Visual Studio Magazine Online
Classic VB Corner

Thoughtful UI Tweaks Can Make All the Difference

ONLINE ONLY

The difference between a successful program and a complete flop may come down to how you treat your users. Here's how to make sure your app's UI doesn't annoy users more than it helps them.

August 11, 2009 · by Karl E. Peterson

Okay, time for another rant. I have a few applications that drive me nuts when I fire them up. Not enough to uninstall and be done with them (until I find a better functional replacement, anyway). But they certainly vex me enough to make me growl quietly while I wait for them to initialize.

What do they do wrong? Play stupid monitor games.

One is just the ultimate in startup stupidity. Every time it fires up, it insists on maximizing itself on the primary monitor. How rude can a programmer be? If the app didn't contain mission-critical algorithms, the UI alone would have sealed its death sentence long ago.

A second annoyance is an app that remembers where it was positioned the last time it was shutdown, and politely recreates itself at the same coordinates. So what's the problem? It throws up an "always on top" splash screen dead-center on the primary monitor for five to eight seconds. No option to turn it off. No way to get work done on the display while the splash screen is up. And since I like to run this app on my secondary monitor, I'm left with both monitors functionally disabled during its initialization. Mind you, if my active window is on the primary monitor, it can still accept input. I just can't see it. Very nice.

Developers, please don't treat your users like this. It's just *not that hard* to remember where your application was the last time it shut down, and to put it back there when it restarts. Last March, I wrote a column on Working with Multiple Monitors that provided a way to determine the coordinates for each display, as well as the entire virtual desktop, and much more.

I've added a few new functions that now give you no excuse not to be as polite as possible when starting your application. I think it's a given that, with few exceptions, the nicest applications remember where they were and put themselves back there when restarted. But "stuff" can happen. Perhaps the user has changed resolution since the last run, or removed a monitor\ and the position you've persisted is no longer valid. That's fairly easily solved by devising a little algorithm that slides a window back onscreen if it's off the visible edge. The following routine uses some of

the functionality provided by my Monitors sample as it's now been added to that sample:

```
Public Function WindowOffscreen( _
   ByVal hWnd As Long, _
   Optional OverlapOK As Boolean = True, _
   Optional ForceOnscreen As Boolean = True) As Boolean

   Dim rW As RECT, rM As RECT, rU As RECT
   Dim Overlapped As Boolean
   ' Collection needs to be initialized.
   If (Monitors Is Nothing) Then MonitorsRefresh

   ' Test whether the specified window is actually within
   ' the visible display area.
   rW = WindowRectAbsolute(hWnd)
   rM = MonitorRect(0)

   ' Quick union tells if window is entirely contained on desktop.
   Call UnionRect(rU, rM, rW)
   If EqualRect(rU, rM) Then Exit Function

   ' Next test is whether rectangle partially intersects desktop.
   If IntersectRect(rU, rM, rW) Then
      If OverlapOK Then Exit Function
   End If

   ' Check each edge, and slide into screen as necessary.
   ' Prefer top/left by setting it last.
   If rW.Right ›rM.Right Then
      Call OffsetRect(rW, (rM.Right - rW.Right), 0)
   End If
   If rW.Bottom ›rM.Bottom Then
      Call OffsetRect(rW, 0, rM.Bottom - rW.Bottom)
   End If
   If rW.Left ‹rM.Left Then
      Call OffsetRect(rW, (rM.Left - rW.Left), 0)
   End If
   If rW.Top ‹rM.Top Then
      Call OffsetRect(rW, 0, (rM.Top - rW.Top))
   End If

   ' If rW has not changed, then it was entirely onscreen before.
   If Not EqualRect(rW, WindowRectAbsolute(hWnd)) Then
      WindowOffscreen = True
      If ForceOnscreen Then
         Call WindowMove(hWnd, rW)
      End If
   End If
End Function
```

We start by collecting the RECT structures that represent the window position as well as the entire virtual desktop. Calling the UnionRect API function creates what its name implies -- a rectangle that's the union of the two supplied. If this resultant rectangle is equal to the original virtual desktop rectangle, we know the entire window is onscreen Optionally, we may next test whether the window partially overlaps the visible display by using the IntersectRect API.

