7 Seiten und Transformationen

206 Geräteunabhängigkeit durch Text
206 Und wie viel macht das dann?
209 Punkte pro Zoll (dpi)
210 Und was ist mit dem Drucker?
211 Manuelle Konvertierung
213 Seiteneinheiten und Seitenskalierung
216 Zeichenstiftbreiten
219 Seitentransformationen
220 Speichern des Grafikstatus
221 Maße in anderen Einheiten
224 Frei wählbare Koordinaten
226 Was nicht funktioniert
226 Hello, World Transform
231 Ein Überblick
232 Lineare Transformationen
234 Vorstellung von Matrizen
235 Die Klasse *Matrix*
237 Scherungen
240 Transformationen kombinieren

Ein Hauptziel jeder grafischen Programmierumgebung ist Geräteunabhängigkeit. Programme sollten unabhängig von der Auflösung problemlos auf unterschiedlichen Bildschirmen und Grafikadaptern ausgeführt werden können. Ferner sollte es möglich sein, Text und Grafiken auf unterschiedlichen Druckern ausgeben zu können, ohne eine Vielzahl spezieller Druckertreiber zu benötigen oder gar für jeden einzelnen Drucker eigene Ausgaberoutinen schreiben zu müssen.

In Kapitel 5 habe ich demonstriert, wie Sie Code zur Grafikausgabe sowohl auf dem Monitor als auch auf dem Drucker schreiben können. Bisher haben wir jedoch nur mit Pixeln gearbeitet

(zumindest, was den Bildschirm angeht; der Drucker ist ein bislang noch ungelüftetes Geheimnis), und Pixel scheinen kaum eine geräteunabhängige Lösung darzustellen.

Geräteunabhängigkeit durch Text

Mit ein wenig Sorgfalt können Pixel geräteunabhängig eingesetzt werden. Eine Möglichkeit besteht darin, sich bei der Grafikausgabe nach der Standardgröße der *Font*-Eigenschaft eines Formulars zu richten. Dieser Ansatz ist insbesondere dann sinnvoll, wenn Sie einfache Grafiken und Text miteinander kombinieren.

Angenommen, Sie programmieren eine einfache Datenbankanwendung und verwenden dabei Symbole zur Darstellung einzelner Karteikarten. Jeder Datensatz wird als 3 × 5 Zoll große Karteikarte angezeigt. Wie groß sind die Karteikarten in Pixeln? Stellen Sie sich eine Schreibmaschine vor. Bei einer Schreibmaschine mit 12-Punkt-Schrift passen auf 1 Zoll Höhe 6 Zeilen, damit passen auf eine Karteikarte der genannten Größe 18 Zeilen Text. Sie können die Höhe der Karteikarte also auf 18 × *Font.Height* Pixel einstellen. Die Breite stellen Sie auf 5/3 dieses Werts ein.

Die Festlegung der Karteikartenbreite auf 5/3 der Höhe setzt voraus, dass die horizontale Auflösung Ihres Ausgabegeräts – die Anzahl der Pixel, die einer bestimmten Maßeinheit wie z.B. einem Zoll entsprechen – der vertikalen Auflösung entspricht. Wenn bei einem Gerät zur Grafikausgabe die horizontale und vertikale Auflösung übereinstimmt, spricht man auch von *quadratischen Pixeln.* Bei der Einführung der ersten Windows-Version 1985 verfügten die meisten Bildschirme *nicht* über diesen Standard; erst ab 1987 waren die quadratischen Pixel mit Einführung der VGA-Grafikkarten (Video Graphics Array) von IBM ein Standard bei PC-kompatiblen Grafikadaptern.

Heutzutage können Sie eigentlich fast immer davon ausgehen, dass der verwendete Bildschirm für Ihr Windows Forms-Programm über dieses Feature verfügt. Ich sage jedoch »fast immer«, da Windows keine quadratischen Pixel voraussetzt, und es ist immer möglich, dass irgendjemand einen Windows-Gerätetreiber für eine spezielle Grafikkarte schreibt, die keine quadratischen Pixel verwendet.

Drucker hingegen weisen heutzutage häufig *keine* quadratischen Pixel auf. Oft unterstützen Drucker verschiedene Auflösungen und die maximale Auflösung ist öfter in der einen Richtung doppelt so hoch wie in der anderen.

Und wie viel macht das dann?

Sehen wir uns die Beziehung zwischen Pixeln und anderen Maßeinheiten einmal genauer an. Angenommen, Sie zeichnen ein Rechteck mit einer Breite und Höhe von 100 Pixeln, gemessen von der oberen linken Ecke des Clientbereichs (oder dem bedruckbaren Bereich des Blatts).

HundredPixelsSquare.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class HundredPixelsSquare
    Inherits PrintableForm
```

```
    Shared Shadows Sub Main()
        Application.Run(New HundredPixelsSquare())
    End Sub
    Sub New()
        Text = "Hundred Pixels Square"
    End Sub
    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        grfx.FillRectangle(New SolidBrush(clr), 100, 100, 100, 100)
    End Sub
End Class
```

Wie groß ist dieses Rechteck auf dem Bildschirm? Und wie groß ist es auf dem Drucker? Ist es überhaupt quadratisch? Bestimmt haben Sie eine grobe Vorstellung davon, wie groß dieses Rechteck auf dem Bildschirm ist, ohne das Programm überhaupt auszuführen – zumindest im Hinblick auf das Größenverhältnis zwischen Rechteck und Bildschirmgröße. Die kleinsten herkömmlichen Bildschirme zeigen 640 × 480 Pixel an. Auf einem solchen Monitor würde das Rechteck etwa 1/6 der Bildschirmbreite und 1/5 der Bildschirmhöhe einnehmen. Moderne Monitore verfügen jedoch über bis zu 2048 × 1536 Pixel, in diesem Fall wäre das Rechteck im Verhältnis zum Gesamtbildschirm sehr viel kleiner.

Es wäre schön, die Auflösung des Bildschirms zu kennen, vielleicht in einer gängigeren Maßeinheit wie z.B. in Punkten pro Zoll (Dots per Inch, dpi). Obwohl dieser Wert jedoch für Drucker sehr genau definiert ist (er wird in der Regel schon auf der Verpackung angegeben), ist er bei Bildschirmen nur sehr schwer anzugeben. Wenn Sie darüber nachdenken, basiert die tatsächliche DPI-Auflösung eines Bildschirms auf zwei Größen: der Größe des Monitors (die Bildschirmdiagonale in Zoll) und den angezeigten Pixeln.

Verwirrenderweise werden die Pixelangaben häufig als *Bildschirmauflösung* bezeichnet. Ich ziehe jedoch den Begriff *Bildpunkte* vor.

Moderne Grafikkarten unterstützen in der Regel mehr als ein halbes Dutzend unterschiedliche Bildpunkteinstellungen, und Monitore sind in verschiedenen Größen erhältlich. Die nachfolgende Tabelle zeigt die ungefähre Auflösung in dpi für verschiedene Monitorgrößen und Bildpunkteinstellungen:

Tatsächliche Auflösung von Bildschirmen in dpi

	Monitorgröße (Diagonale)			
Bildpunkte	**15 Zoll**	**17 Zoll**	**19 Zoll**	**21 Zoll**
640 × 480	57	50	44	40
800 × 600	71	63	56	50
1024 × 768	91	80	71	64
1152 × 870	103/104	90/91	80/81	72/73
1280 × 1024	114/122	100/107	89/95	80/85
1600 × 1200	143	125	111	100
2048 × 1536	183	160	142	128

Ich setze voraus, dass der tatsächliche Anzeigebereich ein Zoll unter der angegebenen Diagonale liegt und der Monitor ein Standardseitenverhältnis von 4:3 aufweist. Ein 21"-Monitor verfügt beispielsweise über einen Anzeigebereich mit einer Bildschirmdiagonalen von 20 Zoll,

was (mit Dank an Herrn Pythagoras) einer horizontalen Abmessung von 16 Zoll und einer vertikalen Abmessung von 12 Zoll entspricht. Bei 1152 × 870 und 1280 × 1024 Bildpunkten weisen horizontale und vertikale Abmessungen kein Verhältnis von 4:3 auf, daher stimmen horizontale und vertikale Auflösung nicht überein – die Abweichung ist jedoch so minimal, dass sie vernachlässigt werden kann.

Wenn wir also auf einem 21"-Monitor den Videomodus 1600 × 1200 wählen, wäre das 100 × 100 Pixel große Rechteck 1 × 1 Zoll groß. Es könnte je nach Einstellung und Monitor jedoch auch 1/2 Zoll oder größer als 2 Zoll sein. Natürlich konfigurieren nur wenige Benutzer einen 21"-Monitor mit 640 × 480 Bildpunkten oder versuchen, auf einem 15"-Monitor einen Videomodus von 2048 × 1536 einzustellen. Die wahrscheinlicheren Auflösungsbereiche sind in der Tabelle die Werte, die diagonal von oben links nach unten rechts aufgeführt sind.

Windows hat gewöhnlich keine Ahnung, wie groß Ihr Monitor ist, und kann die tatsächliche Bildschirmauflösung daher nicht kennen. Und selbst wenn Windows die Größe des Monitors kennen würde: Was geschähe wohl, wenn Sie einen Videoprojektor mit einem 2 m breiten Bildschirm an Ihren Rechner anschließen? Was *sollte* geschehen? Sollte Windows eine niedrigere Auflösung verwenden, weil der Bildschirm größer ist? Das wäre wohl kaum in Ihrem Sinn.

Das Hauptziel bei der Bildschirmanzeige besteht darin, lesbaren Text zu produzieren. Die Standardschrift sollte natürlich so groß sein, dass sie lesbar ist, aber auch nicht viel größer, damit möglichst viel Text auf den Bildschirm passt.

Aus diesem Grund ignoriert Windows im Grunde Monitorgröße und Bildpunkteinstellung und delegiert die Auswahl der Auflösung an eine Very Important Person: Sie!

Ich habe bereits das Dialogfeld *Eigenschaften von Anzeige* erwähnt. Auf der Registerkarte *Einstellungen* können Sie die Einstellungen für die Bildschirmanzeige ändern. (Beachten Sie bitte, dass die Beschreibung dieser Anzeigeeinstellungen auf Windows XP beruht. Bei anderen Windows-Versionen können diese Angaben leicht abweichen.) Auf der Registerkarte *Einstellungen* finden Sie außerdem die Schaltfläche *Erweitert*. (In früheren Windows-Versionen war diese Einstellung etwas umständlich.) Wenn Sie auf diese Schaltfläche klicken, wird ein weiteres Eigenschaftenfenster geöffnet, in dem Sie indirekt eine Auflösung in dpi einstellen. Ich sage »indirekt«, da Sie tatsächlich einen für Ihre Augen angenehmen Schriftgrad für Windows-Systemschriften festlegen. Diese Systemschrift weist eine Größe von 10 Punkt auf. (Schriften werden in Punkt gemessen, wobei ein Punkt 1/72 Zoll entspricht.) Die Pixelgröße der ausgewählten 10-Punkt-Schrift legt implizit eine Bildschirmauflösung in dpi fest. Das Ergebnis ist exakt dasselbe, wie wenn Sie direkt eine DPI-Auflösung angeben würden.

Die Standardeinstellung heißt beispielsweise *Normalgröße* und entspricht 96 dpi. (In früheren Versionen hieß diese Einstellung *Kleine Schriftarten*.) Die Zeichen dieser Normalgröße weisen eine Höhe von 13 Pixeln auf. Wenn angenommen wird, dass es sich um eine 10-Punkt-Schrift handelt, entsprechen diese 13 Pixel einer Höhe von 10/72 Zoll, damit beträgt die Bildschirmauflösung (leicht gerundet) tatsächlich 96 dpi.

Anstelle von *Normalgröße* kann auch die Einstellung *Groß* gewählt werden (die in früheren Windows-Versionen *Große Schriftarten* hieß), in der die Zeichen eine Höhe von 16 Pixeln aufweisen. Wenn diese 16 Pixel einer Höhe von 10/72 entsprechen, führt dies zu einer Bildschirmauflösung von 120 dpi (ebenfalls leicht gerundet).

Übrigens, die Windows-Systemschrift ist *nicht* die Standardschrift, die über die *Font*-Eigenschaft eines Windows Forms-Programms festgelegt wird. In Windows Forms wird die Standardschrift etwas kleiner gewählt, etwa 8 Punkt.*

* Erfahrene Windows-Programmierer werden jetzt natürlich wissen wollen, woher die genannten Zahlen stammen. Ich beziehe mich auf das *TextMetric*-Feld *tmHeight* (es weist für *Normalgröße* den Wert 16, für

Neben *Normalgröße* und *Groß* stehen jedoch noch weitere Optionen zur Auswahl. Sie können auch eine individuelle Einstellung vornehmen. Es wird dann ein Lineal angezeigt, mit dessen Hilfe Sie die Schriftgröße manuell anpassen können. Die Palette reicht hierbei von sehr großen Schriften (mit einer Auflösung von 480 dpi) bis zu sehr kleinen Schriften (mit etwa 19 dpi).

Üblicherweise ist die gewählte Systemschrift tatsächlich größer als die Punktgröße vermuten lässt. Beim Lesen von ausgedruckten Textvorlagen beträgt der Abstand zum Auge im Allgemeinen etwa 30 cm, der Abstand zwischen Auge und Bildschirm beträgt dagegen häufig das Doppelte.

Punkte pro Zoll (dpi)

Das *Graphics*-Objekt verfügt über zwei Eigenschaften, mit der die Auflösung der Grafikausgabe in Punkten pro Zoll (Dots per Inch, dpi) angegeben wird:

Graphics-Eigenschaften (Auswahl)

Eigenschaft	Typ	Zugriff	Beschreibung
DpiX	*Single*	Get	Horizontale Auflösung in dpi
DpiY	*Single*	Get	Vertikale Auflösung in dpi

Hier ein kleines Programm zur Anzeige dieser Werte ohne viel Schnickschnack:

DotsPerInch.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class DotsPerInch
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New DotsPerInch())
    End Sub

    Sub New()
        Text = "Dots Per Inch"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        grfx.DrawString(String.Format("DpiX = {0}" & vbLf & "DpiY = {1}",
                        grfx.DpiX, grfx.DpiY), Font, New SolidBrush(clr), 0, 0)
    End Sub
End Class
```

