Improving Performance in Rich User Interfaces for Embedded Systems

Windows Embedded Compact 7 Technical Article

Writers: Dion Hutchings, David Franklin, Frankie Anderson

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

Performance in applications that include graphical user interfaces (GUI) involves both usability and responsiveness. High performance, rich user interfaces (UI) provide appropriate, useful and intuitive displays, as well as quick and smooth responses to user actions. The providers, creators and developers of embedded devices have begun to add rich UIs to their products, and have likely encountered issues in driving the performance of the end-user experience on those devices. This paper identifies what the performance factors are, how performance is affected by choice of UI design and hardware platform, and where performance improvements can be made.

Introduction

Information and guidance given here will help you match your product hardware platform with your desired customer experience, and will help you to maximize the performance of the user interface (UI) for your product. The proliferation of rich media experiences across the Internet and desktop has raised consumer expectations, significantly raising the performance bar for all devices. The problem of meeting ever-increasing customer expectation can be rationally framed as a trade-off between complexity of the UI and, to a certain extent, the capability (cost) of the hardware platform.

The complexity of your user interface increases rapidly as it becomes richer and more dynamic, and its performance will suffer commensurately if it is launched on an inadequate hardware platform. In this white paper we try to provide some insight into the graphics rendering process to help you determine how to improve it with a combination of hardware, software, and design techniques. Hardware, in particular the graphics processing unit (GPU), has evolved enormously in the past few years, and understanding how it facilitates and affects the performance of your product is crucial.

Performance Challenges for Rich User Interfaces

Embedded systems suppliers and developers have long been optimizing their user interfaces (UIs) to perform on hardware-limited platforms. The advent of rich UIs has actually intensified performance issues by adding more complex paradigms for displaying and interacting with devices. Depictions of three-dimensional objects and animation are two aspects of a rich UI that can have profound effects on the perceived performance of a device.

To get to the bottom of performance issues, we must define metrics for measuring performance that we can use to reasonably compare one UI to another. We also need to analyze differences and similarities between various UI designs, so we can determine their relative complexity.

The central performance trade-off is balancing complexity of the UI with capability of the hardware. A complicated, multilayered user experience simply requires higher capability in the hardware platform. Perhaps the most important factor in determining GUI performance is that it is render-bound—the more pixels that you have to render, the greater the performance cost. As straightforward as this trade-off seems, no easy formula exists for determining hardware requirements for a particular UI scenario, because those requirements can depend heavily on the choice of design for the UI.

Determining where performance bottlenecks occur can be very challenging in a rich UI. For example, the collection of all visual elements that you use in an application UI can be envisioned as a visual tree. This tree represents the accumulation of all visual elements that the application creates, whether basic or elaborate, in code or in markup. A larger visual tree places a heavier computational load on the system and takes longer to parse and load. Also, a rich UI is interactive. As the user exercises the UI, the visual elements change or move across the screen, and the reaction of the device to user inputs must be natural, rapid, and smoothly responsive. When more pixels move and change, again there is a higher computational load and the recalculation of pixels takes longer.

If the performance bottleneck is in parsing and loading the visual tree, rather than in the pixel recalculations, certain types of remedies are used to make improvements. The effects of changes to the UI design or implementation of the design must be well understood to improve the UI performance.

Performance Factors That Can Be Measured and Compared

Factors for the UI

We can define primary performance factors that are directly tied to the complexity of the UI design. These factors can also be somewhat affected by the specific implementation of the interface. There are two major performance measurements in every UI.

- Load time (the time it takes to render the first screen) depends on:
  - Number of UI elements in the design.
  - Types of UI elements in the design.
- Animation speed (the smooth and realistic appearance of movement on the screen) depends on:
  - Number of UI elements that move.
  - Size of the UI elements that move, or portion of the screen that is covered by moving elements.
  - Whether the UI elements are changing or oscillating as they move.

