



Window Compositor Graphics Performance Tuning Guide

Windows Embedded Compact 7 Technical Article

Writer: Jay Treptow

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

When a Windows Embedded Compact device runs multiple applications simultaneously, with each application having its own user interface (UI), the OS must minimize the number and size of display updates to conserve processing resources. However, designing an OS that minimizes the frequency of display updates and the area affected by each update can be a challenging task for OEMs and Windows Embedded Compact developers.

To help with this task, Windows Embedded Compact includes a module named Window Compositor that performs window composition. Window composition reduces the number of required display updates by merging the visual output from one or more applications before Windows Embedded Compact updates the display screen. You can include window composition in your OS design and also use the additional techniques described in this article to provide faster, more efficient display updates.

Introduction

On Windows Embedded Compact devices, multiple applications can run simultaneously, but only a single screen is available to display all program user interfaces (UIs) and output. Because graphics output uses significant processor and memory resources, original equipment manufacturers (OEMs) and application developers must design systems that minimize this overhead, while offering an optimal user experience where one or more applications simultaneously create graphical output.

To help OEMs and developers with this optimization process, Windows Embedded Compact offers a module known as Window Compositor that can be included in the OS. Window Compositor improves graphical performance through a technique known as *window composition*. This technique reduces the number of required display updates by merging the visual output from one or more applications before Windows Embedded Compact updates the display screen. For instructions on adding Window Compositor to an OS, see [Window Compositor Developers Guide](#).

In addition to optimizing graphics output performance, OEMs and developers must consider how their applications use the output display for Windows Embedded Compact, in which output windows from multiple applications can share the same display. Typically, OEM products contain a shell or a human machine interface (HMI), which is responsible for presenting options to the user and responding to user input. On many HMIs, users, over time, add and run applications that are provided by independent software vendors (ISVs). Therefore, any third-party application that an OEM includes in an OS, ideally, is designed so that it shares the screen with other applications and does not rely on the exclusive use of the screen.

Assuming OEMs and application designers have designed their programs so that they can share the output screen with other programs that might be running, the following programming guidelines can help OEMs and developers improve the performance of graphics output for their own and ISV applications for Windows Embedded Compact.

When Performance Improvements Are Not Required

The suggested Window Compositor performance improvements in this article are not required in the following scenarios.

- All applications use Silverlight for Windows Embedded exclusively for all graphics output and for output from the shell or HMI. This means that at least one of the following conditions applies.
 - All the applications that produce graphical output are created with XAML and use Silverlight for Windows Embedded to render the XAML to the display.
 - No applications from ISVs produce graphical output that must be composed into the shell or HMI.

In these two cases, Silverlight for Windows Embedded can efficiently compose the display output.

- An OEM does not include support for Window Compositor in the OS. An OEM omits this feature in the following cases:

- The system that the OEM is developing has serious graphics performance issues. In this case, OEMs may want to avoid the added memory overhead of including Window Compositor as part of the OS.
- The OEM has determined that the system does not require alpha blending to compose windows. Alpha blending makes windows that overlay other windows wholly or partially transparent, and Window Compositor uses alpha blending to compose windows. Although alpha blending can improve the user experience of the visual UI, it requires additional processing overhead that can negatively affect graphics performance. Therefore, OEMs must consider the tradeoff between improved user experience and improved system performance before including Window Compositor and alpha blending in the system.

Optimizing Window Display

If a device uses both Silverlight for Windows Embedded and Win32 for graphics output, we recommend that OEMs and application developers create windows that are optimized for Window Compositor.

When you add Window Compositor to an OS design, Window Compositor renders both Win32 windows and Silverlight windows to the display screen by using alpha blending and window composition. You can adjust the composition settings by calling Window Compositor APIs on the window handle (**HWND**) that you obtain from a call to **CreateWindow**, **CreateWindowEx**, or **IXRVisualHost::GetContainerHWND**.

With alpha blending, Window Compositor composes Silverlight and Win32 windows on the display screen by using a visual style that implements semi-transparency in overlapping window regions to improve the visual appearance. Applications can implement alpha blending either on a pixel-by-pixel basis or for an entire window region.

If your application creates windows, you must consider graphics performance in the parts of the application that create and configure windows. To achieve optimum graphics performance, you must disable alpha blending for the following types of windows.

- Windows for an application such as a shell, which occupy the full screen.
- Windows that have only animation or movement, such as a video player or full-screen picture viewer.

To remove alpha blending on a pixel-by-pixel basis

- Call [SetWindowCompositionFlags](#) and do not include the **WCF_ALPHATRANSARENCY** flag in the *dwFlags* parameter.

To remove alpha blending for an entire window region

- Call [SetWindowOpacity](#) with a value of **255** for the *bOpacity* parameter to create an opaque window.

