.е. сократил количество пикселов в 4 раза), чем обеспечил существенное улучшение производительности.

В следующей главе я покажу, как вычислять значения пикселов алгоритмически во втором потоке выполнения.

Глава 22
От жестов к трансформациям

Основным средством пользовательского ввода в приложении для Windows Phone 7 является касание. Устройство, работающее под управлением Windows Phone 7, имеет экран, который поддерживает минимум четыре точки касания. Распознавание и интерпретация касаний должны быть реализованы приложениями так, чтобы это было естественно и интуитивно понятно для пользователя.

Как уже было продемонстрировано, в распоряжении разработчиков на XNA имеется два основных подхода к обработке сенсорного ввода. Метод *TouchPanel.GetState* позволяет отслеживать простые одиночные касания, каждое из которых идентифицируется с помощью ID, от момента первого соприкосновения пальца с экраном, в ходе перемещения и вплоть до снятия касания. Метод *TouchPanel.ReadGesture* обеспечивает более высокоуровневый интерфейс для элементарной обработки инерции и касания двумя пальцами в форме жестов «pinch» (сведение) и «stretch» (растяжение).

Жесты и свойства

Жесты, поддерживаемые классом *TouchPanel*, соответствуют членам перечисления *GestureType*:

- *Tap* — быстрое касание и снятие
- *DoubleTap* — последовательность двух коротких касаний
- *Hold* — нажатие и удержание в течение одной секунды
- *FreeDrag* — перемещение пальца по экрану
- *HorizontalDrag* — горизонтальная составляющая FreeDrag
- *VerticalDrag* — вертикальная составляющая FreeDrag
- *DragComplete* —касание снято
- *Flick* — движение скольжения одного пальца по экрану
- *Pinch* — перемещение двух пальцев навстречу друг другу или в противоположные стороны
- *PinchComplete* — пальцы сняты с экрана

Для работы с жестами их необходимо активировать посредством свойства *TouchPanel.EnabledGestures* (Жесты активированы). После этого приложение принимает жесты в ходе выполнения перегруженного *Update* класса *Game* в форме структур *GestureSample*, которые включают свойство *GestureType* для идентификации жеста.

GestureSample также определяет четыре свойства типа *Vector2*. Ни одно из этих свойств не действительно для жестов *DragComplete* и *PinchComplete*. Также:

- *Position* действительно для всех жестов, кроме *Flick*.
- *Delta* действительно для всех жестов *Drag*, *Pinch* и *Flick*.
- *Position2* и *Delta2* действительны только для *Pinch*.

Свойство *Position* указывает на текущее положение точки касания относительно экрана. Свойство *Delta* соответствует перемещению точки касания относительно последнего местоположения. Для объекта типа *GestureSample* под именем *gestureSample*:

```
Vector2 previousPosition = gestureSample.Position - gestureSample.Delta;
```

Вектор *Delta* равен нулю в первый момент касания, или если точка касания остается неподвижной.

Предположим, нас интересуют только операции перетягивания. Для этого активируем жесты *FreeDrag* и *DragComplete*. Если требуется отслеживать весь путь пальца по экрану с момента первого касания до момента снятия, воспользуемся одной из двух стратегий: либо будем сохранять значение *Position* первого *FreeDrag* после *DragComplete* и сравнивать его с последующими значениями *Position*, либо будем накапливать значения *Delta* в виде промежуточной суммы.

Рассмотрим простое приложение, в котором пользователь может перемещать небольшое растровое изображение по экрану. В проекте OneFingerDrag (Перетягивание одним пальцем) класс *Game1* имеет поля для хранения *Texture2D* и его местоположения:

Проект XNA: OneFingerDrag Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Vector2 texturePosition = Vector2.Zero;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        TouchPanel.EnabledGestures = GestureType.FreeDrag;
    }
    …
}
```

Обратите внимание, жест *FreeDrag* активируется в конце конструктора.

Перегруженный *LoadContent* загружает тот же *Texture2D*, который мы использовали в проекте RippleEffect в предыдущей главе:

Проект XNA: OneFingerDrag Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
}
```

Перегруженный метод *Update* обрабатывает жест *FreeDrag*, просто увеличивая вектор *texturePosition* на значение свойства *Delta* объекта *GestureSample*:

Проект XNA: OneFingerDrag Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag)
            texturePosition += gesture.Delta;
    }
    base.Update(gameTime);
}
```

И хотя *texturePosition* – это точка, а значение свойства *Delta* объекта *GestureSample* – это вектор, они оба являются значениями *Vector2*, поэтому могут складываться.

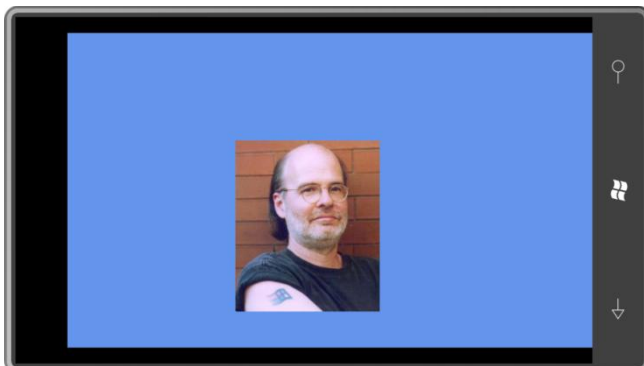
Применение цикла *while* в данном приложении может показаться нецелесообразным, потому что нас интересует лишь один тип жестов. Разве нельзя было бы просто использовать выражение *if*? На самом деле, нет. Мой опыт показывает, что в ходе одного вызова *Update* могут быть доступны несколько жестов одного типа.

Перегруженный *Draw* просто отрисовывает *Texture2D* в точке, соответствующей новому местоположению:

Проект XNA: OneFingerDrag Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(texture, texturePosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Изначально *Texture2D* располагается в верхнем левом углу экрана, но пользователь может перемещать его, проводя пальцем по экрану:



Пользователь может перемещать палец по экрану *абсолютно* произвольно, и текстура будет перемещаться за ним! Данное приложение не проводит проверки попадания точки касания в область *Texture2D*, но это несложно добавить:

```
while (TouchPanel.IsGestureAvailable)
{
    GestureSample gesture = TouchPanel.ReadGesture();

    if (gesture.GestureType == GestureType.FreeDrag)
    {
        if (gesture.Position.X > texturePosition.X &&
            gesture.Position.X < texturePosition.X + texture.Width &&
            gesture.Position.Y > texturePosition.Y &&
            gesture.Position.Y < texturePosition.Y + texture.Height)
        {
            texturePosition += gesture.Delta;
        }
    }
}
```

Как же работает данная логика? При проведении пальцем по экрану вне текстуры она перемещаться не будет, но если в какой-то момент точка касания попадает в область текстуры, последняя начинает перемещаться. Вероятно, желательно сделать так, чтобы текстура перемещалась только в том случае, когда первый *FreeDrag* в последовательности приходится на область текстуры. В противном случае вся последовательность жестов *FreeDrag* вплоть до *DragComplete* должна игнорироваться.

Масштабирование и вращение

Продолжим работу с жестами перетягивания с участием простых геометрических фигур, но будем использовать эти жесты не для перемещения, а для масштабирования и вращения. Для следующих трех приложений помещаем *Texture2D* в центр экрана, и он будет оставаться в центре всегда, пользователь сможет только изменять его размер или вращать посредством касания одним пальцем.

Проект OneFingerScale (Масштабирование одним пальцем) включает на пару полей больше, чем предыдущее приложение:

Проект XNA: OneFingerScale Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Vector2 screenCenter;
    Vector2 textureCenter;
    Vector2 textureScale = Vector2.One;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        TouchPanel.EnabledGestures = GestureType.FreeDrag;
    }
```

```
    …
}
```

Данному приложению необходимо знать координаты центра *Texture2D*, потому что в нем используется развернутая версия вызова метода *Draw* объекта *SpriteBatch*, включающая центр как аргумент. Как помните, аргумент *origin* метода *Draw* – это точка объекта *Texture2D*, соответствующая значению аргумента *position* и используемая как центр масштабирования и вращения.

Обратите внимание, что в качестве значения поля *textureScale* задан вектор **(1, 1)**, что обеспечивает умножение ширины и высоты на 1. Общей ошибкой является задание масштабу нулевого значения, что приводит к полному исчезновению объектов с экрана.

Все инициализированные поля задаются в перегруженном *LoadContent*:

Проект XNA: OneFingerScale Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Viewport viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);

    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
    textureCenter = new Vector2(texture.Width / 2, texture.Height / 2);
}
```

При обработке жеста *FreeDrag* в следующем перегруженном *Update* не делается попытки определить, действительно ли точка касания попадает в область растрового изображения. Растровое изображение располагается в центре экрана и будет масштабироваться в различной степени, поэтому такое вычисление было бы несколько сложным (хотя, безусловно, не невозможным).

Вместо этого перегруженный *Update* демонстрирует использование свойства *Delta* для определения координат предыдущей точки касания. Впоследствии эти данные используются для определения того, насколько далеко точка касания переместилась от центра текстуры (который также является центром экрана) в ходе этого конкретного отрезка жеста:

Проект XNA: OneFingerScale Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag)
        {
            Vector2 prevPosition = gesture.Position - gesture.Delta;

            float scaleX = (gesture.Position.X - screenCenter.X) /
                                (prevPosition.X - screenCenter.X);
            float scaleY = (gesture.Position.Y - screenCenter.Y) /
```

```
                                (prevPosition.Y - screenCenter.Y);

            textureScale.X *= scaleX;
            textureScale.Y *= scaleY;
        }
    }
    base.Update(gameTime);
}
```

Например, центром экрана является точка (400, 240). Предположим, в ходе этого конкретного отрезка жеста свойство *Position* имеет значение (600, 200), и свойство *Delta* – (20, 10). Это означает, что предыдущее местоположение соответствовало точке (580, 190). В горизонтальном направлении расстояние от точки касания до центра увеличилось от 180 пикселов (580 минус 400) до 200 пикселов (600 минус 400); чтобы найти коэффициент масштабирования, делим 200 на 180, получаем 1,11. По вертикали расстояние от центра уменьшилось с 50 пикселов (240 минус 190) до 40 пикселов (240 минус 200); чтобы найти коэффициент масштабирования делим 40 на 50, получаем 0,80. Размер изображения увеличивается на 11% в горизонтальном направлении и на 20% в вертикальном.

Таким образом, умножаем составляющую X вектора масштабирования на 1,11 и составляющую Y – на 0,80. Как и ожидалось, коэффициент масштабирования применяется в перегруженном *Draw*:

Проект XNA: OneFingerScale Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(texture, screenCenter, null, Color.White, 0,
                     textureCenter, textureScale, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Наиболее очевидного эффекта можно добиться, если «взять» изображение за один из углов и перенести этот угол в направлении прямо к центру или от него:

Как видите, ничто не мешает изображению изменить пропорции, что подразумевают приведенные выше вычисления. Проводя пальцами дальше вертикальной или горизонтальной оси экрана, можно даже сжать изображение в точку!

В реальном приложении желательно задать минимальный коэффициент масштабирования, например 0,1 или 0,25, просто чтобы обеспечить пользователю возможность «растянуть» изображение снова.

Также скорее всего в некоторых приложениях потребуется сохранять пропорции изображения. В этом случае для масштабирования в горизонтальном и вертикальном направлениях должен использоваться один коэффициент масштабирования. Можно вычислять два коэффициента масштабирования, как в приложении OneFingerScale, и затем просто находить их среднее. Но это, безусловно, неправильно. Если пользователь знает, что приложение обеспечивает сохранение пропорций, он ожидает соответствующего поведения от изображения и будет масштабировать его, просто перетягивая в горизонтальном или вертикальном направлении.

Можно вычислять оба коэффициента масштабирования и брать наибольший из них. Но это также не вполне правильно. При выполнении OneFingerScale можно заметить, что когда точка касания располагается слишком близко к центру изображения, всего лишь небольшого движения достаточно, чтобы существенно изменить размеры изображения. Если точка касания располагается вблизи горизонтальной оси симметрии, но далеко от вертикальной оси, для равного перемещения точки касания в горизонтальном и вертикальном направлении коэффициенты масштабирования будут разными.

Вероятно, наилучшей стратегией будет проверять значение свойства *Delta* и определять, какая из составляющих, X или Y, имеет наибольшее значение (не учитывая знака), и затем использовать ее для вычисления коэффициента масштабирования. Такая методика реализована в проекте OneFingerUniformScale (Унифицированное масштабирование одним пальцем).

Поля такие же, как и в предыдущем приложении, за исключением того, что коэффициент масштабирования *Vector2* заменен на *float*.

Проект XNA: OneFingerUniformScale Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Vector2 screenCenter;
    Vector2 textureCenter;
    float textureScale = 1;
    …
}
```

Перегруженный *LoadContent* абсолютно такой же, как и в предыдущей версии, а вот обработка жеста в перегруженном *Update* стала более развернутой. Метод проверяет, абсолютное значение какой из составляющих вектора *Delta*, горизонтальной или вертикальной, больше. Также он не выполняет никаких вычислений, если обе составляющие равны нулю, что имеет место в начале жеста, когда палец впервые касается экрана.

Проект XNA: OneFingerUniformScale Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
```

```
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag)
        {
            Vector2 prevPosition = gesture.Position - gesture.Delta;
            float scale = 1;

            if (Math.Abs(gesture.Delta.X) > Math.Abs(gesture.Delta.Y))
            {
                scale = (gesture.Position.X - screenCenter.X) /
                                (prevPosition.X - screenCenter.X);
            }
            else if (gesture.Delta.Y != 0)
            {
                scale = (gesture.Position.Y - screenCenter.Y) /
                                (prevPosition.Y - screenCenter.Y);
            }

            if (!float.IsInfinity(scale) && !float.IsNaN(scale))
            {
                textureScale = Math.Min(10,
                                   Math.Max(0.25f, scale * textureScale));
            }
        }
    }
    base.Update(gameTime);
}
```

Здесь реализована еще одна мера предосторожности: проверка на то, не является ли вычисленное значение бесконечным или является ли оно числом. Это возможно, если точка касания точно совпадает с центром экрана, что приводит к делению на нуль. Также я ограничил общий коэффициент масштабирования диапазоном значений от 0,25 до 10. Значения выбраны довольно произвольно, но позволяют продемонстрировать эту важную концепцию.

Перегруженный метод *Draw* аналогичен используемому в предыдущем приложении, за исключением того что *textureScale* (Масштаб текстуры) типа *float*, а не *Vector2*:

Проект XNA: OneFingerUniformScale Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(texture, screenCenter, null, Color.White, 0,
                     textureCenter, textureScale, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Я пришел к осознанию необходимости задания допустимого максимального значения *textureScale*, поэкспериментировав немного с более ранней версией этого приложения. Я коснулся изображения очень близко к центру экрана, и небольшое движение привело к тому, что оно было увеличено в несколько сотен раз, так что на экране поместились лишь несколько его пикселов! Излишне говорить, что в таком масштабировании нет смысла.

Приложение можно реализовать так, чтобы оно игнорировало жесты вблизи определенной опорной точки. Я делаю это в следующем проекте.

Масштабирование посредством касания одним пальцем используется довольно редко, а вот организация вращения таким образом – очень мощная возможность, широко применяемая и в мире компьютеров, и в реальной жизни. Если на столе перед вами лежит телефон, коснитесь пальцем одного из его углов и потяните телефон к себе. Скорее всего, телефон немного повернется, а уже потом начнет перемещение в вашем направлении.

Очень часто вращение посредством касания одним пальцем используется в сочетании с обычным перетягиванием. Рассмотрим, как это реализовывается. Поля OneFingerRotation (Вращение одним пальцем) подобны используемым в предыдущих приложениях:

Проект XNA: OneFingerRotation Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Vector2 texturePosition;
    Vector2 textureCenter;
    float textureRotation;
    …
}
```

Перегруженный *LoadContent* также аналогичен. Поле *texturePosition* (Местоположение текстуры) инициализируется координатами центра экрана, но при перемещении текстуры по экрану его значение будет меняться:

Проект XNA: OneFingerRotation Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Viewport viewport = this.GraphicsDevice.Viewport;
    texturePosition = new Vector2(viewport.Width / 2, viewport.Height / 2);

    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
    textureCenter = new Vector2(texture.Width / 2, texture.Height / 2);
}
```

В методе *Update* решено сначала сравнивать координаты предыдущего касания и нового касания с координатами центра *Texture2D*, обозначенными в *texturePosition*. В *Update* я представляю эти два местоположения как векторы из центра текстуры в эти точки касания: *oldVector* (Старый вектор) и *newVector* (Новый вектор) (под «старый» и «новый» я подразумеваю «предыдущее» и «текущее» местоположение). Если эти два вектора располагаются под разными углами, угол *textureRotation* (Разворот текстуры) меняется на разницу между ними.

Теперь удалим компонент вращения из изменения этих координат. Все что осталось, должно использоваться для перетягивания текстуры. *oldVector* пересчитывается так, что он сохраняет исходный модуль, но теперь указывает в том же направлении, что и *newVector*. Вычисляется

новая разность между текущими *newVector* и *oldVector*, значение которой и используется для реализации перетягивания:

Проект XNA: OneFingerRotation Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.FreeDrag)
        {
            Vector2 delta = gesture.Delta;
            Vector2 newPosition = gesture.Position;
            Vector2 oldPosition = newPosition - delta;

            // Находим векторы, направленные из центра растрового изображения
            // в точки касания
            Vector2 oldVector = oldPosition - texturePosition;
            Vector2 newVector = newPosition - texturePosition;

            // Отменяем вращение, если точка касания располагается рядом с центром
            if (newVector.Length() > 25 && oldVector.Length() > 25)
            {
                // Находим углы для векторов, проведенных из центра
                // растрового изображения в точки касания
                float oldAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
                float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

                // Корректируем угол поворота текстуры
                textureRotation += newAngle - oldAngle;

                // По сути, вращаем старый вектор
                oldVector = oldVector.Length() / newVector.Length() * newVector;

                // Повторно вычисляем разницу
                delta = newVector - oldVector;
            }
            // Перемещаем текстуру
            texturePosition += delta;
        }
    }
    base.Update(gameTime);
}
```

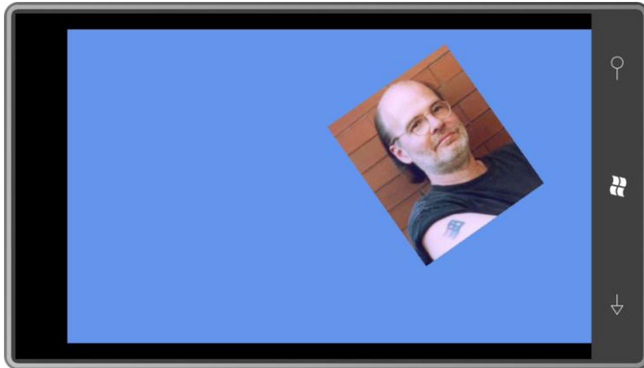
Перегруженный *Draw* использует этот угол поворота, но задает коэффициент масштабирования равным 1:

Проект XNA: OneFingerRotation Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(texture, texturePosition, null, Color.White,
                     textureRotation, textureCenter, 1, SpriteEffects.None, 0);
    spriteBatch.End();
```

```
    base.Draw(gameTime);
}
```

Поэкспериментировав с этим приложением, можно заметить, что перемещения выглядят очень естественными. Изображение можно «схватить» в любой точке и «потянуть». При этом создается эффект, как будто изображение «буксируется» вашим пальцем, точно как это происходит с телефоном на столе.



Конечно, самой распространенной формой масштабирования в приложениях, поддерживающих мультисенсорный ввод, является масштабирование с участием двух пальцев жестами сведения или растяжения. Реализовать это не на много сложнее, чем то что мы уже видели. Мы уже выполняли растяжение и вращение относительно опорной точки. С участием двух пальцев все то же самое, за исключением того что опорной точкой всегда является точка касания второго пальца.

К примеру, коснитесь объекта на экране двумя пальцами. Если один палец остается неподвижным, а другой перемещается, следует ожидать, что все масштабирование и вращение будет выполняться относительно первого, неподвижного, пальца. Если этот первый палец также перемещается, все обусловливаемое им масштабирование и вращение выполняется относительно второго пальца. Если оба пальца перемещаются в одном направлении, выполняется обычное перетягивание.

Участвующие в реализации этого математические вычисления довольно сложны, но, к счастью, на помощь приходят некоторые очень мощные инструменты, предоставляемые XNA.

Трансформации матриц

Традиционно двухмерные графические системы поддерживают операции, называемые *трансформациями*. По сути своей, это математические формулы, применяемые к координатам (*x*, *y*) для получения новых координат (*x*', *y*'). Полностью обобщённые трансформации потенциально могут быть очень сложными, но двухмерные среды разработки графических элементов часто ограничиваются подмножеством трансформаций, называемым *аффинными* («не бесконечными») преобразованиями. К таким преобразованиям относятся и *линейные* трансформации.

Линейные преобразования для переменных *x* и *y* выглядят следующим образом:

$$x' = a_x x + b_x y$$

$$y' = a_y x + b_y y$$

где индексированные *a* и *b* – это константы, определяющие конкретное преобразование. Как видите, *x′* и *y′* являются функциями *x* и *y*, и это очень простые функции. В них переменные всего лишь умножаются на константы, и результаты этого умножения суммируются; *x* и *y* даже не перемножаются, к примеру.

В реализации аффинного переноса добавляется еще одна константа, которая ни на что не умножается:

$$x' = a_x x + b_x y + c_x$$

$$y' = a_y x + b_y y + c_y$$

Очень часто некоторые из этих констант равны нулю. Если a_x, и b_y равны 1, и b_x и a_y равны нулю, формулы представляют тип преобразования, называемый *переносом*:

$$x' = x + c_x$$

$$y' = y + c_y$$

Это преобразование просто обеспечивает смещение объекта в другое место, наподобие того как это было в приложении OneFingerDrag в начале этой главы.

Если b_x и a_y равны нулю, и c_x и c_y тоже равны нулю, тогда мы получаем формулы для *масштабирования*:

$$x' = a_x x$$

$$y' = b_y y$$

Координаты умножаются на коэффициенты, что обусловливает увеличение или уменьшение размеров объектов.

Четыре множителя можно заменить синусом и косинусом определенного угла:

$$x' = \cos(\propto)x - \sin(\propto)y$$

$$y' = \sin(\propto)x + \cos(\propto)y$$

Эти уравнения описывают вращение точки вокруг заданного центра на α градусов. Если всем четырем константам задать значения, не описываемые тригонометрическими функциями, уравнения будут описывать трансформацию *наклонением*, которая превращает квадрат в параллелограмм. Но это настолько же странно, насколько странные сами аффинные преобразования: они никогда не сделают прямую кривой или параллельные линии не параллельными. В результате *аффинных* преобразований координаты никогда не станут бесконечными.

Перенос, масштабирование и вращение – это самые распространенные типы трансформаций. Их можно комбинировать. Для упрощения вычислений трансформации часто представляют в виде матриц 3×3:

$$\begin{vmatrix} a_x & a_y & 0 \\ b_x & b_y & 0 \\ c_x & c_y & 1 \end{vmatrix}$$

Для применения трансформации к точке с координатами (*x*, *y*) точку представляют как матрицу 1×3 с 1 в качестве третьего члена и выполняют умножение матриц:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \cdot \begin{vmatrix} a_x & a_y & 0 \\ b_x & b_y & 0 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & 1 \end{vmatrix}$$

Элементы главной диагонали единичной матрицы равны 1, умножение на нее не приводит к трансформации:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Существенное преимущество представления трансформаций с помощью матриц ощущается при комбинировании трансформаций, что эквивалентно умножению матриц, а эта операция всем хорошо известна. Также общеизвестно, что умножение матриц является некоммутативной операцией, т.е. порядок умножения имеет значение.

Например, необходимо применить к объекту трансформацию масштабирования, чтобы сделать его больше, и затем подвергнуть его трансформации перемещения, чтобы передвинуть в другое место. Если поменять местами эти две операции, т.е. сначала переместить объект и затем изменить его размер, результат будет иным, потому что масштабирование будет применяться и к коэффициентам переноса.

Трансформации в двухмерном пространстве описываются матрицами 3×3, и трансформации в трехмерном пространстве описываются матрицами 4×4. Имеется очень веское основание тому, почему матрица должна иметь на одно измерение больше, чем описываемое координатное пространство – перемещение. Перемещение является очень базовой и широко применяемой трансформацией, но оно не может быть описано линейным преобразованием, в котором координаты *x* и *y* просто умножаются на некоторые коэффициенты. Для представления перемещения как линейного преобразования должно быть добавлено еще одно измерение. Перемещение в двухмерном пространстве – это сдвиг в трехмерном, и поэтому двухмерная точка преобразовывается в трехмерную добавлением единичного значения Z для умножения на матрицу.

Третий столбец матрицы двухмерного аффинного преобразования всегда включает два нуля и 1 в нижнем правом углу. Это то что делает данное преобразование аффинным. (Неаффинные двухмерные преобразования обсудим в конце данной главы.)

После такого длинного вступления удивительно узнать, что XNA в отличие от практически всех остальных графических сред программирования *не* поддерживает структуру, которая инкапсулирует матрицу трансформации 3×3. В XNA матрицы обычно используются для трехмерной графики, поэтому XNA-структура *Matrix* (Матрица) инкапсулирует матрицу 4×4, которая подходит для трехмерной графики, но несколько избыточна для описания двухмерных объектов.

Хотя структура *Matrix* может использоваться с двухмерной графикой – и ее очень удобно применять для составных трансформаций – двухмерный XNA не обеспечивает особой поддержки трансформаций, кроме существенно более развернутой версии вызова метода *Begin* класса *SpriteBatch*:

```
spriteBatch.Begin(SpriteSortMode.Deferred, null, null, null, null, null, matrix);
```

При использовании данной формы вызова *Begin* объект *Matrix* будет оказывать влияние на все вызовы *Draw* и *DrawString* до тех пор, пока не будет вызван *End*. Это может быть очень полезным для применения трансформации к группе графических объектов.

Трансформации к объектам *Vector2* могут также применяться «вручную» посредством нескольких версий статического метода *Vector2.Transform*.

Структура *Matrix* поддерживает очень много статических методов для создания объектов *Matrix*, представляющих различные типы трансформаций. Все они созданы для трехмерной

графики. Рассмотрим, как можно использовать наиболее базовые из них для двухмерной графики:

```
Matrix matrix = Matrix.CreateTranslation(xOffset, yOffset, 0);

Matrix matrix = Matrix.CreateScale(xScale, yScale, 1);

Matrix matrix = Matrix.CreateRotationZ(radians);
```

В качестве последнего аргумента первых двух методов обычно используются коэффициенты переноса или масштабирования для оси Z трехмерного координатного пространства. Обратите внимание, во втором случае я задал третий аргумент равным 1, а не 0. 0 подходит для большинства целей, но если когда-либо вам понадобится инвертировать матрицу, нулевой коэффициент масштабирования испортит все. Также заметьте, что в имени третьего метода упоминается ось Z. Этот метод должен вычислять угол поворота вокруг оси Z для описания вращения в двухмерной плоскости XY.

Структура *Matrix* поддерживает арифметические операторы, поэтому матрицы могут без труда перемножаться для реализации составных трансформаций. Чаще всего умножение матриц используется для представления масштабирования или вращения вокруг определенной точки.

Предположим, у нас имеется точка, представленная как объект *Vector2* под именем *center*. Требуется вычислить матрицу для описания вращения вокруг этой точки на *angle* градусов. Начинать следует с перемещения точки *center* в начало координат, затем применяется вращение (или масштабирование) и еще одно перемещение для возвращения *center* в исходное положение:

```
Matrix matrix = Matrix.CreateTranslation(-center.X, -center.Y, 0);
matrix *= Matrix.CreateRotationZ(angle);
matrix *= Matrix.CreateTranslation(center.X, center.Y, 0);
```

Обратите внимание на операторы умножения.

Структура *Matrix* в XNA включает 16 открытых полей типа *float*, которые представляют все 16 ячеек матрицы 4×4. Эти поля именованы соответственно строке и столбцу соответствующей ячейки в матрице:

$$\begin{vmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ M41 & M42 & M43 & M44 \end{vmatrix}$$

Вместо того, чтобы создавать объекты *Matrix* с помощью статических методов структуры *Matrix*, все эти поля можно задать по отдельности (или задать их все в конструкторе с 16 аргументами). Используемые нами ранее индексированные константы соответствуют этим ячейкам:

$$\begin{vmatrix} a_x & a_y & 0 & 0 \\ b_x & b_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ c_x & c_y & 0 & 1 \end{vmatrix}$$

Поле *M11* – это горизонтальный коэффициент масштабирования, и *M22* – вертикальный; *M41* – горизонтальный коэффициент переноса, и *M42* – вертикальный. Уравнения двухмерных аффинных преобразований можно переписать с использованием имен полей структуры *Matrix*:

$$x' = M11 \cdot x + M21 \cdot y + M41$$

$$y' = M12 \cdot x + M22 \cdot y + M42$$

Знание взаимоотношений между этими полями и трансформациями может помочь в извлечении данных из структуры *Matrix* или создании сокращенных реализаций, которые не включают создания новых объектов *Matrix* и их перемножения. Некоторые из этих методик будут продемонстрированы далее в этой главе.

Жест *Pinch*

Для жеста *Pinch* действительны все четыре свойства *GestureSample* типа *Vector2*: *Position*, *Delta*, *Position2* и *Delta2*. Первые два описывают местоположение и перемещение одного пальца; остальные два представляют второй палец. Это идеально для масштабирования, хотя, вероятно, математика не сразу понятна.

Как правило, желательно поддерживать и *FreeDrag*, и *Pinch*, чтобы пользователь мог применять один или два пальца. После этого требуется принять решение о необходимости поддержки вращения и применении пропорционального или непропорционального масштабирования.

Приложение DragAndPinch (Перетягивание и сведение) обрабатывает оба жеста, *FreeDrag* и *Pinch*, с применением непропорционального масштабирования без вращения. Как обычно, эти жесты активируются в конструкторе. Новое поле, которое мы видим здесь, это объект *Matrix*, инициализированный как единичная матрица черех статическое свойство *Matrix.Identity*:

Проект XNA: DragAndPinch Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Matrix matrix = Matrix.Identity;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        TouchPanel.EnabledGestures = GestureType.FreeDrag | GestureType.Pinch;
    }
    …
}
```

Выражение

```
Matrix matrix = Matrix.Identity;
```

не аналогично выражению:

```
Matrix matrix = new Matrix();
```

Matrix – это структура, и как для всех структур, ее поля инициализируются нулевыми значениями. Объект *Matrix* со всеми нулями не годится ни для чего, поскольку он полностью уничтожает все, к чему применяется. В используемом по умолчанию объекте *Matrix* все

диагональные элементы должны быть заданы равными 1. Именно это обеспечивает свойство *Matrix.Identity*.

Все операции перетягивания и сведения будут применяться к полю *matrix*, которое затем используется в перегруженном *Draw*.

Метод *LoadContent* просто загружает *Texture2D*:

Проект XNA: DragAndPinch Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
     // Создаем новый SpriteBatch, который может использоваться для отрисовки
текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
}
```

Перегруженный *Update* обрабатывает жесты *FreeDrag* и *Pinch*:

Проект XNA: DragAndPinch Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch (gesture.GestureType)
        {
            case GestureType.FreeDrag:
                matrix *= Matrix.CreateTranslation(gesture.Delta.X, gesture.Delta.Y,
0);
                break;

            case GestureType.Pinch:
                Vector2 oldPoint1 = gesture.Position - gesture.Delta;
                Vector2 newPoint1 = gesture.Position;
                Vector2 oldPoint2 = gesture.Position2 - gesture.Delta2;
                Vector2 newPoint2 = gesture.Position2;

                matrix *= ComputeScaleMatrix(oldPoint1, oldPoint2, newPoint2);
                matrix *= ComputeScaleMatrix(newPoint2, oldPoint1, newPoint1);
                break;
        }
    }
    base.Update(gameTime);
}
```

Обратите внимание, что для *FreeDrag* этот метод создает новый *Matrix* из статического метода *Matrix.CreateTranslation* и умножает его на существующее поле *matrix*. Это выражение можно заменить следующими:

```
matrix.M41 += gesture.Delta.X;
matrix.M42 += gesture.Delta.Y;
```

Для жеста *Pinch* метод *Update* разбивает данные на «старые» точки и «новые» точки. Если оба пальца перемещаются относительно друг друга, можно вычислить составной коэффициент масштабирования, рассматривая касания от двух разных пальцев по отдельности. Предположим, первый палец фиксирован в точке, определённой свойством *Position*, и второй перемещается относительно него; а затем второй палец фиксирован в точке *Position*, и первый перемещается относительно него. Каждый из сценариев представляет отдельную операцию масштабирования, которые потому перемножаются. В каждом случае имеется опорная точка (фиксированный палец), старая и новая точка (перемещающийся палец).

Чтобы сделать все правильно, опорная точка первой операции масштабирования должна соответствовать *старым* координатам фиксированного пальца, но для второго коэффициента масштабирования опорной точкой должны быть *новое* местоположение фиксированного пальца. В этом причина несколько ассиметричных вызовов метода *ComputeScaleMatrix* (Вычисление матрицы масштабирования), показанных выше. Рассмотрим сам метод:

Проект XNA: DragAndPinch   Файл: Game1.cs (фрагмент)

```
Matrix ComputeScaleMatrix(Vector2 refPoint, Vector2 oldPoint, Vector2 newPoint)
{
    float scaleX = (newPoint.X - refPoint.X) / (oldPoint.X - refPoint.X);
    float scaleY = (newPoint.Y - refPoint.Y) / (oldPoint.Y - refPoint.Y);

    if (float.IsNaN(scaleX) || float.IsInfinity(scaleX) ||
        float.IsNaN(scaleY) || float.IsInfinity(scaleY) ||
        scaleX <= 0 || scaleY <= 0)
    {
        return Matrix.Identity;
    }

    scaleX = Math.Min(1.1f, Math.Max(0.9f, scaleX));
    scaleY = Math.Min(1.1f, Math.Max(0.9f, scaleY));

    Matrix matrix = Matrix.CreateTranslation(-refPoint.X, -refPoint.Y, 0);
    matrix *= Matrix.CreateScale(scaleX, scaleY, 1);
    matrix *= Matrix.CreateTranslation(refPoint.X, refPoint.Y, 0);

    return matrix;
}
```

Эта опорная точка всегда выполняет здесь двоякую роль: она используется для измерения изменения координат перемещающейся точки касания и также для разграничения вызовов *Matrix.CreateScale* (Создать масштабирование для матрицы) в конце описания структуры для задания масштабирования относительно центральной точки. Эти три вызова в конце можно заменить следующим образом:

```
Matrix matrix = Matrix.Identity;
matrix.M41 -= refPoint.X;
matrix.M42 -= refPoint.Y;

matrix *= Matrix.CreateScale(scaleX, scaleY, 1);

matrix.M41 += refPoint.X;
matrix.M42 += refPoint.Y;
```

Суммарная составная матрица просто передается как последний аргумент вызова метода *Begin* объекта *spriteBatch* в перегруженном *Draw*:

Проект XNA: DragAndPinch Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin(SpriteSortMode.Deferred, null, null, null, null, null,
matrix);
    spriteBatch.Draw(texture, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Если вы предпочитаете использовать более простую форму вызова *Begin*, можно извлечь из объекта *Matrix* данные о масштабировании и местоположении и использовать их в вызове *Draw*:

```
Vector2 scale = new Vector2(matrix.M11, matrix.M22);
Vector2 position = new Vector2(matrix.M41, matrix.M42);

spriteBatch.Begin();
spriteBatch.Draw(texture, position, null, Color.White, 0,
                 Vector2.Zero, scale, SpriteEffects.None, 0);
spriteBatch.End();
```

Структура *Matrix* также поддерживает метод *Decompose* (Разложить), который обеспечивает извлечение составляющих масштабирования, вращения и перемещения. Составляющая вращения представлена в форме *Quaternion* (Кватернион). Это очень широко применяемый инструмент для объемного вращения, но никогда (по моим сведениям) не используемый в двухмерной графике. Заменим вычисления *scale* и *position* следующим:

```
Vector3 scale3;
Quaternion quaternion;
Vector3 translation3;

matrix.Decompose(out scale3, out quaternion, out translation3);

Vector2 scale = new Vector2(scale3.X, scale3.Y);
Vector2 position = new Vector2(translation3.X, translation3.Y);
```

Добавим в DragAndPinch поддержку вращения посредством одного и двух пальцев и назовем это приложение DragPinchRotate. Все осталось неизменным, за исключением перегруженного *Update*.

Проект XNA: DragPinchRotate Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch (gesture.GestureType)
        {
            case GestureType.FreeDrag:
                Vector2 newPoint = gesture.Position;
                Vector2 oldPoint = newPoint - gesture.Delta;
                Vector2 textureCenter = new Vector2(texture.Width / 2,
```

```
texture.Height / 2);
                Vector2 refPoint = Vector2.Transform(textureCenter, matrix);

                matrix *= ComputeRotateAndTranslateMatrix(refPoint, oldPoint,
newPoint);
                break;

            case GestureType.Pinch:
                Vector2 oldPoint1 = gesture.Position - gesture.Delta;
                Vector2 newPoint1 = gesture.Position;
                Vector2 oldPoint2 = gesture.Position2 - gesture.Delta2;
                Vector2 newPoint2 = gesture.Position2;

                matrix *= ComputeScaleAndRotateMatrix(oldPoint1, oldPoint2,
newPoint2);
                matrix *= ComputeScaleAndRotateMatrix(newPoint2, oldPoint1,
newPoint1);
                break;
        }
    }
    base.Update(gameTime);
}
```

В приложении, демонстрирующем вращение посредством касания одним пальцем, которое рассматривалось ранее, позиционирование *Texture2D* всегда осуществлялось относительно его центра, поэтому опорную точку вращения всегда было легко найти. Теперь позиционирование *Texture2D* осуществляется в ходе вызова метода *Draw* объекта *SpriteBatch* в верхнем левом углу экрана, но его фактическое местоположение определяется на основании объекта *Matrix*.

Поэтому в логике *FreeDrag* центр *Texture2D* выражается относительно верхнего левого угла как значение *Vector2*. Затем посредством трансформации текущей матрицы получаем координаты *refPoint* (Опорная точка) относительно экрана.

Логика метода *ComputeRotateAndTranslateMatrix* (Вычислить матрицу вращения и перемещения), вызываемого в методе *Update* для жеста *FreeDrag*, очень похожа на логику вращения посредством касания одним пальцем, за исключением того что трансформации извлекаются и перемножаются:

Проект XNA: DragPinchRotate Файл: Game1.cs (фрагмент)

```
Matrix ComputeRotateAndTranslateMatrix(Vector2 refPoint, Vector2 oldPoint, Vector2
newPoint)
{
    Matrix matrix = Matrix.Identity;
    Vector2 delta = newPoint - oldPoint;
    Vector2 oldVector = oldPoint - refPoint;
    Vector2 newVector = newPoint - refPoint;

    // Отменяем вращение, если точка касания располагается рядом с центром
    if (newVector.Length() > 25 && oldVector.Length() > 25)
    {
        // Находим углы для векторов, проведенных из центра
        // растрового изображения в точки касания
        float oldAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
        float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

        // Вычисляем матрицу вращения
        float angle = newAngle - oldAngle;
        matrix *= Matrix.CreateTranslation(-refPoint.X, -refPoint.Y, 0);
        matrix *= Matrix.CreateRotationZ(angle);
        matrix *= Matrix.CreateTranslation(refPoint.X, refPoint.Y, 0);
```

```
        // По сути, вращаем старый вектор
        oldVector = oldVector.Length() / newVector.Length() * newVector;

        // Повторно вычисляем разницу
        delta = newVector - oldVector;
    }
    // Включаем перемещение
    matrix *= Matrix.CreateTranslation(delta.X, delta.Y, 0);
    return matrix;
}
```

Обратите внимание, что вызов *Matrix.CreateRotationZ* (Создать матрицу вращения относительно оси Z) располагается между двумя вызовами *Matrix.CreateTranslation* (Создать матрицу перемещения) для осуществления вращения относительно опорной точки, в роли которой выступает перенесенный центр *Texture2D*. В конце описания структуры еще один вызов *Matrix.CreateTranslation* обрабатывает составляющую жеста, которая обеспечивает перемещение, после извлечения из него составляющей вращения.

Новый метод *ComputeScaleAndRotateMatrix* получен на основании метода *ComputeScaleMatrix* из предыдущего проекта, в который была добавлена подобная логика. Метод *ComputeScaleAndRotateMatrix* вызывается дважды для любого жеста *Pinch*:

Проект XNA: DragPinchRotate   Файл: Game1.cs (фрагмент)

```
Matrix ComputeScaleAndRotateMatrix(Vector2 refPoint, Vector2 oldPoint, Vector2
newPoint)
{
    Matrix matrix = Matrix.Identity;
    Vector2 oldVector = oldPoint - refPoint;
    Vector2 newVector = newPoint - refPoint;

    // Находим углы для векторов, проведенных из опорной точки в точки касания
    float oldAngle = (float)Math.Atan2(oldVector.Y, oldVector.X);
    float newAngle = (float)Math.Atan2(newVector.Y, newVector.X);

    // Вычисляем матрицу вращения
    float angle = newAngle - oldAngle;
    matrix *= Matrix.CreateTranslation(-refPoint.X, -refPoint.Y, 0);
    matrix *= Matrix.CreateRotationZ(angle);
    matrix *= Matrix.CreateTranslation(refPoint.X, refPoint.Y, 0);

    // По сути, вращаем старый вектор
    oldVector = oldVector.Length() / newVector.Length() * newVector;
    float scale = 1;

    // Определяем коэффициент масштабирования из большей разницы
    if (Math.Abs(newVector.X - oldVector.X) > Math.Abs(newVector.Y - oldVector.Y))
        scale = newVector.X / oldVector.X;
    else
        scale = newVector.Y / oldVector.Y;

    // Вычисляем матрицу масштабирования
    if (!float.IsNaN(scale) && !float.IsInfinity(scale) && scale > 0)
    {
        scale = Math.Min(1.1f, Math.Max(0.9f, scale));

        matrix *= Matrix.CreateTranslation(-refPoint.X, -refPoint.Y, 0);
        matrix *= Matrix.CreateScale(scale, scale, 1);
        matrix *= Matrix.CreateTranslation(refPoint.X, refPoint.Y, 0);
    }
    return matrix;
}
```

Для обеспечения пропорционального масштабирования метод определяет, в каком направлении, по горизонтали или по вертикали, произошло наибольшее перемещение относительно опорной точки. Для этого производится сравнение абсолютных значений разностей *newVector* и *oldVector* (после исключения составляющей вращения). Также обратите внимание, что *Matrix.CreateScale* располагается между двумя вызовами *Matrix.CreateTranslation*, в которых используются координаты опорной точки.

Теперь все готово, чтобы выполнить перемещение и вращение посредством касания одним пальцем и пропорциональное масштабирование и вращение посредством касания двумя пальцами:

Это явно не описано, но перемещать изображение можно также двумя пальцами, если двигать ими в одном направлении.

Скольжение и инерция

В фильме «*Особое мнение* » (2002) Том Круз одним движением руки перетянул объект по экрану компьютера в сторону, и весь мир воскликнул: «Ух ты, здорово!»

Реализация инерции в интерфейсах обработки касания приложений, главным образом, является сферой ответственности разработчика. XNA помогает лишь немного, предоставляя жест *Flick*, который формируется, если пользователь быстро проводит пальцем по экрану. Свойство *Delta* объекта *GestureSample* отражает скорость перемещения пальца в пикселах в секунду. (Во всяком случае, так должно быть, но на самом деле, кажется, его значение ближе к половине фактической скорости.) Скорость представляется как *Vector2*, т.е. кроме величины, учитывается и направление.

Жест *Flick* не включает сведений о местоположении. По сути, это всегда один и тот же жест независимо от того, в каком месте экрана пользователь проводит пальцем. Если требуется реализовать инерцию на основании положения пальца и скорости его перемещения, вероятно, проще всего это сделать на основании жестов *Drag* через деление значений *Delta* на значение свойства *ElapsedGameTime* аргумента *GameTime* метода *Update*.

Чтобы реализовать инерцию, продолжайте перемещение объекта, взяв за основу исходную скорость и учитывая замедление. Если скорость выражена в единицах в секунду, вероятно, замедление выражается в единицах в секунду в квадрате. Каждую секунду скорость уменьшается на значение замедления до тех пор, пока модуль скорости не станет равным нулю. Для вызовов *Update*, которые формируются каждую долю секунды, скорость уменьшается пропорционально.