Factors for Hardware Capability

You can take advantage of hardware features by using specific implementation techniques; however, there is a baseline set of capabilities that affect and possibly limit the performance of any UI. The most important common denominator hardware attributes that directly impact UI performance are:

- Processor (CPU) speed and type
- Graphics processing unit (GPU) speed and capability
- System and graphics memory amount

Analysis of UI Complexity

The hardware platform that you use for your product constrains the complexity that can be built into the UI for your product. However, the increase in functionality of graphics hardware has spawned a number of new techniques; it is now possible to interactively execute graphic rendering algorithms that were traditionally considered too complex.

The area of user experience and human-centered design has benefitted from recent study and advancement. Analyze your product, using the results of this work, to determine the best possible experience for your consumers. Scenarios, derived from studying your customers' needs and expectations, and competitive products, help you to determine the level of complexity you must address to build your product. Determining what makes customers excited about your product helps you understand the components of your application that require the most focus. To achieve maximum impact on user perception, you can single out the critical areas of your UI and build richness and complexity for these areas.

Table 1 shows three basic levels of UI complexity, and enumerates the primary characteristics that define each level, from both the user and the designer perspective.

Table 1 - Levels of UI Complexity

Complexity	User experience	Designer experience
Premium	Premium shell experience Dense UI Rich animations Fluid look and feel Multilayered UI	Flexible graphic design Screen resolution high (> 800 × 600 pixels) Animations occupy large portion of the screen (> 25 percent) Complex animations with high frame rate (20+ fps) Heavy use of vectors and paths
High	Interactive shell experience Instant visual state changes Simple animations Click-response look and feel	Limited graphic design Screen resolution limited Animations of moderate size (< 20 percent) Simple animations with medium frame rate (15+ fps) Static icons All graphic effects offloaded to GPU
Standard	Simple shell experience Limited screen complexity No animations Static look and feel	Little graphic design Screen resolution very limited No animation (effects only) Bitmaps (pre-effected) used instead of paths

Analysis of Hardware Capability

Traditional hardware platforms support traditional UIs. Rendering, the conversion of graphic information to displayable pixels, is entirely CPU-bound on traditional hardware platforms. Rich or complex UIs demand innovative rendering techniques, which are more efficiently and rapidly performed on specially designed graphics hardware. The combination of specialized rendering methods and specialized graphics hardware is commonly called the rendering pipeline or graphics pipeline.

An increasing number of embedded device platforms are available with on-board GPUs designed to speed and improve the rendering process. GPUs incorporate custom microchips that implement graphics processing functions to add special effects to UI elements in a 3-D environment. GPUs implement vertex shaders for the UI elements, and they implement pixel, or fragment shaders, to compute color, depth, and other attributes on a per pixel basis. GPUs generally consist of multiple parallel processing units that are capable of executing rendering conversions on multiple graphic elements simultaneously.

Table 2 shows some recommendations for hardware capability for the three UI complexity levels described in Table 1 of the previous section.

Table 2 - Minimum Hardware Capability for Levels of UI Complexity

Complexity	Minimum Hardware CPU and GPU*	Minimum Hardware Memory Available
Premium	Without GPU, not recommended With GPU, dual 1 GHz processor	512 MB
High	Without GPU, dual 1 GHz processor With GPU, 1 GHz processor	256 MB
Standard	Without GPU, 550 MHz processor With GPU, 550 MHz processor	128 MB

* GPU with support for either **DirectDraw** or **OpenGL** 2.0.

Performance Improvement Solutions

Silverlight for Windows Embedded is a native code (C++) UI framework that delivers impressive graphics with optimized performance on embedded devices. Although it goes without saying that any device rendering graphics for its UI performs better with more memory and a faster processor, accelerating the rendering process is a key to achieving optimal performance results.

Rendering Process Architecture

The Silverlight for Windows Embedded rendering process is driven by the UI design and is performed in the CPU. As shown in Figure 1, it consists of two distinct, computationally intense phases:

- Rasterization turns each vector-based UI element into the pixel-based representation that the device screen can display. The renderer allocates a memory buffer of appropriate size to hold the bitmap image and places the UI element image into that buffer, pixel by pixel.
- Composition layers the memory buffer contents (bitmap images) on top of one another, taking into consideration opacity and transformations (such as scale, skew or translation). The resulting composed image is directly displayed on the device screen.

