CHAPTER

TWENTY-TWO

22

# Graphics in Delphi

- Painting on a form
- Animated buttons
- An image viewer
- Drawing over a bitmap
- Graphical grids and games
- Using TeeChart
- Windows metafiles

In Chapter 6 of *Mastering Delphi 5*, I introduced the `Canvas` object, Windows painting process, and the `OnPaint` event. In this bonus chapter, I'm going to start from this point and continue covering graphics, following a number of different directions. (For all the code discussed here and in *Mastering Delphi 5*, check the Sybex Web site.)

I'll start with the development of a complex program to demonstrate how the Windows painting model works. Then I'll focus on some graphical components, such as graphical buttons and grids. During this part of the chapter we'll also add some animation to the controls.

Finally, this chapter will discuss the use of bitmaps, covering some advanced features for fast graphics rendering, metafiles, the TeeChart component (including its use on the Web), and few more topics related to the overall issue of graphics.

# Drawing on a Form

In Chapter 6, we saw that it is possible to paint directly on the surface of a form in response to a mouse event. To see this behavior, simply create a new form with the following `OnMouseDown` event handler:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

The program *seems* to work fairly well, but it doesn't. Every click produces a new circle, but if you minimize the form, they'll all go away. Even if you cover a portion of your form with another window, the shapes behind that other form will disappear, and you might end up with partially painted circles.

As I detailed in Chapter 6, this direct drawing is not automatically supported by Windows. The standard approach is to store the painting request in the `OnMouseDown` event and then reproduce the output in the `OnPaint` event. This event, in fact, is called by the system every time the form requires *repainting*. However, you'll need to force its activation by calling the `Invalidate` or `Repaint` methods in the mouse-event handler. In other words, Windows knows when the form has to be repainted because of a system operation (such as placing another window in front of your form), but your program must notify the system when painting is required because of user input or other program operations.

## The Drawing Tools

All the output operations in Windows take place using objects of the `TCanvas` class. The output operations usually don't specify colors and similar elements but use the current drawing tools of the canvas. Here is a list of these drawing tools (or *GDI objects*, from the Graphics Device Interface, which is one of the Windows system libraries):

- The `Brush` property determines the color of the enclosed surfaces. The brush is used to fill closed shapes, such as circles or rectangles. The properties of a brush are `Color`, `Style`, and optionally, `Bitmap`.
- The `Pen` property determines the color and size of the lines and of the borders of the shapes. The properties of a pen are `Color`, `Width`, and `Style`, which includes several dotted and dashed lines (available only if the `Width` is 1 pixel). Another relevant subproperty of the `Pen` is the `Mode` property, which indicates how the color of the pen modifies the color of the drawing surface. The default is simply to use the pen color (with the `pmCopy` style), but it is also possible to merge the two colors in many different ways and to reverse the current color of the drawing surface.
- The `Font` property determines the font used to write text in the form, using the `TextOut` method of the canvas. A font has a `Name`, `Size`, `Style`, `Color`, and so on.

**TIP**

Experienced Windows programmers should note that a Delphi canvas technically represents a Windows device context. The methods of the `TCanvas` class are similar to the GDI functions of the Windows API. You can call extra GDI methods by using the `Handle` property of the canvas, which is a handle of an `HDC` type.

## Colors

Brushes, pens, and fonts (as well as forms and most other components) have a `Color` property. However, to change the color of an element properly, using non-standard colors (such as the color constants in Delphi), you should know how Windows treats the color. In theory, Windows uses 24-bit RGB colors. This means you can use 256 different values for each of the three basic colors (red, green, and blue), obtaining 16 million different shades.

However, you or your users might have a video adapter that cannot display such a variety of colors, although this is increasingly less frequent. In this case, Windows either uses a technique called *dithering*, which basically consists of

using a number of pixels of the available colors to simulate the requested one; or it approximates the color, using the nearest available match. For the color of a brush (and the background color of a form, which is actually based on a brush), Windows uses the dithering technique; for the color of a pen or font, it uses the nearest available color.

In terms of pens, you can read (but not change) the current pen position with the `PenPos` property of the canvas. The pen position determines the starting point of the next line the program will draw, using the `LineTo` method. To change it, you can use the canvas's `MoveTo` method. Other properties of the canvas affect lines and colors, too. Interesting examples are `CopyMode` and `ScaleMode`. Another property you can manipulate directly to change the output is the `Pixels` array, which you can use to access (read) or change (write) the color of any individual point on the surface of the form. As we'll see in the BmpDraw example, per pixel operations are very slow in GDI, compared to line access available through the `ScanLines` property.

Finally, keep in mind that Delphi's `TColor` values do not always match plain RGB values of the native Windows representation (`COLORREF`), because of Delphi color constants. You can always convert a Delphi color to the RGB value using the `ColorToRGB` function. You can find the details of Delphi's representation in the *TColor type* Help entry.

# Drawing Shapes

Now I want to extend the Mouse1 example built at the end of Chapter 6 and turn it into the Shapes application. In this new program I want to use the *store-and-draw* approach with multiple shapes, handle color and pen attributes, and provide a foundation for further extensions.

Because you have to remember the position and the attributes of each shape, you can create an object for each shape you have to store, and you can keep the objects in a list. (To be more precise, the list will store references to the objects, which are allocated in separate memory areas.) I've defined a base class for the shapes and two inherited classes that contain the painting code for the two types of shapes I want to handle, rectangles and ellipses.

The base class has a few properties, which simply read the fields and write the corresponding values with simple methods. Notice that the coordinates can be read using the `Rect` property but must be modified using the four positional properties. The reason is that if you add a `write` portion to the `Rect` property,

you can access the rectangle as a whole but not its specific subproperties. Here are the declarations of the three classes:

```
type
  TBaseShape = class
  private
    FBrushColor: TColor;
    FPenColor: TColor;
    FPenSize: Integer;
    procedure SetBrushColor(const Value: TColor);
    procedure SetPenColor(const Value: TColor);
    procedure SetPenSize(const Value: Integer);
    procedure SetBottom(const Value: Integer);
    procedure SetLeft(const Value: Integer);
    procedure SetRight(const Value: Integer);
    procedure SetTop(const Value: Integer);
  protected
    FRect: TRect;
  public
    procedure Paint (Canvas: TCanvas); virtual;
  published
    property PenSize: Integer read FPenSize write SetPenSize;
    property PenColor: TColor read FPenColor write SetPenColor;
    property BrushColor: TColor read FBrushColor write SetBrushColor;
    property Left: Integer write SetLeft;
    property Right: Integer write SetRight;
    property Top: Integer write SetTop;
    property Bottom: Integer write SetBottom;
    property Rect: TRect read FRect;
  end;

type
  TEllShape = class (TBaseShape)
    procedure Paint (Canvas: TCanvas); override;
  end;

  TRectShape = class (TBaseShape)
    procedure Paint (Canvas: TCanvas); override;
  end;
```

Most of the code in the methods is very simple. The only relevant code is in the three `Paint` procedures:

```
procedure TBaseShape.Paint (Canvas: TCanvas);
begin
  // set the attributes
  Canvas.Pen.Color := fPenColor;
```

```
  Canvas.Pen.Width := fPenSize;
  Canvas.Brush.Color := fBrushColor;
end;

procedure TEllShape.Paint(Canvas: TCanvas);
begin
  inherited Paint (Canvas);
  Canvas.Ellipse (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom)
end;

procedure TRectShape.Paint(Canvas: TCanvas);
begin
  inherited Paint (Canvas);
  Canvas.Rectangle (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom)
end;
```

All of this code is stored in the secondary ShapesH (Shapes Hierarchy) unit. To store a list of shapes, the form has a `TList` object data member, named `ShapesList`, which is initialized in the `OnCreate` event handler and destroyed at the end; the destructor also frees all the objects in the list (in reverse order, to avoid refreshing the internal list data too often):

```
procedure TShapesForm.FormCreate(Sender: TObject);
begin
  ShapesList := TList.Create;
end;

procedure TShapesForm.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  // delete each object
  for I := ShapesList.Count - 1 downto 0 do
    TBaseShape (ShapesList [I]).Free;
  ShapesList.Free;
end;
```

The program adds a new object to the list each time the user starts the dragging operation. Since the object is not completely defined, the form keeps a reference to it in the `CurrShape` field. Notice that the type of object created depends on the status of the mouse keys:

```
procedure TShapesForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

```
begin
  if Button = mbLeft then
  begin
    // activate dragging
    fDragging := True;
    SetCapture (Handle);

    // create the proper object
    if ssShift in Shift then
      CurrShape := TEllShape.Create
    else
      CurrShape := TRectShape.Create;

    // set the style and colors
    CurrShape.PenSize := Canvas.Pen.Width;
    CurrShape.PenColor := Canvas.Pen.Color;
    CurrShape.BrushColor := Canvas.Brush.Color;

    // set the initial position
    CurrShape.Left := X;
    CurrShape.Top := Y;
    CurrShape.Right := X;
    CurrShape.Bottom := Y;
    Canvas.DrawFocusRect (CurrShape.Rect);

    // add to the list
    ShapesList.Add (CurrShape);
  end;
end;
```

During the dragging operation we draw the line corresponding to the shape, as I did in the Mouse1 example:

```
procedure TShapesForm.FormMouseMove(Sender: TObject; Shift:
TShiftState;
  X, Y: Integer);
var
  ARect: TRect;
begin
  // copy the mouse coordinates to the title
  Caption := Format ('Shapes (x=%d, y=%d)', [X, Y]);

  // dragging code
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
```

```
      ARect := NormalizeRect (CurrShape.Rect);
      Canvas.DrawFocusRect (ARect);
      CurrShape.Right := X;
      CurrShape.Bottom := Y;
      ARect := NormalizeRect (CurrShape.Rect);
      Canvas.DrawFocusRect (ARect);
    end;
  end;
```

This time, however, I've also added a fix to the program. In the Mouse1 example, if you move the mouse toward the upper-left corner of the form while dragging, the `DrawFocusRect` call produces no effect. The reason is that the rectangle passed as a parameter to `DrawFocusRect` must have a `Top` value that is less than the `Bottom` value, and the same is true for the `Left` and `Right` values. In other words, a rectangle that extends itself on the negative side doesn't work properly. However, at the end it paints correctly, because the `Rectangle` drawing function doesn't have this problem.

To fix this problem I've written a simple function that inverts the coordinates of a rectangle to make it reflect the requests of the `DrawFocusRect` call:

```
function NormalizeRect (ARect: TRect): TRect;
var
  tmp: Integer;
begin
  if ARect.Bottom < ARect.Top then
  begin
    tmp := ARect.Bottom;
    ARect.Bottom := ARect.Top;
    ARect.Top := tmp;
  end;
  if ARect.Right < ARect.Left then
  begin
    tmp := ARect.Right;
    ARect.Right := ARect.Left;
    ARect.Left := tmp;
  end;
  Result := ARect;
end;
```

Finally, the `OnMouseUp` event handler sets the definitive image size and refreshes the painting of the form. Instead of calling the `Invalidate` method, which would cause all of the images to be repainted with a lot of flickering, the program uses the `InvalidateRect` API function:

```
procedure InvalidateRect(Wnd: HWnd;
  Rect: PRect; Erase: Bool);
```

The three parameters represent the handle of the window (that is, the `Handle` property of the form), the rectangle you want to repaint, and a flag indicating whether or not you want to erase the area before repainting it. This function requires, once more, a *normalized* rectangle. (You can try replacing this call with one to `Invalidate` to see the difference, which is more obvious when you create many forms.) Here is the complete code of the `OnMouseUp` handler:

```
procedure TShapesForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ARect: TRect;
begin
  if fDragging then
  begin
    // end dragging
    ReleaseCapture;
    fDragging := False;

    // set the final size
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
    CurrShape.Right := X;
    CurrShape.Bottom := Y;

    // optimized invalidate code
    ARect := NormalizeRect (CurrShape.Rect);
    InvalidateRect (Handle, @ARect, False);
  end;
end;
```

**NOTE**

When you select a large drawing pen (we'll look at the code for that shortly), the border of the frame is painted partially inside and partially outside the frame, to accommodate the large pen. To allow for this, we should invalidate a frame rectangle that is inflated by half the size of the current pen. You can do this by calling the `InflateRect` function. As an alternative, in the `FormCreate` method I've set the `Style` of the `Pen` of the form `Canvas` to `psInsideFrame`. This causes the drawing function to paint the pen completely inside the frame of the shape.

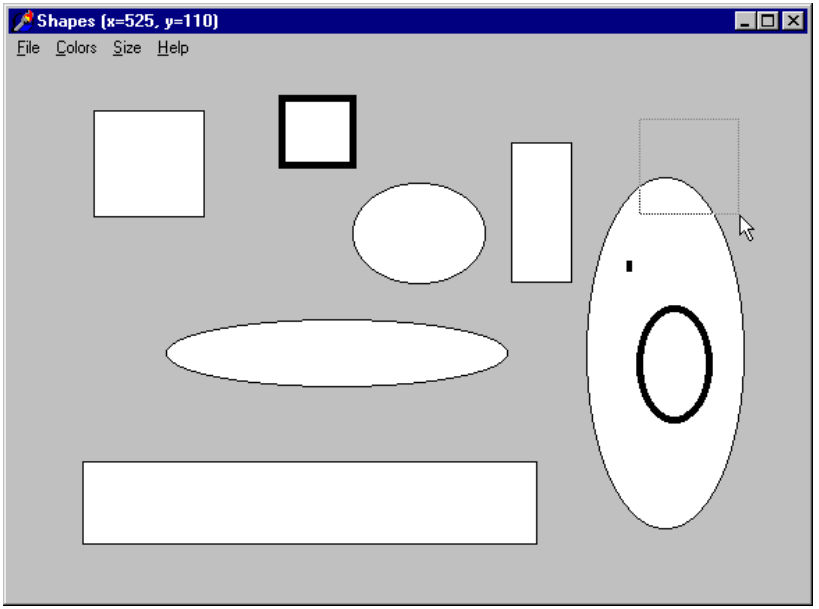
In the method corresponding to the `OnPaint` event, all the shapes currently stored in the list are painted, as you can see in Figure 22.1. Since the painting code affects the properties of the `Canvas`, we need to store the current values and reset them at the end. The reason is that, as I'll show you later in this chapter, the properties of the form's canvas are used to keep track of the attributes selected by the

user, who might have changed them since the last shape was created. Here is the code:

```
procedure TShapesForm.FormPaint(Sender: TObject);
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // store the current Canvas attributes
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repaint each shape of the list
  for I := 0 to ShapesList.Count - 1 do
  begin
    AShape := ShapesList.Items [I];
    AShape.Paint (Canvas);
  end;

  // reset the current Canvas attributes
  Canvas.Pen.Color := OldPenCol;
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

**FIGURE 22.1:**

The Shapes example can be used to draw multiple shapes, which it stores in a list.