Проект FlickInertia (Инерция скольжения) демонстрирует очень простую реализацию инерции. В нем описаны поля *position*, *velocity* (скорость) и константа замедления:

Проект XNA: FlickInertia Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    const float DECELERATION = 1000; // пикселов в секунду в квадрате

    Texture2D texture;
    Vector2 position = Vector2.Zero;
    Vector2 velocity;
    SpriteFont segoe14;
    StringBuilder text = new StringBuilder();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        TouchPanel.EnabledGestures = GestureType.Flick;
    }
    …
}
```

Конструктор активирует только жесты *Flick*. Перегруженный метод *LoadContent* загружает и *Texture2D*, и шрифт для вывода на экран данных состояния (местоположение и скорость):

Проект XNA: FlickInertia Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
}
```

Перегруженный *Update* выполняет несколько операций. Прежде всего, это чтение жестов и сбор всех дополнительных данных о скорости в поле *velocity*. Если скорость меняется, вектор *velocity* умножается на истекшее время в секундах для получения изменения местоположения. Полученное значение добавляется к вектору *position*. После этого вектор *velocity* должен быть уменьшен на величину, соответствующую произведению константы DECELERATION (замедление) и истекшего времени в секундах. Наконец, выполняется форматирование *StringBuilder* для вывода на экран этих двух векторов:

Проект XNA: FlickInertia Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Задаем скорость на основании жеста Flick
    while (TouchPanel.IsGestureAvailable)
```

```
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if (gesture.GestureType == GestureType.Flick)
            velocity += gesture.Delta;
    }

    // Используем скорость для корректировки местоположения и замедления
    if (velocity != Vector2.Zero)
    {
        float elapsedSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
        position += velocity * elapsedSeconds;
        float newMagnitude = velocity.Length() - DECELERATION * elapsedSeconds;
        velocity.Normalize();
        velocity *= Math.Max(0, newMagnitude);
    }

    // Выводим на экран текущие координаты и скорость
    text.Remove(0, text.Length);
    text.AppendFormat("Position: {0} Velocity: {1}", position, velocity);

    base.Update(gameTime);
}
```

Перегруженный *Draw* отрисовывает и *Texture2D*, и *StringBuilder*:

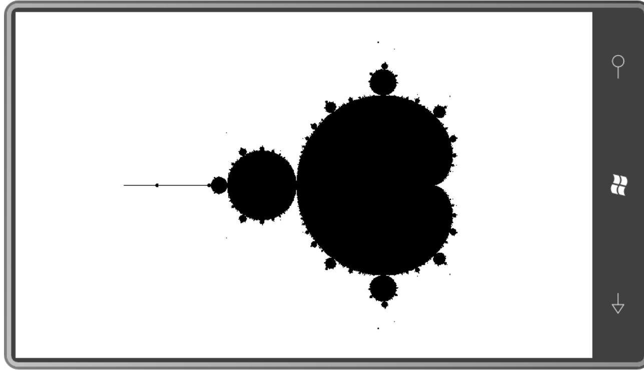
Проект XNA: FlickInertia Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(texture, position, Color.White);
    spriteBatch.DrawString(segoe14, text, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Да, изображение можно полностью задвинуть за границы экрана. Но поскольку приложение отвечает на касания в любом месте экрана, можно провести по экрану снова, чтобы вернуть изображение в область видимости.

Множество Мандельброта

В 1980 году Бенуа Мандельброт (1924–2010), рожденный в Польше французский и американский математик, который работал на IBM, впервые получил графическую визуализацию рекурсивного уравнения с комплексными числами, которое было выведено несколько ранее. Эта визуализация выглядела примерно следующим образом:



С тех пор множество Мандельброта (как его стали называть) стало самой любимой забавой разработчиков ПО.

Множество Мандельброта строится в комплексной плоскости, где горизонтальная ось представляет действительные числа (отрицательные слева и положительные справа), и вертикальная ось представляет мнимые числа (отрицательные внизу и положительные вверху). Возьмем любую точку на плоскости и назовем ее *c*, зададим *z* равным 0:

$$z = 0$$

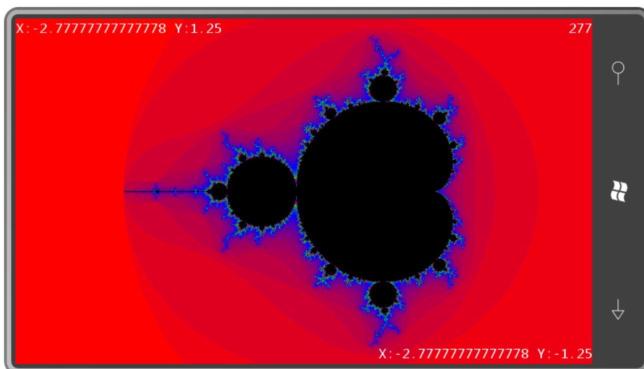
Теперь выполним следующую рекурсивную операцию:

$$z \rightarrow z^2 + c$$

Если модуль *z* не стремится к бесконечности, тогда *c* принадлежит множеству Мандельброта и в приведенном выше снимке экрана закрашивается черным.

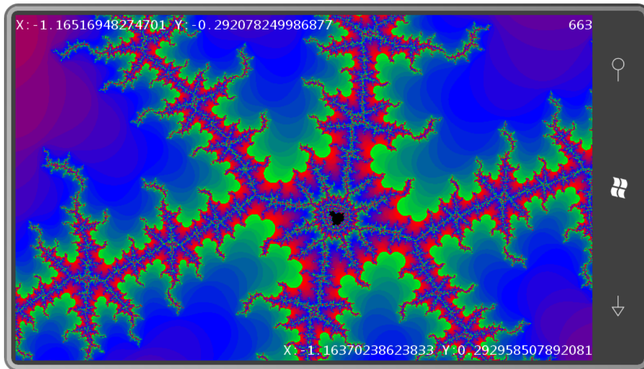
Для некоторых комплексных чисел (например, действительного числа 0) абсолютно очевидно, что число принадлежит множеству. Для других чисел (например, действительного числа 1) абсолютно очевидно, что они не принадлежат этому множеству. Для многих других чисел всего лишь необходимо вычислить значения. К счастью, если после конечного числа итераций модуль *z* превышает 2, мы знаем, что *c* не принадлежит множеству Мандельброта.

Для каждого числа *c*, не принадлежащего множеству Мандельброта, существует связанный с ним коэффициент «итераций», который соответствует числу итераций по вычислению *z*, которые имели место до момента, когда модуль *z* стал больше 2. Многие разработчики, занимающиеся визуализациями множества Мандельброта, используют этот коэффициент итераций для выбора цвета соответствующей точки. Это приводит к тому, что области, не принадлежащие множеству Мандельброта, выглядят намного интересней:



В верхнем правом углу отображаются комплексные координаты, ассоциированные с этим углом, и то же самое для нижнего правого угла. Число в верхнем правом углу – общее число итераций.

Одной из любопытных характеристик множества Мандельброта является то, что независимо от того, насколько увеличивается масштаб изображения, его сложность не уменьшается:



Это характеризует множество Мандельброта как фрактал. Бенуа Мандельброт считается основателем фрактальной геометрии. Учитывая простоту алгоритма, лежащего в основе этого изображения, получаемые результаты просто ошеломительные.

Можно ли представить лучшее приложение для демонстрации перетягивания и масштабирования посредством мультисенсорного ввода, а также алгоритмического формирования значений пикселов для *Texture2D*?

Очень часто при создании приложений с использование множества Мандельброта разработчики задают максимальный коэффициент итераций, например, 100 или 1000. Затем *z* вычисляется для каждого пиксела необходимое число раз, но не более этого заданного максимума. Если к этому моменту значение не расходится, пиксел закрашивается черным. С помощью псевдокода это можно записать следующим образом:

```
Для каждого пиксела
{
        Выполняем не более MAX итераций
        Закрашиваем пиксел черным либо другим цветом
}
```

Проблема с этим подходом в том, что он может обеспечивать неверные результаты при многократном увеличении. Как правило, чем больше увеличение определенной области множества Мандельброт, тем больше итераций необходимо для определения принадлежности пиксела множеству и цвета, которым он должен закрашиваться.

Эта проблема привела меня к другому подходу: мое приложение MandelbrotSet (Множество Мандельброта) изначально закрашивает все пикселы черным и затем во втором потоке выполнения делает следующее:

```
Выполнять бесконечное число раз
{
        Для каждого пиксела
        {
                Выполняем еще одну итерацию, если необходимо
                Возможно, закрашиваем пиксел некоторым цветом
        }
}
```

При таком подходе мы получаем экран, который чем больше времени проходит, тем все более красочным становится. Оборотная сторона в том, что для поддержки структуры данных, необходимой для этого, требуется 17 МБ памяти. Это слишком большой объем,

который невозможно сохранять при захоронении. И общая производительность ниже, чем при более традиционных подходах.

Рассмотрим структуру *PixelInfo*, используемую для хранения данных каждого пиксела. Приложение сохраняет массив этих структур параллельно с обычным массивом *pixels*, используемым для записи данных в *Texture2D*:

Проект XNA: MandelbrotSet Файл: PixelInfo.cs

```
using Microsoft.Xna.Framework;

namespace MandelbrotSet
{
    public struct PixelInfo
    {
        public static int pixelWidth;
        public static int pixelHeight;
        public static double xPixelCoordAtComplexOrigin;
        public static double yPixelCoordAtComplexOrigin;
        public static double unitsPerPixel;

        public static bool hasNewColors;
        public static int firstNewIndex;
        public static int lastNewIndex;

        public double cReal;
        public double cImag;
        public double zReal;
        public double zImag;
        public int iteration;
        public bool finished;
        public uint packedColor;

        public PixelInfo(int pixelIndex, uint[] pixels)
        {
            int x = pixelIndex % pixelWidth;
            int y = pixelIndex / pixelWidth;
            cReal = (x - xPixelCoordAtComplexOrigin) * unitsPerPixel;
            cImag = (yPixelCoordAtComplexOrigin - y) * unitsPerPixel;
            zReal = 0;
            zImag = 0;
            iteration = 0;
            finished = false;
            packedColor = pixels != null ? pixels[pixelIndex] :
Color.Black.PackedValue;
        }

        public bool Iterate()
        {
            double zImagSquared = zImag * zImag;
            zImag = 2 * zReal * zImag + cImag;
            zReal = zReal * zReal - zImagSquared + cReal;

            if (zReal * zReal + zImag * zImag >= 4.0)
            {
                finished = true;
                return true;
            }
            iteration++;
            return false;
        }
    }
}
```

Перейдем сразу к полям экземпляра. Изначально я написал структуру *Complex* (Комплексный) для инкапсуляции комплексных чисел и осуществления операций над ними, но, как обнаружилось, работа с действительной и мнимой частями напрямую существенно повышает производительность. Наша структура сохраняет описываемые выше значения *c* и *z*, текущую итерацию (*iteration*) и переменную *finished* (завершен) типа Boolean, которая принимает значение *true*, когда модуль *z* отличен от бесконечности. В этот момент значение *iteration* может использоваться для определения значения цвета.

Конструктор вычисляет *cReal* (Действительная часть с) и *cImag* (Мнимая часть с) из *pixelIndex* (Индекс пиксела), диапазон допустимых значений которого лежит от 0 до (но не включая) произведения ширины и высоты экрана в пикселах. Значения статических полей *pixelWidth* и *pixelHeight* зависят от размеров экрана и остаются неизменными в ходе всего приложения.

В вычислении *cReal* и *cImag* также участвуют три других статических поля. Поля *xPixelCoordAtComplexOrigin* (Координата x начала координат комплексной плоскости) и *yPixelCoordAtComplexOrigin* указывают горизонтальные и вертикальные координаты в пикселах, соответствующие началу координат комплексной плоскости. Очевидно, что использование типа *double* для этих полей свидетельствует о возможности представления дробных пикселов. Эти два поля меняют значения при операциях переноса. Поле *unitsPerPixel* (Единиц на пиксел) указывает на диапазон действительных или мнимых чисел, ассоциированный в настоящий момент с одним пикселом. Это значение меняется при операциях масштабирования.

Данная структура *PixelInfo* включает больше значений типа *double*, чем все остальные приложения на XNA этой книги вместе взятые. Сначала я, конечно же, сделал все эти значения типа *float* (и *pixelCoordAtComplexOrigin* было типа *Vector2*), но перешел к *double*, как только исчерпал точность *float* при масштабировании. Любопытно, что переход от *float* к *double* очень мало сказался на производительности.

Второй аргумент конструктора является необязательным. Если он присутствует, конструктор будет копировать соответствующий цвет из массива *pixels* в свое поле *packedColor* (Цвет в компактной форме). Вскоре мы рассмотрим, как это работает.

Остальные три статических поля используются для обмена данными внутри потока. Поток, осуществляющий вычисления, задает эти поля при изменении значения цвета; значения полей сбрасываются, когда массив структур *PixelInfo* используется для обновления массива *pixels* и объекта *Texture2D*.

Наконец, метод *Iterate* (Выполнять итерации) осуществляет основные итеративные вычисления, используя умножение, а не вызовы *Math.Pow*, из соображений производительности. *Iterate* возвращает значение *true*, если *z* не стремится к бесконечности.

Благодаря этим статическим полям структуры *PixelInfo* мне удалось сохранить количество полей в производном от *Game* классе в разумных пределах. В данном фрагменте можно увидеть и обычный массив *pixels*, и массив *PixelInfo*. Объект *pixelInfosLock* (Блокировка pixelInfo) используется для синхронизации потоков.

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    Texture2D texture;
```

```
    uint[] pixels;
    PixelInfo[] pixelInfos;
    Matrix drawMatrix = Matrix.Identity;
    int globalIteration = 0;
    object pixelInfosLock = new object();

    SpriteFont segoe14;
    StringBuilder upperLeftCoordText = new StringBuilder();
    StringBuilder lowerRightCoordText = new StringBuilder();
    StringBuilder upperRightStatusText = new StringBuilder();
    Vector2 lowerRightCoordPosition, upperRightStatusPosition;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для всех устройств Windows Phone - 30
кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Задаем полноэкранный режим и активируем жесты
        graphics.IsFullScreen = true;
        TouchPanel.EnabledGestures = GestureType.FreeDrag | GestureType.DragComplete
|
                                     GestureType.Pinch | GestureType.PinchComplete;
    }
    …
}
```

Среди полей также включен объект *Matrix*, который обрабатывает перемещение и масштабирование, но только в процессе выполнения операций жеста. Как только палец снимается с экрана – и поскольку жесты *DragComplete* и *PinchComplete* активированы, приложение может определять, когда это происходит – массивы *pixels* и *PixelInfo* полностью перестраиваются, и объект *Matrix* возвращается к своему значению по умолчанию. Это оказывается одной из самых сложных частей приложения.

Перегруженный *LoadContent* на удивление прост:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    viewport = this.GraphicsDevice.Viewport;
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
}
```

Но такая простота обусловлена лишь тем, что вся остальная инициализация выполняется в сочетании с захоронением. Конечно, я хотел бы сохранять весь массив объектов *PixelInfo*, но учитывая то, что каждый из них размером 44 байта, и размер всего массива достигает 17 мегабайт, можно понять, почему операционная система Windows Phone 7, кажется, сопротивляется моим желаниям.

Вместо этого приложение выполняет захоронение четырех элементов, необходимых для его корректного перезапуска. Статические поля *xPixelCoordAtComplexOrigin*, *yPixelCoordAtComplexOrigin* и в *unitsPerPixel* обеспечивают приложению возможность восстановить массив *PixelInfo*. Кроме того, приложение сохраняет изображение всего экрана как файл в формате PNG и затем восстанавливает и его тоже. При возвращении из

захоронения приложение выглядит точно так же, как до него. Но это лишь обманчивый внешний вид, поскольку все вычисления для каждого пиксела требуется начинать заново. Как следствие, некоторое время после возвращения из захоронения экран может оставаться без изменений.

Код для сохранения и восстановления объектов *Texture2D* при захоронении стандартен и довольно прост, но я решил создать пару методов, которые будут выполнять всю эту работу. Статический класс *Texture2DExtensions* (Расширения двухмерной текстуры) библиотеки Petzold.Phone.Xna включает следующие два метода. Первый – это метод расширения, поэтому он может вызываться прямо для объекта *Texture2D*:

Проект XNA: Petzold.Phone.Xna Файл: Texture2DExtensions.cs (фрагмент)

```
public static void SaveToPhoneServiceState(this Texture2D texture, string key)
{
    MemoryStream memoryStream = new MemoryStream();
    texture.SaveAsPng(memoryStream, texture.Width, texture.Height);
    PhoneApplicationService.Current.State[key] = memoryStream.GetBuffer();
}
```

Этот метод создает объект *MemoryStream* и просто передает его в метод *SaveAsPng* (Сохранить как PNG) объекта *Texture2D*. Сам объект *MemoryStream* не может быть сериализован, но его метод *GetBuffer* возвращает массив байт, который может быть сериализован.

Вспомогательный метод загрузки не является методом расширения, потому что приводит к созданию нового объекта *Texture2D*. Байтовый массив извлекается из хранилища и передается в конструктор *MemoryStream*, и этот *MemoryStream* затем используется со статическим методом *Texture2D.FromStream*:

Проект XNA: Petzold.Phone.Xna Файл: Texture2DExtensions.cs (фрагмент)

```
public static Texture2D LoadFromPhoneServiceState(GraphicsDevice graphicsDevice,
string key)
{
    Texture2D texture = null;

    if (PhoneApplicationService.Current.State.ContainsKey(key))
    {
        byte[] buffer = PhoneApplicationService.Current.State[key] as byte[];
        MemoryStream memoryStream = new MemoryStream(buffer);
        texture = Texture2D.FromStream(graphicsDevice, memoryStream);
        memoryStream.Close();
    }
    return texture;
}
```

Рассмотрим перегруженные *OnActivated* и *OnDeactivated* приложения MandelbrotSet, которые используют описанные выше два метода, и метод под именем *InitializePixelInfo* (Инициализировать PixelInfo):

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void OnActivated(object sender, EventArgs args)
{
    PhoneApplicationService appService = PhoneApplicationService.Current;
```

```
    if (appService.State.ContainsKey("xOrigin") &&
        appService.State.ContainsKey("yOrigin") &&
        appService.State.ContainsKey("resolution"))
    {
        PixelInfo.xPixelCoordAtComplexOrigin = (double)appService.State["xOrigin"];
        PixelInfo.yPixelCoordAtComplexOrigin = (double)appService.State["yOrigin"];
        PixelInfo.unitsPerPixel = (double)appService.State["resolution"];
    }
    else
    {
        // Первоначальный запуск приложения
        PixelInfo.xPixelCoordAtComplexOrigin = 2 * viewport.Width / 3f;
        PixelInfo.yPixelCoordAtComplexOrigin = viewport.Height / 2;
        PixelInfo.unitsPerPixel = Math.Max(2.5 / viewport.Height,
                                           3.0 / viewport.Width);
    }

    UpdateCoordinateText();

    // Повторное создание или восстановление растрового изображения после
захоронения
    texture = Texture2DExtensions.LoadFromPhoneServiceState(this.GraphicsDevice,
                                                            "mandelbrotBitmap");
    if (texture == null)
        texture = new Texture2D(this.GraphicsDevice, viewport.Width,
viewport.Height);

    // Получение данных текстуры и массива пикселов
    PixelInfo.pixelWidth = texture.Width;
    PixelInfo.pixelHeight = texture.Height;
    int numPixels = PixelInfo.pixelWidth * PixelInfo.pixelHeight;
    pixels = new uint[numPixels];
    texture.GetData<uint>(pixels);

    // Создание и инициализация массива PixelInfo
    pixelInfos = new PixelInfo[numPixels];
    InitializePixelInfo(pixels);

    // Запуск потока вычислений
    Thread thread = new Thread(PixelSetterThread);
    thread.Start();

    base.OnActivated(sender, args);
}

protected override void OnDeactivated(object sender, EventArgs args)
{
    PhoneApplicationService.Current.State["xOrigin"] =
PixelInfo.xPixelCoordAtComplexOrigin;
    PhoneApplicationService.Current.State["yOrigin"] =
PixelInfo.yPixelCoordAtComplexOrigin;
    PhoneApplicationService.Current.State["resolution"] = PixelInfo.unitsPerPixel;

    texture.SaveToPhoneServiceState("mandelbrotBitmap");

    base.OnDeactivated(sender, args);
}

void InitializePixelInfo(uint[] pixels)
{
    for (int index = 0; index < pixelInfos.Length; index++)
    {
        pixelInfos[index] = new PixelInfo(index, pixels);
    }

    PixelInfo.hasNewColors = true;
    PixelInfo.firstNewIndex = 0;
    PixelInfo.lastNewIndex = pixelInfos.Length - 1;
}
```

К концу выполнения метода *OnActivated* все поля и объекты инициализированы и готовы к работе, поэтому запускается второй поток выполнения на базе метода *PixelSetterThread* (Поток метода задания значений пикселов). Этот метод выполняется в течение всего остального времени жизни приложения, постоянно перебирая все члены массива *PixelInfo*, индексированные *pixelIndex*, и вызывая метод *Iterate*. Если *Iterate* возвращает *true*, пикселу задается значение цвета:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
void PixelSetterThread()
{
    int pixelIndex = 0;

    while (true)
    {
        lock (pixelInfosLock)
        {
            if (!pixelInfos[pixelIndex].finished)
            {
                if (pixelInfos[pixelIndex].Iterate())
                {
                    int iteration = pixelInfos[pixelIndex].iteration;
                    pixelInfos[pixelIndex].packedColor =
                                        GetPixelColor(iteration).PackedValue;

                    PixelInfo.hasNewColors = true;
                    PixelInfo.firstNewIndex = Math.Min(PixelInfo.firstNewIndex,
pixelIndex);
                    PixelInfo.lastNewIndex = Math.Max(PixelInfo.lastNewIndex,
pixelIndex);
                }
                else
                {
                    // Особый случай: при увеличении масштаба предотвращаем
                    // сохранение цветных блоков внутри множества Мандельброта
                    if (pixelInfos[pixelIndex].iteration == 500 &&
                        pixelInfos[pixelIndex].packedColor !=
Color.Black.PackedValue)
                    {
                        pixelInfos[pixelIndex].packedColor =
Color.Black.PackedValue;

                        PixelInfo.hasNewColors = true;
                        PixelInfo.firstNewIndex =
                                        Math.Min(PixelInfo.firstNewIndex,
pixelIndex);
                        PixelInfo.lastNewIndex =
                                        Math.Max(PixelInfo.lastNewIndex,
pixelIndex);
                    }
                }
            }

            if (++pixelIndex == pixelInfos.Length)
            {
                pixelIndex = 0;
                globalIteration++;
            }
        }
    }
}

Color GetPixelColor(int iteration)
{
    float proportion = (iteration / 32f) % 1;
```

```
    if (proportion < 0.5)
        return new Color(1 - 2 * proportion, 0, 2 * proportion);

    proportion = 2 * (proportion - 0.5f);

    return new Color(0, proportion, 1 - proportion);
}
```

Несмотря на то что для этого потока важно выполнять вычисления максимально быстро, он также пытается взять на себя некоторые функции перегруженного *Update* (грядущего). В статических полях структуры *PixelInfo* этот поток указывает минимальный и максимальный индексы измененных пикселов.

Я упоминал ранее, что в более простых приложениях для вычисления множества Мандельброта обычно задается максимальное число итераций. (В статье Википедии, посвященной множеству Мандельброта, приведен алгоритм псевдокода, где *max_iteration* задано значение 1000.) В своем приложении мне пришлось задать максимум для количества итераций только здесь. Как будет показано вскоре, при масштабировании посредством пары пальцев приложению приходится начинать все с самого начала и использовать новый массив структур *PixelInfo*. Но для целей визуализации и создания приближенного изображения объект *Texture2D* просто растягивается. Это обычно приводит к тому, что некоторые цветные пикселы оказываются в области множества Мандельброта, и используемый здесь алгоритм никогда не обеспечит закрашивание этих пикселов черным опять. Поэтому если число итераций для конкретного пиксела достигает 500, и цвет этого пиксела не черный, он закрашивается черным. Впоследствии этот пиксел может быть опять закрашен в какой-либо другой цвет, но на данный момент нам это не известно.

Рассмотрим первую часть перегруженного метода *Update*. В ней данные цвета из массива *PixelInfo*, вычисленного во втором потоке, передаются в массив *pixels*, и затем объект *Texture2D* обновляется значениями из этого массива:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Обновляем текстуру пикселов массивом pixelInfos
    if (PixelInfo.hasNewColors)
    {
        lock (pixelInfosLock)
        {
            // Передаем новые цвета в массив pixels
            for (int pixelIndex = PixelInfo.firstNewIndex;
                     pixelIndex <= PixelInfo.lastNewIndex;
                     pixelIndex++)
            {
                pixels[pixelIndex] = pixelInfos[pixelIndex].packedColor;
            }

            // Передаем новый pixels в текстуру
            int firstRow = PixelInfo.firstNewIndex / texture.Width;
            int numRows = PixelInfo.lastNewIndex / texture.Width - firstRow + 1;
            Rectangle rect = new Rectangle(0, firstRow, texture.Width, numRows);
            texture.SetData<uint>(0, rect, pixels, firstRow * texture.Width,
                                           numRows * texture.Width);

            // Перезадаем PixelInfo
```

```
            PixelInfo.hasNewColors = false;
            PixelInfo.firstNewIndex = Int32.MaxValue;
            PixelInfo.lastNewIndex = 0;
        }
    }

    // Обновляем отображаемое значение globalIteration
    upperRightStatusText.Remove(0, upperRightStatusText.Length);
    upperRightStatusText.AppendFormat("{0}", globalIteration + 1);
    Vector2 textSize = segoe14.MeasureString(upperRightStatusText);
    upperRightStatusPosition = new Vector2(viewport.Width - textSize.X, 0);
    …
}
```

Панорамирование и масштабирование

Весь остальной код перегруженного *Update* посвящен обработке сенсорного ввода. Идея здесь проста: любое касание, перемещение, изменение масштаба не приводит к необратимым изменениям. Эффект перемещения и масштабирования обеспечивается изменением объекта *Matrix* под именем *drawMatrix* (Матрица рисования), которая используется в вызове *Begin* объекта *SpriteBatch*.

Но как только палец или пальцы пользователя снимаются с экрана, приложение меняет массивы *PixelInfo* и *pixels* соответственно новому расположению и масштабу экрана. Не делается никакой попытки сохранить что-либо, что вышло за границы области просмотра.

Рассмотрим обработку жестов *FreeDrag* и *DragComplete* для операций переноса:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Чтение жестов касания
    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch (gesture.GestureType)
        {
            case GestureType.FreeDrag:
                // Корректируем drawMatrix соответственно смещению
                drawMatrix.M41 += gesture.Delta.X;
                drawMatrix.M42 += gesture.Delta.Y;
                break;

            case GestureType.DragComplete:
                // Обновляем текстуру пикселов массивом pixelInfos с учетом смещения
                lock (pixelInfosLock)
                {
                    pixelInfos = TranslatePixelInfo(pixelInfos, drawMatrix);

                    for (int pixelIndex = 0; pixelIndex < pixelInfos.Length;
pixelIndex++)
                        pixels[pixelIndex] = pixelInfos[pixelIndex].packedColor;

                    PixelInfo.hasNewColors = false;
                    PixelInfo.firstNewIndex = Int32.MaxValue;
                    PixelInfo.lastNewIndex = 0;
                }
                texture.SetData<uint>(pixels);

                drawMatrix = Matrix.Identity;
                globalIteration = 0;
```

```
                break;
            …
        }
        UpdateCoordinateText();
    }
    base.Update(gameTime);
}
```

Когда пользователь водит пальцем по экрану, меняется только *drawMatrix*, но когда пользователь снимает палец с экрана, в ходе обработки жеста *DragComplete* производится вызов метода *TranslatePixelInfo* (Перенести PixelInfo) для переноса элементов в массив структур *PixelInfo* соответственно окончательному положению точки касания. К счастью, пикселы, перемещенные из одной части экрана в другую, можно сохранить; пикселы в новом месте изначально закрашиваются черным. Затем *Update* обеспечивает перенос цветов пикселов из массива *PixelInfo* в массив *pixels* и обновляет из него *Texture2D*. После этого *drawMatrix* может быть возвращен к единичной матрице.

Метод *TranslatePixelInfo* использует окончательные коэффициенты переноса *drawMatrix* для задания новых значений полей *PixelInfo.xPixelCoordAtComplexOrigin* и *PixelInfo.yPixelCoordAtComplexOrigin* и выполняет циклический сдвиг членов *PixelInfo*:

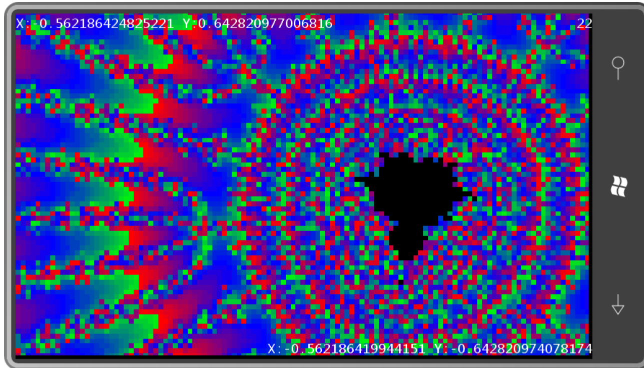
Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
PixelInfo[] TranslatePixelInfo(PixelInfo[] srcPixelInfos, Matrix drawMatrix)
{
    int x = (int)(drawMatrix.M41 + 0.5);
    int y = (int)(drawMatrix.M42 + 0.5);
    PixelInfo.xPixelCoordAtComplexOrigin += x;
    PixelInfo.yPixelCoordAtComplexOrigin += y;
    PixelInfo[] dstPixelInfos = new PixelInfo[srcPixelInfos.Length];

    for (int dstY = 0; dstY < PixelInfo.pixelHeight; dstY++)
    {
        int srcY = dstY - y;
        int srcRow = srcY * PixelInfo.pixelWidth;
        int dstRow = dstY * PixelInfo.pixelWidth;

        for (int dstX = 0; dstX < PixelInfo.pixelWidth; dstX++)
        {
            int srcX = dstX - x;
            int dstIndex = dstRow + dstX;

            if (srcX >= 0 && srcX < PixelInfo.pixelWidth &&
                srcY >= 0 && srcY < PixelInfo.pixelHeight)
            {
                int srcIndex = srcRow + srcX;
                dstPixelInfos[dstIndex] = pixelInfos[srcIndex];
            }
            else
            {
                dstPixelInfos[dstIndex] = new PixelInfo(dstIndex, null);
            }
        }
    }
    return dstPixelInfos;
}
```

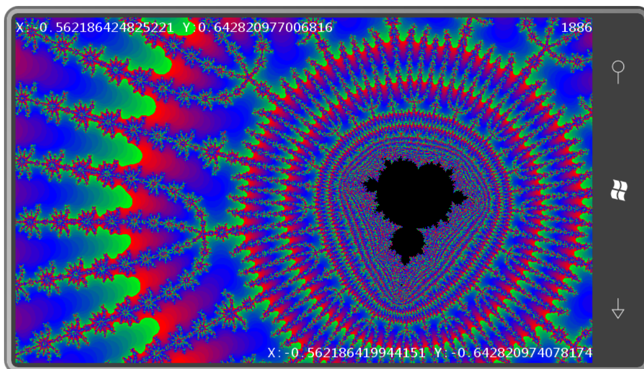
При масштабировании используется противоположный подход. По самой природе множества Мандельброта каждая его точка уникальна и не может быть получена на основании значений соседних точек. По этой причине любая операция масштабирования

должна приводить к повторному созданию массива *PixelInfo* и повторному выполнению всех вычислений с самого начала.

Но визуальные элементы могут быть сохранены как временные приближения. Поэтому *Update* обрабатывает жест *PinchComplete*, применяя трансформацию к массиву *pixels* и затем используя его для задания цветов в массиве *PixelInfo*. В первое мгновение после увеличения масштаба на экране будет отображаться примерно следующее:



Но пройдет немного времени, и мы увидим такое изображение:



Код обработки жеста *Pinch* нам знаком. Единственное отличие в том, что в данном случае он определяет, в каком направлении произошло максимальное перемещение: по вертикали или по горизонтали. Эти данные передаются в метод *ComputeScaleMatrix* (Вычислить матрицу масштабирования):

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Чтение жестов касания
    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch (gesture.GestureType)
        {
            …
            case GestureType.Pinch:
                bool xDominates = Math.Abs(gesture.Delta.X) +
Math.Abs(gesture.Delta2.X) >
                                  Math.Abs(gesture.Delta.Y) +
Math.Abs(gesture.Delta2.Y);

                Vector2 oldPoint1 = gesture.Position - gesture.Delta;
```

```
                Vector2 newPoint1 = gesture.Position;
                Vector2 oldPoint2 = gesture.Position2 - gesture.Delta2;
                Vector2 newPoint2 = gesture.Position2;

                drawMatrix *= ComputeScaleMatrix(oldPoint1, oldPoint2, newPoint2,
                                                 xDominates);
                drawMatrix *= ComputeScaleMatrix(newPoint2, oldPoint1, newPoint1,
                                                 xDominates);
                break;

            case GestureType.PinchComplete:
                // Задаем текстуру из увеличенных пикселов
                pixels = ZoomPixels(pixels, drawMatrix);
                texture.SetData<uint>(pixels);

                // Задаем новые параметры PixelInfo
                PixelInfo.xPixelCoordAtComplexOrigin *= drawMatrix.M11;
                PixelInfo.xPixelCoordAtComplexOrigin += drawMatrix.M41;
                PixelInfo.yPixelCoordAtComplexOrigin *= drawMatrix.M22;
                PixelInfo.yPixelCoordAtComplexOrigin += drawMatrix.M42;
                PixelInfo.unitsPerPixel /= drawMatrix.M11;

                // Повторно инициализируем PpixelInfos
                lock (pixelInfosLock)
                {
                    InitializePixelInfo(pixels);
                }

                drawMatrix = Matrix.Identity;
                globalIteration = 0;
                break;
        }
        UpdateCoordinateText();
    }
    base.Update(gameTime);
}
```

Метод *ComputeScaleMatrix* очень похож на метод с таким же названием из приложения *DragAndPinch*, за исключением того, что он выполняет пропорциональное масштабирование на основании передаваемого в него аргумента типа Boolean:

Проект XNA: MandelbrotSet   Файл: Game1.cs (фрагмент)

```
Matrix ComputeScaleMatrix(Vector2 refPoint, Vector2 oldPoint, Vector2 newPoint,
                          bool xDominates)
{
    float scale = 1;

    if (xDominates)
        scale = (newPoint.X - refPoint.X) / (oldPoint.X - refPoint.X);
    else
        scale = (newPoint.Y - refPoint.Y) / (oldPoint.Y - refPoint.Y);

    if (float.IsNaN(scale) || float.IsInfinity(scale) || scale < 0)
    {
        return Matrix.Identity;
    }

    scale = Math.Min(1.1f, Math.Max(0.9f, scale));

    Matrix matrix = Matrix.CreateTranslation(-refPoint.X, -refPoint.Y, 0);
    matrix *= Matrix.CreateScale(scale, scale, 1);
    matrix *= Matrix.CreateTranslation(refPoint.X, refPoint.Y, 0);

    return matrix;
}
```

Метод *ZoomPixels* (Масштабировать пикселы), вызываемый для жеста *PinchComplete*, получает инверсный *Matrix* и использует его в вычислениях исходных координат пикселов из результирующих координат. К счастью, для инвертирования матрицы необходимо просто вызвать статический метод *Matrix.Invert*. В более ранней версии приложения вызывался *Matrix.CreateScale* (приведен выше), в качестве третьего аргумента в него передавался нуль. Это обеспечивало создание неинвертируемой матрицы, и вызов *Invert* приводил к созданию матрицы со значениями NaN («not a number») во всех полях. Это нехорошо.

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
uint[] ZoomPixels(uint[] srcPixels, Matrix matrix)
{
    Matrix invMatrix = Matrix.Invert(matrix);
    uint[] dstPixels = new uint[srcPixels.Length];

    for (int dstY = 0; dstY < PixelInfo.pixelHeight; dstY++)
    {
        int dstRow = dstY * PixelInfo.pixelWidth;

        for (int dstX = 0; dstX < PixelInfo.pixelWidth; dstX++)
        {
            int dstIndex = dstRow + dstX;
            Vector2 dst = new Vector2(dstX, dstY);
            Vector2 src = Vector2.Transform(dst, invMatrix);
            int srcX = (int)(src.X + 0.5f);
            int srcY = (int)(src.Y + 0.5f);

            if (srcX >= 0 && srcX < PixelInfo.pixelWidth &&
                srcY >= 0 && srcY < PixelInfo.pixelHeight)
            {
                int srcIndex = srcY * PixelInfo.pixelWidth + srcX;
                dstPixels[dstIndex] = srcPixels[srcIndex];
            }
            else
            {
                dstPixels[dstIndex] = Color.Black.PackedValue;
            }
        }
    }
    return dstPixels;
}
```

На этом все самое интересное сделано, но в приложениях по вычислению множества Мандельброта считается *необходимым* каким-то образом отображать текущее положение на комплексной плоскости. Метод *UpdateCoordinateText* (Обновить отображаемые координаты) отвечает за вычисление координат верхнего левого и нижнего правого углов, их форматирование в объектах *StringBuilder* и определение того, где они должны отображаться:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
void UpdateCoordinateText()
{
    double xAdjustedPixelCoord =
        PixelInfo.xPixelCoordAtComplexOrigin * drawMatrix.M11 + drawMatrix.M41;
    double yAdjustedPixelCoord =
        PixelInfo.yPixelCoordAtComplexOrigin * drawMatrix.M22 + drawMatrix.M42;
    double adjustedUnitsPerPixel = PixelInfo.unitsPerPixel / drawMatrix.M11;

    double xUpperLeft = -adjustedUnitsPerPixel * xAdjustedPixelCoord;
    double yUpperLeft = adjustedUnitsPerPixel * yAdjustedPixelCoord;

    upperLeftCoordText.Remove(0, upperLeftCoordText.Length);
```

```
    upperLeftCoordText.AppendFormat("X:{0} Y:{1}", xUpperLeft, yUpperLeft);

    double xLowerRight = xUpperLeft + PixelInfo.pixelWidth * adjustedUnitsPerPixel;
    double yLowerRight = -yUpperLeft + PixelInfo.pixelHeight *
adjustedUnitsPerPixel;

    lowerRightCoordText.Remove(0, lowerRightCoordText.Length);
    lowerRightCoordText.AppendFormat("X:{0} Y:{1}", xLowerRight, yLowerRight);

    Vector2 textSize = segoe14.MeasureString(lowerRightCoordText);
    lowerRightCoordPosition = new Vector2(viewport.Width - textSize.X,
                                          viewport.Height - textSize.Y);
}
```

После всего этого метод *Draw* довольно прост. Заметьте, что методы *Begin* и *End* объекта *SpriteBatch* вызываются дважды. Для первого вызова необходим объект *Matrix*, который перемещает и масштабирует *Texture2D* в ходе его обработки, и второй вызов предназначен для работы с текстовыми элементами:

Проект XNA: MandelbrotSet Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    // Отрисовываем представление множества Мальдеброта
    spriteBatch.Begin(SpriteSortMode.Immediate, null, null, null, null, null,
drawMatrix);
    spriteBatch.Draw(texture, Vector2.Zero, null, Color.White,
                     0, Vector2.Zero, 1, SpriteEffects.None, 0);
    spriteBatch.End();

    // Отрисовываем текстовое представление координат и состояния
    spriteBatch.Begin();
    spriteBatch.DrawString(segoe14, upperLeftCoordText, Vector2.Zero, Color.White);
    spriteBatch.DrawString(segoe14, lowerRightCoordText,
                                    lowerRightCoordPosition, Color.White);
    spriteBatch.DrawString(segoe14, upperRightStatusText,
                                    upperRightStatusPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Компоненты игры

Для завершения данной главы я припас два приложения, обеспечивающие отображение все того же старого *Texture2D*, который использовался в этой главе ранее. Отличие в том, что пользователь сможет определять трансформацию изображения интерактивно, перетягивая углы текстуры.

Мишени для касания и перетягивания в данных приложениях обозначены полупрозрачными дисками в углах *Texture2D*. Неплохо было бы создать для этих дисков код, который можно было бы использовать многократно в разных приложениях. В традиционной среде разработки графических элементов их можно было бы представить как *элементы управления*, но в XNA их называют *компонентами игры*.

Компоненты помогают разбивать XNA-приложения на модули. Они могут наследоваться от класса *GameComponent* (Компонент игры), но часто являются производными от *DrawableGameComponent* (Компонент игры с возможностью отрисовки), что обеспечивает

им возможность выводить что-то на экран в дополнение и поверх того, что отрисовывает метод *Draw* класса *Game*.

Чтобы добавить в проект класс нового компонента, щелкнем правой кнопкой мыши имя проекта, выберем Add и New Item и затем из списка выберем Game Component (Компонент игры). Если мы хотим, чтобы компонент участвовал в рисовании, понадобится изменить базовый класс на *DrawableGameComponent* и перегрузить метод *Draw*.

Как правило, игра создает необходимые экземпляры компонентов либо в конструкторе игры, либо в ходе выполнения метода *Initialize*. Эти компоненты официально становятся частью игры после того, как добавляются в коллекцию *Components*, определенную классом *Game*.

Как и класс *Game*, производные от *DrawableGameComponent*, как правило, перегружают методы *Initialize*, *LoadContent*, *Update* и *Draw*. Когда перегруженный *Initialize* класса *Game* вызывает метод базового класса, вызываются методы *Initialize* всех компонентов. Аналогично с перегруженными методами *LoadComponent*, *Update* и *Draw*.

Как известно, перегруженный метод *Update* обычно обрабатывает сенсорный ввод. Из собственного опыта я знаю, что организовать сенсорный ввод в игровые компоненты несколько проблематично. В итоге кажется, что сама игра и компоненты состязаются за ввод.

Чтобы исправить это, я решил, что мой производный *Game* будет единолично отвечать за вызов *TouchPanel.GetState*, но затем игра будет предоставлять компонентам возможность обработать этот сенсорный ввод. Для реализации данной идеи я создал следующий интерфейс для производных от *GameComponent* и *DrawableGameComponent*:

Проект XNA: Petzold.Phone.Xna Файл: IProcessTouch.cs

```
using Microsoft.Xna.Framework.Input.Touch;

namespace Petzold.Phone.Xna
{
    public interface IProcessTouch
    {
        bool ProcessTouch(TouchLocation touch);
    }
}
```

Если компонент игры реализует этот интерфейс, игра вызывает метод *ProcessTouch* (Обработать сенсорный ввод) компонента игры для каждого объекта *TouchLocation*. Если компонент игры собирается использовать этот *TouchLocation*, его метод *ProcessTouch* возвращает *true*, и игра, возможно, игнорирует этот *TouchLocation*.

Первым я продемонстрирую компонент Dragger (Модуль перетягивания), который входит в библиотеку Petzold.Phone.Xna. *Dragger* наследуется от *DrawableGameComponent* и реализует интерфейс *IProcessTouch*:

Проект XNA: Petzold.Phone.Xna Файл: Dragger.cs (фрагмент, демонстрирующий поля)

```
public class Dragger : DrawableGameComponent, IProcessTouch
{
    SpriteBatch spriteBatch;
    int? touchId;

    public event EventHandler PositionChanged;

    public Dragger(Game game)
        : base(game)
```

```
    {
    }

    public Texture2D Texture { set; get; }
    public Vector2 Origin { set; get; }
    public Vector2 Position { set; get; }

    …
}
```

В конструктор производного от *GameComponent* класса должен передаваться родительский класс *Game*, что позволит компоненту использовать некоторые свойства *Game* (такие как объект *GraphicsDevice*). Производный от *DrawableGameComponent* обычно создает *SpriteBatch* для собственных нужд, так же как это делает производный от *Game*.

Dragger также описывает поле *touchId* (Идентификатор касания) для помощи в обработке сенсорного ввода, открытое событие *PositionChanged* и три открытых свойства: *Texture* типа *Texture2D*, *Vector2* под именем *Origin* (значением которого обычно задается центр *Texture2D*) и еще один *Vector2* для *Position*.

Приложение, использующее *Dragger*, могло бы определять для компонента пользовательский *Texture2D* и задавать его через это открытое свойство *Texture* и в это же время, возможно, задавать свойство *Origin*. Однако *Dragger* определяет для себя свойство *Texture* по умолчанию во время выполнения своего метода *LoadContent*:

Проект XNA: Petzold.Phone.Xna Файл: Dragger.cs (фрагмент)

```
protected override void  LoadContent()
{
    spriteBatch = new SpriteBatch(this.GraphicsDevice);

    // Создаем текстуру по умолчанию
    int radius = 48;
    Texture2D texture = new Texture2D(this.GraphicsDevice, 2 * radius, 2 * radius);
    uint[] pixels = new uint[texture.Width * texture.Height];

    for (int y = 0; y < texture.Height; y++)
        for (int x = 0; x < texture.Width; x++)
        {
            Color clr = Color.Transparent;

            if ((x - radius) * (x - radius) +
                (y - radius) * (y - radius) <
                radius * radius)
            {
                clr = new Color(0, 128, 128, 128);
            }
            pixels[y * texture.Width + x] = clr.PackedValue;
        }
    texture.SetData<uint>(pixels);

    Texture = texture;
    Origin = new Vector2(radius, radius);

    base.LoadContent();
}
```

Класс *Dragger* реализует интерфейс *IProcessTouch*, так что у него есть метод *ProcessTouch*, который вызывается из производного от *Game* для каждого объекта *TouchLocation*. Метод *ProcessTouch* фиксирует касания самого компонента. Когда касание случается, этот метод

сохраняет его ID и фактически присваивает это касание себе до его завершения. Для каждого перемещения точки касания *Dragger* формирует событие *PositionChanged*.

Проект XNA: Petzold.Phone.Xna Файл: Dragger.cs (фрагмент)

```
public bool ProcessTouch(TouchLocation touch)
{
    if (Texture == null)
        return false;

    bool touchHandled = false;

    switch (touch.State)
    {
        case TouchLocationState.Pressed:
            if ((touch.Position.X > Position.X - Origin.X) &&
                (touch.Position.X < Position.X - Origin.X + Texture.Width) &&
                (touch.Position.Y > Position.Y - Origin.Y) &&
                (touch.Position.Y < Position.Y - Origin.Y + Texture.Height))
            {
                touchId = touch.Id;
                touchHandled = true;
            }
            break;

        case TouchLocationState.Moved:
            if (touchId.HasValue && touchId.Value == touch.Id)
            {
                TouchLocation previousTouch;
                touch.TryGetPreviousLocation(out previousTouch);
                Position += touch.Position - previousTouch.Position;

                // Формируем событие!
                if (PositionChanged != null)
                    PositionChanged(this, EventArgs.Empty);

                touchHandled = true;
            }
            break;

        case TouchLocationState.Released:
            if (touchId.HasValue && touchId.Value == touch.Id)
            {
                touchId = null;
                touchHandled = true;
            }
            break;
    }
    return touchHandled;
}
```

Перегруженный *Draw* просто отрисовывает *Texture2D* в новом месте:

Проект XNA: Petzold.Phone.Xna Файл: Dragger.cs (фрагмент)

```
public override void Draw(GameTime gameTime)
{
    if (Texture != null)
    {
        spriteBatch.Begin();
        spriteBatch.Draw(Texture, Position, null, Color.White,
                         0, Origin, 1, SpriteEffects.None, 0);
        spriteBatch.End();
    }
```

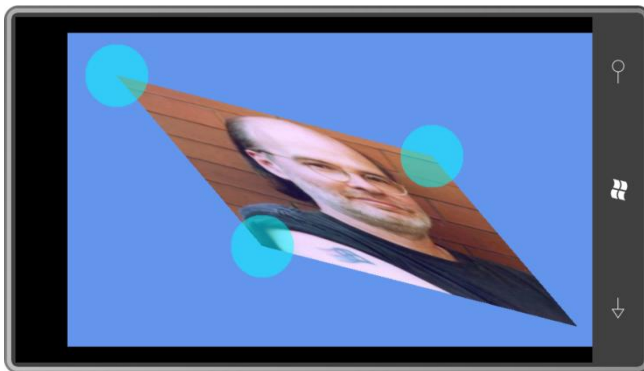
```
    base.Draw(gameTime);
}
```

Теперь применим компонент *Dragger* при рассмотрении намного более сложной трансформации.

Аффинные и неаффинные преобразования

Иногда удобно создать трансформацию, которая обеспечивает отображение определенного набора точек в определенном местоположении. Например, рассмотрим приложение, включающее три экземпляра только что рассмотренного нами компонента *Dragger*, и попробуем поперетягивать углы *Texture2D* по экрану произвольным образом:

Данное приложение использует аффинное преобразование. Это означает, что прямоугольники всегда отображаются в параллелограммы. Четвертый угол недоступен для перетягивания, потому что всегда определяется тремя другими.



Нельзя просто выбрать любые три точки. Ничего не получится, если попытаться сделать внутренний угол больше 180°.

Опишем все это математически. Для начала проще будет принять, что исходное изображение, которые мы будем впоследствии трансформировать, имеет ширину 1 пиксел и высоту тоже 1 пиксел. Мы хотим получить преобразование, обеспечивающее следующие отображения трех углов изображения в три произвольные точки:

$(0, \mathbf{0}) \rightarrow (\boldsymbol{x_0}, \boldsymbol{y_0})$

$(1, \mathbf{0}) \rightarrow (\boldsymbol{x_1}, \boldsymbol{y_1})$

$(0, \mathbf{1}) \rightarrow (\boldsymbol{x_2}, \boldsymbol{y_2})$

Это верхний левый, верхний правый и нижний левый углы, соответственно. Применяя поля объекта *Matrix*, определенного в XNA, получаем такие уравнения преобразования:

$$\boldsymbol{x}' = \mathrm{M}\mathbf{11} \cdot \boldsymbol{x} + \mathrm{M}\mathbf{21} \cdot \boldsymbol{y} + \mathrm{M}\mathbf{41}$$

$$\boldsymbol{y}' = \mathrm{M}\mathbf{12} \cdot \boldsymbol{x} + \mathrm{M}\mathbf{22} \cdot \boldsymbol{y} + \mathrm{M}\mathbf{42}$$

Не сложно применить это преобразование к точкам (0, 0), (1, 0) и (0, 1) и найти элементы матрицы:

$$\mathrm{M11} = x_1 - x_0$$

$$\mathrm{M12} = y_1 - y_0$$

$$\mathrm{M21} = x_2 - x_0$$

$$\mathrm{M22} = y_2 - y_0$$

$$\mathrm{M41} = x_0$$

$$\mathrm{M42} = y_0$$

Статический класс *MatrixHelper* (Вспомогательный класс для работы с матрицей) из библиотеки Petzold.Phone.Xna включает метод *ComputeAffineTransform* (Вычислить аффинное преобразование), который создает объект *Matrix* на основании следующих уравнений:

Проект XNA: Petzold.Phone.Xna Файл: MatrixHelper.cs (фрагмент)

```
static Matrix ComputeAffineTransform(Vector2 ptUL, Vector2 ptUR, Vector2 ptLL)
{
    return new Matrix()
    {
        M11 = (ptUR.X - ptUL.X),
        M12 = (ptUR.Y - ptUL.Y),
        M21 = (ptLL.X - ptUL.X),
        M22 = (ptLL.Y - ptUL.Y),
        M33 = 1,
        M41 = ptUL.X,
        M42 = ptUL.Y,
        M44 = 1
    };
}
```

Этот метод не является открытым, потому что сам по себе он не представляет особой ценности. Он не очень полезен, потому что в используемых им уравнениях за основу взята трансформация изображения один пиксел шириной и один пиксел высотой. Но обратите внимание, что в коде полям *M33* и *M44* задается значение 1. Это не происходит автоматически и необходимо для обеспечения корректной работы матрицы.

Чтобы вычислить *Matrix* для аффинного преобразования, применяемого к объекту конкретного размера, намного полезнее следующий открытый метод:

Проект XNA: Petzold.Phone.Xna Файл: MatrixHelper.cs (фрагмент)

```
public static Matrix ComputeMatrix(Vector2 size, Vector2 ptUL, Vector2 ptUR, Vector2
ptLL)
{
     // Трансформация масштабирования
     Matrix S = Matrix.CreateScale(1 / size.X, 1 / size.Y, 1);
```

```
     // Аффинное преобразование
     Matrix A = ComputeAffineTransform(ptUL, ptUR, ptLL);

    // Произведение двух трансформаций
    return S * A;
}
```

Первая трансформация масштабирует объект, уменьшая его размер до 1×1. После этого к нему применяется вычисленное аффинное преобразование.

Два приведенные выше снимка экрана относятся к проекту AffineTransform (Аффинное преобразование). В нем перегруженный метод *Initialize* создает три экземпляра компонента *Dragger*, задает обработчик события *PositionChanged* и добавляет этот компонент в коллекцию *Components*:

Проект XNA: AffineTransform Файл: Game1.cs (фрагмент)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D texture;
    Matrix matrix = Matrix.Identity;
    Dragger draggerUL, draggerUR, draggerLL;

    …

    protected override void Initialize()
    {
        draggerUL = new Dragger(this);
        draggerUL.PositionChanged += OnDraggerPositionChanged;
        this.Components.Add(draggerUL);

        draggerUR = new Dragger(this);
        draggerUR.PositionChanged += OnDraggerPositionChanged;
        this.Components.Add(draggerUR);

        draggerLL = new Dragger(this);
        draggerLL.PositionChanged += OnDraggerPositionChanged;
        this.Components.Add(draggerLL);

        base.Initialize();
    }
    …
}
```

Не забывайте добавлять компоненты в коллекцию *Components* класса *Game*!

Перегруженный *LoadContent* отвечает за загрузку изображения, которое будет трансформироваться, и инициализацию свойств *Position* трех компонентов *Dragger* в трех углах изображения:

Проект XNA: AffineTransform Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Viewport viewport = this.GraphicsDevice.Viewport;
    texture = this.Content.Load<Texture2D>("PetzoldTattoo");
```

```
    draggerUL.Position = new Vector2((viewport.Width - texture.Width) / 2,
                                     (viewport.Height - texture.Height) / 2);

    draggerUR.Position = draggerUL.Position + new Vector2(texture.Width, 0);
    draggerLL.Position = draggerUL.Position + new Vector2(0, texture.Height);

    OnDraggerPositionChanged(null, EventArgs.Empty);
}
```

Dragger формирует событие *PositionChanged*, только если компонент на самом деле перетягивается пользователем. Поэтому метод *LoadContent* завершается моделированием события *PositionChanged*, которое обеспечивает вычисление исходного *Matrix* на основании размера *Texture2D* и исходных координат компонентов *Dragger*:

Проект XNA: AffineTransform Файл: Game1.cs (фрагмент)

```
void OnDraggerPositionChanged(object sender, EventArgs args)
{
    matrix = MatrixHelper.ComputeMatrix(new Vector2(texture.Width, texture.Height),
                                        draggerUL.Position,
                                        draggerUR.Position,
                                        draggerLL.Position);
}
```

Данное приложение не предполагает никакого другого сенсорного ввода, кроме посредством компонентов *Dragger*. *Dragger* реализует интерфейс *IProcessTouch*, поэтому приложение направляет сенсорный ввод в компоненты *Dragger*. Компоненты *Dragger* отвечают на это собственным перемещением и заданием новых значений свойствам *Position*, что обусловливает формирование событий *PositionChanged*.

Проект XNA: AffineTransform Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    TouchCollection touches = TouchPanel.GetState();

    foreach (TouchLocation touch in touches)
    {
        bool touchHandled = false;

        foreach (GameComponent component in this.Components)
        {
            if (component is IProcessTouch &&
                (component as IProcessTouch).ProcessTouch(touch))
            {
                touchHandled = true;
                break;
            }
        }

        if (touchHandled == true)
            continue;
    }

    base.Update(gameTime);
}
```

Приложение может не задавать обработчики событий *PositionChanged* компонентов *Dragger* и вместо этого проводить сверку значений свойств *Position* при каждом вызове *Update* и пересчитывать *Matrix* на основании этих значений. Однако намного более эффективно пересчитывать *Matrix* только по фактическому изменению значения одного из свойств *Position*.

Перегруженный *Draw* использует *Matrix* для отображения текстуры:

Проект XNA: AffineTransform Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin(SpriteSortMode.Immediate, null, null, null, null, null,
matrix);
    spriteBatch.Draw(texture, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Для AffineTransform необходимо устранить возможность превышения значения 180° для любого из внутренних углов (иначе говоря, изображение должно оставаться выпуклым). Аффинные преобразования могут представлять привычные операции переноса, масштабирования, вращения и сдвига, но они никогда не приводят к превращению квадрата в нечто более экзотичное, чем параллелограмм.

Неаффинные преобразования намного чаще применяются в трехмерной графике, чем в двухмерной. В 3D графике неаффинные преобразования необходимы для реализации эффектов перспективы. Длинная прямая пустынная дорога в 3D-мире должна сужаться, убегая вдаль, точно так же как и в реальном мире. Хотя мы знаем, что стороны дороги остаются параллельными, визуально на бесконечном удалении они сходятся. Этот эффект сужения является характеристикой неаффинных преобразований.

