Who Said It's Hip To Be Square?

by Steven Colagiovanni

In the summer of 1986, the song "Hip To Be Square" by Huey Lewis and The News was a favorite on the American pop music charts. Being square might be OK for Huey Lewis, but what about windows? Since the premiere of Windows 1.0, windows have always been square or rectangular. Do our windows always have to be rectangular?

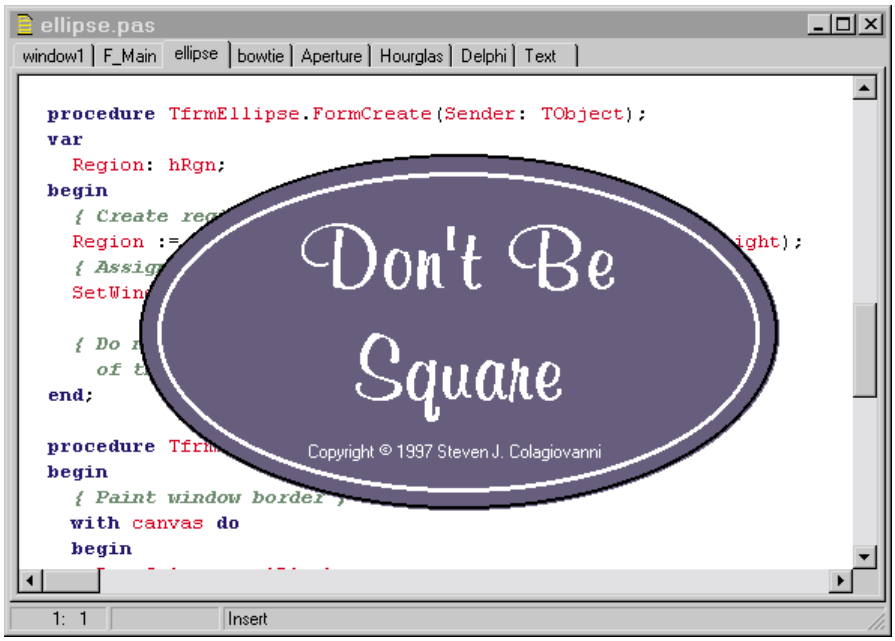
I'm going to tell you that the answer is no! At least not in Windows 95 or NT. In Windows NT 3.51 the Win32 API introduced the `SetWindowRegion` function. This function instructs the operating system to draw any portion of a window inside the indicated region only. Since this technique is only available in Windows NT and Windows 95, you'll need either Delphi 2 or 3. One sample of this technique is the clock included with the Microsoft PowerToys add-on for Windows 95. This clock can have a traditional round clock face and shape and be dragged all over the screen.

In Windows a region can be a rectangle, ellipse, polygon or a combination of two or more of these shapes. The Win32 API allows us to create any shaped region we conceive. These regions can be filled, outlined, inverted, framed, or used for hit-testing (testing mouse down location), among other things. Once you have defined a region, you pass it to the `SetWindowRgn` API function. This makes it possible to have windows that are anything but square.

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. `SetWindowRgn` further restricts drawing to just within the region passed to it.

I probably wouldn't want to have a database application that is shaped like a Christmas tree, but there are many possibilities where you might want a window that is, say, round. Suppose we were designing a new war game using Delphi. In one corner of the screen we could display a radar screen or resource gauge that is round. By making this a round window, the user could drag it to anywhere on the screen and it would block about 30% less of the screen than a square window.

Another place where an oddly shaped window might be appropriate is as a splash screen. I use splash screens in all my applications. This gives the user something to look at while the program loads, and also gives me an opportunity to take credit for all my hard work. Instead of placing the client's logo in a rectangular window (along with our copyright line), we could build a splash screen that is also shaped like the client's logo. It's sad to say, but customers are not impressed that we spend hours tweaking the code to improve performance, or that all our database tables are normalized. However, give them something like a custom shaped splash screen, and you'll hear lots of oohs and ahs.



