Why doesn't the window manager just take over behavior that used to be within the application's purview?

devblogs.microsoft.com/oldnewthing/20060327-16

March 27, 2006

Raymond Chen

A commenter named "Al" wondered why the window manager couldn't just take over behavior that used to be within the application's purview, such as painting the non-client area, in order to avoid problems with applications not responding to messages promptly enough. If the window manager were being rewritten, then perhaps it could. But to do it now would introduce many compatibility issues. First, there are many applications that have subtle dependencies on message ordering or receiving certain types of messages at certain times, even though there is no actual guarantee in the specification that such messages be delivered. There are a large number of applications that rely on `WM_PAINT` messages being delivered even if there is nothing to paint, because they defer some critical computations until the first `WM_PAINT` message, and if something that requires the result of that computation happens before a `WM_PAINT`, they crash. For example, if you launch a program minimized, then right-click on the taskbar button for the program's main window, these programs would crash because the code that handles the system menu uses a pointer variable that the `WM_PAINT` handler initializes or divides by a global variable whose default value is zero but whose value is calculated during `WM_PAINT` handling. To accomodate these programs, the window manager is forced to send "dummy" `WM_PAINT` messages with an empty `rcPaint`. These such messages appear to accomplish nothing, but the hidden agenda is that the program gets its cherished `WM_PAINT` message and can perform whatever operations it is that keeps it from crashing later on. Second, removing customizability of message behavior from the window manager would prevent programs from customizing their appearance in nonstandard ways. Media players are perhaps the most popular example of programs that want to override normal non-client painting in order to present a totally customized window to the user. Would you be happy if a change to Windows meant that you could no longer "skin" your favorite media player application? That said, there have been changes to the window manager over the years to maintain this "air of customizability" while simultaneously intervening on behalf of the user to keep things from going completely to the dogs. For example, if a window stops painting for an extended period of time, Windows would take it upon itself to paint the window with a standard caption bar (even if the application wanted to customize the caption bar), just so that the user would be able to see *something*. Another example of this "message virtualization" is the appending of the phrase

“(Not responding)” to the caption of a window that has stopped responding, and capturing the window contents as they were last visible, drawing those captured window contents in the meantime until the application woke up from its slumber, and even allowing you to move, resize, minimize, and close those unresponsive windows. The infrastructure necessary to support this behavior is quite extensive, because the window manager needs to maintain two sets of bookkeeping. The first is, “What the application thinks the window state is”; if the application asks for the size of its hung window, it needs to be told, “Oh, you’re still that size you were before, don’t you worry your pretty little head”, even though the actual window size on the screen has changed significantly. Once the hung window starts responding to messages again, all the activity that happened “while it was away” needs to be replayed to get the window “back up to speed” with the state of the world. Interesting things happen if the program wanted to customize one of the actions that happened to the “virtual window”. For example, it might want to reject certain window sizes or display a special message before minimizing. Resolving these conflicts in a manner that doesn’t cause applications to crash outright is another of the difficulties of trying to get the virtual and real window states back into sync.

In a sense, therefore, the window manager does take over selected behaviors that used to be within the application’s purview, but it has to do it in a delicate enough manner that neither the application nor the end user will even realize that it’s happening. And that’s what makes it hard.

Raymond Chen

Follow