Groß den Wert 20 auf) minus *tmInternalLeading* (respektive 3 und 4). Der *tmHeight*-Wert eignet sich für Zeilenabstände; *tmHeight* minus *tmInternalLeading* gibt die Punktgröße in Pixeln an (13 für *Normalgröße*, 16 für *Groß*). Verwirrenderweise weist die Standardschrift in Windows Forms eine *Font.Height*-Eigenschaft auf, die ähnliche Werte liefert: 13 für *Normalgröße*, 15 für *Groß*. Hierbei handelt es sich jedoch um einen Zeilenabstandswert, der mit *tmHeight* vergleichbar ist. Die Windows-Systemschrift ist eine 10-Punkt-Schrift; die Standardschrift in Windows Forms ist ungefähr 8 Punkt groß.

Die Werte, die dieses Programm im Clientbereich anzeigt, stimmen mit denen überein, die Sie im Dialogfeld *Eigenschaften von Anzeige* festgelegt haben: 96 dpi, wenn Sie die Einstellung *Normalgröße* gewählt haben, 120 dpi, wenn Sie sich für die Option *Groß* entschieden haben, oder einen anderen Wert, wenn Sie eine benutzerdefinierte Schriftgröße gewählt haben.

Wenn Sie auf den Clientbereich klicken, zeigt die Druckversion die Auflösung für Ihren Drucker, die Sie allerdings wahrscheinlich schon kennen oder auch in der Druckerdokumentation nachgeschlagen können. Moderne Drucker weisen in der Regel Auflösungen von 300, 600, 1200, 2400, 720, 1440 oder 2880 dpi auf.

Und was ist mit dem Drucker?

Weiter oben in diesem Kapitel habe ich das Programm HundredPixelsSquare vorgestellt, mit dem ein 100 × 100 Pixel großes Rechteck angezeigt wird. Dies warf die Frage auf, wie groß dieses Rechteck auf dem Bildschirm angezeigt würde. Die richtige Antwort auf diese Frage ist, dass die tatsächlichen Abmessungen der angezeigten Fläche irrelevant sind. Schließlich bringt es gar nichts, wenn Sie ein Lineal an Ihren Monitor halten, um die Größe eines Bilds zu ermitteln. Der wichtige Punkt ist der, dass *auf* dem Bildschirm angezeigte Lineale übereinstimmen. In dieser Hinsicht lautet die horizontale und vertikale Abmessung eines 100 Pixel großen Rechtecks folgendermaßen:

```
100 / grfx.DpiX
100 / grfx.DpiY
```

Dies entspricht 1,04 Zoll, wenn Sie die Einstellung *Normalgröße* gewählt haben, 0,83 Zoll, wenn Sie sich für die Option *Groß* entschieden haben, oder einem anderen Wert, wenn Sie eine benutzerdefinierte Schriftgröße gewählt haben.

Und auf dem Drucker … Aber vielleicht möchten Sie das selbst herausfinden. Auf dem Drucker führt das Programm HundredPixelsSquare zur Ausgabe eines Rechtecks mit einer Größe von exakt 1 × 1 Zoll. Probieren wir etwas anderes aus. Das folgende Programm versucht, basierend auf den Eigenschaften *DpiX* und *DpiY* des *Graphics*-Objekts eine Ellipse mit einem Durchmesser von 1 Zoll zu zeichnen.

TryOneInchEllipse.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class TryOneInchEllipse
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New TryOneInchEllipse())
    End Sub

    Sub New()
        Text = "Try One-Inch Ellipse"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        grfx.DrawEllipse(New Pen(clr), 0, 0, grfx.DpiX, grfx.DpiY)
    End Sub
End Class
```

Auf dem Bildschirm scheint die Größe der Ellipse in etwa zu stimmen. Auf meinem 600 dpi-Drucker hat die Ellipse allerdings einen Durchmesser von 6 Zoll.

Die an die *Graphics*-Zeichenfunktion übergebenen Koordinaten werden für den Bildschirm offensichtlich in Pixeln angegeben. Für den Drucker scheint dies jedoch nicht zuzutreffen. Tatsächlich werden die an die *Graphics*-Zeichenfunktion übergebenen Koordinatenwerte für die Druckausgabe unabhängig vom Drucker in Einheiten von 0,01 Zoll interpretiert. In Kürze werde ich erklären, wie dies zustande kommt. Das Schöne ist, dass die Bildschirmauflösung wahrscheinlich im Bereich um 100 dpi liegt, und der Drucker wie ein 100-dpi-Gerät behandelt wird. Kurz gesagt: Sie können dieselben Koordinaten für die Ausgabe von Grafiken auf dem Bildschirm und dem Drucker verwenden und erhalten in etwa dieselben Ergebnisse.

Manuelle Konvertierung

Wenn Sie möchten, können Sie die Eigenschaften *DpiX* und *DpiY* des *Graphics*-Objekts zur Anpassung der an die Zeichenfunktion übergebenen Koordinaten einsetzen. Angenommen, Sie möchten Gleitkommakoordinaten verwenden, um in Millimetern zu zeichnen. Sie benötigen in diesem Fall eine Methode, die Millimeter in Pixel konvertiert:

```
Function MMConv(ByVal grfx As Graphics, ByVal ptf As PointF) As PointF
    ptf.X *= grfx.DpiX / 25.4F
    ptf.Y *= grfx.DpiY / 25.4F
    Return ptf
End Function
```

Der an diese Methode übergebene Punkt gibt die gewünschten Millimeter an. Wenn Sie diesen Wert durch 25,4 teilen, erhalten Sie den Wert in Zoll. (Dies ist übrigens eine exakte Berechnung.) Durch Multiplikation dieses Werts mit der Auflösung in dpi erhalten Sie die Pixelmaße.

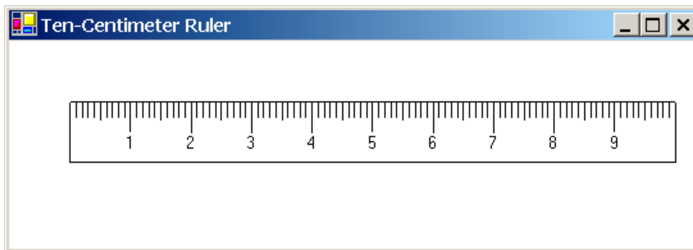
Lassen Sie uns mit diesem Wissen ein 10 cm langes Lineal zeichnen.

TenCentimeterRuler.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class TenCentimeterRuler
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New TenCentimeterRuler())
    End Sub

    Sub New()
        Text = "Ten-Centimeter Ruler"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Const xOffset As Integer = 10
        Const yOffset As Integer = 10
        Dim i As Integer
        Dim pn As New Pen(clr)
        Dim br As New SolidBrush(clr)
        Dim strfmt As New StringFormat()
```

```
        grfx.DrawPolygon(pn, New PointF() _
            { _
                MMConv(grfx, New PointF(xOffset, yOffset)), _
                MMConv(grfx, New PointF(xOffset + 100, yOffset)), _
                MMConv(grfx, New PointF(xOffset + 100, yOffset + 10)), _
                MMConv(grfx, New PointF(xOffset, yOffset + 10)) _
            })
        strfmt.Alignment = StringAlignment.Center

        For i = 1 To 99
            If i Mod 10 = 0 Then
                grfx.DrawLine(pn, _
                    MMConv(grfx, New PointF(xOffset + i, yOffset)), _
                    MMConv(grfx, New PointF(xOffset + i, yOffset + 5)))
                grfx.DrawString((i / 10).ToString(), Font, br, _
                    MMConv(grfx, New PointF(xOffset + i, yOffset + 5)), strfmt)
            ElseIf i Mod 5 = 0 Then
                grfx.DrawLine(pn, _
                    MMConv(grfx, New PointF(xOffset + i, yOffset)), _
                    MMConv(grfx, New PointF(xOffset + i, yOffset + 3)))
            Else
                grfx.DrawLine(pn, _
                  MMConv(grfx, New PointF(xOffset + i, yOffset)), _
                  MMConv(grfx, New PointF(xOffset + i, yOffset + 2.5F)))
            End If
        Next i
    End Sub

    Private Function MMConv(ByVal grfx As Graphics, ByVal ptf As PointF) As PointF
        ptf.X *= grfx.DpiX / 25.4F
        ptf.Y *= grfx.DpiY / 25.4F
        Return ptf
    End Function
End Class
```

Auf dem Bildschirm sieht das Lineal so aus:



Dieses Diagramm enthält auch Text. Woher habe ich gewusst, dass der richtig angezeigt wird? Ich wusste, dass die *Font*-Eigenschaft eine 8-Punkt-Schrift verwendet, deshalb konnte ich davon ausgehen, dass die Schriftzeichen etwa 3 Millimeter hoch sind und damit ungefähr die richtige Größe aufweisen.

Ich habe die Klasse *TenCentimerRuler* von *PrintableForm* abgeleitet, um Folgendes noch einmal ganz deutlich zu machen: Diese Technik funktioniert nicht auf dem Drucker. Mein 600-dpi-Drucker zeigt das Lineal in sechsfacher Vergrößerung an.

Seiteneinheiten und Seitenskalierung

Damit Sie Methoden wie z.B. *MMConv* nicht selbst schreiben müssen, enthält GDI+ ein Feature zur automatischen Skalierung auf die Maße Ihrer Wahl. Grundsätzlich werden die an die *Graphics*-Zeichenfunktion übergebenen Koordinaten mithilfe von Konstanten skaliert, genau wie in der Methode *MMConv.* Sie stellen diese Skalierungsfaktoren jedoch nicht direkt ein. Stattdessen legen Sie diese mithilfe der Eigenschaften *PageUnit* und *PageScale* der *Graphics*-Klasse indirekt fest.

Graphics-Eigenschaften (Auswahl)

Eigenschaft	Typ	Zugriff
PageUnit	*GraphicsUnit*	Get/Set
PageScale	*Single*	Get/Set

Sie stellen die *PageUnit*-Eigenschaft auf einen Wert der Enumeration *GraphicsUnit* ein:

GraphicsUnit-Enumeration

Member	Wert	Beschreibung
World	0	Kann nicht mit *PageUnit* verwendet werden
Display	1	Entspricht für den Bildschirm *Pixel;* 1/100 Zoll für Drucker (Standardwert für Drucker)
Pixel	2	Einheiten in Pixel (Standardwert für den Bildschirm)
Point	3	Einheiten von 1/72 Zoll
Inch	4	Einheiten in Zoll
Document	5	Einheiten von 1/300 Zoll
Millimeter	6	Einheiten in Millimeter

Wenn Sie beispielsweise in Einheiten von 1/100 Zoll arbeiten möchten, können Sie die beiden Eigenschaften folgendermaßen einstellen:

```
grfx.PageUnit = GraphicsUnit.Inch
grfx.PageScale = 0.01
```

Wenn Sie als Koordinate den Wert 1 angeben, entspricht dies dem Wert 0,01". Im Anschluss an diese Aufrufe zeichnet der folgende Aufruf der *DrawLine*-Methode eine 1 Zoll lange Linie:

```
grfx.DrawLine(pn, 0, 0, 100, 0)
```

Dies entspricht exakt 1 Zoll auf dem Drucker und *grfx.DpiX* Pixeln auf dem Bildschirm. Die gleichen Ergebnisse erhalten Sie mit

```
grfx.PageUnit = GraphicsUnit.Document
grfx.PageScale = 3
```

oder

```
grfx.PageUnit = GraphicsUnit.Millimeter
grfx.PageScale = 0.254
```

oder

```
grfx.PageUnit = GraphicsUnit.Point
grfx.PageScale = 0.72
```

Als Standardeinstellungen werden für den Bildschirm *GraphicsUnit.Pixel* und für den Drucker *GraphicsUnit.Display* verwendet, in beiden Fällen erhält *PageScale* den Wert 1. Beachten

Sie, dass der Wert *GraphicsUnit.Display* für Bildschirm und Drucker eine unterschiedliche Bedeutung hat. Beim Bildschirm entspricht dieser Wert *GraphicsUnit.Pixel*, beim Drucker gibt *GraphicsUnit.Display* jedoch Einheiten von 1/100 Zoll an. (In der Dokumentation für die Enumeration *GraphicsUnit* wird behauptet, dass mit *GraphicsUnit.Display* eine Einheit von 1/75 Zoll eingestellt wird; es ist leicht nachzuweisen, dass das falsch ist.)

