Scrollbars part 11: Towards an even deeper understanding of the WM_NCCALCSIZE message

devblogs.microsoft.com/oldnewthing/20030915-00

September 15, 2003

Raymond Chen

The other form of the `WM_NCCALCSIZE` message is the complicated one, when the `WPARAM` is `TRUE`. In this case, the `LPARAM` is a pointer to a `NCCALCSIZE_PARAMS` structure. When Windows sends the `WM_NCCALCSIZE` message, the `NCCALCSIZE_PARAMS` structure is filled out like this:

- rgrc[0] = new window rectangle (in parent coordinates)
- rgrc[1] = old window rectangle (in parent coordinates)
- rgrc[2] = old client rectangle (**in parent coordinates**)

Notice that the client rectangle is given in parent coordinates, not in client coordinates.

When your window procedure returns, Windows expects the `NCCALCSIZE_PARAMS` structure to be filled out like this:

> rgrc[0] = new client rectangle (**in parent coordinates**)

The new client rectangle specifies where the client area of the window should be located, given the new window rectangle.

Furthermore, if you return anything other than 0, Windows expects the remaining two rectangles to be filled out like this:

- rgrc[1] = destination rectangle (in parent coordinates)
- rgrc[2] = source rectangle (in parent coordinates)

(If you return 0, then Windows assumes that the destination rectangle equals the new client rectangle and the source rectangle equals the old client rectangle.)

The source and destination rectangles specify which part of the old window corresponds to which part of the new window. Windows will copy the pixels from the source rectangle to the destination rectangle and preserve their validity. The return value of the `WM_NCCALCSIZE`

message specifies how the bits should be matched up if the two rectangles are not the same size. The default behavior is to align them at the top and left edges.

Let's demonstrate custom valid rectangles with a fresh scratch program. (We'll come back to the scrollbar program.) First, a helper function that computers the "center" of a rectangle.

```
void GetRectCenter(LPCRECT prc, POINT *ppt)
{
    ppt->x = prc->left + (prc->right - prc->left)/2;
    ppt->y = prc->top + (prc->bottom - prc->top)/2;
}
```

Exercise: Why do we use the formula `c = a + (b-a)/2` instead of the simpler `c = (a+b)/2` ?

Here's our new *PaintContent* function:

```
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
    //  For debugging flicker - fill with an annoying color for 1 second
    DWORD dwLimit = GdiSetBatchLimit(1);
    FillRect(pps->hdc, &pps->rcPaint,
             GetSysColorBrush(COLOR_HIGHLIGHT));
    Sleep(1000);
    FillRect(pps->hdc, &pps->rcPaint,
             GetSysColorBrush(COLOR_WINDOW));
    GdiSetBatchLimit(dwLimit);
    //  Draw "concentric" rectangles
    RECT rc;
    GetClientRect(hwnd, &rc);
    POINT ptCenter;
    GetRectCenter(&rc, &ptCenter);
    int limit = max(rc.bottom, rc.right) / 2;
    rc.left = rc.right = ptCenter.x;
    rc.top = rc.bottom = ptCenter.y;
    for (int i = 0; i < limit; i += 10) {
        InflateRect(&rc, 10, 10);
        FrameRect(pps->hdc, &rc, GetSysColorBrush(COLOR_WINDOWTEXT));
    }
}
```

When debugging flicker problems, it helps to insert intentionally ugly background painting and annoying pauses so you can see what you are painting. Note, though, that when you do this, you also need to call *GdiSetBatchLimit* to disable batching. Otherwise, GDI will optimize out the redundant fill and you won't see anything special.

The real work happens inside our `WM_NCCALCSIZE` handler.

```
UINT OnNcCalcSize(HWND hwnd, BOOL fCalcValidRects,
                          NCCALCSIZE_PARAMS *pcsp)
{
    UINT uRc = (UINT)FORWARD_WM_NCCALCSIZE(hwnd, fCalcValidRects,
                                           pcsp, DefWindowProc);
    if (fCalcValidRects) {
        //  Give names to these things
        RECT *prcClientNew = &pcsp->rgrc[0];
        RECT *prcValidDst  = &pcsp->rgrc[1];
        RECT *prcValidSrc  = &pcsp->rgrc[2];
        // Compute the old and new center points
        POINT ptOldCenter, ptNewCenter;
        GetRectCenter(prcValidSrc, &ptOldCenter);
        GetRectCenter(prcClientNew, &ptNewCenter);
        //  Tell USER how the old and new client rectangles match up
        *prcValidDst = *prcClientNew; // use entire client area
        prcValidDst->left += ptNewCenter.x - ptOldCenter.x;
        prcValidDst->top += ptNewCenter.y - ptOldCenter.y;
        uRc = WVR_VALIDRECTS;
    }
    return uRc;
}
    /* Add to WndProc */
    HANDLE_MSG(hwnd, WM_NCCALCSIZE, OnNcCalcSize);
```

How does this work?

If `fCalcValidRects`, then we do extra work to compute our valid rectangles by seeing how much the window content needs to be shifted and shifting the valid destination rectangle by the same amount. USER copies the upper left corner of the valid source rectangle to the upper left corner of the destination rectangle, so shfiting the upper left corner of the destination rectangle lets us adjust where USER will copy the pixels.

Play with this program: Grab the window and resize it. Observe that the central portion of the window client area is copied from the original window and is not redrawn. This has two benefits: First, there is no flicker. Second, this improves redraw performance since you aren't drawing pixels unnecessarily. This second remark is particularly important when using the Remote Desktop feature, since Remote Desktop has to transfer all drawn pixels over the network to the client. The fewer pixels you have to transfer, the more responsive your program will be.

Now that we have a better understanding of the `WM_NCCALCSIZE` message, we can use this knowledge to improve our scrollbars.

Raymond Chen

Follow