Layered Windows

Article06/30/2010

Vadim Gorokhovsky and Lou Amadio
Microsoft Corporation

January 2000

Summary: Describes the new functionality included in Microsoft® Windows® 2000 that provides an efficient way to add transparency and translucency to top-level windows. (8 printed pages)

Contents

Introduction

Microsoft® Windows® 2000 includes several improvements that will heighten the end-user experience through higher quality and a spiced-up UI while making the system easier to use. If you installed a beta build of Windows 2000, you may have already noticed some of these changes. They include an alpha-blended cursor with a shadow, new transition effects including menu and ToolTip fade-in, menu selection fade-out, and an alpha-blended image drag in the shell. All of these effects were implemented by using the new functionality that Windows 2000 introduces, called *layered windows*.

Most end users expect smooth transition effects. It's not natural for information to just pop right in your face. Television does a great job of using fades and slides to give a context of where the new information is going to appear. Computers haven't yet been able to incorporate these effects into the UI very effectively. Just think what a difference there is between the existing UI and the cool UI you constantly see in the movies. Layered windows give the product designers a lot of power to bring "cool" UI closer to reality.

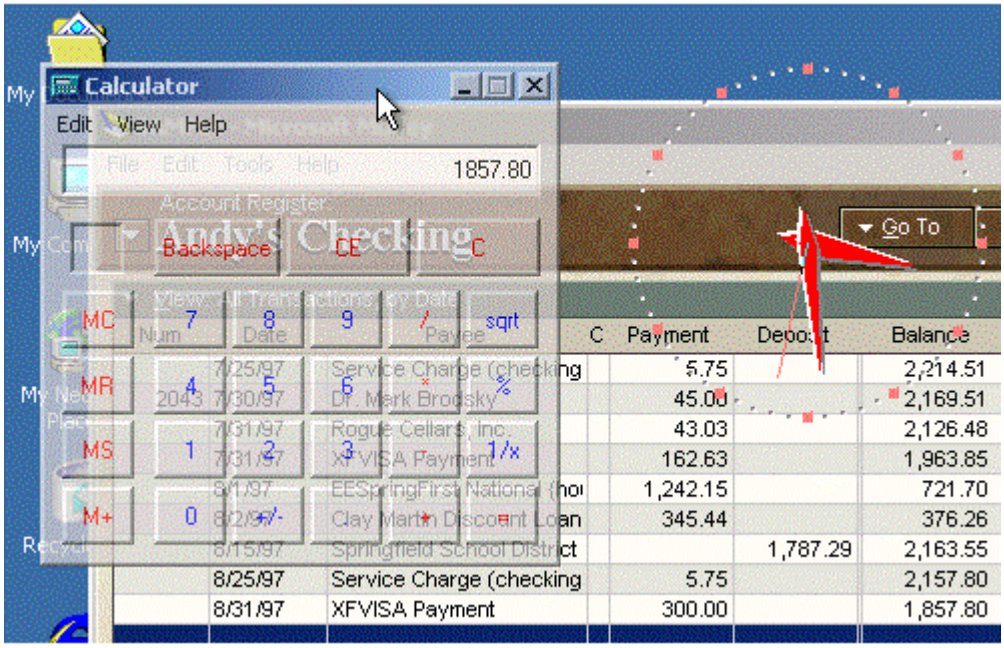

Figure 1. Translucent Calculator and Clock with transparency on top of Microsoft Money help to manage the screen real estate.

Some subtle changes can go a long way in improving the UI. For example, an alpha shadow gives the cursor depth and makes it stand out better on the screen. So, on a large monitor or a multi-monitor system the cursor will be easier to find. Or, take the problem of the end user who makes a menu selection and the menu is subsequently dismissed. Fading out the selected item will give the end user the necessary visual feedback to let them know the right item was selected and non-intrusively fill the time period between the click and the time the actual action caused by the click takes place.

In addition, the desktop real estate is quite limited. While larger monitors and multi-monitor systems help a little, well-designed UI that takes advantage of transparency and translucency can help alleviate this problem. For example, an alpha-blended see-through Outlook reminder pop-up window could be a lot less intrusive and allow the end user to continue his or her work without being dismissed. Or the system could use translucency and transparency to display e-mail and other notifications on the screen without distracting the end user with a pop-up window.

