PROGRAMMING TECHNIQUES

Don't Be Square

Click & Retrieve Source CODE!

Who said windows must be rectangular? Break out of the rectangular rut.

by Karl E. Peterson

It's been more than 10 years since Microsoft released Windows 1.0. Through most of that time, the windows that composed Windows have been rectangular, and they cascaded or tiled in a neat, predictable order. Why not break out of that rectangular rut? I'll show you how to create a form that you can shape in any way imaginable (see Figure 1). The technique is available only in Windows NT and Windows 95, but it is equally accessible from VB3 (using CALL32.DLL, discussed in the *VBPJ* July 1995 Programming Techniques column) or VB4.

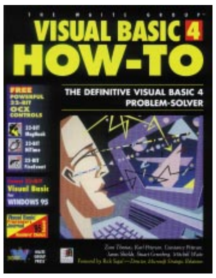
Windows NT 3.51, and later Windows 95, introduced the SetWindowRegion API function. This function instructs the operating system to draw any portion of a window inside the indicated region only—no going outside the lines, so to speak.

In Windows a region can be a rectangle, ellipse, or polygon, or a combination of two or more of these shapes. You can use regions for a variety of purposes. Regions may be filled, outlined, inverted, framed, or used for hit-testing (testing cursor location), among other things.

All windows have clipping regions that define the area in which Windows allows drawing to occur. Clipping regions are updated whenever a window is resized, and when another window partially covers or uncovers a window. SetWindowRegion further restricts drawing to just within the region passed to it.

You create regions by passing descriptive coordinates to one of a handful of functions. You pass the CreatePolygonRgn API an array of POINTAPI structures defining a polygonal region. Windows assumes the polygon is closed. You also specify the polygon fill mode—Alternate or Winding—to choose the method Windows will use when painting the polygon. For demonstration purposes, the two modes are similar. Alternate seems to be somewhat faster, however. If the call is successful, CreatePolygonRgn returns a GDI handle to the newly created region.

Karl E. Peterson is a GIS analyst with a regional transportation planning agency and a member of the Visual Basic Programmer's Journal *Technical Review Board. Based in Vancouver, Washington, he's also an independent programming consultant and a writer. Karl coauthored* Visual Basic 4 How-To, *from Waite Group Press. Online, he's a section leader in the* VBPJ *Forum's 32-Bit Bucket and a Microsoft MVP in the MSBASIC Forum. Contact Karl in either CompuServe location at 72302,3707.*



When you create GDI objects such as pens, brushes, and regions, you typically delete them after you use them. However, when you pass a region handle to SetWindowRgn, the operating system takes control of that object, and you *must not* attempt to use it for anything, or destroy it, after this call. To set a form's window region back to normal, pass a NULL (ByVal 0&) as the region handle. The sample project uses a command button to toggle a form's window region between its normal rectangular shape and a star-shaped polygon (see Listing 1).

To jazz up the demo, the Form_Paint event paints a bright red, star-shaped polygon in the middle of the form using only API calls. First, you create a red brush with CreateSolidBrush. Then, using CreatePolygonRgn, create a region to match the form's window region. Call FillRgn to fill the polygon using the brush passed as the third parameter (see Listing 2). At that point, because they won't be used again, delete the brush and region with calls to DeleteObject. Finally, use the Polyline function to outline the polygon.

Notice that you use a different point array with FillRgn than with SetWindowRgn. Window regions use coordinates relative to the upper-left corner of the window, while drawing functions use coordinates relative to the device context. In this case, the device context of the form is equivalent to the form's client area. You need to calculate two point arrays if you want to draw within the window region you set for your form. Or, at the very least, you need to offset the window region's points from your form's coordinates to reflect the different coordinate system's origins. The easiest way to do this is to add the values for border width and caption height returned by GetSystemMetrics to the coordinates you'll use for drawing on the form (see Listing 3). Set the form's Scalemode to Pixels, because that's the coordinate system GDI uses.

