There's a default implementation for WM_SETREDRAW, but you might be able to do better

devblogs.microsoft.com/oldnewthing/20110124-00

January 24, 2011

Raymond Chen

If your window doesn't have a handler for the `WM_SETREDRAW` message, then `DefWindow-Proc` will give you a default implementation which suppresses `WM_PAINT` messages for your window when redraw is disabled, and re-enables `WM_PAINT` (and triggers a full repaint) when redraw is re-enabled. (This is internally accomplished by making the window pseudo-invisible, but that's an implementation detail you shouldn't be concerned with.) Although the default implementation works fine for simple controls, more complex controls can do better, and in fact they *should* do better, because that's sort of the point of `WM_SETREDRAW`. The intended use for disabling redraw on a window is in preparation for making large numbers of changes to the window, where you don't want to waste time updating the screen after each tiny little change. For example, if you're going to add a hundred items to a list box, you probably want to disable redraw while adding the items so you don't have to suffer through 100 screen refreshes when only one is enough. You've probably seen the programs that forget to suppress redraw when filling a large list box: The application freezes up except for a list box whose scroll bar starts out with a big thumb that slowly shrinks as the number of items increases. I say that this is sort of the point of `WM_SETREDRAW` for a complex control, because if you have a simple control (like a button), there isn't much in the way of "bulk updates" you can perform on it, so there isn't much reason for anybody to want to disable redraw on it anyway. The types of windows for which people want to disable redraw are the types of windows that would benefit most from a custom handler. For example, the list view control has a custom handler for `WM_SETREDRAW` which sets an internal *redraw has been disabled* flag. Other parts of the list view control check this flag and bypass complex screen calculations if is set. For example, when you add an item to a list view while redraw is disabled, the list view control doesn't bother recalculating the new scroll bar position; it just sets an internal flag that says, "When redraw is re-enabled, don't forget to recalculate the scroll bars." If the list view is in auto-arrange, it doesn't bother rearranging the items after each insertion or deletion; it just sets an internal flag to remember to do it when redraw is re-enabled. If you have a regional list view, it doesn't bother recalculating the region; it just sets a flag. And when you finally re-enable drawing, it sees all the little Post-It note reminders that it left lying around and says, "Okay, let's deal with all this stuff that I had been putting off." That way, if you add 100 items, it doesn't perform 99 useless scroll bar calculations, 99 useless auto-arrange repositionings, and create, compute, and then destroy 99 regions. Since some of these calculations are $O(n)$, deferring them when redraw is disabled improves the performance of inserting n items from $O(n^2)$ to $O(n)$. Moral of the story: If you have a control that manages a large number of sub-items, you should have a custom `WM_SETREDRAW` handler to make bulk updates more efficient.

Bonus chatter: Note that using `LockWindowUpdate` as a fake version of `WM_SETREDRAW` does not trigger these internal optimizations. Abusing `LockWindowUpdate` gets you the benefit of not repainting, but you still have to suffer through the various $O(n^2)$ calculations.

Raymond Chen

Follow