BONUS CHAPTER 1

Doing Graphics

Enhance the user interface

Use the Line and Shape controls to highlight areas of your forms, making your user interface more visually pleasing.

Want to show a picture?

The form itself, as well as several controls, enables you to display almost any type of picture. You can even display pictures that are stored in a database.

Change your pictures

With some of Visual Basic's methods you can even modify the pictures that you display.

Create your own pictures

Visual Basic's graphics methods enable you to create many types of graphics-related programs.

Analyze data with graphics

You can even use the graphics methods to create charts and other data analysis tools.

What do you think of when someone mentions graphics? Do you envision the artistic creations of a graphic designer? Do you conjure up images of the last sales presentation that you attended? Maybe you think of computer-aided design (CAD) or intricate data charts. The point is that the *graphics* topic encompasses a very wide range of images and applications.

Defined in the simplest terms, graphics (in the computer world) is the placement of lines, circles, points, and text in a specific pattern on a screen. These objects can be different sizes, shapes, and colors. The purpose of this chapter is to illustrate how to control the placement and characteristics of these objects.

The design and use of graphics is a large and complex subject. This single chapter cannot cover all the bases. However, it provides you with enough information and techniques to get you well on your way to creating great graphics programs. ■

Enhancing the User Interface

A typical user interface for an application consists of one or more forms containing a menu, labels, text boxes, command buttons, and perhaps a few other controls for specific pieces of data. However, without graphics, an otherwise functional interface can be quite boring and unintuitive. You can use graphics to enhance the user interface in the following ways:

- Highlighting specific information on the screen
- Providing a different view of the information
- Providing a more intuitive link to the application's functions

These enhancements can be accomplished through the use of the Line and Shape controls, color, pictures, and drawing methods.

Using the Line and Shape Controls

The Line and Shape controls provide the easiest means to add a graphic element to a form. The controls are drawn on the form at design time and placed where you need them. During the program's execution, these controls can be hidden or moved. Their colors can be changed by setting the appropriate property values in your code.

As you would guess by its name, the Line control places a line on the form. You can control the width of the line, the line style, the color, and the position of the terminal points of the line by setting the control's properties. Figure B1.1 shows several Line controls drawn on a form using the various styles and the `BorderStyle` property options for the Line control.

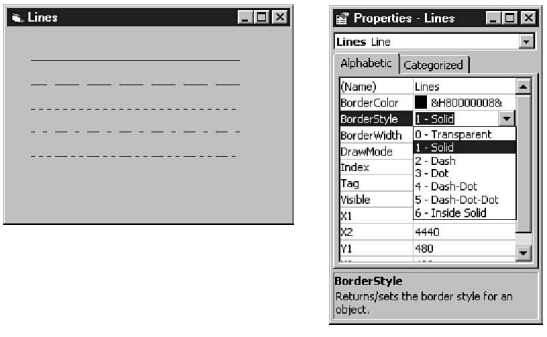


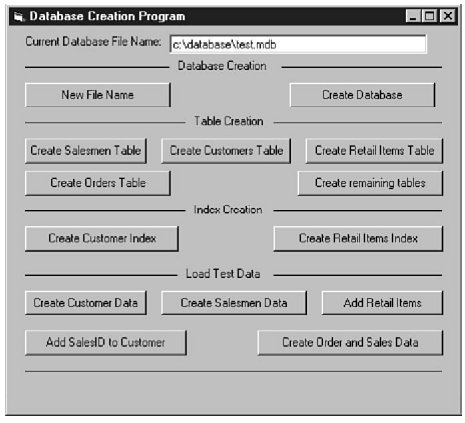
FIG. B1.1
The *Line* control enables you to place lines of different styles on a form. You assign the style by selecting the appropriate `BorderStyle` property.

> **CAUTION**
>
> If you set the `BorderWidth` property of the Line control greater than one, the `BorderStyle` property has no effect.

The Line control can be used on a form to separate areas of the form from one another. For instance, you might want to separate the data display portion of a form from the command

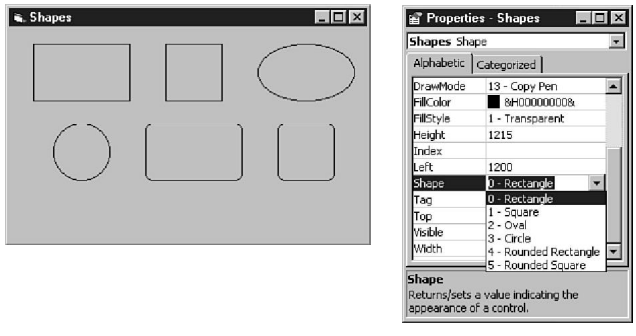
buttons, as shown in Figure B1.2. Or you may want to separate the information on the form into distinct groups. If you are presenting a lot of information on a form, the use of a line is a good way to enable the user to focus on one group of information at a time.

FIG. B1.2
The Line control is used to provide a visual separator between different areas on the form.



The Shape control provides another simple means of placing graphics elements on a form. You could use the Shape control to create any of six shapes. Simply change the control's Shape property (see Figure B1.3).

FIG. B1.3
You can change the Shape property of the Shape control to create any of the shapes pictured.



You can set the `BorderStyle` property of these shapes to any of the six line styles shown in Figure B1.1. You can also select patterns and colors for the shapes by setting the `FillStyle` and `FillColor` properties. The Shape control can be used to enclose various areas of a form and other controls, as shown in Figure B1.4.

NOTE Although a Shape control can visually enclose other controls, it cannot be used as a *container* for other controls, as the Frame and PictureBox controls can. If a frame is placed on a form, then other controls can be placed within the frame's borders. The frame and the controls contained within it then act as one unit. When the frame is moved, the other controls are moved with it, maintaining their relative position within the frame. If a frame is hidden, all controls in the frame are

continues

continued

also hidden. These characteristics are also true of a picture box. Not so with a Shape control. Any controls placed within the border of a Shape control are independent of the shape. If the shape is moved or hidden, the other controls remain unaffected.

▶ **See** "Exploring the Uses of Containers," in Chapter 13.

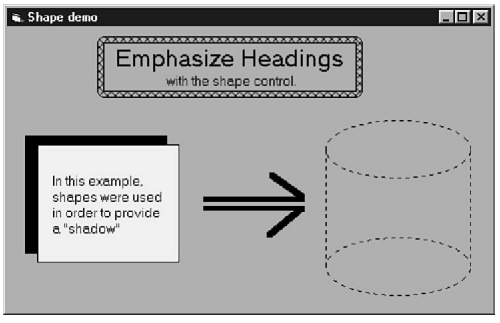


FIG. B1.4
The Shape control can be used to provide a border or other visual effects.

The Shape control can also be used to highlight information in a different manner. Suppose you want to draw your user's attention to the fact that he or she entered an incorrect value in a field. One way to do this is to draw a shape around the text box where the data was entered and to set its `Visible` property to `False`. Then place code to either show or hide the shape (depending on the value of the text) in the `Change` event procedure of the text box. The following code shows a shape when the text box's value is less than zero and hides the shape otherwise:

```
If Text1.Text < 0 Then
   Shape1.Visible = True
Else
   Shape1.Visible = False
End If
```

The `Change` event is triggered whenever the user starts typing in the text box. Therefore, if he or she starts to enter an incorrect value, the shape immediately appears. When an acceptable value is entered, the shape disappears. A good shape for this function is an oval with a `BorderWidth` of three and a red `BorderColor`.

Pictures on the Form

Another way to enhance your screens with graphics is to place pictures on the form. A *picture* is a bitmap file that can contain art, flow diagrams, or photos. The picture can be purely decorative, or it can be used to communicate specific information. You have probably seen pictures used for information in the setup screens of many programs. These pictures tell you about the features and benefits of the program while the installation is running. Visual Basic is capable of displaying many types of graphics files, which are summarized in Table B1.1.