Wenn also das Programm TenCentimeterRuler auch auf dem Drucker funktionieren soll, müssen wir *PageUnit* lediglich auf *GraphicsUnit.Pixel* setzen, dann sollte eigentlich alles in Ordnung sein. Prüfen wir das, indem wir eine von *TenCentimeterRuler* abgeleitete Klasse definieren. Die neue *OnPage*-Methode setzt die Eigenschaft *PageUnit* zurück und ruft anschließend die *DoPage*-Methode der Basisklasse auf.

PrintableTenCentimeterRuler.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class PrintableTenCentimeterRuler
    Inherits TenCentimeterRuler

    Shared Shadows Sub Main()
        Application.Run(New PrintableTenCentimeterRuler())
    End Sub

    Sub New()
        Text = "Printable " & Text
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
                ByVal clr As Color, ByVal cx As Integer, _
                ByVal cy As Integer)
        grfx.PageUnit = GraphicsUnit.Pixel
        MyBase.DoPage(grfx, clr, cx, cy)
    End Sub
End Class
```

Dieses Programm macht keinen Gebrauch von den Argumenten *cx* und *cy* von *DoPage*. Diese Abmessungen – Clientbereich und bedruckbarer Bereich der Druckseite – werden in Einheiten angegeben, die zu dem Standardwert von *PageUnit* kompatibel sind. Im Allgemeinen müssen Sie bei einer Änderung von *PageUnit* wahrscheinlich die Größe für das Ausgabegerät in dieselben Einheiten umrechnen. Auf dieses Problem komme ich gleich noch zurück.

Obwohl die Druckerausgabe jetzt in Pixeln erfolgt, sieht die Schrift weiterhin gut aus. Die über die Eigenschaft *Font* zugängliche Formularschrift wird sowohl auf dem Bildschirm als auch auf dem Drucker als 8-Punkt-Schrift ausgegeben. In Kapitel 9 werden Sie sehen, wie das funktioniert.

Bezüglich des Zeichenstifts für die TenCentimeterRuler-Version von *DoPage* ergibt sich jedoch weiterhin ein Problem:

```
Dim pn As New Pen(clr)
```

Dieser Stift hat standardmäßig eine Breite von 1. Auf dem Bildschirm entspricht dies einer Breite von 1 Pixel. Auf dem Drucker wird diese Breite in 1/100 Zoll umgerechnet. Wenn Sie *PageUnit* in *GraphicsUnit.Pixel* ändern, wird der 1 Einheit breite Zeichenstift anschließend als 1 Pixel breit interpretiert. Auf einem hochauflösenden Drucker ist das Lineal in diesem Fall kaum sichtbar.

Statt weiter am ursprünglichen Programm TenCentimeterRuler herumzubasteln, nutzen wir die Vorteile der Eigenschaften *PageUnit* und *PageScale* und machen die manuelle Konvertierung überflüssig.

TenCentimeterRulerAuto.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms
Class TenCentimeterRulerAuto
    Inherits PrintableForm
    Shared Shadows Sub Main()
        Application.Run(New TenCentimeterRulerAuto())
    End Sub
    Sub New()
        Text = "Ten-Centimeter Ruler (Auto)"
    End Sub
    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Const xOffset As Integer = 10
        Const yOffset As Integer = 10
        Dim i As Integer
        Dim pn As New Pen(clr, 0.25)
        Dim br As New SolidBrush(clr)
        Dim strfmt As New StringFormat()
        grfx.PageUnit = GraphicsUnit.Millimeter
        grfx.PageScale = 1
        grfx.DrawRectangle(pn, xOffset, yOffset, 100, 10)
        strfmt.Alignment = StringAlignment.Center
        For i = 1 To 99
            If i Mod 10 = 0 Then
                grfx.DrawLine(pn, _
                            New PointF(xOffset + i, yOffset), _
                            New PointF(xOffset + i, yOffset + 5))
                grfx.DrawString((i / 10).ToString(), Font, br, _
                             New PointF(xOffset + i, yOffset + 5), strfmt)
            ElseIf i Mod 5 = 0 Then
                grfx.DrawLine(pn, _
                            New PointF(xOffset + i, yOffset), _
                            New PointF(xOffset + i, yOffset + 3))
            Else
                grfx.DrawLine(pn, _
                            New PointF(xOffset + i, yOffset), _
                            New PointF(xOffset + i, yOffset + 2.5F))
            End If
        Next i
    End Sub
End Class
```

Außer der Entfernung der Methode *MMConv* habe ich nur einige wenige Änderungen vorgenommen. Meine *MMConv*-Methode funktionierte nur mit *PointF*-Strukturen, daher habe ich in den ersten Linealprogrammen *DrawPolygon* anstelle von *DrawRectangle* verwendet. Da GDI+ Koordinaten und Größen auf die gleiche Weise skaliert, kann hier *DrawRectangle* verwendet

werden. Eine weitere Änderung findet sich zu Beginn der *DoPage*-Methode, an der Stelle, an der das Programm einen 0,25 Einheiten breiten Zeichenstift erstellt:

```
Dim pn As New Pen(clr, 0.25)
```

Ferner stellt das Programm das *Graphics*-Objekt so ein, dass in Millimetern gezeichnet wird:

```
grfx.PageUnit = GraphicsUnit.Millimeter
grfx.PageScale = 1
```

Sie fragen sich vielleicht, ob es einen Unterschied macht, die Eigenschaften *PageUnit* und *PageScale* vor oder nach der Zeichenstifterstellung einzustellen. Es macht keinen Unterschied, denn *Pen*-Objekte sind geräteunabhängig. Sie sind vor dem Aufruf einer der Methoden zum Zeichnen von Linien nicht mit einem bestimmten *Graphics*-Objekt verknüpft. Erst zu diesem Zeitpunkt wird die Zeichenstiftbreite in den über die Eigenschaften *PageUnit* und *PageScale* angegebenen Einheiten interpretiert. In diesem Fall wird der Zeichenstift als 0,25 Millimeter oder 1/100 Zoll breit ausgewertet. Wenn Sie den Unterschied auf dem Drucker sehen möchten, sollten Sie es mit einem kleineren Wert versuchen (z.B. mit 0,10 Millimeter).

Wenn Sie im *Pen*-Konstruktor keine Breite definieren, wird der Zeichenstift mit einer Breite von 1 Einheit erstellt. In diesem Fall bedeutet dies, dass der Zeichenstift ganze 1 mm breit ist und die Markierungen des Lineals zu einem einzigen Klumpen ineinander laufen. (Probieren Sie es aus!)

Zeichenstiftbreiten

Welche Zeichenstiftbreiten sind für den Drucker geeignet? Sie können sich hier an PostScript orientieren, der sehr bekannten und hoch geachteten Seitenbeschreibungssprache, von der viele High-End-Drucker Gebrauch machen, und sich eine Standardstiftbreite als 1 Punkt bzw. 1/72 Zoll oder 1/3 Millimeter vorstellen. Ich persönlich finde eine Stiftbreite von 1 Punkt zwar etwas klotzig, aber dieser Wert ist leicht zu merken.

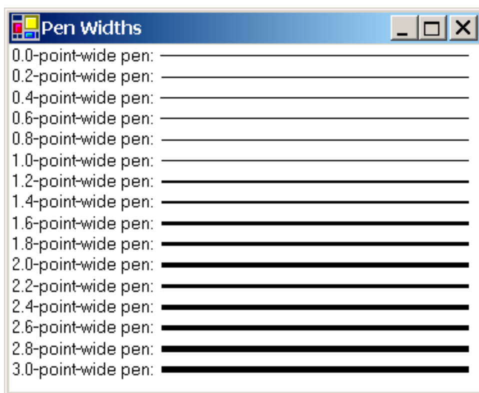
Hier ein Programm, das eine ganze Sammlung von Stiftbreiten in der Einheit Punkt liefert.

PenWidths.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class PenWidths
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New PenWidths())
    End Sub

    Sub New()
        Text = "Pen Widths"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Dim br As New SolidBrush(clr)
        Dim y As Single = 0
        Dim f As Single

        grfx.PageUnit = GraphicsUnit.Point
        grfx.PageScale = 1
```

```
        For f = 0 To 3.1 Step 0.2
            Dim pn As New Pen(clr, f)
            Dim str As String = _
                String.Format("{0:F1}-point-wide pen: ", f)
            Dim szf As SizeF = grfx.MeasureString(str, Font)
            grfx.DrawString(str, Font, br, 0, y)
            grfx.DrawLine(pn, szf.Width, y + szf.Height / 2, szf.Width + 144, y + szf.Height / 2)
            y += szf.Height
        Next f
    End Sub
End Class
```

Sie können die Stiftbreiten zwar auch in Bruchteilen von Pixeln angeben, die Ausgabe kann aber natürlich nur in ganzen Pixelbreiten erfolgen. Auf dem Bildschirm werden viele der von diesem Programm erzeugten Stiftbreiten auf denselben Wert gerundet:



Sie brauchen sich allerdings keine Sorgen zu machen, dass die Stiftbreite auf 0 abgerundet wird und der Stift nicht mehr auf dem Bildschirm angezeigt wird. Stifte werden immer mit einer Mindestbreite von 1 Pixel gezeichnet. Tatsächlich können Sie die Breite im *Pen*-Konstruktor sogar auf 0 stellen und erhalten immer noch 1 Pixel breite Linien, unabhängig von *PageUnit* und *PageScale*.

Obwohl Stifte mit einer Breite von 0 für den Bildschirm kein Problem darstellen, sollten sie auf dem Drucker nicht eingesetzt werden. Auf einem hochauflösenden Laserdrucker sind 1 Pixel breite Linien praktisch nicht sichtbar.

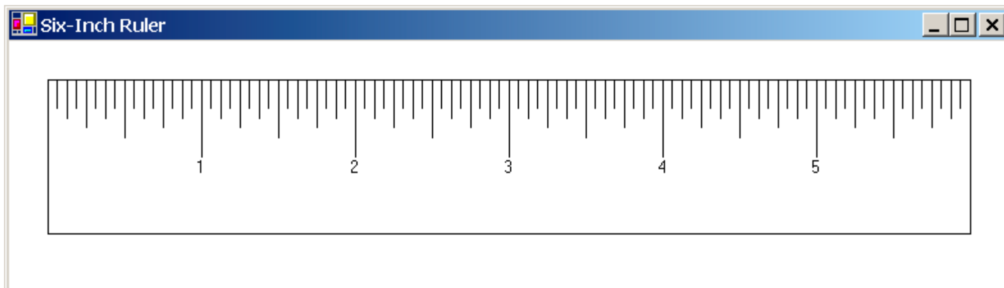
Das nachfolgende Programm zeichnet ein Lineal mit Zollmarkierungen in Einheiten von 1/64 Zoll und mit einer Stiftbreite von 1/128 Zoll.

SixInchRuler.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class SixInchRuler
    Inherits PrintableForm
```

```
    Shared Shadows Sub Main()
        Application.Run(New SixInchRuler())
    End Sub

    Sub New()
        Text = "Six-Inch Ruler"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Const xOffset As Integer = 16
        Const yOffset As Integer = 16
        Dim i As Integer
        Dim pn As New Pen(clr, 0.5)
        Dim br As New SolidBrush(clr)
        Dim strfmt As New StringFormat()

        grfx.PageUnit = GraphicsUnit.Inch
        grfx.PageScale = 1 / 64
        grfx.DrawRectangle(pn, xOffset, yOffset, 6 * 64, 64)
        strfmt.Alignment = StringAlignment.Center

        For i = 1 To 95
            Dim x As Integer = xOffset + i * 4
            Dim y As Integer = yOffset
            Dim dy As Integer

            If i Mod 16 = 0 Then
                dy = 32
                grfx.DrawString((i / 16).ToString(), Font, br, x, y + dy, strfmt)
            ElseIf i Mod 8 = 0 Then
                dy = 24
            ElseIf i Mod 4 = 0 Then
                dy = 20
            ElseIf i Mod 2 = 0 Then
                dy = 16
            Else
                dy = 12
            End If
            grfx.DrawLine(pn, x, y, x, y + dy)
        Next i
    End Sub
End Class
```

Das Lineal sieht folgendermaßen aus:



Sie haben vielleicht bemerkt, dass der Text offenbar von den eingestellten Werten von *PageUnit* und *PageScale* unberührt bleibt. Das liegt daran, dass die über die Formulareigenschaft *Form* zugängliche Schrift als 8-Punkt-Schrift definiert ist und ihre Größe beibehält. In Kapitel 9 werde ich aufzeigen, wie eine Schrift erzeugt wird, deren Größe von den Eigenschaften *PageUnit* und *PageScale* beeinflusst wird.