Due to space limitations, the complete project is not printed here. You can download it from the Magazine Library of the *VBPJ* Forum on CompuServe. Search for the file PT0496.ZIP, which

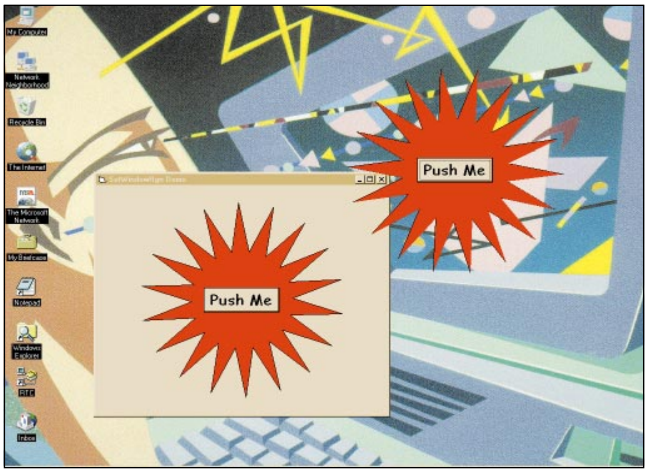


FIGURE 1 ***Shape Up Your Windows.*** *Two instances of the WinRgn demo show it with and without active window region clipping. The command button toggles between these two states.*

contains not only this project but all the code from this column. You can also download the code from *VBPJ's* WWW and MSN sites (for details, see the "How to Reach Us" section in Letters to the Editor). If you decide to build a project using this technique, you'll probably want to add support for dragging the form without the caption bar. For two approaches to this, see either the Microsoft Knowledge Base article Q114593 ("How to Move a Form that Has No Titlebar or Caption") or my book, *Visual Basic 4 How-To* from Waite Group Press.

ANYONE REMEMBER TRIG?

I had to ask that question on CompuServe recently, as my memory had failed me. I needed the formula for calculating an (x, y) pair, given a starting position and a vector (direction and distance). Within a short time, several people responded. I found the formula useful, so I will repeat it for the rest of you whose trigonometry textbooks are also rotting in a box out in your garage. The basic formulas are:

```
x2 = x1 + (Radius * Sin(Theta))
y2 = y1 + (Radius * Cos(Theta))
```

Where (x1, y1) is the starting point, Radius is the distance to move, and Theta is the angle expressed in radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi. I put this technique to use in the CalcRgnPoints routine that calculates the points of the star (see Listing 3).

CONTAINING THE CURSOR WITHIN A FORM OR CONTROL

Occasionally a situation requires a response from the user. One way to get the user's attention is to restrict the cursor's range of movement. Windows provides the ClipCursor API call that performs this cursor clipping. I've built both VB3 and VB4 modules that allow you to define which control or form you'd like the cursor

VB4

```
Private Sub Command1_Click()
  Dim hRgn As Long
  Static UsingPoly As Boolean
  ' Flag variable tracks current state.
  UsingPoly = Not UsingPoly
  If UsingPoly Then
     ' Create a region, then turn on
     ' clipping to that region.
     hRgn = CreatePolygonRgn(rgnPts(0), nPts, ALTERNATE)
     Call SetWindowRgn(Me.hWnd, hRgn, True)
  Else
     ' Turn off clipping.
     Call SetWindowRgn(Me.hWnd, 0&, True)
  End If
End Sub
```

LISTING 1 ***Create, Clip, and Toggle.*** *It takes only two API calls to create a region and tell Windows not to draw any portion of the window that lies outside this region. The command button acts as a toggle, alternately turning on and off the special effect. Listing 3 shows how the point array was derived.*

VB4

```
Private Sub Form_Paint()
  Dim hBrush As Long
  Dim hRgn As Long
  ' Create region and a brush to fill it with.
  hBrush = CreateSolidBrush(vbRed)
  hRgn = CreatePolygonRgn(scnPts(0), nPts, ALTERNATE)
  Call FillRgn(Me.hdc, hRgn, hBrush)
  ' Clean up GDI objects.
  Call DeleteObject(hRgn)
  Call DeleteObject(hBrush)
  ' Draw outline around polygon.
  Call Polyline(Me.hdc, scnPts(0), nPts + 1)
End Sub
```

LISTING 2 ***Draw Only Within the Window Region, Please.*** *Regions are also useful for painting polygons. Here, an offset array of points is used to compensate for the different origins of the form and its client area. Because the array already exists, the Polyline API provides the fastest way to outline it.*

