CHAPTER

8

Graphics Programming with GDI and Fonts

IN THIS CHAPTER

In previous chapters, you worked with a property called `Canvas`. `Canvas` is appropriately named because you can think of a window as an artist's blank canvas on which various Windows objects are painted. Each button, window, cursor, and so on is nothing more than a collection of pixels in which the colors have been set to give it some useful appearance. In fact, think of each individual window as a separate surface on which its separate components are painted. To take this analogy a bit further, imagine that you're an artist who requires various tools to accomplish your task. You need a palette from which to choose different colors. You'll probably use different styles of brushes, drawing tools, and special artist's techniques as well. Win32 makes use of similar tools and techniques—in the programming sense—to paint the various objects with which users interact. These tools are made available through the Graphics Device Interface, otherwise known as the *GDI*.

Win32 uses the GDI to paint or draw the images you see on your computer screen. Before Delphi, in traditional Windows programming, programmers worked directly with the GDI functions and tools. Now, the `TCanvas` object encapsulates and simplifies the use of these functions, tools, and techniques. This chapter teaches you how to use `TCanvas` to perform useful graphics functions. You'll also see how you can create advanced programming projects with Delphi 5 and Win32 GDI. We illustrate this by creating a paint program and animation program.

Delphi's Representation of Pictures: TImage

The `TImage` component represents a graphical image that can be displayed anywhere on a form and is available from Delphi 5's Component Palette. With `TImage`, you can load and display a bitmap file (`.bmp`), a 16-bit Windows metafile (`.wmf`), a 32-bit enhanced metafile (`.emf`), an icon file (`.ico`), a JPEG file (`.jpg`, `.jpeg`) or other file formats handled by add-in `TGraphic` classes. The image data actually is stored by `TImage`'s `Picture` property, which is of the type `TPicture`.

Graphic Images: Bitmaps, Metafiles, and Icons

Bitmaps

Win32 *bitmaps* are binary information arranged in a pattern of bits that represent a graphical image. More specifically, these bits store color information items called *pixels*. There are two types of bitmaps: device-dependent bitmaps (DDB) and device-independent bitmaps (DIB).

As a Win32 programmer, you probably won't be dealing much with DDBs because this format was kept solely for backward compatibility. Device-dependent bitmaps, as the name implies, are dependent on the device in which they're created. Bitmaps in this format, when saved, do not store information regarding the color palette they use nor do they store information regarding their resolution.

In contrast, device-independent bitmaps (DIBs) do store information to allow them to be displayed on any device without radically changing their appearance.

In memory, both DDBs and DIBs are represented with the same structures, for the most part. One key difference is that DDBs use the palette provided by the system, whereas DIBs provide their own palette. To take this explanation further, DDBs are simply native storage, handled by video driver routines and video hardware. DIBs are standardized pixel formats, handled by GDI generic routines and stored in global memory. Some video cards use DIB pixel formats as native storage, so you get DDB=DIB. In general, DIB gives you more flexibility and simplicity, sometimes at a slight performance cost. DDBs are always faster but not as convenient.

Metafiles

Unlike bitmaps, *metafiles* are vector-based graphical images. Metafiles are files in which a series of GDI routines are stored, enabling you to save GDI function calls to disk so that you can redisplay the image later. This also enables you to share your drawing routines with other programs without having to call the specific GDI functions in each program. Other advantages to metafiles are that they can be scaled to arbitrary dimensions and still retain their smooth lines and arcs—bitmaps don't do this as well. In fact, this is one of the reasons the Win32 printing engine is built around the enhanced metafile storage for print jobs.

There are two metafile formats: standard metafiles, typically stored in a file with a `.wmf` extension, and enhanced metafiles, typically stored in a file with an `.emf` extension.

Standard metafiles are a holdover from the Win16 system. Enhanced metafiles are more robust and accurate. Use EMFs if you're producing metafiles for your own applications. If you're exporting your metafiles to older programs that might not be able to use the enhanced format, use the 16-bit WMFs. Know, however, that by stepping down to the 16-bit WMFs, you'll also lose several GDI primitives that EMFs support but WMFs do not. Delphi 5's `TMetafile` class knows about both types of metafiles.

Icons

Icons are Win32 resources that usually are stored in an icon file with an `.ico` extension. They may also reside in a resource file (`.res`). There are two typical sizes of icons in Windows: large icons that are 32×32 pixels, and small icons that are 16×16 pixels. All Windows applications use both icon sizes. Small icons are displayed in the application's upper-left corner of the main window and also in the Windows List view control. Delphi's encapsulation of this control is the `TListView` component. This control appears on the Win32 page of the Component Palette.

Icons are made up of two bitmaps. One bitmap, referred to as the *image*, is the actual icon image as it is displayed. The other bitmap, referred to as the *mask*, makes it possible to achieve transparency when the icon is displayed. Icons are used for a variety of purposes. For example, icons appear on an application's taskbar and in message boxes where the question mark, exclamation point, or stop sign icons are used as attention grabbers.

`TPicture` is a container class for the `TGraphic` abstract class. A *container class* means that `TPicture` can hold a reference to and display a `TBitmap`, `TMetafile`, `TIcon`, or any other `TGraphic` type, without really caring which is which. You use `TImage.Picture`'s properties and methods to load image files into a `TImage` component. For example, use the following statement:

```
MyImage.Picture.LoadFromFile('FileName.bmp');
```

Use a similar statement to load icon files or metafiles. For example, the following code loads a Win32 metafile:

```
MyImage.Picture.LoadFromFile('FileName.emf');
```

This code loads a Win32 icon file:

```
MyImage.Picture.LoadFromFile('FileName.ico');
```

In Delphi 5, `TPicture` can now load JPEG images using the same technique for loading bitmaps:

```
MyImage.Picture.LoadFromFile('FileName.jpeg');
```

Saving Images

To save an image use the `SaveToFile()` method:

```
MyImage.Picture.SaveToFile('FileName.bmp');
```

The `TBitmap` class encapsulates the Win32 bitmap and palette, and it provides the methods to load, store, display, save, and copy the bitmapped images. `TBitmap` also manages palette realization automatically. This means that the tedious task of managing bitmaps has been simplified substantially with Delphi 5's `TBitmap` class, which enables you to focus on using the bitmap and frees you from having to worry about all the underlying implementation details.

NOTE

`TBitmap` isn't the only object that manages palette realization. Components such as `TImage`, `TMetafile`, and every other `TGraphic` descendant also realize their bitmaps' palettes on request. If you build components that contain a `TBitmap` object that might have 256-color images, you'll need to override your component's `GetPalette()` method to return the color palette of the bitmap.

To create an instance of a `TBitmap` class and load a bitmap file, for example, you use the following commands:

```
MyBitmap := TBitmap.Create;
MyBitmap.LoadFromFile('MyBMP.BMP');
```

CAUTION

Another method of loading bitmaps into an application is to load them from a resource file. We'll discuss this method shortly.

To copy one bitmap to another, you use the `TBitmap.Assign()` method, as in this example:

```
Bitmap1.Assign(Bitmap2);
```

You also can copy a portion of a bitmap from one `TBitmap` instance to another `TBitmap` instance or even to the form's canvas by using the `CopyRect()` method:

```
var
  R1: TRect;
begin
  with R1 do
  begin
    Top := 0;
    Left := 0;
    Right := BitMap2.Width div 2;
    Bottom := BitMap2.Height div 2;
  end;
  Bitmap1.Canvas.CopyRect(ClientRect, BitMap2.Canvas, R1);
end;
```

In the preceding code, you first calculate the appropriate values in a `TRect` record and then use the `TCanvas.CopyRect()` method to copy a portion of the bitmap. A `TRect` is defined as follows:

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

This technique will be used in the paint program later in the chapter. `CopyRect()` automatically stretches the copied portion of the source canvas to fill the destination rectangle.

CAUTION

You should be aware of a significant difference in resource consumption for copying bitmaps in the previous two examples. The `CopyRect()` technique doubles the memory

continues

use in that two separate copies of the image exist in memory. The `Assign()` technique costs nothing because the two bitmap objects will share a reference to the same image in memory. If you happen to modify one of the bitmap objects, VCL will clone the image using a copy-on-write scheme.

Another method you can use to copy the entire bitmap to the form's canvas so that it shrinks or expands to fit inside the canvas's boundaries is the `StretchDraw()` method. Here's an example:

```
Canvas.StretchDraw(R1, MyBitmap);
```

We'll discuss `TCanvas`'s methods later in this chapter.

Using the TCanvas Properties

Higher-level classes such as `TForm` and `TGraphicControl` descendants have a `Canvas` property. The canvas serves as the painting surface for your form's other components. The tools that `Canvas` uses to do the drawing are pens, brushes, and fonts.

Using Pens

In this section, we first explain how to use the `TPen` properties and then show you some code in a sample project that uses these properties.

Pens enable you to draw lines on the canvas and are accessed from the `Canvas.Pen` property. You can change how lines are drawn by modifying the pen's properties: `Color`, `Width`, `Style`, and `Mode`.

The `Color` property specifies a pen's color. Delphi 5 provides predefined color constants that match many common colors. For example, the constants `clRed` and `clYellow` correspond to the colors red and yellow. Delphi 5 also defines constants to represent the Win32 system screen element colors such as `clActiveCaption` and `clHighlightText`, which correspond to the Win32 active captions and highlighted text. The following line assigns the color blue to the canvas's pen:

```
Canvas.Pen.color := clblue;
```

This line shows you how to assign a random color to `Canvas`'s `Pen` property:

```
Pen.Color := TColor(RGB(Random(255),Random(255), Random(255)));
```

RGB() and TColor

Win32 represents colors as long integers in which the lowest three bytes each signify a red, green, and blue intensity level. The combination of the three values makes up a valid Win32 color. The `RGB(`*R*`, `*G*`, `*B*`)` function takes three parameters for the red, green, and blue intensity levels. It returns a Win32 color as a long integer value. This is represented as a `TColor` Delphi type. There are 255 possible values for each intensity level and approximately 16 million colors that can be returned from the `RGB()` function. `RGB(0, 0, 0)`, for example, returns the color value for black, whereas `RGB(255, 255, 255)` returns the color value for white. `RGB(255, 0 ,0)`, `RGB(0, 255, 0)`, and `RGB(0, 0, 255)` return the color values for red, green, and blue, respectively. By varying the values passed to `RGB()`, you can obtain a color anywhere within the color spectrum.

`TColor` is specific to VCL and refers to constants defined in the `Graphics.pas` unit. These constants map to either the closest matching color in the system palette or to a defined color in the Windows Control Panel. For example, `clBlue` maps to the color blue whereas the `clBtnFace` maps to the color specified for button faces. In addition to the three bytes to represent the color, `TColor`'s highest order byte specifies how a color is matched. Therefore, if the highest order byte is `$00`, the represented color is the closest matching color in the system palette. A value of `$01` represents the closest matching color in the currently realized palette. Finally, a color of `$02` matches with the nearest color in the logical palette of the current device context. You will find additional information in the Delphi help file under "`TColor type`."

TIP

Use the `ColorToRGB()` function to convert Win32 system colors, such as `clWindow`, to a valid RGB color. The function is described in Delphi 5's online help.

The pen can also draw lines with different drawing styles, as specified by its `Style` property. Table 8.1 shows the different styles you can set for `Pen.Style`.

TABLE 8.1 Pen Styles

Style	*Draws*
`psClear`	An invisible line
`psDash`	A line made up of a series of dashes
`psDashDot`	A line made up of alternating dashes and dots
`psDashDotDot`	A line made up of a series of dash-dot-dot combinations
`psDot`	A line made up of a series of dots
`psInsideFrame`	A line within a frame of closed shapes that specify a bounding rectangle
`psSolid`	A solid line

The following line shows how you would change the pen's drawing style:

```
Canvas.Pen.Style := psDashDot;
```

Figure 8.1 shows how the different pen styles appear when drawn on the form's canvas. One thing to note: The "in between" colors in the stippled lines come from the brush color. If you want to make a black dashed line run across a red square, you would need to set the `Canvas.Brush.Color` to `clRed` or set the `Canvas.Brush.Style` to `bsClear`. Setting both the pen and brush color is how you would draw, for example, a red and blue dashed line across a white square.

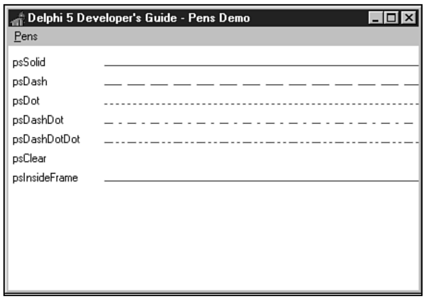


FIGURE 8.1
Different pen styles.

The `Pen.Width` property enables you to specify the width, in pixels, that the pen uses for drawing. When this property is set to a larger width, the pen draws with thicker lines.

NOTE

The stipple line style applies only to pens with a width of 1. Setting the pen width to 2 will draw a solid line. This is a holdover from the 16-bit GDI that Win32 emulates for compatibility. Windows 95/98 does not do fat stippled lines, but Windows NT/2000 can if you use only the extended GDI feature set.

Three factors determine how Win32 draws pixels or lines to a canvas surface: the pen's color, the surface or destination color, and the bitwise operation that Win32 performs on the two-color values. This operation is known as a *raster operation* (ROP). The `Pen.Mode` property specifies the ROP to be used for a given canvas. Sixteen modes are predefined in Win32, as shown in Table 8.2.

TABLE 8.2 Win32 Pen Modes on Source `Pen.Color` (S) and `Destination` (D) Color

Mode	*Result Pixel Color*	*Boolean Operation*
`pmBlack`	Always black	`0`
`pmWhite`	Always white	`1`
`pmNOP`	Unchanged	`D`
`pmNOT`	Inverse of `D` color	`not D`
`pmCopy`	Color specified by `S`	`S`
`pmNotCopy`	Inverse of `S`	`not S`
`pmMergePenNot`	Combination `S` and inverse of `D`	`S or not D`
`pmMaskPenNot`	Combination of colors common to `S` and inverse of `D`	`S and not D`
`pmMergeNotPen`	Combination of `D` and inverse of `S`	`not S or D`
`pmMaskNotPen`	Combination of colors common to `D` and inverse of `S`	`not S and D`
`pmMerge`	Combination of `S` and `D`	`S or D`
`pmNotMerge`	Inverse of `pmMerge` operation on `S` and `D`	`not (S or D)`
`pmMask`	Combination of colors common to `S` and `D`	`S and D`
`pmNotMask`	Inverse of `pmMask` operation on `S` and `D`	`not (S and D)`
`pmXor`	Combination of colors in either `S` or `D` but not both	`S XOR D`
`pmNotXor`	Inverse of `pmXOR` operation on `S` and `D`	`not (S XOR D)`

`Pen.mode` is `pmCopy` by default. This means that the pen draws with the color specified by its `Color` property. Suppose that you want to draw black lines on a white background. If a line crosses over a previously drawn line, it should draw white rather than black.

One way to do this would be to check the color of the area you're going to draw to—if it's white, set `pen.Color` to black; if it is black, set `pen.Color` to white. Although this approach works, it would be cumbersome and slow. A better approach would be to set `Pen.Color` to `clBlack` and `Pen.Mode` to `pmNot`. This would result in the pen drawing the inverse of the merging operation with the pen and surface color. Figure 8.2 shows you the result of this operation when drawing with a black pen in a crisscross fashion.

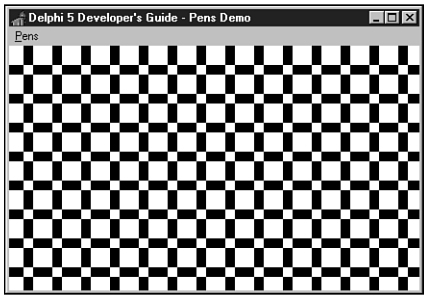


FIGURE 8.2

The output of a `pmNotMerge` *operation.*

Listing 8.1 is an example of the project on the CD that illustrates the code that resulted in Figures 8.1 and 8.2. You'll find this demo on the CD.