Seitentransformationen

Wenn Sie die Eigenschaften *PageScale* und *PageUnit* einstellen, legen Sie die so genannte *Seitentransformation* fest. Diese Transformation kann durch einige einfache Formeln dargestellt werden. Angenommen, die an die *Graphics*-Zeichenmethoden übergebenen Koordinaten sind *Seitenkoordinaten.* (Diese Annahme trifft, wie Sie später sehen werden, eigentlich nur dann zu, wenn Sie lediglich die Eigenschaften *PageScale* und *PageUnit* einstellen.) Ein Punkt kann in Seiteneinheiten als (x_{page}, y_{page}) ausgedrückt werden.

Die Pixelkoordinaten relativ zur oberen linken Ecke des Clientbereichs (oder der oberen linken Ecke des bedruckbaren Seitenbereichs) sind die so genannten *Gerätekoordinaten* oder (x_{device}, y_{device}). Die Seitentransformation richtet sich nach den Eigenschaften *PageUnit, PageScale, DpiX* und *DpiY.*

Formeln für die Seitentransformation

***PageUnit*-Wert**	**Transformationsformeln**
GraphicsUnit.Pixel	$x_{device} = x_{page} \times PageScale$ $y_{device} = y_{page} \times PageScale$
GraphicsUnit.Display (Bildschirm)	$x_{device} = x_{page} \times PageScale$ $y_{device} = y_{page} \times PageScale$
GraphicsUnit.Display (Drucker)	$x_{device} = x_{page} \times PageScale \times DpiX / 100$ $y_{device} = y_{page} \times PageScale \times DpiY / 100$
GraphicsUnit.Inch	$x_{device} = x_{page} \times PageScale \times DpiX$ $y_{device} = y_{page} \times PageScale \times DpiY$
GraphicsUnit.Millimeter	$x_{device} = x_{page} \times PageScale \times DpiX / 25.4$ $y_{device} = y_{page} \times PageScale \times DpiY / 25.4$
GraphicsUnit.Point	$x_{device} = x_{page} \times PageScale \times DpiX / 72$ $y_{device} = y_{page} \times PageScale \times DpiY / 72$
GraphicsUnit.Document	$x_{device} = x_{page} \times PageScale \times DpiX / 300$ $y_{device} = y_{page} \times PageScale \times DpiY / 300$

Allgemein formuliert ergibt dies:

$x_{device} = x_{page} \times PageScale \times DpiX$ / (*GraphicsUnit* Einheiten pro Zoll)

$y_{device} = y_{page} \times PageScale \times DpiY$ / (*GraphicsUnit* Einheiten pro Zoll)

Die Seitentransformation wirkt sich auf alle Koordinaten sämtlicher Zeichenfunktionen der Klasse *Graphics* aus, die bisher besprochen wurden. Ferner wirken sie sich ebenfalls auf die von *MeasureString* zurückgegebenen Informationen sowie auf die in der Klasse *Font* implementierte Version der *GetHeight*-Methode aus, die als Argument ein *Graphics*-Objekt erwartet.

Die Seitentransformation ist ein Merkmal der Klasse *Graphics.* Sie wirkt sich nur auf Member dieser Klasse sowie auf Elemente aus, die (wie z.B. *GetHeight*) als Argument ein *Graphics*-Objekt erwarten. Beispielsweise wirkt sich die Seitentransformation nicht auf die Informationen aus, die über *ClientSize* geliefert werden. *ClientSize*-Werte werden immer in Pixeln angegeben.

Speichern des Grafikstatus

Das Einstellen der Eigenschaften *PageUnit* und *PageScale* des *Graphics*-Objekts hat große Auswirkungen auf die nachfolgende Anzeige von Grafiken. Oft stellen Sie diese Eigenschaften (oder andere Eigenschaften der Klasse *Graphics*) ein, um eine Grafik zu zeichnen oder einige Informationen abzurufen, und wollen sie dann wieder auf die ursprünglichen Werte zurücksetzen.

Die Klasse *Graphics* verfügt zu diesem Zweck über die Methoden *Save* und *Restore,* die genau diese Funktion erfüllen: Sie speichern die Eigenschaften des *Graphics*-Objekts und ermöglichen eine spätere Wiederherstellung dieser Werte. Diese beiden Methoden verwenden die Klasse *GraphicsState* aus dem Namespace *System.Drawing.Drawing2D*.

Graphics-Methoden (Auswahl)

```
Function Save() As GraphicsState
Sub Restore(ByVal gs As GraphicsState)
```

Die Klasse *GraphicsState* weist keine öffentlichen Elemente auf, die irgendwie interessant wären. Sie nutzen sie tatsächlich als Blackbox. Beim Aufruf von

```
Dim gs As GraphicsState = grfx.Save()
```

werden alle aktuellen *Graphics*-Eigenschaften mit Lese-/Schreibzugriff in dem *GraphicsState*-Objekt gespeichert. Anschließend können Sie die Eigenschaften für das *Graphics*-Objekt ändern. Zur Wiederherstellung der gespeicherten Eigenschaften verwenden Sie

```
grfx.Restore(gs)
```

Programmierer mit Win32-Erfahrung kennen dieses Konzept (unter Verwendung der Funktionen *SaveDC* und *RestoreDC*) wahrscheinlich als LIFO-Stack (Last-In/First-Out). Die Windows Forms-Implementierung ist flexibler. Sie könnten bei der Verarbeitung von *OnPaint* beispielsweise drei unterschiedliche *Graphics*-Statuswerte definieren:

```
Dim gs1 As GraphicsState = grfx.Save()

    ' Eigenschaften ändern.
    ⋮

Dim gs2 As GraphicsState = grfx.Save()

    ' Eigenschaften ändern.
    ⋮

Dim gs3 As GraphicsState = grfx.Save()
```

Anschließend können Sie willkürlich und in beliebiger Reihenfolge die Methode *Restore* aufrufen, um eine der drei gespeicherten Statuseinstellungen zu verwenden.

Eine ähnliche Funktionalität bieten die Methoden *BeginContainer* und *EndContainer* der Klasse *Graphics*. Diese Methoden nutzen die Klasse *GraphicsContainer* des Namespaces *System.Drawing.Drawing2D*.

Maße in anderen Einheiten

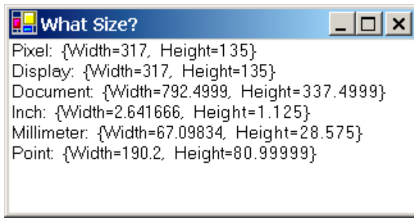
Die Abmessungen des Clientbereichs eines Formulars können über die Eigenschaft *ClientSize* abgerufen werden. Diese Abmessungen werden immer in Pixeln angegeben. Wenn Sie eine neue Seitentransformation einstellen, werden Sie die Abmessungen des Clientbereichs wahrscheinlich nicht in Pixeln angeben wollen, sondern in den Einheiten, die in den Zeichenmethoden verwendet werden.

Es gibt zwei Möglichkeiten, die Clientgröße in anderen Maßeinheiten abzurufen. Die bequemste Methode ist hierbei die Eigenschaft *VisibleClipBounds* des *Graphics*-Objekts. Diese Eigenschaft gibt stets die Abmessungen des Clientbereichs in der Einheit zurück, die mit den Eigenschaften *PageUnit* und *PageScale* eingestellt wurde. Nachfolgend sehen Sie ein Programm, das mithilfe dieser Informationen die Größe des Clientbereichs in sämtlichen möglichen Einheiten anzeigt.

WhatSize.vb

```
Imports System
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Windows.Forms

Class WhatSize
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New WhatSize())
    End Sub

    Sub New()
        Text = "What Size?"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Dim br As New SolidBrush(clr)
        Dim y As Integer = 0
        DoIt(grfx, br, y, GraphicsUnit.Pixel)
        DoIt(grfx, br, y, GraphicsUnit.Display)
        DoIt(grfx, br, y, GraphicsUnit.Document)
        DoIt(grfx, br, y, GraphicsUnit.Inch)
        DoIt(grfx, br, y, GraphicsUnit.Millimeter)
        DoIt(grfx, br, y, GraphicsUnit.Point)
    End Sub

    Private Sub DoIt(ByVal grfx As Graphics, ByVal br As Brush, _
            ByRef y As Integer, ByVal gu As GraphicsUnit)
        Dim gs As GraphicsState = grfx.Save()

        grfx.PageUnit = gu
        grfx.PageScale = 1

        Dim szf As SizeF = grfx.VisibleClipBounds.Size

        grfx.Restore(gs)
        grfx.DrawString(gu.ToString() & ": " & szf.ToString(), Font, br, 0, y)
        y += CInt(Math.Ceiling(Font.GetHeight(grfx)))
    End Sub
End Class
```

Die Methode *DoIt* des Programms WhatSize nutzt *Save* und *Restore*, damit unterschiedliche *PageUnit*-Einstellungen beim Aufruf von *DrawString* und *GetHeight* nicht zu Konflikten mit der Anzeige der Informationen führen. Hier eine typische Anzeige von WhatSize:



Leider verhält es sich beim Drucker etwas anders. Für den Drucker gibt *VisibleClipBounds* unabhängig von der Seitentransformation Werte in 1/100 Zoll zurück. Wenn jedoch der *PageUnit*-Wert für den Drucker in Pixeln angegeben ist, gibt *VisibleClipBounds* den bedruckbaren Seitenbereich in Pixeln zurück. Wenn Sie in den Clientbereich von WhatSize klicken, erhalten Sie eine Bestätigung für dieses anomale Verhalten.

Eine historische Anmerkung: Ich habe 1986 in der Dezemberausgabe des *Microsoft Systems Journal* den allerersten Artikel zur Windows-Programmierung veröffentlicht. Das in diesem Artikel beschriebene Beispielprogramm hieß WSZ, »What Size«, und zeigte die Größe des Clientbereichs in Pixeln, Zoll und Millimetern an. Das hier abgedruckte Programm WhatSize ist eine vereinfachte – und erheblich kürzere – Version dieses Programms.

Der zweite Ansatz zur Bestimmung der Größe des Anzeigebereichs nutzt die Methode *TransformPoints* der Klasse *Graphics:*

TransformPoints-Methoden von *Graphics*

```
Sub TransformPoints(ByVal csDst As CoordinateSpace,
                    ByVal csSrc As CoordinateSpace, ByVal apt As Point())
Sub TransformPoints(ByVal csDst As CoordinateSpace,
                    ByVal csSrc As CoordinateSpace, ByVal aptf As PointF())
```

Die Enumeration *CoordinateSpace* ist im Namespace *System.Drawing.Drawing2D* definiert:

CoordinateSpace-Enumeration

Member	Wert
World	0
Page	1
Device	2

Bisher haben wir den Koordinatenraum *Device* (in Pixeln relativ zur oberen linken Ecke des Clientbereichs) sowie den Koordinatenraum *Page* (in Zoll, Millimeter, Punkten usw.) kennen gelernt. Wenn Sie ein Array aus *Point*- oder *PointF*-Strukturen in Geräteeinheiten verwenden, können Sie diese Werte durch folgenden Aufruf in Seiteneinheiten umwandeln:

```
grfx.TransformPoints(CoordinateSpace.Page, CoordinateSpace.Device, apt)
```

Ich werde in Kürze auf diesen als *World* (Welt) bezeichneten Koordinatenraum zurückkommen.

Hier eine weitere Version des Programms WhatSize, das *TransformPoint* zur Berechnung der Größe des Clientbereichs verwendet.