For more information about which Windows Embedded Compact APIs to use to configure window composition, see **SetWindowOpacity** and **SetWindowCompositionFlags** in the Windows Embedded Compact documentation. For more information about how to add Window Compositor to an OS, see the section [Include Compositor in an OS Design](#) in the [Window Compositor Developers Guide](#).

Creating Clipping Regions

When you create scenes that show simultaneous output from multiple applications, consider creating clipping regions. A clipping region draws a boundary around a region and tells Window Compositor not to perform alpha blending within the region. Clipping regions improve graphics performance by reducing the display area that must be updated each time the system refreshes the display.

For example, you can create a clipping region for a progress bar that occupies the bottom 10 percent of the display in a music player application window (with a static image that occupies the other 90 percent of the display). If you create a clipping region for the progress bar, the entire display does not need to be updated each time the UI for the progress bar is updated.

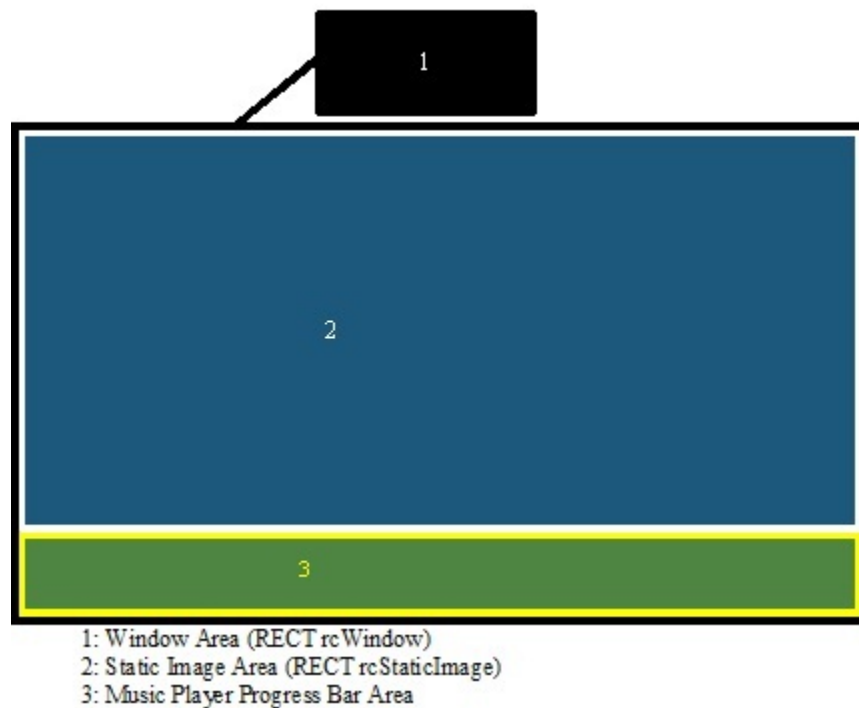


Figure 1 - Static Image Clipping Region

The following sample code shows how to create a clipping region.

```
RECT rcWindow = { 0 };  
HRGN hrgnWindow = NULL;  
RECT rcStaticImage = { 0 };  
HRGN hrgnStaticImage = NULL;  
  
// Get the full window area  
GetWindowRect( hwnd, &rcWindow );  
// Create a full window area region  
hrgnWindow = CreateRectRgnIndirect( &rcWindow );
```

```
// Define the static image area rectangle
rcStaticImage.left = rcWindow.left;
rcStaticImage.top = rcWindow.top;
rcStaticImage.right = rcWindow.right;
rcStaticImage.bottom = rcWindow.bottom * .9;
// Create the static image clipping region.
hrgnStaticImage = CreateRectRgn( rcStaticImage.left,
                                rcStaticImage.top,
                                rcStaticImage.right,
                                rcStaticImage.bottom);

// Create a new window region which excludes the static image area
CombineRgn( hrgnWindow, hrgnWindow, hrgnStaticImage, RGN_DIFF );
// Permit drawing to the progress bar area only
SetWindowRgn( hwnd , hrgnWindow, TRUE );

// After setting the window region, delete the static image
// region but not the window region since GWES now owns it.
if ( NULL != hrgnStaticImage )
{
    DeleteObject( hrgnStaticImage );
}
```

Conclusion

Windows Embedded Compact application developers and OEMs must consider how to improve graphics performance on devices to optimize the user experience of applications. One way to achieve this improvement in devices that use both Silverlight for Windows Embedded and Win32 for graphics output is to include Window Compositor in the OS and use the following techniques to optimize applications for Window Compositor:

- Disable full window and/or pixel-by-pixel alpha blending for full-screen or animation-only windows.
- Create clipping regions to restrict screen updates to areas that do not use alpha blending.

These techniques minimize the number of display updates Window Compositor must perform, thereby improving user perceptions of application performance and providing an improved user experience for applications.

Additional Resources

- [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=183524) (<http://go.microsoft.com/fwlink/?LinkId=183524>)

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.