The other methods of the form are simple. Three of the menu commands allow us to change the colors of the background, the shape borders (the pen), and the internal area (the brush). These methods use the ColorDialog component and store the result in the properties of the form's canvas. This is an example:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  // select a new color for the pen
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute then
    Canvas.Pen.Color := ColorDialog1.Color;
end;
```

The new colors will affect shapes created in the future but not the existing ones. The same approach is used for the width of the lines (the pen), although this time the program also checks to see whether the value has become too small, disabling the menu item if it has:

```
procedure TShapesForm.DecreasePenSize1Click(Sender: TObject);
begin
  Canvas.Pen.Width := Canvas.Pen.Width - 2;
  if Canvas.Pen.Width < 3 then
    DecreasePenSize1.Enabled := False;
end;
```

To change the colors of the border (the pen) or the surface (the brush) of the shape, I've used the standard Color dialog box. Here is one of the two methods:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute then
   Canvas.Pen.Color := ColorDialog1.Color;
end;
```

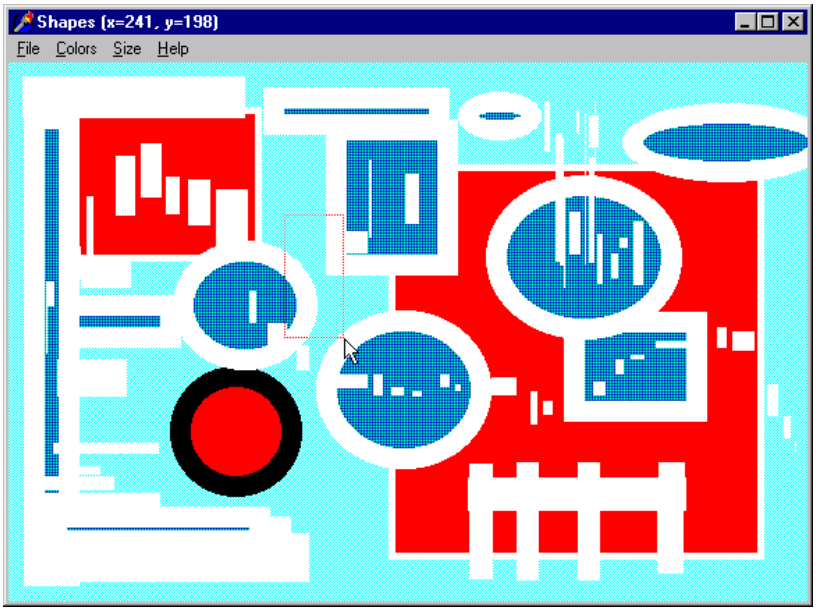
In Figure 22.2 you can see another example of the output of the Shapes program, this time using multiple colors for the shapes and their background. The program asks the user to confirm some operations, such as exiting from the program or removing all the shapes from the list (with the File ➢ New command):

```
procedure TShapesForm.New1Click(Sender: TObject);
begin
  if (ShapesList.Count > 0) and (MessageDlg (
    'Are you sure you want to delete all the shapes?',
    mtConfirmation, [mbYes, mbNo], 0) = idYes) then
  begin
    // delete each object
    for I := ShapesList.Count - 1 downto 0 do
```

```
        TBaseShape (ShapesList [I]).Free;
      ShapesList.Clear;
      Refresh;
    end;
  end;
```

**FIGURE 22.2:**

Changing the colors and the line size of shapes allows you to use the Shapes example to produce any kind of result.




## Printing Shapes

Besides painting the shapes on a form canvas, we can paint them on a printer canvas, effectively printing them! Because it is possible to execute the same methods on a printer canvas as on any other canvas, you might be tempted to add to the program a new method for printing the shapes. This is certainly easy, but an even better option is writing a single output method to use for both the screen and the printer.

As an example of this approach, I've built a new version of the program, called ShapesPr. The interesting point is that I've moved the code of the `FormPaint` example into another method I've defined, called `CommonPaint`. This new method has two parameters, the canvas and a scale factor (which defaults to 1):

```
procedure CommonPaint(Canvas: TCanvas; Scale: Integer = 1);
```

The CommonPaint method outputs the list of shapes to the canvas passed as parameters, using the proper scale factor:

```
procedure TShapesForm.CommonPaint (
  Canvas: TCanvas; Scale: Integer);
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // store the current Canvas attributes
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repaint each shape of the list
  for I := 0 to ShapesList.Count - 1 do
  begin
    AShape := ShapesList.Items [I];
    AShape.Paint (Canvas, Scale);
  end;

  // reset the current Canvas attributes
  Canvas.Pen.Color := OldPenCol;
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

Once you've written this code, the FormPaint and Print1Click methods are simple to implement. To paint the image on the screen, you can call CommonPaint without a scaling factor (so that the default value 1 is used):

```
procedure TShapesForm.FormPaint(Sender: TObject);
begin
  CommonPaint (Canvas);
end;
```

To paint the contents of the form to the printer instead of the form, you can reproduce the output on the printer canvas, using a proper scaling factor. Instead of choosing a scale, I decided to compute it automatically. The idea is to print the shapes on the form as large as possible, by sizing the form's client area so that it takes up the whole page. The code is probably simpler than the description:

```
procedure TShapesForm.Print1Click(Sender: TObject);
var
  Scale, Scale1: Integer;
```

```
begin
  Scale := Printer.PageWidth div ClientWidth;
  Scale1 := Printer.PageHeight div ClientHeight;
  if Scale1 < Scale then
    Scale := Scale1;
  Printer.BeginDoc;
  try
    CommonPaint (Printer.Canvas, Scale);
    Printer.EndDoc;
  except
    Printer.Abort;
    raise;
  end;
end;
```

Of course, you need to remember to call the specific commands to start printing (`BeginDoc`) and commit the output (`EndDoc`) before and after you call the `CommonPaint` method. If an exception is raised, the program calls `Abort` to terminate the printing process anyway.

# Delphi Graphical Components

The Shapes example uses almost no components, aside from a standard color-selection dialog box. As an alternative, we could have used some Delphi components that specifically support graphics:

- You use the PaintBox component when you need to paint on a certain area of a form and that area might move on the form. For example, PaintBox is useful for painting on a dialog box without the risk of mixing the area for the output with the area for the controls. The PaintBox might fit within other controls of a form, such as a toolbar or a status bar, and avoid any confusion or overlapping of the output. In the Shapes example, using this component made no sense, because we always worked on the whole surface of the form.
- You use the Shape component to paint shapes on the screen, exactly as we have done up to now. You could indeed use the Shape component instead of the manual output, but I really wanted to show you how to accomplish some direct output operations. This approach was not much more complex than the one Delphi suggests. Using the Shape component would have been useful to extend the example, allowing a user to drag shapes on the screen, remove them, and work on them in a number of other ways.

- You can use the Image component to display an existing bitmap, possibly loading it from a file, or even to paint on a bitmap, as I'll demonstrate in the next two examples and discuss in the next section.
- If it is included in your version of Delphi, you can use the TeeChart control to create business graphics output, as we'll see toward the end of this chapter.
- You can use the graphical support provided by the bitmap buttons and speed button controls, among others. We'll see later in this chapter how to extend the graphical capabilities of these controls.
- You can use the Animate component to make the graphics more—well, animated. Besides using this component, you can manually create animations by displaying bitmaps in sequence or scrolling them, as we'll see other examples.

As you can see, we have a long way to go to cover Delphi's graphics support from all of its angles.

# Drawing in a Bitmap

I've already mentioned that by using an Image component, you can draw images directly in a bitmap. Instead of drawing on the surface of a window, you draw on a bitmap in memory and then copy the bitmap to the surface of the window. The advantage is that instead of having to repaint the image each time an `OnPaint` event occurs, the component copies the bitmap back to video.

Technically, a `TBitmap` object has its own canvas. By drawing on this canvas, you can change the contents of the bitmap. As an alternative, you can work on the canvas of an Image component connected to the bitmap you want to change. You might consider choosing this approach instead of the typical painting approach if any of the following conditions are true:

- The program has to support freehand drawing or very complex graphics (such as fractal images).
- The program should be very fast in drawing a number of images.
- RAM consumption is not an issue.
- You are a lazy programmer.

The last point is interesting because painting generally requires more code than drawing, although it allows more flexibility. In a graphics program, for example,

if you use painting, you have to store the location and colors of each shape. On the other hand, you can easily change the color of an existing shape or move it. These operations are very difficult with the painting approach and may cause the area behind an image to be lost. If you are working on a complex graphical application, you should probably choose a mix of the two approaches. For casual graphics programmers, the choice between the two approaches involves a typical speed-versus-memory decision: painting requires less memory; storing the bitmap is faster.

## Drawing Shapes

Now let's look at an Image component example that will paint on a bitmap. The idea is simple. I've basically written a simplified version of the Shape example, by placing an Image component on its form and redirecting all the output operations to the canvas of this Image component.

In this example, ShapeBmp, I've also added some new menu items to save the image to a file and to load an existing bitmap. To accomplish this, I've added to the form a couple of default dialog components, OpenDialog and SaveDialog. One of the properties I had to change was the background color of the form. In fact, when you perform the first graphical operation on the image, it creates a bitmap, which has a white background by default. If the form has a gray background, each time the window is repainted, some flickering occurs. For this reason, I've chosen a white background for the form, too.

The code of this example is still quite simple, considering the number of operations and menu commands. The drawing portion is linear and very close to Mouse1, except that the mouse events now relate to the image instead of the form; I've used the `NormalizeRect` function during the dragging; and the program uses the canvas of the image. Here is the `OnMouseMove` event handler, which reintroduces the drawing of points when moving the mouse with the Shift key pressed:

```
procedure TShapesForm.Image1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  ARect: TRect;
begin
  // display the position of the mouse in the caption
  Caption := Format ('ShapeBmp (x=%d, y=%d)', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
```

```
    ARect := NormalizeRect (fRect);
    Canvas.DrawFocusRect (ARect);
    fRect.Right := X;
    fRect.Bottom := Y;
    ARect := NormalizeRect (fRect);
    Canvas.DrawFocusRect (ARect);
  end
  else
    if ssShift in Shift then
      // mark point in red
      Image1.Canvas.Pixels [X, Y] := clRed;
end;
```

Notice that the temporary focus rectangle is painted directly on the form, over the image (and thus not stored in the bitmap). What is different is that at the end of the dragging operation, the program paints the rectangle on the image, storing it in the bitmap. This time the program doesn't call `Invalidate` and has no `OnPaint` event handler:

```
procedure TShapesForm.Image1MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    ReleaseCapture;
    fDragging := False;
    Image1.Canvas.Rectangle (fRect.Left, fRect.Top,
      fRect.Right, fRect.Bottom);
  end;
end;
```

To avoid overly complex file support, I decided to implement the File ➢ Load and File ➢ Save As commands and not handle the Save command, which is generally more complex. I've simply added an `fChanged` field to the form to know when an image has changed, and I've included code that checks this value a number of times (before asking the user to confirm).

The `OnClick` event handler of the File ➢ New menu item calls the `FillArea` method to paint a big white rectangle over the whole bitmap. In this code you can also see how the `Changed` field is used:

```
procedure TShapesForm.New1Click(Sender: TObject);
var
  Area: TRect;
  OldColor: TColor;
```

```
begin
  if not fChanged or (MessageDlg (
    'Are you sure you want to delete the current image?',
    mtConfirmation, [mbYes, mbNo], 0) = idYes) then
  begin
    {repaint the surface, covering the whole area,
    and resetting the old brush}
    Area := Rect (0, 0, Image1.Picture.Width,
      Image1.Picture.Height);
    OldColor := Image1.Canvas.Brush.Color;
    Image1.Canvas.Brush.Color := clWhite;
    Image1.Canvas.FillRect (Area);
    Image1.Canvas.Brush.Color := OldColor;
    fChanged := False;
  end;
end;
```

Of course, the code has to save the original color and restore it later on. A realignment of the colors is also required by the File ➢ Load command-response method. When you load a new bitmap, in fact, the Image component creates a new canvas with the default attributes. For this reason, the program saves the pen's colors and size and copies them later to the new canvas:

```
procedure TShapesForm.Load1Click(Sender: TObject);
var
  PenCol, BrushCol: TColor;
  PenSize: Integer;
begin
  if not fChanged or (MessageDlg (
      'Are you sure you want to delete the current image?',
      mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    if OpenDialog1.Execute then
    begin
      PenCol := Image1.Canvas.Pen.Color;
      BrushCol := Image1.Canvas.Brush.Color;
      PenSize := Image1.Canvas.Pen.Width;
      Image1.Picture.LoadFromFile (OpenDialog1.Filename);
      Image1.Canvas.Pen.Color := PenCol;
      Image1.Canvas.Brush.Color := BrushCol;
      Image1.Canvas.Pen.Width := PenSize;
      fChanged := False;
    end;
end;
```

Saving the current image is much simpler:

```
procedure TShapesForm.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    Image1.Picture.SaveToFile (
      SaveDialog1.Filename);
    fChanged := False;
  end;
end;
```

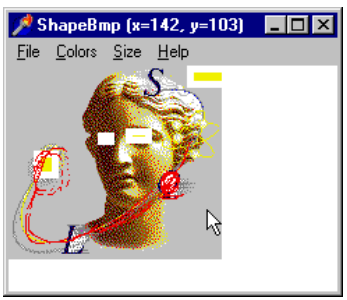
Finally, here is the code of the `OnCloseQuery` event of the form, which uses the `Changed` field:

```
procedure TShapesForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if not fChanged or (MessageDlg (
      'Are you sure you want to delete the current image?',
      mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    CanClose := True
  else
    CanClose := False;
end;
```

ShapeBmp is an interesting program (see Figure 22.3), with limited but working file support. The real problem is that the Image component creates a bitmap of its own size. When you increase the size of the window, the Image component is resized but not the bitmap in memory. Therefore, you cannot draw on the right and bottom areas of the window. There are a number of possible solutions: use the `Constraints` property to set the maximum size of the form, use a fixed border, visually mark the *drawing area* on the screen, and so on. However, I've decided to leave the program as is because it does its job of demonstrating how to draw in a bitmap well enough.

**FIGURE 22.3:**

The ShapeBmp example has limited but working file support: you can load an existing bitmap, draw shapes over it, and save it to disk.



## An Image Viewer

The ShapeBmp program can be used as an image viewer, because you can load any bitmap in it. In general, in the Image control you can load any graphic file type that has been registered with the VCL `TPicture` class. The default file formats are bitmap files (BMP), icon files (ICO), or Windows metafiles (WMF). Bitmap and icon files are well-known formats. Windows metafiles, however, are not so common. They are a collection of graphical commands, similar to a list of GDI function calls that need to be executed to rebuild an image. Metafiles are usually referred to as *vector graphics* and are similar to the graphics file formats used for clip-art libraries. Delphi also ships with JPG support for `TImage`, and third parties have GIF and other file formats covered.

**NOTE**

To produce a Windows metafile, a program should call GDI functions, redirecting their output to the file. In Delphi, you can use a `TMetafileCanvas` and the high-level `TCanvas` methods. Later on, this metafile can be *played* or executed to call the corresponding functions, thus producing a graphic. Metafiles have two main advantages: the limited amount of storage they require compared to other graphical formats, and the device-independence of their output. I'll cover Delphi metafile support later in this chapter.

To build a full-blown image viewer program, ImageV, around the Image component, we only need to create a form with an image that fills the whole client area, a simple menu, and an OpenDialog component:

```
object ViewerForm: TViewerForm
  Caption = 'Image Viewer'
  Menu = MainMenu1
  object Image1: TImage
    Align = alClient
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Open1: TMenuItem...
      object Exit1: TMenuItem...
    object Options1: TMenuItem
      object Stretch1: TMenuItem
      object Center1: TMenuItem
    object Help1: TMenuItem
      object AboutImageViewer1: TMenuItem
  end
  object OpenDialog1: TOpenDialog
    FileEditStyle = fsEdit
```

```
    Filter = 'Bitmap (*.bmp)|*.bmp|
      Icon (*.ico)|*.ico|Metafile (*.wmf)|*.wmf'
    Options = [ofHideReadOnly, ofPathMustExist,
      ofFileMustExist]
  end
end
```

Surprisingly, this application requires very little coding, at least in its first basic version. The File ➢ Exit and Help ➢ About commands are trivial, and the File ➢ Open command has the following code:

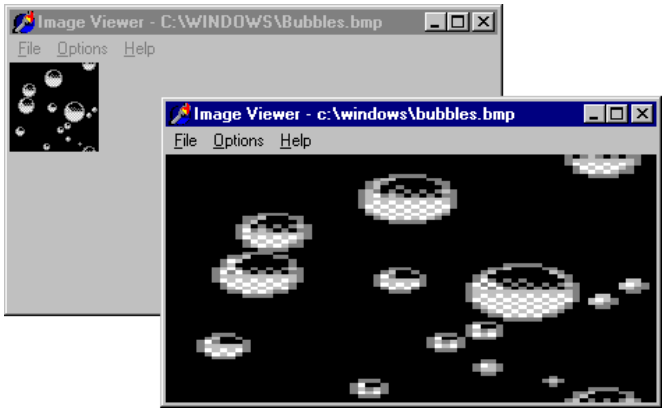
```
procedure TViewerForm.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
    Caption := 'Image Viewer - ' + OpenDialog1.FileName;
  end;
end;
```

The fourth and fifth menu commands, Options ➢ Stretch and Options ➢ Center, simply toggle the component's `Stretch` property (see Figure 22.4 for the result) or `Center` property and add a check mark to themselves. Here is the `OnClick` event handler of the `Stretch1` menu item:

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
end;
```

**FIGURE 22.4:**

Two copies of the ImageV program, which display the regular and stretched versions of the same bitmap



Keep in mind that when stretching an image, you can change its width-to-height ratio, possibly distorting the shape, and that not all images can be properly stretched. Stretching black-and-white or 256-color bitmaps doesn't always work correctly.