Figure 2. Microsoft Agent in Windows 2000 is finally out of the "penalty box," without penalizing system performance the way regional windows did.

The other area where the quality of user experience can be improved is in animating windows. Previously, the animation had to either be constrained by a window rectangle, a "penalty box," or the shape of the animation frame had to be described to the system by a region, which for animated windows led to performance problems associated with frequent repaints in underlying windows. In Windows 2000, the Microsoft Agent is finally out of the "penalty box," without much loss to system performance.

Layered Windows

Windows 2000 introduces a new extended window style bit: WS_EX_LAYERED. When used properly, it can significantly improve performance and visual effects for a window that has a complex shape, animates, or wishes to use alpha-blending effects. The full implementation of layered windows was publicly available for the first time in Windows 2000 Beta 3.

Windows appear as rectangles on the screen clipped by other windows. For a window in an application to look like a circle, it's not enough for the application to simply paint a window as a circle; the system will continue to hit test this window as a rectangle and windows underneath this window will still be clipped by the window's rectangle. So, the window will look and behave like a gray rectangle with a circle in the middle.

Some applications might take a snapshot of the visual bits underneath the window before it was actually shown and later compose those bits with the window bits. This approach doesn't quite work in a multiprocess, multitasking environment, because other windows can paint underneath this window. The application has no way of knowing when such painting occurs or how to somehow retrieve the newly painted bits underneath.

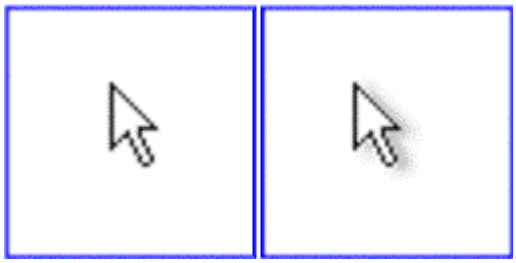

Figure 3. Before and after (enlarged): an alpha shadow gives the cursor depth and makes it stand out better on the screen.

In Windows 95/98 and Windows NT® 4.0 the proper way for an application to create a window with a complex shape, such as a rounded balloon or a cool transparent digital clock, is to specify the shape by supplying the window region representing the shape via the **SetWindowRgn** API. There are several drawbacks to using window regions. If a regional window animates its shape frequently or is dragged on the screen, Windows

will have to ask windows beneath the regional window to repaint. Besides generating more message traffic, the calculations that occur when Windows tries to figure out invalid regions or visible regions become increasingly expensive when a window has an associated region. In addition, using window regions only addresses transparency—that is, color-key effects. It does not address translucency—that is, a way to alpha-blend top-level windows.

This is where layered windows come to the rescue. Layered windows really encompass two different concepts: layering—windows' ability to exhibit sprite-like behavior; and redirection—the system's ability to redirect the drawing of legacy windows into an off-screen buffer.

Using Layered Windows

For any layering to take place, the WS_EX_LAYERED bit needs to be set, either at window creation time or by calling **SetWindowLong** with GWL_EXSTYLE. Next, the developer has a choice: Use the existing Microsoft Win32® painting paradigm by responding to WM_PAINT and/or other paint messages, or make use of a more powerful layering API, **UpdateLayeredWindow**.

To use **UpdateLayeredWindow**, the visual bits for a layered window have to be rendered into a compatible bitmap. Then, via a compatible GDI Device Context, the bitmap is provided to the **UpdateLayeredWindow** API, along with the desired color-key and alpha-blend information. The bitmap can also contain per-pixel alpha information.

Note that when using **UpdateLayeredWindow** the application doesn't need to respond to WM_PAINT or other painting messages, because it has already provided the visual representation for the window and the system will take care of storing that image, composing it, and rendering it on the screen. **UpdateLayeredWindow** is quite powerful, but it often requires modifying the way an existing Win32 application draws.