VB4

```
Private Static Sub CalcRgnPoints()
  ReDim scnPts(0 To nPts) As POINTAPI
  ReDim rgnPts(0 To nPts) As POINTAPI
  Dim angle As Long, theta As Double
  Dim radius1 As Long, radius2 As Long
  Dim x1 As Long, y1 As Long
  Dim xOff As Long, yOff As Long
  Dim n As Long
  '
  ' Some useful constants.
  '
  Const Pi# = 3.14159265358979
  Const DegToRad# = Pi / 180
  '
  ' Calc radius based on client area size.
  '
  x1 = Me.ScaleWidth \ 2
  y1 = Me.ScaleHeight \ 2
  If x1 > y1 Then
     radius1 = y1 * 0.85
  Else
     radius1 = x1 * 0.85
  End If
  radius2 = radius1 * 0.5
  '
  ' Offsets to move origin to upper
  ' left of window.
  '
  xOff = GetSystemMetrics(SM_CXFRAME)
  yOff = GetSystemMetrics(SM_CYFRAME) + _
     GetSystemMetrics(SM_CYCAPTION)
  '
  ' Step through a circle, 10 degrees each
  ' loop, finding points for polygon.
  '
  n = 0
  For angle = 0 To 360 Step 10
     theta = (angle - offset) * DegToRad
     '
     ' First region is for drawing.
     ' One long, one short, one long...
     '
     If n Mod 2 Then
        scnPts(n).x = x1 + (radius1 * (Sin(theta)))
        scnPts(n).y = y1 + (radius1 * (Cos(theta)))
     Else
        scnPts(n).x = x1 + (radius2 * (Sin(theta)))
        scnPts(n).y = y1 + (radius2 * (Cos(theta)))
     End If
     '
     ' Second region is for clipping.
     ' Add offsets.
     '
     rgnPts(n).x = scnPts(n).x + xOff
     rgnPts(n).y = scnPts(n).y + yOff
     n = n + 1
  Next angle
End Sub
```

LISTING 3 ***Offset Plotting.*** *The coordinates of a window's region are relative to the upper-left corner of the window, while GDI drawing functions use coordinates relative to a window's client space. First, you generate one set of points for drawing, then you add offsets to each point so the window's region will coincide.*

restricted to (see Listing 4).

VB4 has added support for the Name property to code modules, and you may preface procedure names with their respective module name, therefore allowing duplicate procedure names in a single project. You can also use this feature to make your code more readable. In the VB4 demo for this topic, I've named the module "Cursor." As an example of how readable this new syntax makes code, consider these calls made after a command button click (see Figure 2):

```
Private Sub Command1_Click()
   Cursor.RestrictToControl Picture1
   Cursor.CenterOnControl Picture1
End Sub
```

The RestrictToControl procedure calls the GetWindowRect API function to determine the size and position of the control. Not all controls support the hWnd property used by the GetWindowRect API, though. If a lightweight control (such as a label, image, or shape) is passed to RestrictToControl, an error trap catches it and gracefully exits the procedure. If no error occurs, the RestrictToControl procedure calls the more generic RestrictToRect procedure, passing the rectangle returned by GetWindowRect. To put the cursor clipping into effect, RestrictToRect calls the ClipCursor API.

In this demo, double-clicking on the picture box releases the cursor clipping. The Release procedure is called from the Picture1_DblClick event, and it in turn calls ClipCursor again. To release cursor clipping, pass a NULL pointer in place of the clipping rectangle:

```
Call ClipCursor(ByVal 0&)
```

Things get just a little trickier if you want to restrict the cursor to your entire form. The problem occurs if the user tries to resize or move the form. This action can release the cursor clipping in certain circumstances. So, in the Cursor module, the Restrict-ToForm procedure sets the clipping rectangle equal to the form's client area by calling the GetClientScrnRect procedure. The GetClientRect API function won't do in this situation because the clipping rectangle must be expressed in screen coordinates. GetClientRect returns the width and height of a client window, but not its left or top positions. Instead, you use GetWindow-Rect, then you call GetSystemMetrics to make adjustments to eliminate the borders and caption (this code is not printed in this column, but you may download it from the online services described elsewhere
in the magazine).

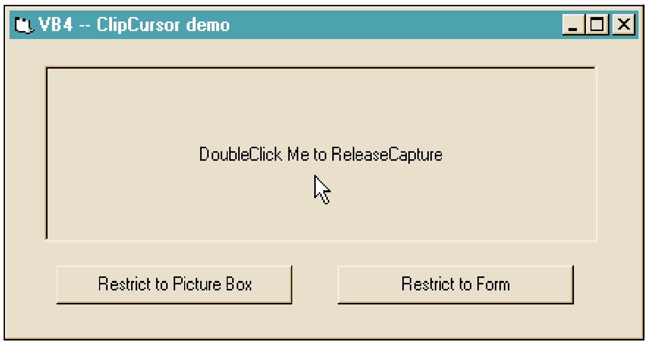


FIGURE 2 ***Cursor Clipping, On Command.*** *Pressing the command buttons in this project restricts the cursor to either the picture box or the form. Double-clicking on the picture box releases the cursor, allowing you to move it anywhere on screen.*