Besides this problem, the application has some other drawbacks. If you select a file without one of the standard extensions, the Image component will raise an exception. The exception handler provided by the system behaves as we would expect; the wrong image file is not loaded, and the program can safely continue. Another problem is that if you load a large image, the viewer has no scroll bars. You can maximize the viewer window, but this might not be enough. The Image components do not handle scroll bars automatically, but the form can do it. I'll further extend this example to include scroll bars in the following paragraph.

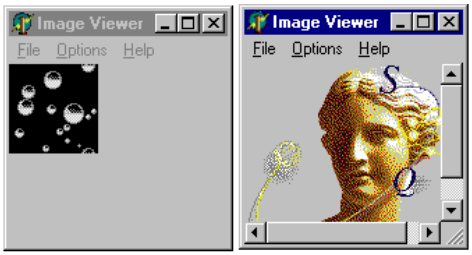
## Scrolling an Image

An advantage of the way automatic scrolling works in Delphi is that if the size of a single big component contained in a form changes, scroll bars are added or removed automatically. A good example is the use of the Image component. If the `AutoSize` property of this component is set to `True` and you load a new picture into it, the component automatically sizes itself, and the form adds or removes the scroll bars as needed.

If you load a large bitmap in the ImageV example, you will notice that part of the bitmap remains hidden. To fix this, you can set the `AutoSize` property of the Image component to `True` and disable its alignment with the client area. You should also set a small initial size for the image. You don't need to make any adjustments when you load a new bitmap, because the size of the Image component is automatically set for you by the system. You can see in Figure 22.5 that scroll bars are actually added to the form. The figure shows two different copies of the program. The difference between the copy of the program on the left and the one on the right is that the first has an image smaller than its client area, so no scroll bars were added. When you load a larger image in the program, two scroll bars will automatically appear, as in the example on the right.

**FIGURE 22.5:**

In the ImageV2 example, the scroll bars are added automatically to the form when the whole bitmap cannot fit into the client area of the form displayed.



Some more coding is required to disable the scroll bars and change the alignment of the image when the Stretch menu command is selected and to restore them when this feature is disabled. Again, we do not act directly on the scroll bars themselves but simply change the alignment of the panel, using its `Stretch` property, and manually calculate the new size, using the size of the picture currently loaded. (This code mimics the effect of the `AutoSize` property, which works only when a new file is loaded.)

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
  if Image1.Stretch then
    Image1.Align := alClient
  else
  begin
   Image1.Align := alNone;
   Image1.Height := Image1.Picture.Height;
   Image1.Width := Image1.Picture.Width;
  end;
end;
```

## Bitmaps to the Max

When the Image control is connected to a bitmap, there are some additional operations you can do, but before we examine them, I have to introduce bitmap formats. There are different types of bitmaps in Windows. Bitmaps can be *device-independent* or not, a term used to indicate whether the bitmap has extra palette management information. BMP files are usually device-independent bitmaps.

Another difference relates to the color depth—that is, the number of different colors the bitmap can use or, in other words, the number of bits required for storing each pixel. In a 1-bit bitmap, each point can be either black or white (to be more precise, 1-bit bitmaps can have a color palette, allowing the bitmap to represent any two colors and not just black and white). An 8-bit bitmap usually has a companion palette to indicate how the 256 different colors map to the actual system colors, a 24-bit bitmap indicates the system color directly. To make things more complex, when the system draws a bitmap on a computer with a different color capability, it has to perform some conversion.

Internally the bitmap format is very simple, whatever the color depth. All the values that make up a line are stored sequentially in a memory block. This is efficient for moving the data from memory to the screen, but it is not an effective way to store information; BMP files are generally very large, and they perform no compression.

**NOTE**

The BMP format actually has a very limited form of compression, known as Run-Length Encoding (RLE), in which subsequent pixels with the same color are replaced by the number of such pixels followed by the color. This can reduce the size of the image, but in some cases it will make it grow. For compressed images in Delphi, you can use the `TJpegImage` class and the support for the JPEG format offered by the `TPicture` class. Actually, all `TPicture` does is to manage a registered list of graphic classes.

The BmpDraw example uses this information about the internal structure of a bitmap and some other technical features to take direct handling of bitmaps to a new level. First, it extends the ImageV example by adding a menu item you can use to display the color depth of the current bitmap, by using the corresponding `PixelFormat` property:

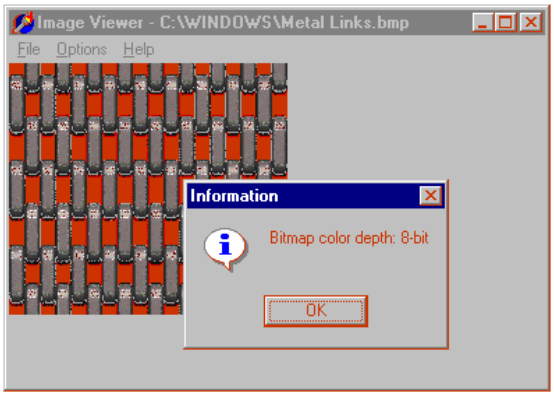
```
procedure TBitmapForm.ColorDepth1Click(Sender: TObject);
var
  strDepth: String;
begin
  case Image1.Picture.Bitmap.PixelFormat of
    pfDevice: strDepth := 'Device';
    pf1bit: strDepth := '1-bit';
    pf4bit: strDepth := '4-bit';
    pf8bit: strDepth := '8-bit';
    pf15bit: strDepth := '15-bit';
    pf16bit: strDepth := '16-bit';
    pf24bit: strDepth := '24-bit';
    pf32bit: strDepth := '32-bit';
    pfCustom: strDepth := 'Custom';
  end;
  MessageDlg ('Bitmap color depth: ' + strDepth,
    mtInformation, [mbOK], 0);
end;
```

You can try loading different bitmaps and see the effect of this method, as shown in Figure 22.6.

What is more interesting is to study how to access the memory image held by the bitmap object. A simple solution is to use the `Pixels` property, as I've done in the ShapeBmp example, to draw the red pixels during the dragging operation. In this program I've added a menu item to create an entire new bitmap pixel by pixel, using a simple mathematical calculation to determine the color. (The same approach can be used, for example, to build fractal images.)

**FIGURE 22.6:**

The color depth of a standard Windows bitmap, as displayed by the BmpDraw example




Here is the code of the method, which simply scans the bitmap in both directions and defines the color of each pixel. Because we are doing many operations on the bitmap, I can store a reference to it in the local `Bmp` variable for simplicity:

```
procedure TBitmapForm.GenerateSlow1Click(Sender: TObject);
var
  Bmp: TBitmap;
  I, J, T: Integer;
begin
  // get the image and modify it
  Bmp := Image1.Picture.Bitmap;
  Bmp.PixelFormat := pf24bit;
  Bmp.Width := 256;
  Bmp.Height := 256;

  T := GetTickCount;
  // change every pixel
  for I := 0 to Bmp.Height - 1 do
    for J := 0 to Bmp.Width - 1 do
      Bmp.Canvas.Pixels [I, J] := RGB (I*J mod 255, I, J);
  Caption := 'Image Viewer - Memory Image (MSecs: ' +
    IntToStr (GetTickCount - T) + ')';
end;
```

Notice that the program keeps track of the time required by this operation, which on my computer takes about six seconds. As you see from the name of the function, this is the slow version of the code.

We can speed it up considerably by accessing the bitmap one entire row at a time. This little-known feature is available through the `ScanLine` property of the bitmap,

which returns a pointer to the memory area of the bitmap line. By taking this pointer and accessing the memory directly, we make the program much faster. The only problem is that we need to know the internal representation of the bitmap. In the case of a 24-bit bitmap, every point is represented by three bytes defining the amount of blue, green, and red (the reverse of the RGB sequence). Here is the alternative code, with a slightly different output (as I've deliberately modified the calculation of the color):

```
procedure TBitmapForm.GenerateFast1Click(Sender: TObject);
var
  Bmp: TBitmap;
  I, J, T: Integer;
  Line: PByteArray;
begin
  // get the image and modify it
  Bmp := Image1.Picture.Bitmap;
  Bmp.PixelFormat := pf24bit;
  Bmp.Width := 256;
  Bmp.Height := 256;

  T := GetTickCount;
  // change every pixel, line by line
  for I := 0 to Bmp.Height - 1 do
  begin
    Line := PByteArray (Bmp.ScanLine [I]);
    for J := 0 to Bmp.Width - 1 do
    begin
      Line [J*3] := J;
      Line [J*3+1] := I*J mod 255;
      Line [J*3+2] := I;
    end;
  end;
  // refresh the video
  Image1.Invalidate;
  Caption := 'Image Viewer - Memory Image (MSecs: ' +
    IntToStr (GetTickCount - T) + ')';
end;
```

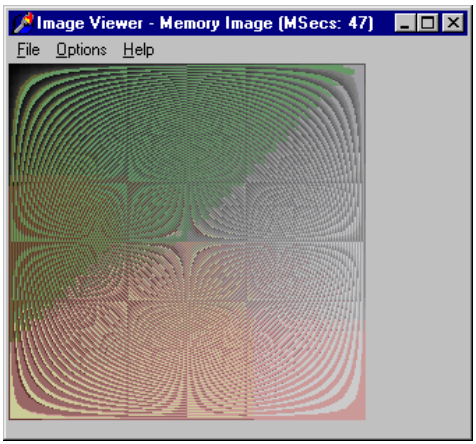
Simply moving a line in memory doesn't cause a screen update, so the program calls `Invalidate` at the end. The output produced by this second method (see Figure 22.7) is very similar, but the time it takes on my computer is about 60 milliseconds. That's about one hundredth the time of the other approach! This technique is so fast that we can use it for scrolling the lines of the bitmap and still produce a fast and smooth effect. The scrolling operation has a few options, so as you select the corresponding menu items, the program simply shows a panel inside the form.

This panel has a trackbar you can use to adjust the speed of the scrolling operation (reducing its smoothness as the speed increases). The position of the trackbar is saved in a local field of the form:

```
procedure TBitmapForm.TrackBar1Change(Sender: TObject);
begin
  nLines := TrackBar1.Position;
  TrackBar1.Hint := IntToStr (TrackBar1.Position);
end;
```

**FIGURE 22.7:**

The drawing you see on the screen is generated by the BmpDraw example in a fraction of a second (as reported in the caption).




In the panel there are also two buttons used to start and stop the scrolling operation. The code of the Go button has two `for` loops. The external loop is used to repeat the scrolling operation, as many times as there are lines in the bitmap. The internal loop does the scrolling operation by copying each line of the bitmap to the previous one. The first line is temporarily stored in a memory block and then copied to the last line at the end. This temporary memory block is kept in a dynamically allocated memory area (`AllocMem`) large enough to hold one line. This information is obtained by computing the difference in the memory addresses of two consecutive lines.

The core of the moving operation is accomplished using Delphi's `Move` function. Its parameters are the variable to be moved, not the memory addresses. For this reason, you have to de-reference the pointers. (Well, this method is really a good exercise on pointers!) Finally, notice that this time we cannot invalidate the entire image after each scrolling operation, as this produces too much flickering in the output. The opposite solution is to invalidate each line after it has been moved, but this makes the program far too slow. As an in-between solution, I decided to invalidate a block of lines at a time, as determined by the `J mod nLines = 0`

expression. When a given number of lines has been moved, the program refreshes those lines:

```
Rect (0, PanelScroll.Height + H - nLines,
  W, PanelScroll.Height + H);
```

As you can see, the number of lines is determined by the position of the TrackBar control.

A user can even change the speed by moving their thumb during the scrolling operation. We also allow the user to press the Cancel button during the operation. This is made possible by the call to `Application.ProcessMessages` in the external `for` loop. The Cancel button changes the `fCancel` flag, which is checked at each iteration of the external `for` loop:

```
procedure TBitmapForm.BtnCancelClick(Sender: TObject);
begin
  fCancel := True;
end;
```

So, after all this description, here is the complete code of the Go button's `OnClick` event handler:

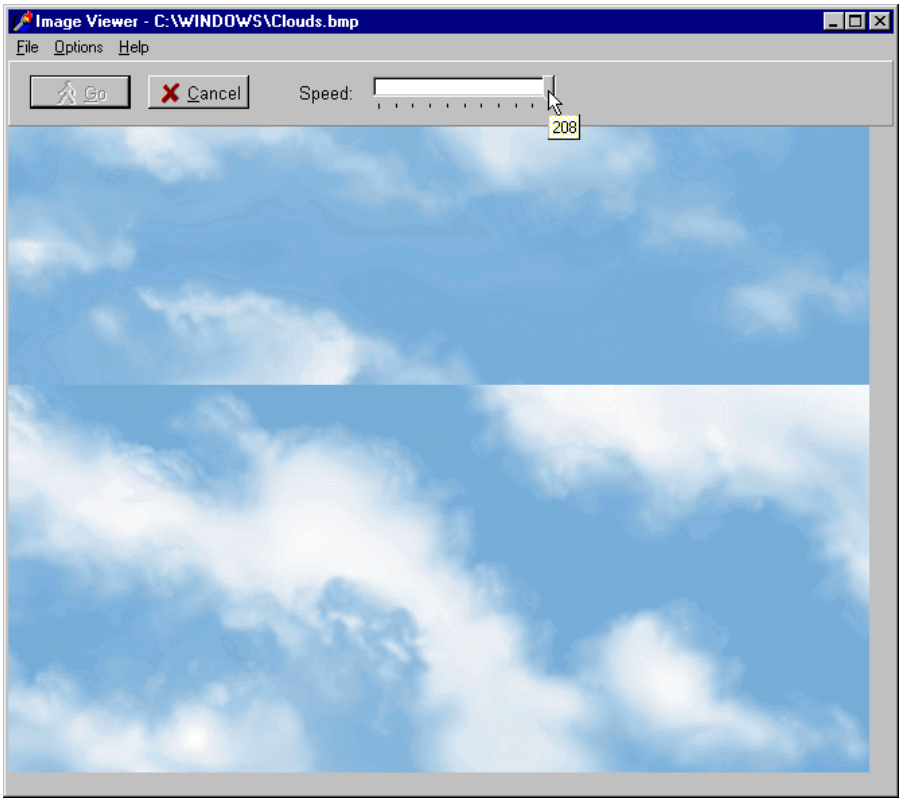
```
procedure TBitmapForm.BtnGoClick(Sender: TObject);
var
  W, H, I, J, LineBytes: Integer;
  Line: PByteArray;
  Bmp: TBitmap;
  R: TRect;
begin
  // set the user interface
  fCancel := False;
  BtnGo.Enabled := False;
  BtnCancel.Enabled := True;

