Controlling Pointer Appearance

Smoothly changing the appearance of pointers in your apps gives users useful feedback about the state of your program.

BY CHRIS ADLER

Visual Basic is probably the most productive language for Windows application development. Almost all of the tedious tasks and overhead required to develop Windows applications are automatically handled by Visual Basic itself. The programmer doesn't need to know that such tasks exist.

Some tasks, however, aren't automatically handled by VB, and many beginning programmers (and even some advanced programmers) overlook them. One of the frequently overlooked manual tasks a programmer must understand is the correct use of the MousePointer property.

Most Visual Basic programmers have used the MousePointer property in their projects. This property provides a simple mechanism for changing the shape of the mouse pointer among the eight predefined shapes provided by Windows. Most programmers use the MousePointer property in two basic ways: displaying an hourglass and displaying the normal arrow.

While changing the shape of a pointer may seem mundane, the shape of the mouse pointer has a substantial impact on the usability of your programs. Although the use of the MousePointer property has nothing to do with the actual speed of your application, it does affect the user's perception of speed.

Chris Adler is vice president of Invoice Central and the author of several successful shareware utilities. He also works as an independent contractor specializing in client/server database applications. He can be reached on Compuserve at 71726,2677.

A MATTER OF PERCEPTION

User perception, in a graphical environment such as Windows, is truly an important matter and should never be underestimated. Think back to a program you have used that was so incredibly slow the hourglass was displayed more often than the normal arrow. Then think about programs you have run where there are only a few brief moments where you had to wait while the hourglass was visible. Which program seemed faster to you?

Changing a pointer's shape communicates the state of the program to the user. Not changing the pointer communicates a completely different message. Have you ever used a program that was carrying out a long operation but did not display an hourglass to notify you of that fact?

If you are like me, you think of the program as being sloppy or poorly written. Such sloppy coding will cause the user to perceive your program as slow, and changing that perception is very difficult once it is in the user's mind.

The MousePointer property can be used on three distinct types of objects in Visual Basic: screen objects, form objects, and most control objects. Setting the screen's MousePointer property controls the pointer used everywhere on the visible screen (even when the mouse is over other applications' windows). This property will override the mouse pointer setting for individual forms and controls. The Form object's MousePointer property controls the pointer only while it is over the form whose property was set. Additionally, most VB controls have a MousePointer property. Setting a control's mouse pointer overrides the Form's mouse pointer only while the pointer is over the control.

Most programmers usually display the hourglass during processes that take a few seconds or longer to complete. This time factor is usually (and erroneously) based on the time it takes to complete a task on the programmer's machine, not the customer's machine. You've heard this before, but let me say it again: when in doubt, always assume your customers have the slowest machines in existence when coding.

When designing a program, most programmers generally test their code with the minimal amount of steps required to test all aspects of their code. Programmers like to program, and most programmers I've met dislike extensively testing what they have programmed, lest they find bugs in their magnificent code.

In a database application I developed a while back, I tested my code by creating a few test records and trying all of the available operations on those records. This tested my code sufficiently to ensure my algorithms worked as designed, but it failed to stress-test my application and provide reasonable performance benchmarks.

Operations that took less than a second on my development machine (a

Pentium 90 with 32 MB of RAM) took tens of seconds and sometimes longer on my customer's machines. I did not implement the hourglass in many of my routines because to me they were instantaneously completed. Changing to an hourglass would only result in a fast flash as the arrow turned to an hourglass and then back to the arrow.

My customers paid the price when they stared at screens with a normal arrow for the mouse pointer, wondering what was going on in the background. Did the program lock up? Ultimately the programmer pays the price for this type of shortsighted coding.

To ensure that the cursor changes when a procedure is being executed, you can wrap all of your code in an hourglass routine that displays the hourglass upon startup and restores the hourglass to the previous pointer upon completion.

