



## **Silverlight for Windows Embedded Graphics and Rendering Pipeline**

**Windows Embedded Compact 7 Technical Article**

Writers: David Franklin, Frankie Anderson

Published: March 2011

Applies To: Windows Embedded Compact 7

### **Abstract**

This document gives an overview of the Silverlight for Windows Embedded rendering process. We explain how the rendering process supports hardware acceleration and which system and UI circumstances benefit from this acceleration. It uses a simple example UI to illustrate in detail how the process works, on either a non-accelerated or a hardware-accelerated platform. The document also describes the components and architecture needed for hardware acceleration, and gives some guidelines for UI designers and developers to maximize the acceleration benefits.

# Introduction

This document gives an overview of the Silverlight for Windows Embedded rendering process and demonstrates, using a simple example, how hardware acceleration improves the animation performance of a UI. Performance tuning and hardware acceleration work normally occurs late in a UI project, but there are design considerations that can affect UI performance, that are known at initial stages of the project.

The Silverlight for Windows Embedded rendering process consists of two phases:

- **Rasterization:** Each UI element is drawn into a memory buffer. Vector elements are translated into bitmap images. Rasterization is handled purely within the CPU.
- **Composition:** The memory buffers are layered on top of one another, taking opacity and any transformations (such as scale or translation) into consideration. Composition can be handled by either the CPU or the graphics processing unit (GPU).

The designer or developer can use cached composition to specify how the particular UI elements are to be grouped and to be composed by the GPU. Appropriate caching can dramatically increase UI frame rate.

## Silverlight Rendering Process

The Silverlight for Windows Embedded rendering process consists of two phases, rasterization and composition. Rasterization is the process of converting vector-based graphics (such as lines, text and rectangles) into the pixel-based representations that can be directly displayed on a screen. The renderer allocates a memory buffer that serves as a bitmap image and rasterizes the vector-based graphics into that bitmap image, pixel-by-pixel. The renderer rasterizes each UI element of a graphics screen into a bitmap image and then composes the individual bitmap images together, producing the image that is ultimately displayed on the device screen.

This process is complicated because the image on the screen is assembled from a set of UI elements, many of which are composed of other UI elements. Internally Silverlight uses a visual tree, which is a hierarchical structure of all the elements that are visible, and that must be rendered. To render a typical screen, the process steps through the elements in the visual tree, and interleaves rasterization and composition to reduce the number of memory buffers that must be allocated.

There are two main threads in Silverlight: the UI thread and the compositor thread. To create a responsive UI, it is important to understand their roles.

The UI thread handles the following tasks:

- Processing user input
- Parsing and creating objects from XAML
- Drawing all visuals, the first time that they are drawn
- Executing per-frame callbacks and other user code

To make your application as responsive as possible, you must keep the UI thread as lightweight and free as possible. The compositor is an ultra lightweight thread that simply combines textures for the GPU; it drives the types of animation that are defined

as storyboards and drawn by manipulating base textures on the GPU. These animation types include:

- Translation – to change the location of the object
- Scaling – to zoom in and out and to create the illusion of depth
- Rotation – to turn the object about a point or an axis
- Deformation – to change the skew or aspect ratio of the object
- Plane Projection – to represent the object by mapping it to a two-dimensional plane

Note that if any of these animations use an opacity mask, a non-rectangular clip, or a texture size for an animated object greater than 2k × 2k pixels, they will be rasterized on the UI thread.

For the remainder of this article we will analyze the example UI pictured in the screen shown in Figure 1. The globe image on the screen is animated and will move around the screen, remaining completely inside the screen boundaries. This screen shows a very simple UI, but it includes all of the key building blocks needed to understand the rendering process.