LISTING 8.1 Illustration of Pen Operations

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
Dialogs, Menus, StdCtrls, Buttons, ExtCtrls;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiPens: TMenuItem;
    mmiStyles: TMenuItem;
    mmiPenColors: TMenuItem;
    mmiPenMode: TMenuItem;
    procedure mmiStylesClick(Sender: TObject);
    procedure mmiPenColorsClick(Sender: TObject);
    procedure mmiPenModeClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure ClearCanvas;
    procedure SetPenDefaults;
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
 // Clear the contents of the canvas
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    Canvas.FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetPenDefaults;
begin
  with Canvas.Pen do
  begin
    Width := 1;
    Mode := pmCopy;
    Style := psSolid;
    Color := clBlack;
  end;
end;

procedure TMainForm.mmiStylesClick(Sender: TObject);
var
  yPos: integer;
  PenStyle: TPenStyle;
begin
  ClearCanvas;     // First clear Canvas's contents
  SetPenDefaults;
  // yPos represent the Y coordinate
  YPos := 20;
  with Canvas do
  begin
    for PenStyle := psSolid to psInsideFrame do
    begin
      Pen.Color := clBlue;
      Pen.Style := PenStyle;
      MoveTo(100, yPos);
```

continues

LISTING 8.1 Continued

```
      LineTo(ClientWidth, yPos);
      inc(yPos, 20);
    end;

    // Write out titles for the various pen styles
    TextOut(1, 10, ' psSolid ');
    TextOut(1, 30, ' psDash ');
    TextOut(1, 50, ' psDot ');
    TextOut(1, 70, ' psDashDot ');
    TextOut(1, 90, ' psDashDotDot ');
    TextOut(1, 110, ' psClear ');
    TextOut(1, 130, ' psInsideFrame ');
  end;
end;

procedure TMainForm.mmiPenColorsClick(Sender: TObject);
var
  i: integer;
begin
  ClearCanvas;    // Clear Canvas's contents
  SetPenDefaults;
  with Canvas do
  begin
    for i := 1 to 100 do
    begin
      // Get a random pen color draw a line using that color
      Pen.Color := RGB(Random(256),Random(256), Random(256));
      MoveTo(random(ClientWidth), Random(ClientHeight));
      LineTo(random(ClientWidth), Random(ClientHeight));
    end
  end;
end;

procedure TMainForm.mmiPenModeClick(Sender: TObject);
var
  x,y: integer;
begin
  ClearCanvas;  // Clear the Canvas's contents
  SetPenDefaults;
  y := 10;
  canvas.Pen.Width := 20;
  while y < ClientHeight do
  begin
    canvas.MoveTo(0, y);
    // Draw a line and increment Y value
    canvas.LineTo(ClientWidth, y);
    inc(y, 30);
  end;
  x := 5;

  canvas.pen.Mode := pmNot;
  while x < ClientWidth do
  begin
    Canvas.MoveTo(x, 0);
    canvas.LineTo(x, ClientHeight);
    inc(x, 30);
  end;
end;

end.
```

Listing 8.1 shows three examples of dealing with the canvas's pen. The two helper functions, `ClearCanvas()` and `SetPenDefaults()`, are used to clear the contents of the main form's canvas and to reset the `Canvas.Pen` properties to their default values as these properties are modified by each of the three event handlers.

`ClearCanvas()` is a useful technique for erasing the contents of any component containing a `Canvas` property. `ClearCanvas()` uses a solid white brush to erase whatever was previously painted on the `Canvas`. `FillRect()` is responsible for painting a rectangular area as specified by a `TRect` structure, `ClientRect`, which is passed to it.

The `mmiStylesClick()` method shows how to display the various `TPen` styles as shown in Figure 8.1 by drawing horizontal lines across the form's `Canvas` using a different `TPen` style. Both `TCanvas.MoveTo()` and `TCanvas.LineTo()` enable you to draw lines on the canvas.

The `mmiPenColorsClick()` method illustrates drawing lines using a different `TPen` color. Here you use the `RGB()` function to retrieve a color to assign to `TPen.Color`. The three values you pass to `RGB()` are each random values within range of `0` to `255`. The output of this method is shown in Figure 8.3.

Finally, the `mmiPenModeClick()` method illustrates how to draw lines using a different pen mode. Here you use the `pmNot` mode to perform the actions previously discussed, resulting in the output shown in Figure 8.2.

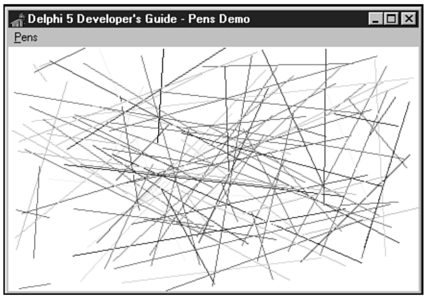


FIGURE 8.3

Output from the `mmiPenColorsClick()` *method.*

Using TCanvas's Pixels

The `TCanvas.Pixels` property is a two-dimensional array in which each element represents a pixel's `TColor` value on the form's surface or client area. The upper-left corner of your form's painting surface is given by

```
Canvas.Pixels[0,0]
```

and the lower-right corner is

```
Canvas.Pixels[clientwidth, clientheight];
```

It's rare that you'll ever have to access individual pixels on your form. In general, you do not want to use the `Pixels` property because it's slow. Accessing this property uses the `GetPixel()`/`SetPixel()` GDI functions, which Microsoft has acknowledged are flawed and will never be efficient. This is because both functions rely on 24-bit RGB values. When not working with 24-bit RGB device contexts, these functions must perform serious color-matching gymnastics to convert the RGB into a device pixel format. For quick pixel manipulation, use the `TBitmap.ScanLine` array property instead. To fetch or set one or two pixels at a time, using `Pixels` is okay.

Using Brushes

This section discusses the `TBrush` properties and shows you some code in a sample project that uses these properties.

Using the TBrush Properties

A canvas's brush fills in areas and shapes drawn on the canvas. This differs from a `TPen` object, which enables you to draw lines to the canvas. A brush enables you to fill an area on the canvas using various colors, styles, and patterns.

`Canvas`'s `TBrush` object has three important properties that specify how the brush paints on the canvas's surface: `Color`, `Style`, and `Bitmap`. `Color` specifies the brush's color, `Style` specifies the pattern of the brush background, and Bitmap specifies a bitmap you can use to create custom patterns for the brush's background.

Eight brush options are specified by the Style property: bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, and bsDiagCross. By default, the brush color is clWhite with a bsSolid style and no bitmap. You can change the color and style to fill an area with different patterns. The example in the following section illustrates using each of the TBrush properties.

TBrush Code Example

Listing 8.2 shows you the unit for a project that illustrates the use of the TBrush properties just discussed. You can load this project from the CD.

LISTING 8.2 TBrush Example

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
Dialogs, Menus, ExtCtrls;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiBrushes: TMenuItem;
    mmiPatterns: TMenuItem;
    mmiBitmapPattern1: TMenuItem;
    mmiBitmapPattern2: TMenuItem;
    procedure mmiPatternsClick(Sender: TObject);
    procedure mmiBitmapPattern1Click(Sender: TObject);
    procedure mmiBitmapPattern2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation
```

continues

LISTING 8.2 Continued

```
{$R *.DFM}

procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
 // Clear the contents of the canvas
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetWindowRect(Handle, R);
    R.TopLeft := ScreenToClient(R.TopLeft);
    R.BottomRight := ScreenToClient(R.BottomRight);
    FillRect(R);
  end;
end;

procedure TMainForm.mmiPatternsClick(Sender: TObject);
begin
  ClearCanvas;
  with Canvas do
  begin
    // Write out titles for the various brush styles
    TextOut(120, 101, 'bsSolid');
    TextOut(10, 101, 'bsClear');
    TextOut(240, 101, 'bsCross');
    TextOut(10, 221, 'bsBDiagonal');
    TextOut(120, 221, 'bsFDiagonal');
    TextOut(240, 221, 'bsDiagCross');
    TextOut(10, 341, 'bsHorizontal');
    TextOut(120, 341, 'bsVertical');

    // Draw a rectangle with the various brush styles

    Brush.Style := bsClear;
    Rectangle(10, 10, 100, 100);
    Brush.Color := clBlack;

    Brush.Style := bsSolid;
    Rectangle(120, 10, 220, 100);

    { Demonstrate that the brush is transparent by drawing
      colored rectangle, over which the brush style rectangle will
      be drawn. }
    Brush.Style := bsSolid;
    Brush.Color := clRed;
    Rectangle(230, 0, 330, 90);

    Brush.Style := bsCross;
    Brush.Color := clBlack;
    Rectangle(240, 10, 340, 100);

    Brush.Style := bsBDiagonal;
    Rectangle(10, 120, 100, 220);

    Brush.Style := bsFDiagonal;
    Rectangle(120, 120, 220, 220);

    Brush.Style := bsDiagCross;
    Rectangle(240, 120, 340, 220);

    Brush.Style := bsHorizontal;
    Rectangle(10, 240, 100, 340);

    Brush.Style := bsVertical;
    Rectangle(120, 240, 220, 340);

  end;
end;

procedure TMainForm.mmiBitmapPattern1Click(Sender: TObject);
begin
  ClearCanvas;
  // Load a bitmap from the disk
  FBitMap.LoadFromFile('pattern.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Draw a rectangle to cover the form's entire
      client area using the bitmap pattern as the
      brush with which to paint. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.mmiBitmapPattern2Click(Sender: TObject);
begin
  ClearCanvas;
  // Load a bitmap from the disk
```

continues

LISTING 8.2 Continued

```
  FBitMap.LoadFromFile('pattern2.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Draw a rectangle to cover the form's entire
      client area using the bitmap pattern as the
      brush with which to paint. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FBitmap.Free;
end;

end.
```

TIP

The `ClearCanvas()` method that you use here is a handy routine for a utility unit. You can define `ClearCanvas()` to take `TCanvas` and `TRect` parameters to which the erase code will be applied:

```
procedure ClearCanvas(ACanvas: TCanvas; ARect: TRect);
begin
 // Clear the contents of the canvas
  with ACanvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ARect);
  end;
end;
```

The `mmiPatternsClick()` method illustrates drawing with various `TBrush` patterns. First, you draw out the titles and then, using each of the available brush patterns, draw rectangles on the form's canvas. Figure 8.4 shows the output of this method.

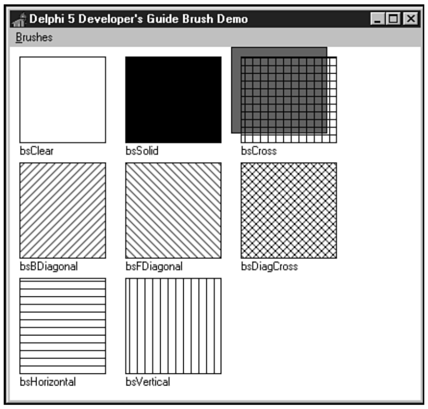


FIGURE 8.4
Brush patterns.

The `mmiBitmapPattern1Click()` and `mmiBitmapPattern2Click()` methods illustrate how to use a bitmap pattern as a brush. The `TCanvas.Brush` property contains a `TBitmap` property to which you can assign a bitmap pattern. This pattern will be used to fill the area painted by the brush instead of the pattern specified by the `TBrush.Style` property. There are a few rules for using this technique, however. First, you must assign a valid bitmap object to the property. Second, you must assign `nil` to the `Brush.Bitmap` property when you're finished with it because the brush does not take ownership of the bitmap object when you assign a bitmap to it. Figures 8.5 and 8.6 show the output of `mmiBitmapPattern1Click()` and `mmiBitmapPattern2Click()`, respectively.

NOTE

Windows limits the size of pattern brush bitmaps to 8×8 pixels, and they must be device-dependent bitmaps, not device-independent bitmaps. Windows will reject brush pattern bitmaps larger than 8×8; NT will accept larger but will only use the top-left 8×8 portion.

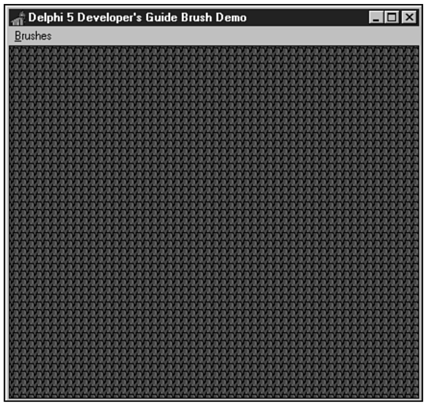


FIGURE 8.5
Output from `mmiBitmapPattern1Click()`.

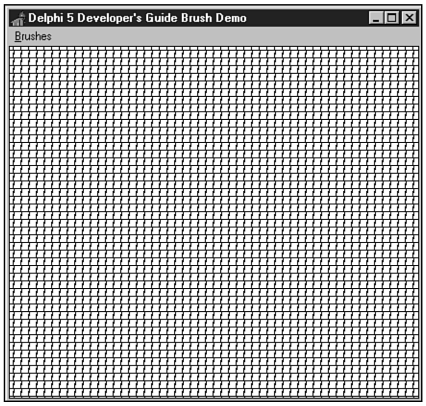


FIGURE 8.6
Output from `mmiBitmapPattern2Click()`.

TIP

Using a bitmap pattern to fill an area on the canvas doesn't only apply to the form's canvas but also to any component that contains a `Canvas` property. Just access the methods and/or properties of the `Canvas` property of the component rather than the form. For example, here's how to perform pattern drawing on a `TImage` component:

```
Image1.Canvas.Brush.Bitmap := SomeBitmap;
try
```

```
    Image1.Canvas.Rectangle(0, 0, Image1.Width, Image1.Height);
  finally
     Image1.Canvas.Brush.Bitmap := nil;
  end;
```

Using Fonts

The `Canvas.Font` property enables you to draw text using any of the available Win32 fonts. You can change the appearance of text written to the canvas by modifying the font's `Color`, `Name`, `Size`, `Height`, or `Style` property.

You can assign any of Delphi 5's predefined colors to `Font.Color`. The following code, for example, assigns the color red to the canvas's font:

```
Canvas.Font.Color := clRed;
```

The `Name` property specifies the Window's font name. For example, the following two lines of code assign different typefaces to `Canvas`'s font:

```
Canvas.Font.Name := 'New Times Roman';
```

`Canvas.Font.Size` specifies the font's size in points.

`Canvas.Font.Style` is a set composed of one style or a combination of the styles shown in Table 8.3.

TABLE 8.3 Font Styles

Value	*Style*
`fsBold`	Boldface
`fsItalic`	Italic
`fsUnderline`	Underlined
`fsStrikeOut`	A horizontal line through the font, giving it a strikethrough appearance

To combine two styles, use the syntax for combining multiple set values:

```
Canvas.Font.Style := [fsBold, fsItalic];
```

You can use `TFontDialog` to obtain a Win32 font and assign that font to the `TMemo.Font` property:

```
if FontDialog1.Execute then
    Memo1.Font.Assign(FontDialog1.Font);
```

The same can be done to assign the font selected in `TFontDialog` to the `Canvas`'s font:

```
Canvas.Font.Assign(FontDialog1.Font);
```

CAUTION

Make sure to use the `Assign()` method when copying `TBitMap`, `TBrush`, `TIcon`, `TMetaFile`, `TPen`, and `TPicture` instance variables. A statement such as

```
MyBrush1 := MyBrush2
```

might seem valid, but it performs a direct pointer copy so that both instances point to the same brush object, which can result in a heap leak. By using `Assign()`, you ensure that previous resources are freed.

This is not so when assigning between two `TFont` properties. Therefore, a statement such as

```
Form1.Font := Form2.Font
```

is a valid statement because the `TForm.Font` is a property whose write method internally calls `Assign()` to copy the data from the given font object. Be careful, however; this is only valid with when assigning `TFont` properties, not `TFont` variables. As a general rule, always use `Assign()`.

Additionally, you can assign individual attributes from the selected font in `TFontDialog` to the `Canvas`'s font:

```
Canvas.Font.Name := Font.Dialog1.Font.Name;
Canvas.Font.Size := Font.Dialog1.Font.Size;
```

We've quickly brushed over fonts here. A more thorough discussion on fonts appears at the end of this chapter.

Using the CopyMode Property

The `TCanvas.CopyMode` property determines how a canvas copies an image from another canvas onto itself. For example, when `CopyMode` holds the value `cmSrcCopy`, this means that the source image will be copied over the destination entirely. `CmSrcInvert`, however, causes the pixels of both the source and destination images to be combined using the bitwise `XOR` operator. `CopyMode` is used for achieving different effects when copying from one bitmap to another bitmap. A typical place where you would change the default value of `CopyMode` from `cmSrcCopy` to another value is when writing animation applications. You learn how to write animation later in this chapter.

To see how to use the `CopyMode` property, take a look at Figure 8.7.

Figure 8.7 shows a form that contains two images, both of which have an ellipse drawn on them. You select a `CopyMode` setting from the `TComboBox` component, and you would get various results when copying the one image over the other by clicking the Copy button. Figures 8.8 and 8.9 show what the effects would be by copying `imgFromImage` to `imgToImage` using the `cmSrcAnd` and `cmSrcInvert` copy modes.

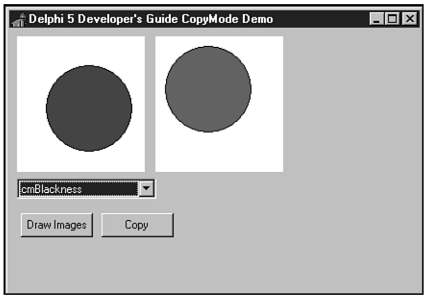


FIGURE 8.7
A form that contains two images to illustrate `CopyMode`*.*

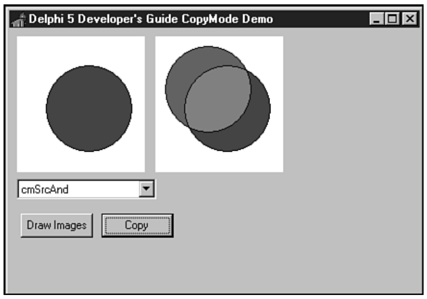


FIGURE 8.8
A copy operation using the `cmSrcAnd` *copy mode setting.*

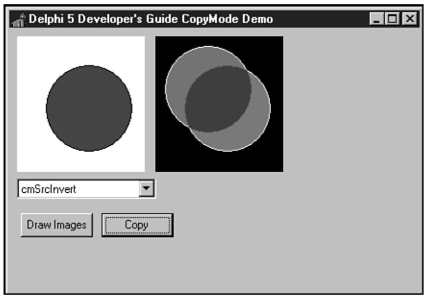


FIGURE 8.9
A copy operation using the `cmSrcInvert` *copy mode setting.*

Listing 8.3 shows the source code for the project, illustrating the various copy modes. You'll find this code on the CD.