Some would argue that procedures that are called only from within other procedures do not require hourglass management if their calling procedure already handles it. This may be true, but using the techniques I'll illustrate, you can effectively add hourglass management to all of your procedures without any adverse side effects. By protecting all of your procedures you ensure your code will always handle the hourglass properly, even when major changes are made to your application.

So you've finished coding your masterpiece and you've included basic hourglass management. Upon starting a procedure you display the hourglass, and when the procedure is complete you set the mouse pointer back to the default pointer or the arrow.

Is this all you need to do? Let's look at a common scenario that illustrates a problem with this basic approach to mouse pointer management.

The code shows two subroutines, TestSubOne and TestSubTwo. TestSubTwo is called by TestSubOne about halfway though the subroutine. Notice the hourglass code at the top and bottom of each procedure. See if you can find the trap this coding method springs on the unsuspecting programmer (see Listing 1).

TestSubOne starts by setting the mouse pointer to an hourglass. The program continues into the next part of the subroutine and some other code is executed. Next, the program calls TestSubTwo to perform

```
Sub TestSubOne ()
  'Show the MousePointer
  'Screen.MousePointer = HOURGLASS
  'Some code here.
  .
  .
  'Call the other routine now
  Call TestSubTwo

  'More code here.. and this code
  'takes a while to complete......
  .

    'Hide the hourglass and
    'restore the arrow now
    Screen.MousePointer = ARROW
End Sub
Sub TestSubTwo ()
  'Show the hourglass
  Screen.MousePointer = HOURGLASS
  'This subroutine performs some
  'operation and then ends
  'Your code here........

  'Restore the hourglass to an
  'arrow now
  Screen.MousePointer = ARROW
End Sub
```

LISTING 1 ***It Isn't Right to Point.*** *The most common (and flawed) approach to hourglass management: If the pointer shape is changed from an arrow to an hourglass in TestSubOne, and another routine requiring a shape change in the cursor is called, the procedure is foiled.*

some other operations. Notice that the first thing TestSubTwo does is change the mouse pointer to an hourglass.

This is done even though the mouse pointer is already an hourglass. This approach is not necessarily bad, but it contributes to this code's hourglass problem.

At the bottom of TestSubTwo, the mouse pointer is returned to an arrow. There is nothing specifically wrong with this, unless the procedure was called from within another procedure. With the mouse pointer as an arrow, the program returns to the middle of TestSubOne.

Now the long operation processes and makes the user wait for it to complete, and all with an ordinary arrow as a mouse pointer. Calling TestSubTwo has foiled the hourglass management of this scenario.

TestSubOne does not know that TestSubTwo turned the hourglass into an arrow and assumes the mouse pointer is an hourglass. This enduring problem need not plague programmers any longer: using either of my two hourglass management routines will cure this problem and help you present a professional and well-polished look and feel to the user.

KEEPING TRACK OF THE SHAPE

Now let's examine the two different mouse pointer management methods and explore their benefits and drawbacks. These methods are known as buffered management and reference count management. Both provide effective and professional mouse pointer management, but with clearly different implementations and requirements on the programmer.

In the buffered management technique, every procedure buffers the mouse pointer in use before your code started, and displays an hourglass while your code executes. Upon ending, the program restores the buffered mouse pointer, returning it to how it was before your code started.

Let's look at the buffered management approach by examining some simple code that shows it in action:

```
Sub TestSubOne ()
   'Buffer for the MousePointer
   Dim iOldPointer As Integer
   'Buffer the current MousePointer
   iOldPointer = Screen.MousePointer
   'Change to an hourglass
   Screen.MousePointer = HOURGLASS

   Your code goes here...

   'Restore the MousePointer
   Screen.MousePointer = iOldPointer
End Sub
```

One thing to notice in the sample is the introduction of the variable iOldPointer. This variable is the buffer used to store the mouse pointer in use before it is changed to an hourglass.