**Figure 1 - Silverlight Example Screen**

The screen in Figure 1 is created using XAML to define a user control. The MainPage.xaml file, which defines the Hello World user control shown in Figure 1, contains the following code block:

```

1:  <UserControl
2:      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:      x:Class="HelloWorld.MainPage"
5:      Width="300" Height="200">
6:
7:      <Grid x:Name="LayoutRoot">
8:          <Grid x:Name="_Background" Background="{StaticResource BlueWashBrush}">
9:              <Image x:Name="_LightHexes" Source="Assets/Hexes.png" Stretch="Fill"/>
10:             <Image x:Name="_DarkHexes" Source="Assets/BlackHexes.png" Stretch="Fill">
11:                 <Image.OpacityMask>
12:                     <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
13:                         <GradientStop Offset="0"/>
14:                         <GradientStop Color="#C0FFFFFF" Offset="0.6"/>
15:                     </LinearGradientBrush>
16:                 </Image.OpacityMask>
17:             </Image>

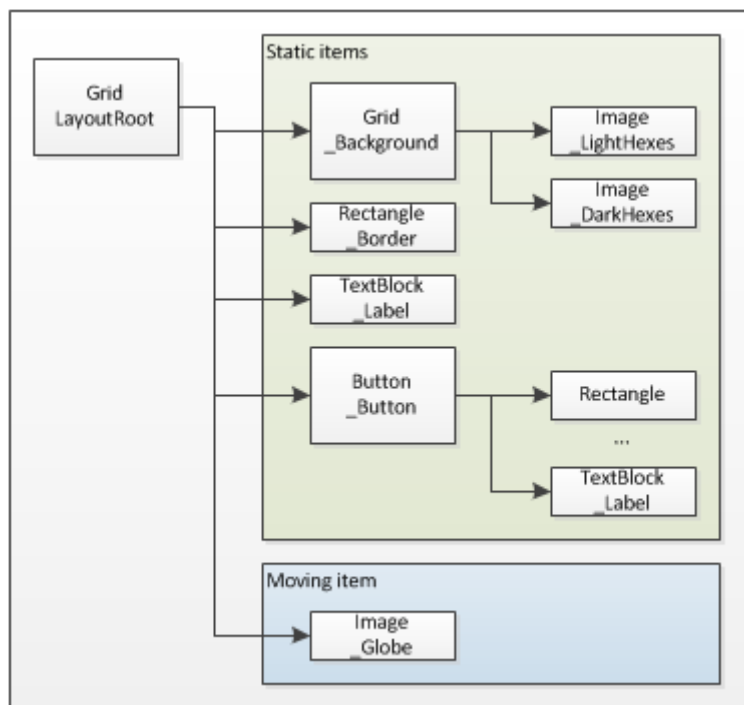
```

```

18:         </Grid>
19:         <Rectangle x:Name="_Border" Fill="#FF969696" Stroke="White"
Margin="0,10,90,10" RadiusX="10" RadiusY="10" HorizontalAlignment="Center"
Width="180" Opacity="0.5"/>
20:         <TextBlock x:Name="_Label" HorizontalAlignment="Center" Margin="0,0,90,15"
VerticalAlignment="Bottom" FontFamily="Verdana" FontSize="21.333" Text="Hello
World" TextWrapping="Wrap" FontWeight="Bold"/>
21:         <Button x:Name="_Button" HorizontalAlignment="Right" Margin="0,0,8,8"
VerticalAlignment="Bottom" Width="75" Content="Close" FontFamily="Verdana"
FontSize="16"/>
22:         <Image x:Name="_Globe" Height="138" HorizontalAlignment="Center"
Margin="0,20,90,0" VerticalAlignment="Top" Width="138"
Source="Assets/Globe138x138.png" Stretch="Fill" RenderTransformOrigin="0.5,0.5"
CacheMode="BitmapCache">
23:             <Image.RenderTransform>
24:                 <TransformGroup> <ScaleTransform/> <SkewTransform/> <RotateTransform/>
<TranslateTransform/> </TransformGroup>
25:             </Image.RenderTransform>
26:         </Image>
27:     </Grid>
28: </UserControl>

```

The visual tree for the Hello World user control specified by this code is shown in Figure 2.



**Figure 2 - Silverlight Example Visual Tree**

The Silverlight for Windows Embedded rendering process uses the following steps to render the visual tree illustrated in Figure 2:

1. Allocate main buffer for LayoutRoot.
2. Rasterize the gradient brush for the \_Background grid into the main buffer.

3. Rasterize `_LightHexes` into a temporary buffer and compose with the main buffer.
4. Rasterize `_DarkHexes` into a temporary buffer and compose with the main buffer.
5. Rasterize the `_Border` rectangle into a temporary buffer and compose with the main buffer.
6. Rasterize the `_Label` text into a temporary buffer and compose with the main buffer.
7. Allocate a temporary buffer for `_Button` and then rasterize and compose all of the `Button` elements (`Rectangle ... _Label`) into this temporary buffer.
8. Compose the `_Button` buffer with the main buffer.
9. Compose `_Globe` with the main buffer.

Note that as the renderer executes the preceding steps, it has the following important characteristics:

- It allocates buffers only as necessary. If it is possible to overlay an element into an existing buffer, it will generally do so.
- It uses the CPU to perform all of the rasterizing and composing operations.

During animation, as the `_Globe` element moves around the screen, none of the other elements change. It is not necessary to perform all of the rasterization and composition of steps 1 through 8 for each frame of the animation. The optimal behavior is to save the results of steps 1 through 8 and then, for each frame of the animation, compose `_Globe` with those saved results. The `CacheMode` attribute, shown at the end of line 22 in the preceding code block, explicitly tells the renderer to treat the `_Globe` element individually, producing the desired results.

In Silverlight for Windows Embedded caching is automatic, and it is not necessary to specify the `CacheMode`. Note that ordering the XAML to group background items and separate cached items is important for the success of the caching algorithm and, ultimately, the animation speed.