LISTING 8.3 Project Illustrating CopyMode Usage

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, ExtCtrls;

type

  TMainForm = class(TForm)
    imgCopyTo: TImage;
    imgCopyFrom: TImage;
    cbCopyMode: TComboBox;
    btnDrawImages: TButton;
    btnCopy: TButton;
    procedure FormShow(Sender: TObject);
    procedure btnCopyClick(Sender: TObject);
    procedure btnDrawImagesClick(Sender: TObject);
  private
    procedure DrawImages;
    procedure GetCanvasRect(AImage: TImage; var ARect: TRect);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.GetCanvasRect(AImage: TImage; var ARect: TRect);
var
  R: TRect;
  R2: TRect;
begin
  R := AImage.Canvas.ClipRect;
  with AImage do begin
    ARect.TopLeft     := Point(0, 0);
    ARect.BottomRight := Point(Width, Height);
  end;
  R2 := ARect;
  ARect := R2;
end;
```

```
procedure TMainForm.DrawImages;
var
  R: TRect;
begin
  // Draw an ellipse in img1
  with imgCopyTo.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyTo, R);
    FillRect(R);
    Brush.Color := clRed;
    Ellipse(10, 10, 100, 100);
  end;

  // Draw an ellipse in img2
  with imgCopyFrom.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyFrom, R);
    FillRect(R);
    Brush.Color := clBlue;
    Ellipse(30, 30, 120, 120);
  end;

end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Initialize the combobox to the first item
  cbCopyMode.ItemIndex := 0;
  DrawImages;
end;

procedure TMainForm.btnCopyClick(Sender: TObject);
var
  cm: Longint;
  CopyToRect,
  CopyFromRect: TRect;
begin
  // Determine the copy mode based on the combo box selection
  case cbCopyMode.ItemIndex of
   0:  cm := cmBlackNess;
   1:  cm := cmDstInvert;
   2:  cm := cmMergeCopy;
```

continues

LISTING 8.3 Continued

```
   3:  cm := cmMergePaint;
   4:  cm := cmNotSrcCopy;
   5:  cm := cmNotSrcErase;
   6:  cm := cmPatCopy;
   7:  cm := cmPatInvert;
   8:  cm := cmPatPaint;
   9:  cm := cmSrcAnd;
   10: cm := cmSrcCopy;
   11: cm := cmSrcErase;
   12: cm := cmSrcInvert;
   13: cm := cmSrcPaint;
   14: cm := cmWhiteness;
   else
     cm := cmSrcCopy;
  end;

  // Assign the selected copymode to Image1's CopyMode property.
  imgCopyTo.Canvas.CopyMode := cm;


  GetCanvasRect(imgCopyTo, CopyToRect);
  GetCanvasRect(imgCopyFrom, CopyFromRect);

  // Now copy Image2 onto Image1 using Image1's CopyMode setting
  imgCopyTo.Canvas.CopyRect(CopyToRect, imgCopyFrom.Canvas, CopyFromRect);
end;

procedure TMainForm.btnDrawImagesClick(Sender: TObject);
begin
  DrawImages;
end;

end.
```

This project initially paints an ellipse on the two `TImage` components: `imgFromImage` and `imgToImage`. When the Copy button is clicked, `imgFromImage` is copied onto `imgToImage1` using the `CopyMode` setting specified from `cbCopyMode`.

Other Properties

`TCanvas` has other properties that we'll discuss more as we illustrate how to use them in coding techniques. This section briefly discusses these properties.

`TCanvas.ClipRect` represents a drawing region of the canvas to which drawing can be performed. You can use `ClipRect` to limit the area that can be drawn for a given canvas.

CAUTION

Initially, `ClipRect` represents the entire `Canvas` drawing area. You might be tempted to use the `ClipRect` property to obtain the bounds of a canvas. However, this could get you into trouble. `ClipRect` will not always represent the total size of its component. It can be less than the `Canvas`'s display area.

`TCanvas.Handle` gives you access to the actual device context that the `TCanvas` instance encapsulates. Device contexts are discussed later in this chapter.

`TCanvas.PenPos` is simply an X,Y coordinate location of the canvas's pen. You can change the pen's position by using the `TCanvas` methods—`MoveTo()`, `LineTo()`, `PolyLine()`, `TextOut()`, and so on.

Using the TCanvas Methods

The `TCanvas` class encapsulates many GDI drawing functions. With `TCanvas`'s methods, you can draw lines and shapes, write text, copy areas from one canvas to another, and even stretch an area on the canvas to fill a larger area.

Drawing Lines with TCanvas

`TCanvas.MoveTo()` changes `Canvas.Pen`'s drawing position on the `Canvas`'s surface. The following code, for example, moves the drawing position to the upper-left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

`TCanvas.LineTo()` draws a line on the canvas from its current position to the position specified by the parameters passed to `LineTo()`. Use `MoveTo()` with `LineTo()` to draw lines anywhere on the canvas. The following code draws a line from the upper-left position of the form's client area to the form's lower-right corner:

```
Canvas.MoveTo(0, 0);
Canvas.LineTo(ClientWidth, ClientHeight);
```

You already saw how to use the `MoveTo()` and `LineTo()` methods in the section covering the `TCanvas.Pen` property.

NOTE

Delphi now supports right-to-left-oriented text and control layouts; some controls (such as the grid) change the canvas coordinate system to flip the X axis. Therefore, if you're running Delphi on a Middle East version of Windows, `MoveTo(0,0)` may go to the top-right corner of the window.

Drawing Shapes with TCanvas

`TCanvas` offers various methods for rendering shapes to the canvas: `Arc()`, `Chord()`, `Ellipse()`, `Pie()`, `Polygon()`, `PolyLine()`, `Rectangle()`, and `RoundRect()`. To draw an ellipse in the form's client area, you would use `Canvas`'s `Ellipse()` method, as shown in the following code:

```
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

You also can fill an area on the canvas with a brush pattern specified in the `Canvas.Brush.Style` property. The following code draws an ellipse and fills the ellipse interior with the brush pattern specified by `Canvas.Brush.Style`:

```
Canvas.Brush.Style := bsCross;
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

Additionally, you saw how to add a bitmap pattern to the `TCanvas.Brush.Bitmap` property, which it uses to fill an area on the canvas. You can also use a bitmap pattern for filling in shapes. We'll demonstrate this later.

Some of `Canvas`'s other shape-drawing methods take additional or different parameters to describe the shape being drawn. The `PolyLine()` method, for example, takes an array of `TPoint` records that specify positions, or pixel coordinates, on the canvas to be connected by a line—sort of like connect the dots. A `TPoint` is a record in Delphi 5 that signifies an X,Y coordinate. A `TPoint` is defined as

```
TPoint = record
  X: Integer;
  Y: Integer;
end;
```

A Code Example for Drawing Shapes

Listing 8.4 illustrates using the various shape-drawing methods of `TCanvas`. You can find this project on the CD.

LISTING 8.4 An Illustration of Shape-Drawing Operations

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Menus;

type
  TMainForm = class(TForm)
```

```
    mmMain: TMainMenu;
    mmiShapes: TMenuItem;
    mmiArc: TMenuItem;
    mmiChord: TMenuItem;
    mmiEllipse: TMenuItem;
    mmiPie: TMenuItem;
    mmiPolygon: TMenuItem;
    mmiPolyline: TMenuItem;
    mmiRectangle: TMenuItem;
    mmiRoundRect: TMenuItem;
    N1: TMenuItem;
    mmiFill: TMenuItem;
    mmiUseBitmapPattern: TMenuItem;
    mmiPolyBezier: TMenuItem;
    procedure mmiFillClick(Sender: TObject);
    procedure mmiArcClick(Sender: TObject);
    procedure mmiChordClick(Sender: TObject);
    procedure mmiEllipseClick(Sender: TObject);
    procedure mmiUseBitmapPatternClick(Sender: TObject);
    procedure mmiPieClick(Sender: TObject);
    procedure mmiPolygonClick(Sender: TObject);
    procedure mmiPolylineClick(Sender: TObject);
    procedure mmiRectangleClick(Sender: TObject);
    procedure mmiRoundRectClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiPolyBezierClick(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
    procedure SetFillPattern;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}


procedure TMainForm.ClearCanvas;
begin
 // Clear the contents of the canvas
  with Canvas do
```

continues

Listing 8.4 Continued

```
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetFillPattern;
begin
 { Determine if shape is to be draw with a bitmap pattern in which
   case load a bitmap. Otherwise, use the brush pattern. }
  if mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := FBitmap
  else
    with Canvas.Brush do
    begin
      Bitmap := nil;
      Color := clBlue;
      Style := bsCross;
    end;
end;

procedure TMainForm.mmiFillClick(Sender: TObject);
begin
  mmiFill.Checked := not mmiFill.Checked;
  { If mmiUseBitmapPattern was checked, uncheck it set the
    brush's bitmap to nil. }
  if mmiUseBitmapPattern.Checked then
  begin
    mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.mmiUseBitmapPatternClick(Sender: TObject);
begin
  { Set mmiFil1.Checked mmiUseBitmapPattern.Checked. This will cause
    the SetFillPattern procedure to be called. However, if
    mmiUseBitmapPattern is being set, set Canvas.Brush.Bitmap to
    nil. }
  mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
  mmiFill.Checked := mmiUseBitmapPattern.Checked;
  if not mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := nil;
end;
```

```
procedure TMainForm.mmiArcClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
    Canvas.Arc(Left, Top, Right, Bottom, Right, Top, Left, Top);
end;

procedure TMainForm.mmiChordClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
  begin
    if mmiFill.Checked then
      SetFillPattern;
    Canvas.Chord(Left, Top, Right, Bottom, Right, Top, Left, Top);
  end;
end;

procedure TMainForm.mmiEllipseClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
end;

procedure TMainForm.mmiPieClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Pie(0, 0, ClientWidth, ClientHeight, 50, 5, 300, 50);
end;

procedure TMainForm.mmiPolygonClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Polygon([Point(0, 0), Point(150, 20), Point(230, 130),
                  Point(40, 120)]);
end;
procedure TMainForm.mmiPolylineClick(Sender: TObject);
begin
  ClearCanvas;
```

continues

LISTING 8.4 Continued

```
  Canvas.PolyLine([Point(0, 0), Point(120, 30), Point(250, 120),
    Point(140, 200), Point(80, 100), Point(30, 30)]);
end;

procedure TMainForm.mmiRectangleClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Rectangle(10 , 10, 125, 240);
end;

procedure TMainForm.mmiRoundRectClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.RoundRect(15, 15, 150, 200, 50, 50);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
  FBitMap.LoadFromFile('Pattern.bmp');
  Canvas.Brush.Bitmap := nil;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FBitmap.Free;
end;

procedure TMainForm.mmiPolyBezierClick(Sender: TObject);
begin
  ClearCanvas;
  Canvas.PolyBezier([Point(0, 100), Point(100, 0), Point(200, 50),
    Point(300, 100)]);
end;

end.
```

The main menu event handlers perform the shape-drawing functions. Two public methods, `ClearCanvas`() and `SetFillPattern()`, serve as helper functions for the event handlers. The first eight menu items result in a shape-drawing function being drawn on the form's canvas. The last two items, "Fill" and "Use Bitmap Pattern," specify whether the shape is to be filled with a brush pattern or a bitmap pattern, respectively.

You should already be familiar with the `ClearCanvas()` functionality. The `SetFillPattern()` method determines whether to use a brush pattern or a bitmap pattern to fill the shapes drawn by the other methods. If a bitmap is selected, it's assigned to the `Canvas.Brush.Bitmap` property.

All the shape-drawing event handlers call `ClearCanvas()` to erase what was previously drawn on the canvas. They then call `SetFillPattern()` if the `mmiFill.Checked` property is set to `True`. Finally, the appropriate `TCanvas` drawing routine is called. The comments in the source discuss the purpose of each function. One method worth mentioning here is `mmiPolylineClick()`.

When you're drawing shapes, an enclosed boundary is necessary for filling an area with a brush or bitmap pattern. Although it's possible to use `PolyLine()` and `FloodFill()` to create an enclosed boundary filled with a pattern, this method is highly discouraged. `PolyLine()` is used specifically for drawing lines. If you want to draw filled polygons, call the `TCanvas.Polygon()` method. A one-pixel imperfection in where the `Polyline()` lines are drawn will allow the call to `FloodFill()` to leak out and fill the entire canvas. Drawing filled shapes with `Polygon()` uses math techniques that are immune to variations in pixel placement.

Painting Text with TCanvas

`TCanvas` encapsulates Win32 GDI routines for drawing text to a drawing surface. The following sections illustrate how to use these routines as well as how to use Win32 GDI functions that are not encapsulated by the `TCanvas` class.

Using the TCanvas Text-Drawing Routines

You used `Canvas`'s `TextOut()` function to draw text to the form's client area in previous chapters. `Canvas` has some other useful methods for determining the size, in pixels, of text contained in a string using `Canvas`'s rendered font. These functions are `TextWidth()` and `TextHeight()`. The following code determines the width and height for the string `"Delphi 5 -- Yes!"`:

```
var
  S: String;
  w, h: Integer;
begin
  S := 'Delphi 5 -- Yes!';
  w := Canvas.TextWidth(S);
  h := Canvas.TextHeight(S);
end.
```

The `TextRect()` method also writes text to the form but only within a rectangle specified by a `TRect` structure. The text not contained within the `TRect` boundaries is clipped. In the line

```
Canvas.TextRect(R,0,0,'Delphi 3.0 Yes!');
```

the string `"Delphi 5 Yes!"` is written to the canvas at location `0,0`. However, the portion of the string that falls outside the coordinates specified by `R`, a `TRect` structure, gets clipped.

Listing 8.5 illustrates using some of the text-drawing routines.

LISTING 8.5 A Unit That Illustrates Text-Drawing Operations

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus;

const
  DString = 'Delphi 5 YES!';
  DString2 = 'Delphi 5 Rocks!';

type

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiText: TMenuItem;
    mmiTextRect: TMenuItem;
    mmiTextSize: TMenuItem;
    mmiDrawTextCenter: TMenuItem;
    mmiDrawTextRight: TMenuItem;
    mmiDrawTextLeft: TMenuItem;
    procedure mmiTextRectClick(Sender: TObject);
    procedure mmiTextSizeClick(Sender: TObject);
    procedure mmiDrawTextCenterClick(Sender: TObject);
    procedure mmiDrawTextRightClick(Sender: TObject);
    procedure mmiDrawTextLeftClick(Sender: TObject);
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation
```

```
{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;

procedure TMainForm.mmiTextRectClick(Sender: TObject);
var
  R: TRect;
  TWidth, THeight: integer;
begin
  ClearCanvas;
  Canvas.Font.Size := 18;
  // Calculate the width/height of the text string
  TWidth := Canvas.TextWidth(DString);
  THeight := Canvas.TextHeight(DString);

  { Initialize a TRect structure. The height of this rectangle will
    be 1/2 the height of the text string height. This is to
    illustrate clipping the text by the rectangle drawn }
  R := Rect(1, THeight div 2, TWidth + 1, THeight+(THeight div 2));
  // Draw a rectangle based on the text sizes
  Canvas.Rectangle(R.Left-1, R.Top-1, R.Right+1, R.Bottom+1);
  // Draw the Text within the rectangle
  Canvas.TextRect(R,0,0,DString);
end;

procedure TMainForm.mmiTextSizeClick(Sender: TObject);
begin
  ClearCanvas;
  with Canvas do
  begin
    Font.Size := 18;
    TextOut(10, 10, DString);
    TextOut(50, 50, 'TextWidth = '+IntToStr(TextWidth(DString)));
    TextOut(100, 100, 'TextHeight = '+IntToStr(TextHeight(DString)));
  end;
end;

procedure TMainForm.mmiDrawTextCenterClick(Sender: TObject);
```

continues

LISTING 8.5 Continued

```
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels }
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text centered by specifying the dt_Center option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Center);
end;

procedure TMainForm.mmiDrawTextRightClick(Sender: TObject);
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text right-aligned by specifying the dt_Right option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Right);
end;

procedure TMainForm.mmiDrawTextLeftClick(Sender: TObject);
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text left-aligned by specifying the dt_Left option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Left);
end;

end.
```

Like the other projects, this project contains the `ClearCanvas()` method to erase the contents of the form's canvas.

The various methods of the main form are event handlers to the form's main menu.

The `mmiTextRectClick()` method illustrates how to use the `TCanvas.TextRect()` method. It determines the text width and height and draws the text inside a rectangle the height of the original text size. Its output is shown in Figure 8.10.

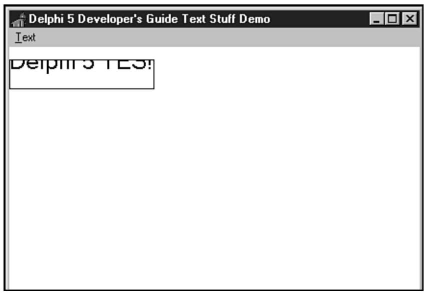
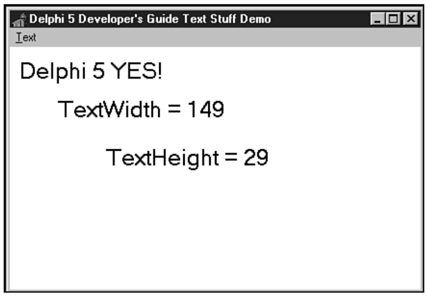


FIGURE 8.10
The output of `mmiTextRectClick()`.

`mmiTextSizeClick()` shows how to determine the size of a text string using the `TCanvas.TextWidth()` and `TCanvas.TextHeight()` methods. Its output is shown in Figure 8.11.



```
Delphi 5 YES!
     TextWidth = 149
           TextHeight = 29
```

FIGURE 8.11
The output of `mmiTextSizeClick()`.

Using Non-TCanvas GDI Text Output Routines

In the sample project, the `mmiDrawTextCenter()`, `mmiDrawTextRight()`, and `mmiDrawTextLeft()` methods all illustrate using the Win32 GDI function `DrawText()`. `DrawText()` is a GDI function not encapsulated by the `TCanvas` class.

This code illustrates how Delphi 5's encapsulation of Win32 GDI through the `TCanvas` class doesn't prevent you from making use of the abundant Win32 GDI functions. Instead, `TCanvas` really just simplifies using the more common routines while still enabling you to call any Win32 GDI function you might need.

If you look at the various GDI functions such as `BitBlt()` and `DrawText()`, you'll find that one of the required parameters is a DC, or *device context*. The device context is accessible through the canvas's `Handle` property. `TCanvas.Handle` is the DC for that canvas.

Device Contexts

Device contexts (DCs) are handles provided by Win32 to identify a Win32 application's connection to an output device such as a monitor, printer, or plotter through a device driver. In traditional Windows programming, you're responsible for requesting a DC whenever you need to paint to a window's surface; then, when done, you have to return the DC back to Windows. Delphi 5 simplifies the management of DCs by encapsulating DC management in the `TCanvas` class. In fact, `TCanvas` even caches your DC, saving it for later so that requests to Win32 occur less often—thus, speeding up your program's overall execution.

To see how to use `TCanvas` with a Win32 GDI function, you used the GDI routine `DrawText()` to output text with advanced formatting capabilities. `DrawText` takes the following five parameters:

Parameter	*Description*
`DC`	Device context of the drawing surface.
`Str`	Pointer to a buffer containing the text to be drawn. This must be a null-terminated string if the `Count` parameter is `-1`.
`Count`	Number of bytes in `Str`. If this value is `-1`, `Str` is a pointer to a null-terminated string.
`Rect`	Pointer to a `TRect` structure containing the coordinates of the rectangle in which the text is formatted.
`Format`	A bit field that contains flags specifying the various formatting options for `Str`.

In the example, you initialize a `TRect` structure using the `Rect()` function. You'll use the structure to draw a rectangle around the text drawn with the `DrawText()` function. Each of the three methods passes a different set of formatting flags to the `DrawText()` function. The `dt_WordBreak` and `dt_Center` formatting flags are passed to the `DrawText()` function to center the text in the rectangle specified by the `TRect` variable `R`. `dt_WordBreak`, `OR`'ed with `dt_Right`, is used to right-justify the text in the rectangle. Likewise, `dt_WordBreak`, `OR`'ed with `dt_Left`, left-justifies the text. The `dt_WordBreak` specifier word-wraps the text within the width given by the rectangle parameter and modifies the rectangle height to bind the text after being word-wrapped.

The output of `mmiDrawTextCenterClick()` is shown in Figure 8.12.

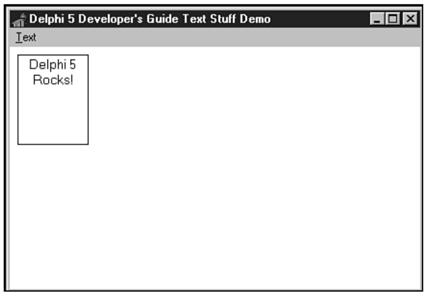


FIGURE 8.12

The output of `mmiDrawTextCenterClick()`.

`TCanvas` also has the methods `Draw()`, `Copy()`, `CopyRect()`, and `StretchDraw()`, which enable you to draw, copy, expand, and shrink an image or a portion of an image to another canvas. You'll use `CopyRect()` when we show you how to create a paint program later in this chapter. Also, Chapter 16, "MDI Applications," shows you how to use the `StretchDraw()` method to stretch a bitmap image onto the client area of a form.

Coordinate Systems and Mapping Modes

Most GDI drawing routines require a set of coordinates that specify the location where drawing is to occur. These coordinates are based on a unit of measurement, such as the pixel. Additionally, GDI routines assume an orientation for the vertical and horizontal axis—that is, how increasing or decreasing the values of the X,Y coordinates moves the position at which drawing occurs. Win32 relies on two factors to perform drawing routines. These are the Win32 coordinates system and the mapping mode of the area that's being drawn to.

Win32 coordinates systems are, generally, no different from any other coordinates system. You define a coordinate for an X,Y axis, and Win32 plots that location to a point on your drawing surface based on a given orientation. Win32 uses three coordinates systems to plot areas on drawing surfaces called the *device*, *logical*, and *world* coordinates. Windows 95 doesn't support world transformations (bitmap rotation, shearing, twisting, and so on). We'll cover the first two modes in this chapter.

Device Coordinates

Device coordinates, as the name implies, refer to the device on which Win32 is running. Its measurements are in pixels, and the orientation is such that the horizontal and vertical axes increase from left to right and top to bottom. For example, if you're running Windows on a 640×480 pixel display, the coordinates at the top-left corner on your device are (0,0), whereas the bottom-right coordinates are (639,479).

Logical Coordinates

Logical coordinates refer to the coordinates system used by any area in Win32 that has a device context or DC such as a screen, a form, or a form's client area. The difference between device and logical coordinates is explained in a moment. The screen, form window, and form client-area coordinates are explained first.

Screen Coordinates

Screen coordinates refer to the display device; therefore, it follows that coordinates are based on pixel measurements. On a 640×480 display, `Screen.Width` and `Screen.Height` are also 640 and 480 pixels, respectively. To obtain a device context for the screen, use the Win32 API function `GetDC()`. You must match any function that retrieves a device context with a call to `ReleaseDC()`. The following code illustrates this:

```
var
  ScreenDC: HDC;
begin
  Screen DC := GetDC(0);
  try
    { Do whatever you need to do with ScreenDC }
  finally
     ReleaseDC(0, ScreenDC);
  end;
end;
```

Form Coordinates

Form coordinates are synonymous with the term *window coordinates* and refer to an entire form or window, including the caption bar and borders. Delphi 5 doesn't provide a DC to the form's drawing area through a form's property, but you can obtain one by using the Win32 API function `GetWindowDC()`, as follows:

```
MyDC := GetWindowDC(Form1.Handle);
```

This function returns the DC for the window handle passed to it.

NOTE

You can use a `TCanvas` object to encapsulate the device contexts obtained from the calls to `GetDC()` and `GetWindowDC()`. This enables you to use the `TCanvas` methods against those device contexts. You just need to create a `TCanvas` instance and then assign the result of `GetDC()` or `GetWindowDC()` to the `TCanvas.Handle` property. This works because the TCanvas object takes ownership of the handle you assign to it, and it will release the DC when the canvas is freed. The following code illustrates this technique:

```
var
  c: TCanvas;
begin
  c := TCanvas.Create;
  try
    c.Handle := GetDC(0);
    c.TextOut(10, 10, 'Hello World');
  finally
    c.Free;
  end;
end;
```

A form's *client-area coordinates* refer to a form's client area whose DC is the `Handle` property of the form's `Canvas` and whose measurements are obtained from `Canvas.ClientWidth` and `Canvas.ClientHeight`.

Coordinate Mapping

So why not just use device coordinates instead of logical coordinates when performing drawing routines? Examine the following line of code:

```
Form1.Canvas.TextOut(0, 0, 'Upper Left Corner of Form');
```

This line places the string at the upper-left corner of the form. The coordinates (0,0) map to the position (0,0) in the form's device context—logical coordinates. However, the position (0,0) for the form is completely different in device coordinates and depends on where the form is located on your screen. If the form just happens to be located at the upper-left corner of your screen, the form's coordinates (0,0) may in fact map to (0,0) in device coordinates. However, as you move the form to another location, the form's position (0,0) will map to a completely different location on the device.

TIP

You can obtain a point based on device coordinates from the point as it's represented in logical coordinates, and vice versa, using the Win32 API functions `ClientToScreen()` and `ScreenToClient()`, respectively. These are also `TControl` methods. Note that this works only with screen DCs associated with a visible control.

continues

For printer or metafile DCs that are not screen based, convert logical pixels to device pixels by using the `LPtoDP()` Win32 function. Also see `DPtoLP() in the Win32 online help`.

Underneath the call to `Canvas.TextOut()`, Win32 does actually use device coordinates. For Win32 to do this, it must "map" the logical coordinates of the DC being drawn to, to device coordinates. It does this using the mapping mode associated with the DC.

Another reason for using logical coordinates is that you might not want to use pixels to perform drawing routines. Perhaps you want to draw using inches or millimeters. Win32 enables you to change the unit with which you perform your drawing routines by changing its mapping mode, as you'll see in a moment.

Mapping modes define two attributes for the DC: the translation that Win32 uses to convert logical units to device units, and the orientation of the X,Y axis for the DC.

NOTE

It might not seem apparent that drawing routines, mapping modes, orientation, and so on are associated with a DC because, in Delphi 5, you use the canvas to draw. Remember that `TCanvas` is a wrapper for a DC. This becomes obvious when comparing Win32 GDI routines to their equivalent `Canvas` routines. Here are examples:

`Canvas` routine: `Canvas.Rectangle(0, 0, 50, 50));`

GDI routine: `Rectangle(ADC, 0, 0, 50, 50);`

When you're using the GDI routine, a DC is passed to the function, whereas the canvas's routine uses the DC that it encapsulates.

Win32 enables to you define the mapping mode for a DC or `TCanvas.Handle`. In fact, Win32 defines eight mapping modes you can use. These mapping modes, along with their attributes, are shown in Table 8.4. The sample project in the next section illustrates more about mapping modes.

TABLE 8.4 Win32 Mapping Modes

Mapping Mode	*Logical Unit Size*	*Orientation (X,Y)*
`MM_ANISOTROPIC`	Arbitrary (x <> y) or (x = y)	Definable/definable
`MM_HIENGLISH`	0.001 inch	Right/up

Mapping Mode	*Logical Unit Size*	*Orientation (X,Y)*
`MM_HIMETRIC`	0.01 mm	Right/up
`MM_ISOTROPIC`	arbitrary (x = y)	Definable/definable
`MM_LOENGLISH`	0.01 inch	Right/up
`MM_LOMETRIC`	0.1 mm	Right/up
`MM_TEXT`	1 pixel	Right/down
`MM_TWIPS`	1⁄1440 inch	Right/up

Win32 defines a few functions that enable you to change or retrieve information about the mapping modes for a given DC. Here's a summary of these functions:

- `SetMapMode()`. Sets the mapping mode for a given device context.
- `GetMapMode()`. Gets the mapping mode for a given device context.
- `SetWindowOrgEx()`. Defines an window origin (point 0,0) for a given DC.
- `SetViewPortOrgEx()`. Defines a viewport origin (point 0,0) for a given DC.
- `SetWindowExtEx()`. Defines the X,Y extents for a given window DC. These values are used in conjunction with the viewport X,Y extents to perform translation from logical units to device units.
- `SetViewPortExtEx()`. Defines the X,Y extents for a given viewport DC. These values are used in conjunction with the window X,Y extents to perform translation from logical units to device units.

Notice that these functions contain either the word `Window` or `ViewPort`. The window or viewport is simply a means by which Win32 GDI can perform the translation from logical to device units. The functions with `Window` refer to the logical coordinate system, whereas those with `ViewPort` refer to the device coordinates system. With the exception of the `MM_ANISOTROPIC` and `MM_ISOTROPIC` mapping modes, you don't have to worry about this much. In fact, Win32 uses the `MM_TEXT` mapping mode by default.

NOTE

`MM_TEXT` is the default mapping mode, and it maps logical coordinates 1:1 with device coordinates. So, you're always using device coordinates on all DCs, unless you change the mapping mode. There are some API functions where this is significant: Font heights, for example, are always specified in device pixels, not logical pixels.

Setting the Mapping Mode

You'll notice that each mapping mode uses a different logical unit size. In some cases, it might be convenient to use a different mapping mode for that reason. For example, you might want to display a line 2 inches wide, regardless of the resolution of your output device. In this instance, `MM_LOENGLISH` would be a good candidate for a mapping mode to use.

As an example of drawing a 1-inch rectangle to the form, you first change the mapping mode for `Form1.Canvas.Handle` to `MM_HIENGLISH` or `MM_LOENGLISH`:

```
SetMapMode(Canvas.Handle, MM_LOENGLISH);
```

Then you draw the rectangle using the appropriate units of measurement for a 1-inch rectangle. Because `MM_LOENGLISH` uses 1⁄100 inch, you simply pass the value `100`, as follows (this will be illustrated further in a later example):

```
Canvas.Rectangle(0, 0, 100, 100);
```

Because `MM_TEXT` uses pixels as its unit of measurement, you can use the Win32 API function `GetDeviceCaps()` to retrieve the information you need to perform translation from pixels to inches or millimeters. Then you can do your own calculations if you want. This is demonstrated in Chapter 10, "Printing in Delphi 5." Mapping modes are a way to let Win32 do the work for you. Note, however, that you'll most likely never be able to get exact measurements for screen displays. There are a few reasons for this: Windows cannot record the display size of the screen—it must guess. Also, Windows typically inflates display scales to improve text readability on relatively chunky monitors. So, for example, a 10-point font on a screen is about as tall as a 12- to 14-point font on paper.

Setting the Window/Viewport Extents

The `SetWindowExtEx()` and `SetViewPortExtEx()` functions enable you to define how Win32 translates logical units to device units. These functions have an effect only when the window's mapping mode is either `MM_ANISOTROPIC` or `MM_ISOTROPIC`. They are ignored otherwise. Therefore, the following lines of code mean that one logical unit requires two device units (pixels):

```
SetWindowExtEx(Canvas.Handle, 1, 1, nil)
SetViewportExtEx(Canvas.Handle, 2, 2, nil);
```

Likewise, these lines of code mean that five logical units require 10 device units:

```
SetWindowExtEx(Canvas.Handle, 5, 5, nil)
SetViewportExtEx(Canvas.Handle, 10, 10, nil);
```

Notice that this is exactly the same as the previous example. Both have the same effect of having a 1:2 ratio of logical to device units. Here's an example of how this may be used to change the units for a form:

```
SetWindowExtEx(Canvas.Handle, 500, 500, nil)
SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
```

NOTE

Changing the mapping mode for a device context represented by a VCL canvas is not "sticky," which means that it may revert back to its original mode. Generally, the map mode must be set within the handler doing the actual drawing.

This enables to you work with a form whose client width and height are 500×500 units (not pixels) despite any resizing of the form.

The `SetWindowOrgEx()` and `SetViewPortOrgEx()` functions enable you to relocate the origin or position (0,0), which, by default, is at the upper-left corner of a form's client area in the `MM_TEXT` mapping mode. Typically, you just modify the viewport origin. For example, the following line sets up a four-quadrant coordinate system like the one illustrated in Figure 8.13:

```
SetViewportOrgEx(Canvas.Handle, ClientWidth div 2, ClientHeight div 2, nil);
```

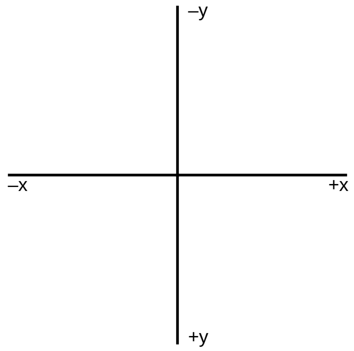


FIGURE 8.13

A four-quadrant coordinate system.

Notice that we pass a `nil` value as the last parameter in the `SetWindowOrgEx()`, `SetViewPortOrgEx()`, `SetWindowExtEx()`, and `SetViewPortExtEx()` functions. The `SetWindowOrgEx()` and `SetViewPortOrgEx()` functions take a `TPoint` variable that gets assigned the last origin value so that you can restore the origin for the DC, if necessary. The `SetWindowExtEx()` and `SetViewPortExtEx()` functions take a `TSize` structure to store the original extents for the DC for the same reason.

Mapping Mode Example Project

Listing 8.6 shows you the unit for a project. This project illustrates how to set mapping modes, window and viewport origins, and windows and viewport extents. It also illustrates how to draw various shapes using `TCanvas` methods. You can load this project from the CD.

LISTING 8.6 An Illustration of Mapping Modes

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, DB, DBCGrids, DBTables;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiMappingMode: TMenuItem;
    mmiMM_ISOTROPIC: TMenuItem;
    mmiMM_ANSITROPIC: TMenuItem;
    mmiMM_LOENGLISH: TMenuItem;
    mmiMM_HIINGLISH: TMenuItem;
    mmiMM_LOMETRIC: TMenuItem;
    mmiMM_HIMETRIC: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure mmiMM_ISOTROPICClick(Sender: TObject);
    procedure mmiMM_ANSITROPICClick(Sender: TObject);
    procedure mmiMM_LOENGLISHClick(Sender: TObject);
    procedure mmiMM_HIINGLISHClick(Sender: TObject);
    procedure mmiMM_LOMETRICClick(Sender: TObject);
    procedure mmiMM_HIMETRICClick(Sender: TObject);
  public
    MappingMode: Integer;
    procedure ClearCanvas;
    procedure DrawMapMode(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
```

```
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;

procedure TMainForm.DrawMapMode(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  Canvas.TextOut(0, 0, (Sender as TMenuItem).Caption);

  // Set mapping mode to MM_LOENGLISH and save the previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MappingMode);
  try
    // Set the viewport org to left, bottom
    SetViewPortOrgEx(Canvas.Handle, 0, ClientHeight, nil);
    { Draw some shapes to illustrate drawing shapes with different
      mapping modes specified by MappingMode }
    Canvas.Rectangle(0, 0, 200, 200);
    Canvas.Rectangle(200, 200, 400, 400);
    Canvas.Ellipse(200, 200, 400, 400);
    Canvas.MoveTo(0, 0);
    Canvas.LineTo(400, 400);
    Canvas.MoveTo(0, 200);
    Canvas.LineTo(200, 0);
  finally
    // Restore previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  MappingMode := MM_TEXT;
end;

procedure TMainForm.mmiMM_ISOTROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  // Set mapping mode to MM_ISOTROPIC and save the previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ISOTROPIC);
```

continues

LISTING 8.6 Continued

```
  try
    // Set the window extent to 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Set the Viewport extent to the Window's client area
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Set the ViewPortOrg to the center of the client area
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
       ClientHeight div 2, nil);
    // Draw a rectangle based on current settings
    Canvas.Rectangle(0, 0, 250, 250);
    { Set the viewport extent to a different value, and
      draw another rectangle. continue to do this three
      more times so that a rectangle is draw to represent
      the plane in a four-quadrant square }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Draw an ellipse in the center of the client area
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    // Restore the previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.mmiMM_ANSITROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  // Set the mapping mode to MM_ANISOTROPIC and save the
  // previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ANISOTROPIC);
  try
    // Set the window extent to 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Set the Viewport extent to that of the Window's client area
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Set the ViewPortOrg to the center of the client area
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
       ClientHeight div 2, nil);
    // Draw a rectangle based on current settings
    Canvas.Rectangle(0, 0, 250, 250);
    { Set the viewport extent to a different value, and
      draw another rectangle. continue to do this three
      more times so that a rectangle is draw to represent
      the plane in a four-quadrant square }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Draw an ellipse in the center of the client area
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    //Restore the previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.mmiMM_LOENGLISHClick(Sender: TObject);
begin
  MappingMode := MM_LOENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIINGLISHClick(Sender: TObject);
begin
  MappingMode := MM_HIENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_LOMETRICClick(Sender: TObject);
begin
  MappingMode := MM_LOMETRIC;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIMETRICClick(Sender: TObject);
 begin
  MappingMode := MM_HIMETRIC;
  DrawMapMode(Sender);
end;

end.
```

The main form's field `MappingMode` is used to hold the current mapping mode that's initialized in the `FormCreate()` method to `MM_TEXT`. This variable gets set whenever the `MMLOENGLISH1Click()`, `MMHIENGLISH1Click()`, `MMLOMETRIC1Click()`, and `MMHIMETRIC1Click()` methods are invoked from their respective menus. These methods then call the method `DrawMapMode()`, which sets the main form's mapping mode to that specified by `MappingMode`. It then draws some shapes and lines using constant values to specify their sizes. When different mapping modes are used when drawing the shapes, they'll be sized differently on the form because the measurements used are used in the context of the specified mapping mode. Figures 8.14 and 8.15 illustrate `DrawMapMode()`'s output for the `MM_LOENGLISH` and `MM_LOMETRIC` mapping modes.

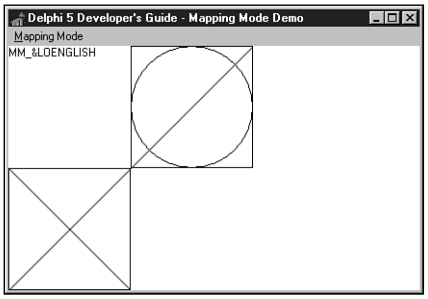


FIGURE 8.14
`DrawMapMode()` *output using* `MM_LOENGLISH` *mapping mode.*

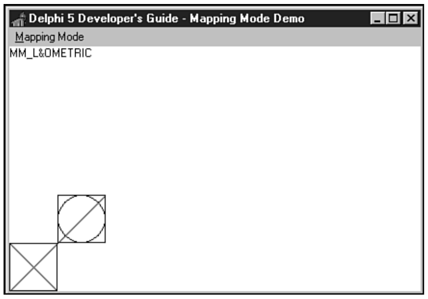


FIGURE 8.15
`DrawMapMode()` *output using* `MM_LOMETRIC` *mapping mode.*

The `mmiMM_ISOTROPICClick()` method illustrates drawing with the form's canvas in the `MM_ISOTROPIC` mode. This method first sets the mapping mode and then sets the canvas's viewport extent to that of the form's client area. The origin is then set to the center of the form's client area, which allows all four quadrants of the coordinate system to be viewed.

The method then draws a rectangle in each plane and an ellipse in the center of the client area. Notice how you can use the same values in the parameters passed to `Canvas.Rectangle()` yet draw to different areas of the canvas. This is accomplished by passing negative values to the X parameter, Y parameter, or both parameters passed to `SetViewPortExt()`.

