Chapter 7

Graphics

G O A L S

After working through this chapter, you will be able to:

- Understand and use the typesetting and computer display concepts associated with monitor and hardcopy displays.
- Describe and use the various built-in coordinate systems provided in the SetMode property of the picture box and the form object.
- Initialize and use user-defined coordinate systems appropriate to application programs.
- Use the **PSet** method to draw points whose coordinates are stored in an array.
- Use the **Line** method to draw lines.
- Use the **Circle** method to draw circles and ellipses of various size at various positions.
- Use the color functions of Visual Basic.

O V E R V I E W

In this chapter you will see that the "Visual" in "Visual Basic" doesn't just refer to the programming environment. You will use picture boxes and graphics commands to help programs communicate. You will learn about coordinate systems, experiment with color, and draw shapes on forms.

Pixels and Twips

Graphics are images, including everything from line drawings to video clips and animation. Adding graphics to your programs makes them fun to run and visually appealing. People like to see moving objects on the screen, so videos and animation hold a special fascination.

Graphics have an educational purpose as well, because you can use them to convey information quickly. The old adage "a picture is worth a thousand words" is not far from the truth.

To use graphics in Visual Basic, you need to understand pixels and twips. And to understand these terms, you need to look back at how graphic systems have evolved. Graphic systems are the hardware and software used to create and present graphics. Until quite recently, these systems were not powerful enough to display crisp images or videos on screen.

Evolving Graphics Systems

Pixels are the building blocks of everything you see on screen – text as well as graphics. The term "pixel" is short for picture element. It is the smallest graphic element of a display on screen. If you look at your computer's display from a short distance—say, less than 1 foot—you can see individual pixels distinctly. In Figure 7-1, a single pixel has been removed from the dot of the letter i. The pixel is small enough that you would not notice the missing pixel unless the words were magnified.

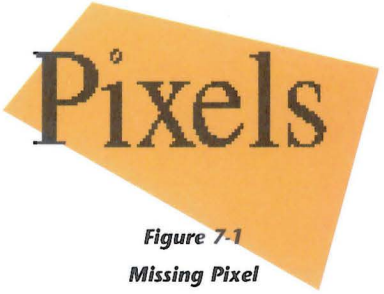


Figure 7-1
Missing Pixel

The graphical capability of a machine is measured by counting the number, size, and range of colors of the pixels that are displayed on the screen, as well as the speed at which the screen can be redrawn. These capabilities are determined mostly by the video card installed in the machine and the monitor on which the card's output appears. The CPU and the architecture of the machine itself affect only the speed of redrawing. The number of pixels—also called the resolution—determines the amount of detail that that can be shown, and the smoothness of lines.

For example, imagine that only a very small number of pixels could be displayed: not much information could be shown. With such a low resolution, only horizontal and vertical lines would look smooth; all other straight lines, and all curves, would appear jagged. The size of the pixels is determined by the resolution, together with the capabilities of the monitor. If the monitor displays pixels that are too big, they "bleed" into each other and give the image a blurry appearance with inaccurate colors.

The number of possible colors that a pixel can have determines the realism of the displayed images. The greater the possible number of colors, the more photographic are the images that can be displayed. With only 16 colors, photographs cannot be accurately represented. However, 256 colors are sufficient for displaying convincing real-world images. The number of colors offered by current video cards ranges from 16 to 16.7 million. Most cards let you select from among several combinations of resolution and number of colors.

EARLY MACHINES

The earliest computers were primarily "text only," so the number of pixels did not matter so much. People used computers to perform tasks like word processing, writing payroll checks, and deciphering codes. "Graphics" was a term used in art studios, not computer rooms. A few systems tried to use graphic displays for serious work, with poor results.

Yesterday's computers were too slow to manipulate graphic images. The more pixels the system could display, the more work it had to do. You cannot display graphics if the system is so slow that you can type faster than the computer can respond.

As computers have become more powerful, graphical capability has improved. Computers today have more and smaller pixels, more colors, and the ability to move large amounts of graphic information in very short times. The screen on a typical home computer today displays anywhere from 640 by 480 pixels to 1024 by 768 pixels or more. Years ago, 256 by 192 seemed like a lot of pixels.

TODAY, THE TWIP

All graphics systems are based on pixels because the pixel is the smallest hardware element of a display screen. The pixel, then, is still the building block used to create images. Now that computers can handle graphics, however, variations in pixel count and color matter more than before. An image could look great on an artist's computer, for example, with a high-resolution monitor. On a monitor with far fewer pixels, though, that same image could look too small, too large, too blurry, or too plain.

You can also see that this variation presents some difficulty to you as a programmer. Suppose that you write code to draw a 30-pixel line on the screen. This line would appear longer on some screens, shorter on other ones. Without a standard number of pixels per screen, you cannot use the pixel as the basis for a drawing system.

The answer is a "logical" rather than a physical basis for a graphics system. This is the twip. For Visual Basic, the twip is the default standard

measure of graphic elements. To a lot of people who have grown up with computers, the term "twip" sounds like a new, made-up word. Twips, however, have been around for a very long time.

When mechanical printing was the norm, type was set using metal blocks, each with a raised letter. The blocks were set in troughs, words separated with spacing blocks, lines separated with long thin spaces called slugs. The trays of letters were inked and pressed against paper.

An extensive vocabulary grew up around the printing presses—a vocabulary still used today, even though most of the old presses are gone. Some of these old words have already been used in the text; font is one of them. Two other words from the old printing press days are points and twips.

Fonts, Points, and Twips

A font is a style of type. When you choose a font, you also choose the size of the font. That size is designated in points. A printer's point, equal to 1/72 of an inch, is the standard of measure for fonts. See Figure 7-2 for an example of fonts and font sizes. A twip is a twentieth of a point. (In fact, "twip" is an abbreviation of the phrase "twentieth of a point.") Thus, there are 1440 twips in one inch. A 12-point font is 12 points x 20 twips/point, or 240 twips high.

When a program draws a straight line of 1440 twips on a printed page, the length of the line is precisely 1 inch. The same line drawn on the screen will typically be longer than 1 inch. The line on the screen measures 1 logical inch. The logical inch is the size of the screen representation of images that occupy 1 inch on a printed page. You may have noticed that programs such as Microsoft Word and Windows Write display a ruler along the top of your documents. These rulers allow you to set margins, columns, and tab stops so that pages will have the appearance you want. However, if you were to measure 1 "inch" as indicated by one of these on-screen rulers, you would find that its length is somewhat longer than 1 inch. (Exactly how long it is depends on the screen resolution you are using.)

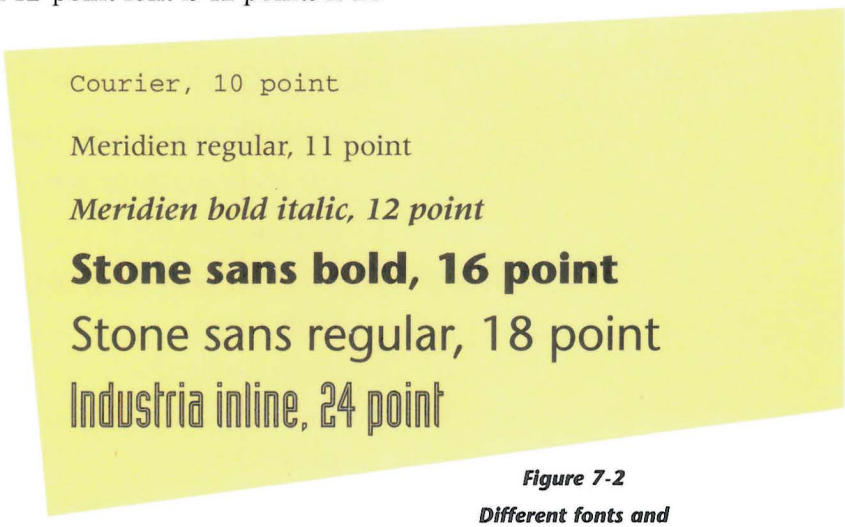


Figure 7-2
Different fonts and font sizes

The logical inch magnifies physical inches. Without this magnification, point sizes that we commonly use for printing would be unusably

small on the screen. Two factors determine the need for on-screen magnification. The first is that we typically view computer monitors from a distance of about 2 feet, whereas we view printed pages at somewhat closer range. The second is that the resolution of the printed page is vastly higher than that of a computer screen. Laser printers print anywhere from 300 to 1200 dpi (dots per inch), and high-quality magazines are printed at resolutions of 2400 dpi and higher. By contrast, there are usually only 96 pixels per logical inch in both the horizontal and vertical dimensions of a screen.

When an image is displayed, Windows still translates twips to pixels. The Screen object in Visual Basic has two read-only properties, TwipsPerPixelX and TwipsPerPixelY, which tell you precisely how the translation is performed. The translation adjusts the final image to maintain the proportions of the original image. The actual adjustment made—the number of pixels per logical inch, in both the vertical and the horizontal dimension—depends on the graphics system. A system with fewer pixels across the screen translates the image differently than a system with more pixels.

In other words, by measuring in twips, you can now instruct a program to draw a line of a specific length. The line will be that length, regardless of the number of pixels displayed on the user's monitor.

QUESTIONS AND ACTIVITIES

1. List some applications that can operate without graphics.
2. List some applications that require graphics.
3. What is a pixel?
4. What is a twip? How big is a twip? Where does the word "twip" come from?
5. What is a font?
6. What is a point? How big is a point? Where does the word point come from?
7. How has the increased speed of computers changed the nature of computer applications over the years?
8. Windows uses two quantities, LOGPIXELSX and LOGPIXELSY, to determine how to scale physical lengths in each dimension for display on the screen. These are the number of pixels per logical inch, in each dimension. Write Visual Basic code that uses **Screen**.TwipsPerPixelX and **Screen**.TwipsPerPixelY to compute these numbers and print them on a form.

Picture Box and Image Box

You have already experimented with drawing freehand in a form. In the previous chapter, you placed circles and lines on a form. You can also import images into a Visual Basic form. These images may have been created in another Windows application. Or you may want to take advantage of images such as clipart that are available publicly.

To import images into a form, you need to place them in a control. Visual Basic provides two controls for this purpose: a picture box and an image box. As you experiment with these two controls in this section, you will see that they have slightly different properties. You can stretch and distort a picture in an image box, for example. You cannot do that with the same picture in a picture box.

Before importing images, you should become familiar with the different types of graphics files. You can tell the type of file from the file extension. Three of the common types of files have these extensions:

- **.bmp** (bitmap)
- **.ico** (icon)
- **.wmf** (Word metafile)

A bitmap is a graphic image stored as a collection of pixels. The pixels are represented in memory as bits or collections of bits. An icon is a small graphic usually used to represent a concept or object. It is limited in size to 32 by 32 pixels. A metafile saves a graphic image as a collection of drawing objects: lines, circles, and colors.