Полная матрица трансформации для трехмерной координатной точки выглядит следующим образом:

$$|\boldsymbol{x} \quad \boldsymbol{y} \quad \boldsymbol{z} \quad \boldsymbol{1}| \times \begin{vmatrix} \boldsymbol{M11} & \boldsymbol{M12} & \boldsymbol{M13} & \boldsymbol{M14} \\ \boldsymbol{M21} & \boldsymbol{M22} & \boldsymbol{M23} & \boldsymbol{M24} \\ \boldsymbol{M31} & \boldsymbol{M32} & \boldsymbol{M33} & \boldsymbol{M34} \\ \boldsymbol{M41} & \boldsymbol{M42} & \boldsymbol{M43} & \boldsymbol{M44} \end{vmatrix} = |\boldsymbol{x'} \quad \boldsymbol{y'} \quad \boldsymbol{z'} \quad \boldsymbol{w'}|$$

Поскольку *x*, *y* и *z* – последние буквы алфавита, для представления четвертого измерения выбрана буква *w*. Прежде всего, для умножения на матрицу 4×4 трехмерную координатную точку необходимо представить как точку в четырехмерной системе координат. Следующие уравнения являются результатом умножения матриц:

$$\boldsymbol{x'} = \mathrm{M}\boldsymbol{11} \cdot \boldsymbol{x} + \mathrm{M}\boldsymbol{21} \cdot \boldsymbol{y} + \mathrm{M}\boldsymbol{31} \cdot \boldsymbol{z} + \mathrm{M}\boldsymbol{41}$$

$$\boldsymbol{y'} = \mathrm{M}\boldsymbol{12} \cdot \boldsymbol{x} + \mathrm{M}\boldsymbol{22} \cdot \boldsymbol{y} + \mathrm{M}\boldsymbol{32} \cdot \boldsymbol{z} + \mathrm{M}\boldsymbol{42}$$

$$\boldsymbol{z'} = \mathrm{M}\boldsymbol{13} \cdot \boldsymbol{x} + \mathrm{M}\boldsymbol{23} \cdot \boldsymbol{y} + \mathrm{M}\boldsymbol{33} \cdot \boldsymbol{z} + \mathrm{M}\boldsymbol{43}$$

$$\boldsymbol{w'} = \mathrm{M}\boldsymbol{14} \cdot \boldsymbol{x} + \mathrm{M}\boldsymbol{24} \cdot \boldsymbol{y} + \mathrm{M}\boldsymbol{34} \cdot \boldsymbol{z} + \mathrm{M}\boldsymbol{44}$$

Для аффинного преобразования *M14*, *M24* и *M34* равны нулю, и *M44* равно 1, поэтому *w′* равна 1, и все преобразование происходит в четырехмерном пространстве. Для неаффинных преобразований *w′* не равна 1, и чтобы спроецировать четырехмерное пространство назад в

трехмерное, из четырехмерной точки (x′, y′, z′, w′) необходимо получить трехмерную точку следующим образом:

$$\left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}\right)$$

Сужение здесь обусловлено делением. Если *M14* имеет положительное значение, к примеру, тогда *w*″ будет расти при увеличении x, и графический объект будет уменьшаться по мере увеличения *x*.

Что происходит, если *w*′ принимает нулевое значение? Процесс вступает в *неаффинную* фазу: координаты могут принимать бесконечные значения. Обычно бесконечные объекты стараются держать вне поля видимости, потому что они зрительно искривляют экран.

Хотя неаффинные преобразования являются неотъемлемой частью трехмерной графики, я даже не был уверен, поддерживает ли *SpriteBatch* двухмерные неаффинные преобразования до тех пор, пока не попробовал их. И был приятно удивлен, когда XNA сказал: «Нет проблем!» А это означает, что неаффинные преобразования могут применяться при разработке двухмерной графики для моделирования эффектов перспективы.

Неаффинное преобразование в двухмерном пространстве может превратить квадрат в простой выпуклый тетрагон – фигуру с четырьмя сторонами, в которой стороны пересекаются только в углах, и ни один из внутренних углов не превышает 180°. Рассмотрим пример:

А тут я выгляжу ну очень умным:

Данное приложение называется NonAffineTransform (Неаффинное преобразование) и очень похоже на AffineTransform. Отличие лишь в наличии четвертого компонента и использовании несколько более сложного метода класса *MatrixHelper* библиотеки Petzold.Phone.Xna. Пользователь получает довольно большую свободу по перемещению углов изображения, и если только он не пытается получить вогнутый четырехугольник, изображение будет растягиваться соответственно.

Опять же математическое описание строится на предположении о том, что высота и ширина исходного изображения, которое мы будем трансформировать, составляют 1 пиксел. Теперь нам необходимо получить преобразование, которое обеспечивало бы проецирование четырех углов квадрата в четыре произвольные точки:

$(0, \mathbf{0}) \rightarrow (\boldsymbol{x_0}, \boldsymbol{y_0})$

$(1, \mathbf{0}) \rightarrow (\boldsymbol{x_1}, \boldsymbol{y_1})$

$(0, \mathbf{1}) \rightarrow (\boldsymbol{x_2}, \boldsymbol{y_2})$

$(1, \mathbf{1}) \rightarrow (\boldsymbol{x_3}, \boldsymbol{y_3})$

Необходимое нам преобразование будет намного проще получить, если разбить его на два преобразования:

$(0, \mathbf{0}) \rightarrow (0,0) \rightarrow (\boldsymbol{x_0}, \boldsymbol{y_0})$

$(1, \mathbf{0}) \rightarrow (1,0) \rightarrow (\boldsymbol{x_1}, \boldsymbol{y_1})$

$(0, \mathbf{1}) \rightarrow (0,1) \rightarrow (\boldsymbol{x_2}, \boldsymbol{y_2})$

$(1, \mathbf{1}) \rightarrow (a, b) \rightarrow (\boldsymbol{x_3}, \boldsymbol{y_3})$

Первое преобразование является неаффинным, я назову его **B**. Второе преобразование я принудительно сделаю аффинным и назову его **A** (от «аффинный»). Составное преобразование – **B**×**A**. Задача – выполнить двойное преобразования для точки (*a*, *b*).

Я уже определил аффинное преобразование, но хочу, чтобы это аффинное преобразование обеспечивало проецирование точки (*a*, *b*) в точку (x_3, y_3). Что это за точка (*a*, *b*)? Если применить к точке (*a*, *b*) аффинное преобразование и разрешить его для *a* и *b*, мы получим:

$$a = \frac{M22 \cdot x_3 - M21 \cdot y_3 + M21 \cdot M42 - M22 \cdot M41}{M11 \cdot M22 - M12 \cdot M21}$$

$$b = \frac{M11 \cdot y_3 - M12 \cdot x_3 + M12 \cdot M41 - M11 \cdot M42}{M11 \cdot M22 - M12 \cdot M21}$$

Теперь обратимся к неаффинному преобразованию, которое должно обеспечивать такое проецирование:

$(0, \mathbf{0}) \rightarrow (0,0)$

$(1, \mathbf{0}) \rightarrow (1,0)$

$(0, \mathbf{1}) \rightarrow (0,1)$

$(1, \mathbf{1}) \rightarrow (\boldsymbol{a}, \boldsymbol{b})$

Обобщенные уравнения для неаффинного преобразования в двухмерной плоскости (с использованием имен полей структуры *Matrix* и включением деления на *w'*) имеют следующий вид:

$$x' = \frac{M11 \cdot x + M21 \cdot y + M41}{M14 \cdot x + M24 \cdot y + M44}$$

$$y' = \frac{M12 \cdot x + M22 \cdot y + M42}{M14 \cdot x + M24 \cdot y + M44}$$

Точка (0, 0) проецируется в точку (0, 0). Это говорит нам о том, что *M41* и *M42* равны нулю, и *M44* не равно нулю. Рискнем и примем *M44* равным 1.

Точка (1, 0) проецируется в (1, 0), что говорит о том, что *M12* равно нулю, и *M14* = *M11* – 1.

Точка (0, 1) проецируется в (0, 1), что говорит о том, что *M21* равно нулю, и *M24* = *M22* – 1.

Точка (1, 1) проецируется в (*a*, *b*), что требует небольших вычислений:

$$M11 = \frac{a}{a+b-1}$$

$$M22 = \frac{b}{a+b-1}$$

И *a* и *b* уже вычислены для аффинного преобразования.

Все эти вычисления включены во второй статический метод *MatrixHelper.ComputeMatrix* библиотеки Petzold.Phone.Xna:

Проект XNA: Petzold.Phone.Xna Файл: MatrixHelper.cs

```
public static Matrix ComputeMatrix(Vector2 size, Vector2 ptUL, Vector2 ptUR,
                                                 Vector2 ptLL, Vector2 ptLR)
{
    // Трансформация масштабирования
    Matrix S = Matrix.CreateScale(1 / size.X, 1 / size.Y, 1);

    // Аффинное преобразование
    Matrix A = ComputeAffineTransform(ptUL, ptUR, ptLL);

    // Неаффинное преобразование
    Matrix B = new Matrix();
    float den = A.M11 * A.M22 - A.M12 * A.M21;
    float a = (A.M22 * ptLR.X - A.M21 * ptLR.Y +
               A.M21 * A.M42 - A.M22 * A.M41) / den;

    float b = (A.M11 * ptLR.Y - A.M12 * ptLR.X +
               A.M12 * A.M41 - A.M11 * A.M42) / den;

    B.M11 = a / (a + b - 1);
    B.M22 = b / (a + b - 1);
    B.M33 = 1;
    B.M14 = B.M11 - 1;
    B.M24 = B.M22 - 1;
    B.M44 = 1;

    // Произведение трех трансформаций
    return S * B * A;
}
```

Я не привожу приложение NonAffineTransform здесь, потому что оно во многом повторяет AffineTransform, лишь с добавлением четвертого компонента *Dragger*, свойство *Position* которого передается во второй метод *ComputeMatrix*.

Самое большое отличие этого нового приложения в том, что неаффинные преобразования намного более забавные!

Глава 23

Использование сенсорного ввода в игровых приложениях

Часто при изучении новой среды разработки мы обзаводимся набором техник, которые не имеют никакого отношения к навыкам, необходимым для создания законченного приложения. Данная глава призвана компенсировать эту проблему, представив два весьма типовых приложения для телефона: PhingerPaint и PhreeCell. Первое – это простое приложение для рисования; второе – версия классической игры пасьянс. Третье приложение, SpinPaint, использует часть кода PhingerPaint, но обеспечивает совершенно другую функциональность.

Все эти приложения используют компоненты, обрабатывают сенсорный ввод с различной степенью сложности и динамически меняют объекты *Texture2D*. Несомненно, данные приложения не дотягивают до уровня коммерческих программных продуктов, но позволяют получить лучшее представление о том, как выглядит «реальная программа».

Еще больше компонентов игры

Когда мы впервые рассматривали динамические объекты *Texture2D* в главе 21, я описывал некоторые простые приложения для рисования посредством сенсорного ввода. Как можно понять из данного снимка экрана, приложение PhingerPaint несколько сложнее:

PhingerPaint включает в общей сложности 14 экземпляров двух классов, *ColorBlock* и *Button*, которые наследуются от *DrawableGameComponent*. Чтобы выбрать цвет для рисования, пользователь касается одного из цветных квадратиков вверху экрана. Внизу экрана предусмотрены кнопки для полной очистки экрана или для сохранения рисунка в специальном зарезервированном для приложений альбоме библиотеки фотографий телефона под названием Saved Pictures. Из библиотеки фотографий изображение может быть отправлено по электронной почте или перенесено на ПК. (Нельзя лишь продолжить работу

над изображением, повторно открыв его в другом сеансе работы. Возможно, когда-нибудь я добавлю эту функциональность.)

Поведение *Button* практически ничем не отличается от традиционной графической кнопки. В обычном состоянии он отображает белый текст и белую рамку. Когда пользователь касается поверхности кнопки, цвета меняются на инверсные, и на кнопке уже отображается черный текст на белом фоне. Если сдвинуть палец с кнопки, цвета возвращаются к исходным, но кнопка продолжает отслеживать это касание. Верните палец на кнопку, и цвета снова поменяются. При снятии касания кнопка формирует событие *Click*.

Я собираюсь использовать этот *Button* в нескольких приложениях, поэтому сделал его частью библиотеки Petzold.Phone.Xna. В данном листинге представлено начало этого класса с описаниями закрытых полей, открытого события, конструктора и открытых свойств:

Проект XNA: Petzold.Phone.Xna Файл: Button.cs (фрагмент)

```
public class Button : DrawableGameComponent, IProcessTouch
{
    SpriteBatch spriteBatch;
    Texture2D tinyTexture;
    Vector2 textPosition;
    bool isPressed;
    int? touchId = null;

    public event EventHandler Click;

    public Button(Game game, string text)
        : base(game)
    {
        Text = text;
    }

    public Rectangle Destination { set; get; }
    public SpriteFont SpriteFont { set; get; }
    public string Text { set; get; }
    …
}
```

Как правило, конструктор компонента игры включает аргумент типа *Game*, который указывает на родителя этого компонента, и из которого базовый класс *GameComponent* получает данные о *GraphicsDevice*. Для текста кнопки я добавил в конструктор аргумент типа *string*, но этот текст также может быть задан позднее через открытое свойство *Text*.

Я решил, что шрифт *Button* и крайне важное свойство *Destination* (Место назначения), которое определяет месторасположения и размер *Button* относительно экрана, будут задаваться в родительском производном от *Game* классе.

Перегруженный метод *LoadContent* в компоненте игры выполняет те же функции, что и в классе игры. Класс *Button* создает крошечный белый *Texture2D* размером 1×1 пиксел для отображения рамки кнопки и фона, инверсного по отношению к рамке цвета.

Проект XNA: Petzold.Phone.Xna Файл: Button.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(this.GraphicsDevice);

    tinyTexture = new Texture2D(this.GraphicsDevice, 1, 1);
    tinyTexture.SetData<uint>(new uint[] { Color.White.PackedValue });
```

```
    base.LoadContent();
}
```

Открытые свойства *SpriteFont* (Шрифт спрайта) и *Destination* класса *Button* могут быть заданы в любой момент после создания *Button*. Поэтому доступ к этим сведениям для определения размера и местоположения текста выполняет метод *Update*:

Проект XNA: Petzold.Phone.Xna Файл: Button.cs (фрагмент)

```
public override void Update(GameTime gameTime)
{
    if (SpriteFont != null && !String.IsNullOrEmpty(Text))
    {
        Vector2 textSize = SpriteFont.MeasureString(Text);
        textPosition =
            new Vector2((int)(Destination.Left + (Destination.Width - textSize.X) /
2),
                        (int)(Destination.Top + (Destination.Height - textSize.Y) /
2));
    }
    base.Update(gameTime);
}
```

Класс *Button* реализует интерфейс *IProcessTouch*, который был рассмотрен в предыдущей главе. Это означает, что у класса есть метод *ProcessTouch*, который вызывается из класса *Game1* для каждого объекта *TouchLocation*. Когда палец впервые касается экрана, *ProcessTouch* проверяет, попадает ли точка касания в прямоугольник, определенный *Destination*. Если попадает, он сохраняет идентификатор касания и, по сути, владеет им до снятия касания.

Проект XNA: Petzold.Phone.Xna Файл: Button.cs (фрагмент)

```
public bool ProcessTouch(TouchLocation touch)
{
    bool touchHandled = false;
    bool isInside = Destination.Contains((int)touch.Position.X,
                                         (int)touch.Position.Y);
    switch (touch.State)
    {
        case TouchLocationState.Pressed:
            if (isInside)
            {
                isPressed = true;
                touchId = touch.Id;
                touchHandled = true;
            }
            break;

        case TouchLocationState.Moved:
            if (touchId.HasValue && touchId.Value == touch.Id)
            {
                isPressed = isInside;
                touchHandled = true;
            }
            break;

        case TouchLocationState.Released:
            if (touchId.HasValue && touchId.Value == touch.Id)
            {
                if (isInside && Click != null)
                    Click(this, EventArgs.Empty);
```

```
                touchId = null;
                isPressed = false;
                touchHandled = true;
            }
            break;
    }
    return touchHandled;
}
```

Если снятие касания происходит в момент, когда точка касания располагается внутри прямоугольника *Destination*, класс *Button* формирует событие *Click*.

Перегруженный *Draw* отрисовывает кнопку. По сути, это рамка, включающая белый прямоугольник, поверх которого располагается несколько меньший черный прямоугольник со строкой текста:

Проект XNA: Petzold.Phone.Xna Файл: Button.cs (фрагмент)

```
public override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();

    if (isPressed)
    {
        // Отрисовываем инвертированный фон
        spriteBatch.Draw(tinyTexture, Destination, Color.White);
    }
    else
    {
        // Отрисовываем рамку и фон кнопки
        Rectangle rect = Destination;

        spriteBatch.Draw(tinyTexture, rect, Color.White);
        rect.Inflate(-3, -3);
        spriteBatch.Draw(tinyTexture, rect, Color.Black);
    }

    // Отрисовываем текст кнопки
    if (SpriteFont != null && !String.IsNullOrEmpty(Text))
        spriteBatch.DrawString(SpriteFont, Text, textPosition,
                               isPressed ? Color.Black : Color.White);

    spriteBatch.End();

    base.Draw(gameTime);
}
```

С другой стороны, *ColorBlock* (Блок цвета) является частью приложения PhingerPaint и не реализует интерфейс *IProcessTouch*. Рассмотрим код этого класса полностью:

Проект XNA: PhingerPaint Файл: ColorBlock.cs (полностью)

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input.Touch;

namespace PhingerPaint
{
    public class ColorBlock : DrawableGameComponent
    {
        SpriteBatch spriteBatch;
```

```
        Texture2D block;

        public ColorBlock(Game game) : base(game)
        {
        }

        public Color Color { set; get; }
        public Rectangle Destination { set; get; }
        public bool IsSelected { set; get; }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(this.GraphicsDevice);
            block = new Texture2D(this.GraphicsDevice, 1, 1);
            block.SetData<uint>(new uint[] { Color.White.PackedValue });

            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            Rectangle rect = Destination;

            spriteBatch.Begin();
            spriteBatch.Draw(block, rect, IsSelected ? Color.White :
Color.DarkGray);
            rect.Inflate(-6, -6);
            spriteBatch.Draw(block, rect, Color);
            spriteBatch.End();

            base.Draw(gameTime);
        }
    }
}
```

За внешний вид *ColorBlock* отвечают три открытых свойства: *Color*, *Destination* и *IsSelected* (Выбран). Обратите внимание, что в методе *LoadContent* создается *Texture2D* размером один пиксел. Этот объект *block* (блок) в методе *Draw* отрисовывается дважды. Сначала он отрисовывается соответственно полным размерам прямоугольника *Destination* темно-серого или белого цвета, в зависимости от значения *IsSelected*. Затем он уменьшается на 6 пикселов со всех сторон и отрисовывается снова с использованием свойства *Color*.

Холст PhingerPaint

Компоненты, создаваемые PhingerPaint, а также некоторые другие необходимые данные хранятся как поля:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
```

```
    Texture2D canvas;
    Vector2 canvasSize;
    Vector2 canvasPosition;
    uint[] pixels;
    List<float> xCollection = new List<float>();

    Button clearButton, saveButton;
    string filename;

    List<ColorBlock> colorBlocks = new List<ColorBlock>();
    Color drawingColor = Color.Blue;
    int? touchIdToIgnore;
    …
}
```

List сохраняет 12 компонентов *ColorBlock*; *drawingColor* (Цвет рисования) – выбранный в настоящий момент цвет. Роль главного холста исполняет, конечно же, объект *Texture2D* под именем *canvas*, значения пикселов этой текстуры хранятся в массиве *pixels*. Объект *xCollection* многократно используется при работе с классом *RoundCappedLine*, который был рассмотрен в главе 21.

В конструкторе для заднего буфера устанавливается портретный режим отображения, но высота задается равной 768, а не 800 пикселам. Таким образом, остается достаточно места для строки состояния, и задний буфер может отображаться в полном размере:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    // Задаем портретный режим, но оставляем достаточно места для строки состояния
    graphics.PreferredBackBufferWidth = 480;
    graphics.PreferredBackBufferHeight = 768;
}
```

Перегруженный метод *Initialize* отвечает за создание компонентов *Button* и *ColorBlack*, частично их инициализирует и добавляет в коллекцию *Components* класса *Game*. Это гарантирует вызов их собственных методов *Initialize*, *LoadContent*, *Update* и *Draw*.

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    // Создаем компоненты Button
    clearButton = new Button(this, "clear");
    clearButton.Click += OnClearButtonClick;
    this.Components.Add(clearButton);

    saveButton = new Button(this, "save");
    saveButton.Click += OnSaveButtonClick;
    this.Components.Add(saveButton);

    // Создаем компоненты ColorBlock
    Color[] colors = { Color.Red, Color.Green, Color.Blue,
                       Color.Cyan, Color.Magenta, Color.Yellow,
```

```
                       Color.Black, new Color(0.2f, 0.2f, 0.2f),
                                    new Color(0.4f, 0.4f, 0.4f),
                                    new Color(0.6f, 0.6f, 0.6f),
                                    new Color(0.8f, 0.8f, 0.8f), Color.White };

    foreach (Color clr in colors)
    {
        ColorBlock colorBlock = new ColorBlock(this);
        colorBlock.Color = clr;
        colorBlocks.Add(colorBlock);
        this.Components.Add(colorBlock);
    }
    base.Initialize();
}
```

Вся остальная инициализация компонентов осуществляется в ходе выполнения перегруженного метода *LoadContent*, когда уже может быть загружен шрифт для компонентов *Button*. Кажется несколько странным явно задать размер заднего буфера в конструкторе и при этом производить более абстрактные вычисления размеров в методе *LoadContent*, но желательно делать код максимально обобщенным и гибким.

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Rectangle clientBounds = this.GraphicsDevice.Viewport.Bounds;
    SpriteFont segoe14 = this.Content.Load<SpriteFont>("Segoe14");

    // Настраиваем компоненты Button
    clearButton.SpriteFont = segoe14;
    saveButton.SpriteFont = segoe14;

    Vector2 textSize = segoe14.MeasureString(clearButton.Text);
    int buttonWidth = (int)(2 * textSize.X);
    int buttonHeight = (int)(1.5 * textSize.Y);

    clearButton.Destination =
        new Rectangle(clientBounds.Left + 20,
                      clientBounds.Bottom - 2 - buttonHeight,
                      buttonWidth, buttonHeight);

    saveButton.Destination =
        new Rectangle(clientBounds.Right - 20 - buttonWidth,
                      clientBounds.Bottom - 2 - buttonHeight,
                      buttonWidth, buttonHeight);

    int colorBlockSize = clientBounds.Width / (colorBlocks.Count / 2) - 2;
    int xColorBlock = 2;
    int yColorBlock = 2;

    foreach (ColorBlock colorBlock in colorBlocks)
    {
        colorBlock.Destination = new Rectangle(xColorBlock, yColorBlock,
                                               colorBlockSize, colorBlockSize);
        xColorBlock += colorBlockSize + 2;

        if (xColorBlock + colorBlockSize > clientBounds.Width)
        {
            xColorBlock = 2;
            yColorBlock += colorBlockSize + 2;
        }
    }
```

```
    canvasPosition = new Vector2(0, 2 * colorBlockSize + 6);
    canvasSize = new Vector2(clientBounds.Width,
                             clientBounds.Height - canvasPosition.Y
                                                 - buttonHeight - 4);
}
```

Метод *LoadContent* завершается вычислением местоположения и размера *Texture2D*, используемого в качестве холста. Но не делает последнего шага *LoadContent* к фактическому созданию этого *Texture2D*, потому что за методом *LoadContent* может последовать вызов перегруженного *OnActivated*, который свидетельствует о том, что приложение либо запускается впервые, либо возвращается из состояния захоронения.

Для приложения PhingerPaint важно реализовать захоронение, потому что пользователи не любят, когда их творческие порывы исчезают с экрана. Поэтому перегруженный *OnDeactivated* сохраняет изображение в *PhoneApplicationService* в формате PNG, и перегруженный *OnActivated* извлекает его оттуда. Я выбрал формат PNG, потому что это формат сжатия без потерь, и мне показалось, что изображение должно восстанавливаться точно в его исходное состояние.

Чтобы немного упростить процесс сохранения и загрузки объекта *Texture2D*, я использовал методы класса *Texture2DExtensions* из библиотеки Petzold.Phone.Xna, которые были рассмотрены в предыдущей главе. Метод *OnActivated* вызывает *LoadFromPhoneService* (Загрузить из сервиса приложения для телефона) для получения сохраненного *Texture2D*. И только если этот объект недоступен, создается и очищается новый *Texture2D*.

Для работы с классом *PhoneApplicationService* необходимо указать ссылки на сборки System.Windows и Microsoft.Phone, а также включить директиву *using* для *Microosft.Phone.Shell*.

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void OnActivated(object sender, EventArgs args)
{
    // Восстанавливаем после захоронения
    bool newlyCreated = false;
    canvas = Texture2DExtensions.LoadFromPhoneServiceState(this.GraphicsDevice,
                                                            "canvas");
    if (canvas == null)
    {
        // В противном случае создаем новый Texture2D
        canvas = new Texture2D(this.GraphicsDevice, (int)canvasSize.X,
                                                    (int)canvasSize.Y);
        newlyCreated = true;
    }

    // Создаем массив pixels
    pixels = new uint[canvas.Width * canvas.Height];
    canvas.GetData<uint>(pixels);

    if (newlyCreated)
        ClearPixelArray();

    // Из State получаем цвет рисунка, инициализируем выбранный ColorBlock
    if (PhoneApplicationService.Current.State.ContainsKey("color"))
        drawingColor = (Color)PhoneApplicationService.Current.State["color"];

    foreach (ColorBlock colorBlock in colorBlocks)
        colorBlock.IsSelected = colorBlock.Color == drawingColor;

    base.OnActivated(sender, args);
}
```

Перегруженный метод *OnDeactivated* сохраняет *Texture2D* с помощью метода расширения *SaveToPhoneServiceState* (Сохранить в состояние сервиса приложения для телефона):

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void OnDeactivated(object sender, EventArgs args)
{
    PhoneApplicationService.Current.State["color"] = drawingColor;
    canvas.SaveToPhoneServiceState("canvas");
    base.OnDeactivated(sender, args);
}
```

Если происходит запуск приложения, *OnActivated* вызывает метод *ClearPixelArray* (Очистить массив Pixel):

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
void ClearPixelArray()
{
    for (int y = 0; y < canvas.Height; y++)
        for (int x = 0; x < canvas.Width; x++)
        {
            pixels[x + canvas.Width * y] = Color.GhostWhite.PackedValue;
        }

    canvas.SetData<uint>(pixels);
}

void OnClearButtonClick(object sender, EventArgs e)
{
    ClearPixelArray();
}
```

Вы заметите, что обработчик событий *Click* для «очистки» *Button* тоже вызывает этот метод. Как помните, класс *Button* формирует событие *Click* на основании сенсорного ввода, и *Button* получает сенсорный ввод, когда родительский класс *Game* вызывает метод *ProcessTouch* из собственного перегруженного *Update*. Это означает, что данный метод *OnClearButtonClick* (При щелчку кнопки очистить) фактически вызывается при вызове перегруженного *Update* этого класса.

Когда пользователь нажимает кнопку с надписью «save», приложение должно вывести на экран некоторое диалоговое окно для ввода имени файла. Приложение на XNA может принимать ввод с клавиатуры двумя способами: применяя низкоуровневый подход с использованием класса *Keyboard* (Клавиатура), и высокоуровневый подход посредством вызова метода *Guide.BeginShowKeyboardInput* (Начать отображение ввода с клавиатуры) из пространства имен *Microsoft.Xna.Framework.GamerServices* (Игровые сервисы). Я выбрал высокоуровневый подход. Метод *Guide.BeginShowKeyboardInput* требует некоторые исходные данные и функцию обратного вызова, что позволяет ему создавать уникальное имя файла на основании текущих даты и времени:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
void OnSaveButtonClick(object sender, EventArgs e)
{
    DateTime dt = DateTime.Now;
    filename =
        String.Format("PhingerPaint-{0:D2}-{1:D2}-{2:D2}-{3:D2}-{4:D2}-{5:D2}",
```

```
                      dt.Year % 100, dt.Month, dt.Day, dt.Hour, dt.Minute,
dt.Second);

    Guide.BeginShowKeyboardInput(PlayerIndex.One, "phinger paint save file",
                                "enter filename:", filename, KeyboardCallback,
null);
}
```

Вызов *Guide.BeginShowKeyboardInput* приводит к вызову метода *OnDeactivated*, после чего на экран выводится следующее:

В этом окне настроены могут быть только текстовые строки заголовков и исходный текст в строке ввода. Данный экран выглядит в портретном режиме намного лучше, чем в альбомном, поскольку в альбомном режиме все заголовки, строка ввода и экранная клавиатура меняют ориентацию, а две кнопки остаются ориентированными по-старому. В итоге окно выглядит очень странно. Одного взгляда на него будет достаточно, чтобы больше никогда не вызывать *Guide.BeginShowKeyboardInput* из приложения, отображающегося в альбомном режиме!

По нажатию кнопки «OK» или «Cancel» приложение повторно активируется, и в *PhingerPaint* вызывается функция обратного вызова:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
void KeyboardCallback(IAsyncResult result)
{
    filename = Guide.EndShowKeyboardInput(result);
}
```

Приложение должно предполагать, что эта функция обратного вызова вызывается асинхронно (как подразумевает аргумент), поэтому здесь не следует делать ничего, кроме вызова *Guide.EndShowKeyboardInput* (Закончить отображение ввода с клавиатуры) и сохранения возвращенного значения в поле. Если пользователь нажимает кнопку «OK», возвращаемым значением является окончательный текст, введенный в строку ввода. Если

пользователь нажимает «Cancel» или кнопку Back, *Guide.EndShowKeyboardInput* возвращает *null*.

Самым подходящим местом обработать это возвращенное значение некоторым образом является следующий вызов перегруженного *Update*:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Если диалоговое окно Save File возвращает значение, сохраняем изображение
    if (!String.IsNullOrEmpty(filename))
    {
        canvas.SaveToPhotoLibrary(filename);
        filename = null;
    }
    …
}
```

Обратите внимание, в коде выполняется проверка того, что поле *filename* (имя файла) не пустое и его значение не *null*, но в конце этому полю опять присваивается значение *null*, что гарантирует однократное сохранение его значения.

SaveToPhotoLibrary (Сохранить в библиотеку фотографий), на самом деле, не метод класса *Texture2D*! Это еще один метод расширения класса *Texture2DExtensions* из библиотеки Petzold.Phone.Xna.

Проект XNA: Petzold.Phone.Xna Файл: Texture2DExtensions.cs (фрагмент)

```
public static void SaveToPhotoLibrary(this Texture2D texture, string filename)
{
    MemoryStream memoryStream = new MemoryStream();
    texture.SaveAsJpeg(memoryStream, texture.Width, texture.Height);
    memoryStream.Position = 0;
    MediaLibrary mediaLibrary = new MediaLibrary();
    mediaLibrary.SavePicture(filename, memoryStream);
    memoryStream.Close();
}
```

Это стандартный код для сохранения *Texture2D* в альбом Saved Pictures библиотеки фотографий телефона. Несмотря на то что PhingerPaint при захоронении сохраняет изображение в формате PNG, изображения, сохраняемые в библиотеке фотографий, должны быть в формате JPEG. Метод *SaveAsJpeg* сохраняет все изображение в *MemoryStream*. После этого положение курсора в *MemoryStream* сбрасывается, и он вместе с именем файла передается в метод *SavePicture* объекта *MediaLibrary*.

Если данное приложение будет развернуто на телефоне, при запуске настольного ПО Zune для обмена данными между Visual Studio и телефоном, этот код сформирует исключение. Zune требует эксклюзивного доступа к библиотеке мультимедиа телефона. Вам придется завершить выполнение приложения Zune и вместо него запустить инструмент WPDTPTConnect: WPDTPTConnect32.exe или WPDTPTConnect64.exe в зависимости от того 32- или 64-разрядная операционная система Windows используется.

Конечно, перегруженный *Update* большей частью посвящен обработке сенсорного ввода. Я решил использовать простой сенсорный ввод, чтобы обеспечить возможность рисования на

холсте несколькими пальцами. *Button* в основном обрабатывает собственный сенсорный ввод с помощью интерфейса *IProcessTouch*, но *ColorBlock* обрабатывается иначе. Метод *Update* в самом классе игры обрабатывает компоненты *ColorBlock* и холст *Texture2D*.

Компоненты *ColorBlock* имеют более примитивную логику, чем *Button*. Простого касания *ColorBlock* достаточно, чтобы этот элемент был выбран, и приложение переключилось к данному цвету. Идентификатор касания сохраняется, и его использование для чего-либо еще не допускается.

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    TouchCollection touches = TouchPanel.GetState();

    foreach (TouchLocation touch in touches)
    {
        // Игнорируем дальнейшие действия касания ColorBlock
        if (touchIdToIgnore.HasValue && touch.Id == touchIdToIgnore.Value)
            continue;

        // Обеспечиваем компонентам Button первоочередное право на касание
        bool touchHandled = false;

        foreach (GameComponent component in this.Components)
            if (component is IProcessTouch &&
                (component as IProcessTouch).ProcessTouch(touch))
            {
                touchHandled = true;
                break;
            }

        if (touchHandled)
            continue;

        // Проверяем на наличие касания ColorBlock
        if (touch.State == TouchLocationState.Pressed)
        {
            Vector2 position = touch.Position;
            ColorBlock newSelectedColorBlock = null;

            foreach (ColorBlock colorBlock in colorBlocks)
            {
                Rectangle rect = colorBlock.Destination;

                if (position.X >= rect.Left && position.X < rect.Right &&
                    position.Y >= rect.Top && position.Y < rect.Bottom)
                {
                    drawingColor = colorBlock.Color;
                    newSelectedColorBlock = colorBlock;
                }
            }

            if (newSelectedColorBlock != null)
            {
                foreach (ColorBlock colorBlock in colorBlocks)
                    colorBlock.IsSelected = colorBlock == newSelectedColorBlock;

                touchIdToIgnore = touch.Id;
            }
            else
            {
                touchIdToIgnore = null;
            }
```

```
        }
        …
    }
    …
}
```

Дальнейшая обработка касания связана непосредственно с рисованием, и для этого требуется лишь проверять значение свойства *State* на равенство *TouchLocationState.Moved*. Это состояние позволяет вызывать метод *TryGetPreviousLocation*, после чего в конструктор класса *RoundCappedLine* из *Petzold.Phone.Xna* могут быть переданы две точки. Таким образом, мы получаем диапазоны пикселов для закрашивания в каждом небольшом фрагменте общего мазка кисти:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Обрабатываем сенсорный ввод
    bool canvasNeedsUpdate = false;
    TouchCollection touches = TouchPanel.GetState();

    foreach (TouchLocation touch in touches)
    {
        …
        // Проверяем на наличие движения рисования
        else if (touch.State == TouchLocationState.Moved)
        {
            TouchLocation prevTouchLocation;
            touch.TryGetPreviousLocation(out prevTouchLocation);

            Vector2 point1 = prevTouchLocation.Position - canvasPosition;
            Vector2 point2 = touch.Position - canvasPosition;

            // Безусловно, надеемся на возвращение touchLocation.Pressure!
            float radius = 12;
            RoundCappedLine line = new RoundCappedLine(point1, point2, radius);

            int yMin = (int)(Math.Min(point1.Y, point2.Y) - radius);
            int yMax = (int)(Math.Max(point1.Y, point2.Y) + radius);

            yMin = Math.Max(0, Math.Min(canvas.Height, yMin));
            yMax = Math.Max(0, Math.Min(canvas.Height, yMax));

            for (int y = yMin; y < yMax; y++)
            {
                xCollection.Clear();
                line.GetAllX(y, xCollection);

                if (xCollection.Count == 2)
                {
                    int xMin = (int)(Math.Min(xCollection[0], xCollection[1]) +
0.5f);
                    int xMax = (int)(Math.Max(xCollection[0], xCollection[1]) +
0.5f);

                    xMin = Math.Max(0, Math.Min(canvas.Width, xMin));
                    xMax = Math.Max(0, Math.Min(canvas.Width, xMax));

                    for (int x = xMin; x < xMax; x++)
                    {
                        pixels[y * canvas.Width + x] = drawingColor.PackedValue;
                    }
                    canvasNeedsUpdate = true;
                }
```

```
            }
        }
    }

    if (canvasNeedsUpdate)
        canvas.SetData<uint>(pixels);

    base.Update(gameTime);
}
```

Всегда радует, когда все подготовлено так, что перегруженному методу *Draw* практически ничего не надо делать. Компоненты *ColorBlock* и *Button* отрисовывают себя самостоятельно, поэтому метод *Draw* здесь лишь формирует визуальное представление *canvas*:

Проект XNA: PhingerPaint Файл: Game1.cs (фрагмент)

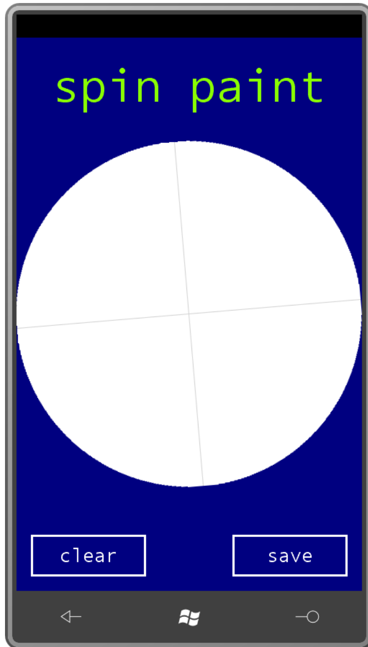
```
protected override void Draw(GameTime gameTime)
{
    this.GraphicsDevice.Clear(Color.Black);

    spriteBatch.Begin();
    spriteBatch.Draw(canvas, canvasPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Небольшой обзор SpinPaint

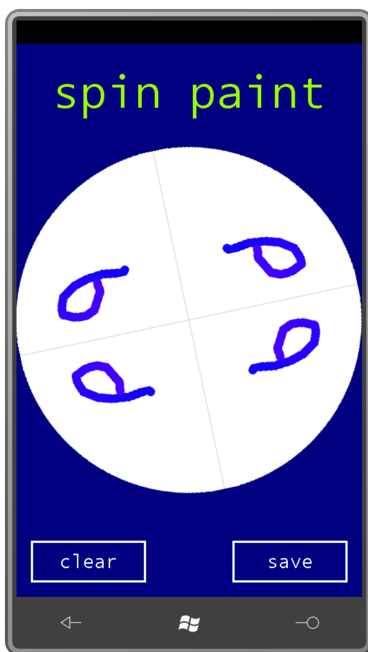
Приложение SpinPaint имеет необычную историю появления. Первую его версию я написал однажды утром, будучи слушателем двухдневных курсов по разработке ПО для Microsoft Surface (это такие компьютеры размером с журнальный столик, разработанные специально для общественных мест). Та версия была написана для Windows Presentation Foundation и могла использоваться одновременно несколькими пользователями, сидящими вокруг устройства.

Сначала я хотел представить Silverlight-версию SpinPaint в главе 14 данной книги для демонстрации *WriteableBitmap*, но производительность приложения была просто ужасной. Первую XNA-версию для Zune HD я написал до того, как у меня появился Windows Phone, и уже ту версию впоследствии я трансформировал в приложение, которое будет рассмотрено в этой главе.

SpinPaint начинает выполнение с отображения белого диска, который вращается с частотой 12 оборотов в минуту. Также можно заметить, что цвет названия приложения циклически меняется каждые 10 секунд:



Когда пользователь касается диска, на нем отрисовывается линия цветом, соответствующим цвету заголовка. Палец выступает в роли кисти, и диск перемещается под ним. При этом отрисовываемая линия отражается во всех четвертях относительно горизонтальной и вертикальной осей.



Продолжая рисовать, можно получить довольно фантазийные изображения:

Несомненно, пользователь захочет нажать кнопку «save», чтобы сохранить полученный результат в библиотеку фотографий телефона и затем послать его по электронной почте друзьям.

Как и в приложении PhingerPaint, для рисования может одновременно использоваться до четырех пальцев. Именно поэтому оба приложения реализуют простой сенсорный ввод, а не интерфейс обработки жестов.

Код SpinPaint

Приложение SpinPaint должно обрабатывать касание очень особым способом. Не только пальцы перемещаются по экрану, но и диск вращается под пальцами, поэтому даже если палец остается неподвижным, он продолжает рисовать. В отличие от PhingerPaint это приложение должно отслеживать каждое касание. Поэтому в нем определен *Dictionary* с целочисленным ключом (им является идентификатор касания), в котором хранятся объекты типа *TouchInfo*. *TouchInfo* – это небольшой внутренний класс *Game1*, сохраняющий два местоположения касания:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    // Поля, участвующие в описании текстуры вращающегося диска
    Texture2D diskTexture;
    uint[] pixels;
    Vector2 displayCenter;
    Vector2 textureCenter;
    int radius;
    Color currentColor;

    // Данные касания и поля, используемые при отрисовке линий
    class TouchInfo
    {
        public Vector2 PreviousPosition;
```

```
        public Vector2 CurrentPosition;
    }
    Dictionary<int, TouchInfo> touchDictionary = new Dictionary<int, TouchInfo>();
    float currentAngle;
    float previousAngle;
    List<float> xCollection = new List<float>();

    // Кнопки и заголовки
    Button clearButton, saveButton;
    SpriteFont segoe14;
    SpriteFont segoe48;
    string titleText = "spin paint";
    Vector2 titlePosition;
    string filename;
    …
}
```

Конструктор определяет для заднего буфера портретный режим отображения, но, как и в PhingerPaint, высота задается равной 768 пикселам, а не 800, чтобы оставалось место для строки состояния:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    // Портретный режим, но предусматривает место для строки состояния вверху
    graphics.PreferredBackBufferWidth = 480;
    graphics.PreferredBackBufferHeight = 768;
}
```

Выделение места под строку состояния означает, что пользователь будет видеть задний буфер полностью.

Два компонента *Button* создаются в ходе выполнения метода *Initialize*. Для них задаются свойства *Text* и обработчики события *Click*, и это пока что все:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    // Создаем компоненты Button
    clearButton = new Button(this, "clear");
    clearButton.Click += OnClearButtonClick;
    this.Components.Add(clearButton);

    saveButton = new Button(this, "save");
    saveButton.Click += OnSaveButtonClick;
    this.Components.Add(saveButton);

    base.Initialize();
}
```

Обратите внимание на крайне важный шаг по добавлению компонентов в коллекцию *Components* класса *Game*. Если не сделать этого, компоненты вообще не появятся на экране, что, вероятно, приведет вас в замешательство. (Я говорю, исходя из собственного опыта.)

Приложение не может позиционировать кнопки, пока не знает их размеров, а эти данные недоступны до тех пор, пока не загружены шрифты, что происходит только в ходе выполнения перегруженного *LoadContent*. Именно в нем для кнопок задается и шрифт, и местоположение:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Получаем данные экрана
    Rectangle clientBounds = this.GraphicsDevice.Viewport.Bounds;
    displayCenter = new Vector2(clientBounds.Center.X, clientBounds.Center.Y);

    // Загружаем шрифты и вычисляем местоположение заголовка
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");
    segoe48 = this.Content.Load<SpriteFont>("Segoe48");
    titlePosition = new Vector2((int)((clientBounds.Width -
                            segoe48.MeasureString(titleText).X) / 2), 20);

    // Задаем шрифты и местоположение кнопок
    clearButton.SpriteFont = segoe14;
    saveButton.SpriteFont = segoe14;
    Vector2 textSize = segoe14.MeasureString(clearButton.Text);
    int buttonWidth = (int)(2 * textSize.X);
    int buttonHeight = (int)(1.5 * textSize.Y);

    clearButton.Destination =
        new Rectangle(clientBounds.Left + 20,
                      clientBounds.Bottom - 20 - buttonHeight,
                      buttonWidth, buttonHeight);
    saveButton.Destination =
        new Rectangle(clientBounds.Right - 20 - buttonWidth,
                      clientBounds.Bottom - 20 - buttonHeight,
                      buttonWidth, buttonHeight);
}
```

Метод *LoadContent* не создает *Texture2D*, используемый для рисования, поскольку эта процедура должна быть включена в логику захоронения.

Как и в PhingerPaint, перегруженный *OnDeactivated* сохраняет изображение в формате PNG, и перегруженный *OnActivated* восстанавливает его. Оба метода вызывают методы класса *TextureExtensions* (Расширения текстуры) библиотеки Petzold.Phone.Xna. Если извлекать нечего, значит, выполняется запуск приложения, и необходимо создать новый *Texture2D*.

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void OnActivated(object sender, EventArgs args)
{
    // Восстанавливаем после захоронения
    bool newlyCreated = false;
    diskTexture = Texture2DExtensions.LoadFromPhoneServiceState(this.GraphicsDevice,
                                                                 "disk");
    // Или создаем новый Texture2D
    if (diskTexture == null)
    {
        Rectangle clientBounds = this.GraphicsDevice.Viewport.Bounds;
        int textureDimension = Math.Min(clientBounds.Width, clientBounds.Height);
        diskTexture = new Texture2D(this.GraphicsDevice, textureDimension,
                                                         textureDimension);
        newlyCreated = true;
```

```
    }

    pixels = new uint[diskTexture.Width * diskTexture.Height];
    radius = diskTexture.Width / 2;
    textureCenter = new Vector2(radius, radius);

    if (newlyCreated)
    {
        ClearPixelArray();
    }
    else
    {
        diskTexture.GetData<uint>(pixels);
    }

    base.OnActivated(sender, args);
}

protected override void OnDeactivated(object sender, EventArgs args)
{
    diskTexture.SaveToPhoneServiceState("disk");
    base.OnDeactivated(sender, args);
}
```

Если создается новый *Texture2D*, он инициализируется массивом *pixels*, который включает круговую область, полностью закрашенную белым цветом, за исключением пары светло-серых линий, которые обеспечивают пользователю видимый эффект вращения диска.

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
void ClearPixelArray()
{
    for (int y = 0; y < diskTexture.Height; y++)
        for (int x = 0; x < diskTexture.Width; x++)
            if (IsWithinCircle(x, y))
            {
                Color clr = Color.White;

                // Линии, разделяющие диск на четверти
                if (x == diskTexture.Width / 2 || y == diskTexture.Height / 2)
                    clr = Color.LightGray;

                pixels[y * diskTexture.Width + x] = clr.PackedValue;
            }
    diskTexture.SetData<uint>(pixels);
}

bool IsWithinCircle(int x, int y)
{
    x -= diskTexture.Width / 2;
    y -= diskTexture.Height / 2;

    return x * x + y * y < radius * radius;
}

void OnClearButtonClick(object sender, EventArgs args)
{
    ClearPixelArray();
}
```

Метод *ClearPixelArray* также вызывается, когда пользователь нажимает кнопку «clear».

Логика обработки кнопки «save» практически идентична применяемой в приложении PhingerPaint:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
void OnSaveButtonClick(object sender, EventArgs args)
{
    DateTime dt = DateTime.Now;
    string filename =
        String.Format("spinpaint-{0:D2}-{1:D2}-{2:D2}-{3:D2}-{4:D2}-{5:D2}",
                      dt.Year % 100, dt.Month, dt.Day, dt.Hour, dt.Minute,
dt.Second);

    Guide.BeginShowKeyboardInput(PlayerIndex.One, "spin paint save file",
                                 "enter filename:", filename, KeyboardCallback,
null);
}
void KeyboardCallback(IAsyncResult result)
{
    filename = Guide.EndShowKeyboardInput(result);
}
```

Как и в PhingerPaint, файл сохраняется в библиотеку фотографий в ходе выполнения перегруженного *Update*:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Если возвращен диалог Save File, сохраняем изображение
    if (!String.IsNullOrEmpty(filename))
    {
        diskTexture.SaveToPhotoLibrary(filename);
        filename = null;
    }
    …
}
```

Процесс рисования

Оставшаяся часть перегруженного *Update* выполняет на самом деле очень сложную работу: рисование на диске на основании сенсорного ввода и вращения диска.

Обработка *Update* начинается с вычисления текущего угла вращающегося диска и текущего цвета рисования:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Диск поворачивается каждые 5 секунд
    double seconds = gameTime.TotalGameTime.TotalSeconds;
    currentAngle = (float)(2 * Math.PI * seconds / 5);

    // Цвета меняются каждые 10 секунд
    float fraction = (float)(6 * (seconds % 10) / 10);

    if (fraction < 1)
        currentColor = new Color(1, fraction, 0);
    else if (fraction < 2)
```

```
        currentColor = new Color(2 - fraction, 1, 0);
    else if (fraction < 3)
        currentColor = new Color(0, 1, fraction - 2);
    else if (fraction < 4)
        currentColor = new Color(0, 4 - fraction, 1);
    else if (fraction < 5)
        currentColor = new Color(fraction - 4, 0, 1);
    else
        currentColor = new Color(1, 0, 6 - fraction);

    // Сначала предполагаем, что палец неподвижен
    foreach (TouchInfo touchInfo in touchDictionary.Values)
        touchInfo.CurrentPosition = touchInfo.PreviousPosition;
    …

}
```

Для любого касания экрана приложение сохраняет объект *TouchInfo* с полями *CurrentPosition* (Текущее местоположение) и *PreviousPosition* (Предыдущее местоположение). Эти координаты всегда отсчитываются относительно холста *Texture2D* без учета вращения. Поэтому данный раздел перегруженного метода *Update* завершается присвоением полю *CurrentPosition* значения поля *PreviousPosition* на основании предположения, что пальцы остаются неподвижными.

После этого *Update* готов заняться точкой касания, сначала вызывая метод *ProcessTouch* каждой из кнопок и затем находя новые координаты текущих касаний или новые касания. За преобразование координат касания относительно экрана в координаты относительно *Texture2D* отвечает небольшой метод *TranslateToTexture* (Перенести на текстуру), который показан здесь после *Update*.