➤ *Figure 1*

```
procedure TfrmEllipse.FormCreate(Sender: TObject);
var
  Region: hRgn;
begin
  { Create region, or window boundaries }
  Region := CreateEllipticRgn(0, 0, Width, Height);
  { Assign the region to the window }
  SetWindowRgn(Handle, Region, True);
  { Do not delete region - Windows now has control of the region. }
end;
```

➤ *Listing 1*

Simple Shapes

Let's assume that we are building an application for a company whose logo is always displayed against a blue oval. We can have an oval shaped window with the company logo inside along with our copyright (see Figure 1). In the

FormCreate event, place the code in Listing 1. When the window is created, an elliptic region (oval) is created with the CreateEllipticRgn API call, where nLeftRect and nTopRect are respectively the *X* and *Y* cordinates of the upper-left corner, and nRightRect and nBottomRect are respectively the *X* and *Y* coordinates for the lower-right corner. All these parameters are integers.

That region is then assigned to the window, which takes on the shape of our oval. If you want the user to be able to resize the window, you will have to repeat or call this code in the FormResize event, so that the oval can be resized.

Normally, when you create GDI objects such as pens, brushes and regions, you typically delete them. Fortunately the VCL takes care of this for us. 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.

After the oddly shaped window is created, you then have to draw what you want visible on that window. Either the form's components are automatically painted, or we can place code in the FormPaint event to paint and draw what we want to display on the window. The black line around the outside of the oval is added in the FormPaint event. See Listing 2.

You will find that writing code to paint the form produces a faster display than placing several graphics components on the form. For the oval I used labels with a shape component to demonstrate the speed difference. Compare the painting of this form with several of the other examples in the demo program.

Polygons

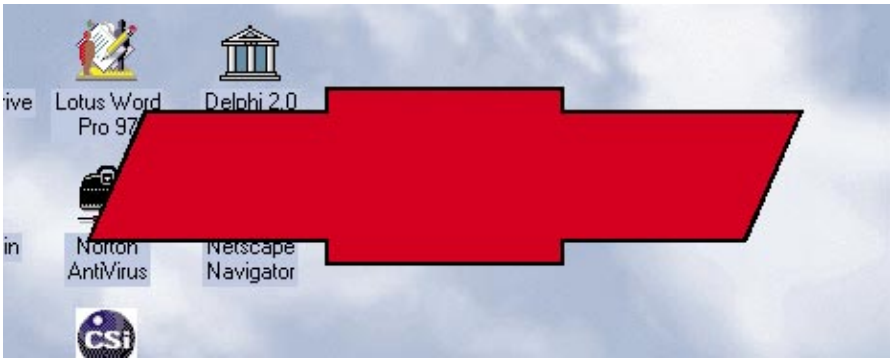
Ovals and circles are easy. For a circle, just make certain that the width and height of the Elliptic region are equal. What about more complex shapes? We can also create polygons and rectangles or squares with rounded corners. The methods for all of these are listed in Table 1. Information on

```
procedure TfrmEllipse.FormPaint(Sender: TObject);
begin
  { Paint window border }
  with canvas do
  begin
    Pen.Color := clBlack;
    Pen.Width := 2;
    Ellipse(1, 1, width-2, height-2);
  end;
end;
```

➤ *Listing 2*

```
procedure TfrmBowTie.FormCreate(Sender: TObject);
var
  Region: hRgn;
  RgnPts: array[0..11] of TPoint;//Window region (outline)
Const
  nPts: integer= 12;
begin
  { Set points of polygon for window border }
  RgnPts[0] := Point(30, 12);
  RgnPts[1] := Point(125, 12);
  RgnPts[2] := Point(125, 0);
  RgnPts[3] := Point(250, 0);
  RgnPts[4] := Point(250, 12);
  RgnPts[5] := Point(375, 12);
  RgnPts[6] := Point(345, 81);
  RgnPts[7] := Point(250, 81);
  RgnPts[8] := Point(250, 93);
  RgnPts[9] := Point(125, 93);
  RgnPts[10] := Point(125, 81);
  RgnPts[11] := Point(0, 81);
  { Create region, or window boundaries from the polygon }
  Region := CreatePolygonRgn(RgnPts[0], nPts, ALTERNATE);
  { Assign the region to the window }
  SetWindowRgn(Handle, Region, True);
end;
```

➤ *Listing 3*



➤ *Figure 2*

these methods can be found in the Win32 Programmer's Reference (WIN32.HLP) which is included with Delphi 2.

