Painting only when your window is visible on the screen

devblogs.microsoft.com/oldnewthing/20030829-00

August 29, 2003

Raymond Chen

Sometimes you want to perform an activity, such as updating a status window, only as long as the window is not covered by another window.

The easiest way to determine this is by not actually trying to determine it. For example, here's how the taskbar clock updates itself:

1. It computes how much time will elapse before the next minute ticks over.
2. It calls `SetTimer` with the amount of time it needs to wait.
3. When the timer fires, it does an `InvalidateRect` of itself and the kills the timer.
4. The `WM_PAINT` handler draws the current time, then returns to step 1.

If the taskbar clock is not visible, because it got auto-hidden or because somebody covered it, Windows will not deliver a `WM_PAINT` message, so the taskbar clock will simply go idle and consume no CPU time at all. Here's how we can make our scratch program do the same thing:

Our scratch program displays the current time. It also puts the time into the title bar so we can see the painting action (or lack thereof) when the window is covered or minimized, by watching the taskbar.

```
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
    TCHAR szTime[100];
    if (GetTimeFormat(LOCALE_USER_DEFAULT, 0, NULL, NULL,
                      szTime, 100)) {
        SetWindowText(hwnd, szTime);
        TextOut(pps->hdc, 0, 0, szTime, lstrlen(szTime));
    }
}
```

Here is the timer callback that fires once we decide it's time to update. It merely kills the timer and invalidates the rectangle. The next time the window becomes uncovered, we will get a `WM_PAINT` message. (And if the window is uncovered right now, then we'll get one almost immediately.)

```
void CALLBACK
InvalidateAndKillTimer(HWND hwnd, UINT uMsg,
                       UINT_PTR idTimer, DWORD dwTime)
{
    KillTimer(hwnd, idTimer);
    InvalidateRect(hwnd, NULL, TRUE);
}
```

Finally, we add some code to our `WM_PAINT` handler to restart the timer each time we paint a nonempty rectangle.

```
void
OnPaint(HWND hwnd)
{
    PAINTSTRUCT ps;
    BeginPaint(hwnd, &ps);
    if (!IsRectEmpty(&ps.rcPaint)) {
        // compute time to next update - we update once a second
        SYSTEMTIME st;
        GetSystemTime(&st);
        DWORD dwTimeToNextTick = 1000 - st.wMilliseconds;
        SetTimer(hwnd, 1, dwTimeToNextTick, InvalidateAndKillTimer);
    }
    PaintContent(hwnd,&ps);
    EndPaint(hwnd, &ps);
}
```

Compile and run this program, and watch it update the time. When you minimize the window or cover it with another window, the time stops updating. If you take the window and drag it to the bottom of the screen so only the caption is visible, it also stops updating: The `WM_PAINT` message is used to paint the client area, and the client area is no longer on-screen.

This method also stops updating the clock when you switch to another user or lock the workstation, though you can't really tell because there's no taskbar you can consult to verify. But you can use your speakers: Stick a call to `MessageBeep(-1);` in the `PaintContent()` function, so you will get an annoying beep each time the time is repainted. When you switch to another user or lock the workstation, the beeping will stop.

This technique of invalidation can be extended to cover the case where only one section of the screen is interesting: Instead of invalidating the entire client area, invalidate only the area that you want to update, and restart the timer only if that rectangle is part of the update region. Here are the changes we need to make.

```
// The magic updating rectangle
RECT g_rcTrigger = { 50, 50, 200, 100 };
```

When the timer fires, we invalidate only the magic rectangle instead of the entire client area. (As an optimization, I disabled background erasure for reasons you'll see later.)

```
void CALLBACK
InvalidateAndKillTimer(HWND hwnd, UINT uMsg,
                       UINT_PTR idTimer, DWORD dwTime) {
    KillTimer(hwnd, idTimer);
    InvalidateRect(hwnd, &g_rcTrigger, FALSE);
}
```

To make it more obvious where the magic rectangle is, we draw it in the highlight color and put the time inside it. By using the `ETO_OPAQUE` flag, we draw both the foreground and background simultaneously. Consequently, we don't need to have it erased for us.

```
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
    TCHAR szTime[100];
    if (GetTimeFormat(LOCALE_USER_DEFAULT, 0, NULL, NULL,
                      szTime, 100)) {
        SetWindowText(hwnd, szTime);
        COLORREF clrTextPrev = SetTextColor(pps->hdc,
                            GetSysColor(COLOR_HIGHLIGHTTEXT));
        COLORREF clrBkPrev = SetBkColor(pps->hdc,
                                GetSysColor(COLOR_HIGHLIGHT));
        ExtTextOut(pps->hdc, g_rcTrigger.left, g_rcTrigger.top,
                   ETO_CLIPPED | ETO_OPAQUE, &g_rcTrigger,
                   szTime, lstrlen(szTime), NULL);
        SetBkColor(pps->hdc, clrBkPrev);
        SetTextColor(pps->hdc, clrTextPrev);
    }
}
```

Finally, the code in the `WM_PAINT` handler needs to check the magic rectangle for visibility instead of using the entire client area.

```
void
OnPaint(HWND hwnd)
{
    PAINTSTRUCT ps;
    BeginPaint(hwnd, &ps);
    if (RectVisible(ps.hdc, &g_rcTrigger)) {
        // compute time to next update - we update once a second
        SYSTEMTIME st;
        GetSystemTime(&st);
        DWORD dwTimeToNextTick = 1000 - st.wMilliseconds;
        SetTimer(hwnd, 1, dwTimeToNextTick, InvalidateAndKillTimer);
    }
    PaintContent(hwnd,&ps);
    EndPaint(hwnd, &ps);
}
```

Run this program and do various things to cover up or otherwise prevent the highlight box from painting. Observe that once you cover it up, the title stops updating.

As I noted above, this technique is usually enough for most applications. There's an even more complicated (and more expensive) method, too, which I'll cover next week.

Raymond Chen

Follow