Why are mouse wheel messages delivered to the focus window instead of the window under the mouse?

devblogs.microsoft.com/oldnewthing/20160420-00

April 20, 2016

Raymond Chen

Douglas asks, "Is there a good reason that mouse wheel messages are sent to the focused window instead of the window under the mouse?"

I don't know if there's a *good* reason, but there's *a* reason.

The mouse wheel turned into one of those features whose primary use is not the original intended use. The original intent of the mouse wheel was to control zooming. Spin the wheel forward to zoom in; spin it backward to zoom out. (Or is it the other way? I can never remember.) In practice, though, people preferred to use the mouse wheel to scroll, and zooming got relegated to a secondary feature activated by holding the `Ctrl` key while turning the wheel.

And it's the `Ctrl` key that is the key that unlocks the puzzle. That, plus some historical context.

The historical context is that the mouse wheel in Windows was introduced by the Microsoft Hardware team in 1996. This was not synchronized with an operating system release, and Windows Update wasn't a thing yet, not that anybody cared because Internet access was not widespread, and those who did were using mostly dial-up.

Therefore, the hardware team had to invent their own mechanism for mouse wheel messages, with the understanding that operating system support for it wouldn't arrive until the next major release of Windows (which turned out to be Windows 98 and Windows NT 4.0).

You can see the result of this in the `zmouse.h` header file buried in your SDK. This is a header file from 1996 that nobody wants to delete because doing so might break somebody. (See also: `multimon.h`.)

The fact that the wheel was intended for zooming rather than scrolling is also evident in the names of the parameters in the header file: The parameter that receives a boolean that tells you whether a wheel mouse is installed is named *pf3DSupport*. It's also apparent in the

header file name: The *z* represents the third coordinate, emphasizing that the wheel is supposed to move you closer to or further away from the document.

Since the hardware team had to invent something and couldn't rely on help from the window manager team, they had to make do with what they could. Wheel support operated by having a dedicated helper program that listened to the mouse hardware. When it detected that the wheel scrolled, it delivered a custom wheel scroll message: `MSH_MOUSEWHEEL`.[1]

Okay, now you have a few problems. Let's start with "What do you put in the message payload?"

You have four pieces of information that are essential:

- The wheel scroll amount.
- The mouse *x*-coordinate.
- The mouse *y*-coordinate.
- The keyboard shift key states.

The scroll amount is essential because that's sort of the point of the message. The mouse coordinates are essential in order for the application to know which object to scroll. And the shift key states are essential because that tells the application what action it should take.

The helper program can get the first piece of essential information because it came from the mouse. The second and third pieces can come from `GetCursorPos`. But what about the fourth piece?

Keyboard state is input-queue-dependent, and the helper program doesn't have access to the input queue of the window that receives the `MSH_MOUSEWHEEL` message. Therefore, in order to preserve sanity, it has to deliver the message to the window with keyboard focus, or a window on the same input queue as the window with keyboard focus, since that's the only input queue that knows whether the `Ctrl` key is down.[2]

Besides, even if the helper program could get access to the keyboard state, it didn't really have anywhere to put it. If you want 16-bit applications to support the mouse wheel (and 16-bit applications were still very important in 1996), you have to pack everything into a 16-bit `WPARAM` and a 32-bit `LPARAM`, which is enough space to hold three 16-bit pieces of information. And we already have three pieces of information. We're all full!

There's no room for the keyboard state in the `MSH_MOUSEWHEEL` message payload, which is just as well because there's no way to get it, so it falls to the application to figure out the keyboard state, which only the window with keyboard focus can do.

We are basically trapped into using the window with keyboard focus.

When the mouse wheel messages were incorporated into the operating system, they decided to give up on supporting 16-bit applications, which opens up another 16 bits of space to provide keyboard state information. And the window manager knows all about keyboard states, so it can figure out what keyboard state to give to the application. But it followed the focus rules to preserve UI compatibility.

It looks like the window manager team in Windows 10 decided to break UI compatibility and deliver the mouse wheel messages to the window that has the mouse. So at least that part of history is now behind us. One down, a few zillion to go.

Bonus trivia: The code name for the mouse that introduced the wheel was *Magellan*, presumably to reflect the emphasis on navigation. You can still see vestiges of this code name in MSDN, for example, in the list of features new to the RichEdit 2.0 control.

[1] A reasonable guess would be that the prefix `MSH` stands for *Microsoft Hardware*.

[2] I guess you could use `GetAsyncKeyState` to see if the `Ctrl` key is down, but that is a race condition because the key may not have been pressed at the time the wheel turned, but it was pressed when you processed the wheel message. (Or vice versa.)

Raymond Chen

Follow