  // get the bitmap of the image and resize it
  Bmp := Image1.Picture.Bitmap;
  W := Bmp.Width;
  H := Bmp.Height;

  // allocate enough memory for one line
  LineBytes := Abs (Integer (Bmp.ScanLine [1]) -
    Integer (Bmp.ScanLine [0]));
  Line := AllocMem (LineBytes);

  // scroll as many items as there are lines
  for I := 0 to H - 1 do
```

```
  begin
    // exit the for loop if Cancel was pressed
    if fCancel then
      Break;

    // copy the first line
    Move ((Bmp.ScanLine [0])^, Line^, LineBytes);

    // for every line
    for J := 1 to H - 1 do
    begin
      // move line to the previous one
      Move ((Bmp.ScanLine [J])^, (Bmp.ScanLine [J-1])^, LineBytes);
      // every nLines update the output
      if (J mod nLines = 0) then
      begin
        R := Rect (0, PanelScroll.Height + J-nLines,
          W, PanelScroll.Height + J);
        InvalidateRect (Handle, @R, False);
        UpdateWindow (Handle);
      end;
    end;

    // move the first line back to the end
    Move (Line^, (Bmp.ScanLine [Bmp.Height - 1])^, LineBytes);
    // update the final portion of the bitmap
    R := Rect (0, PanelScroll.Height + H - nLines,
      W, PanelScroll.Height + H);
    InvalidateRect (Handle, @R, False);
    UpdateWindow (Handle);

    // let the program handle other messages
    Application.ProcessMessages;
  end;

  // reset the UI
  BtnGo.Enabled := True;
  BtnCancel.Enabled := False;
end;
```

You can see a bitmap during the scrolling operation in Figure 22.8. Notice that the scrolling can take place on any type of bitmap, not just the 24-bit bitmaps generated by this program. You can, in fact, load another bitmap into the program and then scroll it, as I did to create the illustration.

**FIGURE 22.8:**

The BmpDraw example allows fast scrolling of a bitmap.




# An Animated Bitmap in a Button

Bitmap buttons are easy to use and can produce better-looking applications than the standard push buttons (the Button component). To further improve the visual effect of a button, we can also think of *animating* the button. There are basically two kinds of animated buttons—buttons that change their glyph slightly when they are pressed and buttons having a moving image, regardless of the current operation. I'll show you a simple example of each kind, Fire and World. For each of these examples, we'll explore a couple of slightly different versions.

## A Two-State Button

The first example, the Fire program, has a very simple form, containing only a bitmap button. This button is connected to a `Glyph` representing a cannon. Imagine

such a button as part of a game program. As the button is pressed, the glyph changes to show a firing cannon. As soon as the button is released, the default glyph is loaded again. In between, the program displays a message if the user has actually clicked the button.

To write this program, we need to handle three of the button's events: `OnMouseDown`, `OnMouseUp`, and `OnClick`. The code of the three methods is extremely simple:

```
procedure TForm1.BitBtnFireMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // load firing cannon bitmap
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire2.bmp');
end;

procedure TForm1.BitBtnFireMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // load default cannon bitmap
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire.bmp');
end;

procedure TForm1.BitBtnFireClick(Sender: TObject);
begin
  PlaySound ('Boom.wav', 0, snd_Async);
  MessageDlg ('Boom!', mtWarning, [mbOK], 0);
end;
```

I've added some sound capabilities, playing a WAV file when the button is pressed with a call to the `PlaySound` function of the MmSystem unit. When you hold down the left mouse button over the bitmap button, the bitmap button is pressed. If you then move the mouse cursor away from the button while holding down the mouse button, the bitmap button is released, but it doesn't get an `OnMouseUp` event, so the firing cannon remains there. If you later release the left mouse button outside the surface of the bitmap button, it receives the `OnMouseUp` event anyway. The reason is that all buttons in Windows capture the mouse input when they are pressed.

## Many Images in a Bitmap

The Fire example used a manual approach. I loaded two bitmaps and changed the value of the `Glyph` property when I wanted to change the image. The BitBtn component, however, can also handle a number of bitmaps automatically. You

can prepare a single bitmap that contains a number of images (or glyphs) and set this number as the value of the `NumGlyphs` property. All such "sub-bitmaps" must have the same size because the overall bitmap is divided into equal parts.

If you provide more than one glyph in the bitmap, they are used according to the following rules:

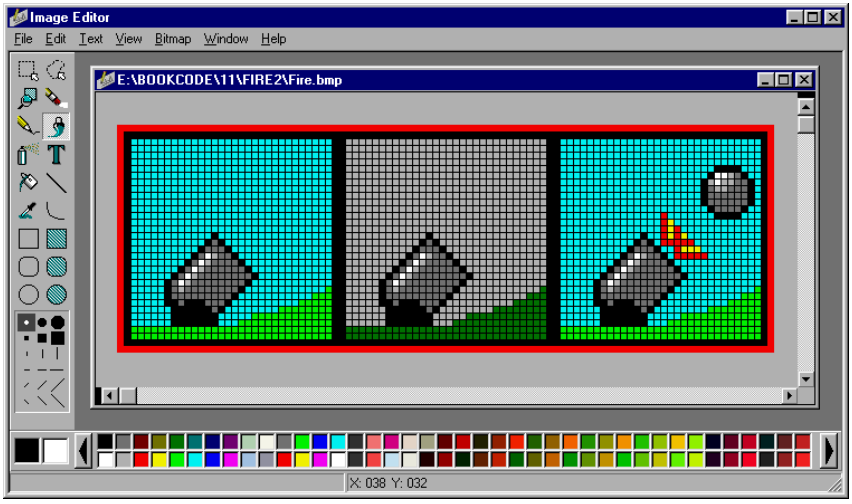
- The first bitmap is used for the released button, the default position.
- The second bitmap is used for the disabled button.
- The third bitmap is used when the button is clicked.
- The fourth bitmap is used when the button remains down, as in buttons behaving as check boxes.

Usually you provide a single glyph and the others are automatically computed from it, with simple graphical changes. However, it is easy to provide a second, a third, and a fourth customized picture. If you do not provide all four bitmaps, the missing ones will be computed automatically from the first one.

In our example, the new version of Fire (named Fire2), we only need the first and third glyphs of the bitmap but are obliged to add the second bitmap. To see how this glyph (the second of the bitmap) can be used, I've added a check box to disable the bitmap button. To build the new version of the program, I've prepared a bitmap of $32 \times 96$ pixels (see Figure 22.9) and used it for the `Glyph` property of the bitmap. Delphi automatically set the `NumGlyphs` property to 3, because the bitmap is three times wider than it is high.

**FIGURE 22.9:**

The bitmap with three images of the Fire2 example, as seen in the Delphi Image Editor



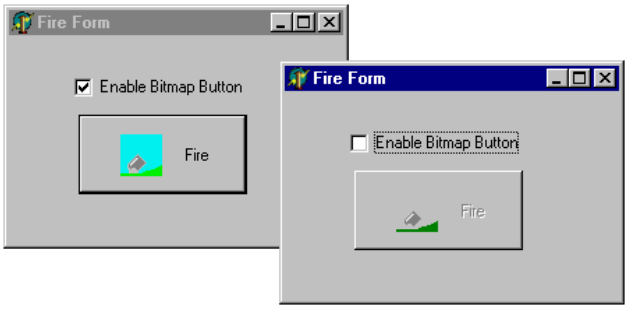
The check box, used to enable and disable the button (so we can see the glyph corresponding to the disabled status), has the following `OnClick` event:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  BitBtnFire.Enabled := CheckBox1.Checked;
end;
```

When you run the program, there are two ways to change the bitmap in the button. You can disable the bitmap button by using the check box (see Figure 22.10), or you can press the button to see the cannon fire. In the first version (the Fire example), the image with the firing cannon remained on the button until the message box was closed. Now (in the Fire2 example) the image is shown only while the button is pressed. As soon as you move outside the surface of the button, or release the button after having pressed it (activating the message box), the first glyph is displayed.

**FIGURE 22.10:**

The enabled and disabled bitmap buttons of the Fire2 example, in two different copies of the application




## The Rotating World

The second example of animation, World, has a button featuring the earth, which slowly rotates, showing the various continents. You can see some samples in Figure 22.11, but, of course, you should run the program to see its output. In the previous example, the image changed when the button was pressed. Now the image changes by itself, automatically. This occurs thanks to the presence of a Timer component, which receives a message at fixed time intervals.

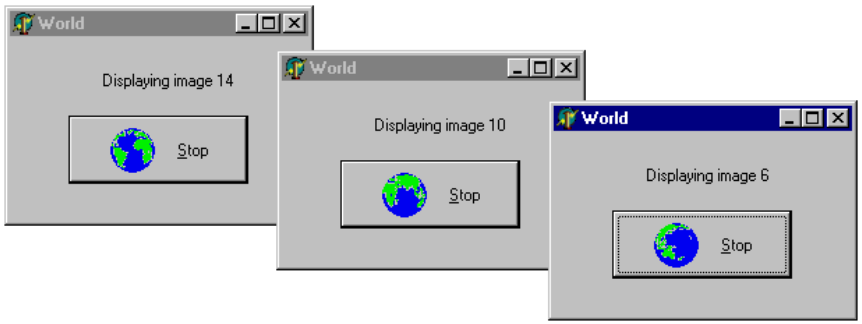
Here is a summary of the component properties:

```
object WorldForm: TWorldForm
  Caption = 'World'
  OnCreate = FormCreate
  object Label1: TLabel...
  object WorldButton: TBitBtn
    Caption = '&Start'
    OnClick = WorldButtonClick
```

```
    Glyph.Data = {W1.bmp}
    Spacing = 15
  end
  object Timer1: TTimer
    Enabled = False
    Interval = 500
    OnTimer = Timer1Timer
  end
end
```

**FIGURE 22.11:**

Some examples of the running World program




The timer component is started and stopped (enabled and disabled) when the user presses the bitmap button with the world image:

```
procedure TWorldForm.WorldButtonClick(Sender: TObject);
begin
  if Timer1.Enabled then
  begin
    Timer1.Enabled := False;
    WorldButton.Caption := '&Start';
  end
  else
  begin
    Timer1.Enabled := True;
    WorldButton.Caption := '&Stop';
  end;
end;
```

As you can see in Figure 22.11, a label above the button indicates which of the images is being displayed. Each time the timer message is received, the image and label change:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := 'Displaying image ' +
```

```
    IntToStr (Count);
  WorldButton.Glyph.LoadFromFile (
    'w' + IntToStr (Count) + '.bmp');
end;
```

In this code, `Count` is a field of the form that is initialized to `1` in the `FormCreate` method. At each timer interval, `Count` is increased modulus `16` and then converted into a string (preceded by the letter *w*). The reason for this limit is simple—I had 16 bitmaps of the earth to display. Naming the bitmap files `W1.BMP`, `W2.BMP`, and so on makes it easy for the program to access them, building the strings with the name at run time.

**NOTE**

The modulus operation returns the remainder of the division between integers. This means that `Count mod 16` invariably returns a value in the range 0–15. Adding one to this return value, we obtain the number of the bitmap, which is in the range 1–16.

## A List of Bitmaps, the Use of Resources, and a ControlCanvas

The World program works, but it is very slow, for a couple of reasons. First of all, at each timer interval, it needs to read a file from the disk, and although a disk cache can make this faster, it is certainly not the most efficient solution. Besides reading the file from disk, the program has to create and destroy Windows bitmap objects, and this takes some time. The second problem depends on how the image is updated: When you change the button's bitmap, the component is completely erased and repainted. This causes some flickering, as you can see by running the program.

To solve the first problem (and to show you a different approach to handling bitmaps), I've created a second version of the example, World2. Here I've added a `TObjectList` Delphi 5 container, storing a list of bitmaps, to the program's form. The form has also some more fields:

```
type
  TWorldForm = class(TForm)
    ...
  private
    Count, YPos, XPos: Integer;
    BitmapsList: TObjectList;
    ControlCanvas: TControlCanvas;
  end;
```

All the bitmaps are loaded when the program starts and destroyed when it terminates. At each timer interval, the program shows one of the list's bitmaps in the bitmap button. By using a list, we avoid loading a file each time we need to display a bitmap, but we still need to have all the files with the images in the directory with the executable file. A solution to this problem is to move the bitmaps from independent files to the application's resource file. This is easier to do than to explain.

To use the resources instead of the bitmap files, we need to first create this file. The best approach is to write a resource script (an RC file), listing the names of the bitmap files and of the corresponding resources. Open a new text file (in any editor) and write the following code:

```
W1  BITMAP  "W1.BMP"
W2  BITMAP  "W2.BMP"
W3  BITMAP  "W3.BMP"
// ... and so on
```

Once you have prepared this RC file (I've named it `WorldBmp.RC`), you can compile it into a RES file using the resource compiler included and the BRCC32 command-line application you can find in the BIN directory of Delphi, and then include it in the project by adding the `{$R WORLDBMP.RES}` directive in the project source code file or in one of the units.

In Delphi 5, however, you can use a simpler approach. You can take the RC file and simply add it to the project using the Project Manager Add command or simply dragging the file to the project. Delphi 5 will automatically activate the resource compiler, and it will then bind the resource file into the executable file. These operations are controlled by an extended resource inclusion directive added to the project source code:

```
{$R 'WORLDBMP.res' WORLDBMP.RC'}
```

Once we have properly defined the resources of the application, we need to load the bitmaps from the resources. For a `TBitmap` object we can use the `LoadFromResourceName` method, if the resource has a string identifier, or the `LoadFromResourceID` method, if it has a numeric identifier. The first parameter of both methods is a handle to the application, known as `HInstance`, available in Delphi as a global variable.

**TIP**

Delphi defines a second global variable, `MainInstance`, which refers to the `HInstance` of the main executable file. Unless you are inside a DLL, you can use one or the other interchangeably.

This is the code of the FormCreate method:

```
procedure TWorldForm.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  Count := 1;
  // load the bitmaps and add them to the list
  BitmapsList := TList.Create;
  for I := 1 to 16 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromResourceName (HInstance,
      'W' + IntToStr (I));
    BitmapsList.Add (Bmp);
  end;
end;
```

**NOTE**

As an alternative, we could have used the ImageList component, but for this example I decided to use a low-level approach to show you all the details involved.

One problem remains to be solved: obtaining a smooth transition from one image of the world to the following one. The program should paint the bitmaps in a canvas using the Draw method. Unfortunately, the bitmap button's canvas is not directly available (and not event protected), so I decided to use a TControlCanvas (usually the internal canvas of a control, but one you can also associate to externally) To use it to paint over a button, we can assign the button to the control canvas in the FormCreate method:

```
ControlCanvas := TControlCanvas.Create;
ControlCanvas.Control := WorldButton;
YPos := (WorldButton.Height - Bmp.Height) div 2;
XPos := WorldButton.Margin;
```

The horizontal position of the button where the image is located (and where we should paint) depends on the Margin of the icon of the bitmap button and on the height of the bitmap. Once the control canvas is properly set, the Timer1Timer method simply paints over it—and over the button:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := Format ('Displaying image %d', [Count]);
  // draw the current bitmap in the control canvas
```

```
    ControlCanvas.Draw (XPos, YPos,
      BitmapsList.Items[Count-1] as TBitmap);
end;
```

The last problem is to move the position of the image when the left mouse button is pressed or released over it (that is, in the `OnMouseDown` and `OnMouseUp` events of the button). Besides moving the image by few pixels, we should update the glyph of the bitmap, because Delphi will automatically display it while redrawing the button. Otherwise, a user would see the initial image until the timer interval elapsed and the component fired the `OnTimer` event. (That might take a while if you've stopped it!) Here is the code of the first of the two methods:

```
procedure TWorldForm.WorldButtonMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    // paint the current image over the button
    WorldButton.Glyph.Assign (
      BitmapsList.Items[Count-1] as TBitmap);
    Inc (YPos, 2);
    Inc (XPos, 2);
  end;
end;
```

# The Animate Control

There is a better way to obtain animation than displaying a series of bitmaps in sequence. Use the Win32 Animate common control. The Animate control is based on the use of AVI (Audio Video Interleaved) files, a series of bitmaps similar to a movie.