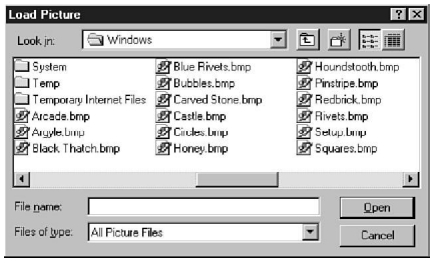
Table B1.1 Graphics Files Compatible with Visual Basic

File Extension	Type of File
.BMP	Windows bitmap.
.DIB	Bitmap file.
.ICO	Windows icon.
.WMF	Windows metafile.
.EMF	Enhanced Windows metafile.
.GIF	Graphics Interchange Format; a file format originally developed by CompuServe as a way to store raster (as opposed to vector) graphics images. Commonly used to store graphics on the Internet.
.JPG	JPEG images, named after the Joint Photographic Experts Group, developer of the format; similar to GIF images, but use compression algorithms to reduce file size. These are also used extensively on the Internet.

There are several ways to add a picture to a form. The picture can be placed on the form itself, or it can be placed in a PictureBox or Image control on the form. The advantages and disadvantages of each of these methods are addressed in the following paragraphs.

Loading a Picture on the Form The simplest way to add a picture to your screen is to add it to the form itself. You can do this at design time by setting the form's `Picture` property from the Properties window. To load a picture, select the `Picture` property, and then press the ellipsis (…) button at the far right of the line. This will call up the Load Picture dialog box shown in Figure B1.5. From this box, you can select the file containing the desired picture.

FIG. B1.5
To place a picture on a form at design time, the Properties window invokes the Load Picture dialog box. Pictures can also be loaded during program execution.



When a picture is loaded on the form, the entire picture is placed on the form, starting in the upper-left corner. If the picture is smaller than the form, space is left below or to the right of the picture. If the picture is larger than the form, the entire picture is still loaded, but only part of it is visible. As the form is resized, the amount of the picture shown changes.

NOTE The picture is always placed starting in the upper-left corner. It cannot be placed anywhere else on the form. If you want a small picture in another area of the form, you must use either a Picture or Image control, as explained in the next two sections. ■

When a picture is loaded on the form, it provides a background for the form. Any other controls added to the form appear on top of the picture. With the exception of the Label and Shape controls, the picture does not show through the background of the control. The Label and Shape controls allow the picture to show through if the `BackStyle` property of the control is set to `Transparent`. Figure B1.6 illustrates controls placed on a form containing a picture. Note the background of the controls.

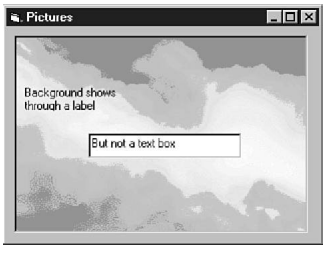


FIG. B1.6
Some controls placed on a form do not allow the picture to show through.

You can also add a picture to a form at run time. This is done by setting the `Picture` property of the form with the `LoadPicture` function as shown in the following code:

```
Form1.Picture = LoadPicture("C:\MYPICT.BMP")
```

You can also remove the picture from the form by specifying a null argument for the `LoadPicture` function:

```
Form1.Picture = LoadPicture("")
```

The key advantage to placing your picture directly on the form is that this method uses fewer system resources than placing the picture in a Picture or Image control. Another benefit of placing the picture on the form is that you can use drawing methods to annotate the picture. For example, you could display a product's picture and then, by using the `Print` method, overlay the price or other database information on top of the picture. This capability is available with the PictureBox control as well, but is not available with the Image control.

Placing the picture directly on the form does, however, have several drawbacks:

- You cannot hide the picture; it can only be loaded or unloaded.
- You cannot control the placement of the picture on the form.
- You can only place one picture on the form at a time.

These drawbacks can be overcome with the use of the PictureBox or Image control. However, the added flexibility comes at the expense of system resources.

Using the Image Control A second way of placing pictures on a form is to use an Image control. The Image control provides a frame for a bitmap or other picture on a form. The control can be placed anywhere on the form and drawn to any desired size. You assign a picture to the Image control by setting the `Picture` property of the control. This can be done either at design time using the Properties window or at runtime using an assignment statement:

```
Image1.picture = LoadPicture("C:\mypic.bmp")
Image2.picture = Image1.Picture
```

One other property of the Image control greatly affects the appearance of any pictures you may use. This is the `Stretch` property. The `Stretch` property determines whether the Image control is sized to fit the picture, or the picture is sized to fit the control as drawn. If the `Stretch` property is set to `False` (the default), the Image control is automatically resized to fit the picture you assign to it. If the `Stretch` property is set to `True`, the picture is automatically resized so that the entire picture fits within the current boundaries of the Image control. This setting causes the overall size of the picture to change and can change the aspect ratio of the picture (the ratio of vertical to horizontal size). Figure B1.7 shows the same picture in several Image controls. The one with the `Stretch` property set to `False` shows the image at its original size, while the others show some possible effects of resizing the Image controls that have their `Stretch` properties set to `True`.

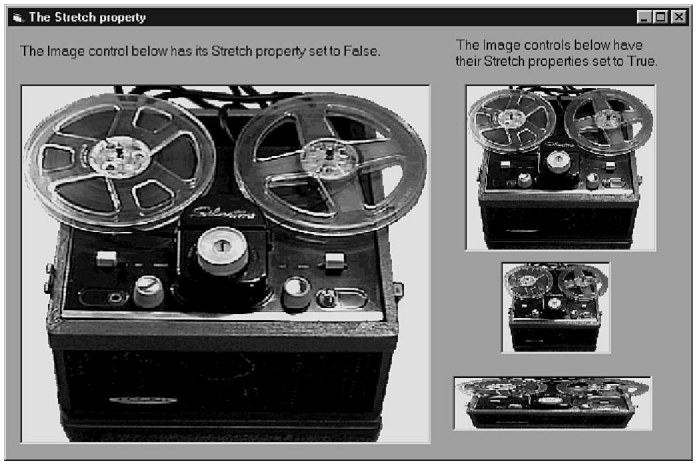


FIG. B1.7
The `Stretch` property of the Image control enables you to resize the picture to the size of the control or vice versa.

TIP If you are showing pictures of different sizes in your application, you should probably set the `Stretch` property to `True`. Otherwise, the size of the Image control will change with each picture and may cause the Image control to overlap other controls on the form. The Image control is anchored at its top-left corner, so only objects to the right and below the control would be affected. If the appearance of the pictures is unsuitable due to stretching, you may want to use the PictureBox control instead of the Image control.

The advantages of using the Image control instead of placing a picture directly on the form include the following:

- You can control the size of the picture's display area.
- You can place the picture anywhere on the form.
- You can place multiple Image controls on a form and use code to move them around.
- You can easily hide the picture using the `Visible` property of the Image control.

Using the Image control does have a few drawbacks, however:

- The Image control uses more resources than placing the picture directly on the form.
- You cannot use drawing methods to modify the picture in the Image control, as you can with a picture placed directly on the form or one in a PictureBox control.
- The Image control cannot serve as a container for other controls the way the PictureBox control can.

Using the PictureBox Control The third way to place a picture on a form is with the PictureBox control. As with the Image control, multiple picture boxes can be placed on a form. The PictureBox control uses the most resources of the three methods. Loading a picture in a PictureBox control is accomplished the same way as loading a picture on a form or into an Image control. The picture can either be loaded at design time or at run time.