The `mmiMM_ANISOTROPICClick()` method performs the same operations except it uses the `MM_ANISOTROPIC` mode. The purpose of showing both is to illustrate the principle difference between the `MM_ISOTROPIC` and `MM_ANISOTROPIC` mapping modes.

Using the `MM_ISOTROPIC` mode, Win32 ensures that the two axes use the same physical size and makes the necessary adjustments to see this is the case. The `MM_ANISOTROPIC` mode, however, uses physical dimensions that might not be equal. Figures 8.16 and 8.17 illustrate this more clearly. You can see that the `MM_ISOTROPIC` mode ensures equality with the two axes, whereas the same code using the `MM_ANISOTROPIC` mode does not ensure equality. In fact, the `MM_ISOTROPIC` mode further guarantees that the square logical coordinates will be mapped to device coordinates such that squareness will be preserved, even if the device coordinates system is not square.

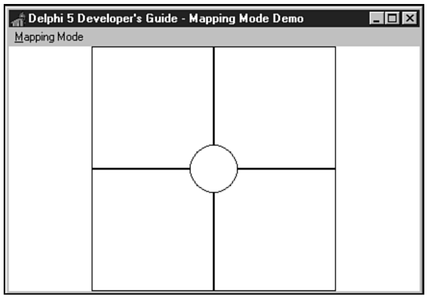


FIGURE 8.16

`MM_ISOTROPIC` *mapping mode output.*

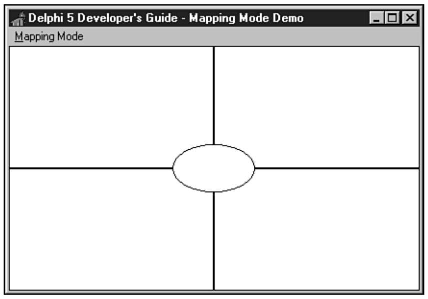


FIGURE 8.17

`MM_ANISOTROPIC` *mapping mode output.*

Creating a Paint Program

The paint program shown here uses several advanced techniques in working with GDI and graphics images. You can find this project on the CD as `DDGPaint.dpr`. Listing 8.7 shows the source code for this project.

LISTING 8.7 The Paint Program: `DDGPaint`

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, ExtCtrls, ColorGrd, StdCtrls, Menus,
  ComCtrls;

const
  crMove = 1;
type

  TDrawType = (dtLineDraw, dtRectangle, dtEllipse, dtRoundRect,
               dtClipRect, dtCrooked);

  TMainForm = class(TForm)
    sbxMain: TScrollBox;
    imgDrawingPad: TImage;
    pnlToolBar: TPanel;
    sbLine: TSpeedButton;
    sbRectangle: TSpeedButton;
    sbEllipse: TSpeedButton;
    sbRoundRect: TSpeedButton;
    pnlColors: TPanel;
    cgDrawingColors: TColorGrid;
    pnlFgBgBorder: TPanel;
    pnlFgBgInner: TPanel;
    Bevel1: TBevel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N2: TMenuItem;
    mmiSaveAs: TMenuItem;
    mmiSaveFile: TMenuItem;
    mmiOpenFile: TMenuItem;
    mmiNewFile: TMenuItem;
    mmiEdit: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    sbRectSelect: TSpeedButton;
    SaveDialog: TSaveDialog;
    OpenDialog: TOpenDialog;
    stbMain: TStatusBar;
    pbPasteBox: TPaintBox;
    sbFreeForm: TSpeedButton;
    RgGrpFillOptions: TRadioGroup;
    cbxBorder: TCheckBox;
    procedure FormCreate(Sender: TObject);
    procedure sbLineClick(Sender: TObject);
    procedure imgDrawingPadMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure cgDrawingColorsChange(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiSaveFileClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure mmiNewFileClick(Sender: TObject);
    procedure mmiOpenFileClick(Sender: TObject);
    procedure mmiEditClick(Sender: TObject);
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure pbPasteBoxMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure pbPasteBoxPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure RgGrpFillOptionsClick(Sender: TObject);
  public
    { Public declarations }
    MouseOrg: TPoint;    // Stores mouse information
    NextPoint: TPoint;   // Stores mouse information
    Drawing: Boolean;    // Drawing is being performed flag
    DrawType: TDrawType; // Holds the draw type information: TDrawType
    FillSelected,        // Fill shapes flag
```

continues

LISTING 8.7 Continued

```
    BorderSelected: Boolean;  // Draw Shapes with no border flag
    EraseClipRect: Boolean;     // Specifies whether or not to erase the
                                // clipping rectangle
    Modified: Boolean;     // Image modified flag
    FileName: String;      // Holds the filename of the image
    OldClipViewHwnd: Hwnd; // Holds the old clipboard view window
    { Paste Image variables }
    PBoxMoving: Boolean;   // PasteBox is moving flag
    PBoxMouseOrg: TPoint;  // Stores mouse coordinates for PasteBox
    PasteBitMap: TBitmap;  // Stores a bitmap image of the pasted data
    Pasted: Boolean;       // Data pasted flag
    LastDot: TPoint;       // Hold the TPoint coordinate for performing
                           // free line drawing
    procedure DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
    { This procedure paints the image specified by the DrawType field
      to imgDrawingPad }
    procedure SetDrawingStyle;
    { This procedure sets various Pen/Brush styles based on values
      specified by the form's controls. The Panels and color grid is
      used to set these values }
    procedure CopyPasteBoxToImage;
    { This procedure copies the data pasted from the Windows clipboard
      onto the main image component imgDrawingPad }
    procedure WMDrawClipBoard(var Msg: TWMDrawClipBoard);
       message WM_DRAWCLIPBOARD;
    { This message handler captures the WM_DRAWCLIPBOARD messages
      which is sent to all windows that have been added to the clipboard
      viewer chain. An application can add itself to the clipboard viewer
      chain by using the SetClipBoardViewer() Win32 API function as
      is done in FormCreate() }
    procedure CopyCut(Cut: Boolean);
    { This method copies a portion of the main image, imgDrawingPad,
       to the Window's clipboard. }
  end;

var
  MainForm: TMainForm;

implementation
uses ClipBrd, Math;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
{ This method sets the form's field to their default values. It then
```

```
  creates a bitmap for the imgDrawingPad.  This is the image on which
  drawing is done. Finally, it adds this application as part of the
  Windows clipboard viewer chain by using the SetClipBoardViewer()
  function. This makes enables the form to get WM_DRAWCLIPBOARD messages
  which are sent to all windows in the clipboard viewer chain whenever
  the clipboard data is modified. }
begin
  Screen.Cursors[crMove] := LoadCursor(hInstance, 'MOVE');

  FillSelected   := False;
  BorderSelected := True;

  Modified := False;
  FileName := '';
  Pasted := False;
  pbPasteBox.Enabled := False;

  // Create a bitmap for imgDrawingPad and set its boundaries
  with imgDrawingPad do
  begin
    SetBounds(0, 0, 600, 400);
    Picture.Graphic := TBitMap.Create;
    Picture.Graphic.Width := 600;
    Picture.Graphic.Height := 400;
  end;
  // Now create a bitmap image to hold pasted data
  PasteBitmap := TBitmap.Create;
  pbPasteBox.BringToFront;
  { Add the form to the Windows clipboard viewer chain. Save the handle
    of the next window in the chain so that it may be restored by the
    ChangeClipboardChange() Win32 API function in this form's
    FormDestroy() method. }
  OldClipViewHwnd := SetClipBoardViewer(Handle);
end;

procedure TMainForm.WMDrawClipBoard(var Msg: TWMDrawClipBoard);
begin
  { This method will be called whenever the clipboard data
    has changed. Because the main form was added to the clipboard
    viewer chain, it will receive the WM_DRAWCLIPBOARD message
    indicating that the clipboard's data was changed. }
  inherited;
  { Make sure that the data contained on the clipboard is actually
    bitmap data. }
  if ClipBoard.HasFormat(CF_BITMAP) then
    mmiPaste.Enabled := True
```

continues

LISTING 8.7 Continued

```
  else
    mmiPaste.Enabled := False;
  Msg.Result := 0;
end;


procedure TMainForm.DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
{ This method performs the specified drawing operation. The
  drawing operation is specified by the DrawType field }
begin
  with imgDrawingPad.Canvas do
  begin
    Pen.Mode := PenMode;

    case DrawType of
      dtLineDraw:
        begin
          MoveTo(TL.X, TL.Y);
          LineTo(BR.X, BR.Y);
        end;
      dtRectangle:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
      dtEllipse:
        Ellipse(TL.X, TL.Y, BR.X, BR.Y);
      dtRoundRect:
        RoundRect(TL.X, TL.Y, BR.X, BR.Y,
          (TL.X - BR.X) div 2, (TL.Y - BR.Y) div 2);
      dtClipRect:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
    end;
  end;
end;

procedure TMainForm.CopyPasteBoxToImage;
{ This method copies the image pasted from the Windows clipboard onto
  imgDrawingPad. It first erases any bounding rectangle drawn by PaintBox
  component, pbPasteBox. It then copies the data from pbPasteBox onto
  imgDrawingPad at the location where pbPasteBox has been dragged
  over imgDrawingPad. The reason we don't copy the contents of
  pbPasteBox's canvas and use PasteBitmap's canvas instead, is because
  when a portion of pbPasteBox is dragged out of the viewable area,
  Windows does not paint the portion pbPasteBox not visible. Therefore,
  it is necessary to the pasted bitmap from the off-screen bitmap }
var
```

```
  SrcRect, DestRect: TRect;
begin
  // First, erase the rectangle drawn by pbPasteBox
  with pbPasteBox do
  begin
    Canvas.Pen.Mode := pmNotXOR;
    Canvas.Pen.Style := psDot;
    Canvas.Brush.Style := bsClear;
    Canvas.Rectangle(0, 0, Width, Height);
    DestRect := Rect(Left, Top, Left+Width, Top+Height);
    SrcRect := Rect(0, 0, Width, Height);
  end;
  { Here we must use the PasteBitmap instead of the pbPasteBox because
    pbPasteBox will clip anything outside if the viewable area. }
  imgDrawingPad.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
  pbPasteBox.Visible := false;
  pbPasteBox.Enabled := false;
  Pasted := False;  // Pasting operation is complete
end;

procedure TMainForm.imgDrawingPadMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Modified := True;
  // Erase the clipping rectangle if one has been drawn
  if (DrawType = dtClipRect) and EraseClipRect then
    DrawToImage(MouseOrg, NextPoint, pmNotXOR)
  else if (DrawType = dtClipRect) then
    EraseClipRect := True; // Re-enable cliprect erasing

  { If an bitmap was pasted from the clipboard, copy it to the
    image and remove the PaintBox. }
  if Pasted then
    CopyPasteBoxToImage;

  Drawing := True;
  // Save the mouse information
  MouseOrg := Point(X, Y);
  NextPoint := MouseOrg;
  LastDot := NextPoint;   // Lastdot is updated as the mouse moves
  imgDrawingPad.Canvas.MoveTo(X, Y);
end;

procedure TMainForm.imgDrawingPadMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
{ This method determines the drawing operation to be performed and
```

continues

LISTING 8.7 Continued

```
  either performs free form line drawing, or calls the
  DrawToImage method which draws the specified shape }
begin
  if Drawing then
  begin
    if DrawType = dtCrooked then
    begin
      imgDrawingPad.Canvas.MoveTo(LastDot.X, LastDot.Y);
      imgDrawingPad.Canvas.LineTo(X, Y);
      LastDot := Point(X,Y);
    end
    else begin
      DrawToImage(MouseOrg, NextPoint, pmNotXor);
      NextPoint := Point(X, Y);
      DrawToImage(MouseOrg, NextPoint, pmNotXor)
    end;
  end;
  // Update the status bar with the current mouse location
  stbMain.Panels[1].Text := Format('X: %d, Y: %D', [X, Y]);
end;

procedure TMainForm.imgDrawingPadMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  { Prevent the clipping rectangle from destroying the images already
    on the image }
    if not (DrawType = dtClipRect) then
      DrawToImage(MouseOrg, Point(X, Y), pmCopy);
  Drawing := False;
end;

procedure TMainForm.sbLineClick(Sender: TObject);
begin
  // First erase the cliprect if current drawing type
  if DrawType = dtClipRect then
     DrawToImage(MouseOrg, NextPoint, pmNotXOR);

  { Now set the DrawType field to that specified by the TSpeedButton
    invoking this method. The TSpeedButton's Tag values match a
    specific TDrawType value which is why the typecasting below
    successfully assigns a valid TDrawType value to the DrawType field. }
  if Sender is TSpeedButton then
    DrawType := TDrawType(TSpeedButton(Sender).Tag);
  // Now make sure the dtClipRect style doesn't erase previous drawings
  if DrawType = dtClipRect then begin
    EraseClipRect := False;
  end;
  // Set the drawing style
  SetDrawingStyle;
end;

procedure TMainForm.cgDrawingColorsChange(Sender: TObject);
{ This method draws the rectangle representing fill and border colors
  to indicate the users selection of both colors. pnlFgBgInner and
  pnlFgBgBorder are TPanels arranged one on to of the other for the
  desired effect }
begin
  pnlFgBgBorder.Color := cgDrawingColors.ForeGroundColor;
  pnlFgBgInner.Color := cgDrawingColors.BackGroundColor;
  SetDrawingStyle;
end;


procedure TMainForm.SetDrawingStyle;
{  This method sets the various drawing styles based on the selections
   on the pnlFillStyle TPanel for Fill and Border styles }
begin
  with imgDrawingPad do
  begin
    if DrawType = dtClipRect then
    begin
      Canvas.Pen.Style := psDot;
      Canvas.Brush.Style := bsClear;
      Canvas.Pen.Color := clBlack;
    end

    else if FillSelected then
      Canvas.Brush.Style := bsSolid
    else
      Canvas.Brush.Style := bsClear;

    if BorderSelected then
      Canvas.Pen.Style := psSolid
    else
      Canvas.Pen.Style := psClear;


    if FillSelected and (DrawType <> dtClipRect) then
     Canvas.Brush.Color := pnlFgBgInner.Color;
```

continues

LISTING 8.7 Continued

```
    if DrawType <> dtClipRect then
      Canvas.Pen.Color := pnlFgBgBorder.Color;
  end;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close; // Terminate application
end;

procedure TMainForm.mmiSaveFileClick(Sender: TObject);
{ This method saves the image to the file specified by FileName. If
  FileName is blank, however, SaveAs1Click is called to get a filename.}
begin
  if FileName = '' then
    mmiSaveAsClick(nil)
  else begin
    imgDrawingPad.Picture.SaveToFile(FileName);
    stbMain.Panels[0].Text := FileName;
    Modified := False;
  end;
end;

procedure TMainForm.mmiSaveAsClick(Sender: TObject);
{ This method launches SaveDialog to get a file name to which
  the image's contents will be saved. }
begin
  if SaveDialog.Execute then
  begin
    FileName := SaveDialog.FileName;  // Store the filename
    mmiSaveFileClick(nil)
  end;
end;

procedure TMainForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
{ If the user attempts to close the form before saving the image, they
  are prompted to do so in this method. }
var
  Rslt: Word;
begin
  CanClose := False; // Assume fail.
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?',
      mtConfirmation, mbYesNOCancel, 0);
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // no need to do anything.
      mrCancel: Exit;
    end
  end;
  CanClose := True;    // Allow use to close application
end;

procedure TMainForm.mmiNewFileClick(Sender: TObject);
{ This method erases any drawing on the main image after prompting the
  user to save it to a file in which case the mmiSaveFileClick event handler
  is called. }
var
  Rslt: Word;
begin
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?', mtConfirmation,
mbYesNOCancel, 0);
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // no need to do anything.
      mrCancel: Exit;
    end
  end;

   with imgDrawingPad.Canvas do begin
     Brush.Style := bsSolid;
     Brush.Color := clWhite;  // clWhite erases the image
     FillRect(ClipRect);      // Erase the image
     FileName := '';
     stbMain.Panels[0].Text := FileName;
   end;
   SetDrawingStyle;   // Restore the previous drawing style
   Modified := False;
end;

procedure TMainForm.mmiOpenFileClick(Sender: TObject);
{ This method opens a bitmap file specified by OpenDialog.FileName. If
  a file was already created, the user is prompted to save
  the file in which case the mmiSaveFileClick event is called. }
var
  Rslt: Word;
begin

  if OpenDialog.Execute then
```

continues

Listing 8.7 Continued

```
  begin

    if Modified then begin
      Rslt := MessageDlg('File has changed, save?',
        mtConfirmation, mbYesNOCancel, 0);
      case Rslt of
        mrYes: mmiSaveFileClick(nil);
        mrNo: ;  // no need to do anything.
        mrCancel: Exit;
      end
    end;

    imgDrawingPad.Picture.LoadFromFile(OpenDialog.FileName);
    FileName := OpenDialog.FileName;
    stbMain.Panels[0].Text := FileName;
    Modified := false;
  end;

end;

procedure TMainForm.mmiEditClick(Sender: TObject);
{ The timer is used to determine if an area on the main image is
  surrounded by a bounding rectangle. If so, then the Copy and Cut
  menu items are enabled. Otherwise, they are disabled. }
var
  IsRect: Boolean;
begin
  IsRect := (MouseOrg.X <> NextPoint.X) and (MouseOrg.Y <> NextPoint.Y);
  if (DrawType = dtClipRect) and IsRect then
  begin
    mmiCut.Enabled := True;
    mmiCopy.Enabled := True;
  end
  else begin
    mmiCut.Enabled := False;
    mmiCopy.Enabled := False;
  end;
end;

procedure TMainForm.CopyCut(Cut: Boolean);
{ This method copies a portion of the main image to the clipboard.
  The portion copied is specified by a bounding rectangle
  on the main image. If Cut is true, the area in the bounding rectangle
  is erased. }
var
```