WhatSizeTransform.vb

```
Imports System
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Windows.Forms

Class WhatSizeTransform
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New WhatSizeTransform())
    End Sub

    Sub New()
        Text = "What Size? With TransformPoints"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Dim br As New SolidBrush(clr)
        Dim y As Integer = 0
        Dim apt() As Point = {New Point(cx, cy)}

        grfx.TransformPoints(CoordinateSpace.Device, CoordinateSpace.Page, apt)

        DoIt(grfx, br, y, apt(0), GraphicsUnit.Pixel)
        DoIt(grfx, br, y, apt(0), GraphicsUnit.Display)
        DoIt(grfx, br, y, apt(0), GraphicsUnit.Document)
        DoIt(grfx, br, y, apt(0), GraphicsUnit.Inch)
        DoIt(grfx, br, y, apt(0), GraphicsUnit.Millimeter)
        DoIt(grfx, br, y, apt(0), GraphicsUnit.Point)
    End Sub

    Private Sub DoIt(ByVal grfx As Graphics, ByVal br As Brush, ByRef y As Integer, _
        ByVal pt As Point, ByVal gu As GraphicsUnit)
        Dim gs As GraphicsState = grfx.Save()

        grfx.PageUnit  = gu
        grfx.PageScale = 1

        Dim aptf() As PointF = {Point.op_Implicit(pt)}
        grfx.TransformPoints(CoordinateSpace.Page, CoordinateSpace.Device, aptf)

        Dim szf As New SizeF(aptf(0))
        grfx.Restore(gs)
        grfx.DrawString(gu.ToString() & ": " & szf.ToString(), Font, br, 0, y)
        y += CInt(Math.Ceiling(Font.GetHeight(grfx)))
    End Sub
End Class
```

Ich habe der Methode *DoIt* ein zusätzliches Argument gegeben: eine *Point*-Struktur, welche die Breite und Höhe des Anzeigebereichs in Pixeln enthält. Beim Bildschirm gibt es kein Problem, da die Argumente *cx* und *cy* von *DoPage* bereits in Pixeln angegeben sind. Für den Drucker trifft dies jedoch nicht zu. Aus diesem Grund baut die *DoPage*-Methode eine *Point*-Struktur aus *cx* und *cy* auf, erstellt ein aus einem Element bestehendes *Point*-Array und übergibt dieses Array an *TransformPoints,* um die Werte in Geräteeinheiten umzuwandeln. Beachten Sie, dass für diesen Aufruf von *TransformPoints* der Zielkoordinatenraum *CoordinateSpace.Device* lautet. *DoIt* verwendet anschließend *TransformPoints,* um die Geräteeinheiten in Seiteneinheiten (*CoordinateSpace.Page*) zu konvertieren.

Frei wählbare Koordinaten

Einige der bisher gezeigten Grafikprogramme haben die Ausgabe auf die Größe des Clientbereichs oder den bedruckbaren Seitenbereich skaliert. Die in diesem Kapitel vorgestellten Programme führten die Zeichenoperationen in festgelegten Einheiten wie Millimetern oder Zoll durch.

Es gibt Situationen, in denen Sie eine Reihe von Koordinaten fest einprogrammieren und auf jede explizite Skalierung der Koordinaten verzichten möchten. Vielleicht möchten Sie beispielsweise die Grafikausgabe unter Verwendung eines Koordinatensystems mit 1000 × 1000 Einheiten programmieren. Dieses Koordinatensystem soll so groß wie möglich sein, jedoch in den Clientbereich bzw. den bedruckbaren Bereich passen.

Das folgende Programm demonstriert, wie Sie das erreichen.

ArbitraryCoordinates.vb

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Class ArbitraryCoordinates
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New ArbitraryCoordinates())
    End Sub

    Sub New()
        Text = "Arbitrary Coordinates"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        grfx.PageUnit = GraphicsUnit.Pixel
        Dim szf As SizeF = grfx.VisibleClipBounds.Size

        grfx.PageUnit = GraphicsUnit.Inch
        grfx.PageScale = Math.Min(szf.Width / grfx.DpiX / 1000, szf.Height / grfx.DpiY / 1000)

        grfx.DrawEllipse(New Pen(clr), 0, 0, 990, 990)
    End Sub
End Class
```

Die Methode *DoPage* stellt zunächst *PageUnit* auf *GraphicsUnit.Pixel* ein. Alleiniger Zweck hierbei ist der Abruf der Eigenschaft *VisibleClipBounds* zur Ermittlung der Größe des Clientbereichs oder der Druckerseite in Pixeln.

Als Nächstes stellt die *DoPage*-Methode *PageUnit* auf Zoll ein:

```
grfx.PageUnit = GraphicsUnit.Inch
```

Ich habe weiter oben bereits die folgenden Transformationsformeln vorgestellt, die bei Verwendung von Zoll für *PageUnit* gelten:

$$x_{device} = x_{page} \times PageScale \times DpiX$$
$$y_{device} = y_{page} \times PageScale \times DpiY$$

Sie möchten, dass x_{page} und y_{page} zwischen 0 und 1000 liegen, während x_{device} und y_{device} zwischen 0 und den Werten der Eigenschaften *Width* bzw. *Height* von *VisibleClipBounds* liegen sollen.

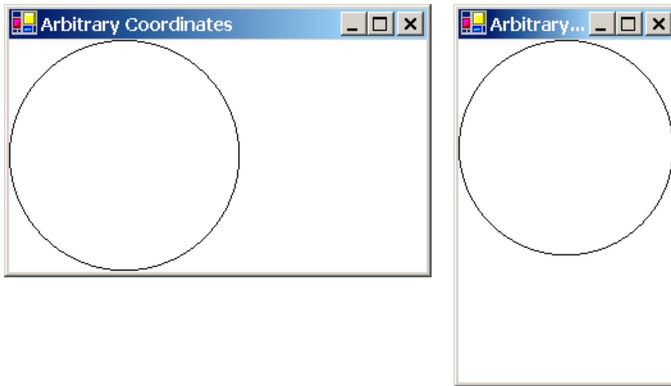
Anders ausgedrückt:

$$Width = 1000 \times PageScale \times DpiX$$
$$Height = 1000 \times PageScale \times DpiY$$

Diese beiden Gleichungen führen allerdings zu zwei unterschiedlichen *PageScale*-Faktoren, und es darf nur einen Faktor geben. Sie benötigen den kleineren der beiden berechneten Werte:

```
grfx.PageScale = Math.Min(szf.Width / grfx.DpiX / 1000, szf.Height / grfx.DpiY / 1000)
```

Das Programm zeichnet in diesem Fall eine Ellipse mit einer Breite und Höhe von 990 Einheiten. (Die Verwendung von 1000 oder 999 für Breite und Höhe führt in einigen Fällen dazu, dass eine Seite der Figur bei großen Fenstern abgeschnitten wird.) Als Ergebnis erhalten Sie eine Figur, die links im Clientbereich angezeigt wird, wenn der Clientbereich breiter als hoch ist. Ist der Clientbereich höher als breit, so wird die Figur oben angezeigt:



Wenn Sie den Kreis ausdrucken, ist er so groß wie der bedruckbare Seitenbereich breit ist.

Es gibt allerdings ein kleines Problem bei diesem Programm. Reduzieren Sie die Fenstergröße einmal so weit es geht. Die Fensterbreite kann nur bis zu einem gewissen Limit reduziert werden, die Höhe des Fensters lässt sich jedoch bis auf 0 verkleinern. In diesem Fall wird eine Ausnahme ausgelöst, da die *DoPage*-Methode *PageScale* auf 0 setzt, ein unzulässiger Wert.

Dieses Problem kann auf verschiedenen Wegen gelöst werden. Am einfachsten ist es, die *DoPage*-Methode abzubrechen, wenn die Höhe des Clientbereichs 0 beträgt:

```
If cy = 0 Then Return
```

Dieses Verhalten ist in Ordnung, da eine Zeichenoperation in diesem Fall ohnehin nicht sinnvoll wäre.

Finden Sie es nicht etwas merkwürdig, dass ungeachtet des Werts 0 für die Maße des Clientbereichs immer noch ein Aufruf der *OnPaint*-Methode durchgeführt wird? Es könnte nicht schaden, zu Beginn der *OnPaint*-Methode eine Anweisung wie diese einzufügen:

```
If pea.ClipRectangle.IsEmpty Then Return
```

Diese Anweisung entspricht der folgenden:

```
If grfx.IsVisibleClipEmpty Then Return
```

Eine sehr spezielle Lösung wäre die Verwendung der Methode *Math.Max* in der Berechnung der Eigenschaft *PageScale*, um Nullwerte zu vermeiden:

```
grfx.PageScale = Math.Min(szf.Width / grfx.DpiX / 1000, Math.Max(szf.Height, 1) / grfx.DpiY / 1000)
```

Sie können auch Ihre Kenntnisse zur Ausnahmebehandlung in Visual Basic .NET unter Beweis stellen und die Anweisung in einem *Try*-Block unterbringen:

```
Try
    grfx.PageScale = Math.Min(szf.Width / grfx.DpiX / 1000, szf.Height / grfx.DpiY / 1000)
Catch
    Return
End Try
```

Eine vielleicht nicht ganz so offensichtliche Vorgehensweise wäre, den Clientbereich von vornherein daran zu hindern, auf eine Höhe von 0 zu schrumpfen. Die shared Eigenschaft *SystemInformation.MinimumWindowSize* gibt eine Größe zurück, deren Wert die Summe der Titelleistenhöhe plus zwei mal die Rahmenhöhe zurückgibt. Der minimale Breitenwert liegt erheblich höher, um zu gewährleisten, dass die Titelleiste des Programms immer teilweise sichtbar bleibt.

Sie können die Eigenschaft *MinimumSize* eines Formulars einstellen, um eine Mindestgröße für das Fenster festzulegen. Fügen Sie die folgende Zeile in den Konstruktor von ArbitraryCoordinates ein:

```
MinimumSize = Size.op_Addition(SystemInformation.MinimumWindowSize, New Size(0, 1))
```

Was nicht funktioniert

Es gibt verschiedene Dinge, die bei der Seitentransformation nicht funktionieren. Zunächst können Sie *PageScale* nicht auf negative Werte setzen, d.h., Sie können *x*-Koordinaten nicht nach links (was in den meisten Fällen sowieso nicht wünschenswert ist) oder *y*-Koordinaten nach oben ansteigen lassen (was für die eher mathematisch Denkenden unter uns ganz nützlich wäre).

Zweitens ist es nicht möglich, für die *x*- und *y*-Richtung unterschiedliche Einheiten zu verwenden. Die Eigenschaften *PageScale* und *PageUnit* gelten immer für beide Achsen. Eine Funktion wie diese

```
grfx.DrawEllipse(pn, 0, 0, 100, 100)
```

zeichnet unabhängig von der Seitentransformation immer einen Kreis. Hierbei gilt nur eine Ausnahme: Wenn Sie als *PageUnit GraphicsUnit.Pixel* einstellen und die horizontale Auflösung für das Ausgabegerät nicht mit der vertikalen Auflösung übereinstimmt. Dieses Problem tritt beim Bildschirm eher selten auf, beim Drucker ist dies allerdings häufiger der Fall.

Drittens ist eine Änderung des Ursprungs nicht möglich. Der Punkt (0, 0) entspricht in Seitenkoordinaten immer der oberen linken Ecke des Clientbereichs bzw. des bedruckbaren Bereichs der Druckerseite.

GDI+ unterstützt jedoch glücklicherweise eine weitere Transformation, die all dies und noch mehr ermöglicht.

Hello, World Transform

Eine von GDI+ unterstützte Transformation ist die *Welttransformation.* Sie verwendet eine übliche 3 × 3-Matrix; anstelle dieser Matrix können jedoch auch einige sehr praktische Methoden verwendet werden. Sehen wir uns zunächst das folgende Programm an, mit dem der erste Absatz von Herman Melvilles *Moby Dick* angezeigt wird.

MobyDick.vb

```
Imports System
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Windows.Forms

Class MobyDick
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New MobyDick())
    End Sub

    Sub New()
        Text = "Moby-Dick by Herman Melville"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)

        ' Hier RotateTransform, ScaleTransform,
        '   TranslateTransform und andere Aufrufe einfügen.