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Получаем все касания
    TouchCollection touches = TouchPanel.GetState();

    foreach (TouchLocation touch in touches)
    {
        // Предоставляем компонентам Button первоочередное право на касание
        bool touchHandled = false;

        foreach (GameComponent component in this.Components)
        {
            if (component is IProcessTouch &&
                (component as IProcessTouch).ProcessTouch(touch))
            {
                touchHandled = true;
                break;
            }
        }

        if (touchHandled)
            continue;

        // Задаем элементы TouchInfo на основании данных касания
        int id = touch.Id;

        switch (touch.State)
        {
            case TouchLocationState.Pressed:
                if (!touchDictionary.ContainsKey(id))
```

```
                    touchDictionary.Add(id, new TouchInfo());

                touchDictionary[id].PreviousPosition =
TranslateToTexture(touch.Position);
                touchDictionary[id].CurrentPosition =
TranslateToTexture(touch.Position);
                break;

            case TouchLocationState.Moved:
                if (touchDictionary.ContainsKey(id))
                    touchDictionary[id].CurrentPosition =
                                TranslateToTexture(touch.Position);
                break;

            case TouchLocationState.Released:
                if (touchDictionary.ContainsKey(id))
                    touchDictionary.Remove(id);
                break;
        }
    }
    …
}

Vector2 TranslateToTexture(Vector2 point)
{
    return point - displayCenter + textureCenter;
}
```

Чтобы учесть вращение диска, предусмотрены поля *previousAngle* (Предыдущий угол) и *currentAngle* (Текущий угол). Теперь *Update* на основании значений этих полей вычисляет две матрицы: *previousRotation* (Предыдущий разворот) и *currentRotation* (Текущий разворот). Обратите внимание, что эти матрицы вычисляются посредством вызовов *Matrix.CreateRotationZ*, но при этом умножаются на трансформации переноса, что обеспечивает расчет вращения относительно центра *Texture2D*:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    // Вычисляем трансформации для вращения
    Matrix translate1 = Matrix.CreateTranslation(-textureCenter.X, -textureCenter.Y,
0);
    Matrix translate2 = Matrix.CreateTranslation(textureCenter.X, textureCenter.Y,
0);

    Matrix previousRotation = translate1 *
                                    Matrix.CreateRotationZ(-previousAngle) *
                                            translate2;
    Matrix currentRotation = translate1 *
                                    Matrix.CreateRotationZ(-currentAngle) *
                                            translate2;
    …
}
```

Когда трансформации вычислены, они могут применяться к полям *PreviousPosition* и *CurrentPosition* объекта *TouchInfo* посредством статического метода *Vector2.Transform* и затем передаваться в *RoundCappedLine* для получения данных, необходимых для отрисовки линии на *Texture2D*:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    bool textureNeedsUpdate = false;

    foreach (TouchInfo touchInfo in touchDictionary.Values)
    {
        // Выполняем рисование из предыдущей в текущую точку
        Vector2 point1 = Vector2.Transform(touchInfo.PreviousPosition,
previousRotation);
        Vector2 point2 = Vector2.Transform(touchInfo.CurrentPosition,
currentRotation);
        float radius = 6;

        RoundCappedLine line = new RoundCappedLine(point1, point2, radius);

        int yMin = (int)(Math.Min(point1.Y, point2.Y) - radius);
        int yMax = (int)(Math.Max(point1.Y, point2.Y) + radius);

        yMin = Math.Max(0, Math.Min(diskTexture.Height, yMin));
        yMax = Math.Max(0, Math.Min(diskTexture.Height, yMax));

        for (int y = yMin; y < yMax; y++)
        {
            xCollection.Clear();
            line.GetAllX(y, xCollection);

            if (xCollection.Count == 2)
            {
                int xMin = (int)(Math.Min(xCollection[0], xCollection[1]) + 0.5f);
                int xMax = (int)(Math.Max(xCollection[0], xCollection[1]) + 0.5f);

                xMin = Math.Max(0, Math.Min(diskTexture.Width, xMin));
                xMax = Math.Max(0, Math.Min(diskTexture.Width, xMax));

                for (int x = xMin; x < xMax; x++)
                {
                    if (IsWithinCircle(x, y))
                    {
                        // Отрисовываем точку во всех четвертях
                        int xFlip = diskTexture.Width - x;
                        int yFlip = diskTexture.Height - y;

                        pixels[y * diskTexture.Width + x] =
currentColor.PackedValue;
                        pixels[y * diskTexture.Width + xFlip] =
currentColor.PackedValue;
                        pixels[yFlip * diskTexture.Width + x] =
currentColor.PackedValue;
                        pixels[yFlip * diskTexture.Width + xFlip] =

currentColor.PackedValue;
                    }
                }
                textureNeedsUpdate = true;
            }
        }
    }

    if (textureNeedsUpdate)
    {
        // Обновляем текстуру значениями массива pixels
        this.GraphicsDevice.Textures[0] = null;
        diskTexture.SetData<uint>(pixels);
    }

    // Подготовка к следующему проходу
    foreach (TouchInfo touchInfo in touchDictionary.Values)
        touchInfo.PreviousPosition = touchInfo.CurrentPosition;
```

```
    previousAngle = currentAngle;

    base.Update(gameTime);
}
```

Сам перегруженный метод *Draw* чрезвычайно мал. Он лишь формирует визуальное представление вращающегося *diskTexture* (Текстура диска) и имени приложения, которое постоянно меняет цвет и отображается вверху экрана:

Проект XNA: SpinPaint Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(diskTexture, displayCenter, null, Color.White,
                     currentAngle, textureCenter, 1, SpriteEffects.None, 0);
    spriteBatch.DrawString(segoe48, titleText, titlePosition, currentColor);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

PhreeCell и колода карт

Изначально я не думал реализовывать в своем пасьянсе PhreeCell какие-либо дополнительные возможности, кроме необходимых для игры. Моя жена – она играет во FreeCell для Windows, и у нее пасьянс сходится практически всегда – абсолютно безапелляционно заявила, что PhreeCell необходимы еще две функции. Первое и самое важное – приложение должно каким-то образом поздравлять пользователя с его победой. Я реализовал это в виде производного от *DrawableGameComponent* компонента *CongratulationsComponent* (Компонент поздравления).

Второй важной возможностью является то, что я назвал «автоматическое перемещение». Если карта по всем правилам может быть перемещена в стопку соответствующей масти в верхнем правом углу поля, и нет никакой причины поступить иначе, карта переносится автоматически. Кроме этих двух, PhreeCell не имеет никаких других дополнений. Нет анимированной раздачи карт в начале игры, место перемещения карты нельзя обозначить простым «щелчком» и нет возможность переноса множества карт одновременно. Нет функции отмены хода, и подсказок тоже нет.

Разработку PhreeCell я начал не с приложения на XNA, а с приложения на Windows Presentation Foundation, формирующего одно растровое изображение размерами 1040 × 448, которое включает 52 игральные карты, каждая из которых 96 пикселов шириной и 112 пикселов высотой. *Canvas* заполняется числами, буквами и символами мастей преимущественно с помощью объектов *TextBlock*. После этого приложение передает *Canvas* в *RenderTargetBitmap* (Сформировать визуальное представление результирующего растрового изображения) и сохраняет результат в файл под именем cards.png. В XNA-проекте PhreeCell этот файл добавлен в содержимое приложения.

В проекте PhreeCell каждая карта – это объект типа *CardInfo* (Данные карты):

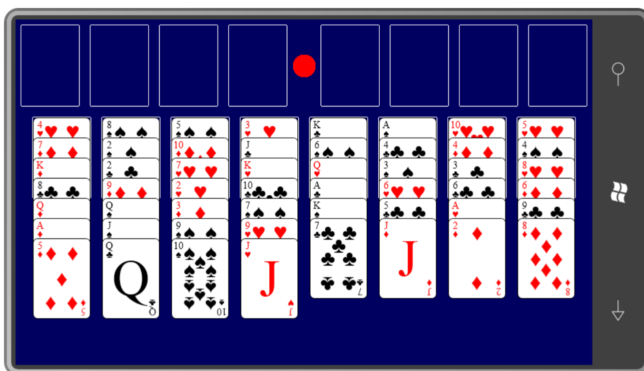
Проект XNA: PhreeCell Файл: CardInfo.cs

```
using System;
using Microsoft.Xna.Framework;

namespace PhreeCell
{
    class CardInfo
    {
        static string[] ranks = { "Ace", "Deuce", "Three", "Four",
                                  "Five", "Six", "Seven", "Eight",
                                  "Nine", "Ten", "Jack", "Queen", "King" };
        static string[] suits = { "Spades", "Clubs", "Hearts", "Diamonds" };

        public int Suit { protected set; get; }
        public int Rank { protected set; get; }

        public Vector2 AutoMoveOffset { set; get; }
        public TimeSpan AutoMoveTime { set; get; }
        public float AutoMoveInterpolation { set; get; }

        public CardInfo(int suit, int rank)
        {
            Suit = suit;
            Rank = rank;
        }

        // Используется для целей отладки
        public override string ToString()
        {
            return ranks[Rank] + " of " + suits[Suit];
        }
    }
}
```

Сначала этот класс имел просто свойства *Suit* (Масть) и *Rank* (Старшинство). Я добавил статические массивы *string* и метод *ToString* для целей отображения в ходе отладки и ввел еще три поля *AutoMove* (Автоматическое перемещение), когда реализовал возможность автоматического перемещения. Сам *CardInfo* не располагает сведениями о том, где фактически располагается карта во время игры. Эти данные сохраняются где-то в другом месте.

Игровое поле

На рисунке представлен исходный экран PhreeCell:



Я исхожу из того, что правила игры всем знакомы. Все 52 карты раскладываются лицом вверх в 8 столбцов, которые в приложении я называю «piles» (стопки). В верхнем левом углу предусмотрено четыре пустых поля для размещения отдельных карт. Я называю эти поля «holds»(свободные ячейки). В верхнем правом углу четыре поля для размещения карт одной

масти по старшинству, начиная с самого низкого ранга. Эти поля я называю «finals» (дом). Красная точка посередине – кнопка повторить игру.

Для удобства я разделил класс *Game1* на два файла. Первый – это обычный файл Game1.cs; второй – файл под именем Game1.Helpers.cs. Файл Game1.cs включает только методы, обычно используемые в небольших играх, реализующих также логику захоронения. В файле Game1.Helpers.cs располагается все остальное. Я создал этот файл, добавив новый класс в проект. В обоих файлах класс *Game1* наследуется от *Game*, и в обоих файлах ключевое слово *partial* указывает на то, что этот класс разделен на несколько файлов. В файлах Helpers нет полей экземпляров, только *const* и *static readonly*. Файл Game1.cs включает одно поле *static* и все поля экземпляров:

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public partial class Game1 : Microsoft.Xna.Framework.Game
{
    static readonly TimeSpan AutoMoveDuration = TimeSpan.FromSeconds(0.25);

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    CongratulationsComponent congratsComponent;

    Texture2D cards;
    Texture2D surface;
    Rectangle[] cardSpots = new Rectangle[16];

    Matrix displayMatrix;
    Matrix inverseMatrix;

    CardInfo[] deck = new CardInfo[52];
    List<CardInfo>[] piles = new List<CardInfo>[8];
    CardInfo[] holds = new CardInfo[4];
    List<CardInfo>[] finals = new List<CardInfo>[4];

    bool firstDragInGesture = true;
    CardInfo touchedCard;
    Vector2 touchedCardPosition;
    object touchedCardOrigin;
    int touchedCardOriginIndex;
    …
}
```

Данное приложение использует только два объекта *Texture2D*. Объект *cards* (карты) – растровое изображение, включающее все 52 карты; каждая отдельная карта отображается через определение прямоугольных фрагментов этого растрового изображения. Объект *surface* (поверхность) – это темно-синяя область (ее можно видеть на представленном снимке экрана), которая включает также белые прямоугольники и красную кнопку. Координаты для позиционирования этих 16 белых прямоугольников – еще восемь располагаются под каждой стопкой карт – хранятся в массиве *cardSpots* (Местоположения карт)

Поле *displayMatrix* (Матрица отображения) – это обычно единичная матрица. Каждому игроку в «свободную ячейку» известно, что иногда стопки карт могут становиться очень длинными. В этом случае *displayMatrix* выполняет масштабирование по вертикали и сжимает все игровое поле. *inverseMatrix* (Обратная матрица) – это обратная матрица, которая необходима для преобразования координат сенсорного ввода относительно экрана в точки сжатого растрового изображения.

Следующий блок полей – это основные структуры данных, используемые приложением. Массив *deck* (колода) включает все 52 объекта, которые были созданы приложением ранее и

повторно используются до завершения выполнения приложения. В ходе игры копии этих карт располагаются в *piles*, *holds* и *finals*. Сначала я думал сделать *finals* таким же массивом, как *holds*, и показывать только верхнюю карту. Но затем я понял, что для функции автоматического перемещения видимым должно быть большее количество карт.

Остальные поля связаны с выбором и перемещением карт. Поле *touchedCardPosition* (Местоположение перемещаемой карты) – это текущее местоположение перемещаемой карты. Поле *touchedCardOrigin* (Источник перемещаемой карты) сохраняет объект, из которого поступила перемещаемая карта. Это либо массив *holds*, либо *piles*, тогда как *touchedCardOriginIndex* (Индекс выбранной карты в источнике) – это индекс данного массива. Эти данные используются для возвращения карты в исходное положение, если пользователь пытается выполнить недопустимое перемещение.

Конструктор *Game1* показывает, что игре требуется игровое поле 800 пикселов шириной и 480 пикселов высотой без строки состояния. Также активируются три типа жестов:

Проект XNA: PhreeCell   Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.IsFullScreen = true;
    Content.RootDirectory = "Content";

    // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
    TargetElapsedTime = TimeSpan.FromTicks(333333);


    graphics.IsFullScreen = true;
    graphics.PreferredBackBufferWidth = 800;
    graphics.PreferredBackBufferHeight = 480;

    // Активируем жесты
    TouchPanel.EnabledGestures = GestureType.Tap |
                                 GestureType.FreeDrag |
                                 GestureType.DragComplete;
 }
```

Метод *Initialize* создает объекты *CardInfo* для массива *deck* и инициализирует массивы *piles* и *finals* объектами *List*. Также здесь создается и добавляется в коллекцию *Components* компонент *CongratulationsComponent*:

Проект XNA: PhreeCell   Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    // Инициализируем deck
    for (int suit = 0; suit < 4; suit++)
        for (int rank = 0; rank < 13; rank++)
        {
            CardInfo cardInfo = new CardInfo(suit, rank);
            deck[suit * 13 + rank] = cardInfo;
        }

    // Создаем объекты List для 8 стопок
    for (int pile = 0; pile < 8; pile++)
        piles[pile] = new List<CardInfo>();

    // Создаем объекты List для четырех 4 окончательных стопок
    for (int final = 0; final < 4; final++)
        finals[final] = new List<CardInfo>();
```

```
    // Создаем компонент поздравления
    congratsComponent = new CongratulationsComponent(this);
    congratsComponent.Enabled = false;
    this.Components.Add(congratsComponent);
    base.Initialize();
}
```

Метод *LoadContent* загружает растровое изображение с изображениями карт и также вызывает два метода части класса *Game1*, реализованной в файле Game1.Helpers.cs:

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Загружаем большое растровое изображение, включающее изображения карт
    cards = this.Content.Load<Texture2D>("cards");

    // Создаем 16 прямоугольных областей для карт и растровую поверхность
    CreateCardSpots(cardSpots);
    surface = CreateSurface(this.GraphicsDevice, cardSpots);
}
```

В приложении для коммерческого использования я бы, безусловно, добавил второй набор карт для небольшого экрана. Такое решение обеспечивает намного лучшее визуальное представление, чем отображение карт в масштабе 60% от их оригинального размера.

Файл Game1.Helpers.cs начинается с описания ряда констант, которые определяют все размеры игрового поля в пикселах:

Проект XNA: PhreeCell Файл: Game1.Helper.cs (фрагмент, демонстрирующий поля)

```
public partial class Game1 : Microsoft.Xna.Framework.Game
{
    const int wCard = 80;        // ширина карты
    const int hCard = 112;       // высота карты

    // Горизонтальные размеры
    const int wSurface = 800;    // ширина поверхности
    const int xGap = 16;         // расстояние между стопками
    const int xMargin = 8;       // поле слева и справа

    // расстояние между "holds" и "finals"
    const int xMidGap = wSurface - (2 * xMargin + 8 * wCard + 6 * xGap);

    // дополнительное поля для второго ряда
    const int xIndent = (wSurface - (2 * xMargin + 8 * wCard + 7 * xGap)) / 2;

    // Вертикальные размеры
    const int yMargin = 8;       // вертикальное поле над верхним рядом
    const int yGap = 16;         // вертикальное поле между рядами
    const int yOverlay = 28;     // ширина видимой части вверху каждой карты в стопке
    const int hSurface = 2 * yMargin + yGap + 2 * hCard + 19 * yOverlay;

    // Кнопка повтора игры
    const int radiusReplay = xMidGap / 2 - 8;
    static readonly Vector2 centerReplay =
                    new Vector2(wSurface / 2, xMargin + hCard / 2);
    …
}
```

Обратите внимание, что *wSurface* – ширина игрового поля – задано равным 800 пикселам, что соответствует ширине большого экрана телефона. Но может возникнуть необходимость сделать размер по вертикали больше 480. В области *piles* может располагаться до 20 перекрывающихся карт. Для обеспечения такой возможности *hSurface* вычисляется как максимально возможная высота, исходя из того, что в стопке может находиться 20 перекрывающихся карт.

Метод *CreateCardSpots* (Создать места размещения карт) использует эти константы для расчёта местоположения 16 объектов *Rectangle*, обозначающих места размещения карт на игровом поле. В верхнем ряду располагаются *holds* и *finals*, нижний ряд предназначен для *piles*:

Проект XNA: PhreeCell Файл: Game1.Helper.cs (фрагмент)

```
static void CreateCardSpots(Rectangle[] cardSpots)
{
    // Верхний ряд
    int x = xMargin;
    int y = yMargin;

    for (int i = 0; i < 8; i++)
    {
        cardSpots[i] = new Rectangle(x, y, wCard, hCard);
        x += wCard + (i == 3 ? xMidGap : xGap);
    }

    // Нижний ряд
    x = xMargin + xIndent;
    y += hCard + yGap;

    for (int i = 8; i < 16; i++)
    {
        cardSpots[i] = new Rectangle(x, y, wCard, hCard);
        x += wCard + xGap;
    }
}
```

Метод *CreateSurface* (Создать поверхность) создает растровое изображение, используемое в качестве игрового поля. Размер этого растрового изображения вычисляется на основании значений *hSurface* (заданного как константа и равного 800) и *wSurface*, значение которого намного превышает 480. Для отрисовки белых прямоугольников и красной кнопки повтора игры этот метод работает напрямую со значениями пикселов растрового изображения:

Проект XNA: PhreeCell Файл: Game1.Helper.cs (фрагмент)

```
static Texture2D CreateSurface(GraphicsDevice graphicsDevice, Rectangle[] cardSpots)
{
    uint backgroundColor = new Color(0, 0, 0x60).PackedValue;
    uint outlineColor = Color.White.PackedValue;
    uint replayColor = Color.Red.PackedValue;
    Texture2D surface = new Texture2D(graphicsDevice, wSurface, hSurface);
    uint[] pixels = new uint[wSurface * hSurface];

    for (int i = 0; i < pixels.Length; i++)
    {
        if ((new Vector2(i % wSurface, i / wSurface) - centerReplay).LengthSquared()
<
                                radiusReplay * radiusReplay)
            pixels[i] = replayColor;
        else
            pixels[i] = backgroundColor;
```

```
    }

    foreach (Rectangle rect in cardSpots)
    {
        // верхушки прямоугольников
        for (int x = 0; x < wCard; x++)
        {
            pixels[(rect.Top - 1) * wSurface + rect.Left + x] = outlineColor;
            pixels[rect.Bottom * wSurface + rect.Left + x] = outlineColor;
        }
        // стороны прямоугольников
        for (int y = 0; y < hCard; y++)
        {
            pixels[(rect.Top + y) * wSurface + rect.Left - 1] = outlineColor;
            pixels[(rect.Top + y) * wSurface + rect.Right] = outlineColor;
        }
    }

    surface.SetData<uint>(pixels);
    return surface;
}
```

Другой статический метод класса *Game1* не требует особых пояснений.

Проект XNA: PhreeCell Файл: Game1.Helper.cs (фрагмент)

```
static void ShuffleDeck(CardInfo[] deck)
{
    Random rand = new Random();

    for (int card = 0; card < 52; card++)
    {
        int random = rand.Next(52);
        CardInfo swap = deck[card];
        deck[card] = deck[random];
        deck[random] = swap;
    }
}

static bool IsWithinRectangle(Vector2 point, Rectangle rect)
{
    return point.X >= rect.Left &&
           point.X <= rect.Right &&
           point.Y >= rect.Top &&
           point.Y <= rect.Bottom;
}

static Rectangle GetCardTextureSource(CardInfo cardInfo)
{
    return new Rectangle(wCard * cardInfo.Rank,
                         hCard * cardInfo.Suit, wCard, hCard);
}

static CardInfo TopCard(List<CardInfo> cardInfos)
{
    if (cardInfos.Count > 0)
        return cardInfos[cardInfos.Count - 1];

    return null;
}
```

GetCardTextureSource (Получить источник текстуры карты) используется в сочетании с большим растровым изображением *cards*. Этот метод просто возвращает объект *Rectangle*, соответствующий конкретной карте. *TopCard* (Верхняя карта) возвращает последний элемент

коллекции *List<CardInfo>*, что необходимо для получения самой верхней карты коллекции *piles* или *finals*.

В конце перегруженного *LoadContent* приложение практически готово к вызову метода *Replay* (Повторить игру), который перетасовывает колоду и «сдает» карты в коллекции *piles*. Но еще необходимо реализовать захоронение. Изначально, до реализации захоронения, это приложение было построено вокруг массивов и коллекций *piles*, *holds* и *finals*. Мне было приятно осознать, что эти три элемента были единственной частью приложения, которую требовалось сохранять и извлекать при захоронении. Но мне не давало покоя то, что эти три объекта включали ссылки на 52 экземпляра *CardInfo*, хранящихся в *deck*, и я хотел сохранить это отношение. Поэтому я пришел к тому, чтобы сохранять и извлекать не экземпляры *CardInfo*, а целочисленные индексы от 0 до 52. Для этого потребовалось небольшое количество довольно скучного кода:

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент)

```
protected override void OnDeactivated(object sender, EventArgs args)
{
    PhoneApplicationService appService = PhoneApplicationService.Current;

    // Сохраняем целочисленные индексы для piles
    List<int>[] piles = new List<int>[8];

    for (int i = 0; i < piles.Length; i++)
    {
        piles[i] = new List<int>();

        foreach (CardInfo cardInfo in this.piles[i])
            piles[i].Add(13 * cardInfo.Suit + cardInfo.Rank);
    }
    appService.State["piles"] = piles;

    // Сохраняем целочисленные индексы для finals
    List<int>[] finals = new List<int>[4];

    for (int i = 0; i < finals.Length; i++)
    {
        finals[i] = new List<int>();

        foreach (CardInfo cardInfo in this.finals[i])
            finals[i].Add(13 * cardInfo.Suit + cardInfo.Rank);
    }
    appService.State["finals"] = finals;

    // Сохраняем целочисленные индексы для holds
    int[] holds = new int[4];

    for (int i = 0; i < holds.Length; i++)
    {
        if (this.holds[i] == null)
            holds[i] = -1;
        else
            holds[i] = 13 * this.holds[i].Suit + this.holds[i].Rank;
    }
    appService.State["holds"] = holds;

    base.OnDeactivated(sender, args);
}

protected override void OnActivated(object sender, EventArgs args)
{
    PhoneApplicationService appService = PhoneApplicationService.Current;

    if (appService.State.ContainsKey("piles"))
```

```
    {
        // Извлекаем целочисленные индексы для piles
        List<int>[] piles = appService.State["piles"] as List<int>[];

        for (int i = 0; i < piles.Length; i++)
        {
            foreach (int cardindex in piles[i])
                this.piles[i].Add(deck[cardindex]);
        }

        // Извлекаем целочисленные индексы для finals
        List<int>[] finals = appService.State["finals"] as List<int>[];

        for (int i = 0; i < finals.Length; i++)
        {
            foreach (int cardindex in finals[i])
                this.finals[i].Add(deck[cardindex]);
        }

        // Извлекаем целочисленные индексы для holds
        int[] holds = appService.State["holds"] as int[];

        for (int i = 0; i < holds.Length; i++)
        {
            if (holds[i] != -1)
                this.holds[i] = deck[holds[i]];
        }
        CalculateDisplayMatrix();
    }
    else
    {
        Replay();
    }
    base.OnActivated(sender, args);
}
```

Замечательная новость в том, что в самом конце перегруженного *OnActivated* вызывается метод *Replay*, который фактически и запускает игру.

Play и Replay

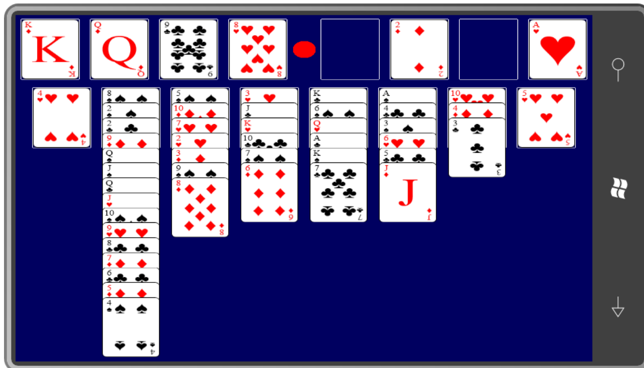
Метод *Replay* находится в классе Game1.Helper.cs:

Проект XNA: PhreeCell Файл: Game1.Helper.cs (фрагмент)

```
void Replay()
{
    for (int i = 0; i < 4; i++)
        holds[i] = null;

    foreach (List<CardInfo> final in finals)
        final.Clear();

    foreach (List<CardInfo> pile in piles)
        pile.Clear();

    ShuffleDeck(deck);

    // Распределение карт по стопкам
    for (int card = 0; card < 52; card++)
    {
        piles[card % 8].Add(deck[card]);
    }
    CalculateDisplayMatrix();
}
```

Данный метод очищает массив *holds* и коллекции *finals* и *piles*, перемешивает колоду карт случайным образом и распределяет их в восемь коллекций в *piles*. Метод завершается вызовом *CalculateDisplayMatrix* (Вычислить матрицу отображения). Это не единственный случай вызова данного метода. Он также вызывается из метода *OnActivated* при восстановлении приложения после захоронения. И после этого при любом перемещении карты из или добавлении в одну из коллекций матрица отображения пересчитывается, просто на всякий случай.

Эта матрица отвечает за определение высоты игрового поля, если его необходимо увеличить для просмотра всех карт в области *piles*. Приложение обрабатывает этот аспект не очень красиво. Все игровое поле просто немного сжимается, включая все карты и даже кнопку «повторить игру»:



Это решение мне не очень нравится, тем не менее рассмотрим метод *CalculateDisplayMatrix*, который все это делает:

Проект XNA: PhreeCell   Файл: Game1.Helper.cs (фрагмент)

```
void CalculateDisplayMatrix()
{
    // Высота окна просмотра равна 480, соответственно
    // предпочтительным настройкам заднего буфера
    int viewportHeight = this.GraphicsDevice.Viewport.Height;

    // Определяем общую требуемую высоту и выполняем масштабирование по вертикали
    int maxCardsInPiles = 0;

    foreach (List<CardInfo> pile in piles)
        maxCardsInPiles = Math.Max(maxCardsInPiles, pile.Count);

    int requiredHeight = 2 * yMargin + yGap + 2 * hCard +
                              yOverlay * (maxCardsInPiles - 1);

    // Задаем матрицу сжатия по оси Y, чтобы обеспечить отображение на экране всех
карт
    if (requiredHeight > viewportHeight)
        displayMatrix = Matrix.CreateScale(1, (float)viewportHeight /
requiredHeight, 1);
    else
        displayMatrix = Matrix.Identity;

    // Находим обратную матрицу для тестирования
    inverseMatrix = Matrix.Invert(displayMatrix);
}
```

Объект *displayMatrix* используется в вызове *Begin* класса *SpriteBatch*, так что применяется ко всем объектам одним махом. Хотя это несколько не соответствует моему привычному порядку изложения, но мы уже готовы рассмотреть метод *Draw* класса *Game1*.

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin(SpriteSortMode.Immediate, null, null, null, null, null,
                      displayMatrix);
    spriteBatch.Draw(surface, Vector2.Zero, Color.White);

    // Отрисовываем holds
    for (int hold = 0; hold < 4; hold++)
    {
        CardInfo cardInfo = holds[hold];

        if (cardInfo != null)
        {
            Rectangle source = GetCardTextureSource(cardInfo);
            Vector2 destination = new Vector2(cardSpots[hold].X, cardSpots[hold].Y);
            spriteBatch.Draw(cards, destination, source, Color.White);
        }
    }

    // Отрисовываем piles
    for (int pile = 0; pile < 8; pile++)
    {
        Rectangle cardSpot = cardSpots[pile + 8];

        for (int card = 0; card < piles[pile].Count; card++)
        {
            CardInfo cardInfo = piles[pile][card];
            Rectangle source = GetCardTextureSource(cardInfo);
            Vector2 destination = new Vector2(cardSpot.X, cardSpot.Y + card *
yOverlay);
            spriteBatch.Draw(cards, destination, source, Color.White);
        }
    }

    // Отрисовываем finals, включая все предыдущие карты (для автоперемещения)
    for (int pass = 0; pass < 2; pass++)
    {
        for (int final = 0; final < 4; final++)
        {
            for (int card = 0; card < finals[final].Count; card++)
            {
                CardInfo cardInfo = finals[final][card];

                if (pass == 0 && cardInfo.AutoMoveInterpolation == 0 ||
                    pass == 1 && cardInfo.AutoMoveInterpolation != 0)
                {
                    Rectangle source = GetCardTextureSource(cardInfo);
                    Vector2 destination =
                            new Vector2(cardSpots[final + 4].X,
                                        cardSpots[final + 4].Y) +
                                  cardInfo.AutoMoveInterpolation *
cardInfo.AutoMoveOffset;
                    spriteBatch.Draw(cards, destination, source, Color.White);
                }
            }
        }
    }

    // Отрисовываем выбранную карту
    if (touchedCard != null)
    {
        Rectangle source = GetCardTextureSource(touchedCard);
        spriteBatch.Draw(cards, touchedCardPosition, source, Color.White);
    }

    spriteBatch.End();
```

```
    base.Draw(gameTime);
}
```

После вызова метода *Begin* объекта *SpriteBatch* и вывода растрового изображения *surface* для игрового поля метод готов к отрисовке карт. Этот процесс начинается с самого простого: четырех возможных карт массива *holds*. Небольшой метод *GetCardTextureSource* возвращает *Rectangle* для позиционирования карты в рамках растрового изображения карт, и массив *cardSpot* обеспечивает координаты для размещения каждой карты.

Следующая часть несколько сложнее. В зоне *piles* для отображения перекрывающихся карт к координатам *cardSpot* должны применяться смещения. По-настоящему проблематичной является зона *finals*. Сложности здесь связаны с реализацией возможности автоматического перемещения. Как мы увидим, карта, отвечающая условиям для автоматического перемещения, удаляется из массива *holds* или коллекции *piles*, в которой она находится, и помещается в коллекцию *finals*. Но это перемещение карты из предыдущего местоположения в новое должно быть анимировано. Для этого в *CardInfo* предусмотрены свойства *AutoMoveOffset* (Смещение для автоперемещения) и *AutoMoveInterpolation* (Интерполяция для автоперемещения).

Метод *Draw* должен отрисовывать все четыре коллекции *finals* последовательно слева направо и затем в рамках каждой коллекции все карты, начиная с самой первой карты (которой всегда является туз) до самой последней, которая располагается в самом верху стопки. Я обнаружил, что это не всегда получается, и анимированная карта иногда на мгновение как будто задвигается в одну из стопок *finals*. Именно поэтому цикл для отображения коллекций *finals* включает два прохода: один для неанимированных карт и другой для любой анимированной в ходе автоматического перемещения карты. (Данное приложение одновременно выполняет анимацию только одной карты, а вот в предыдущей версии обеспечивалась анимация нескольких карт.)

Draw завершается отрисовкой карты, которую пользователь, возможно, перемещает в настоящий момент.

Метод *Update* практически полностью посвящен реализации анимации для автоматического перемещения и обработки касания. Его большая часть с вложенными циклами *foreach* обеспечивает перемещение карт, которые помечены для автоматического перемещения и, следовательно, уже перенесены в коллекции *finals*.

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Обрабатываем автоматическое перемещение карты и, возможно,
    // инициируем следующее автоматическое перемещение
    bool checkForNextAutoMove = false;

    foreach (List<CardInfo> final in finals)
        foreach (CardInfo cardInfo in final)
        {
            if (cardInfo.AutoMoveTime > TimeSpan.Zero)
            {
                cardInfo.AutoMoveTime -= gameTime.ElapsedGameTime;

                if (cardInfo.AutoMoveTime <= TimeSpan.Zero)
                {
```

```
                    cardInfo.AutoMoveTime = TimeSpan.Zero;
                    checkForNextAutoMove = true;
                }
                cardInfo.AutoMoveInterpolation = (float)cardInfo.AutoMoveTime.Ticks
/
                                                        AutoMoveDuration.Ticks;
            }
        }

    if (checkForNextAutoMove && !AnalyzeForAutoMove() && HasWon())
    {
        congratsComponent.Enabled = true;
    }
    …
}
```

На самом деле выбор карт для автоматического перемещения выполняется в конце этого кода посредством вызова метода *AnalyzeforAutoMove* (Анализ возможности автоперемещения), который описан в файле Game1.Helpers.cs. (*AnalyzeforAutoMove* также вызывается позже в перегруженном *Update* после перемещения карты вручную.) Этот метод перебирает все элементы *holds* и *piles* и вызывается метод *CheckForAutoMove* (Проверка возможности автоперемещения) для каждой верхней карты. Если *CheckForAutoMove* возвращает *true*, значит, этот метод уже переместил карту в соответствующую коллекцию *finals*, и ее необходимо убрать с того места, где она находится на экране. Для фактического перемещения в *Update* инициализируются три свойства *CardInfo*, показанные выше:

Проект XNA: PhreeCell Файл: Game1.Helpers.cs (фрагмент)

```
bool AnalyzeForAutoMove()
{
    for (int hold = 0; hold < 4; hold++)
    {
        CardInfo cardInfo = holds[hold];

        if (cardInfo != null && CheckForAutoMove(cardInfo))
        {
            holds[hold] = null;
            cardInfo.AutoMoveOffset += new Vector2(cardSpots[hold].X,
cardSpots[hold].Y);
            cardInfo.AutoMoveInterpolation = 1;
            cardInfo.AutoMoveTime = AutoMoveDuration;
            return true;
        }
    }

    for (int pile = 0; pile < 8; pile++)
    {
        CardInfo cardInfo = TopCard(piles[pile]);

        if (cardInfo != null && CheckForAutoMove(cardInfo))
        {
            piles[pile].Remove(cardInfo);
            cardInfo.AutoMoveOffset += new Vector2(cardSpots[pile + 8].X,
                                    cardSpots[pile + 8].Y + piles[pile].Count *
yOverlay);
            cardInfo.AutoMoveInterpolation = 1;
            cardInfo.AutoMoveTime = AutoMoveDuration;
            return true;
        }
    }
    return false;
}
```

Код реализации логики выбора карт, соответствующих условиям автоматического перемещения (если таковые имеются), оказывается самой длинной частью приложения. Сложность в том, что карта, которая еще может использоваться в стратегии игры, не должна перемещаться в коллекцию *finals*. Например, четверка червей не должна быть перенесена в коллекцию *finals*, если где-то в коллекции *piles* или *holds* еще имеется тройка пик или тройка треф.

Проект XNA: PhreeCell Файл: Game1.Helpers.cs (фрагмент)

```
bool CheckForAutoMove(CardInfo cardInfo)
{
    if (cardInfo.Rank == 0)      // т.е. туз
    {
        for (int final = 0; final < 4; final++)
            if (finals[final].Count == 0)
            {
                finals[final].Add(cardInfo);
                cardInfo.AutoMoveOffset = -new Vector2(cardSpots[final + 4].X,
                                                       cardSpots[final + 4].Y);
                return true;
            }
    }
    else if (cardInfo.Rank == 1)     // т.е. двойка
    {
        for (int final = 0; final < 4; final++)
        {
            CardInfo topCardInfo = TopCard(finals[final]);

            if (topCardInfo != null &&
                topCardInfo.Suit == cardInfo.Suit &&
                topCardInfo.Rank == 0)
            {
                finals[final].Add(cardInfo);
                cardInfo.AutoMoveOffset = -new Vector2(cardSpots[final + 4].X,
                                                       cardSpots[final + 4].Y);
                return true;
            }
        }
    }
    else
    {
        int slot = -1;
        int count = 0;

        for (int final = 0; final < 4; final++)
        {
            CardInfo topCardInfo = TopCard(finals[final]);

            if (topCardInfo != null)
            {
                if (topCardInfo.Suit == cardInfo.Suit &&
                    topCardInfo.Rank == cardInfo.Rank - 1)
                {
                    slot = final;
                }
                else if (topCardInfo.Suit < 2 != cardInfo.Suit < 2 &&
                        topCardInfo.Rank >= cardInfo.Rank - 1)
                {
                    count++;
                }
            }
        }
        if (slot >= 0 && count == 2)
        {
            cardInfo.AutoMoveOffset = -new Vector2(cardSpots[slot + 4].X,
```

```
                                                    cardSpots[slot + 4].Y);
            finals[slot].Add(cardInfo);
            return true;
        }
    }
    return false;
}
```

Ранее в перегруженном *Update* после анимации автоматически перемещаемых карт выполнялась проверка того, не касается ли пользователь карты, пытаясь «выбрать» ее. Выбор конкретной карты может быть допустимым или нет. Если карта уже перемещена и пользователь пытается «положить» эту карту, ее выбор также может быть недопустимым. Допустимость определяется в ходе вызовов *TryPickUpCard* (Попытаться выбрать карту) и *TryPutDownCard* (Попытаться положить карту). Обратите внимание, что точка касания настраивается с помощью *inverseMatrix* соответственно фактическому местоположению карты.

Проект XNA: PhreeCell Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    …
    while (TouchPanel.IsGestureAvailable)
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        // Настраиваем местоположение и его изменение для сжатого изображения
        Vector2 position = Vector2.Transform(gesture.Position, inverseMatrix);
        Vector2 delta = position - Vector2.Transform(gesture.Position -
gesture.Delta,
                                                     inverseMatrix);
        switch (gesture.GestureType)
        {
            case GestureType.Tap:
                // Проверяем, нажата ли кнопка Replay
                if ((gesture.Position - centerReplay).Length() < radiusReplay)
                {
                    congratsComponent.Enabled = false;
                    Replay();
                }
                break;

            case GestureType.FreeDrag:
                // Продолжаем перемещать выбранную карту
                if (touchedCard != null)
                {
                    touchedCardPosition += delta;
                }
                // Делаем попытку выбрать карту
                else if (firstDragInGesture)
                {
                    TryPickUpCard(position);
                }
                firstDragInGesture = false;
                break;

            case GestureType.DragComplete:
                if (touchedCard != null && TryPutDownCard(touchedCard))
                {
                    CalculateDisplayMatrix();

                    if (!AnalyzeForAutoMove() && HasWon())
                    {
                        congratsComponent.Enabled = true;
```

```
                    }
                }
                firstDragInGesture = true;
                touchedCard = null;
                break;
        }
    }
    base.Update(gameTime);
}
```

Оба метода, *TryPickUpCard* и *TryPutDownCard*, реализованы в файле Game1.Helpers.cs и на самом деле определяют правила игры.

TryPickUpCard – более простой из этих двух методов. Он лишь принимает координаты касания и должен определить, на какую карту это касание приходится. Выбрана может быть только карта одной из коллекций *holds* или верхняя карта одной из коллекций *piles*. В противном случае метод даже не пытается определить возможность каких-либо манипуляций с картой:

Проект XNA: PhreeCell Файл: Game1.Helpers.cs (фрагмент)

```
bool TryPickUpCard(Vector2 position)
{
    for (int hold = 0; hold < 4; hold++)
    {
        if (holds[hold] != null && IsWithinRectangle(position, cardSpots[hold]))
        {
            Point pt = cardSpots[hold].Location;

            touchedCard = holds[hold];
            touchedCardOrigin = holds;
            touchedCardOriginIndex = hold;
            touchedCardPosition = new Vector2(pt.X, pt.Y);
            holds[hold] = null;
            return true;
        }
    }

    for (int pile = 0; pile < 8; pile++)
    {
        if (piles[pile].Count > 0)
        {
            Rectangle pileSpot = cardSpots[pile + 8];
            pileSpot.Offset(0, yOverlay * (piles[pile].Count - 1));

            if (IsWithinRectangle(position, pileSpot))
            {
                Point pt = pileSpot.Location;
                int pileIndex = piles[pile].Count - 1;

                touchedCard = piles[pile][pileIndex];
                touchedCardOrigin = piles;
                touchedCardOriginIndex = pile;
                touchedCardPosition = new Vector2(pt.X, pt.Y);
                piles[pile].RemoveAt(pileIndex);
                return true;
            }
        }
    }
    return false;
}
```

Как только карта выбрана, метод *TryPickUpCard* уже задает поля, касающиеся этой выбранной карты, которые затем используются методом *Update* для последующего ее перемещения по экрану.

Метод *TryPutDownCard* позволяет размещать карты в коллекциях *piles* или *finals* либо в массиве *holds*, обеспечивая при этом выполнение правил игры. Если карта не может быть помещена в данную стопку, она просто восстанавливается в исходном местоположении без всякой анимации:

Проект XNA: PhreeCell Файл: Game1.Helpers.cs (фрагмент)

```
bool TryPutDownCard(CardInfo touchedCard)
{
    Vector2 cardCenter = new Vector2(touchedCardPosition.X + wCard / 2,
                                     touchedCardPosition.Y + hCard / 2);

    for (int cardSpot = 0; cardSpot < 16; cardSpot++)
    {
        Rectangle rect = cardSpots[cardSpot];

        // Значительно расширяем прямоугольник местоположения карт для стопок
        if (cardSpot >= 8)
           rect.Inflate(0, hSurface - rect.Bottom);

        if (IsWithinRectangle(cardCenter, rect))
        {
            // Проверяем пуста ли свободная ячейка
            if (cardSpot < 4)
            {
                int hold = cardSpot;

                if (holds[hold] == null)
                {
                    holds[hold] = touchedCard;
                    return true;
                }
            }

            else if (cardSpot < 8)
            {
                int final = cardSpot - 4;

                if (TopCard(finals[final]) == null)
                {
                    if (touchedCard.Rank == 0)  // т.е. туз
                    {
                        finals[final].Add(touchedCard);
                        return true;
                    }
                }
                else if (touchedCard.Suit == TopCard(finals[final]).Suit &&
                         touchedCard.Rank == TopCard(finals[final]).Rank + 1)
                {
                    finals[final].Add(touchedCard);
                    return true;
                }
            }
            else
            {
                int pile = cardSpot - 8;

                if (piles[pile].Count == 0)
                {
                    piles[pile].Add(touchedCard);
                    return true;
```

```
                }
                else
                {
                    CardInfo topCard = TopCard(piles[pile]);

                    if (touchedCard.Suit < 2 != topCard.Suit < 2 &&
                        touchedCard.Rank == topCard.Rank - 1)
                    {
                        piles[pile].Add(touchedCard);
                        return true;
                    }
                }
            }

            // Карта располагалась в заданном прямоугольнике,
            // но ее размещение там было недопустимым
            break;
        }
    }

    // Восстанавливаем карту в ее исходном местоположении
    if (touchedCardOrigin is CardInfo[])
    {
        (touchedCardOrigin as CardInfo[])[touchedCardOriginIndex] = touchedCard;
    }
    else
    {
        ((touchedCardOrigin as
List<CardInfo>[])[touchedCardOriginIndex]).Add(touchedCard);
    }
    return false;
}
```

Но вся эта работа прекращается, когда следующий метод, который просто проверяет, не является ли верхняя карта коллекции *finals* королем, возвращает значение *true*:

Проект XNA: PhreeCell Файл: Game1.Helpers.cs (фрагмент)

```
bool HasWon()
{
    bool hasWon = true;

    foreach (List<CardInfo> cardInfos in finals)
        hasWon &= cardInfos.Count > 0 && TopCard(cardInfos).Rank == 12;

    return hasWon;
}
```

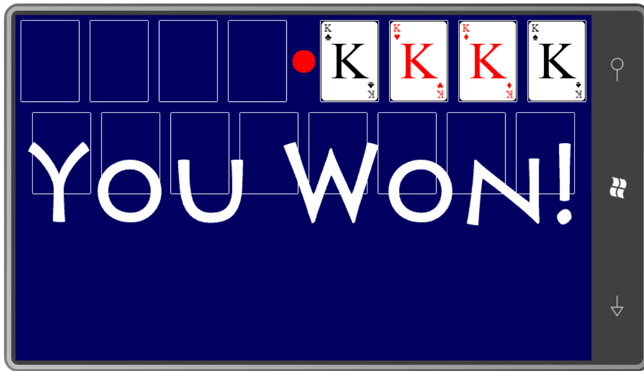
Метод *Update* использует это для активации компонента *CongratulationsComponent*, полный код которого представлен в следующем листинге:

Проект XNA: PhreeCell Файл: CongratulationsComponent.cs

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace PhreeCell
{
    public class CongratulationsComponent : DrawableGameComponent
    {
        const float SCALE_SPEED = 0.5f;                    // полразмера в секунду
        const float ROTATE_SPEED = 3 * MathHelper.TwoPi;    // 3 оборота в секунду
```

```
        SpriteBatch spriteBatch;
        SpriteFont pericles108;
        string congratulationsText = "You Won!";
        float textScale;
        float textAngle;
        Vector2 textPosition;
        Vector2 textOrigin;

        public CongratulationsComponent(Game game) : base(game)
        {
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(this.GraphicsDevice);
            pericles108 = this.Game.Content.Load<SpriteFont>("Pericles108");
            textOrigin = pericles108.MeasureString(congratulationsText) / 2;
            Viewport viewport = this.GraphicsDevice.Viewport;
            textPosition = new Vector2(Math.Max(viewport.Width, viewport.Height) /
2,
                                       Math.Min(viewport.Width, viewport.Height) /
2);
            base.LoadContent();
        }

        protected override void OnEnabledChanged(object sender, EventArgs args)
        {
            Visible = Enabled;

            if (Enabled)
            {
                textScale = 0;
                textAngle = 0;
            }
        }

        public override void Update(GameTime gameTime)
        {
            if (textScale < 1)
            {
                textScale +=
                    SCALE_SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
                textAngle +=
                    ROTATE_SPEED * (float)gameTime.ElapsedGameTime.TotalSeconds;
            }
            else if (textAngle != 0)
            {
                textScale = 1;
                textAngle = 0;
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            spriteBatch.Begin();
            spriteBatch.DrawString(pericles108, congratulationsText, textPosition,
                                   Color.White, textAngle, textOrigin, textScale,
                                   SpriteEffects.None, 0);
            spriteBatch.End();
            base.Draw(gameTime);
        }
    }
}
```

Ничего сверхъестественного: просто выводится постепенно увеличивающийся вращающийся текст, который в итоге размещается в центре экрана:



А теперь можно нажать красную кнопку и начать игру заново. Удачи!

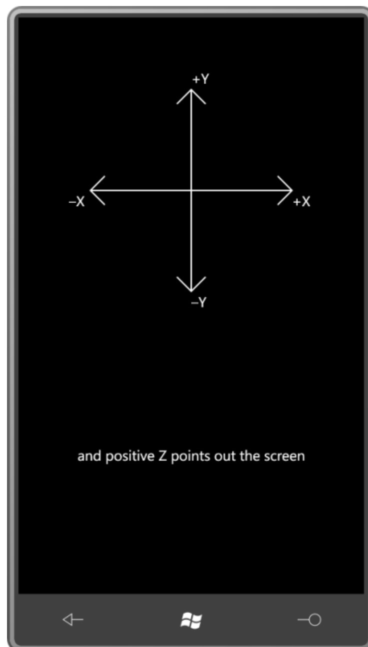
Глава 24

Применение акселерометра в игровых приложениях

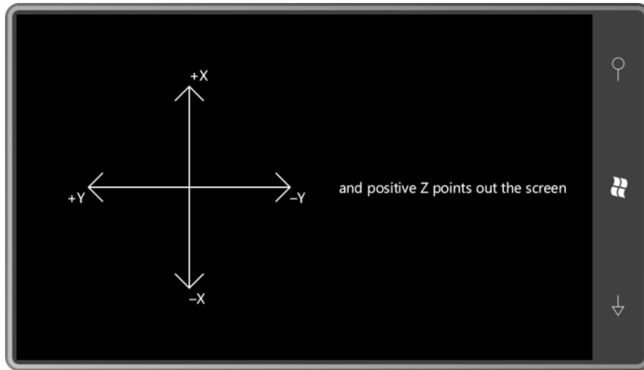
Если главным инструментом пользовательского интерфейса в Windows Phone 7 является касание, то что является вторым по значимости? Конечно, все зависит от поставленной задачи, но во многих приложениях на Silverlight, я полагаю, большую роль по-прежнему будет играть клавиатура. А вот в приложениях на XNA вторым по значимости инструментом пользовательского интерфейса, вероятно, будет акселерометр, особенно в аркадных играх, где перемещение самого телефона может заменить традиционные органы ручного управления. Возьмем, к примеру, игру-симулятор автогонок по треку или городу. Повороты налево и направо могут быть реализованы посредством наклонения телефона вправо и влево, а наклонением телефона вперед и назад можно управлять педалью газа, например.

Трехмерные векторы

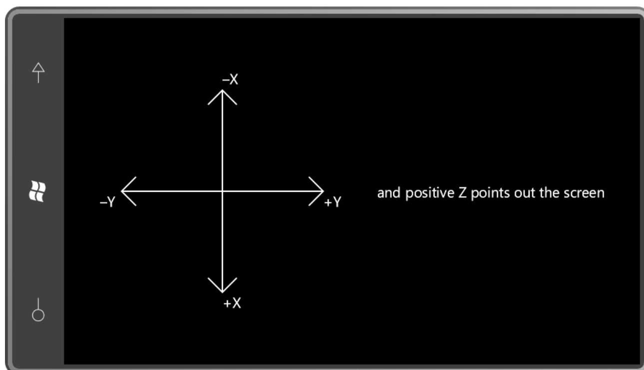
Как было показано в главе 5, акселерометр телефона предоставляет данные в виде трехмерного вектора **(*x*, *y*, *z*)**, описываемого в фиксированной относительно телефона системе координат. Система координат акселерометра остается фиксированной независимо от ориентации телефона и в портретном режиме:



и при левостороннем альбомном расположении:

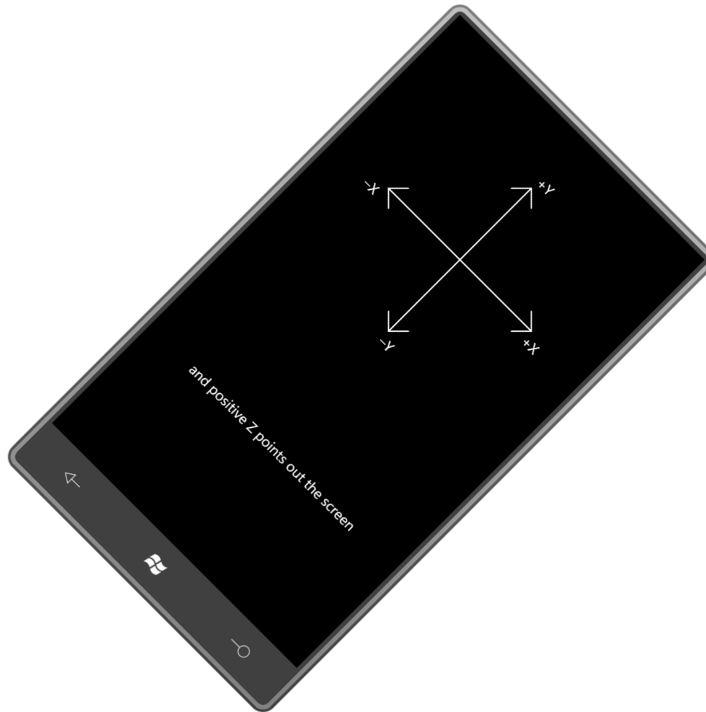


и при правостороннем альбомном расположении:



Обратите внимание, что положительное направление оси Y обращено вверх телефона в портретном расположении, как в обычной декартовой системе координат и системе координат трехмерного XNA. А вот в системе координат двухмерного XNA значения Y увеличиваются по направлению вниз.

Если телефон остается неподвижен, вектор ускорения указывает в точку на экране телефона, расположенную ближе всего к земле. Например, если вектор ускорения равен примерно **(0.7,–0.7, 0)**, значит, телефон располагается следующим образом:



Значение 0,7 в примере – это приближенный квадратный корень из ½. Если найти модуль вектора ускорения по теореме Пифагора

$$\sqrt{x^2 + y^2 + z^2}$$

результат должен быть равен приблизительно 1, когда телефон неподвижен. Вектор ускорения телефона, лежащего на плоской поверхности, равен примерно **(0, 0, –1)**. Это же значение выводит эмулятор телефона.

Я сказал «приблизительно», потому что акселерометр телефона не является прибором высокой точности. Модуль вектора должен выражаться в единицах ускорения свободного падения на поверхности земли, которое традиционно обозначают g, и которое равно примерно 9,8 м/с^2. Но обычно акселерометр телефона дает до 5% погрешности в измерениях.

Составляющие X, Y и Z показаний акселерометра телефона, лежащего на плоской поверхности, могут также давать несколько процентов погрешности. В приложения реализации плотницкого уровня, использующие акселерометр, обычно включена опция для «калибровки» акселерометра, которая на самом деле обеспечивает настройку всех будущих показаний относительно показаниям, фиксируемым в момент нажатия кнопки «калибровать» пользователем.

Данные акселерометра довольно неравномерны, что вы, безусловно, заметили при работе с двумя приложениями, представленными в главе 5. Сглаживание данных также занимает довольно большую часть в приложениях, использующих акселерометр.

В данной главе для сглаживания данных я буду применять очень простую методику фильтрации нижних частот и не буду касаться вопросов калибровки. Библиотеку, которая поможет при решении обоих этих вопросов, можно найти в статье блога Дейва Эдсона (Dave Edson) «Using the Accelerometer on Windows Phone 7»[1] по адресу

---

[1] Использование акселерометра в Windows Phone 7 (*прим. переводчика*).

http://windowsteamblog.com/windows_phone/b/wpdev/archive/2010/09/08/using-the-accelerometer-on-windows-phone-7.aspx.

Во все приложения данной главы должны быть включены ссылки на сборку Microsoft.Devices.Sensors и директивы *using* для пространства имен *Microsoft.Devices.Sensors*.

Улучшенная визуализация «пузырька»

Приложение AccelerometerVisualization (Визуализация акселерометра) – небольшой шаг по улучшению приложения XnaAccelerometer из главы 5. Приложение XnaAccelerometer просто показывало плавающий «пузырек» без какой либо шкалы или числовых значений. В данном приложении добавляется шкала (в виде концентрических кругов) и некоторая текстовая информация:

Приложение AccelerometerVisualization также реализует, вероятно, самый базовый тип сглаживания: фильтр нижних частот, который усредняет текущее значение с предыдущим сглаженным значением. Необработанные показания акселерометра отображаются в строке «Raw» (Необработанный), тогда как сглаженное значение представлено в строке «Avg» («average» – среднее). Отображаются также минимальное и максимальное значения. Они вычисляются с помощью методов *Vector3.Min* и *Vector3.Max*, которые находят минимальное и максимальное значения составляющих X, Y и Z по отдельности. Красный «пузырек» масштабируется соответственно модулю вектора и меняет цвет на зеленый, если составляющая Z вектора имеет положительное значение.

Рассмотрим поля приложения:

Проект XNA: AccelerometerVisualization Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const int BALL_RADIUS = 8;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    SpriteFont segoe14;
    StringBuilder stringBuilder = new StringBuilder();

    int unitRadius;
    Vector2 screenCenter;
    Texture2D backgroundTexture;
    Vector2 backgroundTextureCenter;
    Texture2D ballTexture;
    Vector2 ballTextureCenter;
    Vector2 ballPosition;
```

```
    float ballScale;
    bool isZNegative;

    Vector3 accelerometerVector;
    object accerlerometerVectorLock = new object();

    Vector3 oldAcceleration;
    Vector3 minAcceleration = Vector3.One;
    Vector3 maxAcceleration = -Vector3.One;
    …
}
```

Хотя приложение отображает что-то наподобие «пузырька», который перемещается в направлении, противоположном направлению силы тяжести и вектора ускорения, в приложении этот объект называют «ball» (шар). Поле *oldAcceleration* (Предыдущее значение ускорения) используется для сглаживания значений. При каждом обновлении экрана значение *oldAcceleration* соответствует предыдущему сглаженному («Avg») значению.