**NOTE**

Actually, the Animate control can display only those AVI files that have a single video stream, are uncompressed or compressed with RLE8 compression, and have no palette changes; and if they have sound, it is ignored. In practice, the files corresponding to this requirement are those made of a series of computer bitmaps, not those based on an actual film.

The Animate control can have two possible sources for its animation:

- It can be based on any AVI file that meets the requirements indicated in the note above; to use this type of source, set a proper value for the `FileName` property.

- It can use a special internal Windows animation, part of the common control library; to use this type of source, choose one of the possible values of the `CommonAVI` property (which is based on an enumeration).

If you simply place an Animate control on a form, choose an animation using one of the methods just described, and finally, set its `Active` property to `True`, you'll start seeing the animation performed even at design time. By default, the animation runs continuously, restarting it as soon as it is done. However, you can regulate this effect by using the `Repetitions` property. The default value `-1` causes infinite repetition; use any other value to specify a number of repetitions.

You can also specify the initial and final frame of the sequence, with the `StartFrame` and `StopFrame` properties. These three properties (initial position, final position, and number of repetitions) correspond to the three parameters of the `Play` method, which you'll often use with an Animate control. As an alternative, you can set the properties and then call the `Start` method. At run time, you can also access the total number of frames using the `FrameCount` property: you can use this to execute the animation from the beginning to the end. Finally, for finer control, you can use the `Seek` method, which displays a specific frame.

I've used all of these methods in a simple demo program, which can use both files and the Windows standard animations. The program allows you to choose a file or one of the animations by using a ListBox. I've added an item to this ListBox for each element of the `TCommonAVI` enumeration and used the same order:

```
object ListBox1: TListBox
  Items.Strings = (
    '[Use an AVI file]'
    'Find Folder'
    'Find File'
    'Find Computer'
    'Copy Files'
    'Copy File'
    'Recycle File'
    'Empty Recycle'
    'Delete File')
  OnClick = ListBox1Click
end
```

Thanks to this structure, when the user clicks on the ListBox, simply casting the number of the selected items to the enumerated data type will get the proper value for the `CommonAVI` property.

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Animate1.CommonAVI := TCommonAVI (ListBox1.ItemIndex);
  if (ListBox1.ItemIndex = 0) and
```

```
      OpenDialog1.Execute then
    Animate1.FileName := OpenDialog1.FileName
end;
```

As you can see, when the first item is selected (the value is `caNone`), the program automatically loads an AVI file, using an OpenDialog component. The most important component of the form is the Animate control. Here is its textual description:

```
object Animate1: TAnimate
  AutoSize = False
  Align = alClient
  CommonAVI = aviFindFolder
  OnOpen = Animate1Open
end
```

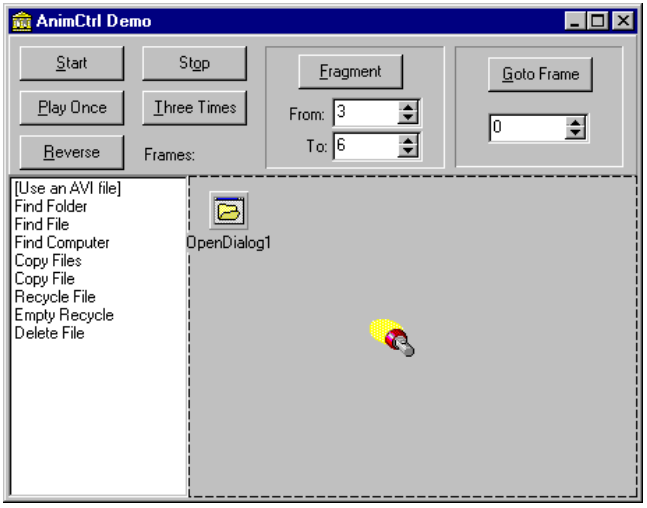
It's aligned to the client area, so that a user can easily resize it depending on the actual size of the frames of the animation. As you can see, I've also defined a handler for an event of this component, `OnOpen`:

```
procedure TForm1.Animate1Open(Sender: TObject);
begin
  LblFrames.Caption := 'Frames ' +
    IntToStr (Animate1.FrameCount);
end;
```

When a new file (or common animation) is opened, the program simply outputs the number of its frames in a label. This label is hosted together with several buttons and a few SpinEdit controls into a big panel, acting as a toolbar. You can see them in the design-time form of Figure 22.12.

**FIGURE 22.12:**

The form of the AnimCtrl example at design time. The Animate control is actually showing an animation, even before running the program.



The Start and Stop buttons are completely trivial, but the Play Once button has some simple code:

```
procedure TForm1.BtnOnceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 1);
end;
```

Things start getting more interesting with the code used to play the animation three times or to play only a fragment of it. Both of these methods are based on the `Play` method:

```
procedure TForm1.BtnTriceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 3);
end;

procedure TForm1.BtnFragmentClick(Sender: TObject);
begin
  Animate1.Play (SpinEdit1.Value, SpinEdit2.Value, -1);
end;
```

The last two button event handlers are based on the `Seek` method. The Goto button simply moves to the frame indicated by the corresponding SpinEdit component, while the Reverse buttons move to each frame in turn, starting with the last one and pausing between each of them:

```
procedure TForm1.BtnGotoClick(Sender: TObject);
begin
  Animate1.Seek (SpinEdit3.Value);
end;

procedure TForm1.BtnReverseClick(Sender: TObject);
var
  Init: TDateTime;
  I: Integer;
begin
  for I := Animate1.FrameCount downto 1 do
  begin
    Animate1.Seek (I);
    // wait 50 milliseconds
    Init := Now;
    while Now < Init + EncodeTime (0, 0, 0, 50) do
      Application.ProcessMessages;
  end;
end;
```

## The Animate Control in a Button

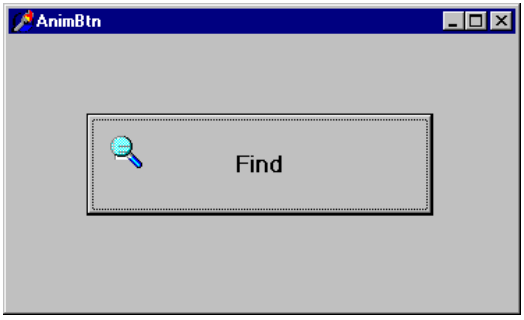
Now that you know how the Animate control works, we can use it to build another animated button. Simply place an Animate control and a large button (possibly with a large font as well) in a form. Then write the following code to make the button the parent window of the Animate control at run time and position it properly:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hDiff: Integer;
begin
  Animate1.Parent := Button1;
  hDiff := Button1.Height - Animate1.Height;
  Animate1.SetBounds (hDiff div 2, hDiff div 2,
    Animate1.Width, Animate1.Height);
  Animate1.Active := True;
end;
```

You can see an example of this effect in Figure 22.13. (The project has the name AnimBtn.) This is indeed the simplest approach to producing an animated button, but it also permits the least control.

**FIGURE 22.13:**

The effect of the Animate control inside a button, as shown by the AnimBtn program




# Graphical Grids

Grids represent another interesting group of Delphi graphical components. The system offers different grid components: a grid of strings, one of images, database-related grids, and a sample grid of colors. The first two kinds of grids are particularly useful because they allow you to represent a lot of information and

let the user navigate it. Of course, grids are extremely important in database programming, and they can be customized with graphics as we've seen in Chapter 10 of *Mastering Delphi 5*.

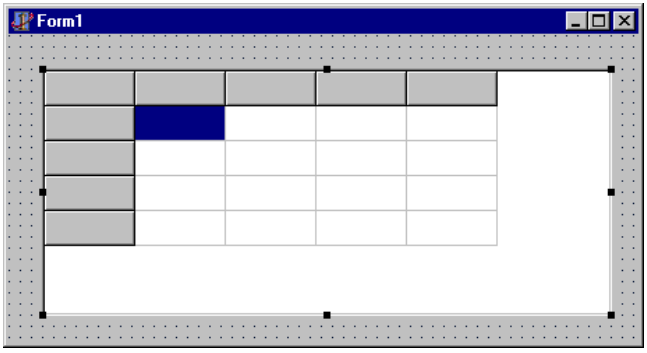
The DrawGrid and StringGrid components are closely related. In fact, the `TStringGrid` class is a subclass of `TDrawGrid`. What use are these grids? Basically, you can store some values, either in the strings related to the StringGrid or in other data structures, and then display selected values, using specific criteria. While grids of strings can be used almost as they are (because they already provide editing capabilities), the grids of generic objects usually require more coding.

Grids, in fact, define the way information is organized for display, not how it is stored. The only grid that stores the data it displays is the StringGrid. All other grids (including the DrawGrid and the DBGrid components) are just data viewers, not data containers. The DBGrid doesn't own the data it displays; it fetches the data from the connected data source. This is sometimes a source of confusion.

The basic structure of a grid includes a number of fixed columns and rows, which indicate the nonscrollable region of the grid (as you can see in Figure 22.14). Grids are among the most complex components available in Delphi, as indicated by the high number of properties and methods they contain. There are a great many options and properties for grids, controlling both their appearance and their behavior.

**FIGURE 22.14:**

When you place a new grid component on a form, it contains one fixed row and one fixed column by default.




In its appearance, the grid can have lines of different sizes, or it can have no lines. You can set the size of each column or row independently of the others because the `RowSize`, `ColWidth`, and `RowHeight` properties are arrays. For the grid's behavior, you can let the user resize the columns and the rows (`goColSizing` and `goRowSizing`), drag entire columns and rows to a new position (`goRowMoving` and `goColumnMoving`), select automatic editing, and allow range selections. Because various options allow users to perform a number of operations on grids,

there are also a number of events related to grids, such as `OnColumnMoved`, `OnDrawCell`, or `OnSetEditText`.

The most important event is probably `OnDrawCell`. In response to this event, a program has to paint a certain cell of the grid. Only string grids can automatically display their contents. The DrawGrid, in fact, doesn't have support for storing data. It is simply a tool for arranging a portion of the screen to display information in a regular format. It is a simple tool but also a powerful one. Methods such `CellRect`, which returns the rectangle corresponding to the area of a cell, or `MouseToCell`, which returns the cell in a specific location, are a joy to use. By handling resizable rows and columns and scrollable grids, they simplify complex tasks and spare the programmer from tedious calculations.

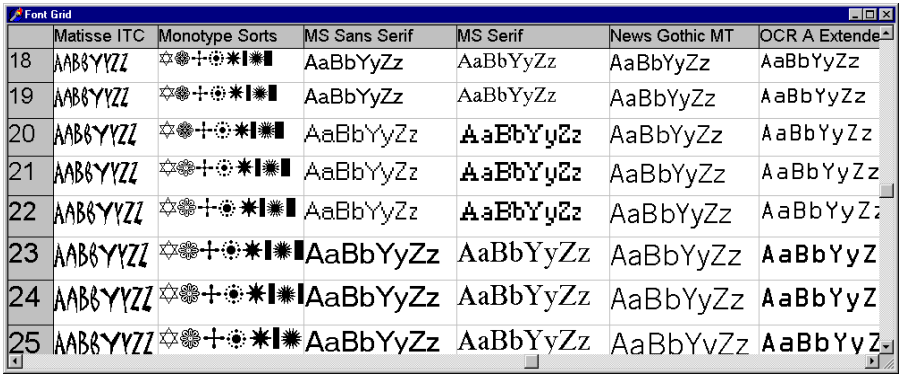
What can you use a grid for? Building a spreadsheet is probably the first idea that comes to mind, but that's probably a little too complex for an example. I've decided to use the StringGrid control in a program that shows the fonts installed in the system and the DrawGrid control in a program that emulates the MineSweeper game.

## A Grid of Fonts

If you place a StringGrid component on a form and set its options properly, you have a full working editor of strings arranged in a grid, without doing any programming at all. To make the example more interesting, I've decided to draw each cell of the grid with a different font, varying both its size and its typeface. You can see the result of the FontGrid program in Figure 22.15.

**FIGURE 22.15:**

An example of the output of the FontGrid application




The form of this program is very simple. You need only place a grid component on a form, align it with the client area, set a few properties and options, and let the program do the rest. The number of columns and rows and their size, in fact,

are computed at run time. The important properties you need to set are `DefaultDrawing`, which should be `False` to let us paint the grid as we like, and `Options`:

```
object Form1: TForm1
  Caption = 'Font Grid'
  OnCreate = FormCreate
  object StringGrid1: TStringGrid
    Align = alClient
    DefaultColWidth = 200
    DefaultDrawing = False
    Options = [goFixedVertLine, goFixedHorzLine,
      goVertLine, goHorzLine, goDrawFocusSelected,
      goColSizing, goColMoving, goEditing]
    OnDrawCell = StringGrid1DrawCell
  end
end
```

As usually happens in Delphi, the simpler the form is, the more complex the code. This example follows that rule, although it has only two methods, one to initialize the grid at start-up and the other to draw the items. The editing, in fact, has not been customized and takes place using the system font. The first of the two methods is `FormCreate`. At the beginning, this method uses the global `Screen` object to access the fonts installed in the system.

