Why does tapping the Alt key cause my owner-draw static control to repaint?

devblogs.microsoft.com/oldnewthing/20161212-00

December 12, 2016

Raymond Chen

A customer had an owner-drawn static control, and they found that when the user pressed the `Alt` key, their static control redrew. This extra redraw was unwanted, presumably because the control takes a long time to draw, and they didn't want to waste the time redrawing something that hadn't changed.

So why does tapping the `Alt` key cause the owner-draw static control to repaint?

Because the state of the keyboard accelerators has changed.

The customer noticed that on the initial draw, the `itemState` had the numeric value of 260, whereas the `itemState` is 4 on the unwanted redraw. The customer noted that 4 is documented as `ODS_DISABLED`, but 260 is not documented.

Okay, well, let's note that the value 260 is documented. It breaks down as `4 | 256` which is `ODS_DISABLED | ODS_NOACCEL`.

Let's walk through what's going on here.

When the static control first paints, the window that contains it is in the "hidden accelerators" state, so the draw flags are `ODS_DISABLED | ODS_NOACCEL`. The `ODS_DISABLED` is a bug, as we saw some time ago. The `ODS_NOACCEL` is telling you to do your owner-draw thing, but don't draw any accelerators.

When the user presses the `Alt` key, the window changes state to "visible accelerators", so the static control asks you to draw once more, with accelerators, as I discussed some time ago.

Okay, now that we know what's going on, what can we do to stop it?

Once you understand the way the `WM_UPDATEUISTATE` and `WM_CHANGEUISTATE` messages interact, you can see two ways out.

One is to ignore the request to draw the control if it is happening in response to a `WM_UPDATEUISTATE` message.

```
LRESULT CALLBACK StaticSubclassProc(
    HWND hwnd, UINT wm, WPARAM wParam, LPARAM lParam,
    UINT_PTR id, DWORD_PTR refData)
{
    LRESULT lres;
    ParentClass *parentClass = (ParentClass*)refData;
    switch (wm)
    {
    case WM_UPDATEUISTATE:
        parentClass->ignoreOwnerDraw = true;
        lres = DefSubclassProc(hwnd, wm, wParam, lParam);
        parentClass->ignoreOwnerDraw = false;
        return lres;
    }
    return DefSubclassProc(hwnd, wm, wParam, lParam);
}

class ParentClass
{
 ...
 void OnDrawItem(const DRAWITEMSTRUCT* pdis)
 {
  if (pdis->CtlID == IDC_MYSTATIC && !ignoreOwnerDraw) {
   DoSlowOwnerDraw(...);
  }
 }

 bool ignoreOwnerDraw = false;
}
```

The idea here is to set a flag if a `WM_UPDATEUISTATE` is in progress, and if the handling of the `WM_UPDATEUISTATE` message results in a request to redraw the control, then ignore it.

I leave as an exercise the code to install and remove the subclass procedure. I do this partly as an actual exercise, and partly to avoid me having to write two versions of the answer, depending on whether the parent is a regular window or a dialog box.

Update: As Adrian notes below, this algorithm fails if the static control chooses merely to invalidate in response to `WM_UPDATESTATE` rather than repaint. By the time the `WM_PAINT` arrives, the flag would already be reset. Fortunately… read on.

Another solution is to prevent the static control from seeing the `WM_UPDATEUISTATE` message at all.

```
LRESULT CALLBACK IgnoreUIStateChangeSubclassProc(
    HWND hwnd, UINT wm, WPARAM wParam, LPARAM lParam,
    UINT_PTR id, DWORD_PTR /* refData */)
{
    switch (wm) {
    case WM_UPDATEUISTATE:
        return DefWindowProc(hwnd, wm, wParam, lParam);
    case WM_NCDESTROY:
        RemoveWindowSubclass(hwnd, IgnoreUIStateChangeSubclassProc, 0);
        break;
    }
    return DefSubclassProc(hwnd, wm, wParam, lParam);
}

BOOL IgnoreUIStateChange(HWND hwnd)
{
 return SetWindowSubclass(hwnd, IgnoreUIStateChangeSubclassProc,
                          1, 0);
}
```

I made this subclass procedure self-unregistering because it has no reference to any other objects, so there are no lifetime issues with letting the subclass procedure outlive the parent class. This makes the function self-contained and consequently generally useful. The `IgnoreUIStateChange` function registers the subclass procedure on any control, at which point the control will ignore any changes to show or hide accelerators or focus rectangles.

The subclass procedure works by intercepting the `WM_UPDATEUISTATE` message and sending it directly to `DefWindowProc` for default processing, bypassing any custom processing in the control itself. Passing the message to `DefWindowProc` allows the normal message propagation to continue, but bypassing the control's window procedure means that the control is never told that the UI state has changed, which means that it never tries to redraw itself to hide or show accelerators or focus rectangles.

Raymond Chen

Follow