You can find bitmap, icon, and metafiles in the Windows, MSOffice, and VB directories. You should also check your directories for collections of clipart and other graphics.

Importing Images into Visual Basic Controls

Follow these steps to explore the picture box and image box controls.

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Change the caption of the form to **The Picture and Image Box Demo**.
3. Click on the picture box icon in the Toolbox.

4. Draw the picture box on the form. Click where you want the top-left corner of the box, then drag to the lower-right corner. See Figure 7-3.

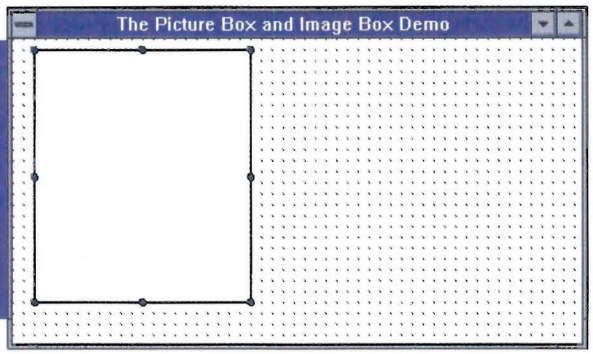


Figure 7-3
Placing a picture box on the form

5 Click on the image box icon in the Toolbox.

6 Draw an image box on the form using the same method as in step 4.

7 To insert an icon or bitmap into a picture box, select the picture box, then bring the Properties window to the front. Double-click on the Picture property of the box. The Load Picture dialog box opens (see Figure 7-4).
This figure shows the computer subdirectory of the icons directory in the Visual Basic directory.

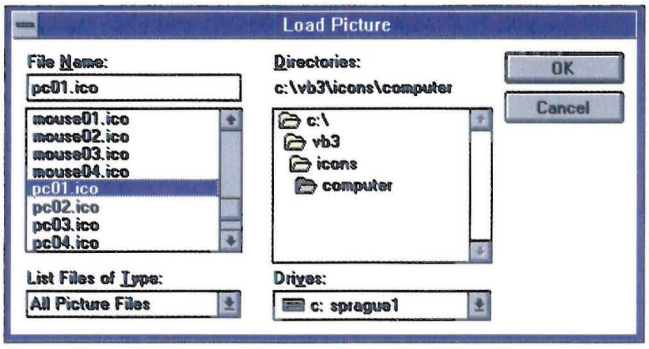


Figure 7-4
Load Picture dialog box, for selecting a graphic to insert in the picture box

8 Select a file. The graphic is then displayed in the picture box. Press Enter or click OK.
In this example, we selected the icon file **pc01.ico**, as shown in Figure 7-4 above. It is shown displayed in Figure 7-5. Feel free to pick a different file, such as **\VB\bitmaps\assorted\happy.bmp**.

9 Select the image box, then bring the Properties window to the front. Double-click on the Picture property. Add the same graphic to the image box.
The boundaries of the image box shrink to fit the icon.

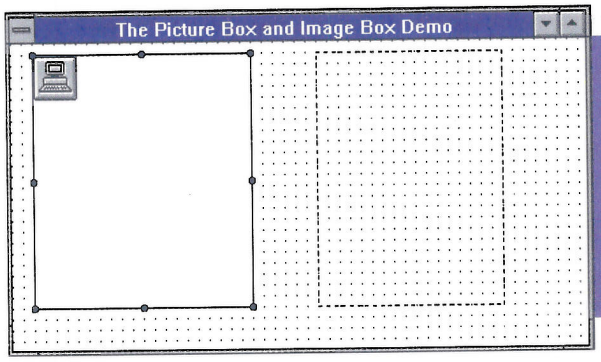


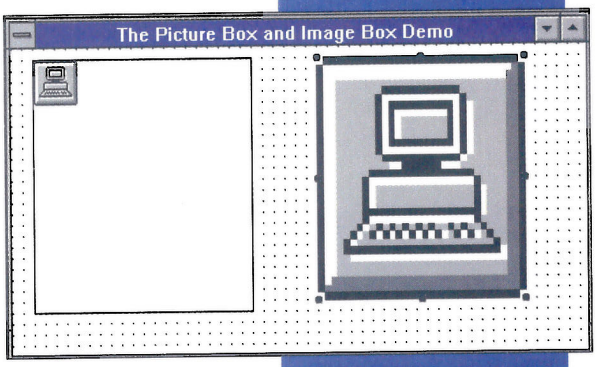
Figure 7-5
Displaying a graphic in the picture box

Experimenting with the Images: Stage 1

With the pictures loaded, you are ready to experiment with the properties of the two different controls:

1. Select the image box. Change the Stretch property of the image box to True.
2. Try resizing the image box several times. The icon stretches to fill the box. For an example, see Figure 7-6.

Figure 7-6
Stretching a graphic in an image box



3. Click on the shape control in the Toolbox and draw the shape on the picture box. Click on the upper-left corner and drag to the lower-right corner.
Shapes and lines can be drawn in a picture box, but not in an image box.

4. With the shape selected, look in the Properties window. The choices for the Shape property of the shape are shown in Figure 7-7.
5. Choose the option called 4-Rounded Rectangle.
6. Move the rounded rectangle so it partially covers the icon in the upper-left corner of the picture box. See Figure 7-8.

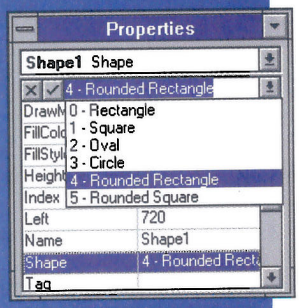


Figure 7-7
Shape options

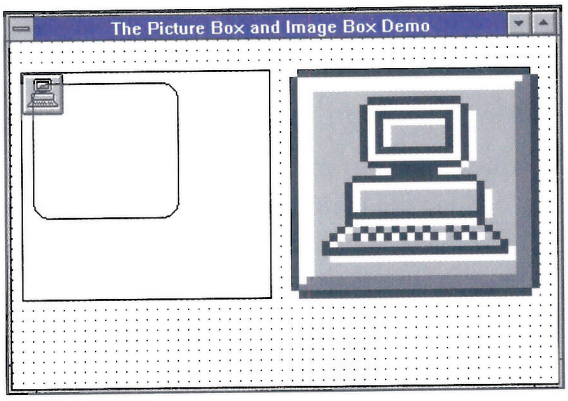


Figure 7-8
Moving the rectangle

7 Use the shape tool to draw a square within the confines of the image box. Resize the image box. Reposition the image box. Is the square a part of the image box?

8 Select the shape drawn and delete it.

Experimenting with the Images: Stage 2

Now change the images and experiment some more:

1 Select the picture box. Double-click on the Picture property and find the metafile directory in the Visual Basic directory. Select the business subdirectory, then click on a filename and press Enter, or click OK. The image will fill the picture box.

2 Add a different drawing to the image box and note the difference between it and the picture box.

3 Add a drawing to the form itself. The form also has a Picture property. Metafiles are good backgrounds for your forms. See Figure 7-9.

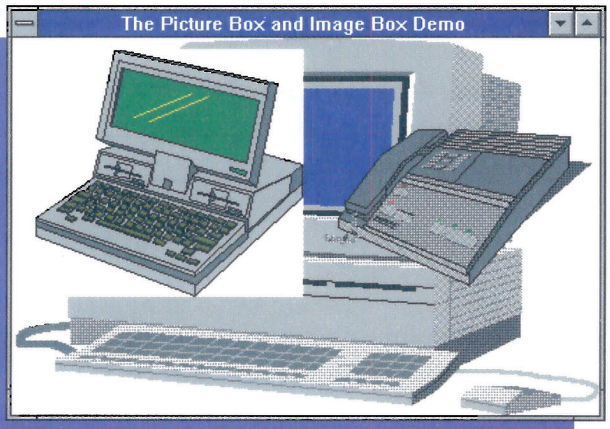


Figure 7-9
Adding drawings to the form

4 Remove the metafile from the form by selecting the Picture property of the form, clicking on the edit area, and pressing the Delete key. The Picture property value changes from (Metafile) to (none).

5 Click on the command button icon in the Toolbox. Click and drag to place the command button in the picture box (see Figure 7-10). (Double-clicking on the command button icon, then moving the button into the picture box will not work.)

6 Select the picture box. Drag the box to a different position within the form. Note the movement of the command button.

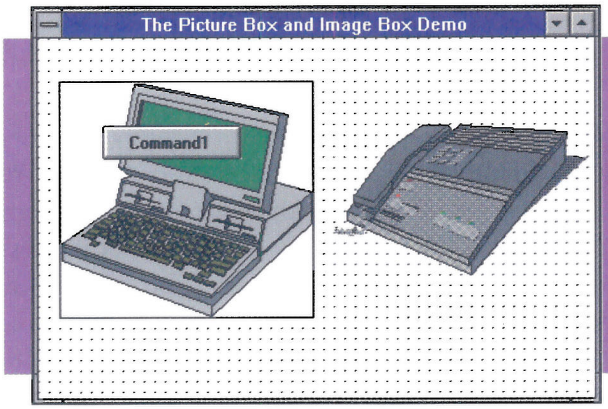


Figure 7-10
Placing a command button on the form

7 Draw a second command button within the boundary of the image box.

8 Select the image box and move it to a different part of the form. The command button does not move.

Reviewing the Boxes

What have you learned from experimenting with these controls? You can import a bitmap, an icon, or a metafile for display in a picture box. Except for the metafile, you cannot change the size of the imported image by stretching or shrinking it to fit the dimensions of the box. You will display only part of the image if that image is larger than the picture box.

The picture box can act as a container for other controls from the Toolbox. Command buttons, labels, and other controls become part of the picture box when they are drawn within the borders of the box. Text can be printed in the box.

You can use the shape control and line control to draw shapes and lines on a picture box or form. You can combine shapes, colors, patterns, and borders to create many different results.

The image box can display the same kind of graphics files as a picture box: bitmaps, icons, or metafiles. If you set the Stretch property of the image box to True, the image will fill the box, regardless of the box's size or the kind of image. Controls cannot be drawn in the box, nor can text be printed.

Typically, programmers use the picture box for plotting points and as a container for controls, images, and printed text. The image box is used just to hold graphic images, particularly if those images need to be sized. The buttons in a Window's toolbar are image boxes. The container holding the buttons is a picture box.

3 Section

Coordinate Systems Display Program

To draw or position graphics accurately, you need coordinate systems. Coordinate systems let you define each position on a screen with absolute accuracy. You can plot points, for example, by identifying the coordinates for that point. A point's coordinates include the row number and the column number for that point. Often, these row and column numbers are in pixels, but they do not have to be.

Visual Basic provides seven different built-in coordinate systems based on different units. Having this number of coordinate systems gives you a great deal of flexibility in the display of images.