По умолчанию приложения на XNA самостоятельно обрабатывают левостороннюю или правостороннюю альбомную ориентацию, и вмешиваться в это поведение по умолчанию практически никогда не требуется, *за исключением* приложений, использующих акселерометр. При левосторонней альбомной ориентации положительное направление оси X акселерометра направлено вверх экрана, и положительное направление оси Y – влево вдоль экрана. При правосторонней альбомной ориентации положительное направление оси X акселерометра направлено вниз экрана, и положительное направление оси Y – вправо вдоль экрана.

Чтобы упростить эту запутанную схему, в конструкторе класса свойству *SupportedOrientations* объекта *GraphicsDeviceManager* (Диспетчер графических устройств) (на который ссылается поле *graphics*) можно задать значение *DisplayOrientation.LandscapeLeft*. В моем приложении этого не сделано, и оно поддерживает ориентацию *LandscapeRight*. Однако в конструкторе задается размер заднего буфера для обеспечения места под строку состояния телефона:

Проект XNA: AccelerometerVisualization Файл: Game1.cs (фрагмент)

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    // Альбомная ориентация, но оставляем место под строку состояния
    graphics.PreferredBackBufferWidth = 728;
    graphics.PreferredBackBufferHeight = 480;
}
```

В перегруженном *Initialize* создается объект *Accelerometer*, задается обработчик событий изменения показаний и выполняется запуск акселерометра:

Проект XNA: AccelerometerVisualization Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try
```

```
    {
        accelerometer.Start();
    }
    catch
    {
    }

    base.Initialize();
}
```

Обработчик событий *ReadingChanged* (Показания изменились) вызывается асинхронно, поэтому правильным поведением будет просто сохранять значение в коде, защитив его блокировкой *lock*:

Проект XNA: AccelerometerVisualization Файл: Game1.cs (фрагмент)

```
void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accerlerometerVectorLock)
    {
        accelerometerVector = new Vector3((float)args.X, (float)args.Y,
(float)args.Z);
    }
}
```

Если немного поэкспериментировать с обработчиком *ReadingChanged*, выясняется, что он вызывается почти 50 раз в секунду. Это даже чаще, чем обновляется экран, поэтому сглаживание может быть целесообразным реализовать именно в этом обработчике. (Я так поступаю в некоторых приложениях данной главы, которые будут рассматриваться далее.)

Перегруженный метод *LoadContent* в данном приложении преимущественно отвечает за подготовку двух текстур: большого поля *backgroundTexture* (Текстура фона), который покрывает всю поверхность концентрическими кругами, и *ballTexture* (Текстура шара), который перемещается по полю. Код, отрисовывающий линии и окружности на *backgroundTexture*, довольно *специализированный*, включает простые циклы и пару методов для задания цветов пикселов.

Проект XNA: AccelerometerVisualization Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(this.GraphicsDevice);

    // Получаем данные экрана и шрифта
    viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);
    segoe14 = this.Content.Load<SpriteFont>("Segoe14");

    // Эквивалент единичного вектора в пикселах
    unitRadius = (viewport.Height - BALL_RADIUS) / 2;

    // Создаем и отрисовываем текстуру фона
    backgroundTexture =
        new Texture2D(this.GraphicsDevice, viewport.Height, viewport.Height);
    backgroundTextureCenter =
        new Vector2(viewport.Height / 2, viewport.Height / 2);

    Color[] pixels = new Color[backgroundTexture.Width * backgroundTexture.Height];
```

```
    // Отрисовываем горизонтальную линию
    for (int x = 0; x < backgroundTexture.Width; x++)
        SetPixel(backgroundTexture, pixels,
                 x, backgroundTexture.Height / 2, Color.White);

    // Отрисовываем вертикальную линию
    for (int y = 0; y < backgroundTexture.Height; y++)
        SetPixel(backgroundTexture, pixels,
                 backgroundTexture.Width / 2, y, Color.White);

    // Отрисовываем окружности
    DrawCenteredCircle(backgroundTexture, pixels, unitRadius, Color.White);
    DrawCenteredCircle(backgroundTexture, pixels, 3 * unitRadius / 4, Color.Gray);
    DrawCenteredCircle(backgroundTexture, pixels, unitRadius / 2, Color.White);
    DrawCenteredCircle(backgroundTexture, pixels, unitRadius / 4, Color.Gray);
    DrawCenteredCircle(backgroundTexture, pixels, BALL_RADIUS, Color.White);

    // Задаем значения пикселов текстуры фона
    backgroundTexture.SetData<Color>(pixels);

    // Создаем и отрисовываем текстуру пузырька
    ballTexture = new Texture2D(this.GraphicsDevice,
                                2 * BALL_RADIUS, 2 * BALL_RADIUS);
    ballTextureCenter = new Vector2(BALL_RADIUS, BALL_RADIUS);
    pixels = new Color[ballTexture.Width * ballTexture.Height];
    DrawFilledCenteredCircle(ballTexture, pixels, BALL_RADIUS);
    ballTexture.SetData<Color>(pixels);
}

void DrawCenteredCircle(Texture2D texture, Color[] pixels, int radius, Color clr)
{
    Point center = new Point(texture.Width / 2, texture.Height / 2);
    int halfPoint = (int)(0.707 * radius + 0.5);

    for (int y = -halfPoint; y <= halfPoint; y++)
    {
        int x1 = (int)Math.Round(Math.Sqrt(radius * radius - Math.Pow(y, 2)));
        int x2 = -x1;

        SetPixel(texture, pixels, x1 + center.X, y + center.Y, clr);
        SetPixel(texture, pixels, x2 + center.X, y + center.Y, clr);

        // Поскольку они симметричны, просто подставляем координаты
        // для симметричной части
        SetPixel(texture, pixels, y + center.X, x1 + center.Y, clr);
        SetPixel(texture, pixels, y + center.X, x2 + center.Y, clr);
    }
}

void DrawFilledCenteredCircle(Texture2D texture, Color[] pixels, int radius)
{
    Point center = new Point(texture.Width / 2, texture.Height / 2);

    for (int y = -radius; y < radius; y++)
    {
        int x1 = (int)Math.Round(Math.Sqrt(radius * radius - Math.Pow(y, 2)));

        for (int x = -x1; x < x1; x++)
            SetPixel(texture, pixels, x + center.X, y + center.Y, Color.White);
    }
}

void SetPixel(Texture2D texture, Color[] pixels, int x, int y, Color clr)
{
    pixels[y * texture.Width + x] = clr;
}
```

Именно эта логика подсказала мне о необходимости явного задания ширины заднего буфера равной 728. Если использовать для ширины значение по умолчанию, 800 пикселов, фактическое представление будет сжиматься на 10%, обеспечивая место для строки состояния. Толщина отрисовываемых линий и окружностей составляет всего лишь 1 пиксел, и для них не реализовано никакого механизма сглаживания, поэтому в случае сжатия экрана их четкость частично будет утрачена.

В перегруженном *Update* происходит несколько любопытных вещей. Этот метод в основном отвечает за получение вектора ускорения и отображение его в графической и числовой форме. Метод сохраняет необработанное значение в поле *newAcceleration* (Новое ускорение), а сглаженное значение – в поле *avgAcceleration* (Среднее ускорение):

Проект XNA: AccelerometerVisualization   Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    Vector3 newAcceleration = Vector3.Zero;

    lock (accerlerometerVectorLock)
    {
        newAcceleration = accelerometerVector;
    }

    maxAcceleration = Vector3.Max(maxAcceleration, newAcceleration);
    minAcceleration = Vector3.Min(minAcceleration, newAcceleration);

    // Сглаживание с использованием фильтра нижних частот
    Vector3 avgAcceleration = 0.5f * oldAcceleration + 0.5f * newAcceleration;

    stringBuilder.Remove(0, stringBuilder.Length);
    stringBuilder.AppendFormat("Raw: ({0:F2}, {1:F2}, {2:F2}) = {3:F2}\n",
                               newAcceleration.X, newAcceleration.Y,
                               newAcceleration.Z, newAcceleration.Length());
    stringBuilder.AppendFormat("Avg: ({0:F2}, {1:F2}, {2:F2}) = {3:F2}\n",
                               avgAcceleration.X, avgAcceleration.Y,
                               avgAcceleration.Z, avgAcceleration.Length());
    stringBuilder.AppendFormat("Min: ({0:F2}, {1:F2}, {2:F2}) = {3:F2}\n",
                               minAcceleration.X, minAcceleration.Y,
                               minAcceleration.Z, minAcceleration.Length());
    stringBuilder.AppendFormat("Max: ({0:F2}, {1:F2}, {2:F2}) = {3:F2}",
                               maxAcceleration.X, maxAcceleration.Y,
                               maxAcceleration.Z, maxAcceleration.Length());

    ballScale = avgAcceleration.Length();
    int sign = this.Window.CurrentOrientation ==
                                    DisplayOrientation.LandscapeLeft ? 1 : -1;
    ballPosition =
        new Vector2(screenCenter.X + sign * unitRadius * avgAcceleration.Y /
ballScale,
                    screenCenter.Y + sign * unitRadius * avgAcceleration.X /
ballScale);
    isZNegative = avgAcceleration.Z < 0;

    oldAcceleration = avgAcceleration;

    base.Update(gameTime);
}
```

Поле *accelerometerVector* сохраняется обработчиком события *ReadingChanged* во втором потоке выполнения с помощью блокировки *lock*, поэтому для выполнения доступа к нему из

основного потока выполнения приложения требуется другая блокировка *lock*, использующая тот же объект:

```
lock (accerlerometerVectorLock)
{
    newAcceleration = accelerometerVector;
}
```

После этого на основании необработанного значения и значения поля *oldAcceleration* вычисляется сглаженное значение:

```
Vector3 avgAcceleration = 0.5f * oldAcceleration + 0.5f * newAcceleration;
```

Это значение *avgAcceleration* используется методом *Update* для целей отображения и замещает значение поля *oldAcceleration*:

```
oldAcceleration = avgAcceleration;
```

Такой тип сглаживания называют *фильтром нижних частот*. Он обеспечивает сглаживание высокочастотных отклонений путем усреднения текущего значения с предыдущими. Если v_0 – это текущее необработанное значение вектора (*newAcceleration*), и v_{-1} – предыдущее необработанное значение, а v_{-2} – это необработанное значение, предшествующее v_{-1}, сглаженное значение вычисляется по формуле:

$$v = \frac{v_0}{2} + \frac{v_{-1}}{4} + \frac{v_{-2}}{8} + \frac{v_{-3}}{16} + \cdots$$

Но сохранять все эти старые значения нет необходимости, потому что все они уже учтены при вычислении *oldAcceleration*.

Влияние каждого значения итеративно снижается вдвое при каждом следующем пересчете. Теоретически все значения постоянно учитываются при сглаживании каждого последующего значения, но уже через секунду (или 30 вызовов перегруженного *Update*) после того, как данное конкретное значение было зафиксировано, его делитель равен миллиону, т.е. его влияние на текущее сглаженное значение ничтожно мало.

С помощью коэффициентов можно настраивать влияние предыдущих и текущих показателей, но в сумме эти коэффициенты должны равняться 1. Например, такое выражение обеспечивает меньшее сглаживание за счет уменьшения воздействия предыдущих значений:

```
Vector3 avgAcceleration = 0.25f * oldAcceleration + 0.75f * newAcceleration;
```

А в этом случае обеспечивается большее сглаживание:

```
Vector3 avgAcceleration = 0.75f * oldAcceleration + 0.25f * newAcceleration;
```

В упоминаемой мною ранее публикации блога Дейв Эдсон описывает фильтр нижних частот несколько иначе. При использовании принятого мною именования переменных его формула принимает следующий вид:

```
Vector3 avgAcceleration = oldAcceleration + alpha * (newAcceleration –
oldAcceleration);
```

где *alpha* меняется в диапазоне от 0 (новое значение не учитывается) до 1 (отсутствие сглаживания). В этой публикации блога также обсуждаются альтернативы фильтра нижних частот, применяемых, когда приложение заинтересовано в неожиданных изменениях или забросах значений.

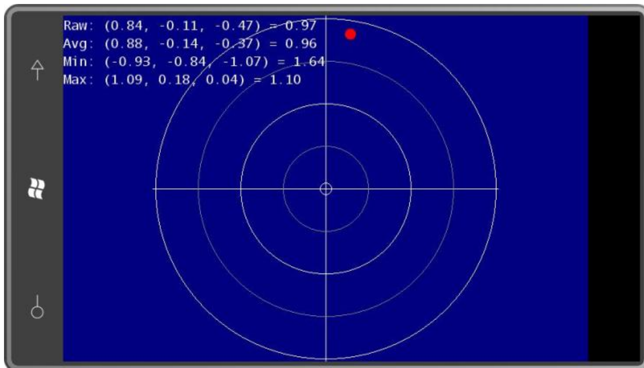
Еще одна любопытная функция *Update* касается введения поправки с учетом двух разных альбомных ориентаций, в основе чего, безусловно, лежит вектор ускорения. Предположим, телефон ориентирован в режиме *LandscapeLeft*:

Положительное направление оси Y акселерометра соответствует отрицательному направлению оси X экрана, и положительное направление оси X акселерометра соответствует отрицательному направлению оси Y экрана. Но наша цель показать «пузырек», который всплывает в направлении, противоположном действию силы тяжести, поэтому

Рост координаты Y акселерометра → рост координаты X экрана

Рост координаты X акселерометра → рост координаты Y экрана

Теперь переворачиваем телефон и ориентируем его в режиме *LandscapeRight*:



Теперь получаем, что

Рост координаты Y акселерометра → уменьшение координаты X экрана

Рост координаты X акселерометра → уменьшение координаты Y экрана

В перегруженном *Update* сначала определяется значение *sign* (знак), которое соответствует 1 для режима *LandscapeLeft* и –1 для *LandscapeRight*:

```
int sign = this.Window.CurrentOrientation == DisplayOrientation.LandscapeLeft ? 1 : -
1;
```

Если оба компонента X и Y сглаженного вектора ускорения равны 0, «пузырек» должен размещаться в точке (*screenCenter.X*, *screenCenter.Y*). Довольно просто. Смещения этого центра должны вычисляться на основании значения *sign* и расстояния от центра до внешнего радиуса:

```
ballPosition =
    new Vector2(screenCenter.X + sign * unitRadius * avgAcceleration.Y,
                screenCenter.Y + sign * unitRadius * avgAcceleration.X);
```

Но я не был удовлетворен этим вычислением. Небольшие неточности в показаниях акселерометра – и «пузырек» полностью «вылетал» за внешний круг и границы экрана. Я

принял решение компенсировать это делением на длину сглаженного вектора. Этот прием уже использовался для масштабирования «пузырька»:

```
ballScale = avgAcceleration.Length();
```

Поэтому я включил его в вычисление местоположения «пузырька»:

```
ballPosition =
    new Vector2(screenCenter.X + sign * unitRadius * avgAcceleration.Y / ballScale,
                screenCenter.Y + sign * unitRadius * avgAcceleration.X / ballScale);
```

Перегруженный *Draw* отрисовывает фон, «пузырек» и четыре строки текста:

Проект XNA: AccelerometerVisualization   Файл: Game1.cs (фрагмент)

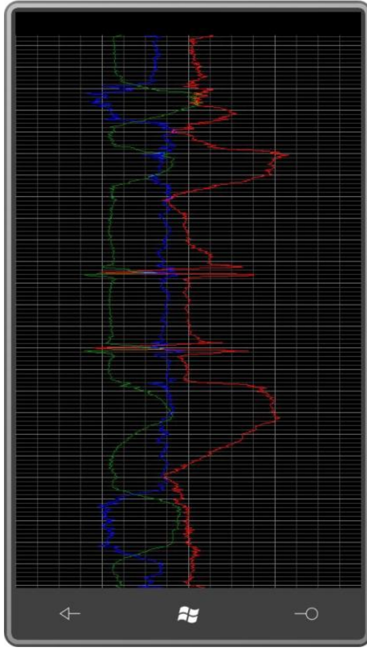
```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(backgroundTexture, screenCenter, null, Color.White, 0,
                     backgroundTextureCenter, 1, SpriteEffects.None, 0);
    spriteBatch.Draw(ballTexture, ballPosition, null,
                     isZNegative ? Color.Red : Color.Lime, 0,
                     ballTextureCenter, ballScale, SpriteEffects.None, 0);
    spriteBatch.DrawString(segoe14, stringBuilder, Vector2.Zero, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Когда вы трясете телефон, заметить перемещения пузырька сложно, но всегда видны изменения минимального и максимального значений. Если аппаратные средства используемого вами телефона аналогичны моему, составляющие X, Y и Z необработанного вектора ускорения никогда не выйдут за рамки диапазона от -2 до 2, более того различие между максимальным и минимальным значением не превысит 3,46. Похоже, это ограничение аппаратных средств.

Графическое представление

Для перемещения и задания направления объектов в приложениях данной главы используются сглаженные показания акселерометра. Но свое применение могут найти и *скачкообразные* изменения показаний акселерометра, например, для моделирования игры в кости. Прежде всего, необходимо понять, как изменяются показания акселерометра при резком перемещении телефона. В этом нам поможет график изменения этих показаний во времени.

Такой график строится приложением AccelerometerGraph (Диаграмма акселерометра). На рисунке показано его типовое представление:



Назначение данного приложения – продемонстрировать пользователю реальные показания акселерометра, поэтому в нем не выполняется никакого сглаживания. Красная линия соответствует значениям X, зеленая – Y, и синяя – Z. (Мнемоническая схема RGB == XYZ.) При портретном режиме отображения график перемещается вверх по экрану по мере того, как снизу добавляются новые значения. Каждый пиксел в вертикальном направлении соответствует одному «тику» обновлению экрана, т.е. 1/30 секунды. Более толстые горизонтальные линии представляют секунды. Более тонкие горизонтальные линии соответствуют 1/5 секунды (6 пикселов). Вертикальная линия в центре соответствует нулевому значению составляющей ускорения. Две другие более толстые вертикальные линии соответствуют значениям 1 (справа) и -1 (слева). Левый край экрана представляет значение -2, и правый край – значение 2. Как продемонстрировало предыдущее приложение, этого должно быть достаточно для соответствующего отображения всего диапазона возможных значений.

В ходе выполнения приложения старые значения как будто «ползут» вверх по экрану. Инстинктивно кажется, что в коде это должно быть реализовано следующим образом: создается объект *Texture2D* размером с экран и при каждом вызове *Update* все его пикселы просто сдвигаются на ширину *Texture2D*, так что верхняя строка исчезает, и новая строка может быть добавлена снизу. Но при таком подходе 30 раз в секунду приходится перемещать слишком большое количество пикселов.

Имеет больше смысла вставлять новые данные в строку, номер которой определяется переменной, получающей приращение при каждом вызове *Update*. (Назовем ее *строкой вставки*.) После этого в методе *Draw* можно дважды отрисовывать разделенный на две части *Texture2D*. Первая часть отображается вверху экрана и начинается со строки, следующей за строкой вставки, и продолжается до конца *Texture2D*. Вторая часть отрисовывается под первой; она начинается вверху *Texture2D* и заканчивается строкой вставки.

Поскольку чтобы обеспечить возможность отображения новых данных, старые данные должны быть удалены из строки вставки, неподвижные линии графика имеет смысл обрабатывать отдельно. Эта часть повторяется в вертикальном направлении через каждые 30 пикселов, поэтому она может быть реализована как небольшое растровое изображение, отображаемое многократно. В полях AccelerometerGraph для *backgroundTexture* задается

высота 30 пикселов, и высота *graphTexture* (Текстура графика) (в которой отображаются красная, зеленая и синяя линии) соответствует размеру экрана.

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    int displayWidth, displayHeight;
    Texture2D backgroundTexture;
    Texture2D graphTexture;
    uint[] pixels;
    int totalTicks;
    int oldInsertRow;
    Vector3 oldAcceleration;

    Vector3 accelerometerVector;
    object accelerometerVectorLock = new object();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        graphics.SupportedOrientations = DisplayOrientation.Portrait;
        graphics.PreferredBackBufferWidth = 480;
        graphics.PreferredBackBufferHeight = 768;
    }
    …
}
```

Конструктор завершается заданием портретной ориентации и определением такой высоты заднего буфера, которая обеспечивала бы отображение строки состояния.

Как обычно, метод *Initialize* запускает акселерометр:

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try
    {
        accelerometer.Start();
    }
    catch
    {
    }
    base.Initialize();
}

void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accelerometerVectorLock)
    {
        accelerometerVector = new Vector3((float)args.X, (float)args.Y,
```

```
(float)args.Z);
    }
}
```

Метод *LoadContent* преимущественно посвящен созданию и инициализации *backgroundTexture*, который включает горизонтальные и вертикальные линии. Хотя код здесь довольно обобщенный, но высота *backgroundTexture* будет вычисляться равной 30 пикселам, и горизонтальные линии будут отрисовываться через каждые 6 пикселов.

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    displayWidth = this.GraphicsDevice.Viewport.Width;
    displayHeight = this.GraphicsDevice.Viewport.Height;

    // Создаем текстуру фона и инициализируем ее
    int ticksPerSecond = 1000 / this.TargetElapsedTime.Milliseconds;
    int ticksPerFifth = ticksPerSecond / 5;
    backgroundTexture = new Texture2D(this.GraphicsDevice, displayWidth,
ticksPerSecond);
    pixels = new uint[backgroundTexture.Width * backgroundTexture.Height];

    for (int y = 0; y < backgroundTexture.Height; y++)
        for (int x = 0; x < backgroundTexture.Width; x++)
        {
            Color clr = Color.Black;

            if (y == 0 || x == backgroundTexture.Width / 2 ||
                         x == backgroundTexture.Width / 4 ||
                         x == 3 * backgroundTexture.Width / 4)
            {
                clr = new Color(128, 128, 128);
            }
            else if (y % ticksPerFifth == 0 ||
                    ((x - backgroundTexture.Width / 2) %
                                    (backgroundTexture.Width / 16) == 0))
            {
                clr = new Color(64, 64, 64);
            }

            pixels[y * backgroundTexture.Width + x] = clr.PackedValue;
        }
    backgroundTexture.SetData<uint>(pixels);

    // Создаем текстуру графика
    graphTexture = new Texture2D(this.GraphicsDevice, displayWidth, displayHeight);
    pixels = new uint[graphTexture.Width * graphTexture.Height];

    // Инициализируем
    oldInsertRow = graphTexture.Height - 2;
}
```

В конце *LoadContent* создается большой *graphTexture*, размер которого соответствует размеру экрана, и воссоздается поле массива *pixels* специально для этого растрового изображения.

LoadContent завершается заданием предпоследней строки *Texture2D* в качестве значения *oldInsertRow* (Предыдущая строка вставки). Как будет показано далее, при первом

вычислении строки вставки в методе *Update* в качестве значения *insertRow* задается последняя строка растрового изображения.

Каждый вызов *Update* обеспечивает отрисовку трех прямых линий на *graphTexture*: красной, зеленой и синей. Если принять, что каждая линия отрисовывается из точки (x_1, y_1) в точку (x_2, y_2), тогда y_2 должна равняться $y_1 + 1$ (если не произойдет ничего такого, что приведет к потере тиков в *Update*). Значения X каждой цветной линии вычисляются на основании составляющих X, Y и Z старого и нового векторов ускорения.

Проблема в том, что y_1 может располагаться внизу объекта *Texture2D*, а y_2 – вверху. В следующем коде я нашел, что проще работать с тремя значениями Y: *oldInsertRow* – это то, что выше я называл y_1, а вот и *newInsertRow* (Новая строка вставки), и *insertRow* (Строка вставки) представляют y_2. Разница в том, что *insertRow* всегда находится в *Texture2D* (то есть меньше высоты растрового изображения), а вот *newInsertRow* может выходить за рамки допустимого диапазона значений. Преимущество *newInsertRow* в том, что его значение всегда гарантированно больше значения *oldInsertRow*. Это несколько упрощает алгоритмы отрисовки линий, поскольку в данном случае не приходится иметь дело с линией, проходящей снизу вверх через все растровое изображение.

Основная задача *Update* – вызов *DrawLines* с передачей в него *oldInsertRow*, *newInsertRow* и старого и нового векторов ускорения:

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    Vector3 acceleration;

    lock (accelerometerVectorLock)
    {
        acceleration = accelerometerVector;
    }

    totalTicks = (int)Math.Round(gameTime.TotalGameTime.TotalSeconds /
                                        this.TargetElapsedTime.TotalSeconds);
    int insertRow = (totalTicks + graphTexture.Height - 1) % graphTexture.Height;

    // newInsertRow гарантированно всегда больше, чем oldInsertRow,
    //  но может быть больше высоты graphTexture!
    int newInsertRow = insertRow < oldInsertRow ? insertRow + graphTexture.Height :
                                              insertRow;
    // Сначала обнуляем pixels
    for (int y = oldInsertRow + 1; y <= newInsertRow; y++)
        for (int x = 0; x < graphTexture.Width; x++)
            pixels[(y % graphTexture.Height) * graphTexture.Width + x] = 0;

    // Отрисовываем три линии на основании старого и нового значений ускорения
    DrawLines(graphTexture, pixels, oldInsertRow, newInsertRow,
              oldAcceleration, acceleration);

    this.GraphicsDevice.Textures[0] = null;

    if (newInsertRow >= graphTexture.Height)
    {
        graphTexture.SetData<uint>(pixels);
    }
    else
    {
```

```
        Rectangle rect = new Rectangle(0, oldInsertRow,
                                    graphTexture.Width, newInsertRow - oldInsertRow
+ 1);
        graphTexture.SetData<uint>(0, rect,
                                    pixels, rect.Y * rect.Width, rect.Height *
rect.Width);
    }

    oldInsertRow = insertRow;
    oldAcceleration = acceleration;

    base.Update(gameTime);
}
```

В конце перегруженного *Update* выполняется обновление *graphTexture* из массива *pixels*. Если *newInsertRow* не выходит за границы растрового изображения, только две строки требуют обновления. В противном случае используется более простая форма вызова *SetData*, решающая проблему «в лоб».

Фактическая отрисовка линий обеспечивается парой методов. Метод *DrawLines* просто разбивает вектор ускорения на три составляющих и выполняет три вызова метода *DrawLine*, в ходе которых вычисляются значения X на основании этих составляющих:

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
void DrawLines(Texture2D texture, uint[] pixels, int oldRow, int newRow,
               Vector3 oldAcc, Vector3 newAcc)
{
    DrawLine(texture, pixels, oldRow, newRow, oldAcc.X, newAcc.X, Color.Red);
    DrawLine(texture, pixels, oldRow, newRow, oldAcc.Y, newAcc.Y, Color.Green);
    DrawLine(texture, pixels, oldRow, newRow, oldAcc.Z, newAcc.Z, Color.Blue);
}

void DrawLine(Texture2D texture, uint[] pixels, int oldRow, int newRow,
              float oldAcc, float newAcc, Color clr)
{
    DrawLine(texture, pixels,
             texture.Width / 2 + (int)(oldAcc * texture.Width / 4), oldRow,
             texture.Width / 2 + (int)(newAcc * texture.Width / 4), newRow, clr);
}
```

Другой метод *DrawLine* реализует простой алгоритм отрисовки линии путем последовательного перебора пикселов на основании того, разница какой из составляющих, X или Y, больше. Я поэкспериментировал здесь с реализацией сглаживания, но мне не удалось добиться хороших результатов.

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
void DrawLine(Texture2D texture, uint[] pixels,
              int x1, int y1, int x2, int y2, Color clr)
{
    if (x1 == x2 && y1 == y2)
    {
        return;
    }

    else if (Math.Abs(y2 - y1) > Math.Abs(x2 - x1))
    {
        int sign = Math.Sign(y2 - y1);

        for (int y = y1; y != y2; y += sign)
```

```
        {
            float t = (float)(y - y1) / (y2 - y1);
            int x = (int)(x1 + t * (x2 - x1) + 0.5f);
            SetPixel(texture, pixels, x, y, clr);
        }
    }
    else
    {
        int sign = Math.Sign(x2 - x1);

        for (int x = x1; x != x2; x += sign)
        {
            float t = (float)(x - x1) / (x2 - x1);
            int y = (int)(y1 + t * (y2 - y1) + 0.5f);
            SetPixel(texture, pixels, x, y, clr);
        }
    }
}

// Обращаем внимание на корректировку Y и применение побитового ИЛИ!
void SetPixel(Texture2D texture, uint[] pixels, int x, int y, Color clr)
{
    pixels[(y % texture.Height) * texture.Width + x] |= clr.PackedValue;
}
```

Метод *SetPixel* (Задать значение пиксела) корректирует координаты Y, которые могут выходить за границы растрового изображения. Также обратите внимание на использование операции ИЛИ. Если синяя и красная линии частично перекрываются, например, тогда перекрывающаяся часть линии будет отрисована пурпурным цветом.

Наконец, метод *Draw* отрисовывает оба объекта *Texture2D*, применяя аналогичную логику, которая обеспечивает многократную отрисовку текстуры до полного заполнения ею экрана:

Проект XNA: AccelerometerGraph Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();

    // Отрисовываем текстуру фона
    int displayRow = -totalTicks % backgroundTexture.Height;

    while (displayRow < displayHeight)
    {
        spriteBatch.Draw(backgroundTexture, new Vector2(0, displayRow),
Color.White);
        displayRow += backgroundTexture.Height;
    }

    // Отрисовываем текстуру графика
    displayRow = -totalTicks % graphTexture.Height;

    while (displayRow < displayHeight)
    {
        spriteBatch.Draw(graphTexture, new Vector2(0, displayRow), Color.White);
        displayRow += graphTexture.Height;
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Следуй за катящимся шаром

В остальных четырех приложениях данной главы поверхность телефона рассматривается как плоскость, по которой может свободно кататься шар.

Сам шар создается в статическом методе *Texture2DExtensions.CreateBall* (Создать шар), который описан в библиотеке Petzold.Phone.Xna:

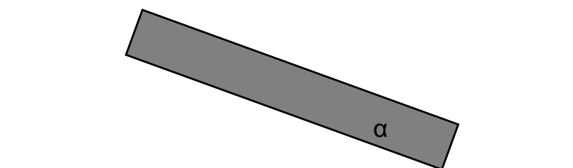
Проект XNA: Petzold.Phone.Xna Файл: Texture2DExtensions.cs (фрагмент)

```
public static Texture2D CreateBall(GraphicsDevice graphicsDevice, int radius)
{
    Texture2D ball = new Texture2D(graphicsDevice, 2 * radius, 2 * radius);
    Color[] pixels = new Color[ball.Width * ball.Height];
    int radiusSquared = radius * radius;

    for (int y = -radius; y < radius; y++)
    {
        int x2 = (int)Math.Round(Math.Sqrt(Math.Pow(radius, 2) - y * y));
        int x1 = -x2;

        for (int x = x1; x < x2; x++)
            pixels[(int)(ball.Width * (y + radius) + x + radius)] = Color.White;
    }
    ball.SetData<Color>(pixels);
    return ball;
}
```

Но это простая часть. Сложность состоит в описании физики перемещения шарика по наклонной плоскости. Начнем с рассмотрения взаимосвязей между вектором акселерометра и углами, описывающими наклонение телефона.

Предположим, телефон просто лежит на плоской поверхности, такой как стол. Представим это следующей схемой:

Теперь приподнимем телефон слева:



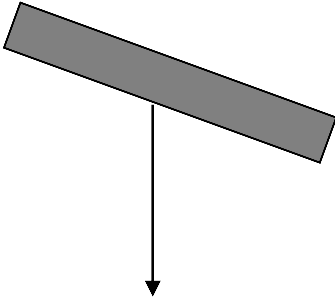
Плоскость телефона образует с поверхностью стола угол α. Можно ли из вектора акселерометра вычислить значение α?

Когда телефон просто лежит на столе, вектор ускорения равен (0, 0, –1). Если наклонить телефон, как показано на рисунке, вектор ускорения, возможно, становится равным (0.34, 0, –0.94). (На первый взгляд можно и не заметить, что квадраты этих чисел в сумме дают 1, но это так.[1])

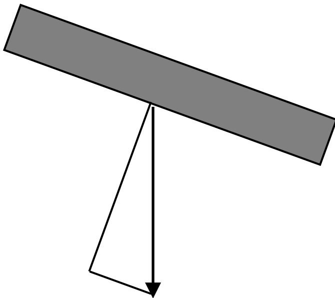
Вектор ускорения всегда направлен к центру земли, т.е. под прямым углом к горизонтальной поверхности:

---

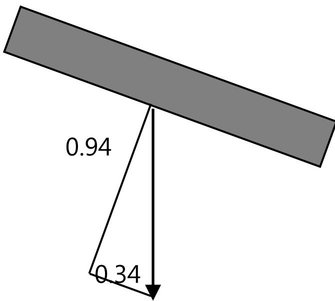
[1] Для простоты дальнейшего описания, приведены округленные значения (*прим. научного редактора*).



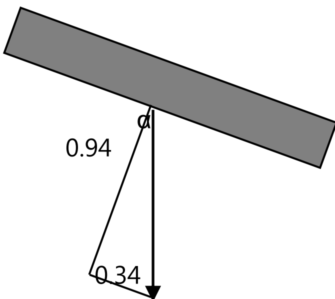
Построим прямоугольный треугольник, где вектор ускорения является гипотенузой, а два катета образованы линиями, параллельными осям X и Z телефона:



Длины катетов равны величинам составляющих X и Z вектора ускорения:



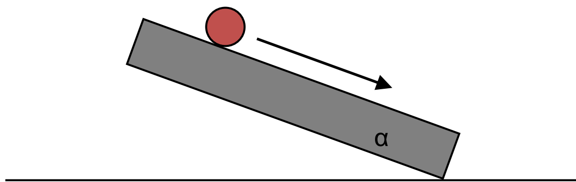
С помощью подобия треугольников можно показать, что угол, противолежащий меньшему катету этого треугольника, равен углу α между плоскостью телефона и горизонтальной поверхностью:



То есть синус α равен 0,34, и косинус – 0,94, а сам α=20°.

Таким образом, угол наклона при наклонении телефона слева направо равен арксинусу составляющей X вектора ускорения; аналогично угол наклона при наклонении телефона в направлении сверху вниз равен арксинусу составляющей Y вектора ускорения.

Теперь скатим шар по поверхности этого наклоненного телефона. Шар, катящийся по наклонной плоскости, безусловно, испытывает меньшее ускорение, чем шар в свободном падении:



Галилей при изучении процесса ускорения использовал шар, катящийся по наклонной поверхности, как замедленную модель свободного падения.

Вычисление ускорения катящегося шара несколько запутанно (для примера, ознакомьтесь с работой A. P. French, *Newtonian Mechanics*[1], W. W. Norton, 1971, страницы 652-653), но без учета трения все сильно упрощается:

ускорение= $\frac{2}{3} \cdot g \cdot \sin(\alpha)$

где g – это ускорение свободного падения, составляющее 9,8 м/с^2. Уже такого количества деталей более чем достаточно для реализации катящегося шара в простом приложении на Windows Phone 7. На самом деле нам достаточно знать, что ускорение пропорционально синусу α. И это чрезвычайно неожиданно, поскольку означает, что ускорение катящегося поперек поверхности телефона шара (при портретной ориентации) пропорционально составляющей X вектора ускорения! Для шара, катящегося вдоль по поверхности, ускорение – это двухмерный вектор, который может быть найден прямо из составляющих X и Y вектора акселерометра.

Приложение TiltAndRoll (Наклон и качение) моделирует перекатывание шара по поверхности экрана на основании наклонения телефона. Когда шар касается одного из краев экрана, он не отскакивает, а, напротив, теряет всю скорость в направлении, перпендикулярном краю. Мяч продолжает катиться вдоль края, если такое поведение соответствует наклонению телефона.

В приложении TiltAndRoll двухмерный вектор ускорения вычисляется из трехмерного вектора акселерометра и умножается на константу GRAVITY (гравитация), единицами измерения которой являются пикселы в секунду в квадрате. Теоретически вычислить значение GRAVITY можно, умножив 9,8 м/с^2 на 39,37 дюймов в метре, и затем на 264 пиксела в дюйме, и еще на 2/3. В результате получим значение около 68000, но на практике оно обусловливает слишком быстрое ускорение шара. Я выбрал значение несколько поменьше, что создало эффект перемещения шара в вязкой жидкости:

Проект XNA: TiltAndRoll Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
```

[1] А.П.Френч, Классическая механика (прим. переводчика).

```
    const float GRAVITY = 1000;      // пикселов в секунду в квадрате
    const int BALL_RADIUS = 16;
    const int BALL_SCALE = 16;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    Texture2D ball;
    Vector2 ballCenter;
    Vector2 ballPosition;
    Vector2 ballVelocity = Vector2.Zero;
    Vector3 oldAcceleration, acceleration;
    object accelerationLock = new object();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Задаем портретную ориентацию как единственно возможную
        graphics.SupportedOrientations = DisplayOrientation.Portrait;
        graphics.PreferredBackBufferWidth = 480;
        graphics.PreferredBackBufferHeight = 768;
    }
    …
}
```

Конечно, если хотите, вы можете увеличить значение GRAVITY.

Чтобы упростить вычисления, конструктор разрешает только портретную ориентацию. Составляющие X и Y вектора акселерометра будут соответствовать координатам экрана, только оси Y акселерометра и экрана являются противоположно направленными.

Метод *Initialize* запускает акселерометр, и обработчик событий *ReadingChanged* самостоятельно обрабатывает сглаживание значений:

Проект XNA: TiltAndRoll Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try { accelerometer.Start(); }
    catch {}

    base.Initialize();
}

void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accelerationLock)
    {
        acceleration = 0.5f * oldAcceleration +
                       0.5f * new Vector3((float)args.X, (float)args.Y,
(float)args.Z);
        oldAcceleration = acceleration;
    }
}
```

Как можно заметить в коде выше, радиус и масштаб шара заданы константами BALL_RADIUS и BALL_SCALE. Поскольку метод *Texture2DExtensions.CreateBall* не делает попытки реализации сглаживания, более гладкое изображение можно получить, сделав шар больше отображаемого размера и заставив XNA выполнять некоторое сглаживание при формировании его визуального представления. При создании радиус шара определяется как произведение BALL_RADIUS и BALL_SCALE, но впоследствии при отображении к нему применяется коэффициент масштабирования 1 / BALL_SCALE.

Проект XNA: TiltAndRoll Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    viewport = this.GraphicsDevice.Viewport;
    ball = Texture2DExtensions.CreateBall(this.GraphicsDevice,
                                          BALL_RADIUS * BALL_SCALE);

    ballCenter = new Vector2(ball.Width / 2, ball.Height / 2);
    ballPosition = new Vector2(viewport.Width / 2, viewport.Height / 2);
}
```

ballPosition (Местоположение шара), инициализированный в *LoadContent* – это точка, хранящаяся как объект *Vector2*. Скорость также хранится как объект *Vector2*, но это действительный вектор, выраженный в пикселах в секунду. Скорость будет меняться только при соударении шара с краями экрана или при наклонении телефона, во всех остальных случаях скорость будет оставаться неизменной благодаря эффекту инерции. Все эти вычисления выполняются в перегруженном *Update*:

Проект XNA: TiltAndRoll Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Вычисляем новую скорость и координаты
    Vector2 acceleration2D = Vector2.Zero;

    lock (accelerationLock)
    {
        acceleration2D = new Vector2(acceleration.X, -acceleration.Y);
    }
    float elapsedSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
    ballVelocity += GRAVITY * acceleration2D * elapsedSeconds;
    ballPosition += ballVelocity * elapsedSeconds;

    // Проводим проверку на соударение с краем
    if (ballPosition.X - BALL_RADIUS < 0)
    {
        ballPosition.X = BALL_RADIUS;
        ballVelocity.X = 0;
    }
    if (ballPosition.X + BALL_RADIUS > viewport.Width)
    {
        ballPosition.X = viewport.Width - BALL_RADIUS;
        ballVelocity.X = 0;
    }
    if (ballPosition.Y - BALL_RADIUS < 0)
```

```
    {
        ballPosition.Y = BALL_RADIUS;
        ballVelocity.Y = 0;
    }
    if (ballPosition.Y + BALL_RADIUS > viewport.Height)
    {
        ballPosition.Y = viewport.Height - BALL_RADIUS;
        ballVelocity.Y = 0;
    }
    base.Update(gameTime);
}
```

Два самых важных вычисления здесь:

```
ballVelocity += GRAVITY * acceleration2D * elapsedSeconds;
ballPosition += ballVelocity * elapsedSeconds;
```

Вектор *acceleration2D* – это просто вектор показаний акселерометра без составляющей Z и с инвертированной координатой Y. Вектор скорости зависит от вектора ускорения и истекшего времени в секундах. Местоположение шара определяется результирующим вектором скорости и истекшим временем в секундах.

В этом прелесть векторов, что нам не нужно знать, наклонен ли телефон в одном направлении со скоростью (и, следовательно, обусловливает увеличение этой скорости), или в противоположную строну (в случае чего скорость уменьшается), или под каким-то другим углом, не имеющим никакого влияния на вектор скорости.

Выражения *if*, завершающие обработку в *Update*, обеспечивают проверку выхода шара за край экрана. В этом случае шар возвращается в окно просмотра, и соответствующая составляющая скорости обнуляется. Здесь не используются выражения *if* и *else*, поскольку шар может одновременно выйти за два края, и этот вариант также должен обрабатываться. В такой ситуации шар останавливается в углу экрана.

Метод *Draw* просто отрисовывает шар с применением коэффициента масштабирования:

Проект XNA: TiltAndRoll   Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();
    spriteBatch.Draw(ball, ballPosition, null, Color.Pink, 0,
                     ballCenter, 1f / BALL_SCALE, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Приложение TiltAndBounce (Наклон и отскок) очень похоже на TiltAndRoll, только в нем шар отскакивает от краев. Это означает, что когда шар касается краев экрана, одна из составляющих его скорости меняет знак на противоположный. Например, если при соударении с правым или левым краем экрана скорость шара была **(*x*, *y*)**, когда он отскакивает, его скорость становится **(–*x*, *y*)**. Но это не соответствует законам физики. Шар должен терять часть скорости при соударении. Чтобы реализовать это, в поля приложения TiltAndBounce включен коэффициент затухания 2/3:

Проект XNA: TiltAndBounce   Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float GRAVITY = 1000;     // пикселы в секунду в квадрате
    const float BOUNCE = 2f / 3;    // доля скорости
    const int BALL_RADIUS = 16;
    const int BALL_SCALE = 16;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    Texture2D ball;
    Vector2 ballCenter;
    Vector2 ballPosition;
    Vector2 ballVelocity = Vector2.Zero;
    Vector3 oldAcceleration, acceleration;
    object accelerationLock = new object();
    …
}
```

Шар, имеющий до соударения с правым или левым краем экрана скорость **(*x*, *y*)**, после соударения перемещается со скоростью **(–BOUNCE·*x*, *y*)**.

Конструктор аналогичен приложению TiltAndRoll, как и перегруженный *Initialize*, и методы *ReadingChanged*, *LoadContent* и *Draw*. Единственным отличием в *Update* является реализация отскока шара в логике соударения с краем:

Проект XNA: TiltAndBounce Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Вычисляем новую скорость и координаты
    Vector2 acceleration2D = Vector2.Zero;

    lock (accelerationLock)
    {
        acceleration2D = new Vector2(acceleration.X, -acceleration.Y);
    }
    float elapsedSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
    ballVelocity += GRAVITY * acceleration2D * elapsedSeconds;
    ballPosition += ballVelocity * elapsedSeconds;

    // Проверяем возможность отскока от края
    bool needAnotherLoop = false;

    do
    {
        needAnotherLoop = false;

        if (ballPosition.X - BALL_RADIUS < 0)
        {
            ballPosition.X = -ballPosition.X + 2 * BALL_RADIUS;
            ballVelocity.X *= -BOUNCE;
            needAnotherLoop = true;
        }
        else if (ballPosition.X + BALL_RADIUS > viewport.Width)
        {
            ballPosition.X = -ballPosition.X - 2 * (BALL_RADIUS - viewport.Width);
            ballVelocity.X *= -BOUNCE;
            needAnotherLoop = true;
        }
```

```
        else if (ballPosition.Y - BALL_RADIUS < 0)
        {
            ballPosition.Y = -ballPosition.Y + 2 * BALL_RADIUS;
            ballVelocity.Y *= -BOUNCE;
            needAnotherLoop = true;
        }
        else if (ballPosition.Y + BALL_RADIUS > viewport.Height)
        {
            ballPosition.Y = -ballPosition.Y - 2 * (BALL_RADIUS - viewport.Height);
            ballVelocity.Y *= -BOUNCE;
            needAnotherLoop = true;
        }
    }
    while (needAnotherLoop);

    base.Update(gameTime);
}
```

В предыдущем приложении шар мог выходить за два края одновременно, но это поведение могло быть обработано с помощью ряда выражений *if*. Логика отскока фактически меняет местоположение шара, в результате чего он может выйти за другой край. Поэтому в данном случае координаты шара должны проверяться постоянно до его стабилизации.

Можно даже расширить эту логику обработки отскока до простой игры. Приложение EdgeSlam (Удар по краям) очень похоже на TiltAndBounce. В этой игре она из сторон экрана выделяется белой линией. Цель – ударить шаром в этот край экрана. Как только шар ударяет выделенную сторону, произвольным образом выделяется одна из других сторон экрана. За каждое соударение с соответствующей стороной игрок получает 1 балл, за соударение с неправильной стороной с игрока снимается 5 штрафных очка. Счет отображается в центре экрана.

Все довольно просто и замечательно до первой ошибки, необходимость обработки которой обычно все усложняет (игры очень похожи на жизнь в этом смысле).

Кроме всех тех же констант, которые мы видели ранее, поля включают еще две новые для подсчета очков:

Проект XNA: EdgeSlam Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float GRAVITY = 1000;     // пикселы в секунду в квадрате
    const float BOUNCE = 2f / 3;    // доля скорости
    const int BALL_RADIUS = 16;
    const int BALL_SCALE = 16;
    const int HIT = 1;
    const int PENALTY = -5;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    Vector2 screenCenter;

    SpriteFont segoe96;
    int score;
    StringBuilder scoreText = new StringBuilder();
    Vector2 scoreCenter;

    Texture2D tinyTexture;
    int highlightedSide;
    Random rand = new Random();
```

```
    Texture2D ball;
    Vector2 ballCenter;
    Vector2 ballPosition;
    Vector2 ballVelocity = Vector2.Zero;
    Vector3 oldAcceleration, acceleration;
    object accelerationLock = new object();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Задаем портретную ориентацию как единственно возможную
        graphics.SupportedOrientations = DisplayOrientation.Portrait;
        graphics.PreferredBackBufferWidth = 480;
        graphics.PreferredBackBufferHeight = 768;
    }
    …
}
```

SpriteFont используется для отображения счета большими цифрами в центре экрана. Объект *tinyTexture* обеспечивает выделение одной из сторон, которая выбирается случайным образом и обозначается значением *highlightedSide* (Выделенная сторона).