## Hardware Acceleration

Many modern device platforms include on-board graphics processing units (GPUs) with 2-D and/or 3-D capabilities. Silverlight for Windows Embedded provides support for using a GPU to accelerate certain types of animations. Hardware acceleration is accomplished by using the GPU (rather than the CPU) to do some critical composition steps in the rendering process.

Silverlight for Windows Embedded supports hardware-based acceleration of graphics for both Microsoft `DirectDraw` and `OpenGL`.

`DirectDraw` is older technology that has some limitations on the types of transformations it can support. `DirectDraw` supports translation, scaling from 50 percent to 200 percent, and rotation by 90 degrees.

The technique used by `OpenGL` technology is to split each pixel-based image into two triangular pieces and store them in the GPU as textures. (GPUs typically use textured triangles as the building blocks for rendering 3-D objects.) During composition, the two triangles are drawn as a "triangle strip," creating a rectangular shape on the screen, as shown in Figure 3.

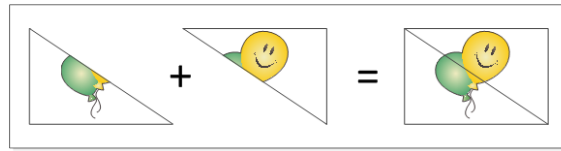


Figure 3 - Triangle Strip

The GPU can compose these triangle strips very rapidly (with trivial CPU use), and the GPU supports a number of simple transformations. The most important were listed earlier in the description of the compositor thread in [Silverlight Rendering Process](#) and they are translation, scaling, rotation, deformation, and plane projection.

Figure 2 uses color shading to show that the UI elements in the visual tree for our example can be divided into two sets:

- The static items that define the background for the animation (light green)
- The moving globe (light blue)

The pixel-based images for each of these sets are stored on the GPU as textures and then, for each frame of the animation, they are composed by the GPU; the translation transform for the moving globe is changed slightly for each frame, creating the illusion of motion.

## Silverlight Rendering with Hardware Acceleration

To support OpenGL hardware acceleration in Silverlight for Windows Embedded, your board support package (BSP) must contain an OpenGL ES 2.0 driver, a simple vertex shader, and a simple fragment shader. For information on how to implement hardware acceleration, see Implement Hardware Acceleration for Graphics in Silverlight for Windows Embedded in the [Silverlight for Windows Embedded Developer's Guide](http://go.microsoft.com/fwlink/?LinkID=204075) (<http://go.microsoft.com/fwlink/?LinkID=204075>) on MSDN. For details about using shaders, see Adding Support for Binary Shaders to the BSP in the [Performance Tuning Guide for Silverlight for Windows Embedded](http://go.microsoft.com/fwlink/?LinkID=204076) (<http://go.microsoft.com/fwlink/?LinkID=204076>) on MSDN.

Continuing with our simple UI example, the Silverlight renderer performs the following steps when it renders the image for the first time:

1. Set up for rasterization.
2. The rendering engine asks the OpenGL plug-in for a buffer that is the size of the display window.
3. Rasterize the first set of non-cached items.
4. The rendering engine steps (in z-order) through the visual tree until a cached object is encountered. It rasterizes and composes (using opacity information) each object into the buffer obtained in step 1. Note that this is all done in the CPU. In our example, the renderer processes objects in the following order: `_Background`, `_LightHexes`, `_DarkHexes`, `_Border`, `_Label`, and `_Button` (and its children).
5. Send the buffer to the GPU.
6. When all of the items in the first set have been processed, the buffer is marked as dirty (indicating that it needs to be refreshed on the screen) and sent to the GPU, using the appropriate calls to the OpenGL Plug-in.

7. Rasterize the first cached item.
8. When the rendering engine reaches a UI element that is cached, it asks the OpenGL plug-in for a buffer that is the size of the element. Then, the rendering engine steps through the visual tree for the cached element, rasterizing each object into the buffer. In our example, `_Globe` does not have any children, so just the single item is rasterized to the buffer.
9. Send the buffer to the GPU.
10. When everything has been processed for the cached UI element, the buffer is marked as dirty and sent to the GPU.
11. Compose the buffers.
12. In our example, there are now two buffers in the GPU. Note that in a larger or more complicated example there would be many buffers. Each of these buffers is stored as two triangular textures. Starting at the bottom of the z-order, the GPU composes the textured triangles corresponding to the buffers from step 3 and step 5.

During each frame of animation, if the XAML that corresponds to one of the buffers does not change, that buffer doesn't need to be redrawn. The XAML run-time engine keeps track of animation changes. For a cached item, simple transformations (translate, scale, rotate, and skew) and opacity changes, which are applied to the cached object as a whole, do not require it to be redrawn.