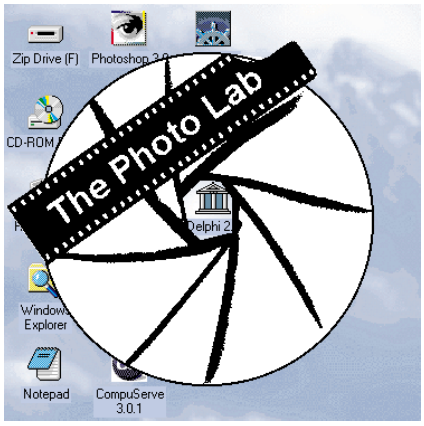
Creating a polygon is easy, but you have to plot all the points that define the polygon region. You pass the CreatePolygonRgn API function an array of TPoint structures defining the polygonal region and the number of points in the polygon. If you look at Listing 3, you'll notice that the 12 points of the polygon are assigned. Each one of these points is connected by a line, in the order given, with the last point being connected to the first. Windows assumes the polygon is closed. The last parameter of CreatePolygonRgn specifies the polygon fill mode which chooses the method Windows will use when painting the polygon. The mode is either WINDING or ALTERNATE.

I have found that ALTERNATE is slightly faster. The resulting window is shown in Figure 2.

Keep in mind that window regions use coordinates relative to the upper-left corner of the window, while drawing and painting functions use coordinates relative to the device context. The device context of the window is equivalent to the form's client

CreateEllipticRgn	Creates an elliptical region
CreateEllipticRgnIndirect	Creates an elliptical region given a RECT structure
CreatePolygonRgn	Creates a polygonal region
CreatePolyPolygonRgn	Creates a region based on multiple polygons
CreateRectRgn	Creates a rectangular region
CreateRectRgnIndirect	Creates rectangular region given a RECT structure
CreateRoundRectRgn	Creates rectangular region with rounded corners

➤ *Table 1: Methods for creating regions*



➤ *Figure 3*

area. If you want to draw within the window region, you need to offset the window region's points from your form's coordinates. One way to do this is to add the values for border width and caption height returned by the `GetSystemMetrics` function to the coordinates you'll use for drawing on the form. An easier way is to set the form's border style to `bsNone`.

Who Put A Hole In My Window?

The `CombineRgn` function allows us to combine regions in several ways. We can merge two regions so that the final region is the sum of both (`RGN_OR`). We can also remove a region from an existing region or window. If we create the first region as a circle and a second region that is a smaller circle in the center of the first, we can produce a final region that is shaped like a donut, complete with the hole in the middle. You can see the background through the hole and even click on icons and windows through the hole!

Figure 3 shows the logo for a fictitious photographic laboratory. The window was created by combining three regions. A polygon region (for the film strip) was combined with a circle (elliptical region) for the illustration of the lens aperture. This produced the outline of the window. Then the resulting region was combined with a small polygon near the center using the `RGN_DIFF` combine mode. This center polygon forms the opening in the center of the lens aperture and has resulted in a hole in our window.

Listing 4 shows the `FormCreate` event for this window. The `CalcRgnPoints` procedure places the points of the two polygons into two `TPoint` arrays. The region for the filmstrip is used by the `CreatePolygonRgn` function to create Region 2. Region 1 is a circle that is created with the `CreateEllipticRgn`. The two regions are merged into one with the `CombineRgn` function, using the `RGN_OR` combine mode. The `CreatePolygonRgn` function uses the `TPoint` array for the aperture and assigns this to Region 2. The `CombineRgn` function is called again, this time using the `RGN_DIFF` combine mode to punch a hole in the middle of the window. The resulting region is then assigned to the window. Table 2 lists the combine modes.

The logo itself is a bitmap, which is stored in an image component with its visible property set to false. The image is painted onto the form with the `Canvas.Draw` function. This was faster than allowing the image component to paint itself. The code for the `FormPaint` event is in Listing 5.