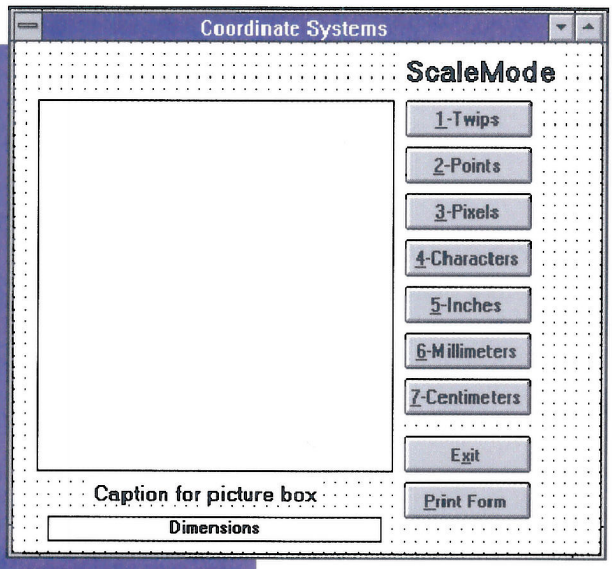


Figure 7-11
Finished Coordinate Systems project

This program uses each of the built-in coordinate systems to draw points in a picture box. The program uses a picture box to display grids of dots, each drawn using a different built-in coordinate system. One command button is used to draw the grid for each coordinate system. The finished form is shown in Figure 7-11.

Starting Out

Before you start building the form, you need to spend some time considering screen coordinate systems. Also, you will need some new methods to build this program.

SCREEN COORDINATES

Pixels are arranged on the screen in rows and columns. The pixel in the top-left corner of the screen is in the zeroth row and the zeroth column. Rows are counted from top to bottom.

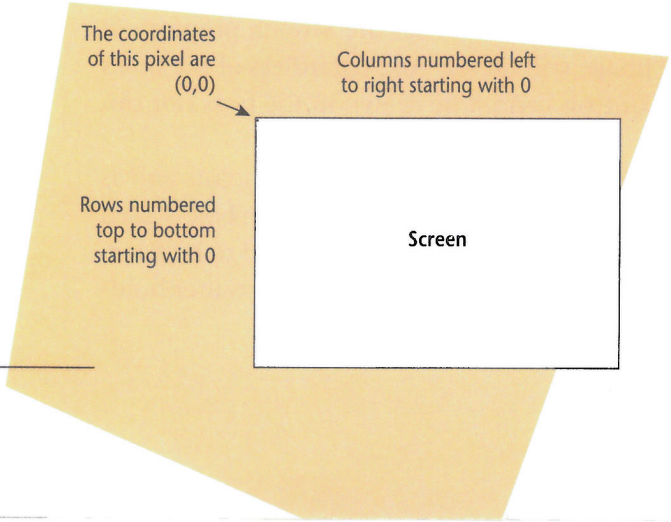


Figure 7-12
Screen coordinates

The first number of the screen coordinate is the column number, or *x* value, of the pixel. The second number is its row number, or *y* value. (see Figure 7-12).

You have already learned why it is awkward to use pixels as the basis for a screen coordinate system. Different screen displays have different numbers of pixels per row or per inch. Programmers now primarily use logical coordinate systems instead.

A logical coordinate system is any system other than pixels. If a coordinate system based on inches designates the length of a line as 1 inch, the line will be 1 logical inch long regardless of the number of pixels per row or per inch. The computer still makes the translation back to pixels, but this is done automatically.

You will be familiar with many of the coordinate systems in Table 7-1. It's not difficult to visualize an *x* and *y* axis marked off at 1-inch intervals or at 10-centimeter intervals. You have used these types of measurement before.

Measurements made using some of these coordinate systems do not provide the same result on different monitors. Forms using a coordinate system based on inches, for example, will be physically smaller on a 14-inch monitor than they are on a 21-inch monitor. Nevertheless, if both monitors are displaying the same resolution, the number of pixels that the form occupies will be the same on each monitor.

Table 7-1. Seven Coordinate Systems

ScaleMode	Coordinate system	Description
1	Twips	1440 per inch
2	Points	72 per inch
3	Pixels	The smallest unit of screen resolution
4	Characters	When printed, a character is 1/6 of an inch high and 1/12 of an inch wide.
5	Inches	
6	Millimeters	
7	Centimeters	

The twip coordinate system is the default for all Visual Basic objects. When the form itself is selected (so that its properties appear in the Properties window), the dimensions of the form, expressed in twips, are displayed on the Visual Basic tool bar (see Figure 7-13). If you move the form, or resize the form, the values change.

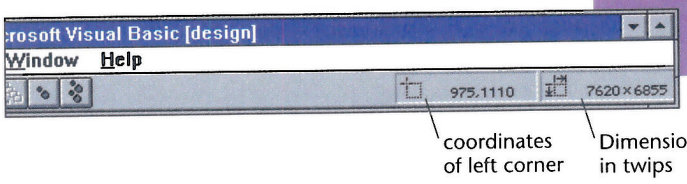


Figure 7-13
Dimensions of the form in twips

The coordinates of the left corner of the form are also displayed. The numbers asssume that the upper-left corner of the screen is the coordinate (0,0).

Figure 7-14
Choosing a coordinate system for a picture box

THE SCALEMODE PROPERTY

You can select a coordinate system for a picture box using the box's ScaleMode property. As you can see in Figure 7-14, Visual Basic automatically gives you a choice of all seven coordinate systems. Double-click to cycle through the choices. When you first open the Properties window for the picture box, you will see the twip system selected for this property. Twips are the default coordinate system for Visual Basic.

The code for the command buttons in this project changes the value of the picture box's ScaleMode property. The command buttons are named for the different coordinate systems. The command button for inches, for example, changes the value of the ScaleMode property to 5-Inch.

THE PSET METHOD

The program places dots in the picture box to create a grid. The dots are spaced according to which coordinate system you have chosen. Looking at the grid of dots, you can then visualize the differences between the coordinate systems.

To place the dots in the picture box, you need the **PSet** method. This method takes coordinate and color information and places dots on the screen. The simplified syntax for this method is:

object.**PSet** (x,y), *color*

Object is the name of the object, a picture box or a form; (x,y) are the screen coordinates of the point; and color is the color of the point to be plotted. If color is omitted, the point is plotted using the foreground color of the object.

Building the Form

Now that you have the background you need, you are ready to build the form:

1. Start Visual Basic. If Visual Basic is already running, select **New Project** from the **File** menu.
2. Change the caption of the form to **Coordinate Systems**.
3. Draw a picture box in the left two-thirds of the form. Name the box **picGrid**. Set the DrawWidth property to 2. This property determines the size of points and lines drawn on the control.

4 Place eight command buttons in a column down the right side of the form. Change the captions and names to those shown in the following table.

Captions	*Names*
&1-Twips	cmdTwip
&2-Points	cmdPoint
&3-Pixels	cmdPixel
&4-Character	cmdCharacter
&5-Inch	cmdInch
&6-Millimeter	cmdMillimeter
&7-Centimeter	cmdCentimeter
E&xit	cmdExit

Figure 7-15 shows the form during the design phase of the project.

5 Add a label above the command buttons with the caption **ScaleMode.** Add two labels below the picture box. The first provides a caption for the picture box. Change the name of the label to **lblScaleMode**. The label displays the current coordinate system used to draw the grid. The second label gives the dimensions of the box as measured with the current coordinate system. Change the name of the label to **lblDimensions**.

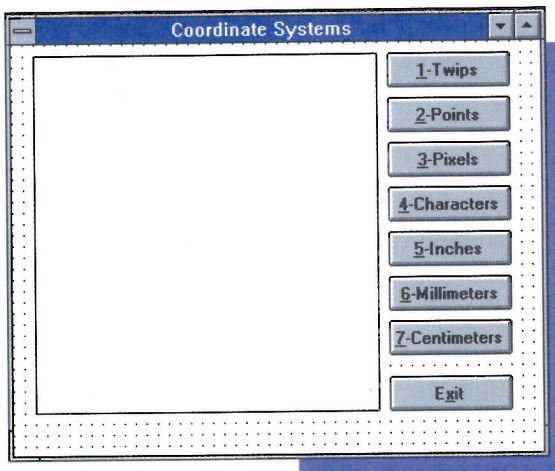


Figure 7-15 Coordinate System form being built

REARRANGING CONTROLS ON A FORM

Individual controls on a form can be moved by selecting the control with a click of the mouse, then dragging the control to a new position. A control can be resized by dragging a handle of the control. To reposition a group of controls, you use the pointer.

With the pointer, draw a temporary frame around a group of controls you want to move. Start by clicking on the upper-left corner and dragging to the lower-right corner. A gray frame shows the controls have been selected.

Once the controls have been selected, they can be moved as a group.

6 Adjust the size of the form to accommodate the additional controls.

7 Save the form and the project.

Finding the Dimensions of the Picture Box

The lower label displays the dimensions of the picture box in the given coordinate system. The measurements will change for each coordinate system.

The values for these dimensions are available in the properties of the picture box called ScaleWidth and ScaleHeight. When you choose a coordinate system using the ScaleMode property, ScaleWidth and ScaleHeight reflect the measure of the picture box expressed in those terms.

CODING THE TWIP BUTTON

Follow these steps to add code to the 1-Twips button.

1 With the form open, double-click on cmdTwip.

2 Add the following lines to Sub cmdTwip_Click():

```
'—Variable declarations
Dim Row As Integer, Col As Integer
Dim Sizes As String
'—Clear the picture box
picGrid.Cls
'—Choose the Twip coordinate system
picGrid.ScaleMode = 1
'—Set DrawWidth to 1 twip
picGrid.DrawWidth = 1
'—Set ScaleMode and Dimensions labels
lblScaleMode.Caption = "Twips: mark every 100 twips"
Sizes = Str$(picGrid.ScaleWidth) & " by " & Str$(picGrid.ScaleHeight)
lblDimensions.Caption = Sizes
'—Nested loop to plot a grid of dots
For Row = 0 To picGrid.ScaleHeight Step 100
   For Col = 0 To picGrid.ScaleWidth Step 100
      picGrid.PSet (Col, Row)
   Next Col
Next Row
'—Set DrawWidth back to two
picGrid.DrawWidth = 2
```

3 Enter the command **End** in cmdExit_Click().

4 Run the program. Click on the 1-Twips button to test the program (see Figure 7-16).

5 Save the form and project files.

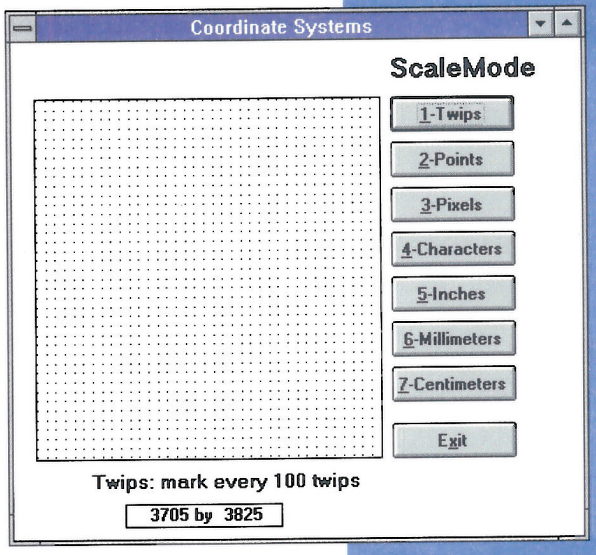


Figure 7-16
Testing the program

Here is an explanation of the code you have entered, by line:

The first two lines declare *Row*, *Col*, and *Sizes* as variables. You are using the first two of these variables to generate the coordinates of the points plotted. The string variable *Sizes* is used to build a string representing the dimensions of the picture box.