ERASING THE DEBUG WINDOW

VB3's Debug window is a useful tool for outputting data while

VB4

```
Option Explicit
' Win32 API Declarations, Type
' Definitions, and Constants
Private Type RECT
   left As Long
   top As Long
   right As Long
   bottom As Long
End Type

Private Declare Function SetCursorPos Lib "user32" _
   (ByVal x As Long, ByVal y As Long) As Long
Private Declare Function ClipCursor Lib "user32" _
   (lpRect As Any) As Long
Private Declare Function GetWindowRect Lib "user32" _
   (ByVal hwnd As Long, lpRect As RECT) As Long

Public Sub RestrictToControl(cntl As Control)
   Dim r As RECT
   ' This routine only accepts controls
   ' which support the hWnd property.
   ' Handle errors by ignoring them.
   On Error Resume Next
   Call GetWindowRect((cntl.hwnd), r)
   If Err.Number = 0 Then
      Call Cursor.RestrictToRect(r)
   End If
End Sub

Public Sub CenterOnControl(cntl As Control)
   Dim r As RECT
   ' This routine only accepts controls
   ' which support the hWnd property.
   ' Handle errors by ignoring them.
   On Error Resume Next
   Call GetWindowRect((cntl.hwnd), r)
   If Err.Number = 0 Then
      Cursor.CenterOnRect r
   End If
End Sub

Private Sub CenterOnRect(lpRect As RECT)
   ' Use API to place cursor at center
   ' of rectangle.
   Call SetCursorPos(lpRect.left + _
      (lpRect.right - lpRect.left) \ 2, _
      lpRect.top + (lpRect.bottom - _
      lpRect.top) \ 2)
End Sub
```

LISTING 4 ***Contain Your Cursor.*** *You can use this module to center and/or contain the cursor within either a form or control. Download a separate VB3 project with the same capabilities along with this month's code from the* VBPJ *Forum on CompuServe, or the* VBPJ *WWW or MSN sites described elsewhere in this column (search for PT0496.ZIP).*

you're testing a program. It's often convenient to have the contents of the Debug window stay intact between invocations of the program, but at other times it would be nice to have a simple method for clearing its contents so new output will stand alone. I'll show you a simple routine I use when I want to repeatedly output new data with each run.

The only secret involved is the class name used by VB3's Debug Window—OFEDT. Given that, you can use the FindWindow API to obtain its handle. If the FindWindow call is successful, the last preparation for clearing the Debug window is to obtain the handle of the text box it uses to display its output. Think of the Debug window as a one-control form, with the text area its only child. A call to GetWindow, passing the Debug window's handle and using the GW_CHILD option, retrieves the handle for the text box. At this point, one more call to SendMessage finishes the job. Passing the WM_SETTEXT message to the text box, along with new text to display, will replace whatever was there. To clear the Debug window, simply pass an empty string.

A few words of precaution are in order. The DebugClear routine will be reliable only if one instance of VB3 is running (see Listing 5). In addition, it shouldn't be called from a compiled EXE. If it were, most likely nothing bad would happen. But, if your application is running on another developer's machine, which also happens to be running VB3, you may unintentionally wipe out the wrong Debug window. Still, in many cases, you will find uses for DebugClear. I used it in a set of benchmarks where a new "report" would be output to the Debug window with each successive run. Clearing the last run at the start of each new set of timings provided nice clean output.

VB4

```
Declare Function FindWindow Lib _
  "User" (ByVal lpClassName As Any, _
  ByVal lpWindowName As Any) As Integer
Declare Function GetWindow Lib _
  "User" (ByVal hWnd As Integer, _
  ByVal wCmd As Integer) As Integer
Declare Function SendMessage Lib _
  "User" (ByVal hWnd As Integer, _
  ByVal wMsg As Integer, ByVal _
  wParam As Integer, lParam As Any) _
  As Long

Global Const GW_CHILD = 5
Global Const WM_SETTEXT = &HC

Sub DebugClear ()
  Dim hDebug As Integer
  Dim hDebugTxt As Integer
  Dim nRet As Integer
  '
  ' Search for handle to Debug
  ' window.
  '
  hDebug = FindWindow("OFEDT", 0&)
  '
  ' If found, get handle to first
  ' child window, the actual
  ' textbox, and use SendMessage to
  ' set it to "".
  '
  If hDebug Then
     hDebugTxt = GetWindow(hDebug, _
        GW_CHILD)
     nRet = SendMessage(hDebugTxt, _
        WM_SETTEXT, 0, ByVal "")
  End If
End Sub
```

LISTING 5 ***Clearing VB3's Debug Window.*** *Place this code into a new module, and then include it in your project whenever you'd like to have the capability of clearing VB3's Debug window for new output.*