The grid has a column for each font as well as a fixed column with numbers representing font sizes. The name of each column is copied from the `Screen` object to the first row of each column (which has a zero index):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  {the number of columns equals the number of fonts plus
  1 for the first fixed column, which has a size of 20}
  StringGrid1.ColCount := Screen.Fonts.Count + 1;
  StringGrid1.ColWidths [0] := 50;

  for I := 1 to Screen.Fonts.Count do
  begin
    // write the name of the font in the first row
    StringGrid1.Cells [I, 0] :=
      Screen.Fonts.Strings [I-1];

    {compute maximum required size of column, getting the width
    of the text with the biggest size of the font in that column}
    StringGrid1.Canvas.Font.Name :=
```

```
        StringGrid1.Cells [I, 0];
      StringGrid1.Canvas.Font.Size := 32;
      StringGrid1.ColWidths [I] :=
        StringGrid1.Canvas.TextWidth ('AaBbYyZz');
  end;
  ...
```

In the last part of the code above, the program computes the width of each column. This is accomplished by evaluating the space occupied by the custom string of text *AaBbYyZz*, using the font of the column (written in the first row, `Cells [I, 0]`) and the biggest font size used by the program (`32`). To compute the space required by the text, you can apply the `TextWidth` and `TextHeight` methods to a canvas with the proper font selected.

The rows, instead, are always `26` and have an increasing height, computed with the approximate formula: `15 + I x 2`. In fact, computing the highest text means checking the height of the text in each column, certainly too complex an operation for this example. The approximate formula works well enough, as you can see in Figure 22.15 and by running the program. In the first cell of each row, the program writes the size of the font, which corresponds to the number of the line plus seven.

The last operation is to store the string "*AaBbYyZz*" in each nonfixed cell of the grid. To accomplish this, the program uses a nested `for` loop. Expect to use nested `for` loops often when working with grids. Here is the second part of the `FormCreate` method:

```
  // defines the number of columns
  StringGrid1.RowCount := 26;
  for I := 1 to 25 do
  begin
    // write the number in the first column
    StringGrid1.Cells [0, I] := IntToStr (I+7);
    // set an increasing height for the rows
    StringGrid1.RowHeights [I] := 15 + I*2;
    // insert default text in each column
    for J := 1 to StringGrid1.ColCount do
      StringGrid1.Cells [J, I] := 'AaBbYyZz'
  end;
  StringGrid1.RowHeights [0] := 25;
end;
```

Now we can study the second method, `StringGrid1DrawCell`, which corresponds to the grid's `OnDrawCell` event. This method has a number of parameters:

- `Col` and `Row` refer to the cell we are currently painting.
- `Rect` is the area of the cell we are going to paint.

- `State` is the state of the cell, a set of three flags, which can be active at the same time: `gdSelected` (the cell is selected), `gdFocused` (the cell has the input focus), and `gdFixed` (the cell is in the fixed area, which usually has a different background color). It is important to know the state of the cell because this usually affects its output.

The `DrawCell` method paints the text of the corresponding element of the grid, with the font used by the column and the size used for the row. Here is the listing of this method:

```
procedure TForm1.StringGrid1DrawCell (Sender: TObject;
  Col, Row: Integer; Rect: TRect; State: TGridDrawState);
begin
  // select a font, depending on the column
  if (Col = 0) or (Row = 0) then
    StringGrid1.Canvas.Font.Name := I
  else
    StringGrid1.Canvas.Font.Name :=
      StringGrid1.Cells [Col, 0];

  // select the size of the font, depending on the row
  if Row = 0 then
    StringGrid1.Canvas.Font.Size := 14
  else
    StringGrid1.Canvas.Font.Size := Row + 7;

  // select the background color
  if gdSelected in State then
    StringGrid1.Canvas.Brush.Color := clHighlight
  else if gdFixed in State then
    StringGrid1.Canvas.Brush.Color := clBtnFace
  else
    StringGrid1.Canvas.Brush.Color := clWindow;

  // output the text
  StringGrid1.Canvas.TextRect (
    Rect, Rect.Left, Rect.Top,
    StringGrid1.Cells [Col, Row]);

  // draw the focus
  if gdFocused in State then
    StringGrid1.Canvas.DrawFocusRect (Rect);
end;
```

The font's name is retrieved by the row 0 of the same column. The font's size is computed by adding 7 to the number of the row. The fixed columns use some default values. Having set the font and its size, the program selects a color for the background of the cell, depending on its possible states: selected, fixed, or normal (that is, no special style). The value of the style's `gdFocused` flag is used a few lines later to draw the typical focus rectangle. When everything is set up, the program can perform some real output, drawing the text and if necessary the focus rectangle, with the last two statements of the `StringGrid1DrawCell` method above.

**TIP**

To draw the text in the grid's cell, I've used the `TextRect` method of the canvas instead of the more common `TextOut` method. The reason is that `TextRect` clips the output to the given rectangle, preventing drawing outside this area. This is particularly important in the case of grids because the output of a cell should not cross its borders. Since we are painting on the canvas of the whole grid, when we are drawing a cell, we can end up corrupting the contents of neighboring cells, too.

As a final observation, remember that when you decide to draw the contents of a grid's cell, you should not only draw the default image but also provide a different output for the selected item, properly draw the focus, and so on.

## Mines in a Grid

The StringGrid component uses the `Cells` array to store the values of the elements and also has an `Objects` property to store custom data for each cell. The DrawGrid component, instead, doesn't have a predefined storage. For this reason, the next example defines a two-dimensional array to store the value of the grid's cells—that is, of the playing field.

The Mines example is a clone of the MineSweeper game included with Windows. If you have never played this game, I suggest you try it and read its rules in the Help file since I'll give only a basic description. When the program starts, it displays an empty field (a grid) in which there are some hidden mines. By clicking the left mouse button on a cell, you test whether or not there is a mine in that position. If you find a mine, it explodes, and the game is over. You have lost.

If there is no mine in the cell, the program indicates the number of mines in the eight cells surrounding it. Knowing the number of mines near the cell, you have a good hint for the following turn. To help you further on, when a cell has zero mines in the surrounding area, the number of mines for these cells is automatically displayed, and if one of them has zero surrounding mines, the process is

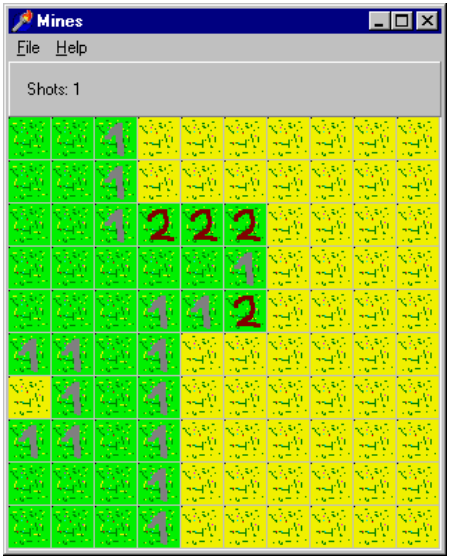
repeated. So if you are lucky, with a single click you might uncover a good number of clear cells (see Figure 22.16).

When you think you have found a mine, simply right-click on the cell; this places a flag there. The program does not say whether your inference is correct; the flag is only a hint for your future attempts. If you later change your mind, you can again right-click on the cell to remove the flag. When you have found all of the mines, you have won, and the game terminates.

Those are the rules of the game. Now we have to implement them, using a DrawGrid as starting point. In this example, the grid is fixed and cannot be resized or modified in any way at run time. In fact, it has square cells of 30 × 30 pixels, which will be used to display bitmaps of the same size.

**FIGURE 22.16:**

The Mines program after a single lucky click. A group of cells with no mines is displayed at once.




The code of this program is complex, and it is not easy to find a starting point to describe it. For this reason, I've added more comments than usual to the source code (in the download files) so you can browse through it to understand what it does. Nonetheless, I'll describe its most important elements. First of all, the program's data is stored in two arrays (declared as `private` fields of the form):

```
Display: array [0 .. NItems - 1, 0 .. NItems -1] of Boolean;
Map: array [0 .. NItems - 1, 0 .. NItems -1] of Char;
```

The first is an array of `Boolean` values that indicate whether an item should be displayed or remain hidden. Notice that the number of rows and columns of this

array is `NItems`. You can freely change this constant, but you should resize the grid accordingly. The second array, `Map`, holds the positions of the mines and flags and the numbers of the surrounding mines. It uses character codes instead of a proper enumeration data type, in order to use the digits `0`–`8` to indicate the number of mines around the cell. Here is a list of the codes:

- *M: Mine* indicates the position of a mine that the user still has not found.
- *K: Known mine* indicates the position of a mine already found by the user and having a flag.
- *W: Wrong mine* indicates a position where the user has set a flag but where there is no mine.
- *0 to 8: Number of mines* indicates the number of mines in the surrounding cells.

The first method to explore is `FormCreate`, executed at start-up. This method initializes a number of fields of the form class, fills the two arrays with default values (using two nested `for` loops), and then sets the mines in the grid. For the number of times defined in a constant (that is, the number of mines), the program adds a new mine in a random position. However, if there was already a mine, the loop should be executed once more because the final number of mines in the `Map` array should equal the requested number. Otherwise the program will never terminate, because it tests when the number of mines found equals the number of mines added to the grid. Here is the code of the loop; it can be executed more than `NMines` times, thanks to the use of the `MinesToPlace` integer variable, which is increased when we try to place a mine over an existing one:

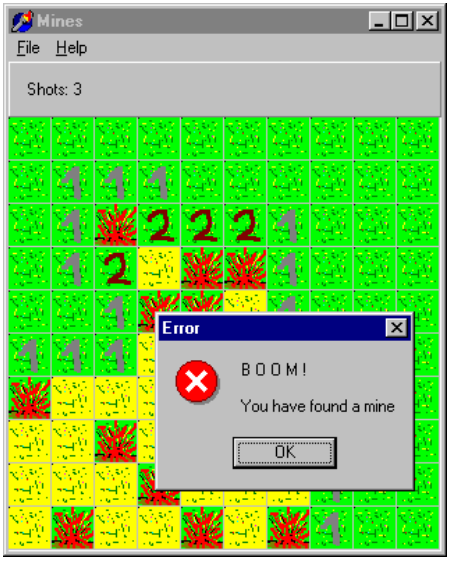
```
Randomize;
// place 'NMines' non-overlapping mines
MinesToPlace := NMines;
while MinesToPlace > 0 do
begin
  X := Random (NItems);
  Y := Random (NItems);
  // if there isn't a mine
  if Map [X, Y] <> 'M' then
  begin
    // add a mine
    Map [X, Y] := 'M';
    Dec (MinesToPlace)
  end;
end;
```

The last portion of the initialization code computes the number of surrounding mines for each cell that doesn't have a mine. This is accomplished by calling the `ComputeMines` procedure for each cell. The code of this function is fairly complex because it has to consider the special cases of the mines near a border of the grid. The effect of this call is to store, in the `Map` array, the character representing the number of mines surrounding each cell.

The next logical procedure is `DrawGrid1MouseDown`. This method first computes the cell on which the mouse has been clicked, with a call to the grid's `MouseToCell` method. Then there are three alternative portions of code: a small one when the game has ended, and the other two for the two mouse buttons. When the left mouse button is pressed, the program checks whether there is a mine (hidden or not), and if there is, it displays a message and terminates the program with an explosion (see Figure 22.17).

**FIGURE 22.17:**

Ouch! You have stepped on a mine.




If there is no mine, the program sets the `Display` value for the cell to `True`, and if there is a `0`, it starts the `FloodZeros` procedure. This method displays the eight items near a visible cell having a value of 0, repeating the operation over and over if one of the surrounding cells also has a value of 0. This recursive call is complex because you have to provide a way to terminate it. If there are two cells near each other, both having a value of 0, each one is in the surrounding area of the other, so they might continue forever to ask the other cell to display itself and its surrounding cells. Again, the code is complex, and the best way to study it may be to step through it in the debugger.

When the user presses the right mouse button, the program changes the status of the cell. The right mouse button action is to toggle the flag on the screen, so a user can always remove an existing flag, if he or she thinks the earlier decision was wrong. For this reason the status of a cell that contains a mine can change from *M* (hidden Mine) to *K* (Known mine) and vice versa; and the status of a cell with no mine can change from a number to *W* (Wrong mine) and vice versa. When all the mines have been found, the program terminates with a congratulation message.

A very important piece of code is at the end of the `OnMouseDown` event response method. Each time the user clicks on a cell and its contents change, that cell should be repainted. If you repaint the whole grid, the program will be slower. For this reason, I've used the Windows API function `InvalidateRect`:

```
MyRect := DrawGrid1.CellRect (Col, Row);
InvalidateRect (DrawGrid1.Handle, @MyRect, False);
```

The last important method is `DrawGrid1DrawCell`. We already used this painting procedure in the last example, so you should remember that it is called for each cell that needs repainting. Fundamentally, this method extracts the code corresponding to the cell, which shows a corresponding bitmap, loaded from a file. Once again, I've prepared a bitmap for each of the images in a new resource file, which is included in the project thanks to Delphi 5's improved Project Manager.

Recall that when using resources, the code tends to be faster than when using separate files, and again, we end up with a single executable file to distribute. The bitmaps have names corresponding to the code in the grid, with a character (*'M'*) in front since the name *'0'* would have been invalid. The bitmaps can be loaded and drawn in the cell with this code:

```
Bmp.LoadFromResourceName (HInstance, 'M' + Code);
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Of course, this takes place only if the cell is visible—that is, if `Display` is `True`. Otherwise, a default undefined bitmap is displayed. (The bitmap name is *'UNDEF'*.) Loading the bitmaps from the resources each time seems slow, so the program could have stored all the bitmaps in a list in memory, as the World2 example earlier in this chapter did. However, this time, I decided to use a different, although slightly less efficient, approach: a cache. This makes sense because we already use resources instead of files to speed up things.

The bitmap cache of Mines is small since it has just one element, but its presence speeds up the program considerably. The program stores the last bitmap it has used and its code; then, each time it has to draw a new item, if the code is the same, it uses the cached bitmap. Here is the new version of the code above:

```
if not (Code = LastBmp) then
```

```
begin
  Bmp.LoadFromResourceName (HInstance, 'M' + Code);
  LastBmp := Code;
end;
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Increasing the size of this cache will certainly improve its speed. You can consider a list of bitmaps as a big cache, but this is probably useless because some bitmaps (those with high numbers) are seldom used. As you can see, some improvements can be made to speed up the program, and much can also be done to improve its user interface. If you have understood this version of the program, I think you'll be able to improve it considerably.

# Using TeeChart

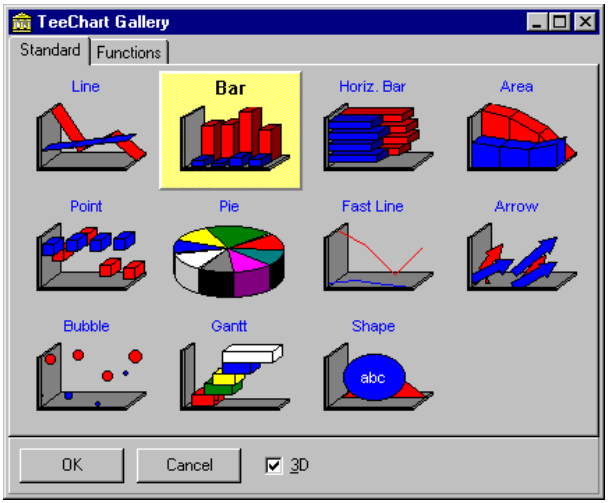
TeeChart is a VCL-based charting component built by David Berneda and licensed to Borland for inclusion in the Developer and Client/Server versions of Delphi. The TeeChart component is very complex: Delphi includes a Help file and other reference material for this component, so I won't spend time listing all of its features. I'll just build a couple of examples. TeeChart comes in three versions: the stand-alone component (in the Additional page of the Component Palette), the data-aware version (in the Data Controls page), and the Report version (in the QuickReport page). Delphi Client/Server also includes a DecisionChart control in the Decision Cube page of the palette. The data-aware version of TeeChart is presented in Chapter 9 of *Mastering Delphi 5*, and I'll use it again later in a Web-oriented example.

**NOTE** Of course, it would be simpler to build an example using the TeeChart Wizard, but seeing all the steps will give you a better understanding of this component's structure.

The TeeChart component provides the basic structure for charting, through a complex framework of charting and series classes and the visual container for charts (the actual control). The actual charts are objects of class `TChartSeries` or derived classes. Once you've placed the TeeChart component on a form, you should create one or more series. To accomplish this, you can open the Chart Component Editor: select the component, right-click to show the local menu of the form designer, and choose the Edit Chart command. Now press the Add button, and choose the graph (or series) you want to add from the many available (as you can see in Figure 22.18).

**FIGURE 22.18:**

The TeeChart Gallery allows you to choose the type of graph, or series.




As soon as you create a new series, a new object of a `TChartSeries` subclass is added to your form. This is the same behavior as the MainMenu component, which adds objects of the `TMenuItem` class to the form. You can then edit the properties of the `TSeries` object in the Chart Component Editor, or you can select the `TChartSeries` object in the Object Inspector (with the Object Selector combo box) and edit its many properties.

The different `TChartSeries` subclasses—that is, the different kinds of graph—have different properties and methods (although some of them are common to more than one subclass). Keep in mind that a graph can have multiple series: if they are all of the same type they will probably integrate better, as in the case of multiple bars. Anyway, you can also have a complex layout with graphs of different types visible at the same time. At times, this is an extremely powerful option.

## Building a First Example

To build this example I placed a TeeChart component in a form and then simply added four 3D Bar series—that is, four objects of the `TBarSeries` class. Then I set up some global properties, such as the title of the chart, and so on. Here is a summary of this information, taken from the textual description of the form:

```
object Chart1: TChart
  AnimatedZoom = True
  Title.Text.Strings = (
    'Simple TeeChart Demo for Mastering Delphi')
  BevelOuter = bvLowered
```