The second way to use layered windows is to continue using the Win32 painting paradigm, but allowing the system to redirect all the drawing for the layered window and its children into an off-screen bitmap. This can be done by calling **SetLayeredWindowAttributes** with the desired constant alpha value and/or the color-key. Once the API has been called, the system will start redirecting all drawing by the window and automatically apply the specified effects.

Figure 4. The shell in Windows 2000 uses translucency and transparency of layered windows to improve the quality of the image drag on the desktop.

Using **UpdateLayeredWindow** attributes may be more efficient at times, because the application may not need to store a memory bitmap for every layered window. The application can keep one memory bitmap and render into it just before calling **UpdateLayeredWindow** for a number of layered windows. Depending on how often the application calls **UpdateLayeredWindow**, it can also delete the bitmap after calling the API. On the other hand, windows redirected by the system will always carry the overhead of having to maintain a memory bitmap the size of the window for every redirected window. This is in addition to the memory normally consumed by a layered window if **UpdateLayeredWindow** was used to display it. Thus, while certainly more convenient, using **SetLayeredWindowAttributes** comes with a price tag. If your window does a fast animation, **UpdateLayeredWindow** is the way to go.

Please remember that once **SetLayeredWindowAttributes** has been called on a layered window, subsequent **UpdateLayeredWindow** calls will fail until the layering style bit is cleared and set again. This is because **SetLayeredWindowAttributes** turns on the redirection of the window's drawing, and once that happens **UpdateLayeredWindow** can give contradictory information as to what the window really looks like.

Hit Testing

Hit testing of a layered window is based on the shape and transparency of the window. This means that the areas of the window that are color-keyed or whose alpha value is zero will let the mouse messages through.

If the layered window has the WS_EX_TRANSPARENT extended window style, the shape of the layered window will be ignored and the mouse events will be passed to the other windows underneath the layered window.

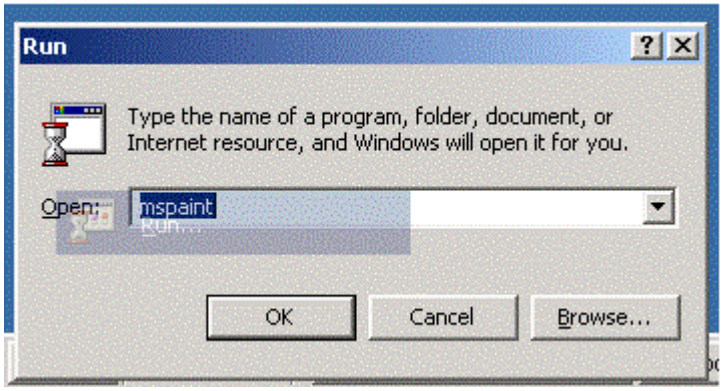

Figure 5. The selected Start menu item "Run" is faded out after the Start menu has been dismissed.

Transition Effects

You don't need to write a lot of code to fade a window in or out. The **AnimateWindow** API can do all the work for you. In fact, this is how the shell's Start menu and other menus do the open fade-in effect.

Internally, **AnimateWindow** will make your windows layered and give the desired transition effect. Although **AnimateWindow** could use the redirection functionality to get the image of the window, it currently uses the WM_PRINT message to get it.

Besides fades, **Animate Window** can also do sliding effects. In fact, if you're writing custom menus, this API can be quite useful. To be a good desktop citizen, your application should figure out whether, when showing the menu, to animate it at all. To get that information, you should query the system via the **SystemParametersInfo** API using SPI_GETMENUANIMATION. Furthermore, you can use SPI_GETMENUFADE to determine whether to use the slide or the fade animation effect. Once you know what effect to use, pass in AW_SLIDE to **AnimateWindow** to get the slide effect and AW_BLEND to get the fade effect.

AnimateWindow also has a parameter that specifies how long the transition should take. Typically, a transition effect shouldn't take longer than 200 milliseconds.

It is true that some power users may not like some of the transition effects enabled by layered windows and will want to turn them off. That's why it's important for applications to be well behaved and query the system metrics via **SystemParametersInfo** to check if the effects are enabled.