Including the Titlebar

We can also include the titlebar in oddly shaped windows. Figure 4

➤ *Listing 4*

```
var
  RgnPts: array[0..6] of TPoint; // Outline of hole
  FlmPts: array[0..7] of TPoint; // Outline of film strip
Const
  rPts: integer = 7;
  fPts: integer = 8;
procedure TfrmAperture.FormCreate(Sender: TObject);
var Region1, Region2: hRgn;
begin
  { Construct Polygons for film strip and hole }
  CalcRgnPoints;
  { Create first region, the circle }
  Region1 := CreateEllipticRgn(30, 10, ClientWidth, ClientHeight);
  { Create second region, the polygon for the film strip }
  Region2 := CreatePolygonRgn(FlmPts[0], fPts, ALTERNATE);
  { Combine the regions, into one region }
  CombineRgn(Region1, Region1, Region2, RGN_OR);
  { Create third region, the hole in the center }
  Region2 := CreatePolygonRgn(RgnPts[0], rPts, ALTERNATE);
  { Create a region that consists of the current region,
      minus the third region (the hole) }
  CombineRgn(Region1, Region1, Region2, RGN_DIFF);
  { Assign the region to the window }
  SetWindowRgn(Handle, Region1, True);
end;
```

➤ *Listing 5*

```
procedure TfrmAperture.FormPaint(Sender: TObject);
begin
  with canvas do begin
    // copy image to window
    Draw(0, 0, Image1.Picture.Bitmap);
    Brush.Style := bsClear;
    { Outline circle for better visibility }
    Pen.Color := clBlack;
    Pen.Width := 2;
    Ellipse(31, 11, width-1, height-1);
  end;
end;
```

shows an oddly shaped window with a title bar. The title bar was included within a polygon outline that was passed to the `CreatePolygonRgn` function.

The Width of the titlebar is obviously the same as the form's width, but the height of the titlebar is the sum of the thickness of the top window frame and the height of the window caption. Fortunately, Windows will tell us what these dimensions are with the `GetSystemMetrics` function:

```
TitlebarHeight :=
  GetSystemMetrics(SM_CYCAPTION) +
  GetSystemMetrics(SM_CYDLGFRAME)+1;
```

The form width and `TitlebarHeight` are then used to set the bottom right point of the title bar.

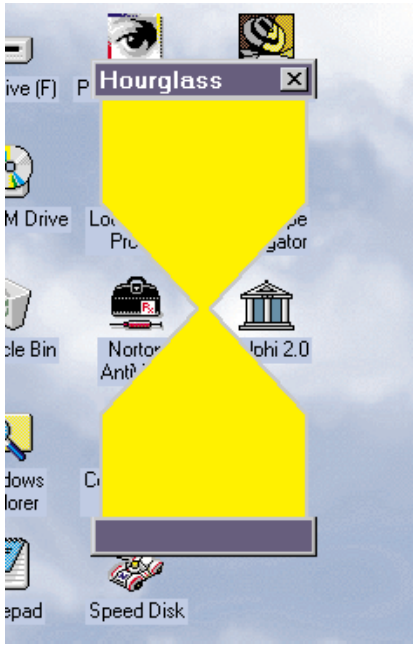
By now you may be wondering if an oddly shaped window can be returned to its standard rectangular shape. By calling `SetWindowRgn` and passing `NULL` (zero in Delphi's case) as the region, the window returns to its rectangular shape. The mouse down event for the hourglass window in Figure 4 returns the window to its rectangular shape. If the right mouse button is clicked, then `SetWindowRgn(Handle, 0, True)` is executed.

Creating Complex Shapes

So far we have covered simple shapes and combinations of these simple shapes. Some corporate logos can be complex designs involving straight lines and curves. They can also have two parts that are not connected. In these cases we can create our window using paths. A path defines a shape as a series of drawing operations. Essentially, an outline of an area. We can draw these paths using a series of lines, curves and arcs. By combining these elements, we can outline anything. To create a path, call `BeginPath`, passing it the device context (`Canvas.Handle`) of the window. Once the program calls `BeginPath`, all drawing to that device context does not appear on the display (or the printer). Instead, the drawing forms the path. When the path is complete, simply call `EndPath`. When you call `EndPath` the path becomes the current path for the device context. We then turn the path (or paths) into a region and assign that region to the window. If you think this sounds easy, it is. The only difficult part is defining the path.