A typical screen may be composed of tens of UI elements, each of which may, in turn, be composed of even more UI elements. The rendering process interleaves rasterization and composition phases to efficiently utilize the available memory; the process will overlay an element into an existing buffer whenever possible.

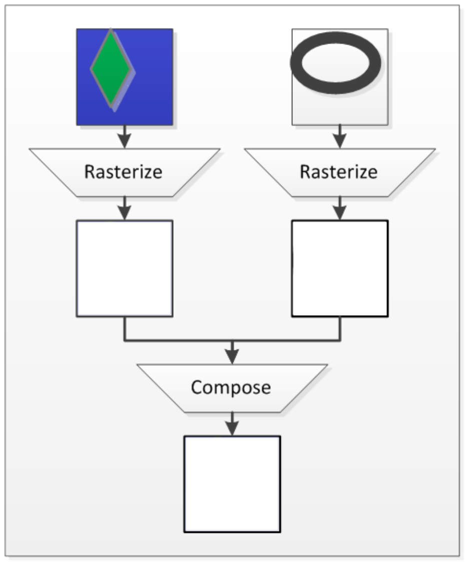

Figure 1 - Silverlight Rendering Process

When UI elements are animated, it wastes processor time to re-rasterize elements that are not moving. In addition, only the portions of the display that are affected by element movement need to be re-composed. To optimize the rendering process, Silverlight for Windows Embedded automatically defers composition of certain elements, so that the composition steps can be more rapidly completed by using a GPU.

Using a GPU to speed the rendering process is commonly referred to as hardware acceleration. If your device does not have a GPU or is otherwise incapable of hardware acceleration, Silverlight uses Graphics Device Interface (GDI) to draw UI objects pixel by pixel onto the primary display surface in z-order, with a corresponding lower rendering speed.

Hardware Acceleration Architecture

Hardware acceleration requires that you use one of the following architectures.

- GPU with an **OpenGL** interface that:
    - Supports **OpenGL Embedded Systems (ES) 2.0**.
    - Supports a simple vertex/fragment shader.
- GPU or video hardware with a **DirectDraw** interface that supports:
    - Per pixel and constant, premultiplied and hardware accelerated alpha blits.
    - Hardware accelerated blits on SCRCCOPY raster operations.
    - Hardware accelerated color fill.
    - 20-MB video memory (or system memory that the GPU can directly operate).

To support OpenGL for Embedded Systems (OpenGL ES) hardware acceleration in Silverlight for Windows Embedded Compact, use the architecture shown in Figure 2.

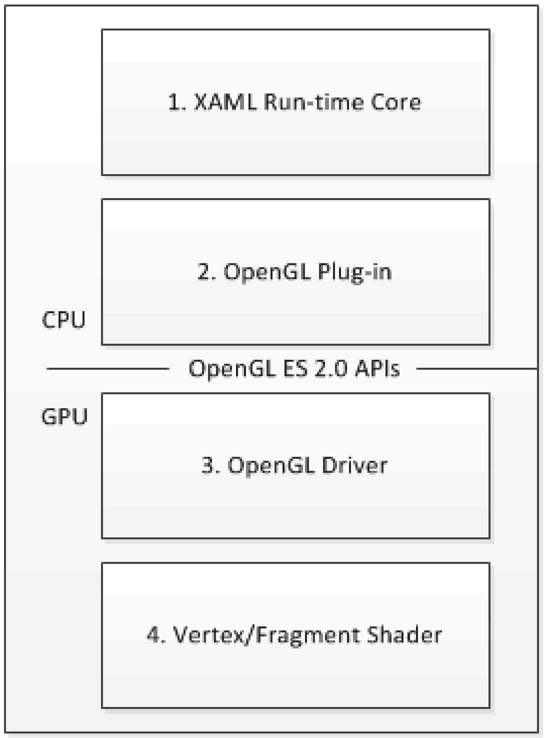

Figure 2 - Hardware Acceleration Architecture

1. **XAML Run-time Core**: This component is the software portion of the Silverlight rendering engine. This component works with the OpenGL Plug-in to provide acceleration.
2. **OpenGL Plug-in**: This component handles the interaction with the OpenGL driver. Silverlight contains a sample version of a plug-in that supports OpenGL ES 2.0. To customize the OpenGL (for example, to support OpenGL 1.2), you modify this component.
3. **OpenGL Driver**: The OGL driver is provided as a binary by the GPU provider and is specific to the chipset on the board support package (BSP).
4. **Vertex/Fragment Shader**: The vertex/fragment shader is about 25 lines of code that manages the interaction with the Open GL Plug-in. Silverlight includes sample code for shaders , and it must be compiled on the GPU into a Shaders.dll. Consult your hardware provider for instructions on compiling shaders for the target GPU.