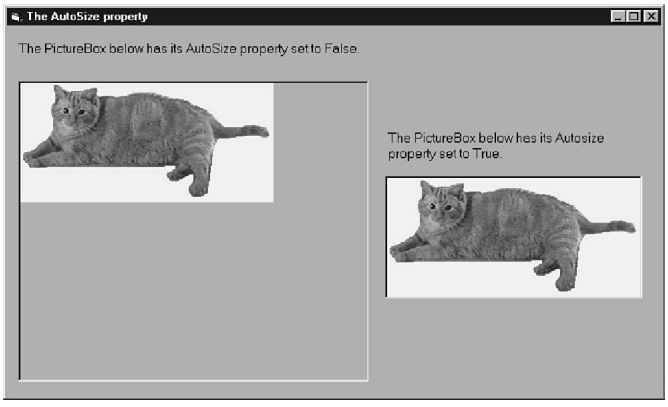
Like the Image control, the PictureBox control enables you to place a picture anywhere on a form and to size the control. However, resizing the PictureBox control does not have the same effect on the picture as resizing an Image control. The default behavior of the PictureBox control is to show only as much of a picture as will fit in its current boundaries. If the picture is larger than the PictureBox control, the upper-left corner of the picture is shown. If the picture is smaller than the PictureBox control, space is shown around the edges of the picture. In either case, the entire picture, displayed or not, is loaded in the PictureBox control and available if the control is resized. Setting the PictureBox's `AutoSize` property to `True` changes this default behavior, causing the PictureBox control to resize itself to fit the current picture. As with the Image control, the top-left corner of the control is anchored in place, and resizing of the control occurs to the right and down. However, don't confuse the `AutoSize` property with the Image control's `Stretch` property. The PictureBox control always preserves the aspect ratio of the picture being shown. Figure B1.8 shows the same picture in each of two `PictureBox` controls—one with the `AutoSize` property set to `False`, and the other with the `AutoSize` property set to `True`.

The PictureBox control also provides you with other capabilities that help you work with pictures. You can use the drawing methods to make changes to the picture, just as you can with the picture on a form. You can hide the picture or move it on the screen, just as you can with the Image control. But the PictureBox control also has some added benefits. Like the Frame control, the PictureBox control can be used as a container for other controls, so that any controls placed on the PictureBox control are treated as a unit with the PictureBox control itself. If the PictureBox control is hidden or moved, the other controls on it are also hidden or moved.

This feature allows the PictureBox control to be used to display multiple views or portions of data on a single form. For more detail on this feature, see "Exploring the Uses of Containers" in Chapter 10, "Using the Windows Common Controls."

▶ **See** "Exploring the Uses of Containers," in Chapter 13.

FIG. B1.8
The *AutoSize* property determines whether the PictureBox control will change size to fit the picture being displayed. Note that the size of the picture itself does not change.

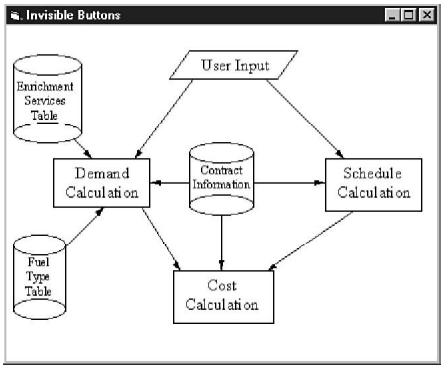


Invisible Buttons

As useful as it is to be able to display a picture on a form, you probably would like to be able to do more with pictures. One way to take advantage of the visual information displayed in a picture is to use it to control part of your application's interface.

For example, consider a flowchart displayed on a form that shows the various input files and calculations in a complex application (see Figure B1.9).

FIG. B1.9
A flowchart can be used to display information about an application.

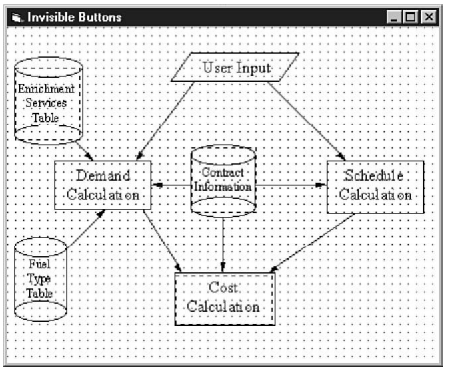


The flowchart helps the user understand how the information in the application is related. But suppose you could set up your application so that the user could access an input file merely by

clicking the file name in the flowchart. This can be done by superimposing Image controls over the flowchart to create invisible buttons.

Load a picture onto the form or into a PictureBox control. Wherever you want an invisible button, place an Image control over the region that you want to activate. Finally, place code in the `Click` event of each Image control to accomplish the task you want performed. The Image control is invisible because it has no border; and with no picture assigned to it, the background of the control is transparent. Figure B1.10 shows the flowchart from Figure B1.9 in design mode so that you can see the invisible buttons. During program execution, these controls would not be seen because their `BorderStyle` property is set to `None` (which is the default). Image controls are needed because the `Shapes` have no events of their own.

FIG. B1.10
Invisible buttons can make a picture part of your user interface.



This flowchart and invisible button application is contained in the file INVISBTN.VBP, which is on the companion CD-ROM.

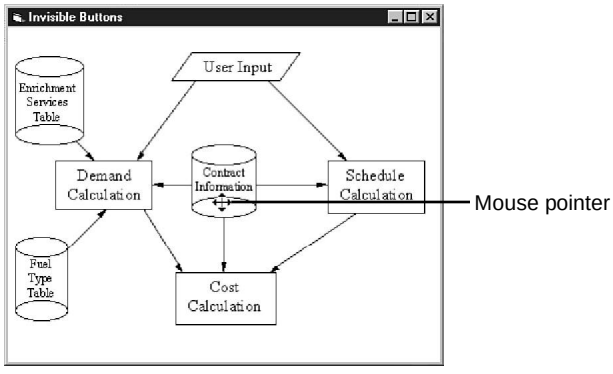
NOTE When the buttons are clicked, a message box appears to tell you which button or area you pressed. ■

Okay, since the buttons are invisible, how do you let the user know where they are and that he can click them? You can use a characteristic of the mouse pointer and the `MouseMove` event of the Image controls to change the mouse pointer when it is over one of the invisible buttons. Set the mouse pointer to an icon as shown in the following code:

```
Image1.MousePointer = vbCustom
Image1.MouseIcon =  LoadPicture("C:\myicon.ico")
```

This code changes the mouse pointer to an icon of your choosing for as long as the mouse is over the Image control. When the mouse is moved off of the control, the mouse pointer reverts to its original style. Figure B1.11 shows the invisible button with the changed mouse pointer over it.

FIG. B1.11
Changing the mouse pointer shows the user where the invisible buttons are located.



This invisible button concept can be used with all types of applications. For example, you can create a demonstration application for a new car. By placing invisible buttons on a picture of the car, you can enable the user to click the parts of the car to obtain information about its features—for example, the application can describe the car's engine capabilities if the user clicks the hood. The invisible buttons can be made any size you want them.

Creating and Managing Graphics

Earlier sections of this chapter have shown how you can use graphics to enhance the user interface through displayed pictures, invisible buttons, and other devices. To obtain the graphics used in these display elements, you can either use a graphics image from a library or create your own with a package such as Paintbrush.

But what if your application needs to be able to create graphics on its own? For example, certain data analysis programs create charts and programs for which the user might need a sketchpad for note taking. The following section discusses how you can create graphics images from within your application and how to store and manage the created graphics.

Creating Graphics

Visual Basic provides several tools that you can use to create graphics. The drawing methods can work on a form or in a PictureBox control. They can also be used with Visual Basic's `Printer` object to send the output to the printer. If no object is specified with the method, the form that is currently the focus will receive the output of the methods.

Visual Basic provides seven basic methods for creating graphics. These methods can be used to create many types of graphics images:

- `Line` method, which draws a line or a box on the target object
- `Circle` method, which draws a circle or oval on the target object
- `PSet` method, which places a single point on the target object

- `Point` method, which returns the color of a specific point
- `PaintPicture` method, which draws an image from another control onto the target object
- `Cls` method, which clears the output area of the target object
- `Print` method, which places text on the target object

NOTE In the following sections, we discuss these various graphics methods. For simplicity, code samples are given with the assumption that the target object is the form itself; therefore, the form name is left out of the syntax for invoking the method. For example, the first code sample below, `Line (1500, 750)-(2000, 750)`, draws a line on the form itself. However, the `Line` method can also apply to a PictureBox control. If you want to use one of these graphics methods on another object to which it applies, you can add the appropriate object name to the method syntax. To invoke the sample above on a PictureBox control named *Picture1*, you could use `Picture1.Line (1500, 750)-(2000, 750)`.