        grfx.DrawString("Call me Ishmael. Some years ago" & ChrW(&H2014) &
            "never mind how long precisely" & ChrW(&H2014) & _
            "having little or no money in my purse, and " & _
            "nothing particular to interest me on shore, I " & _
            "thought I would sail about a little and see " & _
            "the watery part of the world. It is a way I " & _
            "have of driving off the spleen, and " & _
            "regulating the circulation. Whenever I find " & _
            "myself growing grim about the mouth; whenever " & _
            "it is a damp, drizzly November in my soul; " & _
            "whenever I find myself involuntarily pausing " & _
            "before coffin warehouses, and bringing up the " & _
            "rear of every funeral I meet and especially " & _
            "whenever my hypos get such an upper hand of " & _
            "me, that it requires a strong moral principle " & _
            "to prevent me from deliberately stepping into " & _
            "the street, and methodically knocking " & _
            "people's hats off" & ChrW(&H2014) & "then, I " & _
            "account it high time to get to sea as soon as " & _
            "I can. This is my substitute for pistol " & _
            "and ball. With a philosophical flourish Cato " & _
            "throws himself upon his sword; I quietly take " & _
            "to the ship. There is nothing surprising in " & _
            "this. If they but knew it, almost all men in " & _
            "their degree, some time or other, cherish " & _
            "very nearly the same feelings towards the " & _
            "ocean with me.", _
            Font, New SolidBrush(clr), _
            New RectangleF(0, 0, cx, cy))
     End Sub
End Class
```

Hier gibt es nichts Neues, abgesehen davon, dass ich angegeben habe, wo Sie ein oder zwei Zeilen einfügen können. Kompilieren Sie das Programm anschließend neu und beobachten Sie, was geschieht.

Versuchen Sie es zunächst mit folgender Zeile:

```
grfx.RotateTransform(45)
```

Achten Sie darauf, dass Sie diesen Aufruf vor dem Aufruf von *DrawString* einfügen. Als Ergebnis wird der Text um 45 Grad im Uhrzeigersinn gedreht:

Einfach, oder? Beachten Sie, dass der Text sich weiterhin innerhalb des in *DrawString* definierten Rechtecks befindet, das Rechteck wurde jedoch zusammen mit dem Text gedreht. Sie können den Text auch ausdrucken, allerdings wird die Erzeugung der Druckdatei einige Zeit in Anspruch nehmen.

Auf welche Funktionen wirkt sich *RotateTransform* aus? Auf alle bisher besprochenen *Graphics*-Zeichenfunktionen. Wenn wir uns näher mit Grafik beschäftigen, werde ich darauf achten, immer anzugeben, ob sich die Welttransformation auf etwas auswirkt oder nicht.

Das Argument von *RotateTransform* ist ein *Single*-Wert, der negativ oder positiv sein kann. Versuchen Sie es einmal hiermit:

```
grfx.RotateTransform(-45)
```

Der Text wird auf diese Weise um 45° gegen den Uhrzeigersinn gedreht. Der Winkel kann auch über 360° liegen oder weniger als –360° betragen. In unserem Beispiel führt jeder Wert, der nicht in einen Winkel zwischen –90° und 90° umgerechnet werden kann, zu einer Rotation, bei der der Text außerhalb des sichtbaren Fensterbereichs liegt.

Aufeinander folgende Aufrufe von *RotateTransform* sind kumulativ. Die Aufrufe

```
grfx.RotateTransform(5)
grfx.RotateTransform(10)
grfx.RotateTransform(-20)
```

führen daher dazu, dass der Text um 5° gegen den Uhrzeigersinn gedreht wird.

Versuchen Sie diese Zeile:

```
grfx.ScaleTransform(1, 3)
```

Diese Funktion erhöht die Koordinaten und Größen angezeigter Grafiken. Das erste Argument wirkt sich auf die horizontalen Koordinaten- und Größenwerte aus, das zweite Argument verändert die entsprechenden vertikalen Werte. Im Programm MobyDick verursacht der Aufruf dieser Funktion zwar keine Veränderung der Textbreite, er verdreifacht jedoch die Schrifthöhe. Der Aufruf

```
grfx.ScaleTransform(3, 1)
```

wirkt sich dagegen nicht auf die Schrifthöhe, sondern auf deren Breite aus. Das Anzeigerechteck wird entsprechend vergrößert, sodass der Text weiterhin dieselben Zeilenumbrüche aufweist. Diese beiden Effekte können auch kombiniert werden:

```
grfx.ScaleTransform(3, 3)
```

Es handelt sich wieder um *Single*-Werte und sie werden zusammen verwendet. Das Skalieren der horizontalen und vertikalen Größenwerte um den Faktor 3 kann auch durch folgende Aufrufe erzielt werden:

```
grfx.ScaleTransform(3, 1)
grfx.ScaleTransform(1, 3)
```

oder

```
grfx.ScaleTransform(CSng(Math.Sqrt(3)), CSng(Math.Sqrt(3)))
grfx.ScaleTransform(CSng(Math.Sqrt(3)), CSng(Math.Sqrt(3)))
```

Sehr interessant ist hierbei, dass die Vergrößerung des Texts nicht zu unansehnlichen Treppeneffekten führt. Es scheint so, ab ob anstelle einer bloßen Vergrößerung einer vorhandenen Schrift tatsächlich eine andere Schriftgröße verwendet würde.

Können die Skalierungswerte negativ sein? Ja, können sie. Wenn Sie das jedoch jetzt gleich ausprobieren, führt es nicht zum gewünschten Ergebnis. Ich werde in Kürze auf die Verwendung negativer Skalierungswerte und die sich dabei ergebenden erstaunlichen Effekte zu sprechen kommen. Der Skalierungswert darf nicht 0 sein, sonst löst die Funktion eine Ausnahme aus.

So, und nun das Langweiligste zum Schluss. Der *TranslateTransform*-Aufruf kann Koordinaten auch einfach entlang der horizontalen und vertikalen Achse verschieben. Wenn Sie beispielsweise diesen Aufruf

```
grfx.TranslateTransform(100, 50)
```

in das Programm MobyDick einfügen, wird der Text relativ zum Ursprung des Clientbereichs um 100 Pixel nach rechts und um 50 Pixel nach unten verschoben. Wenn Sie diese Version ausdrucken, wird der Text relativ zum Ursprung des bedruckbaren Seitenbereichs um 1 Zoll nach rechts und 1/2 Zoll nach unten verschoben angezeigt. Negative Werte für das erste Argument verschieben den Text nach links aus dem Clientbereich hinaus; negative *y*-Werte verschieben den Text nach oben aus dem Clientbereich hinaus.

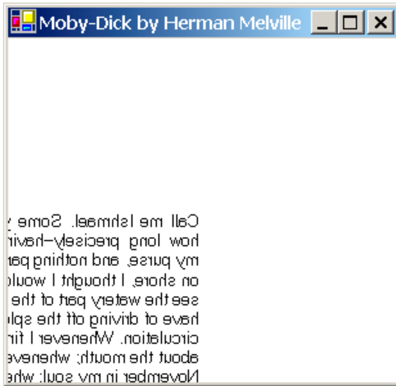
Die Verschiebung des Texts eignet sich jedoch sehr gut zur Verdeutlichung anderer Techniken. Fügen Sie den folgenden Aufruf in das Programm ein:

```
grfx.TranslateTransform(cx \ 2, cy \ 2)
```

Damit beginnt der Text im Mittelpunkt des Client- oder Druckbereichs. Das allein ist noch nicht sonderlich interessant, aber fügen Sie nach dem Aufruf von *TranslateTransform* nun einmal folgende Anweisung ein:

```
grfx.ScaleTransform(-1, 1)
```

Na, wenn das nicht interessant ist! Es geschieht Folgendes: Der Text wird an der vertikalen Achse gespiegelt und erscheint im linken unteren Quadranten des Clientbereichs:



Ersetzen Sie nun den *ScaleTransform*-Aufruf durch den folgenden:

```
grfx.ScaleTransform(1, -1)
```

Jetzt wird der Text an der horizontalen Achse gespiegelt und auf dem Kopf angezeigt. Wieder können beide Effekte kombiniert werden:

```
grfx.ScaleTransform(-1, -1)
```

Jetzt wissen Sie auch, warum Sie für den *ScaleTransform*-Aufruf selbst keine negativen Argumente verwenden können – der Text würde außerhalb des sichtbaren Clientbereichs angezeigt. Sie müssen den Text weiter nach links und nach oben verschieben, damit dieser Effekt zu sehen ist.

Okay, kehren wir nun einmal die Reihenfolge der Aufrufe von *TranslateTransform* und *ScaleTransform* um:

```
grfx.ScaleTransform(-1, 1)
grfx.TranslateTransform(cx \ 2, cy \ 2)
```

Jetzt wird nichts angezeigt. Wie Sie sich vielleicht denken können, liegt das daran, dass der Text irgendwie aus dem sichtbaren Clientbereich verschoben wurde. Sie können den Text auf zwei Arten wieder zurückholen. Eine Möglichkeit besteht darin, als erstes Argument des *TranslateTransform*-Aufrufs einen negativen Wert anzugeben:

```
grfx.ScaleTransform(-1, 1)
grfx.TranslateTransform(-cx \ 2, cy \ 2)
```

Damit wird der Text wieder in der Mitte des Clientbereichs an der vertikalen Achse gespiegelt. Ich erwarte übrigens nicht, dass Sie bereits jetzt nachvollziehen können, wie das funktioniert. Eine leichte Verwirrung Ihrerseits wäre zu diesem Zeitpunkt nicht unangemessen.

Um diese Verwirrung noch ein wenig zu vergrößern, hier eine zweite Variante. Behalten Sie das erste Argument bei, verwenden Sie aber jetzt diese Überladung der Methode *TranslateTransform:*

```
grfx.ScaleTransform(-1, 1)
grfx.TranslateTransform(cx \ 2, cy \ 2, MatrixOrder.Append)
```

Jede der drei bisher behandelten Methoden – *RotateTransform*, *ScaleTransform* und *TranslateTransform* – wurde überladen, um als letztes Argument *MatrixOrder* verwenden zu können, eine im Namespace *System.Drawing.Drawing2D* definierte Enumeration. (Aus diesem Grund habe ich auch noch die notwendige *Imports*-Anweisung am Anfang des Programms MobyDick eingefügt.)

Nachfolgend finden Sie die formalen Definitionen der *Graphics*-Methoden (die bisher besprochenen plus eine weitere):

Graphics-Methoden (Auswahl)

```
Sub TranslateTransform(ByVal dx As Single, ByVal dy As Single)
Sub TranslateTransform(ByVal dx As Single, ByVal dy As Single, ByVal mo As MatrixOrder)
Sub ScaleTransform(ByVal sx As Single, ByVal sy As Single)
Sub ScaleTransform(ByVal sx As Single, ByVal sy As Single, ByVal mo As MatrixOrder)
Sub RotateTransform(ByVal fAngle As Single)
Sub RotateTransform(ByVal fAngle As Single, ByVal mo As MatrixOrder)
Sub ResetTransform()
```

Der Aufruf von *ResetTransform* führt zu einer Zurücksetzung auf die normalen Werte. Die Enumeration *MatrixOrder* verfügt lediglich über zwei Member:

MatrixOrder-Enumeration

Member	Wert	Beschreibung
Prepend	0	Standardwert
Append	1	Kehrt die Anwendungsreihenfolge um

Auf die Auswirkung dieser Enumerationswerte gehe ich im weiteren Verlauf dieses Kapitels noch näher ein.

Ein Überblick

Die Koordinaten, die Sie an die verschiedenen Zeichenmethoden der Klasse *Graphics* übergeben, werden als *Weltkoordinaten* bezeichnet. Weltkoordinaten werden zuerst der Welttransformation unterzogen, das ist das, mit dem wir durch Aufruf von *TranslateTransform, ScaleTransform* und *RotateTransform* herumexperimentiert haben. Ich werde in Kürze noch eine formale Definition der Welttransformation liefern.

Mithilfe der Welttransformation werden Weltkoordinaten in Seitenkoordinaten umgewandelt. Die Seitentransformation – die über die Eigenschaften *PageUnit* und *PageScale* des *Graphics*-Objekts definierte Transformation – wandelt die Seitenkoordinaten in Gerätekoordinaten um, die relativ zur oberen linken Ecke des Clientbereichs oder als bedruckbarer Seitenbereich in Pixeln angegeben werden.

Für Funktionen wie die *Graphics*-Methode *MeasureString* oder die *Font*-Methode *GetHeight* wird dieser Prozess umgekehrt. Gerätekoordinaten werden in diesem Fall zunächst in Seitenkoordinaten, dann in Weltkoordinaten konvertiert und anschließend von der Methode zurückgegeben.

Lineare Transformationen

Sehen wir uns an, welche Wirkungen mathematisch gesehen der Aufruf verschiedener Transformationsmethoden nach sich zieht. Die einfachste dieser Methoden scheint diese zu sein:

```
grfx.TranslateTransform(dx, dy)
```

Hierbei werden die Argumente durch d_x und d_y dargestellt. (Das *d* steht für *Delta,* was in der Mathematik Änderung bedeutet.) Die sich aus diesem Methodenaufruf ergebende Welttransformation lautet folgendermaßen:

$$x_{page} = x_{world} + d_x$$
$$y_{page} = y_{world} + d_y$$

Eigentlich ganz einfach. Wie Sie gesehen haben, führt der *TranslateTransform*-Aufruf zu einer Verschiebung aller Koordinaten.

Ähnlich verhält es sich mit *ScaleTransform:*