```
    object Series1: TBarSeries
      SeriesColor = clRed
      Marks.Visible = False
    end
    object Series2: TBarSeries
      SeriesColor = clGreen
      Marks.Visible = False
    end
    object Series3: TBarSeries
      SeriesColor = clYellow
      Marks.Visible = False
    end
    object Series4: TBarSeries
      SeriesColor = clBlue
      Marks.Visible = False
    end
  end
```

Next I added to the form a string grid and a push button labeled *Update*. This button is used to copy the numeric values of the string grid to the chart. The grid is based on a $5 \times 4$ matrix as well as a line and a column for the titles. Here is its textual description:

```
object StringGrid1: TStringGrid
  ColCount = 6
  DefaultColWidth = 50
  Options = [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goEditing]
  ScrollBars = ssNone
  OnGetEditMask = StringGrid1GetEditMask
end
```

The value 5 for the `RowCount` property is a default, and it doesn't show up in the textual description. (The same holds for the value of 1 for the `FixedCols` and `FixedRows` properties.) An important element of this string grid is the edit mask used by all of its cells. This is set using the `OnGetEditMask` event:

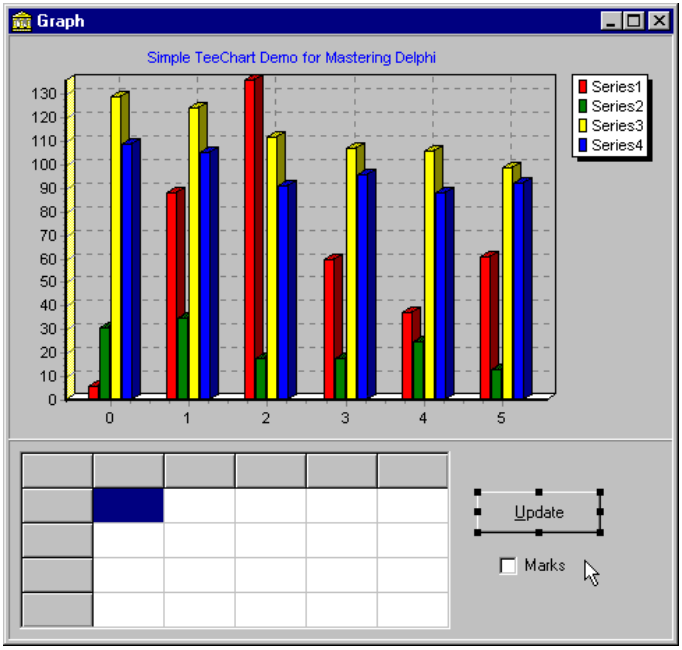
```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject;
  ACol, ARow: Longint; var Value: string);
begin
  // edit mask for the grid cells
  Value := '09;0';
end;
```

There is actually one more component, a check box used to toggle the visibility of the marks of the series. (The *marks* are small yellow tags describing each value; you'll need to run the program to see them.) You can see the form at design time

in Figure 22.19. In this case the series are populated with random values; this is a nice feature of the component, as it allows you to preview the output without entering real data.

**FIGURE 22.19:**

The Graph1 example, based on the TeeChart component, at design time.




## Adding Data to the Chart

Now we simply initialize the data of the string grid and copy it to the series of the chart. This takes place in the handler of the `OnCreate` event of the form. This method fills the fixed items of the grid and the series names, then fills the data portion of the string grid, and finally calls the handler of the `OnClick` event of the *Update* button, to update the chart:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  with StringGrid1 do
  begin
    {fills the fixed column and row,
    and the chart series names}
    for I := 1 to 5 do
      Cells [I, 0] := Format ('Group%d', [I]);
```

```
    for J := 1 to 4 do
    begin
      Cells [0, J] := Format ('Series%d', [J]);
      Chart1.Series [J-1].Title := Format ('Series%d', [J]);
    end;

    // fills the grid with random values
    Randomize;
    for I := 1 to 5 do
      for J := 1 to 4 do
        Cells [I, J] := IntToStr (Random (100));
  end; // with

  // update the chart
  UpdateButtonClick (Self);
end;
```

We can access the series using the component name (as `Series1`) or using the `Series` array property of the chart, as in `Chart1.Series[J-1]`. In this expression, notice that the actual data in the string grid starts at row and column one—the first line and column, indicated by the zero index, are used for the fixed elements—while the chart `Series` array is zero-based.

Another example of updating each series is present in the `OnClick` event handler for the check box; this method toggles the visibility of the marks:

```
procedure TForm1.ChBoxMarksClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 4 do
    Chart1.Series [I-1].Marks.Visible :=
      ChBoxMarks.Checked;
end;
```

But the really interesting code is in the `UpdateButtonClick` method, which updates the chart. To accomplish this, the program first removes the existing data of each chart, and then it adds new data (or *data points,* to use a jargon term):

```
procedure TForm1.UpdateButtonClick(Sender: TObject);
var
  I, J: Integer;
begin
  for I := 1 to 4 do
  begin
    Chart1.Series [I-1].Clear;
    for J := 1 to 5 do
```

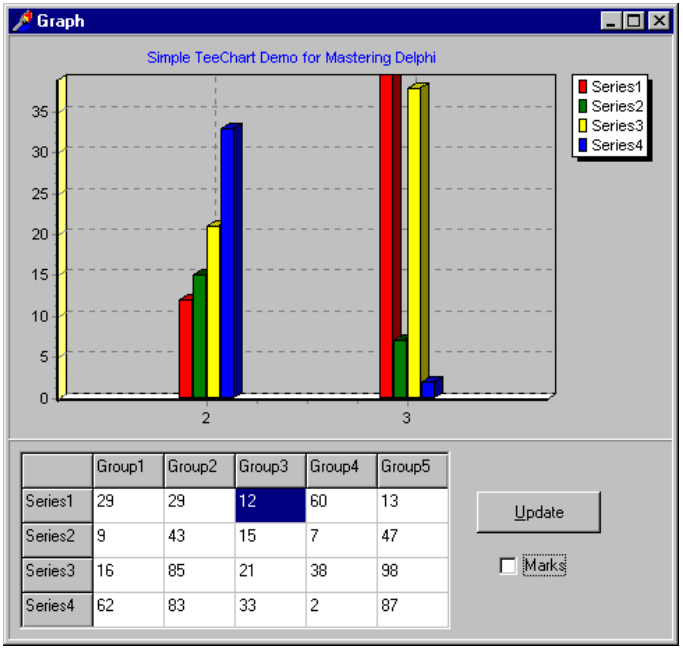
```
        Chart1.Series [I-1].Add (
          StrToInt (StringGrid1.Cells [J, I]),
          '', Chart1.Series [I-1].SeriesColor);
    end;
  end;
```

The parameters of the `Add` method (used when you don't want to specify an `X` value, but only an `Y` value) are the actual value, the label, and the color. In this example the label is not used, so I've simply omitted it. I could have used the default value, `clTeeColor`, to get the proper color of the series. You might use specific colors to indicate different ranges of data.

Once you've built the graph, TeeChart allows you a lot of viewing options. You can easily zoom into the view (simply indicate the area with the left mouse button), zoom out (using the mouse in the opposite way, dragging towards the top left corner), and use the right mouse button to pan the view. You can see an example of zooming in Figure 22.20.

**FIGURE 22.20:**

The form of the Graph1 example at run time. Notice that I've zoomed into the graph.




## Creating Series Dynamically

The Graph1 example shows some of the capabilities of the TeeChart component, but it is based on a single, fixed type of graph. I could have improved it by allowing

some customization of the shape of the vertical bars; instead I chose a more general approach, allowing the user to choose different kinds of series (graphs).

The TeeChart component initially has the same attributes as in the previous example. But the form now has four combo boxes, one for each row of the string grid. Each combo box has four values (Line, Bar, Area, and Point), corresponding to the four types of series I want to handle. To handle these combo boxes in a more flexible way in the code, I've added an array of these controls to the private fields of the form:

```
private
  Combos: array [0..3] of TComboBox;
```

This array is filled with the actual component in the `FormCreate` method, which also selects the initial item of each of them. Here is the new code of `FormCreate`:

```
// fill the Combos array
Combos [0] := ComboBox1;
Combos [1] := ComboBox2;
Combos [2] := ComboBox3;
Combos [3] := ComboBox4;
// show the initial chart type
for I := 0 to 3 do
  Combos [I].ItemIndex := 1;
```

**TIP**

This example demonstrates a common way to create an array of controls in Delphi, something Visual Basic programmers often long for. Actually Delphi is so flexible that arrays of controls are not built-in; you can create them as you like. This approach relies on the fact that you can generally associate the same event handler with different events, something that VB doesn't allow you to do.

All these combo boxes share the same `OnClick` event handler, which destroys each of the current series of the chart, creates the new ones as requested, and then updates their properties and data:

```
procedure TForm1.ComboChange(Sender: TObject);
var
  I: Integer;
  SeriesClass: TChartSeriesClass;
  NewSeries: TChartSeries;
begin
  // destroy the existing series (in reverse order)
  for I := 3 downto 0 do
    Chart1.Series [I].Free;
  // create the new series
  for I := 0 to 3 do
```

```
  begin
    case Combos [I].ItemIndex of
      0: SeriesClass := TLineSeries;
      1: SeriesClass := TBarSeries;
      2: SeriesClass := TAreaSeries;
    else // 3: and default
      SeriesClass := TPointSeries;
    end;
    NewSeries := SeriesClass.Create (self);
    NewSeries.ParentChart := Chart1;
    NewSeries.Title :=
      Format ('Series %d', [I + 1]);
  end;
  // update the marks and update the data
  ChBoxMarksClick (self);
  UpdateButtonClick (self);
  Modified := True;
end;
```

The central part of this code is the `case` statement, which stores a new class in the `SeriesClass` class reference variable, used to create the new series objects and set each one's `ParentChart` and `Title`. I could have also used a call to the `AddSeries` method of the chart in each case branch and then set the `Title` with another `for` loop. In fact, a call such as

```
Chart1.AddSeries (TBarSeries.Create (self));
```

creates the series objects and sets its parent chart at the same time.

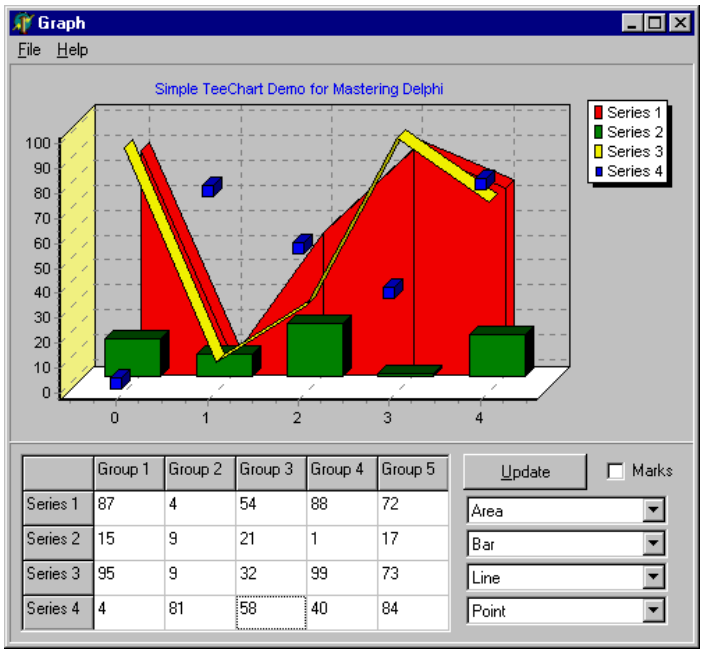
Notice that this new version of the program allows you to change the type of graph for each series independently. You can see an example of the resulting effect in Figure 22.21.

Finally, the Graph2 example has support for saving the current data it is displaying on a file and loads existing files. The program has a `Modified` Boolean variable, used to track whether the user has changed any of the data, and it prompts the user to confirm closing the form when the data has changed. The file support is based on streams and is not particularly complex, because the number of elements to save is fixed (all the files have the same size). Here are the two methods connected with the Open and Save menu items:

```
procedure TForm1.Open1Click(Sender: TObject);
var
  LoadStream: TFileStream;
  I, J, Value: Integer;
begin
  if OpenDialog1.Execute then
```

**FIGURE 22.21:**

Various kinds of graphs, or chart series, displayed by the Graph2 example




```
begin
  CurrentFile := OpenDialog1.Filename;
  Caption := 'Graph [' + CurrentFile + ']';
  // load from the current file
  LoadStream := TFileStream.Create (
    CurrentFile, fmOpenRead);
  try
    // read the value of each grid element
    for I := 1 to 5 do
      for J := 1 to 4 do
      begin
        LoadStream.Read (Value, sizeof (Integer));
        StringGrid1.Cells [I, J] := IntToStr(Value);
      end;
    // load the status of the checkbox and the combo boxes
    LoadStream.Read (Value, sizeof (Integer));
    ChBoxMarks.Checked := Boolean(Value);
    for I := 0 to 3 do
    begin
      LoadStream.Read (Value, sizeof (Integer));
      Combos [I].ItemIndex := Value;
    end;
  finally
```

```
      LoadStream.Free;
    end;
    // fire udpate events
    ChBoxMarksClick (Self);
    ComboChange (Self);
    UpdateButtonClick (Self);
    Modified := False;
  end;
end;

procedure TForm1.Save1Click(Sender: TObject);
var
  SaveStream: TFileStream;
  I, J, Value: Integer;
begin
  if Modified then
    if CurrentFile = '' then // call save as
      SaveAs1Click (Self)
    else
    begin
      // save to the current file
      SaveStream := TFileStream.Create (
        CurrentFile, fmOpenWrite or fmCreate);
      try
        // write the value of each grid element
        for I := 1 to 5 do
          for J := 1 to 4 do
          begin
            Value := StrToIntDef (Trim (
              StringGrid1.Cells [I, J]), 0);
            SaveStream.Write (Value, sizeof (Integer));
          end;
        // save check box and combo boxes
        Value := Integer (ChBoxMarks.Checked);
        SaveStream.Write (Value, sizeof (Integer));
        for I := 0 to 3 do
        begin
          Value := Combos [I].ItemIndex;
          SaveStream.Write (Value, sizeof (Integer));
        end;
        Modified := False;
      finally
        SaveStream.Free;
      end;
    end;
end;
```

## A Database Chart on the Web

In Chapter 20 of *Mastering Delphi 5,* we saw how to create a simple graphic image and return it from a CGI application. We can apply the same approach in returning a complex and dynamic graph built with the `TDBChart` component. Using this component in memory is a little more complex than setting all of its properties at design time, as you'll have to set the properties in the Pascal code. (You cannot use a visual component, such as a DBChart, in a Web Module or any other data module).

In the WebChart ISAPI application I've used the Country.DB table to produce a pie chart with the area and population of the American countries, as in the ChartDb example of Chapter 9 in *Mastering Delphi 5*. The two graphs are generated by two different actions, indicated by the paths `/population` and `/area`. As most of the code is used more than once, I've collected it in the `OnCreate` and `OnAfterDispatch` events of the WebModule.

**WARNING**

As written, this program doesn't support concurrent users. You'll need to add some threading or synchronization code to this ISAPI DLL to make it work with multiple users at the same time. An alternative is to place all the code in the Action event handlers, so that no global object is shared among multiple requests. Or you can turn it into a CGI application.