The statement **picGrid.Cls** clears the picture box. The next line sets the grid to the twips coordinate system. The line **picGrid.DrawWidth = 1** uses the smallest DrawWidth possible (1) for the size of dots. Twips are very small. You want to make sure that different dots do not overlap.

The line that starts *Sizes* = joins the width and length of the picture box (contained in ScaleWidth and ScaleHeight properties) into a single string, then assigns that string to the variable *Sizes*. The following line assigns the value of the variable *Sizes* to be displayed in a label.

Next comes a **For-Next** loop. You set up a loop from 0 to the vertical dimension of the picture box. The variable *Row* represents the vertical position of the dot in the box. The value of the variables in the loop increases by 100 each time through, which spaces the dots 100 twips apart.

The second (nested) **For-Next** loop generates values from 0 up to the maximum horizontal dimension of the box. *Col* represents the horizontal position of the dot. The value of the variables also increases by 100 each time through. The line **picGrid.PSet (Col, Row)** plots the dot at the intersection of the coordinates (*Col*, *Row*).

The last line sets the DrawWidth value back to the design-time setting of 2.

The decision to space the points representing the twip coordinate system 100 units apart was arrived at experimentally. Using a value of 100 seemed to illustrate the relative size of the twip coordinate system well.

CODING THE REST OF THE BUTTONS

You are now ready to code the rest of the buttons. You will be copying the code for the Twips button to the other buttons' subroutines. The

STYLE NOTE

The inner loop, the For Col = loop, is indented. Indentation visually groups statements. The indentation is not a necessary part of the program, but it is certainly a courtesy to anyone reading the program who is unfamiliar with its code. You yourself may fit that description a few months after you have written a program!

spacing of the dots, however, will be different. Instead of 100, use the spacing indicated in the table below.

Coordinate system	*Spacing*
1:Twips	100
2:Points	20
3:Pixels	20
4:Characters	5
5:Inches	½ inch*
6:Millimeter	10 millimeters
7:Centimeter	1 centimeter

* An additional change must be made for the Inch routine. Declare *Row* and *Col* to be of type Single.

1. Open the Code window and select cmdTwip_Click(). Select all the code except the top and bottom lines. Copy the code with Ctrl+C. Now select the routine cmdPoint_Click() and paste the code you just copied. Use Ctrl+V or select the Paste command from the Edit menu.
2. Change the ScaleMode property to 2.
3. Delete or comment out the statement that changes the DrawWidth to 1.
4. Change every reference to Twip, to Point.
5. Change the label text from:

   ```
   Twips: mark every 100 twips
   ```

   to:

   ```
   Points: mark every 20 points
   ```

6. Change the step values of 100 in the two **For-Next** loops to 20. Better yet, replace the values with a constant defined as 20.
7. Run the program and test the code.
8. Copy the code into the subroutines for the other command buttons and modify each one. Use the spacing table for suggested loop increments, and Table 7-1 to set the ScaleMode property. Change the declarations for *Row* and *Col* in the code for the Inches button. Change Integer to Single.
9. Run the program. Test each command button.
   Stop the program. Save the project and form.

QUESTIONS AND ACTIVITIES

1. What are the choices listed for the ScaleMode property? What is the default mode?
2. Change the Coordinate Systems program to use two labels for the dimensions. Instead of joining the width and height in a single string, use two labels, each labeled as the width and height of the picture box.
3. Add a command button to the Coordinate Systems program. The caption is **&Print Form**, the name of the button is **cmdPrintForm**. In the code window of cmdPrintForm_Click(), enter the following line of code:

   ```
   form1.PrintForm
   ```

   Execute the program, click on the centimeter button, and print the form by clicking on the Print Form button.
   With a ruler, measure the distance between the dots. How close is the distance to 1 centimeter? Measure the dimensions of the box. How close is the printed dimension?
4. What does the **PSet** method do?
5. To what objects can the **PSet** method be applied?
6. What information is needed by the **PSet** method? Which is optional? What happens if the optional information is omitted?
7. Comment out the **picGrid.Cls** statements in at least two command buttons. Run the program and choose those two buttons. Restore the lines and run the program again. Try changing **Cls** to **Clear**. **Clear** works for a listbox or a combobox. **Cls** works for a picture box or a form.
   In the statement:

   ```
   For Row = 0 To picGrid.ScaleHeight Step 100
   ```

   what does **Step 100** do?
8. Change the DrawWidth property to 5 in for the cmdTwip button. Run the program and note the result.
9. Why is it a good idea to indent the statements making up the body of a loop?

Color, Lines, and Circles

Nowadays, it's hard to think of graphics without thinking of color. This section introduces you to using color in Visual Basic programs. You then experiment with color using the **Line** and **Circle** methods. You first worked with these methods in Chapter 6.

Using Color

You can specify color using either of two functions: **QBColor** or **RGB**. The former maintains a link with older versions of Basic (Quick Basic), providing only sixteen colors. The latter takes advantage of the more flexible and advanced color system of Windows.

THE QBCOLOR FUNCTION

The **QBColor***(n)* function takes one of sixteen values, 0 through 15, as a parameter and returns a color. The first eight colors are the kind of colors you would expect to see in a box of crayons: red, yellow, blue, green. The second eight are lightened versions of the first eight. See Table 7-2.

Table 7-2 Table of QBColor Values

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Yellow	14	Light Yellow
7	White	15	Bright White

These colors were the ones used by the first color graphics systems for the PC. As you can see, there are not many of these colors. Your color choices for program elements are therefore extremely limited using this function.

The sixteen values of the QBColor function are completely arbitrary.

Today the QBColor function is still used, because sometimes this simple color system is all that's needed to help a program communicate with the user.

THE RGB COLOR SYSTEM

Computer video cards and monitors can now display far more than 16 colors. Each pixel has a "color depth." The color depth is the number of

bits used to represent the color of the pixel. Various graphics systems use 2, 4, 8, 16, 24, and 32 bits per pixel. The amount of video memory, the video chips, and the current settings of the operating system determine the number of colors.

The color system of Visual Basic is based on representing colors as a mixture of three primary colors: red, green, and blue. You identify the specific color using the RGB function. The syntax of this function is:

RGB *(red, green, blue)*

For each color—red, green, and blue—you can indicate a value ranging from 0 to 255. These values represent the relative intensity of that color in the resulting color blend. For example, RGB (50, 100, 80) has a larger green component than the color represented by RGB (50, 50, 8). As you might expect, RGB(0, 0, 0) is black, and RGB(255, 255, 255) is white. The numbers you use to represent the red, green, and blue components of a color must be integers.

So how many colors can you create with the RGB function? If you multiply 256 times 256 times 256, you get over 16.7 million different color possibilities. Far more than the 16 possible with QBColor and older machines!

The RGB function returns a long integer representing the color. You may recall from a previous chapter that long integers are whole numbers that range from approximately 0 to 4 billion (if taken as an unsigned integer). You won't see a complete table of color names with their corresponding long integer values because it would be far too long. Many drawing methods, such as **Line** and **Circle**, take an optional color argument. You can use the value returned by RGB to specify that color.

NUMBERS IN THE PROPERTIES WINDOW

An object on a form can have several color properties. For example, place a rectangular shape on a form, then look in the Properties window. You see these color properties:

BackColor
BorderColor
FillColor

The values for these properties represent a color. These values may look strange, because they are expressed in the hexadecimal number system. The hexadecimal number system is based on powers of sixteen: $16^0 = 1$, $16^1=16$, $16^2=256$. The numerals in base 16, which is another name for the hexadecimal number system, are 0 through 9 and A through F. The decimal number 255 is expressed in base 16 as FF.

SEEING COLOR ON SCREEN

The three parameters of the RGB function correspond to the three electron guns in a picture tube that produce color on the screen. Each of the guns shoots electrons, guided by magnetic plates, to one of three color dots on the inside surface of the screen. When a color dot is hit, it glows.

The higher the value of the parameter, the more intense is that component of the color.

When you double-click a color property, such as BackColor, a window appears containing color swatches from which you can select a color by clicking. This window contains a small but very useful subset of the full range of possible colors.

Hexadecimal numbers are expressed in the Properties window with a leading and trailing ampersand (&) and the letter H. The long integer representing a color has the following structure: &H00bbggrr&, where bb is the hexadecimal value of the blue component, gg is the green component, and rr is the red component.

Experimenting with Color: Stage 1

Follow these steps to investigate color numbers. You will choose different colors from the color palette and examine the corresponding color numbers.

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Change the caption of the default form to **Color Numbers**.
3. Select the BackColor property of the form in the Properties window and note its current value.
4. Double-click on the property to open the color palette. This palette presents only a small fraction of the colors available using the RGB system.

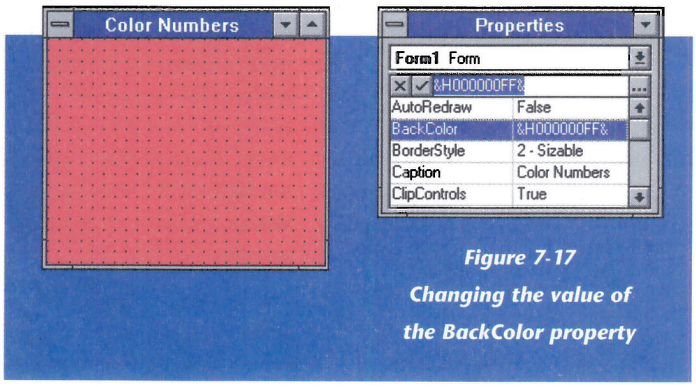


Figure 7-17
Changing the value of the BackColor property

5. Click on a vibrant red. The form changes color. Note the value of the BackColor property. The hexa decimal value, in this case, indicates a strong red component. See Figure 7-17.
6. Double-click on the BackColor property and pick from the palette a bright green. Click on the color and note the value reported in the Properties window.
7. Double-click on the BackColor property and pick a deep blue color. Click on the color and note the value reported in the Properties window.
8. Try other colors from the color palette, noting their color values. Be sure to try black, white, yellow, and magenta.