How would you draw the outline of the letter "D" using straight lines and curves? We'll use a sans-serif font (like Arial or Helvetica) to make the job easier. Obviously we need a vertical line on the left, that runs the height of the character. Extending to the right, on both the top and bottom of the vertical line, we should place two shorter horizontal lines. We then can connect the right ends of the two short lines with an arc. The arc should run clockwise, from the top to the bottom, starting at the 12 o'clock position and ending at the 6 o'clock position.

The code would look something like Listing 6. All the drawing should be done using the Windows API. Some of the functions are the same as Delphi's `canvas` functions, except they have one extra parameter. That parameter, which is the first one, is the handle of the device context we're drawing on.



➤ *Figure 4*

➤ *Table 2: Combine modes*

Value	Description
RGN_AND	Creates the intersection of the two combined regions
RGN_COPY	Creates a copy of the region identified by hrgnSrc1
RGN_DIFF	Combines the parts of hrgnSrc1 that are not part of hrgnSrc2
RGN_OR	Creates the union of two combined regions
RGN_XOR	Creates the union of two combined regions except for any overlapping areas

➤ *Listing 6*

```
procedure TfrmDelphi.FormCreate;
var
  RgnPts: array[0..12] of TPoint;
  Region1: hRgn;
Const
  nPts: integer = 3;
begin
  BeginPath(Canvas.Handle);
  { Outside 'D' }
  MoveToEx(Canvas.Handle, 49, 0, nil);
  RgnPts[0] := Point(62, 2);   // control point
  RgnPts[1] := Point(79, 18);  // control point
  RgnPts[2] := Point(81, 36);  // end point
  PolyBezierTo(Canvas.Handle, RgnPts[0], nPts);
  LineTo(Canvas.Handle, 81, 53);
  RgnPts[0] := Point(79, 70);  // control point
  RgnPts[1] := Point(64, 86);  // control point
  RgnPts[2] := Point(47, 88);  // end point
  PolyBezierTo(Canvas.Handle, RgnPts[0], nPts);
  RgnPts[0] := Point(47, 88);
  RgnPts[1] := Point(0, 88);
  RgnPts[2] := Point(0, 0);
  RgnPts[3] := Point(49, 0);
  PolylineTo(Canvas.Handle, RgnPts[0],4);
  . . .  { More drawing here } . . .
  { 'I' }
  RgnPts[0] := Point(464, 0);
  RgnPts[1] := Point(491, 0);
  RgnPts[2] := Point(491, 88);
  RgnPts[3] := Point(464, 88);
  Polygon(Canvas.Handle, RgnPts[0], 4);
  EndPath(Canvas.Handle);
  { Create region, boundaries from the Path }
  Region1 := PathToRegion(Canvas.Handle);
  { Assign the region to the window }
  SetWindowRgn(Handle, Region1, True);
end;
```

Since we're drawing on the canvas, we pass `Canvas.Handle` as the first parameter. By the way, the API offers some drawing functions that the `Canvas` in Delphi doesn't support. These functions may make the operation easier.

In Listing 6, I started at the top, and drew a bezier curve to almost half way down the right side. Why did I use `PolyBezierTo` instead of `Arc`? `Arc` is not available for constructing paths in Windows 95. It is, however, available in Windows NT. Table 3 outlines the subset of drawing functions available under Windows 95. All the functions listed in Table 3 are available for NT.

By using `PolyBezierTo` instead of `PolyBezier`, after drawing the curve the pen position is placed at the last point drawn. This eliminates having to keep calling `MoveTo`. After completing the arc, I then draw a very short vertical line on the right. This will make the letter look slimmer and taller. Another bezier curve is then drawn from this point to the bottom, in a clockwise direction. I then finish by drawing the three straight lines, first across the bottom, then up the left side, across the top, finally ending at the point where the first curve started from. This defines the path that outlines the letter "D".