If TestSubTwo is modified to handle the mouse pointer in the same way TestSubOne does, you will have a consistent and seamless hourglass between the two subroutines. The biggest concern here is that every procedure using this technique must have a local integer variable to buffer the previous mouse pointer. Each procedure must have its own buffer variable because the previous mouse pointer can vary across procedures.

This approach to tracking the mouse pointer shape is efficient, but each procedure requires extra time to implement (two extra lines of code for each procedure, four if you count the comments).

If you have a large number of routines that need mouse pointer management, you may be interested in the time

savings offered by the Reference Count method instead.

The reference count management technique keeps a count of the times the hourglass is displayed versus the number of times it is returned to an arrow. This technique assumes the hourglass will be visible if any procedures have requested that it be shown, and it has not yet been restored to an arrow.

The Reference Count method keeps track of the mouse pointer by examining the number of times the hourglass was called versus the number of times it was restored to an arrow. Each time an hourglass is needed, the variable that stores the reference count of the hourglass is incremented by one.

Likewise, the same reference count is decreased once for each call to restore the mouse pointer to the arrow. This process works just like the reference count for a Windows DLL.

Each time an application loads the DLL, the reference count is increased by one. When all applications have finished and decreased the reference count accordingly, the DLL is no longer needed and is unloaded from memory (see Listing 2).

The first step in the reference count method is to create two subroutines that will manage our reference counting for us. Both refer to the variable iHourGlassCount. This variable can be declared globally, or better yet, the code can be placed in a module that encapsulates the iHourglassCount and iOldPointer variables. After the variable is declared, create a new module by selecting New Module from the File menu.

Add these two subroutines to the module, and create the two module-level variables by placing these dimensions in the declarations procedure of the new module:

```
Dim iHourglassCount as Integer
Dim iOldPointer as Integer
```

These variables are module-level and only accessible by procedures found in the module. This technique is an easy way to apply object-oriented programming techniques to the traditionally non-object-oriented VB development environment.

```
Sub ShowHourglass ()
  'Increase the hourglass count by 1
  iHourglassCount = iHourglassCount + 1
  'If this is the first call then buffer
'the MousePointer before we change it
  If iHourglassCount = 1 Then
    'Buffer the MousePointer
    iOldPointer = Screen.MousePointer
  End If
  'If the cursor isn't an hourglass then make it so
  If Screen.MousePointer <> HOURGLASS Then
    Screen.MousePointer = HOURGLASS
  End If
End Sub

Sub ReleaseHourglass ()
  'Decrease the Hourglass reference count by 1
  iHourglassCount = iHourglassCount - 1
  'If there are no more requests then release
  'the hourglass
  If iHourglassCount < 1 Then
    'Restore the pointer original pointer
    Screen.MousePointer = iOldPointer
  End If
End Sub
```

LISTING 2 ***Counting Cursor Changes.*** *This listing contains the core code required by the reference count technique. Each time the cursor shape is changed, the reference counter is increased by one, and when unloaded, it is decreased by one.*

```
Sub TestSubOne ()
  'Show the MousePointer
  Call ShowHourglass
  'Some code here.

  'Call the other routine now
  Call TestSubTwo
  'More code here.. and this code takes a while to
  'complete......
  .
  'Release the hourglass now
  Call ReleaseHourglass
End Sub
Sub TestSubTwo ()
  'Show the hourglass
  Call ShowHourglass
  'This subroutine performs some operation
  'and then ends
  'Your code here........
  'Restore the hourglass to an arrow now
  Call ReleaseHourglass
End Sub
```

LISTING 3 ***This One Counts.*** *This code is similar to the hourglass management implementation in Listing 1, only using the reference count hourglass management scheme. This code handles pointer changes faster and in fewer lines than Listing 1.*

THE SHAPE OF THE MOUSE POINTER HAS A SUBSTANTIAL IMPACT ON THE USABILITY OF YOUR PROGRAMS.