9 Select the BackColor property and change the value of the color number in the edit area. The blue, green and red components of the color are in the last six digits, two for each component. Change the values two at a time, pressing Enter to make each change. Note the results.

If you do not see much difference in some of the colors you select, the culprit may be your color video hardware. It may not be adequate to show tiny variations in color. Or the screen of your computer may not support the colors you have selected.

Experimenting with Color: Stage 2

You can use scroll bars to change the red, blue, and green parameters in a color. These changed parameters will then be sent to the RGB function.

To automate color changes in this way, follow these steps:

1 Start Visual Basic. If Visual Basic is already running, choose New Project from the File menu.

2 Change the caption of the form to **Red, Green, Blue**.

3 Use the shape control to put a square on the form.

4 Change the name of the shape to **shpColorBox**.

5 Change the BackStyle property to 1-Opaque.

6 Put three horizontal scrollbars on the form below the shape. Name the bars **hsbRed**, **hsbGreen**, and **hsbBlue**.

7 Set the Min and Max properties of each of the bars to 0 and 255, respectively.

8 Label the bars **Red**, **Green**, and **Blue**. See Figure 7-18.

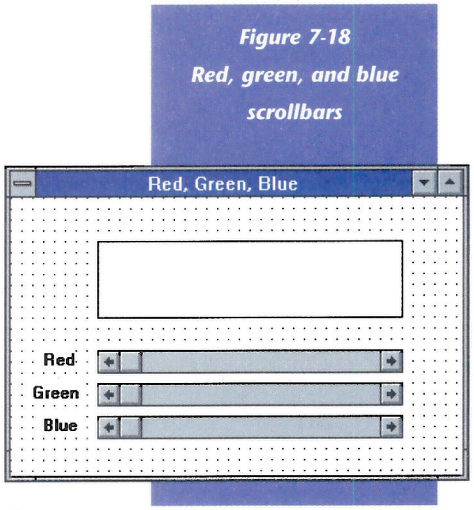


Figure 7-18
Red, green, and blue scrollbars

9 Enter this declaration in the general declarations section of the code. These variables are used in each of the three Change events of the horizontal scroll bars:

```
Dim Red As Integer, Green As Integer, Blue As Integer
```

10 Enter these lines of code in the Change events of each of the three scrollbars:

```
Red = hsbRed.Value
Green = hsbGreen.Value
Blue = hsbBlue.Value
shpColorBox.BackColor = RGB(Red, Green, Blue)
```

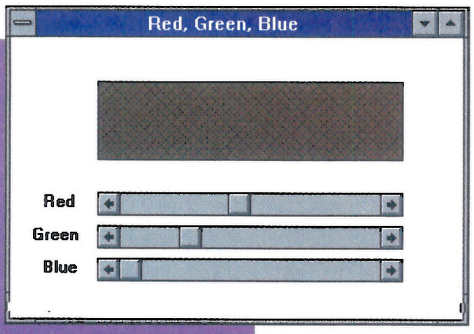


Figure 7-19 Assigning a value to the BackColor property through code

11 Run the program. Experiment with moving the scrollbox on each of the scrollbars.

The code reads each of the three scrollbars through the value property. The three values are sent as parameters to the RGB function. The color returned by that function is assigned to the BackColor property of the rectangle on the form. See Figure 7-19.

Using the Line Method

You have already experimented with the **Line** method as a means of drawing lines. You can also draw boxes and filled boxes with this method. Using the method, you can draw on both forms and picture boxes. And, now that you have begun to work with color, you can control the color of the lines.

The **Line** method has the following syntax:

object.**Line** *(x1,y1)-(x2,y2), color,* BF

In the syntax above, almost everything except the word "Line" is optional. Some valid variations follow:

```
form1.Line (0,1)-(25,40)
Line (0,1)-(25,40)
Line (0,1)-(25,40),QBColor(3)
Line -(25,40), RGB(0, 192, 0)
picture1.Line (0,1)-(25,40), ,B
Line (0,1)-(25,40), ,BF
```

If you leave out the object name, the method draws on the underlying form. You would use this variation to draw graphic images on the form.

The two sets of numbers separated with a dash are coordinates of endpoints. If you include these, the method draws a line between those two points. You can also use the **Line** method to draw a line between points, starting from where the last point was plotted. The coordinates of that last point are stored in the object's CurrentX and CurrentY properties. To do so, you include only the coordinates of the second endpoint, preceded by a minus sign. In other words:

```
Line -(x2,y2)
```

To draw a box using the first coordinate as the top-left corner and the second coordinate as the lower-right corner, include the letter "B" at

the end of the statement. To fill that box with color, include the letter "F" immediately after the "B."

If you do not specify any color information, the **Line** method uses the color specified in the ForeColor property of the object. This property is present in most objects, and you may have already experimented with its use. In a textbox, the property controls the color of the displayed text. In objects that have a **Line** method, the property controls the color of the lines drawn.

EXPERIMENTING WITH LINES AND COLOR

To experiment with lines and color, you are going to use four different variants of the **Line** method. The code you will write divides a form into four quarters called quadrants. In one quadrant, the Line method draws lines up and down. In the second quadrant, you will have empty boxes, and in the third, filled boxes. In the final quadrant, the **Line** method draws lines at a zig-zag. Using the **QBColor** function, you will make all of the lines and boxes different colors (see Figure 7-20).

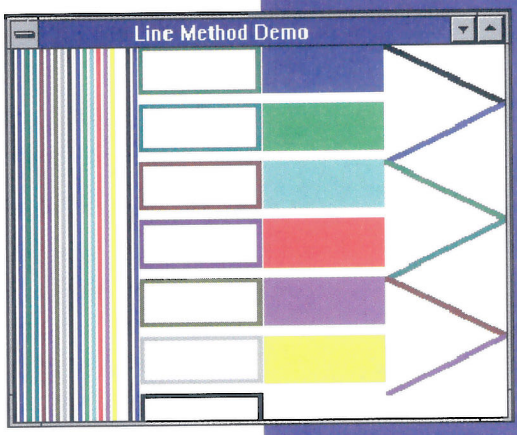


Figure 7-20
The finished Line Method Demo project

To create the program, follow these steps:

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Change the caption of the form to **Line Method Demo**.
3. Enter the following code in the Form_Click event procedure:

```
Dim nWidth, nHeight, nColor, x, y
ScaleMode = 3
nWidth = ScaleWidth
nHeight = ScaleHeight
DrawWidth = 3
'—First quadrant
nColor = 0
For x = 0 To nWidth/4 Step 5
   Line (x, 0)-(x, nHeight), QBColor(nColor Mod 16)
   nColor = nColor + 1 ' Mod keeps nColor value between 0 and 15
Next x
'—Second quadrant
For y = 0 To nHeight Step 40
   Line (nWidth / 4 + 3, y)-(nWidth/2, y + 30), QBColor(nColor Mod 16), B
```

continued

```
    nColor = nColor + 1
Next y
'—Third quadrant
For y = 0 To nHeight Step 40
    Line (nWidth / 2 + 3, y)-(3 * nWidth/4, y + 30), QBColor(nColor Mod 16), BF
    nColor = nColor + 1
Next y
'—Fourth quadrant
PSet (3 * nWidth/4 + 3, 0)
For y = 40 To nHeight Step 80
    Line -(nWidth, y), QBColor(nColor Mod 16)
    nColor = nColor + 1
    Line -(3 * nWidth / 4 + 3, y + 40), QBColor(nColor Mod 16)
    nColor = nColor + 1
Next y
```

4. Run the program. Place your cursor on the form and move it. Why did the lines appear when you moved the mouse?

5. Save the project and form files.

Here is an explanation of the code:

The first line of the code declares the variables and sets up some preliminary values. *nWidth* and *nHeight* store the dimensions of the form. *nColor* is the parameter sent to QBColor to generate a color for the drawing. *x* and *y* are used as variables to control various loops.

In the second line, you did not specify an object with the ScaleMode property, so the default is Form1. **ScaleMode = 3** indicates the pixel coordinate system. You could use other coordinate systems, but using pixels will give the user an idea of the relative ratio of sizes of the pixel to the screen. The ratio of the width of the lines, set by the DrawWidth property, to the spaces between the lines gives a good sense of the size of the pixels on the screen.

In the line nWidth = ScaleWidth, you assign the value of the ScaleWidth property to the variable *nWidth*. This property is the measure of the width of the form in pixels. The prefix in the variable *nWidth* shows it represents a numeric value. You perform the same step for the height of the form in the following line.

To give the line some thickness, **DrawWidth = 3** sets the DrawWidth property to 3. Thicker lines make the colors of the lines more visible.

In the first line of code for the first quadrant, *nColor* is a variable used to control the color of the lines drawn. It is used in the **QBColor** function mod 16 to provide a value for the parameter of the function. (*nColor* mod 16 is equal to the remainder of *nColor* when it is divided by 16.) Set-

ting *nColor* to 0 guarantees that the colors used to draw lines will start with the QBColor (0), which is black.

The first **For-Next** loop draws vertical lines in the first quadrant. The loop limit, *nWidth/4*, confines drawing to the first quadrant. This section of the program shows what is probably the most common use of the Line method: drawing lines between points (from the top to the bottom of the screen) with a color specified by the QBColor function.

After each line is drawn, the *nColor* variable is incremented. Because that value is taken modulo 16 (divided by 16 and just the remainder returned) inside the function, the result will always be an integer between 0 and 15.

The *x* value in this loop specifies the column number of the display. The loop generates values, stepping up by 5 each time through. The width of the screen, *nWidth*, divided by 4, is the number of columns available in the first quadrant.

The width of the line drawn is set to 3. A new band starts every 5 pixels. This leaves a narrow band of white between each band of color.

The second **For-Next** loop sets up the second quadrant of the form. The variant of the **Line** method you used for this quadrant draws colored boxes (due to the "B" at the end of the line). The coordinates you provided act as two corners of each box.

The loop sets up row values stepping from the top of the form to the bottom, incrementing by 40 pixels each time through the loop. The **Line** method draws a box, 30 pixels high and almost equal to the width of the quadrant.

The column values for the boxes range from *nWidth*/4 + 3, 3 pixels beyond the end of the first quadrant, to *nWidth*/2, the middle of the screen. The row values for each box range from *y* to *y*+30, leaving a 10-pixel space between each box.

The code for the third quadrant is almost identical to that for the second. It draws very similar boxes, from column *nWidth*/2 + 3, to 3**nWidth*/4. The difference is that, in this quadrant, you filled the boxes with color by adding an "F" at the end of the Line statement.