➤ *Table 3: Drawing functions available for Windows 95 and NT*

These drawing functions define points in a path in Windows 95 and NT:	
CloseFigure	Closes an open figure in a path
ExtTextOut	Draws a character string using the currently selected font
LineTo	Draws a line from the current position up to, but not including, the specified point
MoveToEx	Updates the current position to the specified point and optionally returns the previous position
PolyBezier	Draws one or more Bézier curves
PolyBezierTo	Draws one or more Bézier curves
Polygon	Draws a polygon consisting of two or more vertices connected by straight lines
Polyline	Draws a series of line segments by connecting the points in the specified array
PolylineTo	Draws one or more straight lines
PolyPolygon	Draws a series of closed polygons
PolyPolyline	Draws multiple series of connected line segments
TextOut	Draws a character string using the currently selected font
These additional drawing functions are available in Windows NT only:	
AngleArc	Draws a line segment and an arc
Arc	Draws an elliptical arc
ArcTo	Draws an elliptical arc
Chord	Draws a chord
Ellipse	Draws an ellipse
Pie	Draws a pie-shaped wedge
PolyDraw	Draws a set of line segments and Bézier curves
Rectangle	Draws a rectangle
RoundRect	Draws a rectangle with rounded corners

Instead of making three successive `LineTo` calls, I could have used `PolyLineTo` instead. `PolyLineTo` takes an array of `TPoint`. The first element is the starting point for the first line. The second element is the end point of the first line, the third the end point of the second line, and so on. `PolyLineTo` essentially calls `MoveTo`, followed by one or more calls of `LineTo`.

Figure 5 shows Delphi spelled out in block letters. The outside and inside outlines of the letters "D" and "P" were created just like our example above. The letters "E", "L", "H" and "I" were created with the polygon function call. This created several complex paths on the canvas. This is also how we create a window with two or more shapes or sections that are not connected. The only thing left is to create a region from the path or paths. This is done with the `PathToRegion` function. We then call `SetWindowRgn` as before.

Using Text

There's an easier way to place text on the screen than forming each letter like we did in the above example. I did it the hard way above to show you how to combine straight lines and curves to create complex shapes. What's the easier way? Use fonts. You can output TrueType text right on the canvas. Each TrueType character is a vector outline: a series of lines and curves. Windows normally fills in this outline so you may never have realized that TrueType fonts are outlines. This is how Windows can create any size character, from 4 to 100 points or more, all with one outline for each character. With each point size the character is stretched proportionately and there are no jaggies to worry about.

Non-TrueType fonts, on the other hand, are bitmap images and are limited to only those sizes of characters stored in the font file. In larger sizes, these characters will have jaggies, as they are only designed to be used as screen fonts.

Figure 6 displays a logo that I remember seeing somewhere. The

logo is primarily made of text, with a few polygons. My apologies to our esteemed Editor for not matching the fonts precisely. I wanted to stay with the Windows default Arial and Times New Roman fonts that all readers would have installed on their computers. It's a fairly close match. Obviously, the two underlines are polygons. "THE" at the left is also a series of polygons. This allowed me to create the rotated text without the use of a `RotateText` function. It also allowed me to create the extra heavy (fat) strokes of the letters, something that the Arial font couldn't do. Part of the code is shown in Listing 7.

Don't bother setting a color for the font, it won't be used. Remember, once the program calls `BeginPath`, all drawing to that device context does not appear on the display. The window created will always have the color assigned to the `Form.Color` property.

If the letters are all the same color, just set the `Form.Color` property to the color you want. If you need to have two or more colors like I did here, in the `FormPaint` event, paint the colors you need.

Listing 8 shows the code for the `Form.Paint` event. I set the `Form.Color` to black, then paint one large red rectangle that takes up most of the top half of the window. There is a smaller red rectangle to cover the descender of the "p". The small red squares between the letters of "MAGAZINE" are painted with a series of red rectangles. By the way, the outline for the squares is done with the Arial font. The square is ANSI character #0183. Make sure `NumLock` is turned on, then hold down the `Alt` key while typing `0183` using the number pad. Release the `Alt` key and the character will appear.