> **Note** If Silverlight cannot find the Shaders.dll library for the GPU, it will compile the default shaders at run time. However, many OpenGL drivers will not support run-time compilation, and compiling the shaders at run time can result in poor performance.

Low-cost platforms, such as ARM9 or similar, typically have inadequate processor speed and lack GPUs. Using Silverlight on such devices can result in insufficient performance.

Performance Improvement Implementation

A simple example that illustrates how you can improve performance with a GPU is a static circle that moves around the interior of a static background. The optimal behavior is to rasterize both the background and the circle and save the resulting images. In this scenario, the pixel-based images for the circle and the background are stored on the GPU as textures and then, for each frame, they are composed by the GPU—the translation transform for the moving circle is changed slightly for each frame, creating the illusion of motion.

When Silverlight renders the example image for the first time, it does the following:

1. Sets up for rasterization by allocating memory buffer for background.
2. Rasterizes background object (walks through visual tree until circle object is encountered).
3. Sends rasterized background buffer to GPU (stores as a texture).
4. Rasterizes circle object (cached object).
5. Sends rasterized circle buffer to GPU (stores as a texture).
6. Composes the buffers (done in the GPU).

The GPU can compose textures very rapidly (and with trivial CPU usage), and the GPU can support a number of simple transformations. The most important are:

- Translation: changing the location of the object.
- Scaling: zooming in and out, to create the illusion of depth.
- Rotation: turning the object about a point or axis.
- Deformation: changing the skew or aspect ratio of the object.

The amount of video memory on the GPU determines the size and number of buffers that it can use for compositing graphics.

Load Time Improvement Examples

Silverlight for Windows Embedded Compact 7 includes features that are specifically designed to improve load time. Support for compiling XAML into Binary XAML (BAML) substantially shortens the time needed for a running application to load and display a new screen. Performance testing shows that load time improvements range from 27 percent to 38 percent by using this approach. Figure 3 illustrates some examples of load time decreases for standard and high category UIs. The times shown in Figure 3 were obtained by using two different minimal hardware platforms.