The data module has a table object, which is properly initialized at design time, and three private fields:

```
private
  Chart: TDBChart;
  Series: TPieSeries;
  Image: TImage;
```

The objects corresponding to these fields are created along with the Web module (and used by subsequent calls):

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
begin
  // open the database table
  Table1.Open;
  // create the chart
  Chart := TDBChart.Create (nil);
  Chart.Width := 600;
  Chart.Height := 400;
  Chart.AxisVisible := False;
  Chart.Legend.Visible := False;
  Chart.BottomAxis.Title.Caption := 'Name';
```

```
  // create the pie series
  Series := TPieSeries.Create (Chart);
  Series.ParentChart := Chart;
  Series.DataSource := Table1;
  Series.XLabelsSource := 'Name';
  Series.OtherSlice.Style := poBelowPercent;
  Series.OtherSlice.Text := 'Others';
  Series.OtherSlice.Value := 2;
  Chart.AddSeries (Series);
  // create the memory bitmap
  Image := TImage.Create (nil);
  Image.Width := Chart.Width;
  Image.Height := Chart.Height;
end;
```

The next step is to execute the handler of the specific action, which sets the pie chart series to the specific data field and updates a few captions:

```
procedure TWebModule1.WebModule1ActionPopulationAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  // set specific values
  Chart.Title.Text.Clear;
  Chart.Title.Text.Add ('Population of Countries');
  Chart.LeftAxis.Title.Caption := 'Population';
  Series.Title := 'Population';
  Series.PieValues.ValueSource := 'Population';
end;
```

This creates the proper DBChart in memory. The final step, again common to the two actions, is to save the chart in a bitmap image, and then format it as a JPEG on a stream, to be later returned from the server-side application. The code is actually similar to that of the previous example:

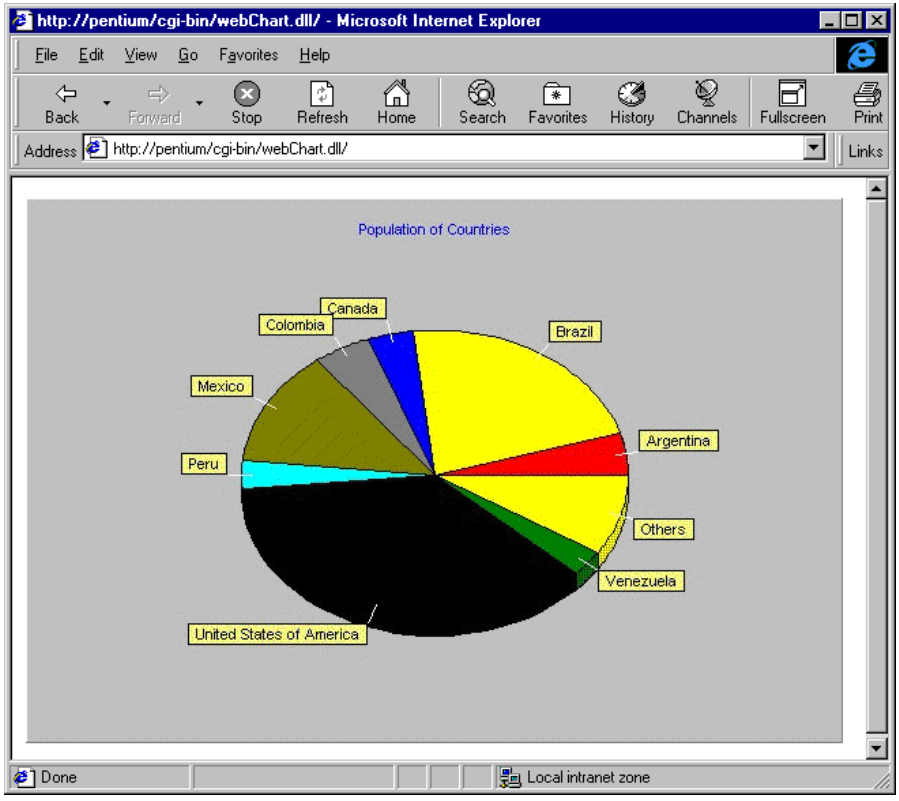
```
procedure TWebModule1.WebModule1AfterDispatch(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Jpeg: TJpegImage;
  MemStr: TMemoryStream;
begin
  // paint the chart on the memory bitmap
  Chart.Draw (Image.Canvas, Image.BoundsRect);
  // create the jpeg and copy the image to it
  Jpeg := TJpegImage.Create;
  try
```

```
    Jpeg.Assign (Image.Picture.Bitmap);
    MemStr := TMemoryStream.Create;
    // save to a stream and return it
    Jpeg.SaveToStream (MemStr);
    MemStr.Position := 0;
    Response.ContentType := 'image/jpeg';
    Response.ContentStream := MemStr;
    Response.SendResponse;
  finally
    Jpeg.Free;
  end;
end;
```

The result, visible in Figure 22.22, is certainly interesting. Optionally, you can extend this application by hooking it to an HTML table showing the database data. Simply write a program with a main action returning the HTML table and a reference to the embedded graphics, which will be returned by a second activation of the ISAPI DLL with a different path.

**FIGURE 22.22:**

The JPEG with the population chart generated by the WebChart application



# Using Metafiles

The bitmap formats covered earlier in this chapter store the status of each pixel of a bitmap, although they usually compress the information. A totally different type of graphic format is represented by vector-oriented formats. In this case the file stores the information required to re-create the picture, such as the initial and final point of each line or the mathematics that define a curve. There are many different vector-oriented file formats, but the only one supported by the Windows operating system is the Windows Metafile Format (WMF). This format has been extended in Win32 into the Extended Metafile Format (EMF), which stores extra information related to the mapping modes and the coordinate system.

A Windows metafile is basically a series of calls to the GDI primitive functions. After you've stored the sequence of calls, you can *replay* them, reproducing the graphics. Delphi supports Windows metafiles through the `TMetafile` and `TMetaFileCanvas` classes, so it's very simple to build an example.

The `TMetafile` class is used to handle the file itself, with methods for loading and saving the files, and properties determining the key features of the file. One of them is the `Enhanced` property, which determines the type of metafile format. Note that when Windows is reading a file, the `Enhanced` property is set depending on the file extension—WMF for Windows 3.1 metafiles and EMF for the Win32 enhanced metafiles.

To generate a metafile, you can use an object of the `TMetafileCanvas` class, connected to the file through its constructors, as shown by the following code fragment:

```
Wmf := TMetafile.Create;
WmfCanvas := TMetafileCanvas.CreateWithComment(
  Wmf, 0, 'Marco', 'Demo metafile');
```

Once you've created the two objects, you can paint over the canvas object with regular calls and, at the end, save the connected metafile to a physical file.

Once you have the metafile (either a brand-new one you've just created or one you've built with another program) you can show it in an Image component, or you can simply call the `Draw` or `StretchDraw` methods of any canvas.

In the WmfDemo example I've written some simple code, just to show you the basics of this approach. The `OnCreate` event handler of the form creates the enhanced metafile, a single object that is used both for reading and writing operations:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  Wmf := TMetafile.Create;
  Wmf.Enhanced := True;
  Randomize;
end;
```

The form of the program has two buttons and the PaintBox components, plus a check box. The first button creates a metafile by generating a series of partially random lines. The result is both shown in the first PaintBox and saved to a fixed file:

```
procedure TForm1.BtnCreateClick(Sender: TObject);
var
  WmfCanvas: TMetafileCanvas;
  X, Y: Integer;
begin
  // create the virtual canvas
  WmfCanvas := TMetafileCanvas.CreateWithComment(
    Wmf, 0, 'Marco', 'Demo metafile');

  try
    // clear the background
    WmfCanvas.Brush.Color := clWhite;
    WmfCanvas.FillRect (WmfCanvas.ClipRect);

    // draws 400 lines
    for X := 1 to 20 do
      for Y := 1 to 20 do
      begin
        WmfCanvas.MoveTo (15 * (X + Random (3)), 15 * (Y + Random (3)));
        WmfCanvas.LineTo (45 * Y, 45 * X);
      end;
  finally
    // end the drawing operation
    WmfCanvas.Free;
  end;

  // show the current drawing and save it
  PaintBox1.Canvas.Draw (0, 0, Wmf);
  Wmf.SaveToFile (ExtractFilePath (
    Application.ExeName) + 'test.emf');
end;
```

**WARNING**

If you draw or save the metafile before the connected metafile canvas is closed or destroyed, these operations will produce no effect at all! This is the reason I call the `Free` method before calling `Draw` and `SaveToFile`.

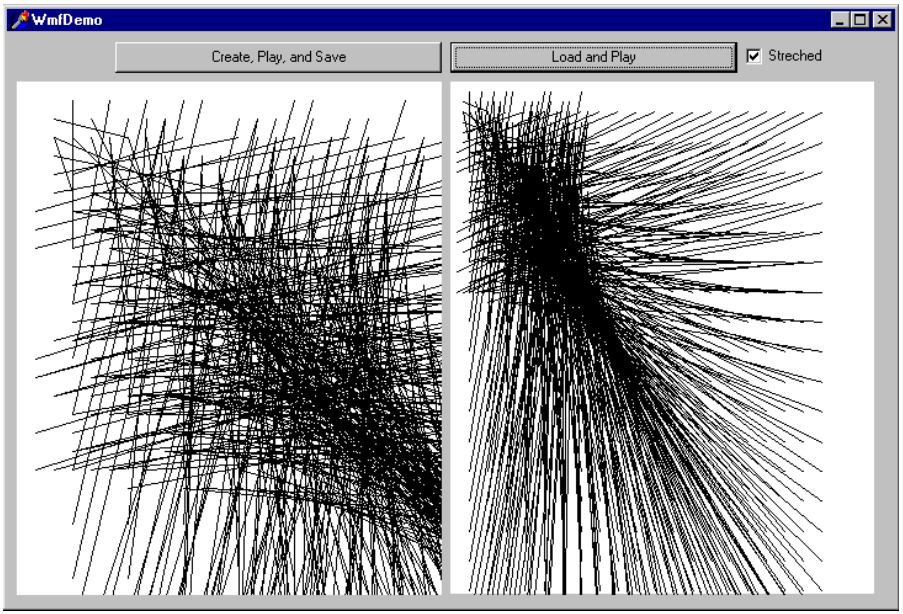
Reloading and repainting the metafile is even simpler:

```
procedure TForm1.BtnLoadClick(Sender: TObject);
begin
  // load the metafile
  Wmf.LoadFromFile (ExtractFilePath (
    Application.ExeName) + 'test.emf');

  // draw or stretch it
  if cbStretched.Checked then
    PaintBox2.Canvas.StretchDraw (PaintBox2.Canvas.ClipRect, Wmf)
  else
    PaintBox2.Canvas.Draw (0, 0, Wmf);
end;
```

Notice that you can reproduce exactly the same drawing but also modify it with the `StretchDraw` call. (The result of this operation is visible in Figure 22.23.) This operation is different from stretching a bitmap, which usually degrades or modifies the image, because here we are scaling by changing the coordinate mapping. This means that while printing a metafile, you can enlarge it to fill an entire page with a rather good effect, something very hard to do with a bitmap.

**FIGURE 22.23:**

The output of the WmfDemo with a stretched metafile.



# Rotating Text

In this chapter, we've covered a lot different examples of the use of bitmaps, and we've created graphics of many types. However, the most important type of graphics we usually deal with in Delphi applications is text. In fact, even when showing a label or the text of an Edit box, Windows still paints it in the same was as any other graphical element. I've actually presented an example of font painting earlier in this chapter in the FontGrid example. Now I'm getting back to this topic with a slightly more unusual approach.

When you paint text in Windows, there is no way to indicate the direction of the font: Windows seems to draw the text only horizontally. However, to be precise, Windows draws the text in the direction supported by its font, which is horizontal by default. For example, we can change the text displayed by the components on a form by modifying the font of the form itself, as I've done in the SideText example. Actually you cannot modify a font, but you can create a new one similar to an existing font:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ALogFont: TLogFont;
  hFont: THandle;
begin
  ALogFont.lfHeight := Font.Height;
  ALogFont.lfWidth := 0;
  ALogFont.lfEscapement := -450;
  ALogFont.lfOrientation := -450;
  ALogFont.lfWeight := fw_DemiBold;
  ALogFont.lfItalic := 0; // false
  ALogFont.lfUnderline := 0; // false
  ALogFont.lfStrikeOut := 0; // false
  ALogFont.lfCharSet := Ansi_CharSet;
  ALogFont.lfOutPrecision := Out_Default_Precis;
  ALogFont.lfClipPrecision := Clip_Default_Precis;
  ALogFont.lfQuality := Default_Quality;
  ALogFont.lfPitchAndFamily := Default_Pitch;
  StrCopy (ALogFont.lfFaceName, PChar (Font.Name));
  hFont := CreateFontIndirect (ALogFont);
  Font.Handle := hFont;
end;
```

This code produced the desired effect on the label of the example's form, but if you add other components to it, the text will generally be printed outside the visible portion of the component. In other words, you'll need to provide this type of support within components, if you want everything to show up properly. For

labels, however, you can avoid writing a new component; instead, simply change the font associated with the form's `Canvas` (not the entire form) and use the standard text drawing methods. The SideText example changes the font of the `Canvas` in the `OnPaint` method, which is similar to `OnCreate`:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  ALogFont: TLogFont;
  hFont: THandle;
begin
  ALogFont.lfHeight := Font.Height;
  ALogFont.lfEscapement := 900;
  ALogFont.lfOrientation := 900;
  ...
  hFont := CreateFontIndirect (ALogFont);
  Canvas.Font.Handle := hFont;
  Canvas.TextOut (0, ClientHeight, 'Hello');
end;
```

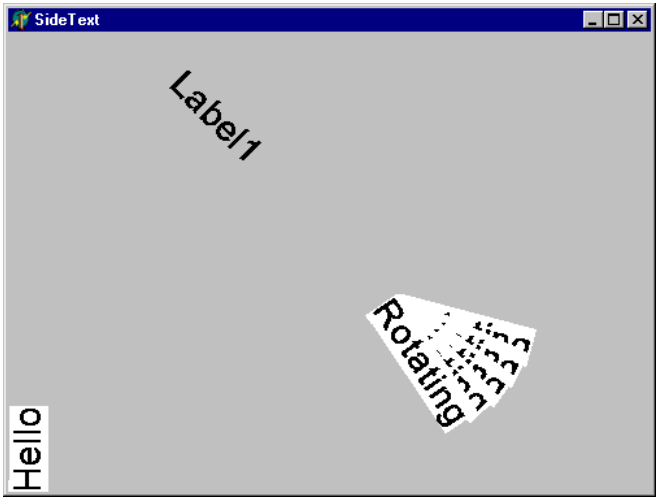
The font orientation is modified also by a third event handler, associated with a timer. Its effect is to rotate the form over time, and its code is very similar to the procedure above, with the exception of the code to determine the font *escapement* (the angle of the font rotation):

```
ALogFont.lfEscapement := - (GetTickCount div 10) mod 3600;
```

With these three different font rotating techniques (label caption, painted text, text rotating over time) the form of the SideText program at runtime looks like Figure 22.24.

**FIGURE 22.24:**

The effects of the SideText example, with some text actually rotating



# Where Do You Go from Here?

In this chapter, we have explored a number of different techniques you can use in Delphi to produce graphical output. We've used the `Canvas` of the form, bitmaps, metafiles, graphical components, grids, and other techniques. There are certainly many more techniques related with graphics programming in Delphi and in Windows in general, including the large area of high-speed games programming.

Allowing you to hook directly with the Windows API, Delphi support for graphics is certainly extensive. However, most Delphi programmers never make direct calls to the GDI system but rely instead on the support offered by existing Delphi components. This topic was introduced in Chapter 13 of *Mastering Delphi 5*.

If you've already read *Mastering Delphi 5*, I hope you've also enjoyed this extra bonus chapter. If you've started by this chapter, the rest of the book has plenty to offer, even in the context of graphics but certainly not only limited to that. Refer to `www.sybex.com` and `www.marcocantu.com` for more information about the book and to download the free source code of this chapter and of the entire book.