For the final quadrant, you create a zigzag line, moving from the top of the form to the bottom. This last section uses the **Line** method to draw from the last used position on the form to whatever coordinate is specified in the **Line** statement:

The code starts by plotting a single point at the top-left corner of the fourth quadrant. The **For-Next** statement starts *y*, the row value, from a position 40 pixels down from the top. The value is incremented by 80 pixels each time through the loop.

Within the loop, a line is drawn from the original point to the right-

hand side of the quadrant, 40 pixels lower than the previous point. From there, a line is drawn back to the left side of the quadrant, also 40 pixels lower than the previous point.

Each time through the loop, you zig from left to right and zag from right to left, each cycle covering 80 pixels top to bottom.

The color variable, *nColor*, is incremented each time. This cycles through each of the 16 possible colors.

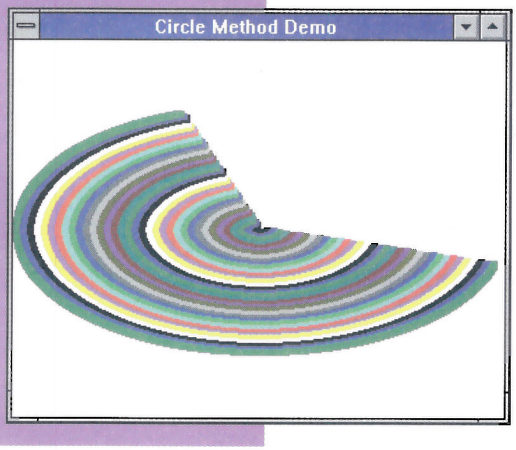


Figure 7-21
Experimenting with the Circle method

Using the Circle Method

Now that you know more about color, you can also expand your use of the **Circle** method. The **Circle** method can draw ellipses as well as circles, and it can even draw partial ellipses. Like the **Line** method, the **Circle** method also lets you specify the color with which it draws.

See Figure 7-21 for the end result of the code you will write.

Follow these steps to explore the **Circle** method.

1. Start Visual Basic. If Visual Basic is already running, choose New Project from the File menu.
2. Enter the following code in the Form_Click event procedure:

```
Dim x, y, cx, cy, rad, nColor
Dim nWidth, nHeight
Const StartPt = 1.9
Const EndingPt = 6
Const Ellipse = .5
ScaleMode = 3
'—The width of the figure in pixels
nWidth = ScaleWidth
'—The height of the window
nHeight = ScaleHeight
'—The coordinates of the center of the figure
cx = nWidth / 2
cy = nHeight / 2
'—The width of the figure drawn
DrawWidth = 5
'—Color variable
nColor = 0
For rad = 5 To nWidth / 2 Step 5
```

```
        Circle (cx, cy), rad, QBColor(nColor Mod 16), StartPt, EndingPt, Ellipse
        nColor = nColor + 1    ' cycle through the colors
    Next rad
```

Let's concentrate on the **Circle** statement because most of this code is similar to the code used in the Line Demo program.

Because we are drawing directly on the form, we don't need to specify an object. The Circle method will draw directly on the underlying form. Following the method name, Circle, are the coordinates of the center of the circle. Then comes the radius, expressed in pixels. ScaleMode is set to 3, the pixel coordinate system. The color of the circle is cycled through the 16 colors possible with the QBColor function, controlled by the value of the variable *nColor*.

The next two values in the Circle statement are interesting. Figure 7-32 shows just a portion of an ellipse. You can select what portion of the ellipse to draw by including a *StartPt* and *EndingPt*. The starting point and the ending point are specified in radians.

In Figure 7-21, an angle in standard position is shown. One side of the angle, the initial side, points due east. The angle is measured counterclockwise from the initial side. The **Circle** method uses measures of angles in the standard position (see Figure 7-22) to give the starting and ending points of the figure drawn.

In the example above, the starting point is 1.9 radians. The ending point is 6 radians, or 6/6.28 (.955) of the circle. If you omit the start and end points, the entire ellipse is drawn.

The last number in the statement, .5, indicates the shape of the figure drawn. If you omit the number, the method draws a circle. If the number is less than 1, a horizontal ellipse is drawn. If the number is greater than 1, a vertical ellipse is drawn. These aren't really different figures at all. A circle is just a special case of the ellipse.

RADIAN ANGLE MEASURE

The measure of an angle is commonly given in degrees. A circle has 360 degrees. Another way of measuring angles is to use radians. There are two times pi, where pi = 3.1415927, radians in a circle. Using the radian measure simplifies the expression of many formulas in physics. The angle measures used by Visual Basic in the Circle method are radian measures.

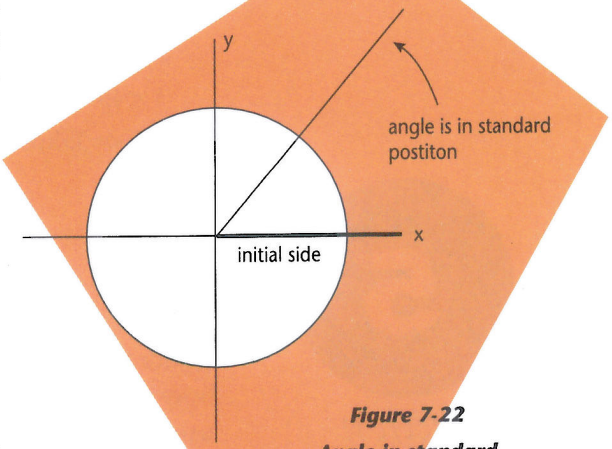


Figure 7-22
Angle in standard position

QUESTIONS AND ACTIVITIES

1. Describe what this statement will do, as well as the location of the resulting graphic:

```
Line (-7,10)-(1,0), ,B
```

2. Place a textbox on a form. Set the Text property to "This tests the ForeColor property". In the Properties window, double-click on the ForeColor property and experiment with different colors. Do the same with the BackColor property.

3. How many colors are theoretically available using the RGB system?
4. In the code for the first quadrant of the Line Method Demo program, change the DrawWidth property to 1 and run the program. Change DrawWidth to 3, 5, 7, and 10. Run the program for each change and note the results.
5. To control the value of the *nColor* variable, the Mod operator is used. This keeps the value of *nColor* between 0 and 15. Add an **If-Then** statement that resets the value of *nColor* to 0 when the current value is 16. After what statement would this statement go?
6. If the width of a picture box were *W*, and you wanted to divide the area into thirds, what expression would you use for the column number of the line dividing the first third from the second? The second from the third?
7. Describe the purpose of each of the expressions in the **Circle** statement.
8. What determines the units of measurement used for the radius of the circle?
9. An angle in standard position may be measured with at least two different units. What are they? How many of each are there in a circle?
10. In the Circle demo program, change the value of the constant *Ellipse*. Try values in the following ranges. After each change, run the program and click the form.

    0 < Ellipse < .5

    .5 < Ellipse < 1

    Ellipse = 1

    1 < Ellipse < 1.5

    1.5 <= Ellipse < 2

The Quadratic Formula Program

Quadratic equations pop up pretty often in math and science. Knowing how to solve a quadratic equation is a useful skill. Visual Basic can be used to build a very good quadratic equation solver. Not only will the program calculate and display the solutions to the equation, the program can calculate and display a table of values that satisfy the graph of the equation and draw the graph itself.

The Freefall program from Chapter 3 uses a quadratic equation to calculate the distance an object falls while under the influence of gravity. With a method of solving a quadratic equation, the Freefall program could be turned around to calculate the time it takes to fall a given distance, instead of calculating the distance from the time.

A complete solution to a quadratic equation might include the following:

- The solutions to the equation: the values of x that make the equation true
- A list of ordered pairs that satisfy the graph of the equation
- A graph of the equation

In thinking about how you will solve this problem, do you see the need for the following:

- Textboxes to enter the coefficients of the equations
- Command buttons to execute the calculations, clear the input boxes, and exit the program
- The quadratic formula to calculate the solutions
- A listbox to display the ordered pairs of values that satisfy the graph of the equation
- An array to hold the ordered pairs generated for the listbox and the graph
- A picture box to display the graph of the equation

From the users' point of view, they will see a form similar to the one in Figure 7-23. Users are prompted to enter coefficients of the quadratic equation in three different textboxes. The focus is initially on the box labeled A:. As the user enters values, the focus shifts to each of the next two textboxes. When a user presses Enter from the C textbox, focus shifts to the Calculate button.

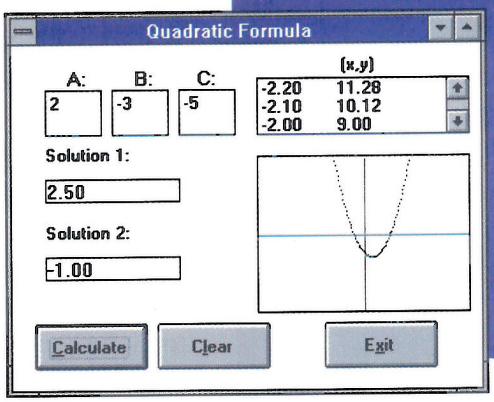


Figure 7-23
The Quadratic Formula project

At that point, if the user clicks on the Calculate button, the following occurs:

- The roots (which are solutions of the quadratic equation) are calculated and displayed.
- A table of x and y values is displayed.
- A graph of the function is drawn in a picture box.

The x values in the table and the graph range from -10 to 10. The corresponding y values are generated from the quadratic function entered by the user.

Background

The program uses textboxes to enter the coefficients of a quadratic equation in standard form:

$ax^2 + bx + c = 0$

Using the quadratic formula, the solutions are found:

Discriminant = b^2 - 4ac

r1 = (-b + sqr(discriminant)) / (2a)

r2 = (-b - sqr(discriminant)) / (2a)

The discriminant is an important intermediate value that reveals the nature of the solutions to the equation. If the discriminant is negative, there are no "real" roots—that is, the only roots are imaginary numbers. Otherwise, the roots are calculated and displayed.

The program generates and displays a table of *x,y* values for the function. The standard form of the equation, $ax^2 + bx + c = 0$, is turned into a function by replacing 0 with the variable y.

$y = ax^2 + bx + c$

The values used for *x* range from -10 through 10.

An array is used to store the values generated for the table. Those same values are used to plot the graph of the function in a picture box. This graph will be a visual display of the table of values.

Starting Out

Before starting to build the form for this project, you need to learn about the new methods and statements you are going to use. You will learn how to set up a coordinate system with any desired scale and origin.

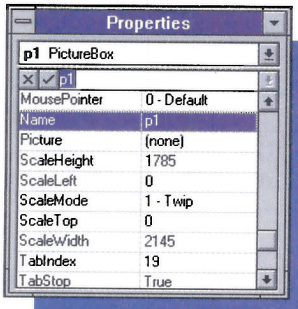


Figure 7-24 Looking at the ScaleTop and ScaleLeft properties of a picture box

CUSTOM COORDINATE SYSTEMS

Besides the seven built-in coordinate systems, you can create a custom coordinate system. To do so, you set the SetMode property to 0.