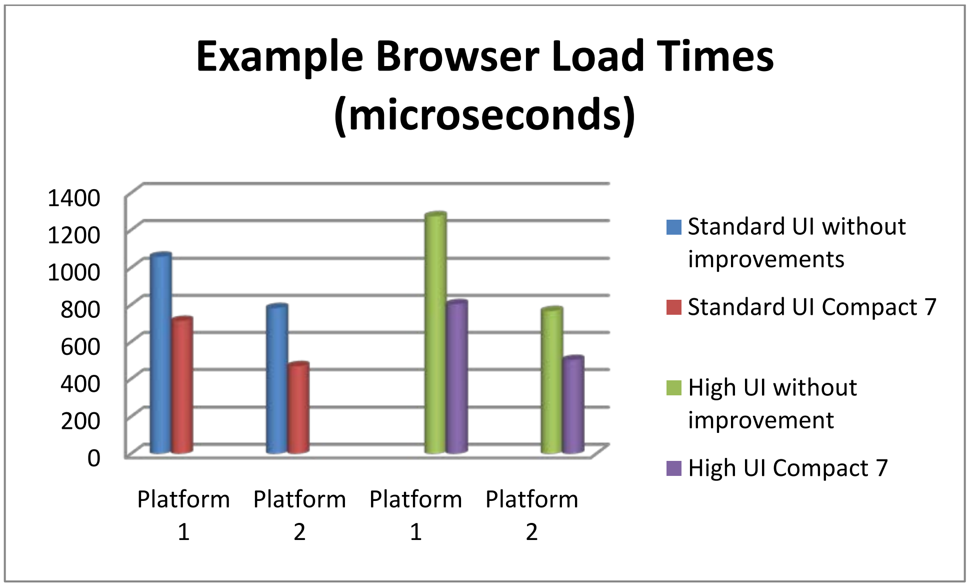

Figure 3 - Browser Load Examples

Silverlight for Windows Embedded now supports encoding Portable Network Graphics (PNG) files in a less-compressed format so that they can be read into memory faster. Specific test examples speed up this process by 24 percent to 40 percent, depending on the hardware platform and image characteristics. Load time improvements for two image types are shown below: graphic images, which are constructed manually and typically used as building blocks of UIs, and photo images, which are generated by a camera and typically used as shell backgrounds, or manipulated directly in applications. Figure 4 shows load times of PNG images of two sizes, 800 × 640 (512,000) pixels and 4 megapixels (MP).

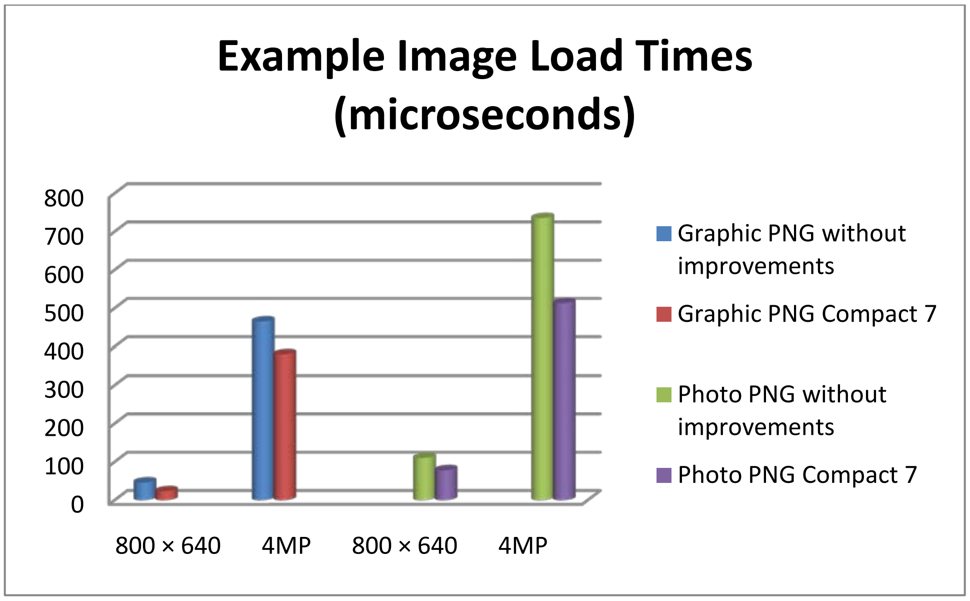

Figure 4 - Image Load Examples

Animation Speed Improvement Examples

Silverlight for Windows Embedded Compact 7 implements automatic support for hardware acceleration. Cached composition now occurs automatically; developers no longer specify it manually. Silverlight selects appropriate UI elements and applies cached composition to them to gain animation speed (performance tests show improvements on the order of 4 to 6 times faster than without automatic composition caching) when used in conjunction with hardware acceleration. Figure 5 shows two examples (wizard and music player) of animation speed obtained on a static background display with full screen sliding transition responses to pan and flick gestures, and two examples (panel and keyboard) of animation speed on an animated background with sliding transition responses to pan and flick gestures. These results were obtained on minimal hardware platforms both with and without OGL-capable GPUs.