Let's revisit the TestSubOne and TestSubTwo subroutines again, this time with the reference count approach to mouse pointer management (see Listing 3). This technique is much easier to implement and requires fewer lines than the buffered approach. Programmers will find it simple to use VB's replace feature to replace their existing hourglass code with the reference count code lines.

One exception to hourglass management that must be handled manually by the programmer is user input. Code that requires interaction from the user should appropriately change the mouse pointer to an arrow until the user has completed the input process, then restore the hourglass when input is complete.

In these instances, it is appropriate to explicitly set the mouse pointer to the desired shape. In my example, a subroutine displays a message box while temporarily handling the hourglass. Because the hourglass is in use when it is temporarily changed to an arrow, hard-coding it is acceptable.

If the exact state of the mouse pointer is unknown when allowing input, use the buffered management approach to handle the temporary mouse pointer change. You will never go wrong if you

use some buffered management routines in conjunction with the reference count method when handling input from the user (see Listing 4).

Prematurely exiting a procedure requires that you restore the hourglass to the appropriate state before you leave, or the user may be faced with a perpetual hourglass, even when the program is idle.

Here is a sample that will accidentally strand the hourglass as the mouse pointer when DataIsComplete is True:

```
Sub IsDataOK
   Screen.MousePointer = HOURGLASS
   If DataIsComplete then
      Exit Sub
   Else
      MsgBox "Your data is _
         incomplete", MB_OK, "Oh Oh"
   End If
   Screen.MousePointer = ARROW
End Sub
```

By placing code immediately before the Exit Sub, this problem can be avoided.

DON'T TOUCH THAT BUTTON!

So you've finished coding your program and everything works as planned. The user can click on a button and begin a long operation, promptly changing the mouse pointer to an hourglass to notify the user of the lengthy process. What happens if the user clicks on the same button he just clicked on while the mouse pointer is an hourglass? Is it discarded or is it processed?

In this case, the click is processed just like any other event in Visual Basic. But if the cursor is an hourglass, doesn't that mean "wait until I am done"?

In Visual Basic the hourglass is basically just a picture, no more than one of many available mouse pointers. Visual Basic does not care what the mouse pointer is set to: it's business as usual. Now, think about how your code is written. What happens if the user clicks on a button that starts a long task, and then presses it again, causing two instances of the procedure to be executing simultaneously.

Most procedures aren't designed to handle this type of action. You will probably run into a trappable error that promptly brings down your application. You can even start a recursive process and be greeted by an "Out of Stack Space" error.

Then how do you disable input to all of the controls on a form during long operations? Do you set the Enabled property of all of your controls to False? This causes the control to ignore input, but at the expense of graying the text and changing the physical look of the controls unnecessarily. Simply change the Enabled property of the form to False to disable all controls on the form.

When a window is disabled, its controls are still completely functional, though limited to the user. Control events can still be generated programmatically, even when the control they are generated by is disabled to the user.

A good example is the Click event of a command button. Setting the Value property of the command button will still fire the Click event, as if the user clicked on the button. Tying up these loose ends will allow your program to accurately communicate its current state to the user, and account for the overexcited user who doesn't believe the hourglass is what it seems. ■

```
Sub ConfirmAddress
   'Request the hourglass
   Call ShowHourglass
   'Some code here
   .
   'Temporarily show the hourglass
   iOldPointer = Screen.MousePointer
   Screen.MousePointer = ARROW
   sMessage = "Are you sure you want to exit?"
   iReturnValue = MsgBox(sMessage, MB_OK, "Please Confirm")
   'Return the hourglass to the previous pointer
   Screen.MousePointer = iOldPointer
   'Do something here
   .
   'Release the Hourglass
   Call ReleaseHourglass
End Sub
```

LISTING 4 ***Hybrid Pointer Management.*** *This example combines the benefits of the reference count and buffered mouse management methods during user input. While the code handles the cursor changes, the programmer still needs to account for excess Click events while the program is executing.*