Using the *Line* Method You use the `Line` method to draw lines and boxes on the form. To draw a line, you need to provide the `Line` method with the starting and ending points of the line. If you omit the starting point, the method draws a line from the current position to the ending point. The following code draws a triangle on a form:

```
Line (1500, 750)-(2000, 750)
Line -(2000, 1250)
Line -(1500, 750)
```

As with a form's `Left` and `Top` properties, the coordinate system's origin is at the upper-left corner. The `Line` method enables you to specify the color used to draw the line. You can also use the `Line` method to draw a box on the form by including the optional `B` argument. In this case, the coordinates passed to the `Line` method specify the top-left and bottom-right corners of the box. This example shows how to draw a red box from the upper-left coordinate (2000, 2000) to the lower-right coordinate (2500, 2500):

```
Line (2000, 2000)-(2500, 2500), vbRed, B
```

When drawing a box, you can see the form behind it if the `FillStyle` property of the form is set to transparent. If you want the box to be filled, you can also specify the optional `F` argument, which fills the box with the same color used to draw the border. In the following example, you get a filled blue box:

```
Line -(3000, 3000), vbBlue, BF
```

In the preceding example, the starting point is omitted; therefore, the box is drawn from the current position to the ending point.

As you can see, the commands to draw lines and boxes are quite simple. The key to controlling the appearance of lines and boxes is setting the drawing properties of the form (or other object that is receiving the graphics). The graphics methods use the object's current settings when drawing. The properties that affect the graphics methods are summarized in Table B1.2. The effects of some of these properties are shown in Figure B1.12.

Table B1.2 Drawing Properties that Affect the Appearance of Graphics Drawn with the Graphics Methods

Property Name	Purpose
DrawMode	Determines how the color used to draw the border of the object interacts with objects already on the screen
DrawStyle	Determines the pattern used to draw the border of the object
DrawWidth	Determines the width of the line used to draw the border of the object
FillColor	Determines the color used to fill an object
FillStyle	Determines the fill pattern used to fill an object
ForeColor	Determines the primary color used in drawing the border of an object

The values of these properties are explained in Visual Basic's Help system. All of these properties can be set for a form or PictureBox control at design time. However, they are most useful when they are set at run time, when they can be set for a single drawing operation. The following code segment stores the form's drawing properties in variables, draws a series of boxes, and then returns the properties to their original settings:

```
frmset1 = Form1.DrawStyle
frmset2 = Form1.DrawWidth
frmset3 = Form1.FillColor
frmset4 = Form1.FillStyle
frmset5 = Form1.ForeColor
Line (1000, 1000)-(1500, 1500), , B
Form1.DrawStyle = 2
Form1.FillStyle = 2
Line -(2000, 2000), , B
Form1.FillColor = &hff
Line -(2500, 2500), , B
Form1.DrawWidth = 3
Form1.ForeColor = &hff0000
Line -(3000, 3000)
Form1.DrawStyle = frmset1
Form1.DrawWidth = frmset2
Form1.FillColor = frmset3
Form1.FillStyle = frmset4
Form1.ForeColor = frmset5
Line (100, 100)-(500, 500), , B
```

Figure B1.12 displays the appearance of the form after executing this code sample.

Using the *Circle* Method The `Circle` method allows you to draw circles, ellipses, arcs, and pies on the form. The simplest form of the method is the following:

```
Circle (X, Y), R
```

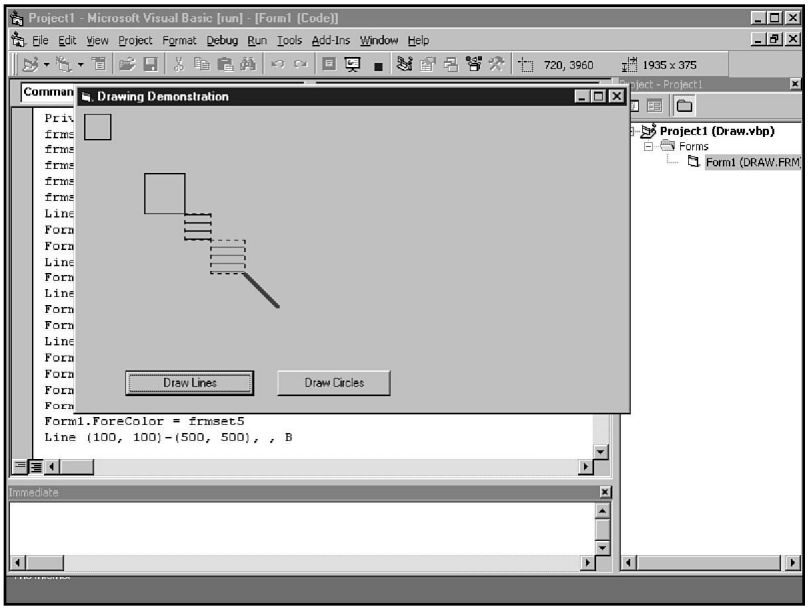


FIG. B1.12
The Line method can be used to draw lines and boxes on an object such as a form or printer.

This command draws a circle of radius R with a center at the position specified by X and Y. As with the `Line` method, the pattern and color of the circle's border and fill are determined by the settings of the drawing properties. The full syntax of the `Circle` method is as follows:

```
object.Circle [Step] (x, y), radius, [color, start, end, aspect]
```

Several optional arguments can be used with the `Circle` method. See the Visual Basic help system for the complete syntax. You use the `start` and `end` arguments of the method to draw an arc rather than a full circle. The values of `start` and `end` are the angles from horizontal expressed in radians. (The value of an angle in radians is determined by multiplying the angle in degrees by π/180.) The values of `start` and `end` can range from zero to 2π, or zero to –2π. If the values of both the `start` and `end` arguments are negative, the method draws a pie (an arc with lines extending to the center of the circle). If both values are positive, a simple arc is drawn. If one of the values is negative, a line is drawn from that end of the arc to the center.

Another argument that can be specified is the *aspect* argument. This argument is used to draw an ellipse or an oval. The aspect is the ratio of the vertical size of the ellipse to the horizontal size. An aspect of one draws a circle. If the aspect is greater than one, the ellipse is taller than it is wide. If the aspect is less than one, the ellipse is wider than it is tall. In all cases, the radius specified in the method sets the size of the longer dimension. The code for drawing several shapes with the `Circle` method is the following. (The results of this code are shown in Figure B1.13.)

```
'Circle
Circle (500, 500), 500
'Arc
```

```
Circle (1500, 1500), 500, , 1.57, 0
'Piece of pie
Circle (2500, 2500), 500, , -4.7, -6.2
'Ellipse
Circle (3500, 3500), 500, , , , 1.5
'Another piece of pie
Circle (4500, 4500), 500, , -0.01, -1.57, 1.5
```

NOTE In the previous code, there's a "missing" argument, which is after the radius but before the `start` and `end` arguments. This place is reserved for the optional `color` argument. If this argument is not present, the color used to draw the circle is the `ForeColor` property of the object on which it's drawn. Notice also that even though the color argument is left out, its place is "saved" by an extra comma.

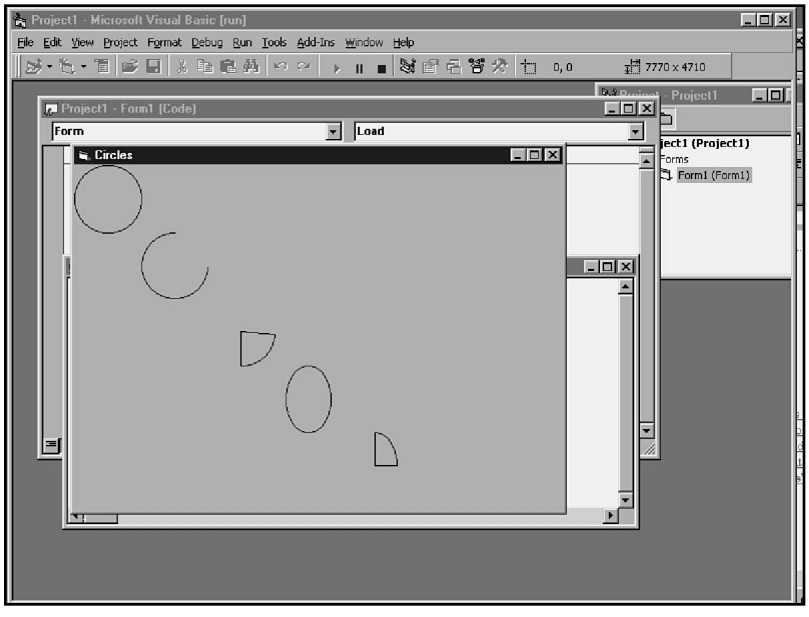


FIG. B1.13
The Circle method has several optional parameters that allow it to draw ellipses, arcs, and pies.