Wrap Up

The complete source code for everything discussed here is assembled in demo program REGIONS.EXE and of course is on the floppy disk included with this issue. The program was completed with Delphi 2.0 and should work in Delphi 3. The code is pretty well commented, so you shouldn't have any problem following it. Since our main purpose behind creating an oddly shaped form was to us it as a splash screen, I used the last example as the splash screen for the demo program. The code in the project file and in the main form file shows how to use a window as a splash screen.

At the start of the article I mentioned that we could create a round radar screen or resource gauge that the user can drag to any position on the screen. If the window doesn't have a title bar, like our blue oval, how can the user

➤ *Figure 5*

➤ *Figure 6*

```
procedure TfrmText.FormCreate;
var
  RgnPts: array[0..11] of TPoint;
  Region1: hRgn;
begin
  BeginPath(Canvas.Handle);
  SetBkMode(Canvas.Handle, TRANSPARENT); { allows us to outline letters }
  with canvas do
  begin
    { Use canvas to paint 'Delphi' }
    Font.Style := [fsBold];
    Font.Name := 'Times New Roman';
    Font.Size := 130;
    TextOut(62, 0, 'Delphi');
    . . .  { More drawing here } . . .
  end;{ canavs drawing }
  { Create first underline }
  MoveToEx(Canvas.Handle, 0, 165, nil);
  LineTo(Canvas.Handle, 311, 165);
  LineTo(Canvas.Handle, 311, 175);
  LineTo(Canvas.Handle, 0, 175);
  LineTo(Canvas.Handle, 0, 165);
  EndPath(Canvas.Handle);
  { Create region, boundaries from the Path }
  Region1 := PathToRegion(Canvas.Handle);
  { Assign the region to the window }
  SetWindowRgn(Handle, Region1, True);
end;
```

➤ *Listing 7*

drag it? The code that allows the user to drag the window is in Listing 9. The procedure `WMNCHitTest` captures the mouse button down event and if the mouse button is the left one, reports to windows that the user clicked on the title bar. Windows then allows the window to be dragged.

That wraps up all the loose ends. Information can be found in the Win32 Programmer's Reference (WIN32.HLP) included with Delphi 2.0 on any of the Windows API functions discussed in this article.

Steven J. Colagiovanni is a Photographic Technician with a major photographic manufacturer and is currently living in Los Angeles, California. He is a member of the Los Angeles Delphi User Group and programs in Delphi as a hobby. He can be reached at Steve_Cola@compuserve.com

```
procedure TfrmText.FormPaint(Sender: TObject);
begin
  with canvas do begin
    { Paint red areas of form }
    Pen.Color := clRed;
    Brush.Color := clRed;
    Brush.Style := bsSolid;
    Rectangle(60, 0, width, 160);// Delphi
    Rectangle(312, 155,  365, 195);// P extension
    Rectangle( 48, 195,  76, 260);// Dot after M
    Rectangle(126, 195, 154, 260);// Dot after A
    Rectangle(205, 195, 233, 260);// Dot after G
    Rectangle(283, 195, 309, 260);// Dot after A
    Rectangle(355, 195, 382, 260);// Dot after Z
    Rectangle(410, 195, 438, 260);// Dot after I
    Rectangle(486, 195, 516, 260);// Dot after N
  end;
end;
```

➤ *Listing 8*

```
unit bowtie;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls;
type
  TfrmBowTie = class(TForm)
  private
    procedure WMNCHitTest(Var Msg: TMessage);
    message WM_NCHITTEST;
  public
  end;
var frmBowTie: TfrmBowTie;
implementation
{$R *.DFM}
procedure TfrmBowTie.WMNCHitTest(Var Msg: TMessage);
begin
  { Respond to left mouse button down, so we can drag window }
  if GetAsyncKeyState(VK_LButton) < 0 then
    Msg.Result := HTCaption
  else
    Msg.Result := HTClient;
end;
end.
```

➤ *Listing 9*