Перегруженный метод *Initialize* и метод акселерометра *ReadingChanged* аналогичны виденным ранее. *LoadContent* создает *tinyTexture* и загружает шрифт, а также создает шар:

Проект XNA: EdgeSlam Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    viewport = this.GraphicsDevice.Viewport;
    screenCenter = new Vector2(viewport.Width / 2, viewport.Height / 2);

    ball = Texture2DExtensions.CreateBall(this.GraphicsDevice,
                                          BALL_RADIUS * BALL_SCALE);

    ballCenter = new Vector2(ball.Width / 2, ball.Height / 2);
    ballPosition = screenCenter;

    tinyTexture = new Texture2D(this.GraphicsDevice, 1, 1);
    tinyTexture.SetData<Color>(new Color[] { Color.White });

    segoe96 = this.Content.Load<SpriteFont>("Segoe96");
}
```

Метод *Update* начинается так же, как и в приложении TiltAndBounce, но большой цикл *do* несколько сложнее в данном случае. При соударении шара с одним из краев требуется скорректировать счет в зависимости от того, в какую из сторон попал шар: выделенную или нет:

Проект XNA: EdgeSlam Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
```

```
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Вычисляем новую скорость и координаты
    Vector2 acceleration2D = Vector2.Zero;

    lock (accelerationLock)
    {
        acceleration2D = new Vector2(acceleration.X, -acceleration.Y);
    }
    float elapsedSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
    ballVelocity += GRAVITY * acceleration2D * elapsedSeconds;
    ballPosition += ballVelocity * elapsedSeconds;

    // Проверяем возможность отскока от края
    bool needAnotherLoop = false;
    bool needAnotherSide = false;

    do
    {
        needAnotherLoop = false;

        if (ballPosition.X - BALL_RADIUS < 0)
        {
            score += highlightedSide == 0 ? HIT : PENALTY;
            ballPosition.X = -ballPosition.X + 2 * BALL_RADIUS;
            ballVelocity.X *= -BOUNCE;
            needAnotherLoop = true;
        }
        else if (ballPosition.X + BALL_RADIUS > viewport.Width)
        {
            score += highlightedSide == 2 ? HIT : PENALTY;
            ballPosition.X = -ballPosition.X - 2 * (BALL_RADIUS - viewport.Width);
            ballVelocity.X *= -BOUNCE;
            needAnotherLoop = true;
        }
        else if (ballPosition.Y - BALL_RADIUS < 0)
        {
            score += highlightedSide == 1 ? HIT : PENALTY;
            ballPosition.Y = -ballPosition.Y + 2 * BALL_RADIUS;
            ballVelocity.Y *= -BOUNCE;
            needAnotherLoop = true;
        }
        else if (ballPosition.Y + BALL_RADIUS > viewport.Height)
        {
            score += highlightedSide == 3 ? HIT : PENALTY;
            ballPosition.Y = -ballPosition.Y - 2 * (BALL_RADIUS - viewport.Height);
            ballVelocity.Y *= -BOUNCE;
            needAnotherLoop = true;
        }
        needAnotherSide |= needAnotherLoop;
    }
    while (needAnotherLoop);

    if (needAnotherSide)
    {
        scoreText.Remove(0, scoreText.Length);
        scoreText.Append(score);
        scoreCenter = segoe96.MeasureString(scoreText) / 2;
        highlightedSide = rand.Next(4);
    }

    base.Update(gameTime);
}
```

Если в ходе обработки отскока переменная *needAnotherSide* (Необходима другая сторона) принимает значение *true*, перегруженный *Update* завершается обновлением объекта *StringBuilder* под именем *scoreText* (Счет) и выбором новой стороны-мишени случайным образом. Поле *scoreText* остается незаданным, пока игрок не заработал каких-то баллов. Сначала я задавал этому полю значение нуль, но в начале игры шар расположен в центре экрана, и шар внутри нуля выглядел очень странно!

Перегруженный метод *Draw*, исходя из *highlightedSide*, определяет, где должен размещаться *tinyTexture*, и также отвечает за отображение счета:

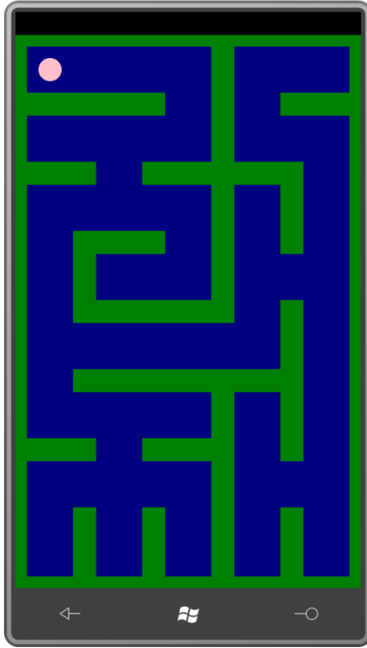
Проект XNA: EdgeSlam Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();

    Rectangle rect = new Rectangle();

    switch (highlightedSide)
    {
        case 0: rect = new Rectangle(0, 0, 3, viewport.Height); break;
        case 1: rect = new Rectangle(0, 0, viewport.Width, 3); break;
        case 2: rect = new Rectangle(viewport.Width - 3, 0, 3, viewport.Height);
break;
        case 3: rect = new Rectangle(3, viewport.Height - 3, viewport.Width, 3);
break;
    }

    spriteBatch.Draw(tinyTexture, rect, Color.White);

    spriteBatch.DrawString(segoe96, scoreText, screenCenter,
                           Color.LightBlue, 0,
                           scoreCenter, 1, SpriteEffects.None, 0);

    spriteBatch.Draw(ball, ballPosition, null, Color.Pink, 0,
                     ballCenter, 1f / BALL_SCALE, SpriteEffects.None, 0);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Лабиринт

Одним из самых естественных применений функциональности катящегося шара является игра лабиринт. Несомненно, в связи с этим необходимо решить несколько проблем, включая создание случайного лабиринта и реализацию логики, которая обеспечит удержание шара в границах «коридоров» лабиринта.

Немного изучив алгоритмы поколений лабиринтов, я остановил свой выбор на, возможно, самом простом. Он называется «рекурсивное деление» и обеспечивает создание лабиринта, который выглядит примерно, как показано на рисунке:



Как видим, это не очень интересный лабиринт, но его основным преимуществом является то, что из любой его области можно попасть в любую другую область.

Концептуально вся область лабиринта – это сетка ячеек. В данном примере мы видим пять ячеек по горизонтали и восемь ячеек по вертикали, всего 40. Каждая из этих ячеек может быть окружена «стеной» максимум с трех из сторон. Рассмотрим простую открытую структуру из библиотеки Petzold.Phone.Xna, которая описывает такую ячейку:

Проект: Petzold.Phone.Xna Файл: MazeCell.cs

```
namespace Petzold.Phone.Xna
{
    public struct MazeCell
    {
        public bool HasLeft { internal set; get; }
        public bool HasTop { internal set; get; }
        public bool HasRight { internal set; get; }
        public bool HasBottom { internal set; get; }

        public MazeCell(bool left, bool top, bool right, bool bottom) : this()
        {
            HasLeft = left;
            HasTop = top;
            HasRight = right;
            HasBottom = bottom;
        }
    }
}
```

Свойство *HasTop* (Закрыта сверху) всех верхних ячеек имеет значение *true*; свойство *HasLeft* (Закрыта слева) имеет значение *true* для всех ячеек вдоль левого края; и аналогично для правого края и низа.

Массив объектов *MazeCell* (Ячейка лабиринта), составляющих лабиринт, создается и сохраняется в объекте *MazeGrid* (Сетка лабиринта). Единственный конструктор *MazeGrid* принимает ширину и высоту, выраженные числом ячеек (например, 5 и 8 для приведенного

выше примера). Рассмотрим конструктор *MazeGrid*, три открытых свойства, а также объект *Random*:

Проект: Petzold.Phone.Xna Файл: MazeGrid.cs (фрагмент)

```
public class MazeGrid
{
    Random rand = new Random();

    public MazeGrid(int width, int height)
    {
        Width = width;
        Height = height;
        Cells = new MazeCell[Width, Height];

        for (int y = 0; y < Height; y++)
            for (int x = 0; x < Width; x++)
            {
                Cells[x, y].HasLeft = x == 0;
                Cells[x, y].HasTop = y == 0;
                Cells[x, y].HasRight = x == Width - 1;
                Cells[x, y].HasBottom = y == Height - 1;
            }

        MazeChamber rootChamber = new MazeChamber(0, 0, Width, Height);
        DivideChamber(rootChamber);
    }

    public int Width { protected set; get; }
    public int Height { protected set; get; }
    public MazeCell[,] Cells { protected set; get; }
    …
}
```

Конструктор *MazeGrid* завершается созданием объекта типа *MazeChamber* (Секция лабиринта), размер которого соответствует размеру *MazeGrid*, и вызовом рекурсивного метода *DivideChamber* (Разделить секцию), который мы вскоре рассмотрим. В алгоритме формирования лабиринта секция – это прямоугольная сетка ячеек, внутренняя область которой освобождена от перегородок. Каждая секция сначала делится надвое посредством установления произвольным образом (но, как правило, вдоль меньшего размера) перегородки с одним просветом, который также размещается произвольным образом. В результате этого процесса создаются две секции, соединенные этим просветом. Такое последовательное деление на секции продолжается до тех пор, пока секции не достигают размера ячейки.

MazeChamber – это класс библиотеки Petzold.Phone.Xna. Рассмотрим полный код класса, который имеет собственное поле *Random*:

Проект: Petzold.Phone.Xna Файл: MazeChamber.cs

```
using System;

namespace Petzold.Phone.Xna
{
    internal class MazeChamber
    {
        static Random rand = new Random();

        public MazeChamber(int x, int y, int width, int height)
            : base()
        {
```

```
            X = x;
            Y = y;
            Width = width;
            Height = height;
        }

        public int X { protected set; get; }
        public int Y { protected set; get; }
        public int Width { protected set; get; }
        public int Height { protected set; get; }

        public MazeChamber Chamber1 { protected set; get; }
        public MazeChamber Chamber2 { protected set; get; }

        public int Divide(bool divideWidth)
        {
            if (divideWidth)
            {
                int col = rand.Next(X + 1, X + Width - 1);
                Chamber1 = new MazeChamber(X, Y, col - X, Height);
                Chamber2 = new MazeChamber(col, Y, X + Width - col, Height);
                return col;
            }
            else
            {
                int row = rand.Next(Y + 1, Y + Height - 1);
                Chamber1 = new MazeChamber(X, Y, Width, row - Y);
                Chamber2 = new MazeChamber(X, row, Width, Y + Height - row);
                return row;
            }
        }
    }
}
```

Метод *Divide* (Разделить) фактически разделяет одну секцию на две на основании выбранных случайным образом строки и столбца и создает два новых объекта *MazeChamber*. Рекурсивный метод *DivideChamber* объекта *MazeGrid* отвечает за вызов этого метода *Divide* и окружение получившихся ячеек перегородками с трех сторон:

Проект: Petzold.Phone.Xna Файл: MazeGrid.cs (фрагмент)

```
void DivideChamber(MazeChamber chamber)
{
    if (chamber.Width == 1 && chamber.Height == 1)
    {
        return;
    }

    bool divideWidth = chamber.Width > chamber.Height;

    if (chamber.Width == 1 || chamber.Height >= 2 * chamber.Width)
    {
        divideWidth = false;
    }
    else if (chamber.Height == 1 || chamber.Width >= 2 * chamber.Height)
    {
        divideWidth = true;
    }
    else
    {
        divideWidth = Convert.ToBoolean(rand.Next(2));
    }

    int rowCol = chamber.Divide(divideWidth);
```

```
    if (divideWidth)
    {
        int col = rowCol;
        int gap = rand.Next(chamber.Y, chamber.Y + chamber.Height);

        for (int y = chamber.Y; y < chamber.Y + chamber.Height; y++)
        {
            Cells[col - 1, y].HasRight = y != gap;
            Cells[col, y].HasLeft = y != gap;
        }
    }
    else
    {
        int row = rowCol;
        int gap = rand.Next(chamber.X, chamber.X + chamber.Width);

        for (int x = chamber.X; x < chamber.X + chamber.Width; x++)
        {
            Cells[x, row - 1].HasBottom = x != gap;
            Cells[x, row].HasTop = x != gap;
        }
    }

    DivideChamber(chamber.Chamber1);
    DivideChamber(chamber.Chamber2);
}
```

Также я понял, что должен обобщить логику отскока, и для этого мне необходимо найти хороший способ представления геометрического сегмента линия, который бы позволял вычислять точки пересечения и осуществлять другие полезные операции. Эту структуру я назвал *Line2D*. Сегмент линия определяется двумя точками, которые также описывают свойство *Vector* и свойство *Normal* (Нормаль) (перпендикуляр к вектору), таким образом, концептуально линия имеет направление и также «внутреннюю» и «внешнюю» стороны.

Проект: Petzold.Phone.Xna Файл: Line2D.cs

```
using System;
using Microsoft.Xna.Framework;

namespace Petzold.Phone.Xna
{
    // представляем линию как pt1 + t(pt2 - pt1)
    public struct Line2D
    {
        public Line2D(Vector2 pt1, Vector2 pt2) : this()
        {
            Point1 = pt1;
            Point2 = pt2;

            Vector = Point2 - Point1;
            Normal = Vector2.Normalize(new Vector2(-Vector.Y, Vector.X));
        }

        public Vector2 Point1 { private set; get; }
        public Vector2 Point2 { private set; get; }
        public Vector2 Vector { private set; get; }
        public Vector2 Normal { private set; get; }
        public float Angle
        {
            get
            {
                return (float)Math.Atan2(this.Point2.Y - this.Point1.Y,
                                         this.Point2.X - this.Point1.X);
            }
        }
```

```
        public Line2D Shift(Vector2 shift)
        {
            return new Line2D(this.Point1 + shift, this.Point2 + shift);
        }

        public Line2D ShiftOut(Vector2 shift)
        {
            Line2D shifted = Shift(shift);
            Vector2 normalizedVector = Vector2.Normalize(Vector);
            float length = shift.Length();

            return new Line2D(shifted.Point1 - length * normalizedVector,
                              shifted.Point2 + length * normalizedVector);
        }

        public Vector2 Intersection(Line2D line)
        {
            float tThis, tThat;

            IntersectTees(line, out tThis, out tThat);

            return Point1 + tThis * (Point2 - Point1);
        }

        public Vector2 SegmentIntersection(Line2D line)
        {
            float tThis, tThat;

            IntersectTees(line, out tThis, out tThat);

            if (tThis < 0 || tThis > 1 || tThat < 0 || tThat > 1)
                return new Vector2(float.NaN, float.NaN);

            return Point1 + tThis * (Point2 - Point1);
        }

        void IntersectTees(Line2D line, out float tThis, out float tThat)
        {
            float den = line.Vector.Y * this.Vector.X - line.Vector.X *
this.Vector.Y;

            tThis = (line.Vector.X * (this.Point1.Y - line.Point1.Y) -
                     line.Vector.Y * (this.Point1.X - line.Point1.X)) / den;

            tThat = (this.Vector.X * (this.Point1.Y - line.Point1.Y) -
                     this.Vector.Y * (this.Point1.X - line.Point1.X)) / den;
        }

        public override string ToString()
        {
            return String.Format("{0} --> {1}", this.Point1, this.Point2);
        }

        public static bool IsValid(Vector2 vector)
        {
            return !Single.IsNaN(vector.X) && !Single.IsInfinity(vector.X) &&
                   !Single.IsNaN(vector.Y) && !Single.IsInfinity(vector.Y);
        }
    }
}
```

Поскольку линия описывается параметрическими уравнениями, не составляет особого труда найти точки пересечения, подставляя предполагаемые значения *t*, связанные с точкой на линии.

Все эти предварительные мероприятия являются подготовкой к проекту TiltMaze (Лабиринт с наклонением). Поля проекта включают *tinyTexture*, используемый для отображения перегородок сетки. Чрезвычайно важную роль в этом приложении играет коллекция *List* объектов *Line2D* под именем *borders* (рамки). Объекты *Line2D* в коллекции *borders* определяют контуры перегородок, разделяющих ячейки. Каждая перегородка имеет ширину, которая определяется константой WALL_WIDTH.

Проект XNA: TiltMaze Файл: Game1.cs (фрагмент, демонстрирующий поля)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    const float GRAVITY = 1000;     // пикселы в секунду в квадрате
    const float BOUNCE = 2f / 3;    // доля скорости
    const int BALL_RADIUS = 16;
    const int BALL_SCALE = 16;
    const int WALL_WIDTH = 32;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Viewport viewport;
    Texture2D tinyTexture;

    MazeGrid mazeGrid = new MazeGrid(5, 8);
    List<Line2D> borders = new List<Line2D>();

    Texture2D ball;
    Vector2 ballCenter;
    Vector2 ballPosition;
    Vector2 ballVelocity = Vector2.Zero;
    Vector3 oldAcceleration, acceleration;
    object accelerationLock = new object();

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        // Частота кадров по умолчанию для устройств Windows Phone - 30 кадров/с
        TargetElapsedTime = TimeSpan.FromTicks(333333);

        // Задаем портретную ориентацию как единственно возможную
        graphics.SupportedOrientations = DisplayOrientation.Portrait;
        graphics.PreferredBackBufferWidth = 480;
        graphics.PreferredBackBufferHeight = 768;
    }
    …
}
```

Как обычно, перегруженный *Initialize* определяет объект *Accelerometer*, и обработчик *ReadingChanged* сохраняет сглаженное значение.

Проект XNA: TiltMaze Файл: Game1.cs (фрагмент)

```
protected override void Initialize()
{
    Accelerometer accelerometer = new Accelerometer();
    accelerometer.ReadingChanged += OnAccelerometerReadingChanged;

    try { accelerometer.Start(); }
    catch { }

    base.Initialize();
```

```
}

void OnAccelerometerReadingChanged(object sender, AccelerometerReadingEventArgs
args)
{
    lock (accelerationLock)
    {
        acceleration = 0.5f * oldAcceleration +
                       0.5f * new Vector3((float)args.X, (float)args.Y,
(float)args.Z);
        oldAcceleration = acceleration;
    }
}
```

Большая часть метода *LoadContent* посвящена построению коллекции *borders*, и я совершенно недоволен этим кодом. (Он в моем первоочередном списке на переработку, как только у меня найдется свободное время.) Данный код рассматривает отдельно каждую ячейку и затем отдельно каждую сторону этой ячейки. Если с определенной стороны ячейки имеется перегородка, рамка этой ячейки определяется тремя объектами *Line2D*:

Проект XNA: TiltMaze Файл: Game1.cs (фрагмент)

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который может использоваться для отрисовки текстур
    spriteBatch = new SpriteBatch(GraphicsDevice);

    viewport = this.GraphicsDevice.Viewport;

    // Создаем текстуру для перегородок лабиринта
    tinyTexture = new Texture2D(this.GraphicsDevice, 1, 1);
    tinyTexture.SetData<Color>(new Color[] { Color.White });

    // Создаем шар
    ball = Texture2DExtensions.CreateBall(this.GraphicsDevice,
                                          BALL_RADIUS * BALL_SCALE);

    ballCenter = new Vector2(ball.Width / 2, ball.Height / 2);
    ballPosition = new Vector2((viewport.Width / mazeGrid.Width) / 2,
                               (viewport.Height / mazeGrid.Height) / 2);

    // Инициализируем коллекцию borders
    borders.Clear();

    // Создаем объекты Line2D для перегородок лабиринта
    int cellWidth = viewport.Width / mazeGrid.Width;
    int cellHeight = viewport.Height / mazeGrid.Height;
    int halfWallWidth = WALL_WIDTH / 2;

    for (int x = 0; x < mazeGrid.Width; x++)
        for (int y = 0; y < mazeGrid.Height; y++)
        {
            MazeCell mazeCell = mazeGrid.Cells[x, y];
            Vector2 ll = new Vector2(x * cellWidth, (y + 1) * cellHeight);
            Vector2 ul = new Vector2(x * cellWidth, y * cellHeight);
            Vector2 ur = new Vector2((x + 1) * cellWidth, y * cellHeight);
            Vector2 lr = new Vector2((x + 1) * cellWidth, (y + 1) * cellHeight);
            Vector2 right = halfWallWidth * Vector2.UnitX;
            Vector2 left = -right;
            Vector2 down = halfWallWidth * Vector2.UnitY;
            Vector2 up = -down;

            if (mazeCell.HasLeft)
            {
                borders.Add(new Line2D(ll + down, ll + down + right));
```

```
                borders.Add(new Line2D(ll + down + right, ul + up + right));
                borders.Add(new Line2D(ul + up + right, ul + up));
            }
            if (mazeCell.HasTop)
            {
                borders.Add(new Line2D(ul + left, ul + left + down));
                borders.Add(new Line2D(ul + left + down, ur + right + down));
                borders.Add(new Line2D(ur + right + down, ur + right));
            }
            if (mazeCell.HasRight)
            {
                borders.Add(new Line2D(ur + up, ur + up + left));
                borders.Add(new Line2D(ur + up + left, lr + down + left));
                borders.Add(new Line2D(lr + down + left, lr + down));
            }
            if (mazeCell.HasBottom)
            {
                borders.Add(new Line2D(lr + right, lr + right + up));
                borders.Add(new Line2D(lr + right + up, ll + left + up));
                borders.Add(new Line2D(ll + left + up, ll + left));
            }
        }
}
```

Проблема в том, что в коллекции *borders* слишком много объектов *Line2D*. Они часто дублируются для одной и той же или примыкающих ячеек. Такое дублирование абсолютно избыточно, поскольку все эти объекты *Line2D*, по сути, накладываются на одну общую перегородку.

Сама по себе проблема не так велика, но избыточные объекты *Line2D* оказывают негативное влияние на производительность приложения. Это становится абсолютно очевидным, когда шар скатывается вдоль длинной перегородки. Он, кажется, немного спотыкается, как будто натыкается на одну из этих невидимых границ и отскакивает от нее.

Решение данной проблемы перенесем в логику метода *Update*. Как я говорил ранее, мне требовалось найти более обобщённый метод реализации отражения шара от перегородок. В результате я пришел к следующему подходу, который мне тоже не очень нравится. Он использует коллекцию *borders* и привносит собственные небольшие ошибки:

Проект XNA: TiltMaze Файл: Game1.cs (фрагмент)

```
protected override void Update(GameTime gameTime)
{
    // Обеспечиваем возможность выхода из игры
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Вычисляем новую скорость и координаты
    Vector2 acceleration2D = Vector2.Zero;

    lock (accelerationLock)
    {
        acceleration2D = new Vector2(acceleration.X, -acceleration.Y);
    }
    float elapsedSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
    ballVelocity += GRAVITY * acceleration2D * elapsedSeconds;
    Vector2 oldPosition = ballPosition;
    ballPosition += ballVelocity * elapsedSeconds;

    bool needAnotherLoop = false;

    do
    {
```

```
        needAnotherLoop = false;

        foreach (Line2D line in borders)
        {
            Line2D shiftedLine = line.ShiftOut(BALL_RADIUS * line.Normal);
            Line2D ballTrajectory = new Line2D(oldPosition, ballPosition);
            Vector2 intersection = shiftedLine.SegmentIntersection(ballTrajectory);
            float angleDiff = MathHelper.WrapAngle(line.Angle -
ballTrajectory.Angle);

            if (Line2D.IsValid(intersection) && angleDiff > 0 &&
                Line2D.IsValid(Vector2.Normalize(ballVelocity)))
            {
                float beyond = (ballPosition - intersection).Length();
                ballVelocity = BOUNCE * Vector2.Reflect(ballVelocity, line.Normal);
                ballPosition = intersection + beyond *
Vector2.Normalize(ballVelocity);
                needAnotherLoop = true;
                break;
            }
        }
    }
    while (needAnotherLoop);

    base.Update(gameTime);
}
```

Для каждого объекта *Line2D* в коллекции *borders* этот код вызывает метод *ShiftOut* (Выдвинуть) структуры, которая создает еще одну линию на внешней стороне перегородки, отстоящую от линии края на величину BALL_RADIUS и удлиненную на BALL_RADIUS со всех сторон. Я использую этот новый объект *Line2D* как пограничную линию. Она непроницаема для центра шара и обеспечивает поверхность, от которой этот центр шара может отскакивать.

В этом подходе две проблемы. Во-первых, он не годится для углов. Если я на самом деле хочу предотвратить прохождение шара сквозь границу, эта граница в углах должна быть образована дугой, являющейся четвертью окружности. Во-вторых, в отличие от составляющих перегородки объектов *Line2D* этот новый *Line2D* «торчит» из стены, что и обусловливает этот эффект «спотыкания», о котором я говорил ранее.

Перегруженный *Draw* вообще не использует коллекцию *borders*, но реализует подобную логику для отрисовки иногда перекрывающихся прямоугольных текстур:

Проект XNA: TiltMaze Файл: Game1.cs (фрагмент)

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Navy);

    spriteBatch.Begin();

    // Отрисовываем перегородки лабиринта
    int cellWidth = viewport.Width / mazeGrid.Width;
    int cellHeight = viewport.Height / mazeGrid.Height;
    int halfWallWidth = WALL_WIDTH / 2;

    for (int x = 0; x < mazeGrid.Width; x++)
        for (int y = 0; y < mazeGrid.Height; y++)
        {
            MazeCell mazeCell = mazeGrid.Cells[x, y];

            if (mazeCell.HasLeft)
            {
```

```
                Rectangle rect = new Rectangle(x * cellWidth,
                                               y * cellHeight - halfWallWidth,
                                               halfWallWidth, cellHeight +
WALL_WIDTH);
                spriteBatch.Draw(tinyTexture, rect, Color.Green);
            }

            if (mazeCell.HasRight)
            {
                Rectangle rect = new Rectangle((x + 1) * cellWidth - halfWallWidth,
                                               y * cellHeight - halfWallWidth,
                                               halfWallWidth, cellHeight +
WALL_WIDTH);
                spriteBatch.Draw(tinyTexture, rect, Color.Green);
            }

            if (mazeCell.HasTop)
            {
                Rectangle rect = new Rectangle(x * cellWidth - halfWallWidth,
                                               y * cellHeight,
                                               cellWidth + WALL_WIDTH,
halfWallWidth);
                spriteBatch.Draw(tinyTexture, rect, Color.Green);
            }

            if (mazeCell.HasBottom)
            {
                Rectangle rect = new Rectangle(x * cellWidth - halfWallWidth,
                                               (y + 1) * cellHeight - halfWallWidth,
                                               cellWidth + WALL_WIDTH,
halfWallWidth);
                spriteBatch.Draw(tinyTexture, rect, Color.Green);
            }
        }

    // Отрисовываем шар
    spriteBatch.Draw(ball, ballPosition, null, Color.Pink, 0,
                     ballCenter, 1f / BALL_SCALE, SpriteEffects.None, 0);

    spriteBatch.End();