The ScaleHeight and ScaleWidth properties of picture boxes are used to report, or to set, the height and width of the box. The properties ScaleTop and ScaleLeft define the values associated with the top row and first column, respectively. Figure 7-24 shows the default settings.

The default settings for these values are both 0. This indicates the top left corner has the coordinate 0,0. In this program, a coordinate system with *x* values from -10 to 10 and *y* values from -20 to 20 will display graphs nicely. Using the same scale for all the graphs will help give an idea of the relationship between various equations. Some graphs may

not be visible on the given scale at all. Even this is informative, because it shows where the roots are not.

You may want to modify the program to adopt a coordinate system that will always display the graph of the equation in a "nice" way. To do so, you would use high and low values from the list of ordered pairs to calculate scale factors to fit the graph to the screen. Unfortunately, if this scaling is carried out too far, all the graphs will look exactly alike. If this is the case, the graphs are meaningless.

Visual Basic gives us a way to set up these coordinate values, simultaneously setting appropriate values for all the Scale properties. This is the **Scale** method.

The command to set up the coordinate system described above is:

```
pctGraph.Scale (-10,20)-(10, -20)
```

The object to which the method is applied is the picture box named pctGraph. The command **Scale** follows this name after a period. Then comes a space and the coordinates of the upper-left corner of the coordinate system. After a dash, you add the coordinates of the lower-right corner.

Once this statement is executed, the values of all the Scale properties for the picture box are set. In this case, the values are shown in Figure 7-25.

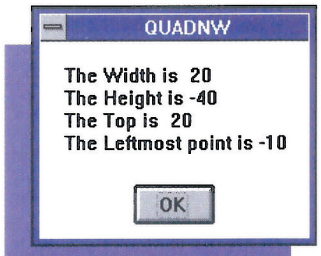


Figure 7-25
Scale property values

DIAGNOSTIC CODE

The message box in Figure 7-25 is one you would use for diagnostic purposes. Programmers often insert diagnostic code into a procedure during the development of a program and then comment the code out by putting in single quotation marks at the beginning of each line.

The diagnostic code can help find a problem when the program doesn't behave the way you expect. Commenting the lines out removes them from the executable program. Later, the lines can be restored to their diagnostic function, should the need arise again because of subsequent modifications.

Using a message box for diagnostic code interrupts the program and requires the user to click OK for the program to continue. Almost the same purpose is accomplished by adding the line:

```
Debug.Print msg
```

This line prints the same information in the Debug window. See Figure 7-26. Unlike **MsgBox**, **Debug.Print** does not pause program execution until the user performs an action.

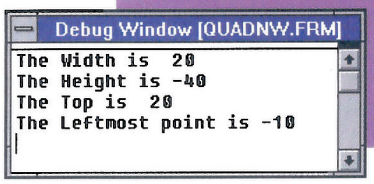


Figure 7-26
Scale values in Debug window

REDIM STATEMENT

An array declaration such as:

```
Dim YValues(1 To 41) As Single
```

fixes the size of the *YValues* array to 41. The size of the array is set when the program is written and may not be changed.

The **ReDim** statement allows the size of an array to be changed while the program is running. For instance, the statement:

```
ReDim YValues(1 To TotalPoints * 2) As Single
```

uses the value of *TotalPoints* in an expression to calculate the size of the array.

The total number of points used depends on the increment between successive points on the graph. The span of *x* values is -10 to 10, or 21 units (remember to count 0). If the increment between successive values is 0.1, the total number of values generated is 21/.1, or 210. The array needs this many spaces to store values.

The increment between successive values is a constant defined in the program. This allows the programmer to change it if more or less resolution is desired. Any change in the increment necessitates a change in the number of values stored in the array. Thus, the array is resized according the number of values needed, as determined by the span of values and the increment between them.

Setting up the Form

The form contains three labeled textboxes in which users enter the coefficients of the equation. It also contains two labels to display the solutions (if they exist), a listbox, a picture box, and command buttons to Calculate, Clear, and Exit.

Follow these steps to set up the form.

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Change the caption of the form to **Quadratic Formula**.
3. Place three textboxes in the upper-left portion of the form. Name the boxes **txtA**, **txtB**, and **txtC**. Delete the text from each.
4. Over each textbox, place a label. Caption the labels **A:**, **B:**, and **C:**. Set the FontSize to 9.75. Set the AutoSize property to True. See Figure 7-27.

DUPLICATE CONTROLS

You can use edit commands to put identical controls on a form. To experiment, put a textbox on a form. Select the box and press Ctrl+C to copy the control. Press Ctrl+V to paste the control on the form. The new control appears in the upper-left corner of the form. The new control retains all the properties of the original. If you changed the BorderStyle of the control, for example, the copied control retains that changed BorderStyle.

When the duplicate control is created, Visual Basic gives you the option of creating a control array. Creating a control array means creating more than one control with the same name. If you are just trying to duplicate controls, there is no reason to create a control array. Just say No.

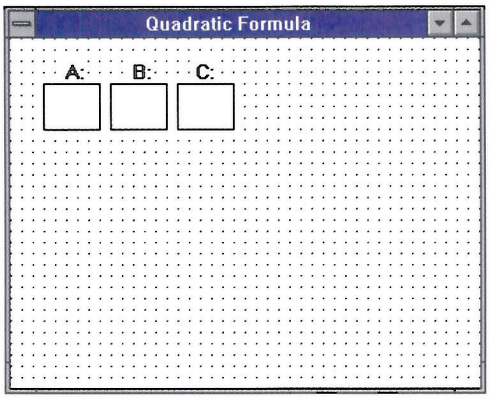


Figure 7-27
Placing textboxes on the Quadratic Formula form

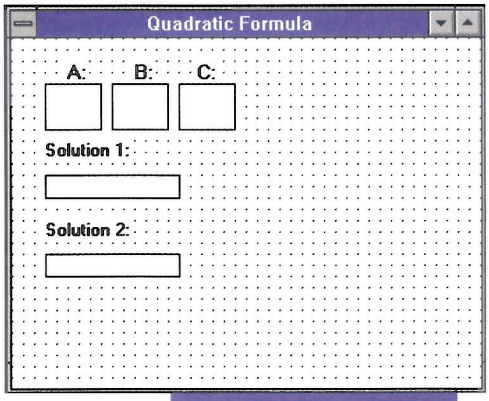


Figure 7-28
Placing labels on the form

5 Place two more labels on the form for the solutions of the equation. Set their BorderStyle to Single. Name the labels **lblSolution1**, **lblSolution2**. Delete the captions. Change the FontSize of the labels to 9.75.

6 Place two more labels on the form to identify the solution labels. Change their captions to **Solution 1** and **Solution 2**. See Figure 7-28.

7 Place a listbox in the upper-right corner of the form. Change the name of the box to **lstTable**.

8 Place a picture box for the graph below the listbox. Change the name of the box to **pctGraph**. The coordinate system for the box will be set in the code. See Figure 7-29.

9 Place three command buttons across the bottom of the form. Change their captions to **&Calculate**, **C&lear**, and **E&xit**. Change the names to **cmdCalculate**, **cmdClear**, and **cmdExit**.

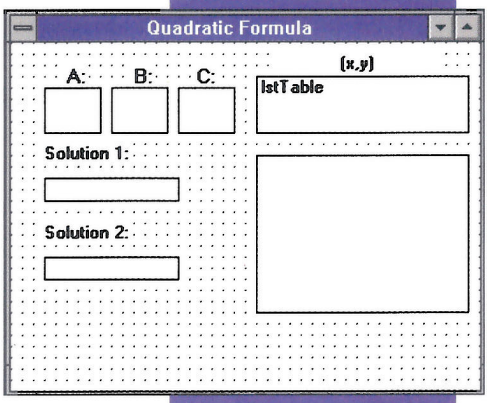


Figure 7-29
Placing a picture box on the form

Writing the Code for the Calculate button

Follow these steps to add code to the quadratic formula project:

1 Double-click on the cmdCalculate button to enter the Code window.

2 Include this code:

```
'—Declarations
Dim a As Single, b As Single, c As Single
Dim Disc As Single
Dim Root1 As Single, Root2 As Single
```

continued

```
'—Collect information from textboxes
a = Val(txtA)
b = Val(txtB)
c = Val(txtC)
'—If a is 0, the equation is not quadratic
If a <> 0 Then
   '—Calculate the Discriminant
   Disc = b * b - 4 * a * c
   '—If the discriminant is less than zero: there are no solutions
   If Disc < 0 Then
      lblSolution1 = "No solution"
      lblSolution2 = ""
   Else
      '—Find the solutions, calculate table, and graph
      Root1 = (-b + Sqr(Disc)) / (2 * a)
      Root2 = (-b - Sqr(Disc)) / (2 * a)
      '—Display solutions
      lblSolution1 = Format(Root1, "Fixed")
      lblSolution2 = Format(Root2, "Fixed")
      '—Generate table
      Dim x As Single, y As Single
      Dim PointNumber As Integer
      Const Increment = .1
      ReDim YValues(1 To 21 / Increment) As Single
      PointNumber = 1
      For x = -10 To 10 Step Increment
         y = a * x * x + b * x + c
         '—Store y value in the array
         YValues(PointNumber) = y
         PointNumber = PointNumber + 1
         lstTable.AddItem Format(x, "Fixed") & Chr(9) & Format$(y, "Fixed")
      Next x
      '—Set up picture box with user defined coordinates
      pctGraph.Scale (-10, 20)-(10, -20)
      '—Diagnostic Code
      Dim nl As String, msg As String
      '—New line character
      nl = Chr$(13) & Chr$(10)
      msg = "The Width is " & Str$(pctGraph.ScaleWidth) & nl
      msg = msg & "The Height is " & Str$(pctGraph.ScaleHeight) & nl
      msg = msg & "The Top is " & Str$(pctGraph.ScaleTop) & nl
      msg = msg & "The Leftmost point is " & Str$(pctGraph.ScaleLeft)
```

```
            MsgBox msg
            Debug.Print msg
            '—Draw the x and y axes
            pctGraph.Line (-10, 0)-(10, 0), QBColor(3)
            pctGraph.Line (0, -20)-(0, 20), QBColor(3)
            PointNumber = 1
            '—Plot the graph
            For x = -10 To 10 Step Increment
               pctGraph.PSet (x, YValues(PointNumber))
               PointNumber = PointNumber + 1
            Next x
         End If
      End If
```

Here is an explanation of the code for the Calculate button:

In the first three lines, you declare variables for the coefficients *a*, *b*, and *c*; the discriminant, *Disc;* and the two solutions, *Root1* and *Root2*. Solutions to an equation are often called the roots of the equation.

