How can I detect whether my PC is in tablet mode?

devblogs.microsoft.com/oldnewthing/20160706-00

July 6, 2016

Raymond Chen

Tablet Mode, introduced in Windows 10, is a blah blah blah blah.

Okay, enough with the introduction.

From a Store app, you detect whether you are in tablet mode by inspecting the `UserInteractionMode` for your view. Sample code for this is given in the UserInteractionMode sample, but the short version is that you do this:

```
UIViewSettings^ uiViewSettings = UIViewSettings::GetForCurrentView();
UserInteractionMode mode = uiViewSettings->UserInteractionMode;
switch (mode)
{
case UserInteractionMode::Touch:
  // PC is in tablet mode or other touch-first environment
  break;

case UserInteractionMode::Mouse:
  // PC is not in tablet mode or other mouse-first environment
  break;
}
```

The user interaction mode is a per-view property because the system may have multiple monitors, some of which are in tablet mode and some of which aren't. An app can detect when the user interaction mode of a view has changed by listening for the `SizeChanged` event.

This is a general convention for view properties: View properties that affect layout generally raise the `SizeChanged` event when they change. The idea behind this is that this gives you a single event to trigger the recalculation of your app's layout policy. If multiple things change at once, such as the window size, the user interaction mode, and the full-screen mode, then you run only one layout recalculation instead of three.

Okay, that's great for a Store app, but what about your classic desktop app? How does a classic desktop app learn whether a window is on a monitor that is in tablet mode?

You basically do exactly the same thing as the Store app: You get the `UIViewSettings` and ask it for the user interaction mode. The gotcha is that `GetForCurrentView` doesn't make sense in a desktop app because desktop apps don't have have a `CoreApplicationView`.

The answer is to use the interop interface `IUIViewSettingsInterop`. The general design pattern for the interop interface is that it each view-related static method has a corresponding method on the interop interface that takes a window handle instead of a view.

We'll see these as we go through the Little Program. (Remember, Little Programs do little to no error checking.) Start with the scratch program and make these changes:

```
#include <wrl/client.h>
#include <wrl/wrappers/corewrappers.h>
#include <windows.ui.viewmanagement.h>
#include <UIViewSettingsInterop.h>
#include <tchar.h> // Huh? Why are you still using ANSI?

namespace WRL = Microsoft::WRL;
namespace vm = ABI::Windows::UI::ViewManagement;

WRL::ComPtr<vm::IUIViewSettings> g_viewSettings;

vm::UserInteractionMode g_mode = vm::UserInteractionMode_Mouse;
```

So far, we're just declaring some global thingies. In a real program, these would probably be instance members of some C++ class, but I'm being lazy.

```
void CheckTabletMode(HWND hwnd)
{
  if (g_viewSettings)
  {
    vm::UserInteractionMode currentMode;
    g_viewSettings->get_UserInteractionMode(&currentMode);
    if (g_mode != currentMode)
    {
      g_mode = currentMode;
      // This sample just updates some text.
      InvalidateRect(hwnd, nullptr, true);
    }
  }
}
```

Okay, here's the part where we read the current user interaction mode from the `IUIView-Settings`, and if it changed, we do whatever we do when the user interact mode changes. In a real problem, we would do some relayout, but in this sample program, we're just going to update a string, so we invalidate our window so we can draw the new string.

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
  WRL::ComPtr<IUIViewSettingsInterop> interop;
  Windows::Foundation::GetActivationFactory(WRL::Wrappers::HStringReference(
     RuntimeClass_Windows_UI_ViewManagement_UIViewSettings).Get(),
     &interop);

  interop->GetForWindow(hwnd, IID_PPV_ARGS(&g_viewSettings));

  CheckTabletMode(hwnd);
  return TRUE;
}
```

Okay, now things get exciting. To get the `UIViewSettings` for a window, you first get the activation factory (which is where all the static methods hang out) and ask for the `IUIView-SettingsInterop` interface. From there, call `GetForWindow`, which is the window-based version of `GetForCurrentView`.

That's the only wrinkle. Once you have the `UIViewSettings`, you can get its user interaction mode as usual.

```
void
OnDestroy(HWND hwnd)
{
  g_viewSettings.Reset();
  PostQuitMessage(0);
}
```

Naturally, we need to clean up when we're done.

```
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
  PCTSTR message = TEXT("?");

  // adapt to the new mode! We just update our string.
  switch (g_mode)
  {
  case vm::UserInteractionMode_Mouse: message = TEXT("Mouse"); break;
  case vm::UserInteractionMode_Touch: message = TEXT("Touch"); break;
  }

  TextOut(pps->hdc, 0, 0, message, _tcslen(message));
}
```

Our `PaintContent` function prints the current mode.

Wait, what about the `SizeChanged` event? Oh, right, for classic Win32, you can just use the `WM_WINDOWPOSCHANGED` message, which will give us a chance to see if we moved to a monitor that is in a different tablet mode state from where we were before.

```
void
OnWindowPosChanged(HWND hwnd, LPWINDOWPOS lpwpos)
{
  CheckTabletMode(hwnd);
}
```

The last wrinkle is the case where the global tablet mode state changes.

```
void OnSettingsChange(HWND hwnd, LPCTSTR sectionName)
{
  if (sectionName &&
      lstrcmpi(sectionName, TEXT("UserInteractionMode")) == 0)
  {
    CheckTabletMode(hwnd);
  }
}
```

When the global tablet mode state changes, the shell broadcasts the `"UserInteraction-Mode"` setting change notification.

```
HANDLE_MSG(hwnd, WM_WINDOWPOSCHANGED, OnWindowPosChanged);
HANDLE_MSG(hwnd, WM_WININICHANGE, OnSettingsChange);
```

And finally, we hook up our message handlers.

There you go, a program that knows whether it is in tablet mode.

Raymond Chen

Follow