    base.Draw(gameTime);
}
```

Конечно, я знаю, что должен делать: я должен использовать более обобщенный подход при создании этих перегородок. Необходимо знать, сколько перегородок встречается в каждом пересечении, и определять только истинные контуры этих перегородок.

Как и многие программные проекты, которые никогда по-настоящему не заканчиваются, подозреваю, эта книга тоже не окончена.

Предметный указатель

"

"d", описания пространств имен ... 26

"mc", описания пространств имен ... 26

.

.NET-атрибуты ... 142

.NET-свойства, и свойства-зависимости ... 266

{

{bin}\bin\Debug, каталог

файлы XAP ... 32

A

ACCELERATION, константа ... 652

Accelerometer, класс ... 85

метод Dispose ... 85

метод Start ... 85

метод Stop ... 85

свойство State ... 85

AccelerometerReadingEventArgs, аргументы ... 86

accelerometerVector, поле ... 90

accelerometerVectorLock, переменная ... 90

AcceptsReturn, свойство ... 259

Action, свойство ... 59

Activated, события ... 122

ActiveSong, свойство ... 629

ActiveSongChanged, события ... 629

activeStrokes, словарь ... 222

ActualHeight, свойство ... 42

ActualWidth, свойство ... 42

Add New Item, диалоговое окно ... 103

Add Reference, диалоговое окно ... 281

Adder, класс ... 328–29

Album, коллекции ... 620

AlbumCollection, класс ... 620

AlbumInfo, класс ... 620–22

свойство Album ... 622

свойство ShortAlbumName ... 622

AlbumPage, класс ... 629–31

создание экземпляров ... 627

Albums, свойство

типа AlbumCollection ... 620–21

AllowReadStreamBuffering, свойство ... 73

AlternativeSlider, проект ... 533–36

AltSlider, элементы управления ... 536

свойства Style ... 537

AltSliderDemo, проект ... 536–38

AmOrPm, проект ... 324

Analog Clock, приложение

Path Markup Syntax ... 390–94

angle, параметр ... 162

angle, переменная

вычисление ... 655–56, 668

приращение ... 653

Angle, свойство, привязка данных ... 302

AngleX и AngleY, свойства ... 159–60

Animated, присоединенное свойство ... 457

AnimatedInfinity, проект ... 466–67

AnimatedSpiral, проект ... 454

animationStartTime ... 662

App, класс ... 23, 102

создание объекта PhoneApplicationFrame ... 24

хранение данных страницы ... 109–11

App, тег ... 20

App.g.cs, файлы ... 24

метод InitializeComponent 24
проблемы 26
App.xaml, файл
раздел ApplicationLifetimeObjects 619
раздел Resources 148
App.xaml, файлы 23
класс PhoneApplicationService 114
корневой элемент 24
описания пространств имен 24
App.xaml.cs, файлы 23
обработчики событий 122
структура 24
Application, класс 23–24
свойство Resources 145
Application.Current, свойство 44, 123
ApplicationBar
атрибут x:Name 213, 215
доступ из кода 213
значки 210–17
кнопки 210, 427, 434, 473, 484
месторасположения 210
обработчики кнопок 216
образец 211
ориентация 213
свойства ForegroundColor и BackgroundColor 213
свойство Buttons 212
свойство IsVisible 213
свойство Opacity 214
ApplicationBar, свойство 210, 213
ApplicationBarIconButton, объекты 212
атрибут x:Name 216
ApplicationIcon.png, файлы 21
Applications = Code + Markup (Петзольд) 395
ApplyBitSettingsToBitmap 436
apps.game, атрибут 620
Arc, графический элемент
синтаксис 390
ArcSegment, класс 376–82
свойство IsLargeArc 376–78
свойство RotationAngle 380
свойство SweepDirection 376
AreaBoundingBox, объекты 99–100
ARGB-значения цвета 408–10
args, аргумент 43
args.Handled, выражение 66
args.ManipulationContainer, свойство 63
args.OriginalSource, свойство 63
Arrange, метод 194, 197
ArrangeOverride, метод 194–95, 197, 291, 295, 597
перегрузка 195
Artist, свойство
типа Artist 620
Artists, свойство
типа ArtistCollection 620
Ascender Corporation 32
Assisted GPS (A-GPS) 92–95
AutoPlay, свойство 173, 216
AutoReverse, свойство 447
availableChildSize, аргумент 291
availableSize, аргумент 196, 291
неограниченный 290
availableSize.Width, неограниченный 200

B

Back, кнопка 17

для завершения приложений ... 78
использование при навигации ... 116–17
навигация ... 107
Background, свойство ... 136–37, 177
задание для состояния Pressed ... 518–19
значение по умолчанию ... 136
привязки шаблонов ... 533–34
Background.png, файлы ... 21
BareBonesSlider, проект ... 530–33
BeginInvoke, метод ... 87
вызов ... 87–88
перегрузки ... 87
BeginStoryboard, теги ... 462
BeginTime, свойство ... 447
BetterGradientButton, класс ... 270–71
BetterGradientButtonDemo, приложение 267–74
Bevel, соединения ... 354
BezierSegment, класс ... 387, 389
BinaryClock, проект ... 325–28
Binding
для свойства Foreground ... 332
для свойства UpdateSourceTrigger ... 339
Binding, класс ... 300
свойство Converter ... 302
свойство ConverterParameter ... 305
свойство Path ... 300
Binding, объекты ... 300
Binding, расширение разметки ... 299, 500
конвертеры, использование ... 306
порядок свойств ... 300
ресурсы, ссылки на них ... 304
свойство Source ... 315
часть «Path=» ... 317, 319
BindingExpression, объекты ... 339
метод UpdateSource ... 339
BindingMode, перечисление ... 302
BindingMode.OneTime, опция ... 302
BindingMode.OneWay, опция ... 302
BindingMode.TwoWay, опция ... 302
BindingOperations.SetBinding, метод ... 300
BindingValidationError, события ... 340
BindToSelf, проект ... 307
Bing ... 16
Bing Maps ... 95
BitmapCache, свойство ... 453
BitmapImage, класс ... 73–75, 400
свойство CreateOptions ... 401
свойство UriSource ... 401
BitmapImage, объекты ... 78
загрузка по URI ... 402
создаваемые из потоков ... 406
BitmapSource, класс ... 74, 400
метод SetSource ... 400
свойство PixelHeight ... 400
свойство PixelWidth ... 400
BitSelectDialog, элемент управления ... 431, 433
видимость ... 436
BooleanToVisibilityConverter, конвертер ... 323–26
Border, класс
для TextBlock ... 506–8
свойство Child ... 177
свойство Visibility, значение Collapsed ... 593
Border, элементы ... 165–68
в StackPanels ... 183

BorderBrush, свойство 285
явное задание в коде 508
BorderedText, класс 310
BorderedTextDemo, приложение 310
BorderThickness, свойство 166
явное задание в коде 508
Bounds, свойство 365
типа Rect 364
Brush, класс, иерархия класса 175
Brush, объекты, как значения свойства Content 500
btn, переменная 270
Build Action
Content 75
Resource 75
сравнение Content и Resource 75–76
Build Action, поле
Content 211
Button, класс 235–38
RadialGradientBrush, как значение содержимого 499
TextBlock как значение свойства Content 504
визуальная обратная связь 514–22
визуальные состояния 515
вставка изображений 239–41
вставка кнопки 240
значения свойств по умолчанию 509–10
значения свойства Content 239–41
иерархия классов 242–44
настройка представления 506–14
неактивное состояние 514–22
объект Content, отображение 499
описание Style 253
ориентация 235
отображение текста 240
пользовательский класс, унаследованный от 263–66
размер по умолчанию 235
свойства-зависимости 267
свойство Border 236
свойство Content 238
свойство ContentTemplate 500
свойство IsEnabled 237
свойство Padding 513
события Click 238
содержимое 236
содержимое, позиционирование 236
стили 253–54
стиль темы 242
элемент Clock как значение свойства Content 501, 505
элемент Image как значение свойства Content 501
элемент RadialGradientBrush как значение свойства Content 501, 504
ButtonBase, класс 242, 254
ButtonCornucopia, проект 242–44
Buttons, коллекция 212
ButtonSetAndAnimate, приложение 493
ButtonStyles, проект 253–54
ButtonTree, приложение 501–5
By, значение 449

C

C#
создание экземпляров 133
C#, популярность 17
CalculateNewSum, метод 329

CameraCaptureTask, объекты 76, 78
CancelDrag, метод 538
Canvas 177, 200–205, 345–47
  Line 346
  дочерние элементы, компоновка 204
  дочерние элементы, определение размеров 200
  задание размеров 347
  касание 206–7
  методы SetLeft и SetTop 293
  определение размеров 200
  позиционирование элементов 200
  свойства Canvas.Left и Canvas.Top 293, 346–47
  свойства Left и Top 202, 204
  свойство ZIndex 205, 348
Canvas.Left, свойство 293
  анимации 455
  координаты 346–47
  цель анимации 468
Canvas.Top, свойство 293
  анимации 455
  координаты 346–47
  цель анимации 468
CanvasClone, класс 293–97
  дочерний элемент 295
CanvasCloneDemo, проект 293–97
CanvasSetLeft, вызовы 204–5
CanvasSetTop, вызовы 204–5
car.png 664
carCenter, поле 665
CardFile, класс 601, 603
CarOnInfinityCourse, проект 674–76
CarOnOvalCourse, проект 671–73
CarOnPolylineCourse, проект 666–70
CarOnRectangularCourse, проект 664–66
Center, свойство, как цель анимации 469–70, 484
CenterOfRotationX, свойство 490–91
CenterOfRotationY, свойство 490–91
CenterOfRotationZ, свойство 490–91
CenterX и CenterY, свойства 158
CharacterRegions, теги 34
CheckBox, класс 244
Child, свойство 177
Children, коллекции
  компоновка элементов 179–80
  компоновка элементов, переопределение 205
  перебор элементов 195–96
Children, свойство 142, 144, 177, 266
  доступ к системе компоновки 194
  типа TimelineCollection 447
ChooserBase, класс 76
CircularGradient, проект 410–11
Click, обработчики
  анимации, запуск 485
ClickAndSpin, проект 443–46
ClientBounds, свойство 49
Clip, свойство 391
Clock, класс 312–14
Closing, событие 122
CLR-свойство 268–69
  включение 269
Collapsed 185
CollectionChanged, события 570
CollectionViewSource, класс 582–84

объекты SortDescription 584
свойство Source 583
Color, анимации 442
Color, аргумент 657–58
Color, свойство
анимация 468
изменения в нем, выявление 281
Color, структура
в Silverlight 62
метод Lerp 663
Color.Black 658
Color.Blue 658
Color.Green 658
Color.Red 658
ColorAnimation, класс 442, 447
свойство Easing 443
свойство EasingFunction 483
ColorAnimationUsingKeyFrames, класс 443, 447
ColorBits, значения
инициализация 432
ColorBits, массив 431
ColorBitsChanged, события 431
ColorColumn, класс 277
ColorColumn, элементы управления
ссылки x:Name 280
ColorPresenter, класс 561–66
как ресурс 564
свойство Color 564
свойство Colors 564
свойство DisplayName 564
свойство Name 564
Colors, класс 556, 562
ColorScroll, проект 229–34
ComboBox элементы управления
раскрытие 549
ComboBox, класс 547
свойство ItemContainerStyle 553
ComboBox, элементы управления .... 547, см. элементы управления списками
создание экземпляров 548
шаблон по умолчанию 548
ComboBoxItem, класс 553
CommonStates, имя группы 515
Complete, метод
вызов 63
Completed, событие 435
Complex, структура 336–37
ComposerInfo, класс 622
свойство Composer, привязка 626
compositeSize, переменная 196, 291
CompositeTransform, класс 156, 160, 162
без имени 164
CompositionTarget, класс 163
CompositionTarget.Rendering, события. 163, 247, 439, 452–58
замена анимациями Silverlight 454
ComputeTangents 675
Connect, инструмент 619
Content, каталог
добавление содержимого 659
добавление существующих элементов 659
Content, объекты, отображение в деревьях визуальных элементов 499
Content, свойство 111, 238–41, 275
Panel 240
задание кисти как значение 497

присвоение объектов ... 497
теги свойства-элемента ... 238
типа object ... 238–41
типа UIElement ... 241
цвета ... 238–41
ContentControl, класс ... 238
DataTemplate ... 499
в деревьях визуальных элементов ... 505
дерево визуальных элементов после ContentPresenter ... 504–5
для пользовательских элементов управления ... 542
наследование ... 275
производные ... 241
у ContentControl есть свойство Foreground ... 518
элементы TextBlock ... 585
ContentControl, производные ... 275
ControlTemplate и DataTemplate ... 552
свойство Content ... 497
содержимое, отображение ... 512
ContentManager, тип ... 34–35
ContentPanel, поле ... 43
ContentPresenter, класс ... 504, 512
для элементов управления списками ... 552
свойство ContentTemplate ... 512
ContentProperty, атрибуты ... 142–43, 168
ContentTemplate, свойство ... 505
привязка ... 512
Control, класс ... 27
для пользовательских элементов управления ... 542
защищенные виртуальные методы ... 229
иерархия классов ... 241
наследование ... 227–29, 275
свойства ... 228
свойства Border и TextBlock ... 236
свойство HorizontalContentAlignment 237, 514
свойство IsEnabled ... 228, 237
свойство IsTabStop ... 228
свойство Padding ... 237
свойство TabIndex ... 228
свойство TabNavigation ... 228
свойство Template ... 228
свойство VerticalContentAlignment ... 237, 514
ControlTemplate ... 497, 505–14, см. также шаблоны
для элементов управления списками ... 552
назначение ... 505
по умолчанию ... 552
представление элементов управления ... 552
редактирование ... 522
свойство-элемент, задание ... 506
ControlTemplate, свойство
для элементов управления списками ... 557
Convert, метод ... 303
аргумент parameter ... 305
аргумент value ... 303, 325
ConvertBack, метод ... 303
Converter, свойство ... 302
месторасположение ... 328
ConverterParameter, свойство ... 305
месторасположение ... 328
строка, задание ... 325
CornerRadius, свойство ... 166
CornflowerBlue, цвет ... 37
CreateOptions, свойство ... 401

cspline ... 675
CubicBezier, приложение ... 385–90
CumulativeManipulation, свойство ... 165
CurrentOrientation, свойство ... 48
Curve, класс ... 673
  интерполяция кубическим сплайном Эрмита ... 675
  метод Evaluate ... 675
  свойство Keys ... 673
Curve, объекты
  координаты X и Y ... 674
Curve, экземпляры ... 673
CurveKey, класс ... 673
  значение Value ... 673
  свойство Position ... 673
CurveKey, объекты ... 673
CurveKeyCollection, коллекция ... 673
CurveTangent.Smooth ... 675
CustomButtonTemplate, приложение ... 519–22

D

DashOffsetAnimation, проект ... 465–66
Data, свойство ... 364
DataContext, свойство
  задание ... 316–21
  задание в качестве значения свойства Source ... 317
  задание для ссылки на свойства ... 320
  задание объекта Binding в качестве значения ... 317
  передача по дереву визуальных элементов ... 316
  с привязками ElementName ... 318
DataTemplate
  для элементов управления списками ... 552
  назначение ... 505
DataTemplate, свойство 497, см. также шаблоны
  ItemsPanelTemplate ... 589–94
  гистограммы ... 589–94
  для элементов управления списками ... 557
  задание элемента управления в качестве значения ... 598
  коллекция Resources, определение ... 500
  производительность ... 553
  свойство ContentTemplate объекта Button, задание ... 498
  совместное использование ... 501
DatatTemplate, свойство
  для формирования визуального представления элементов управления списками ... 547
DateTime, объекты ... 312
DateTimeFormatInfo, класс ... 320
DayOfWeek, перечисление ... 320
DayOfWeek, свойство ... 320
Deactivated, событие ... 122
DecimalBitToBrushConverter, класс ... 325–26
decimalSeparator, поле ... 247
Default, опция обновления ... 338–40
DelayCreation ... 401
delegate, ключевое слово ... 87
Delta, свойство ... 58
DeltaManipulation, свойство ... 165
DependencyObject, аргумент
  определение родителя ... 295
DependencyObject, класс ... 27, 86, 204, 267
  GetAnimationBaseValue ... 448
  иерархия классов ... 447
  классы, унаследованные от ... 266

метод GetValue 268
метод SetValue 268, 293
свойства-зависимости, хранение 273
DependencyProperty, класс 267
DependencyProperty, поле
инициализация 268
описание 269
существование 274
DependencyProperty, тип 267
создание 267
DependencyProperty.Register, метод 267, 294
DependencyProperty.RegisterAttached, метод294
DependencyPropertyChangedEventArgs, свойство 271
свойства OldValue и NewValue 271
Deserialize, метод 220, 572
DesignerHeight, атрибут 26
DesignerWidth, атрибут 26
DesiredSize, свойство 196, 291, 296
destination, аргумент 659
DirectlyOver, свойство 59–61
перемещение мыши, приостановка 61
DiscreteDoubleKeyFrame, ключевые кадры...443, 459, 469
DiscreteObjectKeyFrame, класс 443
Dispatcher, класс 86
DispatcherTimer, класс 49–51, 160, 439, 572
DisplayName, свойство 564
DisplayTime, метод 247
Distance, метод 641
DistanceSquared, метод 641
DLL, создание 275
double, анимации 442
Double, структуры 146
double, тип 42
DoubleAnimation, класс 443, 447
свойство Easing 443
свойство EasingFunction 483
DoubleAnimationUsingKeyFrames, класс 443, 447
SplineDoubleKeyFrame 480
для анимаций типа double 459
DoubleKeyFrame 459
DownloadProgress, события 74
DragAndScale, проект 164–65
DragCompleted, события 538
DragDelta, события 538
DragStarted, события 538
Draw, метод 36–37, 49, 91
аргумент Color 657–58
объекты SpriteInfo, перебор 662
перегрузка 663
растровые изображения, формирование визуального представления 69
с переменным цветом текста 57
формируемый по умолчанию код 37
Draw, метод (SpriteBatch)
Rectangle 658
аргумент Color 657–58
аргумент position 657
аргумент Texture2D 657
объекты Rectangle, допускающие пустое значение 658
объекты SpriteBatch 659
разновидности 657–59
угол поворота 658
Draw, перегрузка 638

объекты RenderTarget2D и Texture2D..........677
DrawString, метод.. 37
координаты..636
нулевое масштабирование..............................649
разновидности 648–49
Duration, свойство...447
dxdt и dydt, переменные 672

E

EaseIn 483
EaseInOut...483
EaseOut...483
Easing, свойство...443
EasingColorKeyFrame, класс..................................483
EasingDoubleKeyFrame, класс......................459, 483
EasingDoubleKeyFrame, ключевые кадры.......443
EasingFunction, свойство................................ 485–86
типа EasingFunctionBase....................................483
EasingFunctionBase, класс.......................................483
свойство EasingMode..483
EasingFunctionBase, производные.............. 483–87
значения свойств по умолчанию 487
EasingFunctionDialog, страница 485
свойство EasingFunction 486
EasingPointKeyFrame, класс...................................483
ElapsedGameTime, свойство 637
ElementName, свойство..298
Ellipse, класс...154–55, 344
векторная графика 344
конечная высота и ширина 200–203
метод MeasureOverride 196
размер 193
свойства Fill и Stroke...155
свойства Height и Width.................................... 155
ellipseAngle 672
EllipseChain, проект... 200–203
EllipseGeometry, класс 364
координаты, задание... 484
свойства RadiusX и RadiusY.............................. 366
трансформации 371
EllipseMesh, приложение................................. 202–5
ElPasoHighSchool, проект 566–80
EmbossedText, проект.......................................156–58
Empty, свойство
типа Geometry 364
EnableCacheVisualization, флаг 458
EnableFrameRateCounter, флаг 456–58
EnableRedrawRegions, флаг 457
Enter, клавиша, функциональность 259
EvenOdd, правила заливки 359–60
EventTrigger, теги 462
ExpandingCircles, проект 462–64
Explicit, опция обновления 338–40
Expression Blend.. 439, 497
Extensible Application Markup Language (XAML) ....см. также Расширяемый язык разметки приложений
Extensions, класс 402

F

FadableToggleButton, класс 525
тема по умолчанию 528
FadableToggleDemo, приложение 528–29
FadeInOnLoaded, проект.................................461–63
Figures, свойство, типа PathFigureCollection . 374
Fill, значение свойства Stretch 361
Fill, свойство.. 155

задание пустого Binding 556
типа Brush 345
FillBehavior, свойство
значение HoldEnd..449
значение Stop 459
FillBehaviour, свойство
параметр HoldEnd 495
FillRule, свойство 359–60, 374
finalChildSize 291
finalSize, аргумент 197
Fixed, присоединенное свойство.......................457
Flat, наконечники 353–54, 356
FlipHorizontally 648
FlipToggleButton, класс 523–25
FlipToggleDemo, проект................................. 523–25
FlipVertically 648
FlyAwayHello, приложение............................ 659–63
FM-радио 19
FocusStates, имя группы 515
FontFamilies, приложение............................. 169–70
FontFamily, класс
метод ToString 549
FontFamily, настройки 26
FontFamily, объекты
заполнение элементов управления 548
заполнение элементов управления ими ...550
FontFamily, свойство 169
наследование...134
привязка к элементам ListBox................. 559–60
FontName, теги.. 33
в приложениях на XNA 33
присваивание имен 33
теги Size 33
FontSize, свойство 31, 283
наследование 134
пункты и пикселы.. 30–32
размещение параметров..........................241–42
редактирование 30–32
FontStretch, свойство 168
наследование 134
FontStyle, свойство...168
наследование 134
FontWeight, свойство 168
наследование 134
Foreground, атрибут 26, 30
Foreground, свойство 62
Binding...332
задание в XAML..138
задание для состояния Pressed...............518–19
задание программно 282
наследование 134, 277
наследование через дерево визуальных элементов 514
привязки шаблонов 533–34
стили темы 282
Forever, значение 448
FrameBasedVsTimeBased, проект.................439–42
FrameworkDispatcher.Update, метод 618–20
FrameworkElement, класс 27, 44
иерархия классов 227
метод SetBinding 274, 300, 339
наследование 228
свойство DataContext 316
свойство Resources..145
свойство Style 253

система компоновки ... 194
цели привязки ... 301
FrameworkElement, свойство ... 42
From, свойство ... 448–49
FromArgb, метод ... 62

G

Game, класс ... 35
метд OnDeactivated ... 125
метод Draw ... 657
метод OnActivated ... 125
перегрузки метода Draw ... 657
Game1, класс ... 34, 636
настройки заднего буфера ... 46–47
перегрузка Draw ... 663
поля ... 34
создание экземпляров ... 34–35
Game1, конструкторы
выполнение ... 34–35
Game1.cs, файлы ... 34–35
GamePad, класс ... 37
GameTime, аргумент ... 637
свойство ElapsedGameTime ... 637
свойство TotalGameTime ... 637, 662
GameWindow, класс
свойство ClientBounds ... 49
свойство CurrentOrientation ... 48
событие OrientationChanged ... 48
GenerateImageTile ... 420
generic.xaml, файл ... 525, 526–28
ControlTemplate по умолчанию ... 544
Style по умолчанию ... 536, 544
Genres, свойство
типа GenreCollection ... 620
GeoCoordinateWatcher, класс ... 92–93, 95
Geometry, класс ... 364
иерархия классов ... 364
открытые свойства ... 364
свойство Bounds ... 365
свойство Transform ... 370
трансформирование ... 370
GeometryGroup, класс ... 364, 373
GeoPositionAccuracy, перечисление ... 92
GestureSample, объекты ... 58
GestureType, перечисление ... 58
GestureType, свойство ... 58
get, методы доступа ... 268
GetAlbumArt, метод ... 620, 622
GetBindingExpression, метод ... 339
GetCapabilities, метод ... 54
GetCoordinate, объекты ... 93
расстояние между ... 93
GetPictureFromToken, метод ... 438
GetPrimaryTouchPoint(refElement), свойство ... 60
GetRandomPicture, метод ... 82
GetResourceStream, метод ... 75
GetTemplateChild ... 530, 543
GetThumbnail, метод ... 620, 622
GetTouchPoints(refElement), свойство ... 60
GetValue, метод ... 268, 273, 668, 670, 675–76
аргумент Boolean ... 670
вызов ... 269
GIF, формат ... 67
Global Positioning System (GPS) ... см. также Глобальная система позиционирования
GlobalOffsetX, свойство ... 491

GlobalOffsetY, свойство 491
GlobalOffsetZ, свойство 491
GoBack, метод 104, 107
GpaBarChart, проект 592–94
GPS, Global Positioning System .... см. также Глобальная система позиционирования
GradientAccent, проект 152–53
GradientAnimation, проект 468
GradientStop, объекты 139
GradientStopCollection, свойство 139
GradientStops, свойство 139
graphics, поля 35
  инициализация 35
GraphicsDevice, класс
  свойство Viewport 36
Grid, элементы 26–27, 345–47
  TextBlock, размещение 504
  векторная графика 347
  вложение 207, 230
  дочерние элементы 207
  наследование новых классов 195
  одна ячейка 178–79
  организация содержимого 40
  ориентация 179, 209
  отсчет строк и столбцов 208
  панель/сетка для содержимого и область содержимого 28
  перегрузки методов 195
  перекрытие элементов 179
  потомки 178
  размер элементов 179
  с одной ячейкой 195
  свойства Grid.Row и Grid.Column 208
  свойства Grid.RowSpan и Grid.ColumnSpan 208
  свойства HorizontalAlignment и VerticalAlignment 209
  свойство ColumnDefinitions 207
  свойство RowDefinitions 207
  смещение 393
  строки и столбцы 207
  строки и столбцы, высота и ширина 208
  строящиеся по координатам элементы 347
  функциональность компоновки 207
  характеристики 207–9
  элементы управления, задание размеров 285
GridWithFourElements, проект 178
GrowingPolygons, проект 362–64

H

HasArt, свойство
  типа bool 620
Header, свойство 605
  привязка 626
HeaderTemplate, свойство 614
HorizontalAlignment, свойство 28, 39–41
  дочерние элементы Canvas 200
  значение Stretch 66, 513
  настройки по умолчанию 41
HorizontalContentAlignment, свойство 514
  TemplateBinding 514
HorizontalItemsControl, проект 587–89
HorizontalScrollBarVisibility, свойство 187, 189
HorzBar.png 659
Host, свойство 457
HybridClock, проект 160–62
HyperlinkButton, класс 243

I

IconUri, поле 212
IConvertible, метод 332
Id, свойство 55, 59
if, выражения, изменение значений свойств 314
IFormattable, интерфейс 337
IgnoreImageCache 401
Image, элемент 69–70, 73–75
  свойство Source 70, 73
Image, элементы 170–73
  источник вызовов Render 406
  метод MeasureOverride 196
  подразумеваемый размер 193
  растровые изображения, размещение на них 410–16
ImageBrush, класс 175, 344
ImageExperiment, проект 170–73
ImageFailed, события 71, 74
ImageOpened, события 71, 74
ImageSource, класс 73–75, 400
infinity, вычисление размеров ячейки 291
Initialize, метод 35
Initialize, перегрузка 674
InitializeComponent, метод 24, 43
InkPresenter, класс 218, 221
Inlines, свойство 168–69
INotifyCollectionChanged, интерфейс 561, 570
INotifyPropertyChanged, интерфейс 310–11, 313
INotifyPropertyChanged, события 566, 580
InputScope, свойство 256
  значения 257
  параметр TelephoneNumber 256
InputScope, свойство-элемент 257
InputScopeNameValue, перечисление 257
Intellisense 257
Internet Explorer 19
InterpolationFactor, свойство 660, 662
Invalidate, метод 401
InvalidateArrange, метод 289, 295–96
InvalidateMeasure, метод 289, 295, 596
Invoke, метод 86
isAnimationGoing, поле 661
ISaveFileDialogCompleted, интерфейс 426–27
IsChecked, свойство
  типа bool 244
  типа Nullable <bool> 244
IsChecked, свойство-зависимость
  доступ обработчика событий 287
  переключение 286
  типа bool 285
IsClosed, свойство 374–75
IsConnected, свойство 54
IsDragging, метод 538
IsEnabled, свойство 228, 515
  значение false 288
IsFilled, свойство 375
isForward, значение, переключение 485
isForward, поле 484
isGoingUp 636–38
IsItemItsOwnContainerOverride, метод 557
IsItemsHost, свойство 177
IsLargeArc, свойство 376–78
IsolatedStorageFile, объекты 126
IsolatedStorageFile.GetUserStoreForApplication, метод 126
IsolatedStorageSettings, класс 121–24

IsolatedStorageSettings, объекты 123
IsOpen, свойство
задание 602
обработчик события изменения значения свойства 602
IsPressed, свойство 515
IsSelected, свойство......................553
IsTabStop, свойство......................228
IsVisible, свойство...................... 45
ItemCollection, тип 549
ItemContainerGenerator, свойство 553
ItemContainerStyle, свойство......................553
Items, свойство
типа ItemCollection......................549
ItemsControl, класс 547
ItemsPresenter 587
без ScrollViewer 589
выравнивание 590
иерархия классов......................547
метод IsItemItsOwnContainerOverride......................557
наследование......................275
производные......................605
прокрутка......................581
свойство DataTemplate......................598
свойство ItemsPanel......................587
ItemsControl, элементы управления
DataTemplate......................576
горизонтальное отображение 587–89
дерево визуальных элементов, отображение 549–52
логика выбора......................547
отображение идентичных имен классов...576
прокрутка......................549
ItemsControls, класс
ControlTemplate...................... 552
ItemsControlsFromXaml, проект......................554–57
ItemsControlsVisualTrees, проект 549–52
ItemsPanel, свойство 587
ItemsPanelTemplate
DataTemplate......................589–94
UniformStack...................... 589
для панелей элементов управления списками 552
ItemsPanelTemplate, свойство
для элементов управления списками 557
ItemsPresenter, StackPanel...................... 552
ItemsPresenter, класс 552, 587
VirtualizingStackPanel 553
ItemsSource, свойство
привязка......................561–66, 626
ItemTemplate...................... 626
IValueConverter, тип...................... 303, 585

J

JeuDeTaquin, проект......................416–23
JiggleButton, пользовательские элементы управления 461
JiggleButtonTryout, проект 458–61
Jot, приложение......................217–21
ApplicationBar 223–27
закрытые поля 221
захоронение...................... 218
изолированное хранилище......................218–19
касание 221–23
конструктор класса...................... 220
метод Load 219–20
область содержимого 221

обработчики событий Click для кнопок .. 223–24
окна сообщений 225
ориентация 227
отображение пунктов меню 224
параметры приложения 218
цвета фона и переднего плана 219
JPEG, формат 67
загрузка и сохранение 402
jpegBits, массив, сохранение 434
JPEG-формат
сохранение изображений 422

K

keyframe, объекты
свойство KeyTime 459–60
свойство Value 459
KeyFrames, свойство 443
Keys, свойство 673
KeySpline, свойство 472
KeyTime, свойство 459, 472

L

Label, свойство 277
lapSpeed, переменная 645
lastDateTime, поле 52
Launching, событие 122
LayoutRoot, элемент, размер по вертикали 45
LayoutTransform, свойство 167
LayoutUpdated, событие 42
Left, свойство 202, 204
перемещение элементов 206–7
Line, класс .... 344, см. также линии
в панелях Canvas 346
координаты 346
свойства типа double 345–46
Line, элементы
в Grid 347
перекрытие 347–48
порядок 348
разметка 348
LinearDoubleKeyFrame, ключевые кадры 443, 459–60
LinearGradientBrush, класс 139
анимация 466–67
свойство GradientStops 139
LineBreak, объекты 168
LineGeometry, класс 364
свойства StartPoint и EndPoint 365
свойства-зависимости 365
LineSegment, класс 375
ListBox, элементы управления ....547, см. также элементы управления списками
ControlTemplate 553
DataTemplate 576
ItemsSource 575
выбор 557–61
высота и ширина, задание явно 561
дерево визуальных элементов 552–54
конейнеры 553
объекты SolidColorBrush 558
отображение выбранных элементов ... 577–79
отображение идентичных имен классов .. 576
отображение, форматирование 555
прокрутка 549
свойство DisplayMemberPath 555
свойство ItemContainerStyle 553
свойство SelectedItem 577

создание экземпляров 548
ListBoxItem, класс 553
ControlTemplate 553
свойство IsSelected 553
ListBoxSelection, приложение 557–61
LoadBitmap, метод.... 435–36
повторная активация после захоронения 438
LoadContent, метод.... 35, 48, 52, 56, 72, 637, 643, 652
turnPoints, вычисление....664
переменные, задание 645
траектория, определение 672
Loaded, событие 42
Loaded, события
триггер анимации.... 461–68
LoadedPivotItem, события....613
LoadingPivotItem, события 613
LoadSettings, метод 123
LocalOffsetX, свойство 491
LocalOffsetY, свойство 491
LocalOffsetZ, свойство 491
lock, выражения 90

M

MainPage, класс 25, 102
описания частичных классов....24–26
свойство SupportedOrientations.... 40
MainPage, объекты.... 24
MainPage.g.cs, файлы 26
проблемы 26
MainPage.xaml, файлы 25–26
дерево визуальных элементов.... 27
редактирование.... 28
MainPage.xaml.cs, файлы 23
структура....24
Manipulation, события
ScrollViewer 600
обработка....164–65, 387
переопределения 286
формирование.... 367
элемент управления Thumb.... 538
ManipulationCompleted, события 59, 62
ManipulationContainer.... 367
ManipulationDelta, события 59, 62, 165
ManipulationDeltaEventArgs, аргументы.... 165
ManipulationOrigin 367
ManipulationStarted, события 59, 62
ManipulationStartedEventArgs, аргумент 65
MapItemsControl, класс.... 547
margin, значение 665
Margin, свойство.... 41, 61
TemplateBinding.... 513
для столбцов 185
свойство FrameworkElement....42
mask, переменная 436
Math.Atan2, метод....411, 671–72, 676
Matrix3DProjection, объекты 488
MatrixTransform, класс.... 156
maxChildSize 291
MaximumRotation, свойство.... 660
MaximumTouchCount, свойство 55
MaxLength, свойство.... 257
maxScale 650
MAXSPEED, константа.... 652
maxTotalHeight 597
Measure, метод 194

MeasureOverride, метод 194–95, 290, 596
задачи 195–96
перегрузка 195
MeasureString, метод...36, 53
масштабирование по вертикали....................651
MediaElement, элементы 173, 215–16
свойство AutoPlay..216
MediaLibrary, класс........................... 79, 82, 618, 629
метод SavePicture 402, 424
свойства... 620–21
свойство SavedPictures 424
MediaPlayer, класс 629
метод Play 629
метод Stop 632
методы MovePrevious и MoveNext 629
методы Pause, Resume и Stop..........................629
свойство PlayPosition...629
свойство Queue..629
свойство State...629
события MediaStateChanged и ActiveSongChanged..........................629–31, 633
MediaQueue, объекты..629
MediaState, перечисление.....................................629
MediaStateChanged, события 629
MessageBox.Show 225
Microsoft Research Maps, сервис 95
Microsoft Silverlight...............................See Silverlight
Microsoft Visual Studio 2010 Express for Windows Phone...............см. Visual Studio 2010 Express for Windows Phone
Microsoft Windows Phone 7см. Windows Phone 7
Microsoft.Devices.Sensors, библиотека 85
Microsoft.Devices.Sensors, пространство имен85, 89
Microsoft.Phone, пространство имен................ 402
Microsoft.Phone.Controls, библиотека
элементы управления Pivot и Panorama.... 605
Microsoft.Phone.Controls, пространства имен .25
Microsoft.Phone.Controls, пространство имен 605
Microsoft.Phone.Controls.Primitives, пространство имен... 605
Microsoft.Phone.Shell, пространство имен 45, 114, 210–11
Microsoft.Phone.Tasks, пространство имен 76
Microsoft.Xna.Framework DLL 81
Microsoft.Xna.Framework, библиотека 424
Microsoft.Xna.Framework, пространство имен 673
Microsoft.Xna.Framework.Media, пространство имен 79, 424, 618
MinimumOverlap, свойство.......................... 595, 599
minTotalHeight 597
Miter, соединения 354–55
Mode, свойство 302
по умолчанию... 302
Monochromize, приложение..........................424–29
MoveInGrid, проект..469–70
MoveOnCanvas, проект....................................468–69
MoviePlayer, проект.................................211, 215–17
MultiplyConverter 590–91
MusicByComposer
ссылка Microsoft.Phone.Controls..................... 618
MusicByComposer, проект 615–18
DataTemplate.. 626
ItemsControl 627
воспроизведение альбомов 629–31
дерево визуальных элементов 625–26

запуск ... 625
захоронение ... 627, 631
класс AlbumInfo ... 620–22
класс AlbumPage ... 629–31
класс ComposerInfo ... 622
класс MusicPresenter ... 622–25
класс SongTitleControl ... 632–34
кнопка Play/Pause ... 631
метод FrameworkDispatcher.Update ... 618–20
названия песен и истекшее время воспроизведения, отображение ... 632–34
остановка воспроизведения ... 632–33
сетка для содержимого ... 627
строка запроса ... 627
существование и возобновление ... 632
файл AlbumPage.xaml ... 628
элемент управления Pivot ... 625–26

MusicPresenter, класс ... 622–25
свойство Current ... 624

N

NaiveTextMovement, проект ... 636–38
Name, свойство ... 43
типа string ... 620
Navigate, метод ... 103, 107–8
вызов ... 104
NavigationContext, свойство ... 109
NavigationEventArgs, аргументы ... 111
NavigationService, класс
метод GoBack ... 104, 107
метод Navigate ... 107
NavigationService, свойство ... 103
New Project, диалоговое окно ... 20
new, выражения ... 36
NewSize, свойство ... 44
NonZero, правила заливки ... 360

O

Object, анимации ... 442
Object, класс
его производные ... 210
иерархия классов ... 447
ObjectAnimationUsingKeyFrames, класс ... 443, 447
ObservableCollection, класс ... 570, 573, 575
интерфейс INotifyCollectionChanged ... 561
свойство Count ... 573
OnActivated, метод ... 125, 129
OnAppbarFormatClick, метод ... 250
OnAppbarSetStrokeWidthClick, метод ... 226
OnApplyTemplate, метод ... 543
OnApplyTemplate, перегрузки метода ... 530
OnCameraCaptureTaskCompleted, метод ... 78
OnDeactivated, метод ... 125, 129
OnDownloadStringCompleted, метод обратного вызова ... 572
OneTimeText, приложение ... 254–58
состояние приложения, восстановление ... 258
OnGeoWatcherPositionChanged, метод ... 98
OnManipulationCompleted, события ... 368
перегрузка ... 286
OnManipulationDelta, метод
перегрузка ... 206, 367, 477
OnManipulationStarted, метод
маршрутизация событий ... 66
обработчик ... 65
перегрузка ... 63–65, 102, 206, 367
OnManipulationStarted, событие
перегрузка ... 286, 433, 492

OnNavigatedFrom, метод 111–12, 429, 437
перегрузка 119, 424, 426, 486
OnNavigatedTo, метод 111, 426, 429, 437–38
вызов...109
перегрузка 109, 119, 486
OnOrientationChanged, метод 45–46
переопределение 45
OnPropertyChanged, метод 311, 313
OnTextBlockManipulationStarted, метод, маршрутизация событий 66
OnTextBoxChanged, метод.....................................261
OnTimerTick, метод 50
OnToggleButtonChecked, метод..........................247
OnTouchFrameReported, метод 59
OnValueChanged, метод 278
OnWebClientOpenReadCompleted, метод..72–73
Opacity, свойство 185–87, 214
анимация 455, 464
наследование..187
OpacityMask, свойство 173–75
OpenReadCompleted, события 73
Orientation, перечисление 180–81
Orientation, свойство..............................46, 181, 595
OrientationChanged, события...................45, 48–49
origin, аргумент..648, 665
origin, поле..650
OriginalSource, свойство... 65
OverlapPanel 595
maxTotalHeight и minTotalHeight 597
Orientation, свойство................................... 595–96
компоновка дочерних элементов 595
размер дочернего элемента 595–96
свойство MinimumOverlap 595
override, ключевое слово...................................... 194

P

Padding, свойство 41, 513
мишени касания...61
PageOrientation, перечисление...................46, 181
значения 233
Paint, приложение
прозрачность растровых изображений 664
Panel, класс ....27, 177–78, см. также панели
Canvas..см. Canvas
Grid...см. Grid, элементы
StackPanel .... см. StackPanel, элементы
иерархия классов 177
как значение свойства Content 240
наследование классов............................. 194–200
наследование свойства RenderTransform.. 187
потомки 177
свойство Background 177
свойство Children....................................... 177, 266
свойство IsItemsHost 177
функциональность.......................................194–95
PanningBackgroundLayer, класс 615
PanningLayer, объекты 615
PanningTitleLayer, класс 615
Panorama элементы управления
элементы PanoramaItem, занимаемое пространство 612
Panorama, класс... 547
Panorama, элементы управления 605
коллекция Items 605
навигация 612
окно просмотра 613
привязки 614

свойство Background 612
свойство Header 605
свойство HeaderTemplate 614
свойство SelectedIndex 613
свойство SelectedItem 613
свойство Title 611–13
события SelectionChanged 613
тексты Header 612
шаблон 615
PanoramaDemonstration, приложение 611
PanoramaItem, параметр типа 605
Paragraph, класс 169
parameter, аргумент 325
ParametricTextMovement, проект 645–48
PARGB32, формат пикселов 409
Pass, класс
свойство Data 364
PasswordBox, элементы управления 254
Path Markup Syntax 390–94, 412
для описания Data 465
Path, класс 344–45
EventTrigger, присоединение 464
свойство Data со значением null 414
Path, свойство 298–300
Path, элементы 364–69, 412
ScaleTransform 373, 481
Style 392
градиентные цвета, применение 414–15
случайная кисть Fill 368
создание 412, 414
трансформации 369–70
формирование визуальных представлений в растровых матрицах 415
pathDirection, поле 642–44
PathFigure, класс 374
свойство IsClosed 375
свойство IsFilled 375
свойство Segments 374
PathFigure, объекты 375
PathGeometry, класс 364, 374–75, 412
описание в строке 390
получение из коллекции Resources 412–14
свойство Figures 374
свойство FillRule 374
PathSegment, объекты 374–75
pathVector, поле 642–44
PenLineCap, перечисление 353, 356
PenLineJoin, перечисление 354
PerspectiveRotation, приложение 489–90
Petzold.Phone.Silverlight, проект 275–81
элемент управления TapSlideToggle 283–88
PhoneAccentBrush, ресурс 151
PhoneAccentColor, ресурс 151
PhoneApplicationFrame, класс 102, 105
свойство RootVisual 24
события OrientationChanged 45
PhoneApplicationFrame, объекты 27
создание 24
PhoneApplicationPage, класс 25, 102, 618
класс MainPage 25
описание пространства имен 26
перечисление SupportedPageOrientation 39
свойство Orientation 180
свойство SupportedOrientations 180
события OrientationChanged 45

PhoneApplicationPage, объекты 27
свойство Background 404
PhoneApplicationPage, теги 606
PhoneApplicationService, класс 114
использование в XNA 125
свойство Current 114
свойство State 114
PhoneApplicationService, обработчики событий 220
PhoneApplicationService, объекты 122
данные состояния, хранение 417
PhoneApplicationService, поле 427
PhoneBackgroundBrush, ресурс 285
PhotoChooserTask, класс 417, 426–27
Completed, событие 406
инициация 435
метод Show 406
свойства PixelWidth и PixelHeight 418
PhotoChooserTask, объекты 79, 406
Picture, объекты 424
PictureCollection 424
PictureDecoder.DecodeJpeg, метод 402
Pivot, класс 547
ItemTemplate 626
свойство HeaderTemplate 626
свойство Title 606
Pivot, элементы управления 605
PivotItem 617
коллекция Items 605
навигация 612
окно просмотра 613
привязки 614
приложение MusicByComposer 615–18
свойство Header 605, 612
свойство HeaderTemplate 614
свойство SelectedIndex 613
свойство SelectedItem 613
свойство Title 611
события 613
события SelectionChanged 613
шаблон 614
PivotDemonstration, проект 606–11
PivotHeadersControl, класс 547, 614
PivotItem, элементы управления
Ellipse 608
ListBox 606–8, 615–17
ScrollViewer 609–10
анимации 610–11
заголовки, месторасположения 611
параметр типа 605
прокрутка 607–10
размер области содержимого 607
свойство Header 607
свойство Header, привязка 626
создание 626
PixelHeight, свойство 400, 418
Pixels, массив 408
запись 410
копирование пикселов 419–20
хранение пикселов 435
pixels, поле 436
Pixels, свойство 401, 408–11
индексация 408
PixelWidth, свойство 400, 418
PlaneBackground

свойство 542, 545

PlaneProjection, объекты 488

RotationX, RotationY и RotationZ 488

вращение 488

свойства CenterOfRotationX, CenterOfRotationY и CenterOfRotationZ490–91

свойства GlobalOffsetX, GlobalOffsetY и GlobalOffsetZ 491

свойства LocalOffsetX, LocalOffsetY и LocalOffsetZ 491

свойство Projection 488

pLap 650

PNG, формат 67

для изображений значков 210

поддержка прозрачности 664

Point, анимации 442

Point, объекты

добавление во время выполнения 362–64

замена в коде 363

Point, свойства, как цель анимации 469–72

Point, структура 640

PointAnimation, класс 442, 447

свойство Center, как цель 484

свойство Duration 484

свойство Easing 443

свойство EasingFunction 483

функции сглаживания 483–87

PointAnimationUsingKeyFrames, класс 443, 447

PointCollection, отсоединение от класса 364

PointDragger, класс 386–87

Points, свойство 348

PolyBezierSegment, класс 389

Polygon, класс 344

замкнутые фигуры 359

свойство FillRule 359–60

Polyline, класс .... 344, 348–52, см. также линии

для создания кривых 349–52

кисть Fill 359

координаты, задание 484

окружности, отрисовка 350–51

свойство Points 348

свойство Stroke 350

свойство StrokeThickness 350

точки, множество 348

PolylineInterpolator, класс 666, 668–70

метод GetValue 668, 670

свойство Vertices 667

PolyLineSegment, класс 375

PolyQuadraticBezierSegment, класс 389

PortraitOrLandscape, свойство 40

position, поле 657

вычисление 676

Position, свойство 55, 59–60, 660

вычисление значений 60

PositionChanged, события 93

PositionOffset, свойство 660

Posterizer, приложение 430–38

PresentationFrameworkCollection, класс, наследование 361

Pressed, состояние

обозначение 518–19

Pressed, состояния 541

PressureFactor, свойство 218

Program.cs, файлы 34

Projection, свойство 442, 488–89

PropertyChanged, события 311, 313, 569, 572

PropertyChangedEventArgs, класс 311
PropertyChangedEventHandler, делегаты......311
PropertyMetadata, конструктор......267
  аргумент double......294
  аргументы...... 273–74
PropertyName, свойство......311
PublicClasses, приложение...... 189–92

Q

QuadraticBezier, приложение...... 382–86
QuadraticBezierSegment, класс......389
QuadraticEquations1, проект...... 337–40
QuadraticEquations2, проект...... 340–43
QuadraticEquationSolver, класс...... 333–37
QueryString, свойство......109
  строковый ключ «token»......438
QuickBarChart, приложение...... 291–92
QuickNotes, приложение......217, 258–62
  параметры, сохранение, загрузка и предоставление......259
QuickNotesSettings, класс......258

R

RadialGradientBrush, касс
  Button, задание содержимого......499
RadialGradientBrush, класс...... 141–42
  анимация......481
RadioButton, класс......244
  свойства Tag, текстовые строки......250
RangeBase, класс...... 229–34
  свойства......229
Rect, аргумент......197
Rectangle, класс......155, 344
  векторная графика......344
  свойства RadiusX и RadiusY......155
Rectangle, объекты
  допускающие пустое значение......658
RectangleGeometry, класс......364
  свойства RadiusX и RadiusY......365
  свойство Rect......365
RecursivePageCaptures, проект...... 402–5
RelativeSource, привязки данных......307
Render, метод...... 401, 404
  вызовы......406
RenderingEventArgs, объекты......163
RenderTansformOrigin, свойство......158
RenderTransform, свойство......156, 167, 370–72
  анимации......455
  наследование в дочерних элементах панели......187
RenderTransformOrigin......369
RepeatBehavior, свойство......447–48
  значение Forever......448
RepeatButton, элементы управления243, 530–33
  новый шаблон......533–34
  свойство Template......533
ResourceDictionary.MergedDictionary, свойство-элемент......523
Resources, коллекции......144–45
Resources, коллекция
  CollectionViewSource......583
  ControlTemplate......522
  Storyboard, определение......478
  конвертер привязок......558
  описания анимаций и раскадровок......450
  открытые свойства, задание......326
  свойство Style......149–50, 522
  сетки для содержимого......188

Resources, свойство....145
  имена ключей....148
  индексация....148
Resources. коллекция
  определение DataTemplate....500
ResourceSharing, проект.... 147–48
RgbColorScoller, класс.... 280–81
  XAML-файл.... 279–80
RGB-цвета
  битовое разрешение, уменьшение.... 430–38
  предварительно умноженные значения цвета....410
  преобразование в черно-белое изображение.... 427–29
RIA, Rich Internet Applications.... см. также Насыщенные Интернет-приложения
Rich Internet Applications (RIA).... см. также Насыщенные Интернет-приложения
RootVisual, свойство.... 44
RotatedSpiral, проект.... 452–54
RotateTransform, класс....156, 158
RotateTransforms, объекты, сброс....394
RotatingText, проект.... 163–64
rotation, аргумент....648, 658, 672
Rotation, свойство....660
RotationAngle, свойство....380
RotationX, свойство....488
  анимация....489
RotationY, свойство....488
  анимация....489
RotationZ, свойство....488
  анимация....489
Round, наконечники....353, 356
Round, соединения....354
RoutedEventArgs, аргумент....65
RoutedPropertyChangedEventArgs.... 278
RoutedPropertyChangedEventHandler.... 278
Run, класс.... 321
  свойство Text.... 266
Run, объекты.... 168

S

Saved Pictures, альбом....82, 424, см. также библиотека изображений
SavedPictures, свойство.... 424
SaveFileDialog, класс....428–29
SaveFileDialog, страница.... 424
  метод SetTitle....425–26
  область содержимого.... 425
SaveFileDialog.xaml, страница.... 428
SaveFileDialogCompleted, метод....426, 428–29
SaveJpeg, метод....402, 424, 429
SavePicture, метод.... 402, 424
SaveSettings, метод.... 123
scale, поле.... 650
scaledRadius....351–52
ScaleTextToViewport, проект....649–51
ScaleTransform, класс....156, 372–73
  элементы Path.... 481
ScaleTransfotm, класс
  свойства CenterX и CenterY.... 158
ScaleX и ScaleY, свойства.... 158
ScrollBarVisibility, перечисление.... 187
ScrollContentPresenter.... 553
ScrollViewer, элементы управления....187–92
  автоматический перенос текста.... 192
  вертикальный StackPanel.... 200
  горизонтальная прокрутка....587–89

значение Padding ... 188
значения свойства Content ... 241
события Manipulation ... 600
Search, кнопки ... 17
SecondPage, класс ... 104
Segments, свойство ... 374
SelectedIndex, свойство ... 557, 613
задание значения 0 ... 559
SelectedItem, свойство ... 557, 577, 613
null ... 560
как цель привязки ... 557
типа ColorPresenter ... 565
SelectedText, свойство ... 261
SelectionChanged, события ... 261, 557, 613
SelectionStart, свойство ... 261
Selector, класс ... 547
свойство SelectedIndex ... 557
свойство SelectedItem ... 557
Self, свойство ... 307
sender, аргумент ... 43, 63
set, методы доступа ... 268
защищенные ... 314
Set, статический метод ... 445
SetBinding, метод ... 274, 300, 339
SetLeft, метод ... 293
SetSource, метод ... 73–75, 400
Setter, объекты, наследование ... 522
Setter, описания ... 149
SetTextBlockText, метод ... 87
Settings, класс ... 125–26, 457
Settings, страница ... 29
SetTitle, метод ... 425–26, 429
SetTop, метод ... 293
SetValue, метод ... 204, 268, 273, 293
вызов ... 204–5, 269
Sex, свойство
привязка ... 585
SexToBrushConverter ... 584–86
Shape, класс
задаваемые свойства ... 345
иерархия классов ... 344–45
свойство Stretch ... 361
Shapes, библиотека ... 344–45
Silverlight
Visual State Manager ... 461
в сервисах облака ... 16
векторная графика...344, см. также векторная графика
визуальные объекты ... 26
деревья визуальных элементов ... 27
динамическая компоновка ... 39
и XAML ... 131–32
игры ... 16
класс WebClient ... 74
навигация ... 102–8
наследование свойств ... 134
обработка касания, простого ... 59–62
обработка касания, сложного ... 62–64
обработка маршрутизированных событий ... 65–66
поддержка анимации ... 439
поддержка в Windows Phone 7 ... 15
популярность ... 16
приложения и утилиты ... 15
проект “Здравствуй, мир” ... 20

разработка программ для Windows Phone 7 ...15–17

рамки ... 27

сенсорный ввод... 54

слой визуальной компоновки... 51

стили...144

структура Color... 62

файлы выделенного кода ... 23

форматы растровых изображений... 67

центрирование элементов... 41

элемент Image ...69–70

элементы ... 27

элементы, организация... 40

Silverlight for Windows Phone

поддержка привязок...307

расширение разметки Binding ...299

Silverlight for Windows Phone Toolkit...177

SilverlightAccelerometer, программа...85–88

SilverlightAccessLibrary, программы ...81–82

SilverlightBetterTombstoning, проект... 120–21

SilverlightFlawedTombstoning, проект... 118–20

SilverlightHelloPhone, проект...20–23

SilverlightHelloPhone, пространство имен... 26

SilverlightHost, объект...457

SilverlightInsertData, проект ... 112–14

SilverlightIsolatedStorage, программа... 122–24

SilverlightLocalBitmap, проект ...69–70

SilverlightLocationMapper, проект... 96–101

SilverlightOrientationDisplay, проект ...45–46

SilverlightPassData, проект ...108–9

SilverlightRetainData, проект... 114–16

SilverlightShareData, проект ... 110–12

SilverlightSimpleClock, проект ...49–51

SilverlightSimpleNavigation, проект... 102–8

SilverlightTapHello1, программа ...62

SilverlightTapHello2, проект ...64

SilverlightTapHello3, программа ... 65–66

SilverlightTapToDownload1, проект... 73–75

SilverlightTapToLoad, проект ... 75–76

SilverlightTapToShoot, программа ... 77–78

SilverlightTouchHello, проект... 61–62

SilverlightWhatSize, проект... 42–45

SimpleGrid, проект ... 208–9

SingleCellGridDemo, проект...195–98

SIP...19

Size, свойство ...59

Size, структура ... 193

Size, теги...33

SizeChanged, метод ...42

SizeChanged, обработчики событий...43

SizeChanged, события...42, 44–45, 49

Canvas... 203

SkewTransform, класс ...156, 159–60

Slider, класс...229–34

Slider, шаблоны ...530–38

Slider, элемент управления

теги TemplateVisualStateAttribute... 530

Slider, элементы управления ...229–34

HorizontalTemplate и VerticalTemplate .530–32

диапазон значений по умолчанию... 231

изменение значения... 277

новые шаблоны ...530–32

свойства Orientation... 231

свойство Value ... 233

события ValueChanged ... 233

теги TemplatePartAttribute 530
SliderBindings, приложение 298–300
конвертер 303–6
SliderSum, проект 329–30
SliderSumWithColor, проект 332
SmsComposeTask, приложение....258
Soft Input Panel (SIP) 19
вызов.... 254–62
SolidColorBrush, объекты
в ListBox....558
создание 138
Solution Explorer, диалоговое окно Add Reference 281
Song, объекты
привязка 628
свойство Album....621
свойство Artist 621
свойство Duration 621
свойство Name....621
SongCollection, объекты 621
перемещение по 629
получение....629
приостановка, остановка, возобновление воспроизведения 629
Songs, свойство
типа SongCollection.... 620–21
SongTitleControl, класс 632–34
дерево визуальных элементов....632
привязка свойства Song 632–33
события MediaPlayer.ActiveSongChanged ..633
SortDescription, класс 583
SortDescription, объекты....583
множество....584
Source, свойство....70, 583
URL 71
для привязок данных.... 307, 314
SPEED, константа....636–38
speed, переменная.... 652
приращение 653
сброс в 0 653
SpeedRatio....460–61
SpiteInfo, класс
свойство InterpolationFactor.... 662
SplashScreenImage.jpg, файлы....21
SplineColorKeyFrame, класс.... 472
SplineDoubleKeyFrame, класс 459, 472
SplineDoubleKeyFrame, ключевые кадры 443
SplineKeyFrameExperiment, приложение .473–79
SplinePointKeyFrame, класс 472
SpriteBatch, аргумент 663
SpriteBatch, класс 69, 648
метод Draw 657–59
SpriteBatch, объекты 37, 659
spriteBatch, поля....34
SpriteEffects, перечисление 658
члены FlipHorizontally и FlipVertically 648
SpriteFont, класс
метод MeasureString.... 36, 53
SpriteInfo, класс 659
InterpolationFactor 663
метод Draw 663
свойство InterpolationFactor.... 660
свойство MaximumRotation 660
свойство Position 660
свойство PositionOffset 660

свойство Rotation ... 660

SpriteInfo, объекты

перебор ... 662

SpriteInfo, объекты

создание и инициализация ... 661

Square, наконечники ... 353–54, 356

SquaringTheCircle, проект ... 470–72

Srtoke, свойство

типа Brush ... 345

StackPanel, элементы ... 26–27

в ScrollViewer ... 200

в элементе Border ... 183

вертикальный ... 198–200

вложение ... 184–85

горизонтальная ориентация ... 587–89

дочерние элементы, размеры ... 193

дочерние, компоновка ... 179–81

заполнение с применением отражения . 187–92

использование при навигации ... 116–17

используемое пространство ... 182

конкатенация текста ... 182–84

наследование классов ... 198–200

ориентация ... 180–82

позиционирование ... 182

размеры ... 185

родительский контейнер ... 182

свойство Orientation ... 181

таблицы ... 185

элементы ... 179–82

элементы управления ScrollViewer ... 187–92

StackPanelTable, проект ... 184–85

StackPanelWithFourElements, проект ... 179–81

StandardFlatteningTolerance, свойство

типа double ... 364

Start, кнопка ... 17

StartPoint, свойство ... 374

State, свойство ... 55, 114

доступ ... 121

State, словарь

для промежуточных данных ... 114–16

доступ к элементам ... 121

удаление ... 121

хранение ... 119–21

хранение элементов ... 114–16, 121

StaticResource

конвертеры ... 304

StaticResource, объекты ... 146

StaticResource, расширение разметки ... 304

встроенные ресурсы, доступ ... 151

Stop Debugging, опция ... 22

Stopwatch, класс ... 247

переключение секундомера ... 244–53

Stopwatch, объект

сброс ... 248

Stopwatch, объекты ... 247

настройка ... 252

StopWatch, проект ... 244–53

захоронение ... 251–53

область содержимого ... 245

обновления отображения ... 247

описания полей ... 246–47

определение ApplicationBar ... 248

формат истекшего времени ... 245, 249

Storyboard, класс ... 447

атрибуты TargetName 451
атрибуты TargetProperty 451, 469
методы Begin и Stop 452
методы Pause и Resume 452
свойство Children 447
свойство TargetName 469
Storyboard, объекты 445
запуск 478
определение 458
определение в коллекции Resources 478
Stream, аргумент 400
Stream, объекты 73
Stream, тип 72
Stretch, перечисление 41, 361
Stretch, свойство 66, 70, 361
типа Stretch 345
Stretch, элементы 171
String, теги, разграничение элементов 557
StringBuilder, агрумент
перегрузки 53
StringBuilder, аргумент 51–53
метод DrawString 648
поле состояния, обновление 652
StringBuilder, объекты 36
StringFormatConverter, класс 305, 554
создание экземпляра 305
StringToFontFamilyConverter 560
Stroke, объекты 218
добавление 223
хранение 223
Stroke, свойство 155
StrokeCollection 218
StrokeDashArray, свойство 355, 357, 394, 465
типа DoubleCollection 345
StrokeDashCap, свойство 356, 465
типа PenLineCap 345
StrokeDashOffset, свойство 357
типа double 345
цель анимации 466
StrokeEndLineCap, свойство 353
типа PenLineCap 345
StrokeLineJoin, свойство 354
типа PenLineJoin 345
StrokeMiterLimit, свойство
типа double 345
StrokeStartLineCap, свойство 353
типа PenLineCap 345
StrokeThickness, свойство
анимация 464
типа double 345
Student, класс 566–69
привязка данных 569
свойство Sex 584–86
StudentBody, класс 569–70
StudentBodyItemsControl, проект 585–87
StudentBodyListBox, проект 579–80
StudentBodyPresenter, класс 570–72, 579, 592
создание экземпляров 572
StudentCard, объекты 601
свойство IsOpen, задание 602
StudentCardFile, проект 597–604
MinimumOverlap 599
область содержимого 598
файл MainPage.xaml 597

Students, коллекция, индексация 573
students.xml, файл 570
доступ 572
SturdentCardFile, проект
обработчик событий Touch.FrameReported 600
Style, свойство 149–50
BasedOn 150
Setter 231
TargetType 149
атрибут x:Key 253
для элементов управления списками 557
корневого элемента 524
определение по умолчанию 526
свойство TargetType 253
свойство Template, определение 510
StyleInheritance, проект 150–51
StyleSharing, проект 149–50
StylusPoint, объекты 218
SubdivideBitmap, приложение 405–8
SunnyDay, приложение 381–82
SupportedOrientations, свойство ... 39, 40, 48, 180
PortraitOrLandscape 45
SupportedPageOrientation, перечисление 39, 180
SuppressDraw, метод 51, 53, 653
SuspendMousePromotionUntilTouchUp(), свойство 60
SuspendMousePromotionUntilTouchUp, метод 60
suspensionAdjustment, поле 247, 252
SweepDirection, свойство 376
SweepIntoView, приложение 490–93
System.ComponentModel, пространство имен 311, 583
System.Device.Location, пространство имен 93, 96
System.Diagnostics, пространство имен 244
System.Globalization, пространство имен 320
System.IO, пространство имен 96
System.IO.IsolatedStorage, пространство имен 114, 121
System.Net, библиотека 72
System.Reflection, пространство имен 190
System.Text, пространство имен 652
System.Windows, библиотека 242
System.Windows, пространства имен 25
System.Windows.Control.Primitives, пространство имен 539
System.Windows.Controls, пространство имен 228
System.Windows.Controls.Primitives, пространство имен 228, 530
System.Windows.Data, пространство имен ... 300, 583
System.Windows.Ink, пространство имен 221
System.Windows.Input 218
System.Windows.Input, пространство имен ... 257
System.Windows.Interop, пространство имен 457
System.Windows.Markup, пространство имен 394
System.Windows.Media.Animation, пространство имен 439
System.Windows.Media.Imaging, пространство имен 75, 96, 400, 402
System.Windows.Navigation, пространство имен 109
System.Windows.Shapes, пространство имен 154, 344
System.Windows.Threading, пространство имен 50, 86, 160
SystemTray, класс 45, 205

T

t, вычисление 675
TabIndex, свойство 228
TabNavigation, свойство 228
Tag, свойство 250, 433
  декодирование 433
  производные от класса EasingFunctionBase 487
TapForTextBlock, проект 132–34
TapSlideToggle, элементы управления 283–88
TapSlideToggleDemo, приложение 286–88
Target, свойство, задание для анимаций 444–46
TargetProperty, свойство, задание для анимаций 444–46
TargetType, свойство 149, 253
  Button 254
  Control 254
  FrameworkElement 254
  производные стили 254
tCorner, переменная 654
TelephonicConversation, проект 187–89
Template, свойство 505
  Style, определение 510
  как свойство-элемент 506
TemplateBinding 508–9
  для ContentPresenter 512–13
  для свойства Margin 513
TemplatedItemsControl, класс 547
  производные 605
TemplatedParent 509
TemplatePartAttribute, теги 530
TemplateVisualStateAttribute, теги 530
TerraService 96
Text, поле 212
Text, свойство 266
  вставка 257
  доступ 257
  как цель привязки 333
  получение значение 257
  типа string 257
TextBlock элементы
  метод MeasureOverride 196
  подразумеваемый размер 193
  свойство Inlines 183
TextBlock, класс 28
  в коде 132–34
TextBlock, элементы 26–27
  Grid, размещение 504
  атрибут ContentProperty 168
  атрибут Foreground 30
  в ContentControl 586
  вставка 28
  отображение результатов 333
  размер 193
  рамка 165–68, 506–8
  редактирование 28
  свойства Width и Height 42
  свойства для задания шрифтов 168–70
  свойство Margin 41, 166
  свойство Padding 41, 166
  свойство RenderTansformOrigin 158
  свойство Tag 433
  свойство TextDecorations 168