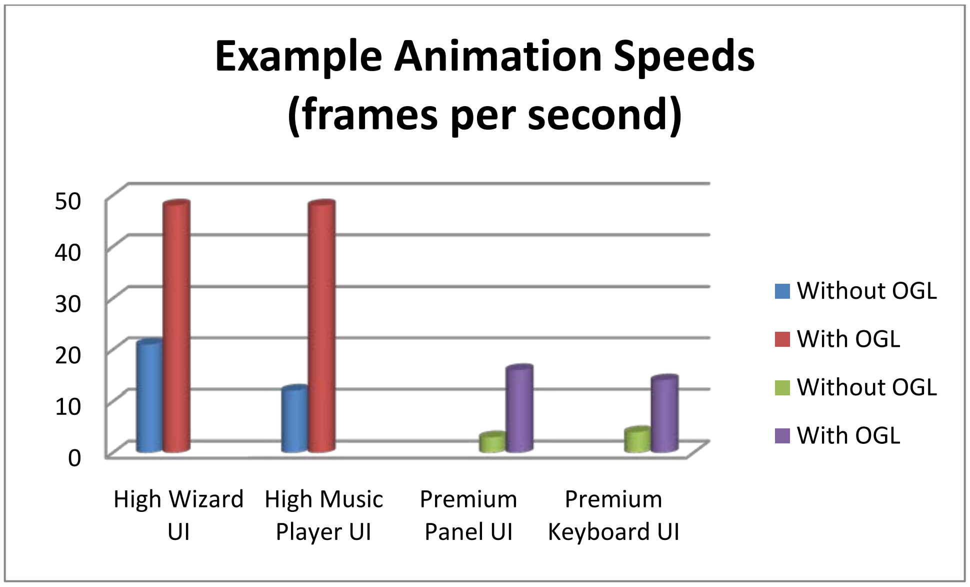

Figure 5 - Animation Speed Examples

Key Implementation Considerations for Improving UI Performance

Load time is significantly improved in Silverlight for Windows Embedded Compact 7, mainly by modifications to compilation of BAML and PNG files for loading graphic or photographic images. BAML files are smaller and faster to read than the original XAML files, which lowers load time. The following practices can increase the gain of these improvements:

- Load image assets from resource binaries.
- Create controls for self-contained elements; invoke controls only when necessary. Examples include History/Favorites, Zoom options panel and status bar elements.
- Use images instead of controls when possible. Check whether buttons can be represented with images and whether to use stack panels or list boxes.
- Use images instead of paths when possible; use a minimal number of precision points in paths.
- Replace storyboard elements with visual states.
- Share styles when appropriate. For example, if you want all buttons in your design to look the same, you can define a button style and apply it to all of the buttons.
- Prune unnecessary visual state groups and empty or default values out of the XAML. Examples are: <Storyboard/> and Stretch="Fill".
- Design for minimal use of screen real estate. For example, use visibility = collapsed instead of opacity = 0.
- Specify target types for styles and for control templates that are inside resources.

Animation speed is made faster in Silverlight for Windows Embedded Compact 7 by automatically invoking cached composition. The most effective way to improve animation is to use faster hardware or to use a less complex UI. Some other effective ways to improve animation are:

- Limit large-scale animations to use transformations that can be hardware accelerated. For example, use a translation transformation, rather than modifying x and y positions of an object.
- Structure the XAML to improve buffer usage.

Windows Embedded Compact 7 Compact Test Kit (CTK) includes sample performance tests to help you analyze how applications perform on your hardware platforms. These samples can assist you in determining how to maximize performance of your UI.

Conclusion

The information presented here can help you systematically and rationally improve the UI performance of your embedded devices. Load time and animation speed are the main factors that determine how well a UI performs. Higher UI complexity negatively affects both of these performance factors, but increased capability of modern hardware platforms mitigates some of these effects. The latest release of Silverlight for Windows Embedded Compact 7 includes significant improvements in base performance of more complex UIs, and includes sample performance tests to help you determine the performance of your UI on your hardware platform.

Additional Resources

- Microsoft Silverlight for Windows Embedded (http://go.microsoft.com/fwlink/?LinkId=192016)
- Silverlight for Windows Embedded Reference (http://go.microsoft.com/fwlink/?LinkId=192017)
- Microsoft Silverlight (http://go.microsoft.com/fwlink/?LinkId=192018)
- Differences Between Silverlight for the Web and Silverlight for Windows Embedded (http://go.microsoft.com/fwlink/?LinkId=192019)
- Windows Embedded (http://go.microsoft.com/fwlink/?LinkId=192020)

Copyright