The cost of continuously-visible affordances with dynamic states

devblogs.microsoft.com/oldnewthing/20070122-05

January 22, 2007

Raymond Chen

Serge Wautier asks, "Why are the copy/cut/paste buttons not disabled when there's nothing to copy/cut/paste?", noting that the back/forward buttons do disable themselves when navigation is not possible in that direction. To get to this question, we'll first go back in time a bit to a world without toolbars. In those early days, these dynamic options such as copy/cut/paste appeared solely on the Edit menu. Since the contents of Edit menu were visible only when the user clicked on it, the cut/copy/paste options needed to be updated only when the menu was visible. In other words, during `WM_INITMENUPOPUP` handling. This is also why it is somewhat risky to post `WM_COMMAND` messages which correspond to a menu item to a window which is not prepared for it. The only way an end-user can generate that `WM_COMMAND` message is by going through the menu: clicking the top-level menu to show the drop-down menu, then clicking on the menu item itself. Most programs do not maintain the menu item states when the menu is closed since there's no point in updating something the user can't see. Instead, they do it only in response to the `WM_INITMENUPOP` message. Lazy evaluation means that the user doesn't pay for something until they use it. In this case, paying for the cost of calculating whether the menu item should be enabled or not. Depending on the program, calculating whether a menu item should be enabled can turn out to be rather expensive, so it's natural to avoid doing it whenever possible. ("I can do nothing really fast.") When toolbars showed up, things got more complicated. Now, the affordances are visible **all the time**, right there in the toolbar. How do you update something continuously without destroying performance? The navigation buttons disable and enable themselves dynamically because the conditions that control their state satisfy several handy criteria.

- The program knows when the state has potentially changed. (The program maintains the navigation history, so it knows that the button states need to be recalculated only when a navigation occurs.)
- Computing the state is relatively cheap. (All the program has to check is whether there is a previous and next page in the navigation history Since the navigation history is typically maintained as a list, this is easy to do.)

- They change in proportion to user activity within the program. (Each state change can be tied to a user's actions. They don't change on their own.)
- They change rarely. (Users do not navigate a hundred times per second.)

Since the program knows when the navigation stack has changed, it doesn't have to waste its time updating the button states when nothing has changed. Since recalculating the state is relatively cheap, the end user will not see the main user interface slow down while the program goes off to determine the new button state after each navigation. And finally, the state changes rarely, so that this cheap calculation does not multiply into an expensive one.

The copy/cut/paste buttons, on the other hand, often fail to meet these criteria. First, the copy and cut options:

- The program knows when the state has potentially changed. (Whenever the selection changes.) — good
- Computing the state is not always cheap. (For example, determining whether an item in Explorer can be cut or copied requires talking to its namespace handler, which can mean loading a DLL. If the item on the clipboard is a file on the network, you may have to access a computer halfway around the world.) — often bad
- It changes in proportion to user activity within the program. (Each state change can be traced to the user changing the selection.)
- They change with high frequency. (Dragging a rectangle to make a group selection changes the selection each time the rectangle encloses a new item.) — bad

Paste is even worse.

- The program doesn't know when the state has potentially changed. (The clipboard can change at any time. Yes, the program could install a clipboard viewer, but that comes with its own performance problems.) — bad
- Computing the state is not cheap. (The program has to open the clipboard, retrieve the data on it, and see whether it is in a format that can be pasted. If the clipboard contents are delay-rendered, then the constant probing of the clipboard defeats the purpose of delay-rendered clipboard data, which is to defer the cost of generating clipboard data until the user actually wants it. For Explorer, it's even worse, because it has to take the data and ask the selected item whether it can accept the paste. Doing this means talking to the namespace handler, which can mean loading a DLL. And if the file on the clipboard is on the network, the paste handler may need to open the file to see if it is in a format that can be pasted.) — bad
- It can change out of proportion to user activity. (Any time any other program copies something to the clipboard, the toolbar has to update itself. Then can happen even when the user is not using the program that has the toolbar! Imagine if Explorer started saturating your network because you copied a lot of UNC paths to the clipboard while editing some text file.) — bad

- The frequency of change is unknown. (The clipboard is a shared resource, and who knows what other people might be using it for.) — bad

This is one of those balancing acts you have to do when designing a program. How much performance degredation are you willing to make the user suffer through in order to get a feature they may never even notice (except possibly in a bad way)?

Raymond Chen

Follow