```
   CopyBitMap: TBitmap;
   DestRect, SrcRect: TRect;
   OldBrushColor: TColor;
begin
  CopyBitMap := TBitMap.Create;
  try
    { Set CopyBitmap's size based on the coordinates of the
      bounding rectangle }
    CopyBitMap.Width := Abs(NextPoint.X - MouseOrg.X);
    CopyBitMap.Height := Abs(NextPoint.Y - MouseOrg.Y);
    DestRect := Rect(0, 0, CopyBitMap.Width, CopyBitmap.Height);
    SrcRect := Rect(Min(MouseOrg.X, NextPoint.X)+1,
                    Min(MouseOrg.Y, NextPoint.Y)+1,
                    Max(MouseOrg.X, NextPoint.X)-1,
                    Max(MouseOrg.Y, NextPoint.Y)-1);
    { Copy the portion of the main image surrounded by the bounding
      rectangle to the Windows clipboard }
    CopyBitMap.Canvas.CopyRect(DestRect, imgDrawingPad.Canvas, SrcRect);
    { Previous versions of Delphi required the bitmap's Handle property
      to be touched for the bitmap to be made available. This was due to
      Delphi's caching of bitmapped images. The step below may not be
      required. }
    CopyBitMap.Handle;
    // Assign the image to the clipboard.
    ClipBoard.Assign(CopyBitMap);
    { If cut was specified the erase the portion of the main image
      surrounded by the bounding Rectangle }
    if Cut then
      with imgDrawingPad.Canvas do
      begin
        OldBrushColor := Brush.Color;
        Brush.Color := clWhite;
        try
          FillRect(SrcRect);
        finally
          Brush.Color := OldBrushColor;
        end;
      end;
  finally
    CopyBitMap.Free;
  end;
end;

procedure TMainForm.mmiCutClick(Sender: TObject);
begin
  CopyCut(True);
```

continues

LISTING 8.7 Continued

```
end;

procedure TMainForm.mmiCopyClick(Sender: TObject);
begin
  CopyCut(False);
end;

procedure TMainForm.mmiPasteClick(Sender: TObject);
{ This method pastes the data contained in the clipboard to the
  paste bitmap. The reason it is pasted to the PasteBitmap, an off-
image elsewhere on to the main image. This is done by having the pbPasteBox,
  a TPaintBox component, draw the contents of PasteImage. When the
  user if done positioning the pbPasteBox, the contents of TPasteBitmap
  is drawn to imgDrawingPad at the location specified by pbPasteBox's
  location.}
begin
  { Clear the bounding rectangle }

  pbPasteBox.Enabled := True;
  if DrawType = dtClipRect then
  begin
    DrawToImage(MouseOrg, NextPoint, pmNotXOR);
    EraseClipRect := False;
  end;

  PasteBitmap.Assign(ClipBoard);   // Grab the data from the clipboard
  Pasted := True;
  // Set position of pasted image to top left
  pbPasteBox.Left := 0;
  pbPasteBox.Top := 0;
  // Set the size of pbPasteBox to match the size of PasteBitmap
  pbPasteBox.Width := PasteBitmap.Width;
  pbPasteBox.Height := PasteBitmap.Height;

  pbPasteBox.Visible := True;
  pbPasteBox.Invalidate;
end;

procedure TMainForm.pbPasteBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
{ This method set's up pbPasteBox, a TPaintBox for being moved by the
  user when the left mouse button is held down }
begin
  if Button = mbLeft then
  begin
    PBoxMoving := True;
    Screen.Cursor := crMove;
    PBoxMouseOrg := Point(X, Y);
  end
  else
    PBoxMoving := False;
end;

procedure TMainForm.pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
{ This method moves pbPasteBox if the PBoxMoving flag is true indicating
  that the user is holding down the left mouse button and is dragging
  PaintBox }
begin
  if PBoxMoving then
  begin
    pbPasteBox.Left := pbPasteBox.Left + (X - PBoxMouseOrg.X);
    pbPasteBox.Top := pbPasteBox.Top + (Y - PBoxMouseOrg.Y);
  end;
end;

procedure TMainForm.pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
{ This method disables moving of pbPasteBox when the user lifts the left
  mouse button }
  if PBoxMoving then
  begin
     PBoxMoving := False;
     Screen.Cursor := crDefault;
  end;
  pbPasteBox.Refresh; // Redraw the pbPasteBox.
end;

procedure TMainForm.pbPasteBoxPaint(Sender: TObject);
{ The paintbox is drawn whenever the user selects the Paste option
  form the menu. pbPasteBox draws the contents of PasteBitmap which
  holds the image gotten from the clipboard. The reason for drawing
  PasteBitmap's contents in pbPasteBox, a TPaintBox class, is so that
  the user can also move the object around on top of the main image.
  In other words, pbPasteBox can be moved, and hidden when necessary. }
var
  DestRect, SrcRect: TRect;
begin
  // Display the paintbox only if a pasting operation occurred.
```

continues

LISTING 8.7 Continued

```
  if Pasted then
  begin
    { First paint the contents of PasteBitmap using canvas's CopyRect
      but only if the paintbox is not being moved. This reduces
      flicker }
    if not PBoxMoving then
    begin
      DestRect := Rect(0, 0, pbPasteBox.Width, pbPasteBox.Height);
      SrcRect := Rect(0, 0, PasteBitmap.Width, PasteBitmap.Height);
      pbPasteBox.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
    end;
    { Now copy a bounding rectangle to indicate that pbPasteBox is
      a moveable object. We use a pen mode of pmNotXOR because we
      must erase this rectangle when the user copies PaintBox's
      contents to the main image and we must preserve the original
      contents. }
    pbPasteBox.Canvas.Pen.Mode := pmNotXOR;
    pbPasteBox.Canvas.Pen.Style := psDot;
    pbPasteBox.Canvas.Brush.Style := bsClear;
    pbPasteBox.Canvas.Rectangle(0, 0, pbPasteBox.Width,
      pbPasteBox.Height);
  end;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Remove the form from the clipboard chain
  ChangeClipBoardChain(Handle, OldClipViewHwnd);
  PasteBitmap.Free; // Free the PasteBitmap instance
end;

procedure TMainForm.RgGrpFillOptionsClick(Sender: TObject);
begin
  FillSelected   := RgGrpFillOptions.ItemIndex = 0;
  BorderSelected := cbxBorder.Checked;
  SetDrawingStyle;
end;

end.
```

How the Paint Program Works

The paint program is actually quite a bit of code. Because it would be difficult to explain how it works outside of the code, we've added ample comments to the source. We'll describe the general functionality of the paint program here. The main form is shown in Figure 8.18.

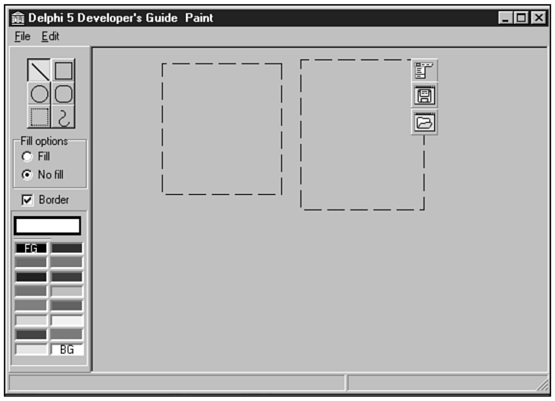


FIGURE 8.18

The main form for the paint program.

NOTE

Notice that a `TImage` component is used as a drawing surface for the paint program. Keep in mind that this can only be the case if the image uses a `TBitmap` object.

The main form contains a main image component, `imgDrawingPad`, which is placed on a `TScrollBox` component. `imgDrawingPad` is where the user performs drawing. The selected speed button on the form's toolbar specifies the type of drawing that the user performs.

The user can draw lines, rectangles, ellipses, and rounded rectangles as well as perform free-form drawing. Additionally, a portion of the main image can be selected and copied to the Windows Clipboard so that it can be pasted into another application that can handle bitmap data. Likewise, the paint program can accept bitmap data from the Windows Clipboard.

TPanel Techniques

The fill style and border type are specified by the Fill Options radio group. The fill and border colors are set using the color grid in the `ColorPanel` shown in Figure 8.18.

Clipboard Pasting of Bitmap Data

To paste data from the Clipboard, you use an offscreen bitmap, `PasteBitMap`, to hold the pasted data. A `TPaintBox` component, `pbPasteBox`, then draws the data from `PasteBitMap`. The reason for using a `TPaintBox` component for drawing the contents of `PasteBitMap` is so the user can move `pbPasteBox` to any location on the main image to designate where the pasted data is to be copied to the main image.

Attaching to the Win32 Clipboard Viewer Chain

Another technique shown by the paint program is how an application can attach itself to the Win32 Clipboard viewer chain. This is done in the `FormCreate()` method by the call to the Win32 API function `SetClipboardViewer()`. This function takes the handle of the window attaching itself to the chain and returns the handle to the next window in the chain. The return value must be stored so that when the application shuts down, it can restore the previous state of the chain using `ChangeClipboardChain()`, which takes the handle being removed and the saved handle. The paint program restores the chain in the main form's `FormDestroy()` method. When an application is attached to the Clipboard viewer chain, it receives the `WM_DRAWCLIPBOARD` messages whenever the data on the Clipboard is modified. You take advantage of this by capturing this message and enabling the Paste menu item if the changed data in the Clipboard is bitmap data. This is done in the `WMDrawClipBoard()` method.

Bitmap Copying

Bitmap copy operations are performed in the `CopyCut()`, `pbPasteBoxPaint()`, and `CopyPasteBoxToImage()` methods. The `CopyCut()` method copies a portion of the main image selected by a bounding rectangle to the Clipboard and then erases the bounded area if the `Cut` parameter passed to it is `True`. Otherwise, it leaves the area intact.

`PbPasteBoxPaint()` copies the contents of the offscreen bitmap to `pbPasteBox.Canvas` but only when `pbPasteBox` is not being moved. This helps reduce flicker as the user moves `pbPasteBox`.

`CopyPasteBoxToImage()` copies the contents of the offscreen bitmap to the main image `imgDrawingPad` at the location specified by `pbPasteBox`.

Paint Program Comments

As mentioned earlier, much of the functionality of the paint program is documented in the code's commentary. It would be a good idea to read through the source and comments and step through the code so that you can gain a good understanding of what's happening in the program.

Performing Animation with Graphics Programming

This section demonstrates how you can achieve simple sprite animation by mixing Delphi 5 classes with Win32 GDI functions. The animation project resides on the CD as `Animate.dpr`. Listing 8.8 shows the main form, which contains the main form functionality.

LISTING 8.8 The Animation Project's Main Form

```
unit MainFrm;

interface
```

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, Stdctrls, AppEvnts;

{$R SPRITES.RES } // Link in the bitmaps

type

  TSprite = class
  private
    FWidth: integer;
    FHeight: integer;
    FLeft: integer;
    FTop: integer;
    FAndImage, FOrImage: TBitMap;
  public
    property Top: Integer read FTop write FTop;
    property Left: Integer read FLeft write FLeft;
    property Width: Integer read FWidth write FWidth;
    property Height: Integer read FHeight write FHeight;
    constructor Create;
    destructor Destroy; override;
  end;

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiSlower: TMenuItem;
    mmiFaster: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    appevMain: TApplicationEvents;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiSlowerClick(Sender: TObject);
    procedure mmiFasterClick(Sender: TObject);
    procedure appevMainIdle(Sender: TObject; var Done: Boolean);
  private
    BackGnd1, BackGnd2: TBitMap;
    Sprite: TSprite;
    GoLeft,GoRight,GoUp,GoDown: boolean;
    FSpeed, FSpeedIndicator: Integer;
    procedure DrawSprite;
  end;
```

continues

LISTING 8.8 Continued

```
const

  BackGround = 'BACK2.BMP';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

constructor TSprite.Create;
begin
  inherited Create;
  { Create the bitmaps to hold the sprite images that will
    be used for performing the AND/OR operation to create animation }
  FAndImage := TBitMap.Create;
  FAndImage.LoadFromResourceName(hInstance, 'AND');

  FOrImage := TBitMap.Create;
  FOrImage.LoadFromResourceName(hInstance, 'OR');

  Left := 0;
  Top := 0;
  Height := FAndImage.Height;
  Width := FAndImage.Width;

end;

destructor TSprite.Destroy;
begin
  FAndImage.Free;
  FOrImage.Free;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Create the original background image
  BackGnd1 := TBitMap.Create;
  with BackGnd1 do
  begin
    LoadFromResourceName(hInstance, 'BACK');
    Parent := nil;
    SetBounds(0, 0,  Width, Height);
  end;
  // Create a copy of the background image
  BackGnd2 := TBitMap.Create;
  BackGnd2.Assign(BackGnd1);


  // Create a sprite image
  Sprite := TSprite.Create;

  // Initialize the direction variables
  GoRight := true;
  GoDown := true;
  GoLeft := false;
  GoUp := false;


  FSpeed := 0;
  FSpeedIndicator := 0;

  { Set the application's OnIdle event to MyIdleEvent which will start
    the sprite moving }

  // Adjust the form's client width/height
  ClientWidth := BackGnd1.Width;
  ClientHeight := BackGnd1.Height;

end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Free all objects created in the form's create constructor
  BackGnd1.Free;
  BackGnd2.Free;
  Sprite.Free;
end;

procedure TMainForm.DrawSprite;
var
  OldBounds: TRect;
begin

  // Save the sprite's bounds in OldBounds
  with OldBounds do
  begin
    Left := Sprite.Left;
    Top := Sprite.Top;
    Right := Sprite.Width;
```

continues

LISTING 8.8 Continued

```
    Bottom := Sprite.Height;
  end;

  { Now change the sprites bounds so that it moves in one direction
    or changes direction when it comes in contact with the form's
    boundaries }
  with Sprite do
  begin
    if GoLeft then
      if Left > 0 then
         Left := Left - 1
      else begin
         GoLeft := false;
         GoRight := true;
      end;

    if GoDown then
      if (Top + Height) < self.ClientHeight then
         Top := Top + 1
      else begin
        GoDown := false;
        GoUp := true;
      end;

    if GoUp then
      if  Top > 0 then
         Top := Top - 1
      else begin
        GoUp := false;
        GoDown := true;
      end;

    if GoRight then
    if (Left + Width) < self.ClientWidth then
      Left := Left + 1
    else begin
      GoRight := false;
      GoLeft := true;
    end;
  end;

  { Erase the original drawing of the sprite on BackGnd2 by copying
    a rectangle from BackGnd1 }
  with OldBounds do
    BitBlt(BackGnd2.Canvas.Handle, Left, Top, Right, Bottom,
           BackGnd1.Canvas.Handle, Left, Top, SrcCopy);
  { Now draw the sprite onto the off-screen bitmap. By performing the
   drawing in an off-screen bitmap, the flicker is eliminated. }
  with Sprite do
  begin
    { Now create a black hole where the sprite first existed by And-ing
      the FAndImage onto BackGnd2 }
    BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
           FAndImage.Canvas.Handle, 0, 0, SrcAnd);
    // Now fill in the black hole with the sprites original colors
    BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
           FOrImage.Canvas.Handle, 0, 0, SrcPaint);
  end;

  { Copy the sprite at its new location to the form's Canvas. A
    rectangle slightly larger than the sprite is needed
    to effectively erase the sprite by over-writing it, and draw the
    new sprite at the new location with a single BitBlt call }
  with OldBounds do
     BitBlt(Canvas.Handle, Left - 2, Top - 2, Right + 2, Bottom + 2,
            BackGnd2.Canvas.Handle, Left - 2, Top - 2, SrcCopy);

end;

procedure TMainForm.FormPaint(Sender: TObject);
begin
  // Draw the background image whenever the form gets painted
  BitBlt(Canvas.Handle, 0, 0, ClientWidth, ClientHeight,
         BackGnd1.Canvas.Handle, 0, 0, SrcCopy);
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiSlowerClick(Sender: TObject);
begin
  Inc(FSpeedIndicator, 100);
end;

procedure TMainForm.mmiFasterClick(Sender: TObject);
begin
  if FSpeedIndicator >= 100 then
    Dec(FSpeedIndicator, 100)
  else
    FSpeedIndicator := 0;
```

continues

LISTING 8.8 Continued

```
end;

procedure TMainForm.appevMainIdle(Sender: TObject; var Done: Boolean);
begin
  if FSpeed >= FSpeedIndicator then
  begin
    DrawSprite;
    FSpeed := 0;
  end
  else
    inc(FSpeed);

Done := False;
end;

end.
```

The animation project consists of a background image on which a sprite, a flying saucer, is drawn and moved about the background's client area. The background is represented by a bitmap consisting of scattered stars (see Figure 8.19).

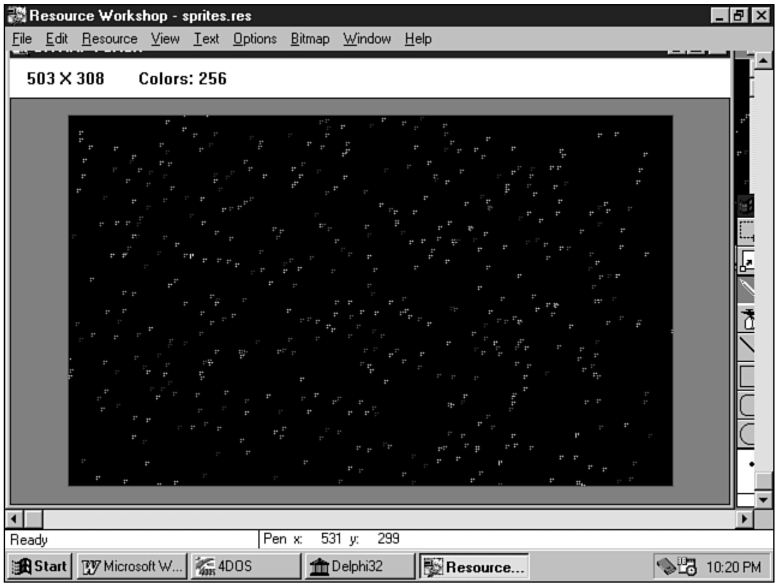


FIGURE 8.19
The background of the animation project.

The sprite is made up of two 64×32 bitmaps. More on these bitmaps later; for now, we'll discuss what goes on in the source code.

The unit defines a `TSprite` class. `TSprite` contains the fields that hold the sprite's location on the background image and two `TBitmap` objects to hold each of the sprite bitmaps. The `TSprite.Create` constructor creates both `TBitmap` instances and loads them with the actual bitmaps. Both the sprite bitmaps and the background bitmap are kept in a resource file that you link to the project by including the following statement in the main unit:

```
{$R SPRITES.RES }
```

After the bitmap is loaded, the sprite's boundaries are set. The `TSprite.Done` destructor frees both bitmap instances.

The main form contains two `TBitmap` objects, a `TSprite` object, and direction indicators to specify the direction of the sprite's motion. Additionally, the main form defines another method, `DrawSprite()`, which has the sprite-drawing functionality. The `TApplicationEvents` component is a new Delphi 5 component that allows you to hook into the `Application`-level events. Prior to Delphi 5, you had to do this by adding `Application`-level events at runtime. Now, with this component, you can do all event management for `TApplication` at design time. We will use this component to provide an OnIdle event for the `TApplication` object.

Note that two private variables are used to control the speed of the animation: `FSpeed` and `FSpeedIndicator`. These are used in the `DrawSprite()` method for slowing down the animation on faster machines.