That leaves the cases where the entire window is offscreen, or portions of the window are overlapping and that's not desirable. To remedy this, we test each edge by comparing it to that of the virtual desktop. If we find that our window has slipped off the edge, calling the OffsetRect API is a bit easier than the math. Not that the math isn't extremely basic, it's just easy to get rectangle elements mixed up as you cut/paste/edit the four separate calculations. Using the Left edge as an example, here's the API way:

```
If rW.Left ‹rM.Left Then
   Call OffsetRect(rW, (rM.Left - rW.Left), 0)
End If
```

This simply shifts the rW rectangle to the right (positive X direction) by the difference between the left edge of the monitor and the left edge of the window. But we could "just as easily" do it with pure math, like this:

```
If rW.Left ‹rM.Left Then
   rW.Right = rM.Left + (rW.Right - rW.Left)
   rW.Left = rM.Left
End If
```

After adjusting our rW rectangle so that it's entirely onscreen, we can compare it to the rectangle of the original window, and if they're different we know that the window was (at least partially) off-screen. If the caller requested, we can use the new rectangle to slide the window back on-screen.

Using this simple strategy, you may persist with confidence your window position when shutting down, knowing that if the environment changes before you run again you can easily recover. You could also modify the routine above such that it forced a window onto any specific monitor by simply changing the index requested of the Monitors collection in the call to MonitorRect.

With a Twist

An interesting twist was being discussed the other day on msnews. It was suggested that some applications will always start up on the monitor where the mouse is. Hmmm. Cool? Might be. Probably depends on the application and its audience. Worth considering though, and extremely easy to implement. Determining which monitor the mouse is over just takes a couple lines of code – calling the GetCursorPos API, then passing those coordinates to the MonitorFromPoint function in my Monitors sample:

```
Public Function MonitorFromMouse() As Long
   Dim pt As POINTAPI
   ' Find out which monitor the mouse is over.
   Call GetCursorPos(pt)
   MonitorFromMouse = MonitorFromPoint(pt.x, pt.y)
End Function
```

Moving a window to the same location on a different monitor means obtaining that window's relative position to the monitor it's on, then adjusting that to the relative position on the new monitor and converting that to the absolute screen coordinates. Harder to say than to encapsulate, thankfully:

```
Public Sub WindowSwitchMonitor(ByVal hWnd As Long, ByVal Index As Long)
   Dim rW As RECT, rM As RECT
   ' Place window in same position on different monitor.
   rM = MonitorRect(Index)
   rW = WindowRectRelative(hWnd)
   ' Adjust for relative position.
   Call OffsetRect(rW, rM.Left, rM.Top)
   ' Move it to new position.
   Call WindowMove(hWnd, rW)
End Sub

Public Function WindowRectAbsolute(ByVal hWnd As Long) As RECT
   ' Supply absolute screen coordinates of window.
   Call GetWindowRect(hWnd, WindowRectAbsolute)
End Function

Public Function WindowRectRelative(ByVal hWnd As Long) As RECT
   Dim rW As RECT, rM As RECT
   ' Supply screen coordinates of window relative to the
   ' monitor it is primarily on.  Gather basic facts...
   rM = MonitorRect(MonitorFromWindow(hWnd))
   rW = WindowRectAbsolute(hWnd)
   ' Adjust for relative position.
   Call OffsetRect(rW, -rM.Left, -rM.Top)
   WindowRectRelative = rW
End Function
```

The GetWindowRect API always gives us the absolute window coordinates, which we can then offset by the top/left coordinates of the two monitors in question. Looks messy; works great. This combination of techniques allows you to startup on whatever monitor the mouse is currently hovering over, with this code in Form_Load:

```
' Make sure form first appears on the monitor that
' the mouse is over.
Call WindowSwitchMonitor(Me.hWnd, MonitorFromMouse())
```

Add that right after any code you may have to set the absolute position. I'm not sure I'm going to use this myself, but it's an interesting thought.

That leaves persisting the window position at closure. Please leave me some comments below, if you think that capability ought to be added to my Monitors sample as well.

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.