Using the *PSet* Method The `PSet` method is used to draw a single point on the form using the color specified by the `ForeColor` property. The size of the point drawn is dependent on the setting of the `DrawWidth` property. A larger `DrawWidth` setting produces a larger point. The `PSet` method draws the point at the coordinates specified in the argument of the method. The following code will draw a point at position 100, 100:

```
PSet (100, 100)
```

One use of the `PSet` method is to provide a freehand drawing capability for the users of your application. This use is covered in the later section "Creating a Sketchpad Application."

Using the *PaintPicture* Method The `PaintPicture` method enables you to place all or part of a picture from one object into a specific location in another object. Also, by carefully setting the

height and width of the source and target objects, you can enlarge or reduce the size of the source picture. The following code paints part of a picture from a PictureBox control named `picSource` onto the base form:

```
frmdest.PaintPicture picSource.Picture, 50, 50, 750, 750, 0, 0, 500, 500
```

As you can see from the line of code, the `PaintPicture` method contains several arguments:

- *Source picture* All or part of this source picture is drawn in a "target region" on the destination object.
- *Target X and Y* These two numerical arguments specify the coordinates of the upper-left corner of the target region. Therefore, in the previous example, drawing occurs on frmDest 50 twips from the left edge and 50 twips from the top edge.
- *Target Size* Specifies the horizontal and vertical size of the target region. If this size is different from the size of the source picture or region, the picture is stretched or compressed to fit the target region.
- *Source X and Y* The third pair of numbers in the command specifies the upper-left corner of the source region—that is, the part of the picture being copied.
- *Source Size* Specifies the height and width of the source region.

The first three arguments are the only ones required for the `PaintPicture` method. All other arguments are optional. If only the three required arguments are specified, the entire source picture is copied to the target at full size.

The code shown in this listing takes a piece of a picture from the PictureBox control, enlarges it by 50 percent, and places it on the form. The results of this command are shown in Figure B1.14.

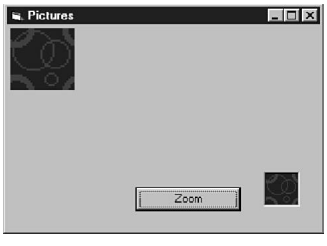


FIG. B1.14
The `PaintPicture` method can copy all or part of a picture from one object to another and enlarge or reduce the picture.

Some uses of the `PaintPicture` method in creating graphics would be the following:

- To provide a zoom feature for looking more closely at specific regions of a picture. This feature would be useful for implementing print preview in your application.
- To make multiple copies of a picture (or a portion of one) on a target object.
- To clear a specified region of a picture or combine several pictures.
- To print the contents of a PictureBox control—for example, to put a company logo on your report.
- To be able to create data point markers for a chart.

Using the *Print* Method Although the Print method is not usually thought of as a graphics method, it works the same way. Its primary use is to place text on a form, PictureBox control, or the Printer object. The Print method can be used in conjunction with the graphics methods to create charts or drawings or to annotate existing bit maps. The Print method itself is quite simple. The following code displays a single line of text at the current position on the form:

```
Print "This is a one-line test."
```

Remember, these code examples assume that the graphics methods are directed to the current form. To print the line of text in the previous sample to the printer, you could invoke the Printer object's Print method:

```
Printer.Print "This is a one-line test."
```

The output of the Print method is controlled by the settings of five properties of the object being printed on. These properties are as follows:

- CurrentX This sets the horizontal position for the starting point of the text.
- CurrentY This sets the vertical position for the starting point of the text.
- Font This determines the font type and size used for the text.
- ForeColor This determines the color of the text.
- FontTransparent On a form or picture, this determines whether or not the background behind the text will show through the spaces in the text.

Other Methods The other two graphics methods mentioned at the beginning of this section are the Point method and the Cls method. These two methods each perform a single function. The Point method returns the RGB color setting of a single specified point. The Cls method, which stands for "Clear Screen" (a carryover from DOS-based programming languages that needed to clear the entire screen before writing more information), clears all graphics drawn with the graphics methods from a form, PictureBox control, or Image control. The Cls method has no effect on any *controls* that are on the object. It only clears *graphics* that were drawn at run time.

Creating a Sketchpad Application

To further illustrate how the different graphics methods work, this section discusses how to create a sketchpad application. This application is similar to the familiar Windows Paintbrush application. The starting form for the application is shown in Figure B1.B1.

This sketchpad application only creates bit map (.BMP) drawings. After being created, any object placed on the screen is handled simply as a series of points. The objects cannot be selected later to be resized or moved. In contrast to this application, programs like Microsoft PowerPoint save information about the size, type, location, and other characteristics of an object. This feature allows the object to be re-selected and those characteristics to be modified in order to change the object's appearance at a later time.

Setting Up the Toolbar The sketchpad application uses Image controls to create a toolbar, which enables the user to select which type of object to draw. The toolbar is set up by placing

three arrays of Image controls on the form. Two of the arrays are used to store the image of the "up" button and the "down" position for each tool. The third image array acts as the actual toolbar. When the button for a tool is pressed, two results occur. First, the down image of the button is displayed to indicate which tool was selected. Second, a variable is set to tell the program which drawing tool to use. The sketchpad application provides the user with the capability to draw the following six objects:

- Line
- Open Box
- Filled Box
- Open Circle
- Filled Circle
- Freehand Sketch

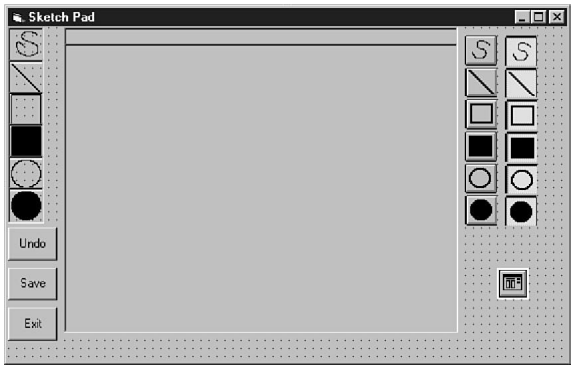


FIG. B1.15
You can create a Paintbrush-style application by using the drawing methods.

You use the Image control array to make programming simpler. The index of the control array defines the tool used to draw on the PictureBox control. An example of the code for setting the Image control pictures and the tool type is shown here. The variable `inToolNum` is declared in the declarations section of the form and is initially set to zero, the number for the Freehand tool:

```
Private Sub imgToolBar_Click(Index As Integer)
   'Reset the button for the previously used tool to the Up position
   imgToolbar(inToolNum).Picture = imgUp(inToolNum).Picture

   'Set the button for the newly selected tool to the Down position
   imgToolBar(Index).Picture = imgDown(Index).Picture

   'Set the inToolNum variable to the new tool
   inToolNum = Index
End Sub
```

Using the Various Drawing Tools The sketchpad enables the user to press the mouse button and then drag the mouse to draw an object, just like VB controls or any Windows drawing

program. The sketchpad application provides a PictureBox control on which the user may draw.

▶ **See** "Introducing Events," in Chapter 7.

To implement the various drawing tools, you need to work with three events: `MouseDown`, `MouseMove`, and `MouseUp`. These events correspond with the user's actions as he or she is drawing an object.

The MouseDown event is responsible for telling the program that the user is drawing on the PictureBox control and for setting the initial position of the object. The purpose of this initial point depends on the type of object being drawn. Table B1.3 shows the purpose of the initial point for each of the objects in the sketchpad application.