## Design Considerations

When designing a UI that will take advantage of GPU-based hardware acceleration, use the following guidelines:

- **Use UI elements that are optimized for hardware acceleration.**  
See Using UI Elements in your Application that are Optimized for Hardware in the [Performance Tuning Guide for Silverlight for Windows Embedded](http://go.microsoft.com/fwlink/?LinkID=204076) (<http://go.microsoft.com/fwlink/?LinkID=204076>) on MSDN.
- **Limit the XAML UI to a maximum size of 2048 × 2048 pixels for OpenGL applications.**  
OpenGL has a limitation that surfaces cannot be larger than 2048 × 2048. If you attempt to create an OpenGL surface that is larger than 2048 × 2048 pixels, it will appear as a black rectangle on the screen. Silverlight for Windows Embedded will not automatically split a large surface into multiple smaller ones, so it is important to keep this limitation in mind when designing for very large screens.
- **Limit large-scale animations to use transformations that can be hardware-accelerated.**  
Silverlight permits many attributes to be animated, but only a few of these can be hardware-accelerated. Examples of animations that cannot be accelerated include repositioning an item by changing margins and animating gradients and colors. Performing these types of animations on a small portion of the screen may not cause any performance difficulties but, for example, animating a screen-sized radial gradient is almost certain to overtax the CPU. Note that if a UI element is changing during an animation, it will be repeatedly rerendered, negating the benefits of hardware acceleration.
- **Structure the XAML to improve buffer use.**  
Following are some examples to illustrate this guideline:



- If there is a set of static elements that serve as a background, group them together and put them at the start of the XAML, so that they will all be placed in the same buffer in the GPU.
- Make sure that dynamic items don't "pollute" a set of static elements. For our example, if we want to have the button draw attention by gradually changing its brightness, we want to make sure that it isn't in the same buffer as the static background items. Otherwise, each brightness change would require that the entire background be rasterized and composed again.
- Don't unnecessarily create extra buffers. Buffers use memory in both the CPU and GPU, so extra buffers must not be created without reason. In our example, `_Globe` is placed at the end of XAML and only two buffers are needed. But, if `_Globe` was placed earlier in the XAML, the background items would be rendered in two separate buffers, meaning that three buffers would be used.
- **Use the Visibility or Opacity properties to efficiently hide or display elements in your UI.**

While an object is hidden (visibility = collapsed), Silverlight will not hold visual data for the object in video memory, will not walk the tree in the render pass, and will not do any hit-test work. When the object is visible again (visibility = visible), the entire tree content has to be drawn again (re-layout). If, and only if, the object is cached, setting opacity to zero to hide the object causes Silverlight to hold a texture for that tree in video memory without affecting the performance of the compositor. When the object is visible again (opacity > 0) it will be processed normally without redrawing the content. You will likely obtain the best results by manipulating the opacity property, but with multiple or complex trees, using the visibility property may be the best approach. Try both approaches and see which works best for you.

## Conclusion

This document contains an overview of the Silverlight for Windows Embedded rendering process, which consists of two phases:

- Rasterization, which is the translation of vector-based UI images into bitmap images. This phase is handled purely within the CPU.
- Composition, which is layering bitmap images over each other, taking opacity and transformations into account. The composition phase is handled by either the CPU or the GPU.

A simple example demonstrates how the rendering phases are interleaved and how hardware acceleration improves the animation performance of a UI.

The designer or developer can specify, using definition order and cached composition, how the UI elements are to be grouped and be composed by the GPU, and how this can dramatically increase UI animation speed, particularly in situations that use hardware acceleration.

## Additional Resources

- [Improving Performance in Rich User Interfaces for Embedded Systems](http://go.microsoft.com/fwlink?LinkID=197420) (<http://go.microsoft.com/fwlink?LinkID=197420>)
- [Silverlight for Windows Embedded](http://go.microsoft.com/fwlink/?LinkId=192016) (<http://go.microsoft.com/fwlink/?LinkId=192016>)



- [Silverlight for Windows Embedded Reference](http://go.microsoft.com/fwlink/?LinkId=192017)  
(http://go.microsoft.com/fwlink/?LinkId=192017)
- [Microsoft Silverlight](http://go.microsoft.com/fwlink/?LinkId=192018) (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)  
(http://go.microsoft.com/fwlink/?LinkId=192019)
- [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=183524) (http://go.microsoft.com/fwlink/?LinkId=183524)
- [Silverlight for Windows Embedded Developer's Guide](http://go.microsoft.com/fwlink/?LinkID=204075)  
(http://go.microsoft.com/fwlink/?LinkID=204075)
- [Performance Tuning Guide for Silverlight for Windows Embedded](http://go.microsoft.com/fwlink/?LinkID=204076)  
(http://go.microsoft.com/fwlink/?LinkID=204076)

## Copyright

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.