The `FormCreate()` event handler creates both `TBitmap` instances and loads each with the same background bitmap. (The reason you use two bitmaps will be discussed in a moment.) It then creates a `TSprite` instance, and sets the direction indicators. Finally `FormCreate()` resizes the form to the background image's size.

The `FormPaint()` method paints the `BackGnd1` to its canvas, and the `FormDestroy()` frees the `TBitmap` and `TSprite` instances.

The `appevMainIdle()` method calls `DrawSprite()`, which moves and draws the sprite on the background. The bulk of the work is done in the `DrawSprite()` method.`appevMainIdle()` will be invoked whenever the application is in an idle state. That is, whenever there are no actions from the user for the application to respond to.

The `DrawSprite()` method repositions the sprite on the background image. A series of steps is required to erase the old sprite on the background and then draw it at its new location while preserving the background colors around the actual sprite image. Additionally, `DrawSprite()` must perform these steps without producing flickering while the sprite is moving.

To accomplish this, drawing is performed on the offscreen bitmap, `BackGnd2`. `BackGnd2` and `BackGnd1` are exact copies of the background image. However, `BackGnd1` is never modified, so it's a clean copy of the background. When drawing is complete on `BackGnd2`, the modified area of `BackGnd2` is copied to the form's canvas. This allows for only one `BitBlt()` operation to the form's canvas to both erase and draw the sprite at its new location. The drawing operations performed on `BackGnd2` are as follows.

First, a rectangular region is copied from `BackGnd1` to `BackGnd2` over the area occupied by the sprite. This effectively erases the sprite from `BackGnd2`. Then the `FAndImage` bitmap is copied to `BackGnd2` at its new location using the bitwise `AND` operation. This effectively creates a black hole in `BackGnd2` where the sprite exists and still preserves the colors on `BackGnd2` surrounding the sprite. Figure 8.20 shows `FAndImage`.

In Figure 8.20, the sprite is represented by black pixels, and the rest of the image surrounding it consists of white pixels. The color black has a value of `0`, and the color white has the value of `1`. Tables 8.5, and 8.6 show the results of performing the `AND` operation with black and white colors.

TABLE 8.5 `AND` Operation with Black Color

Background	*Value*	*Color*
`BackGnd2`	`1001`	Some color
`FAndImage`	`0000`	Black
Result	`0000`	Black

TABLE 8.6 `AND` Operation with White Color<$AND operation; with white color>

Background	*Value*	*Color*
`BackGnd2`	`1001`	Some color
`FAndImage`	`1111`	White
Result	`1001`	Some color

These tables show how performing the `AND` operation results in blacking out the area where the sprite exists on `BackGnd2`. In Table 8.5, `Value` represents a pixel color. If a pixel on `BackGnd2` contains some arbitrary color, combining this color with the color black using the `AND` operator results in that pixel becoming black. Combining this color with the color white using the `AND` operator results in the color being the same as the arbitrary color, as shown in Table 8.6. Because the color surrounding the sprite in `FAndImage` is white, the pixels on `BackGnd2` where this portion of `FAndImage` is copied retain their colors.

After copying `FAndImage` to `BackGnd2`, `FOrImage` must be copied to the same location on `BackGnd2` to fill in the black hole created by `FAndImage` with the actual sprite colors. `FOrImage` also has a rectangle surrounding the actual sprite image. Again, you're faced with getting the sprite colors to `BackGnd2` while preserving `BackGnd2`'s colors surrounding the sprite. This is accomplished by combining `FOrImage` with `BackGnd2` using the `OR` operation. `FOrImage` is shown in Figure 8.21.

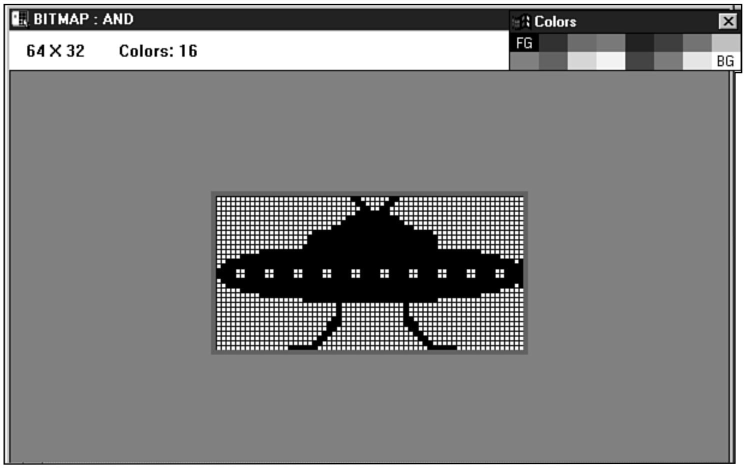


FIGURE 8.20

`FAndImage` *for a sprite.*

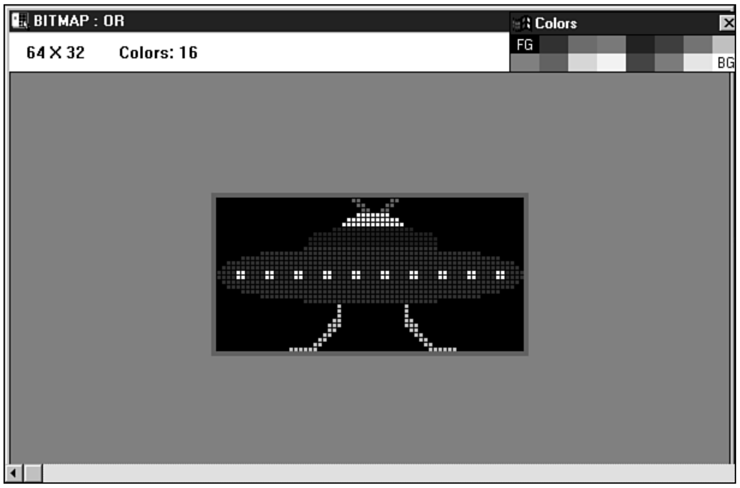


FIGURE 8.21

`FOrImage`.

Notice that the area surrounding the sprite image is black. Table 8.7 shows the results of performing the `OR` operation on `FOrImage` and `BackGnd2`.

TABLE 8.7 An `OR` Operation with the Color Black

Background	*Value*	*Color*
`BackGnd2`	`1001`	Some color
`FAndImage`	`0000`	Black
Result	`1001`	Black

Table 8.7 shows that if `BackGnd2` contains an arbitrary color, using the `OR` operation to combine it with black will result in `BackGnd2`'s color.

Recall that all this drawing is performed on the offscreen bitmap. When the drawing is complete, a single `BitBlt()` is made to the form's canvas to erase and copy the sprite.

The technique shown here is a fairly common method for performing animation. You might consider extending the functionality of the `TSprite` class to move and draw itself on a parent canvas.

Advanced Fonts

Although the VCL enables you to manipulate fonts with relative ease, it doesn't provide the vast font-rendering capabilities provided by the Win32 API. This section gives you a background on Win32 fonts and shows you how to manipulate them.

Types of Win32 Fonts

There are basically two types of fonts in Win32: GDI fonts and device fonts. GDI fonts are stored in font resource files and have an extension of `.fon` (for raster and vector fonts) or `.tot` and `.ttf` (for TrueType fonts). Device fonts are specific to a particular device, such as a printer. Unlike with the GDI fonts, when Win32 uses a device font for printing text, it only needs to send the ASCII character to the device, and the device takes care of printing the character in the specified font. Otherwise, Win32 converts the font to a bitmap or performs the GDI function to draw the font. Drawing the font using bitmaps or GDI functions generally takes longer, as is the case with GDI fonts. Although device fonts are faster, they are device-specific and often very limiting in what fonts a particular device supports.

Basic Font Elements

Before you learn how to use the various fonts in Win32, you should know the various terms and elements associated with Win32 fonts.

A Font's Typeface, Family, and Measurements

Think of a font as just a picture or *glyph* representing a character. Each character has two characteristics: a typeface and a size.

In Win32, a font's *typeface* refers to the font's style and its size. Probably the best definition of typeface and how it relates to a font is in the Win32 help file. This definition says, "A typeface is a collection of characters that share design characteristics; for example, Courier is a common typeface. A font is a collection of characters that have the same typeface and size."

Win32 categorizes these different typefaces into five font families: Decorative, Modern, Roman, Script, and Swiss. The distinguishing font features in these families are the font's serifs and stroke widths.

A *serif* is a small line at the beginning or end of a font's main strokes that give the font a finished appearance. A *stroke* is the primary line that makes up the font. Figure 8.22 illustrates these two features.

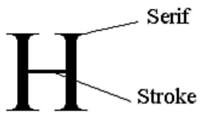


FIGURE 8.22
Serifs and strokes.

Some of the typical fonts you'll find in the different font families are listed in Table 8.8.

TABLE 8.8 Font Families and Typical Fonts

Font Family	*Typical Fonts*
Decorative	Novelty fonts: Old English
Modern	Fonts with constant strike widths that may or may not have serifs: Pica, Elite, and Courier New
Roman	Fonts with variable stroke widths and serifs: Times New Roman and New Century SchoolBook
Script	Fonts that look like handwriting: Script and Cursive
Swiss	Fonts with variable stroke widths without serifs: Arial and Helvetica

A font's size is represented in points (a *point* is 1⁄72 of an inch). A font's height consists of its *ascender* and *descender*. The ascender and descender are represented by the `tmAscent` and `tmDescent` values as shown in Figure 8.23. Figure 8.23 shows other values essential to the character measurement as well.

Characters reside in a character *cell*, an area surrounding the character that consists of white space. When referring to character measurements, keep in mind that the measurement may include both the character glyph (the character's visible portion) and the character cell. Others may refer to only one or the other.

Table 8.9 explains the meaning of the various character measurements.

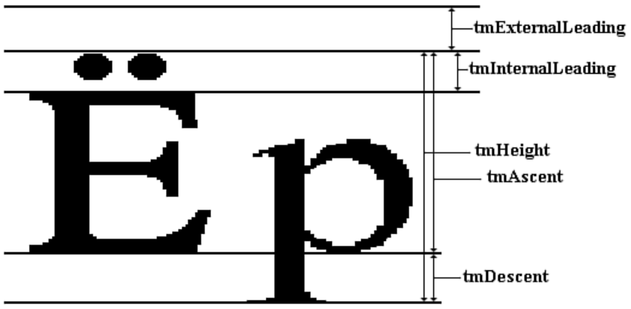


FIGURE 8.23
Character measurement values.

TABLE 8.9 Character Measurements

Measurement	*Meaning*
External leading	The space between text lines
Internal leading	The difference between the character's glyph height and the font's cell height
Ascent	Measurement from the baseline to the top of the character cell
Descent	Measurement from the baseline to the bottom of the character cell
Point size	The character height minus `tmInternalLeading`
Height	The sum of ascent, descent, and internal leading
Baseline	The line on which characters sit

GDI Font Categories

There are essentially three separate categories of GDI fonts: raster fonts, vector fonts (also referred to as *stroke fonts*), and TrueType fonts. The first two existed in older versions of Win32, whereas the latter was introduced in Windows 3.1.

Raster Fonts Explained

Raster fonts are basically bitmaps provided for a specific resolution or *aspect ratio* (ratio of the pixel height and width of a given device) and font size. Because these fonts are provided in specific sizes, Win32 can synthesize the font to generate a new font in the requested size, but it can do so only to produce a larger font from a smaller font. The reverse is not possible because the technique Win32 uses to synthesize the fonts is to duplicate the rows and columns that make up the original font bitmap. Raster fonts are convenient when the size requested is available. They're fast to display and look good when used at the intended size. The disadvantage is that they tend to look a bit sloppy when scaled to larger sizes, as shown in Figure 8.24, which displays the Win32 System font.

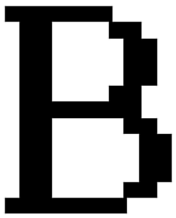


FIGURE 8.24

A raster font.

Vector Fonts Explained

Vector fonts are generated by Win32 with a series of lines created by GDI functions as opposed to bitmaps. These fonts offer better scalability than do raster fonts, but they have a much lower density when displayed, which may or may not be desired. Also, the performance of vector fonts is slow compared to raster fonts. Vector fonts lend themselves best to use with plotters but aren't recommended for designing appealing user interfaces. Figure 8.25 shows a typical vector font.

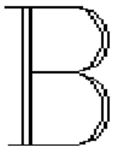


FIGURE 8.25

A vector font.

TrueType Fonts Explained

TrueType fonts are probably the most preferred of the three font types. The advantage to using TrueType fonts is that they can represent virtually any style of font in any size and look pleasing to the eye. Win32 displays TrueType fonts by using a collection of points and hints that describe the font outline. *Hints* are simply algorithms to distort a scaled font's outline to improve its appearance at different resolutions. Figure 8.26 shows a TrueType font.

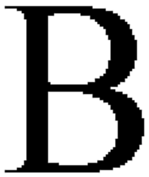


FIGURE 8.26

A TrueType font.

Displaying Different Font Types

So far, we’ve given you the general concepts surrounding Window’s font technology. If you’re interested in getting down to the many nuts and bolts of fonts, take a look at the Win32 online help file on “Fonts Overview,” which provides you with a vast amount of information on the topic. Now, you’ll learn how to use the Win32 API and Win32 structures to create and display fonts of any shape and size.

A Font-Creation Sample Project

The example to follow illustrates the process of instantiating different font types in Windows. The project also illustrates how to obtain information about a rendered font. This project is located in on the CD as `MakeFont.dpr`.

How the Project Works

Through the main form, you select various font attributes to be used in creating the font. The font then gets drawn to a `TPaintBox` component whenever you change the value of one of the font’s attributes. (All components are attached to the `FontChanged()` event handler through their `OnChange` or `OnClick` events.) You also can view information about a font by clicking the Font Information button. Figure 8.27 shows the main form for this project. Listing 8.9 shows the unit defining the main form.

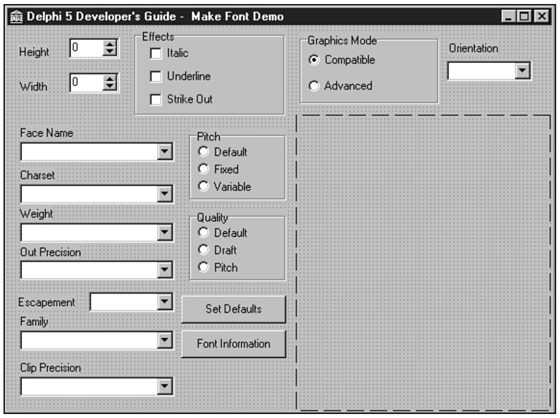


FIGURE 8.27

The main form for the font-creation project.

LISTING 8.9 The Font-Creation Project

```
unit MainFrm;

interface
```

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask, Spin;

const

// Array to represent the TLOGFONT.lfCharSet values
CharSetArray: array[0..4] of byte = (ANSI_CHARSET, DEFAULT_CHARSET,
  SYMBOL_CHARSET, SHIFTJIS_CHARSET, OEM_CHARSET);

// Array to represent the TLOGFONT.lfWeight values
WeightArray: array[0..9] of integer =
 (FW_DONTCARE, FW_THIN, FW_EXTRALIGHT, FW_LIGHT, FW_NORMAL, FW_MEDIUM,
  FW_SEMIBOLD, FW_BOLD, FW_EXTRABOLD, FW_HEAVY);

// Array to represent the TLOGFONT.lfOutPrecision values
OutPrecArray: array[0..7] of byte = (OUT_DEFAULT_PRECIS,
  OUT_STRING_PRECIS, OUT_CHARACTER_PRECIS, OUT_STROKE_PRECIS,
  OUT_TT_PRECIS, OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS,
  OUT_TT_ONLY_PRECIS);

// Array to represent the TLOGFONT.lfPitchAndFamily higher four-bit
// values
FamilyArray: array[0..5] of byte = (FF_DONTCARE, FF_ROMAN,
  FF_SWISS, FF_MODERN, FF_SCRIPT, FF_DECORATIVE);

// Array to represent the TLOGFONT.lfPitchAndFamily lower two-bit values
PitchArray: array[0..2] of byte = (DEFAULT_PITCH, FIXED_PITCH,
   VARIABLE_PITCH);

// Array to represent the TLOGFONT.lfClipPrecision values
ClipPrecArray: array[0..6] of byte = (CLIP_DEFAULT_PRECIS,
  CLIP_CHARACTER_PRECIS, CLIP_STROKE_PRECIS, CLIP_MASK, CLIP_LH_ANGLES,
  CLIP_TT_ALWAYS, CLIP_EMBEDDED);

// Array to represent the TLOGFONT.lfQuality values
QualityArray: array[0..2] of byte = (DEFAULT_QUALITY, DRAFT_QUALITY,
  PROOF_QUALITY);

type

  TMainForm = class(TForm)
    lblHeight: TLabel;
    lblWidth: TLabel;
    gbEffects: TGroupBox;
    cbxItalic: TCheckBox;
```

continues

LISTING 8.9 Continued

```
    cbxUnderline: TCheckBox;
    cbxStrikeOut: TCheckBox;
    cbWeight: TComboBox;
    lblWeight: TLabel;
    lblEscapement: TLabel;
    cbEscapement: TComboBox;
    pbxFont: TPaintBox;
    cbCharSet: TComboBox;
    lblCharSet: TLabel;
    cbOutPrec: TComboBox;
    lblOutPrecision: TLabel;
    cbFontFace: TComboBox;
    rgPitch: TRadioGroup;
    cbFamily: TComboBox;
    lblFamily: TLabel;
    lblClipPrecision: TLabel;
    cbClipPrec: TComboBox;
    rgQuality: TRadioGroup;
    btnSetDefaults: TButton;
    btnFontInfo: TButton;
    lblFaceName: TLabel;
    rgGraphicsMode: TRadioGroup;
    lblOrientation: TLabel;
    cbOrientation: TComboBox;
    seHeight: TSpinEdit;
    seWidth: TSpinEdit;
    procedure pbxFontPaint(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure btnFontInfoClick(Sender: TObject);
    procedure btnSetDefaultsClick(Sender: TObject);
    procedure rgGraphicsModeClick(Sender: TObject);
    procedure cbEscapementChange(Sender: TObject);
    procedure FontChanged(Sender: TObject);
  private
    { Private declarations }
    FLogFont: TLogFont;
    FHFont:   HFont;
    procedure MakeFont;
    procedure SetDefaults;
  public
    { Public declarations }
  end;

var
```

```
  MainForm: TMainForm;

implementation
uses FontInfoFrm;