Table B1.3 Initial Point of a Drawing Operation Has Different Meanings for Each Object

Object	Use of Initial Point
Line	One of two points defining the line
Box (open or filled)	One of the corners of the box
Circle (open or filled)	One corner of a box which would bound the circle
Freehand	The first point drawn

A variable has been defined to tell the program whether to draw objects while the mouse is in motion. This variable, `DrawNow`, has a value of either `True` or `False`. The `MouseDown` event sets this variable to `True`. The following code is placed in the `MouseDown` event to enable the drawing methods. In the code, the variables `curX` and `curY` are the coordinates of the initial point. The variables `oldX` and `oldY` are the coordinates of the last mouse position. These variables are important in the `MouseMove` event, as you will see:

```
Private Sub PicMain_MouseDown(Button As Integer, Shift As Integer, _
     X As Single, Y As Single)
   DrawNow = True
   curX = X
   curY = Y
   oldX = X
   oldY = Y
End Sub
```

The `MouseMove` event is the main workhorse of the sketchpad application. If the mouse button is down (`DrawNow` is set to `True`), the code in the `MouseMove` event draws the selected object between the initial point of the drawing and the current mouse position. The event contains a set of cases to handle the various types of objects that might be drawn. You might notice that the open and filled boxes and the open and filled circles use the same code. This is because in the `MouseMove` event, it is only necessary to show the outline of a filled object while the drawing is in progress.

You might also notice that the `Line` method is used for the Freehand drawing instead of the `PSet` method. The reason for this is that the `MouseMove` event is triggered at certain intervals, not as a continuous event. Rapid movements of the mouse leave gaps in the lines drawn with the `PSet` method. When the `Line` method is used, a line is drawn between the last position of the mouse and the current position, thereby providing a continuous line. The code for the `MouseMove` event is shown in Listing B1.1.

Listing B1.1 MouseMove.Frm—The *MouseMove* Event Draws the Outline of the Object as the Mouse Is Dragged

```
Private Sub PicMain_MouseMove(Button As Integer, Shift As Integer, _
       X As Single, Y As Single)
If DrawNow Then
   Select Case inToolNum
      Case 0 'Freehand drawing tool
         PicMain.Line (oldX, oldY)-(X, Y)
         oldX = X
         oldY = Y
      Case 1 'Lines
         PicMain.Line (curX, curY)-(oldX, oldY), PicMain.BackColor
         PicMain.Line (curX, curY)-(X, Y)
         oldX = X
         oldY = Y
      Case 2,3 'Open and filled boxes
         PicMain.Line (curX, curY)-(oldX, oldY), PicMain.BackColor, B
         PicMain.Line (curX, curY)-(X, Y), , B
         oldX = X
         oldY = Y
      Case 4,5 'Open and filled circles
         cntX = curX + Int((X - curX) / 2)
         cntY = curY + Int((Y - curY) / 2)
         radX = Abs(curX - X)
         radY = Abs(curY - Y)
         radcir = IIf(radX > radY, radX, radY) / 2
         If radX = 0 Then
             aspcir = 1
         Else
             aspcir = radY / radX
         End If
         PicMain.Circle (oldX, oldY), oldrad, PicMain.BackColor, , , _
             oldasp
         PicMain.Circle (cntX, cntY), radcir, , , , aspcir
         oldX = cntX
         oldY = cntY
         oldasp = aspcir
         oldrad = radcir
   End Select
End If
End Sub
```

When the user is finished drawing and releases the mouse button, the MouseUp event creates the final drawing for the object and turns off the drawing mode (see Listing B1.2). Much of the same code is used in the MouseUp event as in the MouseMove event, but separate cases have been added for the filled boxes and circles.

Listing B1.2 MouseUp.Frm—The *MouseUp* Event Renders the Final Drawing and Turns Off the Drawing Mode

```
Private Sub PicMain_MouseUp(Button As Integer, Shift As Integer, _
     X As Single, Y As Single)
If DrawNow Then
   Select Case inToolNum
      Case 1  'Line
         PicMain.Line (curX, curY)-(oldX, oldY), PicMain.BackColor
         PicMain.Line (curX, curY)-(X, Y)
      Case 2  'Open Box
         PicMain.FillStyle = vbFSTransparent
         PicMain.Line (curX, curY)-(oldX, oldY), PicMain.BackColor, B
         PicMain.Line (curX, curY)-(X, Y), , B
      Case 3  'Filled Box
         PicMain.FillStyle = vbFSSolid
         PicMain.Line (curX, curY)-(oldX, oldY), PicMain.BackColor, B
         PicMain.Line (curX, curY)-(X, Y), , B
         PicMain.FillStyle = vbFSTransparent
      Case 4  'Open Elipse
         cntX = curX + Int((X - curX) / 2)
         cntY = curY + Int((Y - curY) / 2)
         radX = Abs(curX - X)
         radY = Abs(curY - Y)
         radcir = IIf(radX > radY, radX, radY) / 2
         If radX = 0 Then
             aspcir = 1
         Else
             aspcir = radY / radX
         End If
         PicMain.FillStyle = vbFSTransparent
         PicMain.Circle (oldX, oldY), oldrad, PicMain.BackColor, , , _
            oldasp
         PicMain.Circle (cntX, cntY), radcir, , , , aspcir
      Case 5  'Filled elipse
         cntX = curX + Int((X - curX) / 2)
         cntY = curY + Int((Y - curY) / 2)
         radX = Abs(curX - X)
         radY = Abs(curY - Y)
         radcir = IIf(radX > radY, radX, radY) / 2
         If radX = 0 Then
             aspcir = 1
         Else
             aspcir = radY / radX
         End If
```

continues

Listing B1.2 Continued

```
            PicMain.FillStyle = vbFSSolid
            PicMain.Circle (oldX, oldY), oldrad, PicMain.BackColor, , , _
               oldasp
            PicMain.Circle (cntX, cntY), radcir, , , , aspcir
            PicMain.FillStyle = vbFSTransparent
      End Select
   End If
   DrawNow = 0
   End Sub
```

Creating an Undo Feature Most applications that perform drawing functions have an Undo feature that allows the user to restore the picture to its state prior to the last drawing operation. An Undo function can be implemented in the sketchpad application by adding a second PictureBox control to the form, placing some additional code in the MouseDown event, and adding an Undo button. The second PictureBox control contains a copy of the image in the PictureBox control that contains the "primary" drawing. This second PictureBox control has its Visible property set to False so that it is not visible when the application is running. The second picture is updated each time a new drawing operation is started. This update is performed in the MouseDown event as shown in the following code:

```
picUndo.Picture = picMain.Image
```

NOTE In the previous code, a picture was copied to a picture box's Picture property by assigning to the property another picture box's Image property. The Image property was used because it not only contains the loaded picture, but also the results of any drawing operations. ■

To undo an operation, the code simply copies the picture in the second PictureBox control back to the main PictureBox control, returning the drawing area back to the way it was before the last drawing operation. This code is shown in the following line:

```
picMain.Picture = picUndo.Picture
```

Saving the Picture Finally, you should give your users the capability of saving the pictures they create. The drawings on a PictureBox control can be saved as a bit map file using the SavePicture function. This function requires the name of the source picture and the name of the output file. You probably want to use the CommonDialog control to enable the user to specify a name for the output file. For a drawing, the source of the picture is the Image property of the PictureBox control or the form on which the drawing was made. The following code gets a file name using the CommonDialog control (named GetFile) and stores the drawing created by the sketchpad application:

```
GetFile.Filter = "Bitmap Files (*.BMP)|*.bmp"
GetFile.DefaultExt = "BMP"
GetFile.ShowSave
DataName = GetFile.FileName
SavePicture PicMain.Image, DataName
```

Bit Map Annotation

The sketchpad application showed how you can create graphics with the graphics methods and how to store the graphics in files. Because the PictureBox control is also capable of displaying an existing graphics file, you can use the sketchpad program to modify graphics from other sources. You might want to use this program to annotate fax images before sending them on to another person. You can also use the sketchpad to make changes to bit maps created by others.