```
grfx.ScaleTransform(sx, sy)
```

Das *s* steht für *Skalierung.* Diese Welttransformation verwendet anstelle einer Addition eine Multiplikation:

$$x_{page} = s_x \cdot x_{world}$$
$$y_{page} = s_y \cdot y_{world}$$

Dieser Skalierungseffekt ist einer Seitentransformation sehr ähnlich.

Beim Aufruf von

```
grfx.RotateTransform(α)
```

mit einem Winkel von α ergibt sich ... keine Angst, Sie müssen nicht raten. Die sich ergebende Transformation ist offensichtlich etwas komplexer und sieht so aus:

$$x_{page} = x_{world} \cdot \cos(\alpha) + y_{world} \cdot \sin(\alpha)$$
$$y_{page} = -x_{world} \cdot \sin(\alpha) + y_{world} \cdot \cos(\alpha)$$

Anhand der folgenden kleinen Tabelle mit Sinus- und Cosinus-Werten können Sie sich davon überzeugen, dass diese Formeln tatsächlich stimmen:

Winkel α	Sinus	Cosinus
0	0	1
45	$\sqrt{1/2}$	$\sqrt{1/2}$
90	1	0
135	$\sqrt{1/2}$	$-\sqrt{1/2}$
180	0	–1
225	$-\sqrt{1/2}$	$-\sqrt{1/2}$
270	–1	0
315	$-\sqrt{1/2}$	$\sqrt{1/2}$
360	0	1

Übrigens: Wenn Sie mit dieser Materie bereits aus anderen grafischen Programmierumgebungen vertraut sind, werden Sie bemerken, dass die beiden Formeln für die Rotation etwas seltsam

aussehen. Dies liegt daran, dass GDI+ die Rotation im Uhrzeigersinn beschreibt. In stärker mathematisch ausgeprägten Umgebungen findet die Rotation gegen den Uhrzeigersinn statt. Bei Drehungen gegen den Uhrzeigersinn ist der Sinuswert in der ersten Formel negativ und in der zweiten Formel positiv.

Alle drei Transformationen zusammen können in zwei Formeln ausgedrückt werden:

$$x_{page} = s_x \cdot x_{world} + r_x \cdot y_{world} + d_x$$
$$y_{page} = r_y \cdot x_{world} + s_y \cdot y_{world} + d_y$$

Hierbei stehen s_x, s_y, r_x, r_y, d_x und d_y für Konstanten, die eine bestimmte Transformation definieren. Die Skalierungsfaktoren s_x und s_y, sowie die Verschiebungsfaktoren d_x und d_y haben Sie bereits kennen gelernt. Ferner haben Sie erfahren, dass bestimmte, durch trigonometrische Funktionen spezieller Winkel definierte Kombinationen aus s_x, s_y, r_x und r_y zu einer Rotation führen können. Die Faktoren r_x und r_y selbst haben ebenfalls eine Bedeutung, und die grafische Auswirkung dieser Konstanten wird in Kürze zu sehen sein.

Diese beiden Formeln werden zusammen als *allgemeine lineare Transformation der Ebene* bezeichnet.* Obwohl x_{page} und y_{page} Funktionen von x_{world} und y_{world} sind, haben diese Formeln nichts mit Potenzen von x_{world} oder y_{world} zu tun. Die lineare Eigenschaft der Welttransformation impliziert verschiedene Einschränkungen:

- Die Welttransformation wandelt eine Gerade stets in eine andere Gerade um. Geraden werden nie zu gekrümmten Linien.
- Ein paralleles Linienpaar kann nicht in zwei nicht paralle Linien transformiert werden.
- Zwei Objekte gleicher Größe können nie in zwei Objekte unterschiedlicher Größe transformiert werden.
- Parallelogramme (inklusive Rechtecke) werden immer in Parallelogramme transformiert; Ellipsen werden stets auch wieder in eine Ellipse umgewandelt.

Wenn Sie am Anfang eines *Paint-* oder *PrintPage*-Ereignisses eine neue, unveränderte *Graphics*-Klasse verwenden, wird die angewendete Welttransformation als *Identitätstransformation* bezeichnet: Die Faktoren s_x und s_y werden auf 1 gesetzt, alle weiteren Faktoren werden auf 0 eingestellt. Die Methode *ResetTransform* stellt das *Graphics*-Objekt wieder auf die Identitätstransformation zurück.

Wie Sie gesehen haben, werden nacheinander ausgeführte Aufrufe von *TranslateTransform*, *ScaleTransform* und *RotateTransform* kumuliert. Die sich ergebende Welttransformation ist je nach Aufrufreihenfolge dieser Methoden jedoch unterschiedlich. Der Grund hierfür ist schnell erklärt. Was jetzt folgt, ist kein schöner Anblick mehr, wenn Sie sich also während der Angst einflößenden Abschnitte lieber die Augen zuhalten möchten, geht das in Ordnung.

Angenommen, es liegt eine Welttransformation mit der Bezeichnung T_1 vor:

$$x' = s_{x1} \cdot x + r_{x1} \cdot y + d_{x1}$$
$$y' = r_{y1} \cdot x + s_{y1} \cdot y + d_{y1}$$

Statt die Welt- und Seitenkoordinaten mithilfe von Indizes anzugeben, lauten die Weltkoordinaten hier einfach x und y, die Seitenkoordinaten lauten x' und y'. Nehmen wir weiter an, es gibt eine zweite Transformation mit der Bezeichnung T_2 mit anderen Faktoren:

$$x' = s_{x2} \cdot x + r_{x2} \cdot y + d_{x2}$$
$$y' = r_{y2} \cdot x + s_{y2} \cdot y + d_{y2}$$

* Siehe Anthony J. Pettofrezzo: *Matrices and Transformations*. New York: Dover, 1978, Kapitel 3, hier insbesondere die Abschnitte 3–7.

Die Anwendung von T_1 auf die Weltkoordinaten und eine anschließende Anwendung von T_2 auf das Ergebnis führt zu dieser Transformation:

$$x' = s_{x2} \cdot s_{x1} \cdot x + s_{x2} \cdot r_{x1} \cdot y + s_{x2} \cdot d_{x1} + r_{x2} \cdot r_{y1} \cdot x + r_{x2} \cdot s_{y1} \cdot y + r_{x2} \cdot d_{y1} + d_{x2}$$
$$y' = r_{y2} \cdot s_{x1} \cdot x + r_{y2} \cdot r_{x1} \cdot y + r_{y2} \cdot d_{x1} + s_{y2} \cdot r_{y1} \cdot x + s_{y2} \cdot s_{y1} \cdot y + s_{y2} \cdot d_{y1} + d_{y2}$$

Wenn Sie diese Gleichungen umformen, erhalten Sie Folgendes:

$$x' = (s_{x2} \cdot s_{x1} + r_{x2} \cdot r_{y1}) \cdot x + (s_{x2} \cdot r_{x1} + r_{x2} \cdot s_{y1}) \cdot y + (s_{x2} \cdot d_{x1} + r_{x2} \cdot d_{y1} + d_{x2})$$
$$y' = (r_{y2} \cdot s_{x1} + s_{y2} \cdot r_{y1}) \cdot x + (r_{y2} \cdot r_{x1} + s_{y2} \cdot s_{y1}) \cdot y + (r_{y2} \cdot d_{x1} + s_{y2} \cdot d_{y1} + d_{y2})$$

Wenn Sie zuerst T_2 und dann T_1 anwenden, erhalten Sie ein anderes Ergebnis:

$$x' = s_{x1} \cdot s_{x2} \cdot x + s_{x1} \cdot r_{x2} \cdot y + s_{x1} \cdot d_{x2} + r_{x1} \cdot r_{y2} \cdot x + r_{x1} \cdot s_{y2} \cdot y + r_{x1} \cdot d_{y2} + d_{x1}$$
$$y' = r_{y1} \cdot s_{x2} \cdot x + r_{y1} \cdot r_{x2} \cdot y + r_{y1} \cdot d_{x2} + s_{y1} \cdot r_{y2} \cdot x + s_{y1} \cdot s_{y2} \cdot y + s_{y1} \cdot d_{y2} + d_{y1}$$

Umgeformt lautet dies:

$$x' = (s_{x1} \cdot s_{x2} + r_{x1} \cdot r_{y2}) \cdot x + (s_{x1} \cdot r_{x2} + r_{x1} \cdot s_{y2}) \cdot y + (s_{x1} \cdot d_{x2} + r_{x1} \cdot d_{y2} + d_{x1})$$
$$y' = (r_{y1} \cdot s_{x2} + s_{y1} \cdot r_{y2}) \cdot x + (r_{y1} \cdot r_{x2} + s_{y1} \cdot s_{y2}) \cdot y + (r_{y1} \cdot d_{x2} + s_{y1} \cdot d_{y2} + d_{y1})$$

Und aus genau diesem Grund erhalten Sie je nach Reihenfolge der Aufrufe von *ScaleTransform* und *TranslateTransform* unterschiedliche Ergebnisse.

Vorstellung von Matrizen

Wenn es in der Mathematik unübersichtlich wird (wie z.B. in den soeben gezeigten Berechnungen), besteht die Lösung in der Regel darin, nicht etwa Elemente wegzulassen, sondern etwas Neues einzuführen. Hier ist die Einführung einer *Matrix* sinnvoll, denn die Matrizenalgebra ist weithin bekannt (zumindest unter Mathematikern). Sie können eine lineare Transformation durch eine Matrix darstellen; das Anwenden mehrerer Transformationen entspricht einer Multiplikation der Matrizen.

Eine *Matrix* ist die Anordnung von Zahlen in Form einer rechteckigen Tabelle. Nachfolgend sehen Sie eine Matrix mit drei Spalten und zwei Zeilen:

$$\begin{vmatrix} 27 & 9 & 14 \\ 3 & 0 & 88 \end{vmatrix}$$

Arrays werden üblicherweise durch Großbuchstaben dargestellt. Bei der Multiplikation zweier Matrizen

$$A \times B = C$$

muss die Anzahl der Spalten in Matrix A mit der Zeilenanzahl in Matrix B übereinstimmen. Die Anzahl der Zeilen im Produkt C entspricht der Zeilenanzahl von Matrix A. Die Anzahl der Spalten in C entspricht der Spaltenzahl in Matrix B. Die Zahl in der i-ten Zeile und der j-ten Spalte in C ist gleich der Summe der Produkte aus den Zahlen der i-ten Zeile von A und den entsprechenden Werten in der j-ten Spalte von B.* Die Matrizenmultiplikation ist nicht kommutativ, das Produkt $A \times B$ ist also nicht notwendigerweise gleich dem Produkt $B \times A$.

Wenn wir es nicht auch mit einer Verschiebung zu tun haben würden, könnten wir die Weltkoordinaten (x, y) als eine 1×2-Matrix und die Transformationsmatrix als 2×2-Matrix darstellen.

---

* Beispiele siehe Pettofrezzo, Abschnitte 1–2.

Sie multiplizieren diese zwei Matrizen und stellen die erhaltenen Seitenkoordinaten (x', y') als eine weitere 1 × 2-Matrix dar:

$$\begin{vmatrix} x & y \end{vmatrix} \times \begin{vmatrix} s_x & r_y \\ r_x & s_y \end{vmatrix} = \begin{vmatrix} x' & y' \end{vmatrix}$$

Die Anwendung der Multiplikationsregeln auf die Matrix führt zu folgenden Formeln:

$$x' = s_x \cdot x + r_x \cdot y$$
$$y' = r_y \cdot x + s_y \cdot y$$

Diese Formeln sind jedoch nicht ganz vollständig, denn zur Welttransformation gehört ja auch noch die Verschiebung. Damit die Matrizenmultiplikation richtig funktioniert, müssen Welt- und Seitenkoordinaten auf 1 × 3-Matrizen erweitert werden und die Transformation ist eine 3 × 3-Matrix:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} s_x & r_y & 0 \\ r_x & s_y & 0 \\ d_x & d_y & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & 1 \end{vmatrix}$$

Die sich ergebenden Formeln lauten folgendermaßen:

$$x' = s_x \cdot x + r_x \cdot y + d_x$$
$$y' = r_y \cdot x + s_y \cdot y + d_y$$

Die Art der Transformation kann durch eine so genannte *Matrixtransformation* dargestellt werden.

Eine Matrixtransformation ohne Auswirkung weist Skalierungsfaktoren von 1 auf, r und d haben den Faktor 0:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Diese Matrix wird als *Identitätsmatrix* bezeichnet.

Die Klasse *Matrix*

Die Matrixtransformation ist in der Klasse *Matrix* gekapselt, die wiederum im Namespace *System.Drawing.Drawing2D* definiert ist. Es stehen zur Erstellung eines *Matrix*-Objekts vier Konstruktoren zur Verfügung, von denen nachfolgend zwei aufgeführt sind:

Matrix-Konstruktoren (Auswahl)

