How can I reposition my controls so they aren't covered by the touch keyboard?

devblogs.microsoft.com/oldnewthing/20150608-00

June 8, 2015

Raymond Chen

Last week, we saw how to tell the Windows 8 touch keyboard to appear automatically in a desktop program when focus enters an edit control. But we did nothing to prevent the touch keyboard from covering the edit control.

To find out when the touch keyboard appears and disappears, you register with the framework input pane. The framework input pane tells you the current location of the keyboard and informs you when the keyboard state changes. (Note that it does not tell you about the keyboard when it is in floating mode, on the theory that when the user floats the keyboard, the user has taken responsibility for making sure it doesn't cover anything interesting.)

Take our program from last week and make these changes:

```
#include <strsafe.h>

using namespace Microsoft::WRL;

// Add immediately before DoLayout

ComPtr<IFrameworkInputPane> g_frameworkInputPane;
DWORD g_cookie;
RECT g_rcKeyboard;
```

In a real program, of course, we wouldn't use global variables, but this is a Little Program.

```
void Relayout(HWND hwnd,
  bool isKeyboardShowing = false);

class Handler : public RuntimeClass<
  RuntimeClassFlags<RuntimeClassType::ClassicCom>,
  IFrameworkInputPaneHandler>
{
public:
  Handler(HWND hwnd) : m_hwnd(hwnd) { }

  STDMETHODIMP Hiding(BOOL fEnsureFocusedElementInView) override
  {
    SetRectEmpty(&g_rcKeyboard);
    Relayout(m_hwnd);
    return S_OK;
  }

  STDMETHODIMP Showing(RECT *prcScreenLocation,
      BOOL fEnsureFocusedElementInView) override
  {
    g_rcKeyboard = *prcScreenLocation;
    Relayout(m_hwnd, true);
    return S_OK;
  }
private:
  HWND m_hwnd;
};
```

Our `Handler` class implements the `IFrameworkInputPaneHandler` interface so it can be called when the input pane shows and hides. When this happens, we update the keyboard rectangle and ask our window to update its layout in response to the new information. We pass a special flag that indicates whether the call is in response to the keyboard showing, because we want to do extra work in that case.

```
void
DoLayout(HWND hwnd, int cx, int cy, bool isKeyboardShowing = false)
{
  // Just for fun, put the keyboard position in the title bar.
  if (IsRectEmpty(&g_rcKeyboard)) {
    SetWindowText(hwnd, TEXT("Keyboard is not visible"));
  } else {
    TCHAR message[256];
    StringCchPrintf(message, ARRAYSIZE(message),
      TEXT("Keyboard is at (%d,%d)-(%d,%d)"),
      g_rcKeyboard.left, g_rcKeyboard.top,
      g_rcKeyboard.right, g_rcKeyboard.bottom);
    SetWindowText(hwnd, message);
  }

  if (g_hwndChild) {
    int cyEdit = cy;
    if (!IsRectEmpty(&g_rcKeyboard)) {
      RECT rcEdit = { 0, 0, cx - 100, cy };
      RECT rcKeyboardClient = g_rcKeyboard;
      MapWindowRect(nullptr, hwnd, &rcKeyboardClient);
      RECT rc;
      if (IntersectRect(&rc, &rcEdit, &rcKeyboardClient)) {
        cyEdit = min(rcKeyboardClient.top, cy);
      }
    }
    MoveWindow(g_hwndChild, 0, 0, cx - 100, cyEdit, TRUE);
    if (isKeyboardShowing) {
      SendMessage(g_hwndChild, EM_SCROLLCARET, 0, 0);
    }
  }
  if (g_hwndButton) {
    MoveWindow(g_hwndButton, cx - 100, 0, 100, 50, TRUE);
  }
}
```

First, we update the title bar to show where we think the keyboard is, just so it's easier to follow what's happening. And then the actual action: If the keyboard is visible and it overlaps the edit control, then we resize the edit control to avoid it. And if the keyboard is showing, then we scroll the edit control so that the caret is visible. We don't want to force the caret visible in the general case, because that would cause the contents to scroll at unexpected times.

```
void
OnMove(HWND hwnd, int x, int y)
{
  Relayout(hwnd);
}
```

When the window moves, we want to perform relayout, because the window may have moved in such a way that the edit control is obscured by the keyboard.

```
void
Relayout(HWND hwnd, bool isKeyboardShowing);
{
  RECT rc;
  GetClientRect(hwnd, &rc);
  DoLayout(hwnd, rc.right, rc.bottom, isKeyboardShowing);
}
```

This function is kind of anticlimactic. To perform relayout, we get the client rectangle and ask `DoLayout` to lay out the contents inside that rectangle.

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
  ...
  EnableTouchKeyboardFocusTracking();

  CoCreateInstance(__uuidof(FrameworkInputPane), nullptr,
    CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&g_frameworkInputPane));
  g_frameworkInputPane->AdviseWithHWND(hwnd,
    Make<Handler>(hwnd).Get(),
    &g_cookie);

  return TRUE;
}
```

Here is how the ball gets set into motion: When we create the window, we also register our handler with the framework input pane, so that we are called back when the keyboard moves.

```
void
OnDestroy(HWND hwnd)
{
  if (g_cookie) {
    g_frameworkInputPane->Unadvise(g_cookie);
  }
  g_frameworkInputPane = nullptr;
  PostQuitMessage(0);
}
```

And some bookkeeping: When the window is destroyed, we unregister from the framework input pane. We also need to release the framework input pane before COM get uninitialized. (This wouldn't have been necessary if we had put the framework input pane in a member variable, since the member variable would be destructed at window destruction. But we were lazy and used a global variable, and now we pay the price.)

If you take this program out for a ride, you'll see that it manages to resize the edit control so that it is not covered by the touch keyboard. But there's still a problem: What if the window is near the bottom of the screen, and the user calls up the touch keyboard? The entire edit

control ends up obscured! No amount of resizing will fix this. We'll look at this problem next time.

Raymond Chen

Follow