To annotate bit maps, you need to add a `LoadPicture` function to the sketchpad application so that you can import the picture into the editing area. The code for this operation is shown here. This code again uses the CommonDialog control to obtain the name of the file to be edited:

```
stTemp = "Bitmap Files (*.BMP)|*.bmp|Icon Files | *.ico |All Files|*.*"
GetFile.Filter = stTemp
GetFile.DefaultExt = "BMP"
GetFile.ShowOpen
DataName = GetFile.FileName
PicMain.Picture = LoadPicture(DataName)
```

Using a Database to Store Pictures

There are two ways to use pictures with databases. The most obvious is to store a *pointer* to the name of the picture file in the database. However, an Access database created in Visual Basic can store pictures in the database itself. The field type to use for pictures is a *long binary* field. If the database is bound to a Data control, the field containing the pictures can be bound to either a PictureBox control or an Image control. If a PictureBox control is used, the drawing methods discussed in this section can be used to create or edit the pictures in the database.

It is important to know how to save the changes to the pictures back into the database. A problem arises in saving these changes because the data field containing the picture is bound to the `Picture` property of the PictureBox control. The drawings that are made with the graphics methods are not part of the `Picture` property, but rather part of the `Image` property. It is therefore necessary to copy the `Image` property to the `Picture` property prior to the update of the database, which you can do with the following line of code:

```
PicMain.Picture = PicMain.Image
```

Once the drawing or annotations have been copied to the `Picture` property, the drawing is stored in the database. An application that requires a number of sketches can benefit from using this technique to store the drawings. When you use the database, all the drawings are stored in one file rather than a series of bitmap files. It is also possible to add other fields to the database that contain a description of the drawing and possibly supporting information or notes.

CAUTION

If you use a great number of pictures in a database, it may grow to an unmanageable size!

Analyzing Data with Graphics

This final topic in the chapter deals with analyzing data or information with the use of graphics or charts. Often charts are very useful in giving users a better feel for the information than numbers alone can give. The percentage of your household budget devoted to debt reduction is presented far more dramatically by a pie chart than by the mere presentation of dollar amounts. Charts can also help a user spot trends in the data that would not be possible from viewing only the numeric data.

Generating charts with the graphics methods is much more difficult than using the Chart control, but there are some advantages to the effort:

- You can create multiple charts on the same PictureBox control (such as a bar chart for regional sales by month and a pie chart for total sales by region).
- You can place multiple axes on the same chart, enabling you to plot multiple variables (for example, a chart of engine temperature and coolant pressure versus speed).
- You can create charts that change with time.
- Graphics methods do not have the overhead of distributing a custom control.
- You can superimpose the chart on another graphic for special effects.

This section discusses the general programming aspects of creating your own charts and then looks in detail at some of the advantages mentioned in the list.

Creating a Simple Chart

To begin the discussion, consider the pie chart shown in Figure B1.16. To create this chart in a program, you follow these steps:

1. Calculate the total value of all the points.
2. Convert the value of each point to a fraction of the total.
3. Convert the fractional value to the radian values of a circle.
4. Set the `FillColor` property for each point.
5. Draw the pie shape for each of the points.

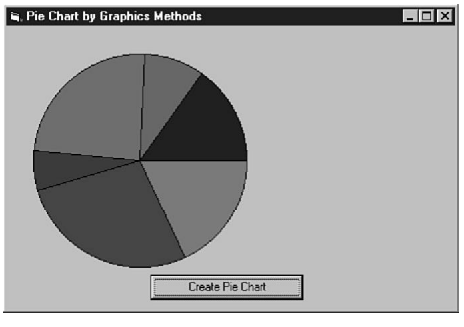


FIG. B1.16
By using the `Circle` method to draw arcs, you can create a pie chart.

Listing B1.3 demonstrates how you can perform these steps for the graph illustrated in Figure B1.16.

Listing B1.3 ChartFrm—Creating a Pie Chart

```
Sub CreatePieChart()
    Const SLICES = 6
    Const PIEX = 2000
    Const PIEY = 2000
    Const PIERADIUS = 1500
    Dim inData(1 To SLICES) As Integer
    Dim inCounter As Integer
    Dim inTotal As Integer
    Dim lgStartPoint As Single
    Dim lgEndPoint As Single
    Dim lgPercent As Single
    Dim lgSliceSize As Single


    'First, we create sample data points
    'with random numbers from 1 to 20
    For inCounter = 1 To SLICES
        inData(inCounter) = Int(20 * Rnd + 1)
    Next inCounter

    'Next, we'll add all the data points
    'together to get a total value.
    For inCounter = 1 To SLICES
        inTotal = inTotal + inData(inCounter)
    Next inCounter

    'Finally, we calculate each data point's
    'percent of the total value and plot it.
    frmMain.FillStyle = vbFSSolid
    lgStartPoint = -0.001
    For inCounter = 1 To SLICES
        lgPercent = inData(inCounter) / inTotal
        lgSliceSize = lgPercent * 2 * 3.14159
        lgEndPoint = lgStartPoint - lgSliceSize
        lgEndPoint = IIf(lgEndPoint < -6.2831, -6.2831, lgEndPoint)
        frmMain.FillColor = QBColor(inCounter)
        frmMain.Circle (PIEX, PIEY), PIERADIUS, , lgStartPoint, lgEndPoint
        lgStartPoint = lgEndPoint
    Next inCounter

End Sub
```

This is a somewhat generic routine for creating a pie chart. In a real program, of course, the data would not be generated randomly. Notice also, the `QBColor` function is used to assign a color to a particular data point. QBColor (QB stands for "QuickBasic," which was Visual

Basic's granddaddy) converts integer color codes used by older versions of BASIC into an equivalent hexadecimal color code that can be used by Visual Basic. I just used it here because the data is random and the color doesn't matter anyway.

What Methods to Use for Different Chart Types

As you saw previously, a pie chart can be created using the `Circle` method. Most of the chart types described for the graphic control can be easily ("easy" being a relative term) created with the graphics methods. Each graphic type uses one or more of the graphics methods to create the charts. Table B1.4 shows several of the most common chart types and the methods used to create them.

Table B1.4 GraphicChart.frm—Different Graphics Methods Are Used to Create Different Types of Charts

Chart Type	Graphics Method
Pie	`Circle`
Bar	`Line` (drawing boxes)
Gantt	`Line` (drawing lines or boxes)
Line	`Line`
Scatter	`PSet`, `PaintPicture`, `Print`
High-Low-Close	`Line`

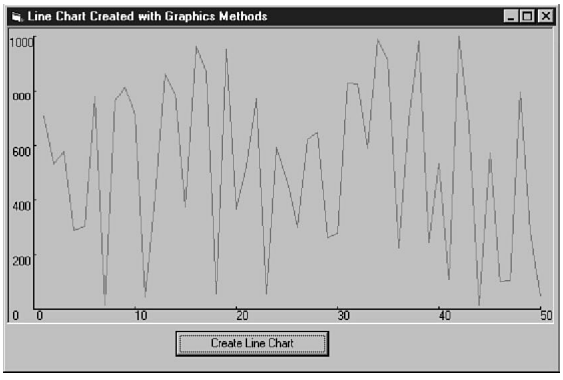
Symbols can be drawn on any of the chart types using the `PaintPicture` method; or, if the symbols are simple characters, the `Print` method can be used to place the character on the chart. You may also notice that the chart indicates that the `Print` method can produce scatter charts. This process works the same way as placing symbols on the chart. You establish the necessary position of the symbol using the `CurrentX` and `CurrentY` properties and then print the symbol.

Determining Where to Place Points on the Chart

The previous section, "Creating a Simple Chart," showed how to use the `Circle` method to create a pie chart. The code used to produce the chart assumed that the size of the form and the position of the chart were predetermined. This may not always be the case. If you are drawing a chart on a form whose size can change, you need to calculate the position of the chart on the form. In addition, for many chart types (such as line or bar), you need to establish the range of the horizontal and vertical coordinates that are available for showing data.