  события Manipulation 65
  ссылка на них 43

центрирование ... 66
TextBox, элементы управления ... 254
Text типа string ... 257
ввод с клавиатуры ... 254–62
обновление ... 339
свойство InputScope ... 256
свойство MaxLength ... 257
свойство SelectedText ... 261
свойство TextWrapping ... 255
событие SelectionChanged ... 261
фокус ввода ... 261–62
TextConcatenation, проект ... 182–84
TextCrawl, приложение ... 653–56
TextDecorations, свойство ... 168
textPosition, переменная
вычисление ... 649, 655
задание ... 649
координата Y ... 636
определение ... 649
определение размера ... 36
пересчет ... 636
приращение ... 642–44
фиксированное значение ... 636
textSize, свойство
хранение ... 56
Texture2D, аргумент ... 657
Texture2D, класс ... 68
метод FromStream ... 73
Texture2D, объекты
загрузка ... 661
Texture2D, тип данных ... 67
TextWrapping, свойство ... 255
TheEasingLife, проект ... 483–87
Themes, каталог ... 525
Themes, папка ... 526–28
Thickness, структура ... 41, 146
this, ключевое слово ... 35
Thumb, элементы управления ... 530, 538–41
Style ... 540
визуальное состояние Pressed ... 541
метод CancelDrag ... 538
метод IsDragging ... 538
новый шаблон ... 533–34
обработчик событий DragDelta ... 539
перемещение по поверхности ... 541–43
позиционирование ... 538–41
свойство Template ... 533
событя DragStarted, DragDelta и DragCompleted ... 538
шаблон по умолчанию ... 538
ThumbBezier, приложение ... 538–41
Tick, события ... 49
TileBrush, класс ... 175
tileImages, массив ... 417
TimeDisplay, проект ... 315–16
Timeline, класс
иерархия классов ... 447
свойство AutoReverse ... 447
свойство BeginTime ... 447
свойство Duration ... 447
свойство RepeatBehavior ... 447–48
TimeSpan, объекты ... 163, 247
для песен ... 629
свойство TotalMinutes ... 441
TimeStamp, свойство ... 60

Title, свойство 606
TitleAndAppbarUpdate, метод 226–27
TitleTemplate, свойство 614
tLap, переменная 645, 650
To, свойство 449
ToggleButton, класс 244
  настройка с помощью шаблонов 520–22
  свойство IsThreeState 244
ToggleButtond, класс
  свойство IsChecked 244
  события 244
ToggleSwitchButton, класс 244
Top, свойство 202, 204
  перемещение элементов 206–7
ToString, метод 336, 549
TotalGameTime, свойство 637, 662, 675
TotalMinutes, свойство 441
Touch Panel, класс
  метод GetCapabilities 54
Touch.FrameReported, события 59, 222, 594
TouchAction, свойство 594
TouchAndDrawCircles, проект 366–69
TouchCanvas, проект 206–7
TouchCollection, коллекции 55
TouchDevice, объекты 59
  свойство DirectlyOver 59
  свойство Id 59
TouchDevice, свойство 59
TouchFrameEventArgs, объект 60
  свойство GetPrimaryTouchPoint(refElement) 60
  свойство GetTouchPoints(refElement) 60
  свойство SuspendMousePromotionUntilTouchUp() 60
TouchLocation, объекты 55
TouchPanel, класс 54
  метод ReadGesture 58
  распознавание жестов 57–59
TouchPanel.GetState, метод 653
TouchPanelCapabilities, объекты 55
TouchPoint, класс 59
  свойство Action 59
  свойство Position 59–60
  свойство Size 59
  свойство TouchDevice 59
TouchToStopRotation, проект 651–53
Transform, класс 156
Transform, свойство 370
  типа Transform 364
TransformExperiment, проект 158
TransformGroup, класс 156
TranslateTransform, класс 155–57
  анимация 467
  корректировки времени 480
  совместное использование 160
TranslateTransform, объекты 407
TranslateTransform, цель анимации 468
TranslateX и TranslateY, свойства 162
Triangle, наконечники 353, 356
Triggers, свойство 462
  размещение 462
TrueType, шрифты 169
TruncationConverter, класс 303–5
TryGetPreviousLocation, метод 55
TryGetValue, метод 121
turnPoints, массив 664

TwelveHourClock, класс 321–24

U

UIElement, класс.. 27

в качестве XAML-ресурса 145

свойство Opacity 173

свойство Projection...................................442, 488

свойство Visibility 185–87

система компоновки 194

трансформации 156

UIElementCollection, класс 361

UIElementCollection, тип 177

UIThreadVsRenderThread, проект 455–57

UI-поток

выполняемые в нем анимации 455–57

Uniform, значение свойства Stretch..................361

UniformGrid...288

свойства Rows и Columns......................... 288–89

ячейки, задание размеров 288–89

UniformStack, класс 291–92

обработчик события изменения значения свойства 289

описание свойства-зависимости 289

UniformToFill, значение свойства Stretch........361

UnloadedPivotItem, события 613

UnloadingPivotItem, события................................613

Update, метод 36–37, 56, 636–38

аргумент GameTime 637

вычисление угла...672

параметрические уравнения 646

расположение текста, вычисление..........48–49

сенсорный ввод, проверка на наличие........ 54

упрощение 667

формируемые по умолчанию код 37

UpdateSource, метод 339

UpdateSourceTrigger, перечисление................ 339

UpdateSourceTrigger, свойство 339

Uri, объекты для навигации 105

URI, относительные 103

URI, свойство 111

UriSource, свойство.. 401

URL, как значение свойства Source изображения 71

UserControl, класс 229

для пользовательских элементов управления 541

наследование классов................................275–83

наследование от 598

свойство Content 241, 275

UserControl, производные

XAML-файл... 275

атрибуты свойств... 276

визуальные элементы 275

корневой элемент... 276

описания свойств и событий.......................... 277

элемент управления TapSlideToggle 283–88

UserControl, теги 396

UserState, аргументы.. 100

using, директивы..24

V

value, аргумент 325

Value, свойство 459, 472

как цель анимации 494

типа byte 278

ValueChanged, события 278, 544

конструктор и обработчик 278–79

определение 278

ValueToBrushConverter ... 592

Vector2, объекты

  полилинии, определение ... 670

Vector2, структуры ... 34, 641

  арифметические операции ... 640–42

  метод Distance ... 641

  метод DistanceSquared ... 641

Vector2.Lerp, метод ... 643, 650, 670

Vector2.One, свойство ... 642

Vector2.UnitX, свойство ... 642

Vector2.UnitY, свойство ... 642

Vector2.Zero, свойство ... 642

Vector3, структуры ... 89

VectorGraphicsDemos, приложение ... 399

VectorTextMovement, приложение ... 642–44

VectorToRaster, приложение ... 412–16

VertBar.png ... 659

VerticalAlignment, свойство ... 28, 39–41

  дочерние элементы Canvas ... 200

  значение Stretch ... 66

  настройки по умолчанию ... 41

VerticalContentAlignment, свойство ... 514

  TemplateBinding ... 514

VerticalScrollBarVisibility, свойство ... 187, 189

VerticalStackPanelDemo, проект ... 198–200

Vertices, коллекция, закрытая ... 670

Vertices, свойство ... 667, 670

Viewport, свойство ... 36

VirtualizingStackPanel, класс ... 553

  производительность ... 553

Visibility, перечисление ... 185

  в WPF ... 187

Visibility, свойство ... 185–87

  компоновка ... 185–87

  привязка к ... 338

Visual State Manager ... 461, 514–22

  местоположение разметки ... 603

  разметка ... 516

  с производными от UserControl ... 602

Visual Studio

  Intellisense ... 257

  библиотека динамической компоновки, создание ... 275

  отладка ... 619

  отладка приложений для работы с камерой ... 77

  эмуляторы, подключение ... 21

Visual Studio 2010 Express for Windows Phone 20, см. также Visual Studio

VisualState, теги ... 516

  анимации ... 516–18

  для состояния Normal ... 516–18

VisualStateGroups, теги ... 516

VisualStateManager.GoToState, метод ... 515, 602

VisualTreeHelper, класс ... 503

  метод GetParent ... 295

W

WebClient, класс ... 71–72, 572

  в Silverlight ... 74

  свойство AllowReadStreamBuffering ... 73

wide-VGA (WVGA), размеры экранов ... 18

Wi-Fi ... 19

Windows Azure ... 16

Windows Live ... 16

Windows Phone 7 ... 15

  готовность к работе с облаком ... 16

описание оборудования 17–19
опция Device 21
опция Emulator 21
ориентация, реакция не ее изменение . 39–40
поддержка Silverlight 15
поддержка XNA 15
разработка программ для 15–17

Windows Phone 7, устройства .... см. также устройства; также телефоны
аппаратные возможности 19
развертывание програм 21
размеры экранов 17
экраны, поддерживающие мультисенсорный ввод 54–66

Windows Phone Application, проекты
имя проекта 21
конструктор 20
папка Properties 20
файлы WMAppManifest.xml 20

Windows Phone Application, шаблон 20
Windows Phone Developer Registration, приложение 23
Windows Phone Marketplace 16
Windows Phone Panorama Application, проекты 606
Windows Phone Pivot Application, проекты 606
Windows Phone Portrait Page, страницы 103
Windows Phone User Control 276
добавление нового 523
Windows Presentation Foundation .... см. WPF
тег App 620
WMAppManifest.xml, файлы 20
WorldMap, проект 545–46
WPDTPTConnect32.exe/WPDTPTConnect64.exe, программа 77
WPDTPTConnect32/WPDTPTConnect64, приложение 619

WPF
ImageSource 400
UniformGrid 288
анимации 461
имена панелей 194
мозаичные шаблоны 175
перечисление Visibility 187
привязки RelativeSource 307
свойство LayoutTransform 167
трансформации 156
триггеры 461

WrapPanel 177
WriteableBitmap, класс 344, 400–402
UIElement 402–8
конструкторы 401
метод Invalidate 401
метод Render 401, 404
метод SaveJpeg 424
размеры 405
свойство Pixels 401, 408–11
WriteableBitmap, метод 402
WVGA, размеры экранов 18

X

x
Class, атрибут
в файле App.xaml 26
в файле MainPage.xaml 24
x, описания пространств имен 24
X, свойство

анимация ... 459
x:Key, атрибут ... 523
x:Key, атрибут ... 146
x:Name, атрибут ... 43, 148, 163
для ApplicationBarIconButton ... 216
XAML ... 23, 131–32
анимации ... 439, 450–52
градиентные кисти, создание ... 138–40
для компоновки визуальных элементов ... 131
кисти ... 138–41
коллекции ресурсов ... 144–45
конвертирование в объекты ... 394
логическое отрицание ... 323
множество точек ... 348
наследование свойств ... 134–35
наследование стилей ... 150–51
недействительный, обработка ... 398
ограничения ... 133
повторяющаяся разметка ... 144
привязки данных ... 298
привязки передачи и приема ... 328–33
присоединенные свойства, задание ... 451
разделы ресурсов ... 145
свойства содержимого ... 142–44
свойство Foreground, задание ... 138
синтаксис свойство-элемент ... 135–36
содержимое элемента управления списками, определение ... 554–57
создание экземпляров ... 131
стили ... 149–50
текст, отображение ... 323
цвета ... 136–38
шаблоны ... 132
эксперименты ... 132, 134–35
XAML, Extensible Application Markup Language ... см. также Расширяемый язык разметки приложений
XAMLClickAndSpin, приложение ... 450–52
XamlCruncher ... 394–99
XamlCruncherTextBox, класс ... 397
XamlExperiment, проект ... 134–35
XamlPad ... 395
XamlParseException, исключения ... 265, 301
XamlReader.Load, метод ... 132, 394
XamlReader.Save, метод ... 394
XamlResult, события ... 396
XAML-ресурсы ... 144–45
бизнес-объекты INotifyPropertyChanged ... 311
встроенные, доступ ... 151
доступ ... 146, 148
идентификаторы x:Name ... 148
имена ключей ... 146
использование из кода ... 148
классы ... 314
определение ... 146
совместное использование ... 145
типы double ... 145–47
хранение ... 145
XAML-файлы
навигация ... 105
учитывающие размер экрана ... 17
XAP-файлы ... 32
Xbox Live ... 16
xmlns, описания пространств имен ... 24
XmlSerializer, класс ... 570
XNA

двухмерная система координат 639
для приложений.. 16
захоронение и параметры 124–29
обработка простого касания 54–57
ориентация 46–49
отрисовка.. 51
поддержка в Windows Phone 7 15
популярность 16
приложение "здравствуй, телефон" 38
прозрачность изображений............................664
разработка программ для Windows Phone 7 15–17
распознавание жестов 57–59
сенсорные панели 54
сенсорный ввод.. 54
структура Point 640
структура Vector2 640
тип вектор... 83
XNA Game Studio, процесс встраивания шрифтов 32
XNA, создание текстуры 68–69
XnaAccelerometer, проект 88–92
XnaLocalBitmap, программа 68–69
XnaLocation, проект 93–95
XnaOrientableHelloPhone, проект 48
XnaTapHello, проект 57–59
XnaTapToBrowse, программа 79–80
XnaTombstoning, приложение 124–29
XnaTouchHello, проект 56–57
XnaWebBitmap, проект....................................... 72
xValues, массив...675
XYSlider, класс
свойство PlaneBackground...............................542
XYSlider, элементы управления 541–43

Y

yValues, массив 675

Z

ZIndex, свойство.......................................205, 347–48
Zune, настольное ПО................................18, 23, 615

А

автоматический перенос текста
и горизонтальная прокрутка 190
и параметры ScrollViewer 192
Адамс, Джон 617
Адес, Томас... 617
акселерометр 19, 83–88
проект XnaAccelerometer............................ 88–92
система координат 83
активация приложений 118
алгоритмы сжатия «с потерями» 67
альбомный режим..18
альбомный режим отображения
ограничение режимов отображения телефона 48
альбомы, воспроизведение............................... 618
анимации
XAML-based 450–52
анимации по ключевым кадрам 458–61
в UI-потоке 455–57
в тегах VisualState 516–18
в элементах управления PivotItem.........610–11
возвращение к исходному значению......... 459
время, корректировка 480
выполнение... 661
выполняющиеся непрерывно 462
выравнивание фигур 481

завершение 484
загрузка процессора 441
задание строкой 459
запуск 444–46, 458, 478
инициация 660
коллекции 447
моделирование свободного падения 479
непрерывное выполнение 464
несглаженность, как избежать 459–60, 466
обратная перемотка 447
объекты Storyboard 445
определение 445
отложенный запуск 447
перезапуск 448
повтор 447–48
посредством CompositionTarget.Rendering 452–58
приоритетность свойств 493–96
присоединенные свойства 468–72
присоединенные свойства, задание 445
продолжительности 451
простые 162–64
прыгающий мяч 479–83
сброс к базовым значениям 448–49
свойства как цели 263
скорость 443, 460
смены состояний 439
совместное использование ресурсов 450–52
сплайновые ключевые кадры 472–79
сравнение анимации с синхронизацией по кадрам и по времени 439–42
сравнение с анимациями посредством CompositionTarget.Rendering 454
тип свойства 442
трансформаций 156
трансформации перспективы 488–93
триггер по событию Loaded 461–68
ускорение 460–61
флаги для диагностики проблем производительности 456–58
функции сглаживания 483–87
цели анимаций ....см. также целевые объекты
щелчок и разворот 443–49
анимации с синхронизацией по времени 439–42
анимации с синхронизацией по кадрам 439–42
зависимость от оборудования 441
анимации типа double
DoubleAnimationUsingKeyFrames 459
обрабатывающие потоки 455
анимация
цели анимации 442–43
анимация прыгающего мяча 479–83
анимированный свойства 447
анонимные методы 87–88
аппаратная клавиатура
ввод с нее 254–62
дополнительная программная клавиатура 397
аппаратные возможности 19
аппаратные клавиатуры 19
программы, соответствующее расположение 45
аппаратные кнопки 17
аппаратные средства 17–19
аппаратные устройства GPS 19
Армстронг, Нил 171
асинхронное чтение 73

асинхронный Веб-доступ 572
атрибуты ....27, см. XML-атрибуты
свойства-атрибуты 135
атрибуты XML 27
афинные преобразования 488
аффинные преобразования .... 159, см. также трансформации

Б

байтовые массивы
преобразование в MemoryStream....435
Бах, Иоганн Себастьян 617
Безье 55, правило....465
Безье, Пьер Этьен....382
бесконечные циклы 279
библиотека динамической компоновки (DLL), создание 275
библиотека изображений
выбор фотографий.... 424–29
запись имени файла....429
приложения расширений для обработки фотографий 430
растровые изображения, сохранение....402
сохранение изображений 424–29
сохранение фотографий.... 424–29
библиотека классов Silverlight для создания анимаций, синхронизация по времени 442
библиотека фотографий 79–82
доступ 79
организация доступа....618
библиотеки
объявления пространств имен XML....281
библиотеки XNA, доступ к библиотеке изображений 424
библиотеки, общие 15
бизнес-объекты
как источники привязки 310–11
привязка к 310
элементы управления списками, заполнение 566–80
бинарные объекты, загрузка из Интернета....71
битовое разрешение
как наложение.... 430
уменьшение 430–38
битовые флаги....58
большой экран 17
Боттичелли, Сандро 529

В

ввод.... см. также сенсорный ввод
элементы для этого.... 228
ввод посредством клавиатуры....65
ввод посредством мыши 65
ввод посредством стилуса....65
ввод с клавиатуры
TextBox 254–62
веб-сервисы, доступ с помощью прокси....96
векторная графика 154–55, 344–99
ArcSegment 376–82
Canvas....345–47
Grid....345–47
Path Markup Syntax 390–94
PathGeometry....374–75
ZIndex 347–48
библиотека Shapes....344–45
в растровых матрицах....412–16
векторы .... см. векторы
вырезание.... 390
геометрические элементы....364–67

геометрические элементы, группировка 373–74
геометрические элементы, трансформирование ... 369–73
динамические многоугольники ... 361–64
заливки ... 358–60
кривые Безье ... 382–90
линии ... See lines
многоугольники ... 358–60
наконечники ... 353–58
обзор ... 638–42
перекрытие ... 347–48
перемещение спрайтов ... 642–44
полилинии ... 348–52
преобразование в растровую ... 401
произвольные кривые ... 348–52
свойство Stretch ... 361
соединения ... 353–58
создание кривых с помощью класса Polyline ... 349–52
спирали, создание ... 351–52
точки ... 639
трансформации ... 369–73
штрихи ... 353–58
элементы Path ... 364–69
векторы
длина линии ... 641
местоположения ... 639
модули, сравнение ... 641
модуль ... 83, 638–39
направление ... 83, 638–39, 642
обозначение ... 638
определение ... 638
представление ... 640–41
радианы, преобразование ... 642
сложение, вычитание, умножение ... 640–42
векторы ускорения ... 83
направление ... 84–85
отображение ... 85–88
скорость ... 84
вертикальная прокрутка..189, см. также полосы прокрутки
вибрация ... 19
видеофайлы, воспроизведение ... 173
видимость объектов, и касание ... 186
визуальная обратная связь от элементов управления
изменение ориентации ... 230
описание в XAML ... 275
отображение неактивного состояния ... 288
разделение на модули ... 275
визуальная обратная связь элементов управления ... 514–22
кэширование визуальных объектов ... 453
перемещение и изменение ... 452–54
производительность ... 453
свойства-зависимости ... 300
визуальные объекты ... 26
визуальные примитивы ... 228
визуальные состояния ... 515
взаимное исключение ... 515
изменения, реакция на них ... 515, 521
определение ... 515
состояние Normal ... 516–18
состояние Pressed ... 518–19
визуальные элементы
как источники привязки ... 299

привязка к источникам данных......... см. также привязки данных

визуальный стиль..505

вложение

элементов StackPanel................................. 184–85

элементы Grid...207

вращение.. 158, 162, 648

вокруг осей X, Y и Z..488

направление..488

относительно точки...649

сложное..488

текста... 651–56

угол, вычисление....................................... 658, 672

ускорение..652

время игры...638

встроенные ресурсы, доступ................................151

вторичные страницы в качестве диалоговых окон..108, 112–14

выбор, отображение...553

вывод изображения, в режиме «letterbox»..... 47

выключатели, три состояния....................... 283–88

выполнение

прерывание.. 22

выполнение программы, прерывание............. 22

вырезание

анимации..455

заполнение областей...375

с помощью геометрических элементов.....390

Г

географические координаты................................ 93

геометрические элементы........... 364–67, 369–73

вырезание...390

группировка.. 373–74

получение границ..414

размер, вычисление..414

трансформации..369–73

гистограммы

DataTemplate..589–94

создание...291–92

Глобальная система позиционирования..........92

горизонтальная прокрутка...... 189–90, см. также полосы прокрутки

ГП, анимации...439, 455

градиент..152–53

градиентные кисти

для контрастных цветов..................................152–53

значения смещения..139–40

начальная и конечная точки..........................140

создание...138–40

градиентные цвета, применение...............414–15

графические поля

свойство SupportedOrientations........................48

графические элементы

векторная графика..344–99

растеризация...457

растровая графика.......................344, 400–438

упрощение..453

графический процессор (ГП)

анимации..439

двухмерные анимации...................................455

грязные области..457

Д

данные

отображение в виде картотеки........... 595–604

передача на страницы................................. 108–9

поля, хранение..114

презентаторы ... 561–66
промежуточные данные ... 114
совместное использование страницами 109–14
хранение вне экземпляров ... 114–16
данные состояния
объекты PhoneApplicationService, хранение ... 417
хранение ... 119–21
хранение в изолированном хранилище ... 114
датчики ... 19
двунаправленные привязки ... 301–3, 337
источник, обновление ... 338–40
двухмерная система координат ... 639
двухмерные игры ... 16
деактивация приложений ... 118
действие игры ... 16
делегаты ... 86
деревья визуальных элементов ... 27
ApplicationBar ... 210
TemplateBinding ... 508–9
анализ ... 501–5
замена ... 228
объект Content, отображение ... 499
отображение ... 503, 501–5
проход системы компоновки ... 192
свойства пользовательских элементов управления, передача ... 277
упрощение ... 453
формирование событий ... 233
шаблоны ... 497
элементов управления списками, отображение ... 549–52
Джонсон, Роберт ... 615
диалоговые окна
создания ... 249–50
диалоговые окна, вторичные страницы ... 108, 112–14
динамическая компоновка ... 29, 39
свойство HorizontalAlignment ... 39
свойство VerticalAlignment ... 39
динамические многоугольники ... 361–64
дисплеи
пропорции ... 18
разрешение ... 31
сравнение портретного и альбомного режимов отображения ... 18
энергопотребление ... 18
дисплей
частота обновления ... 162
длинный пробел, символ ... 189
длинный пробел, символ Unicode ... 189
дополнительная программная клавиатура ... 397
дочерние элементы
дочерние элементы, вызовы Measure ... 295
компоновка ... 192–93, 197
размер ... 195–96, 198
расположение в ряд друг над другом ... 199
трансформации ... 167
дуги
начальные и конечные точки, определение алгоритмически ... 381–82
окружности, на основании ... 376–79
отрисовка ... 376–81
поведение отрисовки по умолчанию ... 376
эллипсы, на основании ... 379–81

Ж

ЖК-дисплеи, энергопотребление ... 18

З

завершение приложений 118

завершение, приложения 119

в стеке 117

заголовки страницы 102, 103

загрузка элементов из Интернета 71

задачи выбора 76

захоронение 78–79

задачи выполнения 76

задачи запуска

захоронение 78–79

задний буфер 46–47

настройки по умолчанию 47

заливки

для многоугольников 359

для полилиний 359

пересечения контурных линий 359

правила заливки 359–60

запуск приложений 118

захоронение 78–79, 102, 116–18

в XNA 124–29

изображения 416–23

массив jpegBits, сохранение 434

обработка 429

приложения для отсчета истекшего времени 251–53

состояния игры 422–23

знак бесконечности, отрисовка 673–74

значение приращения 597

значения DateTime, вычисление 52

значения пикселов

альфа-канал 409

компоненты 409

значения по умолчанию 135

для свойства Background 136

определение 267

приоритетность 266

значения свойств

TemplateBinding 508–9

задание, с помощью привязки данных 308–10

механизмы уведомления об изменениях310–11, 315

по умолчанию 509–10

явное задание в коде 508

значки

активация и деактивация 214–15

объекта ApplicationBar 210–17

ориентация 213

папка 211

И

игровой цикл XNA 16

игровые циклы 35

игры .... 16, см. также программы на XNA

игры на XNA

альбомные режимы отображения, ограничение 48

изменение ориентации, реагирование 48

портретное расположение 46

изменение компоновки, уведомление 42

изменения в тексте

сохранение 261

изменения данных

отображение 566

сохранение 397

уведомление 310–11, 315

изменения текста

сохранение....397
изменения, отображения....566
изображение
камера, создание....76–79
изображения ....см. также растровые изображения
библиотеки изображений, сохранение в них 424–29
для значков....210
захоронение....416–23
из Интернета....71–73
логика перемещения....420–22
перемешивание, логика рандомизации....421
разделение и перемешивание....416–23
растяжение....171–72
трансформация....172–73
формат JPEG, сохранение....422
изолированное хранилище....121–24
для параметров приложения....121–24, 245
для хранения данных между выполнениями 116
доступ....114
хранение состояния....114
имена ключей в разделе ресурсов....148
имена свойств....446
имена файлов, предложение....436
инерция
события Manipulation....62
инициализация объектов в XAML....131
Интернет
загрузка элементов из....71
изображения из....71–73
интерполяция
вычисление....662
интерполяция кубическим сплайном Эрмита 675
с помощью Vector2.Lerp....670
с помощью Vector2.SmoothStep....670
интерполяция кубическим сплайном Эрмита 675
источники данных....см. также объекты-источники
визуальные элементы, связывание....298, см. также привязки данных
источники привязки, объекты данных в их качестве....627
исходные страницы
передача данных от них....108–9

К

кавычки в расширениях разметки....146
камера, создание изображений....76–79
камеры....19
картографический сервис....95–101
картотека....595–604
касание
Canvas....206–7
видимость объектов....186
касания, выявление....165
перемещение текста, реакция....651–53
приложение Jot....221–23
касание, отслеживание....55
касания
отслеживание....55
касательные к кривой....675
для сплайна, вычисление....676
квадратичные кривые Безье....382–86
параметрические уравнения....385
кисти

в XAML ... 138–41
мозаичные кисти ... 175
свойство Content, задание ... 497
совместное использование ... 145–48
кисть Fill для Polyline ... 359
клавиатуры, аппаратные и экранные ... 19
классы
наследование новых классов ... 195, 263
ссылки ... 280
экземпляры, ссылки ... 271–72
классы пользовательских элементов управления
наследование от класса Panel ... 194–200
создание ... 263
ключевые кадры ... 458–61
значения по истечении времени ... 443
изменения скорости ... 472
на базе сплайнов ... 472–79
продолжительность ... 460
кнопки
активация и деактивация ... 214–15, 217, 226
анимация ... 443–49, 458–61
нулевой размер ... 405
обработчики событий Click ... 215, 223
производные от класса EasingFunctionBase ... 487
рамки вокруг ... 506–8
код
XAML-ресурсы, использование ... 148
для обработки событий ... 131
класс TextBlock ... 132–34
сортировка элементов ... 582–84
элементы управления списками, заполнение ... 548–54
кода, загрузка растровых изображений ... 75–76
коллекции
выбор элементов ... 547, см. также Selector, класс
динамическое добавление и удаление элементов ... 561
добавление элементов ... 549
изменения, ответ на них ... 361
как источник CollectionViewSource ... 583
музыка ... 620–21
отображение ... 547, см. также элементы управления списками
теги ... 142
компиляция ... 26
компоненты приложения
организация в горизонтальном направлении ... 605, см. также Panorama, элементы управления; также Pivot, элементы управления
компоновка, постраничная навигация ... 605
конвертеры ... 303–6
StaticResource, ссылки на них ... 304
для числовых типов данных ... 331–32
конвертер BooleanToVisibilityConverter323–26
открытые свойства ... 331
формируемые исключения, перехват ... 340
конкатенация текста в элементах StackPanel ... 182–84
конструкторы экземпляров, выполняющиеся один раз ... 624
конструкторы, триггеры событий ... 233
контейнер для содержимого
элементы StackPanel ... 182
контейнеры ... 553
контрастные цвета ... 151

для градиентной кисти ... 152–53
контуры, перемещение спрайтов вдоль 663–76
координаты
класса Line ... 346
отрицательные ... 347
корневой элемент
присваивание имен ... 310
корневые элементы ... 24
свойство Style ... 524
кривые
касательные ... 675
касательные к ... 676
класс Polyline ... 349–52
кривые Безье ... 382–90
параметрические уравнения, использование для создания ... 349–50
перемещение спрайта вдоль них ... 673–76
произвольные кривые ... 348–52
кривые Безье ... 382–90
квадратичные ... 382–86
множества ... 389
опорные точки, задаваемые пользователем ... 475
четверть окружности ... 465
круги ... 155
кубические кривые Безье ... 385–90
границы ... 389
параметрические уравнения ... 386
куча, распределение памяти ... 36

Л

Леди Гага ... 615
линейная интерполяция ... 643
линии ... см. также Line, класс; также Polyline, класс
наконечники ... 353–54
пунктирные линии ... 355–58
соединения ... 353–55
Ллойд, Сэм ... 416
локальные параметры ... 135
анимации, приоритетность ... 495
приоритетность ... 266
локальные растровые изображения, загрузка из кода ... 75–76
лямбда-выражения ... 87

М

маленький экран ... 18
массив значений типа int ... 408
массивы байтов ... 422–23
масштабирование ... 161, 648
на основании размера экрана ... 675
относительно точки ... 649
отсутствие масштабирования ... 649
по вертикали ... 651
текст ... 158
центр ... 648
масштабирование по вертикали ... 651
межстрочный интервал ... 30
меню ... см. ApplicationBar
меню Debug, опция Stop Debugging ... 22
место касания
выявление ... 55
методы экземпляров
доступ ... 270
статические методы, вызов ... 271–72
механизмы уведомления ... 310–11, 315
мишени касания, размеры ... 61
многоугольники, динамические ... 361–64

моделирование свободного падения...... 479–83
модуляция цветового канала ...657
мозаичные кисти ...175
монитор
частота обновления...162
Монтеверди, Клаудио...615
музыкальная библиотека ... 615–18
воспроизведение музыки...629
организация доступа... 620–21
отображение музыки...629
мультисенсорный ввод
сохранение данных...222
Мунк, Эдвард ...529

Н

навигация ...102–8
XAML-файлы...105
в Silverlight...102–8
возвращение на страницу...104
диалоговые окна...108
исходные страницы и страницы перехода107
по стеку... 116–17
постраничная навигация ...605
структура стека...107
нажатие клавиш, обновление... 338–40
наклонение... 159–60
наконечники линий... 353–54
пунктирные линии ...356
наследование
значений свойств...508
наследование свойств... 241–42
привязки данных...310
приоритетность ...266
производные стили ... 254
стилей ...150–51
наследование свойств ...134–35, 241–42, 508
приоритетность ... 266
Насыщенные Интернет-приложения ...16
начальные страницы ... 107
неактивное состояние, обозначение ...514–22
неафинные преобразования...488, см. трансформации
недействительный ввод, визуальное обозначение ...340–43
непрозрачность ...173–75
несериализуемые объекты, хранение ... 129
новые проекты, создание ...20

О

область содержимого ...28
линии сетки ... 473
обрабатывающий поток... 439
выполняемые в нем анимации 455–57, 470
обработка касания
простого, в Silverlight... 59–62
простого, в XNA... 54–57
сложного, в Silverlight... 62–64
обработка маршрутизированных событий .. 65–66, 340
обработка простого касания
в Silverlight ... 59–62
в XNA ... 54–57
обработка сложного касания в Silverlight 62–64
обработчик событий Completed ...484–85
обработчик событий DragDelta ... 543
совместное использование... 539
обработчик событий DragStarted ... 543

обработчик событий ManipulationStarted 62–63, 65
обработчик событий OnAccelerometerReadingChanged 86, 89
обработчик событий OnColorChanged 269
обработчики кнопок 494
обработчики событий 43
в привязках данных 298
параметры приложения, сохранение 128
параметры приложения, хранение 129
прикрепление к событиям в коде 49–53
создание 42
обработчики событий Click 215
анимации, определение и запуск 444
для кнопок 223–24
обработчики событий ColorChanged 282
обработчики событий Completed 78, 79
обработчики событий CompositionTarget.Rendering 456
синхронизация с частотой кадров 440
обработчики событий Loaded 97
обработчики событий OnColorColumnValueChanged 281
обработчики событий PointChanged 388–89
обработчики событий ReadingChanged 85
обработчики событий SizeChanged 160–61, 202–3, 543
обработчики событий TextChanged 258, 339
обработчики событий Touch.FrameReported 59–62, 222–23, 600
совместное использование 601
обработчики событий ValueChanged 233, 281, 546
обработчики событий изменения значения свойства 267, 269
вызовы метода InvalidateMeasure 289
для присоединенных свойств 294–95
для свойства IsOpen 602
для свойства Label 277
закрытый флаг 274
написание кода 271
поддержка свойств-зависимостей 361
реализация 271
совместное использование свойствами 271
статические 286
общие библиотеки 15
объектная ссылка не указывает на экземпляр объекта, ошибки 265
объектные элементы 135
объекты
конвертирование в XAML 394
определенные как поля 78
потокобезопасность 50
связывание ...298, см. также привязки данных
сериализуемые объекты 115
текущий поток, доступ из 86
трансформации сужения 488
объекты Dictionary для данных касания 55
объекты данных в качестве источников привязки 627
объекты-источники .... 298–300, см. также привязки данных
бизнес-объекты 310–11
инициация обновлений 339
механизмы уведомления 310–11, 315
обновление 338–40
объекты данных 627
преобразование данных 303–6
присваивание имен 298

свойства 311
свойство ElementName 298
свойство Path 299
объявления пространств имен XML 281
одинарные кавычки в коде конвертера 306
окна сообщений 225
окно просмотра
извлечение..48–49
как поле 637
текст, масштабирование............................. 649–51
окружности
аппроксимация с помощью кривых Безье 465
местоположение, указание...............................366
отрисовка.. 350–51
точки на них 670
точки на них, вычисление 674
Олдрин, Баз 171
операции перетягивания 367, 384, 385–88
очистка 368
операции рисования
коэффициенты переноса 368
новая, инициация 367
описания .NET-свойств 268
описания пространств имен 24, 197
опорные точки объектов KeySpline 477
опорный элемент 60
определение местоположения 19, 92–95
картографический сервис 95–101
параллели 93
широта 93
опрос 20
ориентация 18, 39–53
ApplicationBar 213
в XNA 46–49
реакция телефона на ее изменение 39–40
элементы Grid 179, 209
элементы StackPanel....................................180–82
ОСИД..18
ось вращения 490
открытые поля типа DependencyProperty...... 267
открытые свойства
в классах конвертеров 331
как параметры приложения 220
коллекция Resources, задание 326
открытые статические поля 267
отладка 619
отладка приложений для работы с камерой ..77
относительные координаты 158
отображаемое время, обновление......................52
отображение панели задач 44
отражение
заполнение StackPanel................................187–92
заполнение элементов управления 486
отрисовка по требованию..51
ошибки времени выполнения 233, 254
ошибочный ввод, обработка 333, 338–40

П

память, куча, распределение 36
панели........................27, 40, см. также Panel, класс
UniformGrid.. 288
виртуализирующие панели............................. 553
вызовы ArrangeOverride.................................... 295
для ItemsPresenter 552
дочение элементы, задание размера и местоположения 295–96

метод InvalidateArrange 289
методы InvalidateArrange 295
определение размеров, расстановка и пересчет размеров элементов компоновки 289
перегрузки методов 194, 195
присоединенные свойства, изменения 295
свойства, автоматическая обработка 195
свойство ZIndex 205, 348
со свойствами.. 288–93
элементы управления списками 587–89
ячейки, вычисление размеров 289–91
панели заголовков, позиционирование........... 96
панель задач 27
отображение на экране....................................... 44
при альбомном режиме отображения......... 45
сокрытие 47
панель приложения 210–17
папки для хранения изображений 69
параллели... 93
параллелизм, сохранение 488
параметрические уравнения
для эллипсов 670–72
перемещение текста 644–47
параметрические уравнения, создание кривых 349–50
параметры приложения 122
в изолированном хранилище 245
доступ 222
загрузка...122
сохранение в обработчиках событий128, 129
хранение 122, 121–24
хранение в изолированном хранилище 114
переворачивание...648
перегруженные методы навигации...........426–27
перегрузки методов в классе Panel 194, 195
передача данных на страницы....................... 108–9
переключение задач 116–18
перекрытие в векторной графике 347–48
перемещение мыши, приостановка....................61
перемещение спрайта.............. 673–76, см. также перемещение текста вдоль кривых
вдоль контуров...663–76
вдоль полилиний...666–70
вдоль эллипсов...670–73
по прямоугольной траектории................663–66
перемещение спрайтов 636
с использованием векторной графики642–44
перемещение текста... 636
в одном измерении.....................................636–38
вычисления в методе Update 636–38
дистанция перемещения 638
задание темпа 638
замедление и ускорение 647
изменение направления перемещения...644–45
константа SPEED 636–38
константа TEXT.. 637
местоположение текста, вычисление ..643–45
местоположение, вычисление 645, 647
огибание углов 653–56
приращение координат 643–44
простой подход 636–38
с использованием параметрических уравнений...644–47
с помощью векторов..................................642–44
скорость, вычисление 645–46
скорость, отображение 652

средняя точка, вычисление 643
ускорение 651
пересчет размеров элементов компоновки, инициация 596
пикселы
прозрачность 408–10
пикселы и пункты, сравнение 30–32
платформы, выбор... 15
плотность пикселей.. 22
поле состояния, обновление 652
ползунки, использование....................................229
полилинии 348–52
динамические... 362–64
перемещение спрайта вдоль.................. 666–70
полосы прокрутки... 187–92
использование 229
полосы проктурки
параметры 200
пользовательские интерфейсы fluid 605
пользовательские настройки, переопределение 136
пользовательские элементы управления
ContentControl, наследование от..................542
Control, наследование от 542
UserControl, наследование от 541
описание элемента Style 525–29
свойства, передача в дерево визуальных элементов..277
сменные шаблоны..275
создание 541–46
пользовательский ввод ....27, см. также сенсорный ввод
в программах на XNA 36
мультисенсорный 19
элементы для этого... 228
поля
объекты, определение...78
свойства-зависимости 267–68
хранение данных 114
портретный режим отображения 18
постраничная навигация...................................... 605
постукивание..55
поток пользовательского интерфейса (UI-поток) 86
потоки компоновщика................................ 439, 455
потокобезопасность, события и объекты Silverlight 50
правые системы координат 84
предварительно умноженные альфа-значения 408–10, 412–16
презентаторы 561–66
преобразование данных в привязках данных 303–6
привязки данных...................................131, 298–343
Path= part, исключение.................................... 575
TemplateBinding... 508–9
двунаправленный режим...........330, см. также двунаправленные привязки
инициация обновлений 339
источник this 307–10
источники и цели 298–300
к бизнес-объектам 310
к свойству Visibility.. 338
конвертеры....................................... 302–6, 325–28
корневой элемент, присваивание имен 310
механизмы уведомления.................310–11, 315
наследование через дерево визуальных элементов 310
новые значения свойств, задание 308–10

обновления привязки TextBox ... 333–43
определение ... 298
относительные источники ... 307
параметр Mode ... 301
повторения в них, устранение ... 316–18
преобразование данных ... 303–6
привязки элемент-имя ... 298–300
пример ... 625–26
принятие решений ... 321–24
приоритетность свойств-зависимостей ... 301
разделение ... 574
свойства, задание ... 263
свойства-зависимости ... 300–302
свойство DataContext ... 316–21
свойство Source ... 307
серверы привязки ... 311–16
сервисы привязки для передачи и приема ... 328–33
синтаксис ... 300
статические классы ... 624
функциональность ... 298
цели ... 475
цели и режим ... 300–302

привязки элемент-имя ... 298–300
свойство DataContext ... 318

пригодные для повторного применения элементы управления ... 275

приложение ... см. также приложения на XNA; приложения на Silverlight; также приложения на Silverlight; также приложения на XNA

приложение "здравствуй, телефон" на XNA ... 38

приложение Windows Phone Developer Registration ... 23

приложение для создания заметок ... 258–62

приложениясм. также программы на Silverlight; также программы на XNA
активация ... 118
деактивация ... 118
завершение ... 118
запуск ... 118
написание ... 15
совместное использование ресурсов ... 523

приложения для набора текстов ... 258

приложения для работы с камерой, отладка . 77

приложения для работы с музыкальными файлами
классы пространства имен Microsoft.Xna.Framework.Media ... 620
обложка альбома ... 620
проект MusicByComposer ... 615–20
сервисы XNA ... 618

приложения расширений для обработки фотографий ... 19, 430–38

приложения с поддержкой мультисенсорного ввода ... 19

приложения с реализацией мультисенсорного ввода
тестирование ... 54

приложения, моделирующие часы ... 49–53

приложения, не зависящие от ориентации экрана ... 45

принудительное задание свойства, приоритетность ... 496

приоритетность
анимаций ... 495
принудительное задание свойства ... 496
свойства ... 242

приоритетность параметров ... 266–67

присоединенные свойства .. 45, 202, 205, 293–97

анимация 468–72
доступ 294
задание в XAML 451
задание для анимаций 445
свойства ZIndex 205
синтаксис 293
проверка совпадения, и заполнение областей 375
программы .см. также программы на Silverlight; также программы на XNA
доступ к картам 16
компиляция 26
на Windows Phone Marketplace 16
учет местонахождения пользователя 16
программы на Silverlight
XNA-классы 81
динамическая компоновка 29
ориентация 18
папки для хранения изображений 69
программы на XNA
значения с плавающей точкой 34
игровые циклы 36
имена ресурсов 34
инициализация 35
каталог Content 35
класс WebClient 71–72
ориентация 18
пользовательский ввод 36
проекты для содержимого 33
содержимое, загрузка 35
спрайты, отображение 36
статические экраны 51
структуры 36
файлы C# 34–35
шрифты 32–35
шрифты, добавление 32
шрифты, редактирование 33
проект "Здравствуй, мир" на Silverlight 20
проекты библиотек
UserControl, создание 276
добавление новых элементов 276
проекты для содержимого, XNA 33
проекты приложений 275
проекты содержимого, добавление растровых изображений 68
прозрачность 137, 664
пикселов 408–10
прозрачность изображений 664
прозрачность растровых изображений 664
прозрачный белый 410
прозрачный черный 410
производительность
визуальные объекты 453
проблемы, флаги для диагностики 456–58
соотношение размеров 47
производные классы 263
от UserControl 275–83
произвольные кривые в векторной графике 352
прокси
вызов 98
для веб-сервиса
связь 96
использование 97
промежуточные данные 114
в XNA, хранение 125

хранение в словаре State 114–16
пропорции 18
пространства имен .NET, и пространства имен XML, ассоциирование 145–47, 263–64
пространства имен XML
пространства имен .NET
ассоциирование 145–47, 263–64
пространства имен XML, и пространства имен .NET, ассоциирование 145–47
прямоугольники
перемещение спрайта по периметру.. 663–66
Публичная библиотека города Эль-Пасо 566
пунктирные линии 355–58
пункты и пикселы, сравнение 30–32
пункты меню
обработка 225–26
отображение 224

Р

разделы ресурсов 145, 147
имена ключей 148
определение ресурсов 146
размеры дисплея 17
плотность пикселей 22
получение 49
эмулятор 22
размеры шрифтов
изменение 260
размеры шрифтов, пункты и пикселы 30–32
разработка программ 15–17
процесс разработки 20
разрешение, дисплей 31
рамки 27
страницы в них 27
рамки для элементов 165–68
раскадровки
запуск 490
остановка 452
продолжительность 447
распознавание жестов 57–59
растровая графика 344, 400–438
векторная графика, преобразование 401
растровые изображения .... 67–82, см. также изображения, см. также изображения
Build Action, сравнение Content и Resource 75–76
алгоритмы сжатия 67
векторная графика 412–16
встраевыемые или загружаемые 71
для изображений значков 210
добавление в проекты содержимого 68
добавление как ссылок 70
иерархия классов 400–402
изменение размеров 171–72
код, загрузка из 75–76
кэширование 458
масштабирование 92
метод Draw и его разновидности 657
мозаичное заполнение 98–101
обработка на уровне пикселов 67
определение цвета 663
отображение 400
отрисовка элементов .... 400–402, см. также WriteableBitmap, класс
пикселы, доступ 401
позиционирование 68–69
разделение на части 405–8
размер 400

растяжение 171–72
соотношение размеров 70
сохранение 82, 400
сохранение в библиотеку изображений ...402
ссылка по URI 400
форматы файлов 67
формирование алгоритмически 344
формирование визуального представления 69
растровый шрифт 651
расширения разметки 146
Binding 299
кавычки 146
расширения разметки XAML 146
Расширяемый язык разметки приложений 16
реализация многозадачности в UI 116–18
режим «letterbox» 47
ресурсы .... см. также XAML-ресурсы
совместное использование 116
совместное использование приложениями 523
хранение 24
шаблоны, определение 506
ресурсы проекта, добавление в XAP-файлы .. 32
ресурсы, добавление в XAP-файлы 32

С

свободное падение 651
свойства
анимация .... 263, 442–43, см. также анимации
доступ через привязки данных 311–16
задание через привязки данных 263
задание через стили 263
использование 319
как объекты-источники 311
обновление значением другого свойства 298, см. также привязки данных
параметры, приоритетность 266–67
передача в дерево визуальных элементов 277
приоритетность 242
приоритетность и анимации 493–96
приращение 449
присоединенные свойства 202, 293–97
свойства экземпляров 270
свойства-зависимости 266–74
сравнение свойств и свойств-зависимостей 274
свойства содержимого 142–44
свойства экземпляров, доступ 270
свойства, допускающие пустое значение 110
свойства, присоединенные 45
свойства-атрибуты 135
свойства-зависимости 134, 266–74
.NET-свойства 266
базовые значения 448–49
для визуальных объектов 301
для привязок данных 300–302
задание 269
изменения, сигнализация о них 278
как цели анимации 442
механизмы уведомления 310, 315
поддержка обработчиков событий изменения значения свойства 361
применение 263–66
приоритетность 301
приоритетность и анимации 495
производных от класса Button 266

синтаксис.. 266–68
создание..294
сравнение со свойствами..................................274
только для чтения...274
фактические значения 273
свойства-элементы 135–36
цвета...138
целевые свойства 298
свойство Property типа DependencyProperty 271
сенсорные панели.. 54
сенсорный ввод218, см. также жесты, см. также касание
в Silverlight.. 54
в XNA.. 54
касание, определение 602
маршрутизация.. 65
объект Dictionary.. 55
серверы привязки 311–16
сервис определения географического местоположения 92–95
сервисы 19
сервисы XNA, воспроизведение музыки........618
сервисы облака 16
сервисы принудительных уведомлений........... 20
сервисы уведомлений, принудительное уведомление 20
сериализация XML 570
сериализуемые объекты.............................115, 126
сетка для содержимого 28
коллекция Resources..188
масштабирование 393
элементы, размещение..................................... 40
символ возврата каретки 168
символ перевода строки 168
синтаксис описания графических элементов 390–94
синтаксис свойство-элемент 135–36
свойство Template, задание 506
система компоновки 192–93
Canvas... 200
видимость объектов 185–87
дерево визуальных элементов, проход по нему 192–93
класс Panel 177–78
свойство Opacity 185–87
системы координат
относительные координаты............................ 158
системы координат, правая 84
скорость
события Manipulation 62
словари
хранение данных 622
словарь
для промежуточных данных....................114–16
хранение 119–21
хранение данных 114–16
смена страниц 490–93
сменные шаблоны 275
смены состояний 439
события
потокобезопасность 50
события Manipulation.. 62–64
источник 65
события завершения 97
события изменения ориентации экрана... 45–46
события касания
приравнены к событиям мыши 60

свойства Margin и Padding 42
события мыши, приравнены к событиям касания 60
совместное использование данных страницами 109–14
совместное использование ресурсов 116
описываемые в XAML анимации 450–52
содержимое
выравнивание 514
отображение в производных от ContentControl 512
соединения линий 353–55
создание текстуры на XNA 68–69
создание экземпляров 131
в C# 133
сокрытие элементов 185–87
соотношение размеров
производительность и энергопотребление 47
растровые изображения 70
сортировка 582–87
состояние игры, сохранение 422–23
состояние страницы 118–21
состояния элементов управления 439
спирали 351–52
сплайновые ключевые кадры 472–79
сплайны
опорные точки, перемещение 473
отслеживание кривой 476
сплины 382
спрайты 16, 33
изменение направления движения 671
отображение 36
ссылки, добавление как растровых изображений 70
ссылочные типы, значения по умолчанию .. 267
стандартные файлы Silverlight 23–29
стандартные шаблоны по умолчанию 522
статические классы, и привязки данных 624
статические конструкторы, выполнение 624
статические методы 204, 269
методы экземпляров, вызов 271–72
присоединенные свойства, доступ к ним . 294
стек
навигация 107, 116–17
очистка 117
реализация многозадачности 117
стек приложения ..... см. стек
стили 149–50
в Silverlight 144
для класса Button 253–54
наследование 150–51
неявные 151
применение 149
приоритетность 150, 266
свойства, задание 263
совместное и повторное использование 522–25
трансформации 159
стили темы 242, 282
приоритетность параметров 266
столбцы, параметр Margin 185
страницы 27
данные состояния, хранение 119–21
передача данных 108–9
переход к 111–12
совместное использование данных 109–14
создание 103

страницы перехода
- в качестве диалоговых окон 108, 112–14
- вызов методов 112
- задание свойств........112
- инициализация 113
- передача данных в них........108–9

страницы перехода 107

стратегия Майкрософт на рынке мобильных телефонов........ 15

строки........см. также строки текста
- загрузка из Интернета........ 71
- позиционирование........ 94
- хранение 121
- целевые свойства, задание........446

строки HTML-запросов........109

строки текста
- размещение........ 40

строки форматирования .NET 306

строковые типы........ 52

строковый ключ «token» 438

структуры........ 36

Т

таблицы, элементы StackPanel 185

таймеры, тактовые частоты........162

теги
- для коллекций 142
- теги BeginStoryboard 462
- теги EventTrigger........462
- теги PhoneApplicationPage 606
- теги Setter, для шаблонов 506
- теги String, разграничение элементов 557
- теги TemplatePartAttribute 530
- теги TemplateVisualStateAttribute........530
- теги UserControl........ 396
- теги VisualState........516–18
- теги VisualStateGroups 516
- теги свойства-элемента........ 238
- теги свойств-элементов 136
- удаление 142

теги свойств-элементов........ 136

текст
- анимация........ 457
- вращение........439–42, 648, 651–56
- масштабирование........ 158, 648
- масштабирование до заполнения окна просмотра 649–51
- масштабирование по вертикали 651
- межстрочный интервал 30
- отображение с помощью XAML 323
- переворачивание........ 648
- позиционирование на экране 648–50
- положение, вычисление........ 654
- привязка к элементам ListBox 559
- форматирование, задание........ 321
- центрирование........ 636
- центрирование по горизонтали 637
- частые изменения........ 648
- эффект выпуклого текста........156–58

текстовые сообщения, отправка........254–62

текстовые строки........см. также строки
- в элементах управления списками........ 556
- определение размера........36
- совместное использование........ 145

текстуры........67–82, см. также растровые изображения
- сведения о цвете 657–58

соотношение размеров 80
текущий поток, доступ к объектам 86
телефоны
аппаратные возможности 19
аппаратные кнопки 17
библиотеки фотографий 79–82
клавиатуры 19
разблокирование для разработки 23
устройства GPS 19
темы 151
контрастный цвет 29
цвет фона 29
теорема Пифагора 349, 641
тестирование
проекты Windows Phone Class Library 275
тестирование кода реализации мультисенсорного ввода 54
технология емкостного касания 19
технология ОСИД (органический светоизлучающий диод), энергопотребление 18
типы значения, значения по умолчанию 267
точечный пунктир 357–58
точка вставки .... см. точка вставки текста
точка вставки текста 261
SelectionLength, свойство 261
местоположение 261–62
свойство SelectionStart 261
точки
месторасположения 639
точки касания 19
трансформации 155–62, 369–73
EllipseGeometry 371
анимации 455
анимация 156, 442
афинные и неафинные преобразования .. 488
аффинные и не аффинные преобразования 159
в стилях 159
вращения 158, 162
геометрических элементов 369–73
дочерние элементы 167
изображений 172–73
масштабирование 158, 161
наклонение 159–60
объектов Geometry 370
перспектива 488
порядок 159
события манипуляций, обработка 164–65
сочетание 160
трансформации визуального представления 166–68
центры фигур, задание 369
трансформации визуального представления 166–68
трансформации перспективы 442
анимация 488–93
трансформации проекции
для смены страниц 490–93
трансформации проекций 162
трансформации сужения 488
трехмерное пространство
вращение 490
моделирование 488
трехмерные игры 16
триггеры 461
триггеры событий, размещение 461

У

уровень нивелира ... 88–92
положение «пузырька», вычисление ... 91
размер пузырька, вычисление ... 91
ускорение
вращение ... 652
устройства
аппаратные возможности ... 19
размеры дисплея ... 17
размеры экранов ... 17
устройство Windows Phone 7
процессоры ARM ... 89
утилиты, написание ... 15

Ф

файлы App.xaml
структура ... 23–24
файлы выделенного кода ... 16, 23
фигуры ... 154–55
круги ... 155, см. также круги
объемное изображение ... 155
панели, пространство по вертикали ... 180
произвольные координаты ... 392–93
трансформации ... 155–62
эллипсы ... 154–55, см. также эллипсы
фильмы ... 173
флаги для диагностики в анимациях ... 456–58
фокус ввода, задание ... 261–62
фоновые цвета
цвет CornflowerBlue ... 37
формат истекшего времени ... 245, 249
поле suspensionAdjustment ... 252
форматирование
строки ... 337
текст ... 321
фотографии, создание с помощью камеры . 76–79
фрагменты кода, учитывающие размер экрана ... 17
функции преобразования ... 647–48
линейные ... 647
функции сглаживания ... 483–87
выбор ... 485
сохранение выбора ... 487

Х

Хигтон, Дженифер ... 615
хранение данных вне экземпляров ... 114–16
хранилище, изолированное ... 245, см. также изолированное хранилище

Ц

цвет текста
изменение ... 340–43
изменение в зависимости от значений ... 331–33
цвета
в XAML ... 136–38
задание ... 137
как свойства-элементы ... 138–41
цвета кисти, анимация ... 468
цвета шрифта, изменение ... 30
цветовое пространство scRGB ... 137
цветовые темы ... 19, 29–30, 151
целевые объекты ... 298–300
инициация обновлений ... 339
преобразование данных ... 303–6
свойства элементов ... 475
свойства-зависимости ... 315

ссылка по имени 451

целевые свойства

FrameworkElement, наследование от . 300–302

задание привязки 300

параметр Mode 301

присваивание объекта Binding 298

свойства-зависимости 300–302

свойства-элементы 298

строки, задание 446

цели анимации 442–43

свойства-зависимости 442

цели привязки .... см. целевые объекты

центрирование 41

циклы for, вычисление объектов Point 350

Ч

части, шаблон 530

частичные классы 23

частота кадров, визуализация 457

частота обновления 162

частоты обновления, экран 36

часы, динамически обновляемые 498–99

числа, совместное использование 145

Ш

шаблон

части 530

шаблон Windows Phone Application 20

шаблоны 132, 497

TemplateBinding 508–9

в качестве стилей 506

визуальные элементы управления, настройка 228

значения свойств по умолчанию 509–10

привязки 497

применение 543

пример 625–26

редактирование 522

ресурсы, определение 506

сменные шаблоны 275

совместное и повторное использование. 497, 522–25

шаблоны Slider 530–38

шаблоны элементов управления, и анимации 439

шестнадцатеричный формат задания цвета 137

широта 93

шрифты

встроенные 32

растровый шрифт 651

теги CharacterRegions 34

шрифты TrueType 169

Э

экземпляры

хранение в виде полей 114

хранение данных вне 114–16

экземпляры страниц, хранение в виде полей 114

экран

грязные области 457

обновления 439

экранные клавиатуры 19

экраны

возможности мультисенсорного ввода 19

ориентация .... 18, см. также ориентация

размеры 17

разрешение 31

частоты обновления 36

экраны, поддерживающие мультисенсорный ввод 17, 54–66

элементы.......... 27, 227, см. также XML-элементы

Silverlight.... 27

в слое визуальной компоновки 51

высота и ширина.... 42

динамическая компоновка.... 29

динамическое добавление и удаление элементов....561

для пользовательского ввода....228

для представления 228

задание высоты 590

задание высоты и ширины 582

имена, присваивание.... 43

компоновка, переопределение....205

множество свойств, отображение 578–79

непрозрачность.... 173–75

объектные элементы....135

организация 40

отличный от нулевого размер....405

отображение.... 547, 576–78, см. также элементы

отображение в виде картотеки 595–604

отображение в горизонтальной ориентации 587–89

произвольное позиционирование....202

разграничение 557

размещение.... 40

рамки 165–68

сведения о размере.... 42, 192–93

свойства Width и Height, задание....155

свойства-элементы 135–36

свойство Tag....250

сокрытие 185–87

сортировка по полу 583–85

сравнение с элементами управления ..227–29

форматирование 581–82

цвет отображения.... 584

элементы Silverlight....см. также элементы

сравнение с элементами управления ..227–29

элементы XML....27

элементы представления 228

элементы управления

визуальное представление, настройка .....497, см. также ControlTemplate

визуальные состояния 515

визуальные элементы 228

визуальные элементы, отображение неактивного состояния 288

визуальные элементы, переопределение. 228

для взаимодействия.... 228

заполнение.... 548, 550

значения по умолчанию....525–28

изменение размеров.... 285

настройка 505–14

неактивное состояние, обозначение ...514–22

новые имена, ссылка....523–25

отступы 513

привязка к заданным значениям ....508, см. также TemplateBinding

пригодные для повторного применения элементы управления.... 275

производные от класса Control....227–29

свойство Tag 250

сенсорный ввод, распознавание....476

сменные шаблоны 275

создание.... 229

состояния....439

сравнение с элементами 227–29
элементы управления для ввода текста .. 254–62
эффекты затемнения 516
элементы управления для ввода текста.. 254–62
элементы управления для взаимодействия ..228
элементы управления списками 547
дерево визуальных элементов, отображение 400–438
отображение, свойства 555
отображение, форматирование 555
панель, изменение 587–89
представление... 580–82
содержимое, определение в XAML...... 554–57
сортировка 582–87
текстовые строки 556
числа 556
шаблоны...552
элементы, добавление в код.................... 548–54
эллипсы
анимация 462–64
вращение 369
местоположение, указание..............................366
параметрическая форма 670–72
перемещение спрайта по нему..............670–73
размер... 155
точки на них.. 670
эмулятор
акселерометр 88
время запуска..21
всплывающее меню..22
выход 23
кнопка Back 22
мультисенсорный ввод, тестирование 54
повторное выполнение..23
развертывание програм 21
размер дисплея 22
эмуляторы
музыкальная библиотека 619
производительность 457
энергопотребление...18
энергопотребление, и соотношение размеров 47
эффект выпуклого текста...............................156–58
эффекты затемнения 516

Я

явное задание значений свойств в коде........ 508

Об авторе

Чарльз Петзольд пишет о разработке программного обеспечения для операционных систем на базе Windows уже в течение 24 лет. Он является автором таких известных книг, как *Programming Windows*[1] (5 издание, Microsoft Press, 1998), и шести книг о программировании на платформе .NET, включая *3D Programming for Windows: Three-Dimensional Graphics Programming for the Windows Presentation Foundation*[2] (Microsoft Press, 2007). Также его перу принадлежат две уникальные книги, являющиеся исследованием на стыке вычислительной техники, математики и истории: *Code: The Hidden Language of Computer Hardware and Software*[3] (Microsoft Press, 1999) и *The Annotated Turing: A Guided Tour though Alan Turing's Historic Paper on Computability and the Turing Machine*[4] (Wiley, 2008). Петзольд проживает в Нью-Йорке. Его сайт можно найти по адресу www.charlespetzold.com.

---

[1] Программирование в Windows (*прим. переводчика*).

[2] 3D-программирование для Windows: трехмерная графика для Windows Presentation Foundation (*прим. переводчика*).

[3] Код: скрытый язык общения аппаратного и программного обеспечения компьютера (*прим. переводчика*).

[4] Комментарии к Тьюрингу: обзор исторического документа Алана Тьюринга по вычислимости и машине Тьюринга (*прим. переводчика*).