Examples of Using Layered Windows

If you want a dialog box to come up as a translucent window:

1. Create the dialog box as usual.
2. On WM_INITDIALOG, set the layered bit of the window's extended style and call **SetLayeredWindowAttributes** with the desired alpha value.

The code might look like this:

```
// Set WS_EX_LAYERED on this window
SetWindowLong(hwnd, GWL_EXSTYLE,
        GetWindowLong(hwnd, GWL_EXSTYLE) | WS_EX_LAYERED);
// Make this window 70% alpha
SetLayeredWindowAttributes(hwnd, 0, (255 * 70) / 100, LWA_ALPHA);
```

Note that the third parameter of **SetLayeredWindowAttributes** is a value that ranges from 0 to 255, with 0 making the window completely transparent and 255 making it completely opaque. This parameter mimics the more versatile BLENDFUNCTION of the AlphaBlend API.

If you want to make this window completely opaque again, remove the WS_EX_LAYERED bit by calling **SetWindowLong** and then ask the window to repaint. Removing the bit is desired to let the system know that it can free up some memory associated with layering and redirection. The code might look like this:

```
// Remove WS_EX_LAYERED from this window styles
SetWindowLong(hwnd, GWL_EXSTYLE,
        GetWindowLong(hwnd, GWL_EXSTYLE) & ~WS_EX_LAYERED);
// Ask the window and its children to repaint
RedrawWindow(hwnd, NULL, NULL, RDW_ERASE | RDW_INVALIDATE | RDW_FRAME |
        RDW_ALLCHILDREN);
```

To fade out a particular area of your window, such as a selection fade-out effect for a custom menu, you can create a dedicated layered window that uses either **UpdateLayeredWindow** or **SetLayeredWindowAttributes** to do the job. For example, the following function—similar to what the shell does to fade out the menu selection—will create a layered window and provide to the system the initial visual bits for it defined by a rectangle on the screen:

```
VOID FadeRect(RECT* prc, HDC hdc)
{
    BOOL fFade = FALSE;
    HWND hwnd;
```

```
    SIZE size;
    POINT ptSrc = {0, 0};
    BLENDFUNCTION blend;

    // Be nice and respect the user's wishes: Do they want the fade?
    SystemParametersInfo(SPI_GETSELECTIONFADE, 0, &fFade, 0);
    if (!fFade)
        return;
    hwnd = CreateWindowEx(WS_EX_LAYERED | // Layered Windows
            WS_EX_TRANSPARENT | // Don't hittest this window
            WS_EX_TOPMOST | WS_EX_TOOLWINDOW,
            gszFade, gszFade, WS_POPUP | WS_VISIBLE, prc->left,
            prc->top, 0, 0, NULL, (HMENU)0, ghinst, NULL);
    size.cx = prc->right - prc->left;
    size.cy = prc->bottom - prc->top;

    blend.BlendOp = AC_SRC_OVER;
    blend.BlendFlags = 0;
    blend.AlphaFormat = 0;
    blend.SourceConstantAlpha = gbAlpha;

    UpdateLayeredWindow(hwnd, NULL, NULL, &size, hdc, &ptSrc, 0,
            &blend, ULW_ALPHA);
    // Finally set the animation timer
    SetTimer(hwnd, ID_TIMER, 25, NULL);
}
```

Subsequently, on a timer the window can fade out by providing the new alpha value:

```
    case WM_TIMER:
        {
            BLENDFUNCTION blend;
            blend.BlendOp = AC_SRC_OVER;
            blend.BlendFlags = 0;
            blend.AlphaFormat = 0;
            blend.SourceConstantAlpha = gbAlpha;

            UpdateLayeredWindow(hwnd, NULL, NULL, NULL, NULL, NULL,
                    NULL, &blend, ULW_ALPHA);
            if (gbAlpha > 25) {
                gbAlpha -= 25;
            } else {
                DestroyWindow(hwnd);
            }
        }
        break;
```

Conclusion

Layered windows present an efficient way to add transparency and translucency to top-level windows. They enable developers to easily incorporate modern UI and cool transition effects into new as well as already existing applications.