```
Matrix()
Matrix(ByVal sx As Single, ByVal ry As Single,
       ByVal rx As Single, ByVal sy As Single,
       ByVal dx As Single, ByVal dy As Single)
```

Der zweite Konstruktor ermöglicht die Angabe aller sechs Konstanten, durch die die Matrixtransformation definiert wird. Die Skalierungsfaktoren *sx* und *sy* müssen hierbei ungleich 0 sein. (Sonst erhalten Sie einen Ausnahmefehler.)

Die Klasse *Graphics* verfügt über eine *Transform*-Eigenschaft mit Lese-/Schreibzugriff, bei der es sich um ein *Matrix*-Objekt handelt:

Graphics-Eigenschaft (Auswahl)

Eigenschaft	Typ	Zugriff
Transform	*Matrix*	Get/Set

Jeder Aufruf von *TranslateTransform, ScaleTransform, RotateTransform* oder *ResetTransform* wirkt sich auf die Eigenschaft *Transform* aus. Sie können sie auch direkt einstellen. Der Aufruf

```
grfx.Transform = New Matrix(1, 0, 0, 1, 0, 0)
```

bewirkt dasselbe wie *ResetTransform*.

Die Klasse *Matrix* verfügt über fünf Eigenschaften, die alle schreibgeschützt sind:

Matrix-Eigenschaften

Eigenschaft	Typ	Zugriff	Beschreibung
Elements	*Single()*	Get	Sechs Transformationskonstanten
OffsetX	*Single*	Get	Transformationskonstante d_x
OffsetY	*Single*	Get	Transformationskonstante d_y
IsIdentity	*Boolean*	Get	Diagonale aus Einsen
IsInvertible	*Boolean*	Get	Kann invertiert werden

Sehen wir uns nun eine zusammengesetzte Transformation an. Angenommen, der erste Aufruf lautet folgendermaßen:

```
grfx.ScaleTransform(2, 2)
```

Ihr Programm könnte die Ergebnismatrix durch folgenden Aufruf untersuchen:

```
Dim afElements() As Single = grfx.Transform.Elements
```

Die angezeigten Werte 2, 0, 0, 2, 0, 0 können in Form der folgenden Matrix dargestellt werden:

$$\begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Anschließend führen Sie diesen Aufruf aus:

```
grfx.TranslateTransform(100, 100)
```

Dieser Aufruf allein würde zu folgender Matrix führen:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix}$$

Die neue Transformation setzt sich jedoch aus beiden Methodenaufrufen zusammen. Die Matrix für den zweiten Aufruf wird mit der vorhandenen *Transform*-Eigenschaft multipliziert und das Ergebnis ist die neue *Transform*-Eigenschaft:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix} \times \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 200 & 200 & 1 \end{vmatrix}$$

Testen Sie nun, was passiert, wenn Sie die Aufrufe von *ScaleTransform* und *TranslateTransform* in umgekehrter Reihenfolge ausführen:

```
grfx.TranslateTransform(100, 100)
grfx.ScaleTransform(2, 2)
```

Wieder wird die resultierende Transformation durch Multiplikation der zweiten Matrix mit der ersten Matrix berechnet:

$$\begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix} = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 100 & 100 & 1 \end{vmatrix}$$

Sie erhalten diese Transformation auch durch folgende Syntax:

```
grfx.ScaleTransform(2, 2)
grfx.TranslateTransform(100, 100, MatrixOrder.Append)
```

Das Argument *MatrixOrder.Append* gibt an, dass die neue Transformation zur vorhandenen hinzugefügt wird. Der Standardwert lautet *MatrixOrder.Prepend*.

Die Klasse *Graphics* verfügt über eine weitere Methode für die Welttransformation:

***Graphics*-Methoden (Auswahl)**

```
Sub MultiplyTransform(ByVal matx As Matrix)
Sub MultiplyTransform(ByVal matx As Matrix, ByVal mo As MatrixOrder)
```

Diese Methode ermöglicht die Multiplikation einer vorhandenen Transformation mit einer neuen.

In Kapitel 15 werde ich noch näher auf die Klasse *Matrix* eingehen.

Scherungen

Kehren wir zum Programm MobyDick zurück und fügen wir die folgende Anweisung ein:

```
grfx.Transform = New Matrix(1, 0, 0, 3, 0, 0)
```

Diese Anweisung hat dieselbe Auswirkung wie folgender Aufruf:

```
grfx.ScaleTransform(1, 3)
```

Wir haben bisher noch nicht ausprobiert, was geschieht, wenn nur die Faktoren r_x und r_y verwendet werden. Sehen Sie sich folgenden Aufruf an:

```
grfx.Transform = New Matrix(1, 0, 0.5, 1, 0, 0)
```

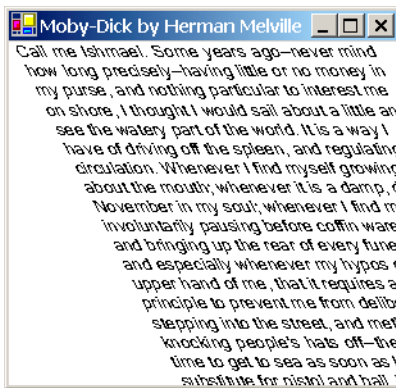
Dieser Aufruf führt zu folgender Transformationsmatrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Und hier sehen Sie die Transformationsformeln:

$x' = x + 0.5 \cdot y$
$y' = y$

Beachten Sie, dass die x-Koordinatenwerte um den y-Wert erhöht werden. Wenn y gleich 0 ist (der Anfang des Clientbereichs), wird keine Transformation durchgeführt. Bei ansteigendem y-Wert (im Clientbereich abwärts) erhöht sich der x-Wert gleichermaßen. Der sich ergebende Effekt wird als *Scherung* bezeichnet.



Der hier gezeigte Effekt ist genauer gesagt eine *horizontale Scherung* oder *x-Scherung*. Leider beginnt das Wort *Scherung* mit dem gleichen Buchstaben wie *Skalierung*, daher habe ich den Scherungsfaktor in den obigen Transformationsformeln durch den Buchstaben r gekennzeichnet.

Sie können auch eine *vertikale* oder *y-Scherung* angeben:

```
grfx.Transform = New Matrix(1, 0.5, 0, 1, 0, 0)
```

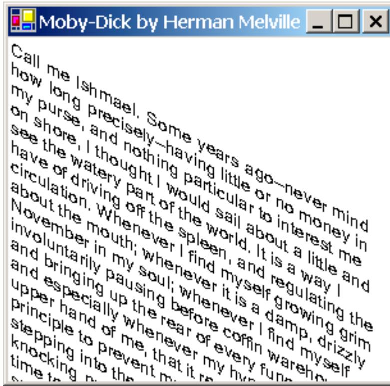
Diese Matrix sieht so aus:

$$\begin{vmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Die Transformationsformeln lauten:

$x' = x$
$y' = 0.5 \cdot x + y$

Beachten Sie, dass jede Textzeile weiterhin am linken Rand des Clientbereichs beginnt:



Die Rotation ist eigentlich eine Kombination aus horizontaler und vertikaler Scherung. Einige Kombinationen, z.B. die nachfolgende, funktionieren allerdings nicht:

```
grfx.Transform = New Matrix(1, 1, 1, 1, 0, 0)
```

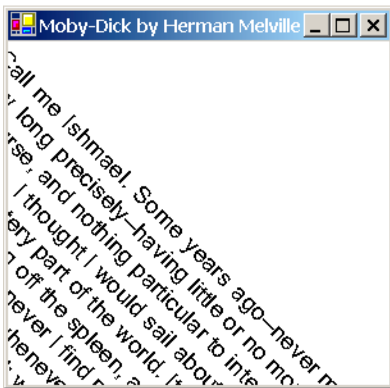
Dieser Aufruf definiert die folgende Transformation:

$$x' = x + y$$
$$y' = x + y$$

Durch diese Transformation würde das Bild zu einer einzigen Zeile zusammengequetscht. Bevor dies geschieht, wird jedoch eine Ausnahme ausgelöst. Der folgende Aufruf funktioniert dagegen:

```
grfx.Transform = New Matrix(1, 1, -1, 1, 0, 0)
```

und führt zu folgender Anzeige:



Wenn Sie die ersten vier Argumente auf die Quadratwurzel von 1/2 setzen:

```
grfx.Transform = New Matrix(0.707, 0.707, -0.707, 0.707, 0, 0)
```

erzielen Sie dasselbe Ergebnis wie durch den Aufruf, mit dem wir ursprünglich begonnen haben:

```
grfx.RotateTransform(45)
```

Transformationen kombinieren

Seitentransformationen werden theoretisch überhaupt nicht benötigt. Die Seitentransformation sorgt lediglich für eine Skalierung, und eine Skalierung können Sie neben vielem anderen problemlos auch über eine Welttransformation vornehmen. In den Kapitel 9 und 11 werden Sie noch sehen, dass Bitmaps von der Seitentransformation häufig *nicht* beeinflusst werden, allerdings von der Welttransformation. Auch ist es häufig sinnvoll, die beiden Arten von Transformationen miteinander zu kombinieren, besonders dann, wenn Sie Figuren mit einer bestimmten Größe zeichnen möchten, die anschließend der Welttransformation unterzogen werden.

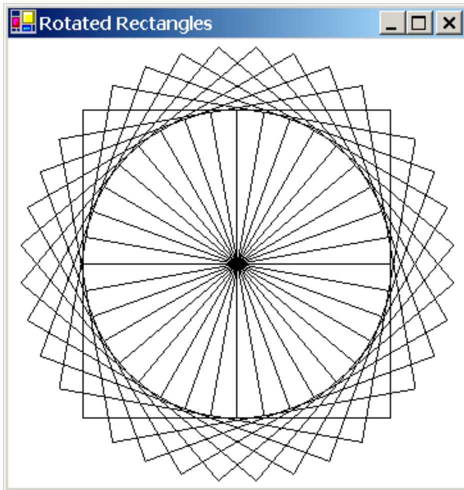
Das folgende Programm zeichnet 36 Rechtecke mit einer Seitenlänge von 1 Zoll, die um den Mittelpunkt des Anzeigebereichs rotiert werden.

RotatedRectangles.vb

```
Imports System
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Windows.Forms

Class RotatedRectangles
    Inherits PrintableForm

    Shared Shadows Sub Main()
        Application.Run(New RotatedRectangles())
    End Sub

    Sub New()
        Text = "Rotated Rectangles"
    End Sub

    Protected Overrides Sub DoPage(ByVal grfx As Graphics, _
            ByVal clr As Color, ByVal cx As Integer, ByVal cy As Integer)
        Dim i As Integer
        Dim pn As New Pen(clr)

        grfx.PageUnit = GraphicsUnit.Pixel
        Dim aptf() As PointF = {grfx.VisibleClipBounds.Size.ToPointF()}
        grfx.PageUnit = GraphicsUnit.Inch
        grfx.PageScale = 0.01

        grfx.TransformPoints(CoordinateSpace.Page, CoordinateSpace.Device, aptf)

        grfx.TranslateTransform(aptf(0).X / 2, aptf(0).Y / 2)

        For i = 0 To 35
            grfx.DrawRectangle(pn, 0, 0, 100, 100)
            grfx.RotateTransform(10)
        Next i
    End Sub
End Class
```

Der schwierige Part ist hier die Berechnung der Argumente für den *TranslateTransform*-Aufruf, der zum Verschieben des Weltkoordinatenursprungs in den Mittelpunkt des Anzeigebereichs erforderlich ist. Die *DoPage*-Methode stellt die Seiteneinheiten auf Pixel um, damit sie die *VisibleClipBounds*-Eigenschaft in Pixeln erhält. *DoPage* wechselt dann zu einer Seiteneinheit von 1/100 Zoll und wandelt die Breiten- und Höhenwerte des Anzeigebereichs in Seitenkoordinaten um. Im *TranslateTransform*-Aufruf werden diese Werte dann durch 2 geteilt, bevor sie verwendet werden.

Die *For*-Schleife ist einfach: Es wird ein 100 Einheiten hohes und breites Rechteck am Punkt (0, 0) gezeichnet. Anschließend führt der *RotateTransform*-Aufruf zur Vorbereitung des nächsten Schleifendurchlaufs eine Rotation um 10° durch. Und so sieht die Programmausgabe dann aus:



Zu wissen, wie die Rotation von Objekten um einen Ursprung funktioniert, wird sich im Programm AnalogClock in Kapitel 10 noch als nützlich erweisen.