{$R *.DFM}

procedure TMainForm.MakeFont;
begin
  // Clear the contents of FLogFont
  FillChar(FLogFont, sizeof(TLogFont), 0);
  // Set the TLOGFONT's fields
  with FLogFont do
  begin
    lfHeight         := StrToInt(seHeight.Text);
    lfWidth          := StrToInt(seWidth.Text);
    lfEscapement     :=
      StrToInt(cbEscapement.Items[cbEscapement.ItemIndex]);
    lfOrientation    :=
      StrToInt(cbOrientation.Items[cbOrientation.ItemIndex]);
    lfWeight         := WeightArray[cbWeight.ItemIndex];
    lfItalic         := ord(cbxItalic.Checked);
    lfUnderline      := ord(cbxUnderLine.Checked);
    lfStrikeOut      := ord(cbxStrikeOut.Checked);
    lfCharSet        := CharSetArray[cbCharset.ItemIndex];
    lfOutPrecision   := OutPrecArray[cbOutPrec.ItemIndex];
    lfClipPrecision  := ClipPrecArray[cbClipPrec.ItemIndex];
    lfQuality        := QualityArray[rgQuality.ItemIndex];
    lfPitchAndFamily := PitchArray[rgPitch.ItemIndex] or
       FamilyArray[cbFamily.ItemIndex];
    StrPCopy(lfFaceName, cbFontFace.Items[cbFontFace.ItemIndex]);
  end;
  // Retrieve the requested font
  FHFont := CreateFontIndirect(FLogFont);
  // Assign to the Font.Handle
  pbxFont.Font.Handle := FHFont;
  pbxFont.Refresh;
end;

procedure TMainForm.SetDefaults;
begin
  // Set the various controls to default values for ALogFont
  seHeight.Text           := '0';
  seWidth.Text            := '0';
  cbxItalic.Checked       := false;
  cbxStrikeOut.Checked    := false;
```

continues

LISTING 8.9 Continued

```
  cbxUnderline.Checked    := false;
  cbWeight.ItemIndex      := 0;
  cbEscapement.ItemIndex  := 0;
  cbOrientation.ItemIndex := 0;
  cbCharset.ItemIndex     := 1;
  cbOutPrec.Itemindex     := 0;
  cbFamily.ItemIndex      := 0;
  cbClipPrec.ItemIndex    := 0;
  rgPitch.ItemIndex       := 0;
  rgQuality.ItemIndex     := 0;
  // Fill CBFontFace TComboBox with the screen's fonts
  cbFontFace.Items.Assign(Screen.Fonts);
  cbFontFace.ItemIndex := cbFontFace.Items.IndexOf(Font.Name);
end;

procedure TMainForm.pbxFontPaint(Sender: TObject);
begin
  with pbxFont do
  begin
    { Note that in Windows 95, the graphics mode will always
      be GM_COMPATIBLE as GM_ADVANCED is recognized only by Windows NT.}
    case rgGraphicsMode.ItemIndex of
      0: SetGraphicsMode(pbxFont.Canvas.Handle, GM_COMPATIBLE);
      1: SetGraphicsMode(pbxFont.Canvas.Handle, GM_ADVANCED);
    end;
    Canvas.Rectangle(2, 2, Width-2, Height-2);
    // Write the fonts name
    Canvas.TextOut(Width div 2, Height div 2, CBFontFace.Text);
  end;
end;

procedure TMainForm.FormActivate(Sender: TObject);
begin
  SetDefaults;
  MakeFont;
end;

procedure TMainForm.btnFontInfoClick(Sender: TObject);
begin
  FontInfoForm.ShowModal;
end;

procedure TMainForm.btnSetDefaultsClick(Sender: TObject);
begin
  SetDefaults;
  MakeFont;
end;

procedure TMainForm.rgGraphicsModeClick(Sender: TObject);
begin
  cbOrientation.Enabled := rgGraphicsMode.ItemIndex = 1;
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
  MakeFont;
end;

procedure TMainForm.cbEscapementChange(Sender: TObject);
begin
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
end;

procedure TMainForm.FontChanged(Sender: TObject);
begin
  MakeFont;
end;

end.
```

In `MAINFORM.PAS`, you'll see several array definitions that will be explained shortly. For now, notice that the form has two private variables: `FLogFont` and `FHFont`. `FLogFont` is of type `TLOGFONT`, a record structure used to describe the font to create. `FHFont` is the handle to the font that gets created. The private method `MakeFont()` is where you create the font by first filling the `FLogFont` structure with values specified from the main form's components and then passing that structure to `CreateFontIndirect()`, a Win32 GDI function that returns a font handle to the new font. Before you go on, however, you need to understand the `TLOGFONT` structure.

The TLOGFONT Structure

As stated earlier, you use the `TLOGFONT` structure to define the font you want to create. This structure is defined in the `WINDOWS` unit as follows:

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..lf_FaceSize - 1] of Char;
end;
```

You place values in the TLOGFONT's fields that specify the attributes you want your font to have. Each field represents a different type of attribute. By default, most of the fields can be set to zero, which is what the Set Defaults button on the main form does. In this instance, Win32 chooses the attributes for the font and returns whatever it pleases. The general rule is this: The more fields you fill in, the more you can fine-tune your font style. The following list explains what each TLOGFONT field represents. Some of the fields may be assigned constant values that are predefined in the WINDOWS unit. Refer to Win32 help for a detailed description of these values; we show you only the most commonly used ones here:

Field Value	*Description*
`lfHeight`	The font height. A value greater than zero indicates a cell height. A value less than zero indicates the glyph height (the cell height minus the internal leading). Set this field to zero to let Win32 decide a height for you.
`lfWidth`	The average font width. Set this field to zero to let Win32 choose a font width for you.
`lfEscapement`	The angle (in tenths of degrees) of rotation of the font's baseline (the line along which characters are drawn). In Windows 95/98, the text string and individual characters are drawn using the same angle. That is, `lfEscapement` and `lfOrientation` are the same. In Windows NT, text is drawn independently of the orientation angle of each character in the text string. To achieve the latter, you must set the graphics mode for the device to `GM_ADVANCED` using the `SetGraphicsMode()` Win32 GDI function. By default, the graphics mode is `GM_COMPATIBLE`, which makes the Windows NT behavior like Windows 95. This font-rotation effect is only available with TrueType fonts.
`lfOrientation`	Enables you to specify an angle at which to draw individual characters. In Windows 95/98, this has the same value as `lfEscapement`. In Windows NT, the values may be different. (See `lfEscapement`.)

`lfWeight`	This affects the font density. The `WINDOWS` unit defines several constants for this field, such as `FW_BOLD` and `FW_NORMAL`. Set this field to `FW_DONTCARE` to let Win32 choose a weight for you.
`lfItalic`	Nonzero means *italic*; zero means *nonitalic*.
`lfUnderline`	Nonzero means *underlined*; zero means *not underlined*.
`lfStrikeOut`	Nonzero means that a line gets drawn through the font, whereas a value of zero does not draw a line through the font.
`lfCharSet`	Win32 defines the character sets: `ANSI_CHARSET=0`, `DEFAULT_CHARSET=1`, `SYMBOL_CHARSET=2`, `SHIFTJIS_CHARSET=128`, and `OEM_CHARSET = 255`. Use the `DEFAULT_CHARSET` by default.
`lfOutPrecision`	Specifies how Win32 should match the requested font's size and characteristics to an actual font. Use `TT_ONLY_PRECIS` to specify only TrueType fonts. Other types are defined in the `WINDOWS` unit.
`lfClipPrecision`	Specifies how Win32 clips characters outside a clipping region. Use `CLIP_DEFAULT_PRECIS` to let Win32 choose.
`lfQuality`	Defines the font's output quality as GDI will draw it. Use `DEFAULT_QUALITY` to let Win32 decide, or you may specify `PROOF_QUALITY` or `DRAFT_QUALITY`.
`lfPitchAndFamily`	Defines the font's pitch in the two low-order bits. Specifies the family in the higher four high-order bits. Table 8.8 displays these families.
`lfFaceName`	The typeface name of the font.

The `MakeFont()` procedure uses the values defined in the `constant` section of `MainForm.pas`. These array constants contain the various predefined constant values for the `TLOGFONT` structure. These values are placed in the same order as the choices in the main form's `TComboBox` components. For example, the choices for the font family in the `CBFamily` combo box are in the same order as the values in `FamilyArray`. We used this technique to reduce the code required to fill in the `TLOGFONT` structure. The first line in the `MakeFont()` function

```
fillChar(FLogFont, sizeof(TLogFont), 0);
```

clears the `FLogFont` structure before any values are set. When `FLogFont` has been set, the line

```
FHFont := CreateFontIndirect(FLogFont);
```

calls the Win32 API function `CreateFontIndirect()`, which accepts the `TLOGFONT` structure as a parameter and returns a handle to the requested font. This handle is then set to the `TPaintBox.Font`'s handle property. Delphi 5 takes care of destroying the `TPaintBox`'s previous font before making the assignment. After the assignment is made, you redraw `pbxFont` by calling its `Refresh()` method.

The `SetDefaults()` method initializes the `TLOGFONT` structure with default values. This method is called when the main form is created and whenever the user clicks the Set Defaults button. Experiment with the project to see the different effects you can get with fonts, as shown in Figure 8.28.

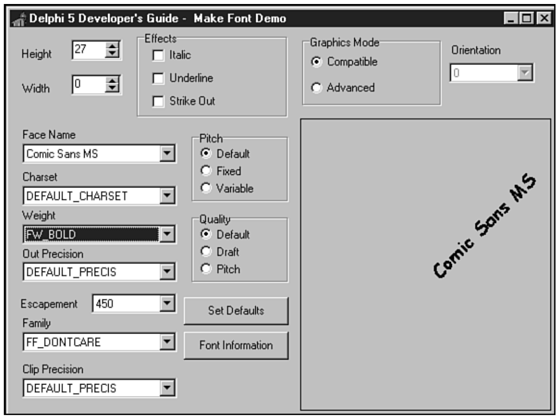


FIGURE 8.28

A rotated font.

Displaying Information About Fonts

The main form's Font Information button invokes the form `FontInfoForm`, which displays information about the selected font. When you specify font attributes in the `TLOGFONT` structure, Win32 attempts to provide you with a font that best resembles your requested font. It's entirely possible that the font you get back from the `CreateFontIndirect()` function has completely different attributes than what you requested. `FontInfoForm` lets you inspect your selected font's attributes. It uses the Win32 API function `GetTextMetrics()` to retrieve the font information.

`GetTextMetrics()` takes two parameters: the handle to the device context whose font you want to examine and a reference to another Win32 structure, `TTEXTMETRIC`. `GetTextMetrics()` then updates the `TTEXTMETRIC` structure with information about the given font. The `WINDOWS` unit defines the `TTEXTMETRIC` record as follows:

```
TTextMetric = record
   tmHeight: Integer;
   tmAscent: Integer;
   tmDescent: Integer;
   tmInternalLeading: Integer;
   tmExternalLeading: Integer;
   tmAveCharWidth: Integer;
   tmMaxCharWidth: Integer;
   tmWeight: Integer;
   tmItalic: Byte;
   tmUnderlined: Byte;
   tmStruckOut: Byte;
   tmFirstChar: Byte;
   tmLastChar: Byte;
   tmDefaultChar: Byte;
   tmBreakChar: Byte;
   tmPitchAndFamily: Byte;
   tmCharSet: Byte;
   tmOverhang: Integer;
   tmDigitizedAspectX: Integer;
   tmDigitizedAspectY: Integer;
end;
```

The `TTEXTMETRIC` record's fields contain much of the same information we've already discussed about fonts. For example, it shows a font's height, average character width, and whether the font is underlined, italicized, struck out, and so on. Refer to the Win32 API online help for detailed information on the `TTEXTMETRIC` structure. Listing 8.10 shows the code for the font information form.

LISTING 8.10 Source to the Font Information Form

```
unit FontInfoFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;

type

  TFontInfoForm = class(TForm)
    lbFontInfo: TListBox;
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

continues

LISTING 8.10 Continued

```
var
  FontInfoForm: TFontInfoForm;

implementation
uses MainFrm;

{$R *.DFM}

procedure TFontInfoForm.FormActivate(Sender: TObject);
const
  PITCH_MASK: byte = $0F;  // Set the lower order four bits
  FAMILY_MASK: byte = $F0; // Set to higher order four bits
var
  TxMetric: TTextMetric;
  FaceName: String;
  PitchTest, FamilyTest: byte;
begin

  // Allocate memory for FaceName string
  SetLength(FaceName, lf_FaceSize+1);

  // First get the font information
  with MainForm.pbxFont.Canvas do
  begin
    GetTextFace(Handle, lf_faceSize-1, PChar(FaceName));
    GetTextMetrics(Handle, TxMetric);
  end;

  // Now add the font information to the listbox from
  // the TTEXTMETRIC structure.
  with lbFontInfo.Items, TxMetric do
  begin
    Clear;
    Add('Font face name:     '+FaceName);
    Add('tmHeight:     '+IntToStr(tmHeight));
    Add('tmAscent:     '+IntToStr(tmAscent));
    Add('tmDescent:     '+IntToStr(tmDescent));
    Add('tmInternalLeading:     '+IntToStr(tmInternalLeading));
    Add('tmExternalLeading:     '+IntToStr(tmExternalLeading));
    Add('tmAveCharWidth:     '+IntToStr(tmAveCharWidth));
    Add('tmMaxCharWidth:     '+IntToStr(tmMaxCharWidth));
    Add('tmWeight:     '+IntToStr(tmWeight));

    if tmItalic <> 0  then
      Add('tmItalic: YES')
    else
      Add('tmItalic: NO');

    if tmUnderlined <> 0 then
      Add('tmUnderlined: YES')
    else
      Add('tmUnderlined: NO');

    if tmStruckOut <> 0 then
      Add('tmStruckOut: YES')
    else
      Add('tmStruckOut: NO');

    // Check the font's pitch type
    PitchTest := tmPitchAndFamily and PITCH_MASK;
    if (PitchTest and TMPF_FIXED_PITCH) = TMPF_FIXED_PITCH then
      Add('tmPitchAndFamily-Pitch: Fixed Pitch');
    if (PitchTest and TMPF_VECTOR) = TMPF_VECTOR then
      Add('tmPitchAndFamily-Pitch: Vector');
    if (PitchTest and TMPF_TRUETYPE) = TMPF_TRUETYPE then
      Add('tmPitchAndFamily-Pitch: True type');
    if (PitchTest and TMPF_DEVICE) = TMPF_DEVICE then
      Add('tmPitchAndFamily-Pitch: Device');
    if PitchTest = 0 then
      Add('tmPitchAndFamily-Pitch: Unknown');

    // Check the fonts family type
    FamilyTest := tmPitchAndFamily and FAMILY_MASK;
    if (FamilyTest and FF_ROMAN) = FF_ROMAN then
      Add('tmPitchAndFamily-Family: FF_ROMAN');
    if (FamilyTest and FF_SWISS) = FF_SWISS then
      Add('tmPitchAndFamily-Family: FF_SWISS');
    if (FamilyTest and FF_MODERN) = FF_MODERN then
      Add('tmPitchAndFamily-Family: FF_MODERN');
    if (FamilyTest and FF_SCRIPT) = FF_SCRIPT then
      Add('tmPitchAndFamily-Family: FF_SCRIPT');
    if (FamilyTest and FF_DECORATIVE) = FF_DECORATIVE then
      Add('tmPitchAndFamily-Family: FF_DECORATIVE');
```

continues

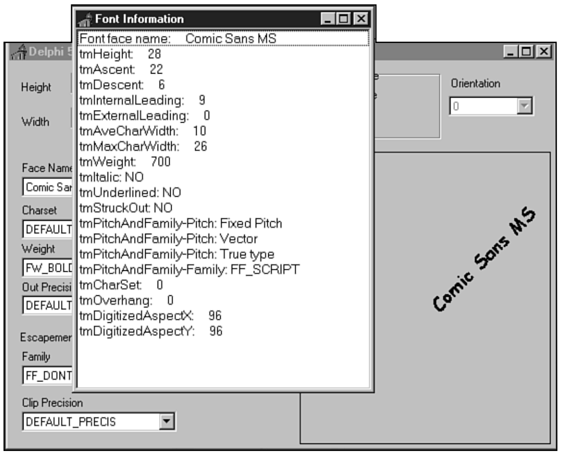
LISTING 8.10 Continued

```
    if FamilyTest = 0 then
      Add('tmPitchAndFamily-Family: Unknown');

    Add('tmCharSet:     '+IntToStr(tmCharSet));
    Add('tmOverhang:     '+IntToStr(tmOverhang));
    Add('tmDigitizedAspectX:     '+IntToStr(tmDigitizedAspectX));
    Add('tmDigitizedAspectY:     '+IntToStr(tmDigitizedAspectY));
  end;
end;

end.
```

The `FormActive()` method first retrieves the font's name with the Win32 API function `GetTextFace()`, which takes a device context, a buffer size, and a null-terminated character buffer as parameters. `FormActivate()` then uses `GetTextMetrics()` to fill `TxMetric`, a `TTEXT-METRIC` record structure, for the selected font. The event handler then adds the values in `TxMetric` to the list box as strings. For the `tmPitchAndFamily` value, you mask out the high- or low-order bit, depending on the value you're testing for, and add the appropriate values to the list box. Figure 8.29 shows `FontInfoForm` displaying information about a font.



```
Font face name:    Comic Sans MS
tmHeight:   28
tmAscent:   22
tmDescent:   6
tmInternalLeading:   9
tmExternalLeading:   0
tmAveCharWidth:   10
tmMaxCharWidth:   26
tmWeight:   700
tmItalic: NO
tmUnderlined: NO
tmStruckOut: NO
tmPitchAndFamily-Pitch: Fixed Pitch
tmPitchAndFamily-Pitch: Vector
tmPitchAndFamily-Pitch: True type
tmPitchAndFamily-Family: FF_SCRIPT
tmCharSet:   0
tmOverhang:   0
tmDigitizedAspectX:   96
tmDigitizedAspectY:   96
```

FIGURE 8.29

The font information form.

Summary

This chapter presented you with a lot of information about the Win32 Graphics Device Interface. We discussed Delphi 5's `TCanvas`, its properties, and its drawing methods. We also discussed Delphi 5's representation of images with its `TImage` component as well as mapping modes and Win32 coordinates systems. You saw how you can use graphics programming techniques to build a paint program and perform simple animation. Finally, we discussed fonts—how to create them and how to display information about them. One of the nice things about the GDI is that working with it can be a lot of fun. Entire books have been written on this subject alone. Take some time to experiment with the drawing routines, create your own fonts, or just fool around with the mapping modes to see what type of effects you can get.