This section walks you through this process using a line chart containing 50 data points with a random value of 0 to 1,000 as an example. (The program code for the chart is shown in Listing B1.3.) The actual chart is shown in Figure B1.17.

FIG. B1.17
The chart pictured was created by graphics methods rather than a custom control.



The first step in determining where the data for the chart should be placed is to find the size of the form or PictureBox control that can contain the chart. Horizontal and vertical size are determined using the `ScaleWidth` and `ScaleHeight` properties, respectively. These properties determine the maximum space available for the output of the chart.

Next, you determine the amount of space needed for labels on the two axes. This space is handled by the `TextHeight` and `TextWidth` methods. You will want the maximum length of all the labels on the chart. For this sample, the Y-axis ranges from 0 to 1,000 in increments of 200. Therefore, the maximum `TextWidth` would be for 1,000. Along the X-axis, the values range from 1 to 50 in increments of 10, but all the values should have the same text height. The `TextWidth` determines the margin between the edge of the output object (form or picture box) and the Y-axis. Similarly, the `TextHeight` gives the margin between the bottom of the output object and the X-axis. If you also want a margin at the top and right of the chart, you have to establish values for these as well. The code for the sample chart uses half the `TextWidth` and `TextHeight` margins for the right and top margins, respectively.

Subtracting the margin sizes from the total size of the object gives you the size of the area where you will draw the line graph. The size of this area determines the scaling factor that you need to use to place data points. For the Y-axis values, this is the height of the drawing area divided by the maximum value. For the X-axis value, the scaling factor is the width of the drawing area divided by the maximum X value. To obtain the drawing position of any point you need to perform the following steps:

1. Multiply the X value by the horizontal scaling factor to determine the distance from the Y-axis.
2. Add the distance obtained in tep 1 to the width of the left margin to obtain the actual X position on the output object.
3. Multiply the Y value by the vertical scaling factor to determine the distance from the X-axis.
4. Subtract the distance obtained in tep 3 from the position of the X-axis to obtain the actual Y position on the output object.

You might notice that the last step instructs you to subtract the value obtained in Step 3 from the position of the X-axis. The reason for this step is that the vertical position coordinates of an object increase from top to bottom. Therefore, a point above the X-axis has a smaller number for the vertical position than the axis itself (see Listing B1.4).

Listing B1.4 GraphicChrt.Frm—Creating a Chart with Graphics Methods

```
Sub CreateLineChart()
    Dim inCounter As Integer
    Dim inMaxX As Integer
    Dim inMaxY As Integer
    Dim inLmarg As Integer
    Dim inRmarg As Integer
    Dim inBmarg As Integer
    Dim inTmarg As Integer
    Dim inScaleX As Integer
    Dim inScaleY As Integer
    Dim inYPos As Integer
    Dim inXPos As Integer

    picChart.ForeColor = vbBlack
    picChart.Cls

    'Determine maximum size of chart
    inMaxX = picChart.ScaleWidth
    inMaxY = picChart.ScaleHeight

    'Determine chart margins, including
    'width for the axis labels
    inLmarg = picChart.TextWidth("1000")
    inBmarg = picChart.TextHeight("50")
    inRmarg = inMaxX - 0.5 * inLmarg
    inTmarg = 0.5 * inBmarg
    inBmarg = inMaxY - inBmarg

    'Determine scale factors for each axis
    inScaleX = (inRmarg - inLmarg) / 50
    inScaleY = (inBmarg - inTmarg) / 1000

    'Draw axes
    picChart.Line (inLmarg, inTmarg)-(inLmarg, inBmarg)
    picChart.Line -(inRmarg, inBmarg)

    'Draw labels and tic marks for vertical axis
    For inCounter = 1 To 6
        picChart.CurrentX = 5
        inYPos = inBmarg - ((inCounter - 1) * 200 * inScaleY)
        picChart.CurrentY = inYPos
        picChart.Print Right(Str((inCounter - 1) * 200), 4)
        picChart.Line (inLmarg, inYPos)-(inLmarg + 40, inYPos)
    Next inCounter

    'Draw labels and tic marks for horizontal axis
    For inCounter = 1 To 6
```

```
        inXPos = inLmarg + ((inCounter - 1) * 10 * inScaleX)
        picChart.CurrentX = inXPos
        picChart.CurrentY = inBmarg + 5
        picChart.Print Right(Str((inCounter - 1) * 10), 2)
        picChart.Line (inXPos, inBmarg)-(inXPos, inBmarg - 40)
    Next inCounter

    'Draw Random Points
    picChart.ForeColor = vbRed
    For inCounter = 1 To 50
        inXPos = inLmarg + (inCounter * inScaleX)
        inYPos = inBmarg - (1000 * Rnd * inScaleY)
        If inCounter = 1 Then
            picChart.CurrentX = inXPos
            picChart.CurrentY = inYPos
        Else
            picChart.Line -(inXPos, inYPos)
        End If
    Next inCounter

End Sub
```

Dynamic, or Time-Dependent, Graphs

One of the advantages of creating your own data analysis charts is that you can create a chart that changes with time. To create this *dynamic*, or *time-dependent, chart*, you need a way to add points to the chart at specified intervals. An example you may have seen is the Windows NT Performance monitor, which can be used to provide a timed graph of CPU usage. There are two ways to handle plotting time-dependent information. You can track all the information—adding new points but never removing old points—or you can track some number of points representing the most recent measurements (for instance, the last 100 points).

Tracking a number of recent points is usually the preferable method of developing a dynamic chart. The advantages of this method are that you have a limited number of points that keep system resource requirements down, and you do not have to constantly recalculate the scale factors to account for additional points.

The following sample code segments build a dynamic chart based on our previous example. The same code is used to draw the axes and set up the chart. First, move the code from Listing B1.3 to a separate procedure called `SetupChart`. Next, remove the last `For` loop, which draws all the points. Add the lines

```
SetupChart
inCounter = 1
```

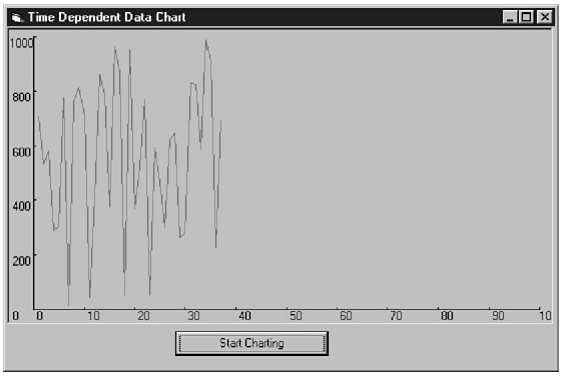
to the form's `Load` event. Finally, move all the variable declarations from `SetupChart` to the form's general declarations section so that they are visible to other procedures.

The new code needed is a function that draws one segment of the line graph at a time. This function can then be placed in a Timer control's Timer event to fill the chart gradually:

```
Private Sub Timer1_Timer()

    inXPos = inLmarg + (inCounter * inScaleX)
    inYPos = Int(inBmarg - (1000 * Rnd * inScaleY))

    If inCounter = 1 Then
        picChart.CurrentX = inXPos
        picChart.CurrentY = inYPos
    Else
        picChart.Line -(inXPos, inYPos)
    End If

    inCounter = inCounter + 1

    If inCounter > 50 Then SetupChart

End Sub
```

Notice in the code above that when the 50th data point is reached, the graph simply starts over from the left. Figure B1.18 illustrates the chart created by this code.

FIG. B1.18
You can create a dynamic chart that changes as new points are added.



From Here...

In this chapter, you learned about which controls and objects can be used to add pictures to a program. You also learned how to use some low-level graphics to draw on the screen or printer. These methods can be used for creating visual effects or informational displays, such as charts.

- For a more detailed discussion of the PictureBox control's capabilities, Chapter 10, "Using the Windows Common Controls."
- To learn more ways to add visual appeal to your forms, see Chapter 19, "Designing User Interfaces."
- To find out how to use these enhancements in a manner that is useful to the end user, see Chapter 25, "Extending ActiveX Controls."