In the next few lines, you entered the commands to collect the values of the coefficients from the textboxes. If the value of *a* is 0, the equation is not a quadratic equation. The first **If-Then** statement prevents calculations if *a* is 0. The **End If** statement is at the very end of the code.

The value of the discriminant determines whether there are solutions to the equation. The four lines following the **If a<>0** statement calculate the discriminant and test it. If there are no solutions, the label marked Solution 1 is set to "No solution".

If the discriminant is not negative, the equation has solutions that can be calculated. You use the four lines following the **Else** statement to calculate and display the solutions.

In preparation for building the table of *x* and *y* values, you declare variables and constants in the five lines following the **'Generate table** comment. *x* ranges from -10 to 10. *y* is calculated from *x* and the values of *a*, *b*, and *c*. *PointNumber* is a subscript into the *YValues* array. *Increment* is a constant. The value of *x* is increased by this amount each time through the loop. If *Increment* is smaller, more values of *y* are calculated.

The **ReDim** statement reserves space for the *YValues* array. The amount of space is calculated by dividing *Increment* into 21.

You set up the loop for *x* in the line **For x = -10...**, then calculate a value for *y* in the next line. Then you store the *y* value in the *YValues* array. The line **PointNumber = PointNumber + 1** adds 1 to the array

subscript, *PointNumber*. The following line joins the values of *x* and *y* together as strings and adds them to the listbox, lstTable.

Then comes a **Scale** statement, which sets up the coordinate system for the picture box. The range of the *x* values matches the range used in the **For-Next** loop.

The eight lines starting with **Dim n1 As String, msg As String** are diagnostic code. This code displays the settings for the Scale properties. These properties are set by the **Scale** method.

The two lines starting with **pctGraph.Line...** draw the *x* and *y* axes in the picture box. The following line sets *PointNumber* to 1. *PointNumber* is the subscript into the *YValues* array. The **For-Next** loop plots the points in the picture box.

The first **End If** statement finishes the **If Disc < 0** statement. The second finishes the **If *a* <> 0** statement.

Entering Code for the Clear and End Buttons

You have much less code to enter for the Clear and Exit buttons:

1. Enter the following code for the Clear command button. The code clears the textboxes for the coefficients, the labels for the solutions, the listbox containing the table, and the picture box displaying the graph.

```
txtA = ""
txtB = ""
txtC = ""
lblSolution1 = ""
lblSolution2 = ""
lstTable.Clear
pctGraph.Cls
txtA.SetFocus
```

2. In the cmdExit_Click() subroutine, enter **End**.

Finishing Up

Now you are ready to try out the program:

1. Save the project and form files.
2. Run the program. Enter the following values and note the results. After a trial or two, stop the program and comment out the code that creates the message box. Run the program and complete the trials.

a	*b*	*c*
1	1	1
1	1	-1
1	3	-3
1	-3	-3
-1	5	7
-1	5	-7
1	6	9
0	2	9
25	10	1

QUESTIONS AND ACTIVITIES

The following exercises refer to the program presented in this section.

1. Instead of plotting points, alter the program to connect the dots by using the **Line** method to plot line segments.
2. Alter the program to allow the user to plot more than one graph in the picture box, allowing a comparison of functions.
3. Alter the program to allow the user to set the x and y limits of the graph. Instead of using -10 to 10 for x and -20 to 20 for y, allow the user to enter values for each.
4. Mark the coordinate axes (using the **Circle** method to make a large dot) at each unit.
5. It is often important to know where the graph of a quadratic equation crosses the x axis. How does a table of x and y values help a user discover an approximate range for these points? Run the program and, using just the table of values, estimate the roots of the equation where $a = 2$, $b = 5$, and $c = -8$.
6. Why is it necessary to store only the y values generated from the quadratic equation? Why not store the x values in the array as well?
7. What is the SetMode property value for a user-defined coordinate system?
8. Write the Visual Basic commands that initialize the picture box pctGraph for the following custom coordinate systems.

Upper left	*Lower Right*
(-5,5)	(5,-5)
(0,50)	(10,0)
(0,10)	(20,-10)

9. Add a statement or two to the program that prints each value of *x* and *y* into the Debug window.
10. Using the Help system as a resource, find and record a brief description of the kinds of message boxes available and the codes needed to create them. Alter the message box code in the program to use a message box with a big exclamation point.
11. Rewrite the code of the Quadratic Formula program to display an error message if the value for the lead coefficient, *a*, is zero.

A pixel is the smallest graphic element of a display screen. The word is short for "picture element."

Screen dimensions are listed as pixels across by pixels down, for instance, 640x480 is a typical VGA resolution.

A font is a style of type. A printer's point is about 1/72 of an inch. A twip is 1/20 of a point.

Visual Basic provide seven built-in coordinate systems based on twips, points, pixels, characters, inches, millimeters, and centimeters. User-defined coordinate systems are also supported.

The picture box has many uses. It can be:

- Drawn upon with drawing controls like the shape and line controls from the toolbox;
- Drawn upon by graphics methods, like **Line**, **PSet**, and **Circle**;
- A container for labels, textboxes, or other controls from the toolbox;
- A container for an icon, a bitmap, or a metafile.

The image box allows an icon or bitmap to be stretched and distorted.

The value of the ScaleMode property determines the kind of coordinate system used. The ScaleWidth and ScaleHeight properties hold the dimensions of forms and picture boxes. These values depend on the size of the object and the coordinate system chosen in the ScaleMode property.

The **PSet** method will plot a point of a given color on the screen. It syntax is:

object.**PSet***(x,y), color*

where (*x,y*) is the coordinate of the point, and *color* is the color of the plotted point. The first coordinate is the column number (or horizontal position) and the second is the row number (or vertical position).

The **Cls** method clears the contents of a picture box or a form.

The DrawWidth property sets the size of the dot drawn by any of the drawing methods, like **PSet** and **Line**.

This command:

```
p1.Scale (-10,20)-(10, -20)
```

sets up a user-defined coordinate system with (-10,20) as the coordinate of the upper-left corner of the graph and (10,-20) as the lower-right corner of the graph.

You should insert lines of diagnostic code that help troubleshoot programs. When the code has been debugged, the lines of diagnostics can be commented out.

A variety of different MsgBox boxes are available.

The **QBColor** function generates one of 16 colors. The **RGB** function lets the user specify 256 choices each for the red, green, and blue components of a composite color. The color is represented by a long integer.

The **Line** method draws a line between two points. It can draw a box, using the two coordinates as diagonal corners of the box, or a line from the previously plotted point to a point contained in the **Line** statement. The **Circle** method draws a complete or partial circle or ellipse.

1. The Draw a Circle Program

Write a program that allows the user to enter the center and radius of a circle in textboxes. Draw that circle in the default color on a form using (-10,10) as the upper-left corner of the graph and (10,-10) as the lower-right corner of the graph.

2. The Random Circle Program

Write a program to generate and display 50 random circles on a form. Use the same coordinate system described above. Use the built-in random number generator to generate *x* and *y* coordinates for the centers, and the value of *r* for the radius.

The **Rnd** function will return a random Single between 0 and 1. Multiply by 11 to get a Single between 0 and 11. Assign the result to an integer variable. Use that result for the *x* value of the coordinate. Repeat for the *y* value and for the radius of the circle.

Use an InputBox with a default value of 1 to query the user for the shape of the figure to draw. Once that value is entered, use it for each of the 50 random figures.

continued

3. The Skyline Program

Use the Shape and Line controls from the Toolbox to draw a city skyline on a form. Save for use as a background.

4. The Line Graph Program

The data below represents a student's performance on a number of exams. Write a program to enter the pairs of numbers, display them in a listbox, and draw a line graph. Assume there are no more than 10 tests in the list. Use the data and the layout suggested in Figure 7-30.

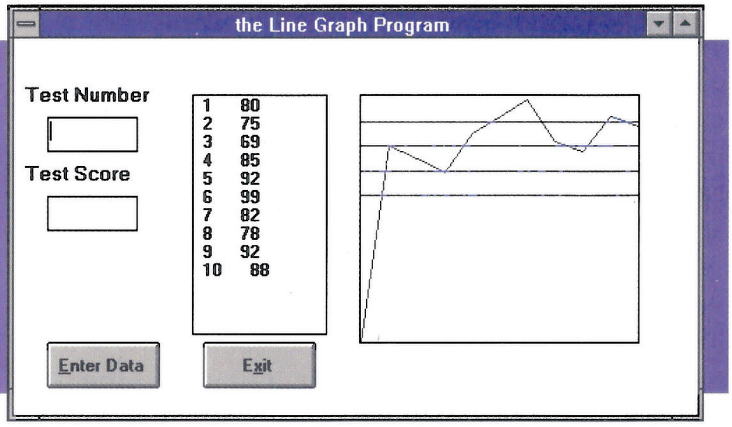


Figure 7-30
Line Graph program

5. The Random Color Program

Draw a picture box on the form. Draw a label on the picture box and change the caption to **Random Color**. Set the Alignment to Center, the FontSize to 13.5, and BackStyle to Transparent.

Declare three integer variables, *Red*, *Green*, and *Blue*. Give each a random value between 0 and 255 by multiplying *Rnd* by 256.

In the Click event of the picture box, insert code to change the BackColor of the picture box to a random color using *Red*, *Green*, and *Blue* in the RGB function. Put the statements in a loop that runs from 1 to 1000.

Run the program, click on the picture box, and note the results.

6. The Crazy Circle Program

In the MouseDown procedure of the default form, declare two Static variables, *Oldx* and *Oldy*. Declare a variable for the radius and calculate using the following formula:

```
Radius = Sqr((oldx - x) ^ 2 + (oldy - y) ^ 2)
```

x and *y* are parameters passed to the procedure. They give the location of the mouse pointer. Set the DrawWidth of the form to 2. Use the **Circle** method to draw a circle with center (*x*,*y*) and a radius of *Radius*.

Reset *Oldx* to *x*, and *Oldy* to *y*.

Once the program runs, change the **Circle** method to draw the circles with random colors.

7. The Linear Equation Solver

In the last section, the Quadratic Program used the quadratic formula to solve a quadratic equation whose coefficients were entered in text boxes. Write a program to solve and graph a linear equation whose coefficients are entered in text boxes.

The standard form of a linear equation is:

$ax + b = c$

The solution to the equation is $x = (c - b)/a$

The equation to graph is $y = ax + b - c$

8. The Falling How Long Program

The complete equation describing the distance, d_f, an object undergoing a constant acceleration a, moving with an initial velocity of v_o, an initial position of d_i, for time t is:

$d_f = d_i + v_o t + at^2/2$

With a slight rearrangement, this becomes:

$y = (a/2)\ t^2 + v_o t + d_i - d_f$

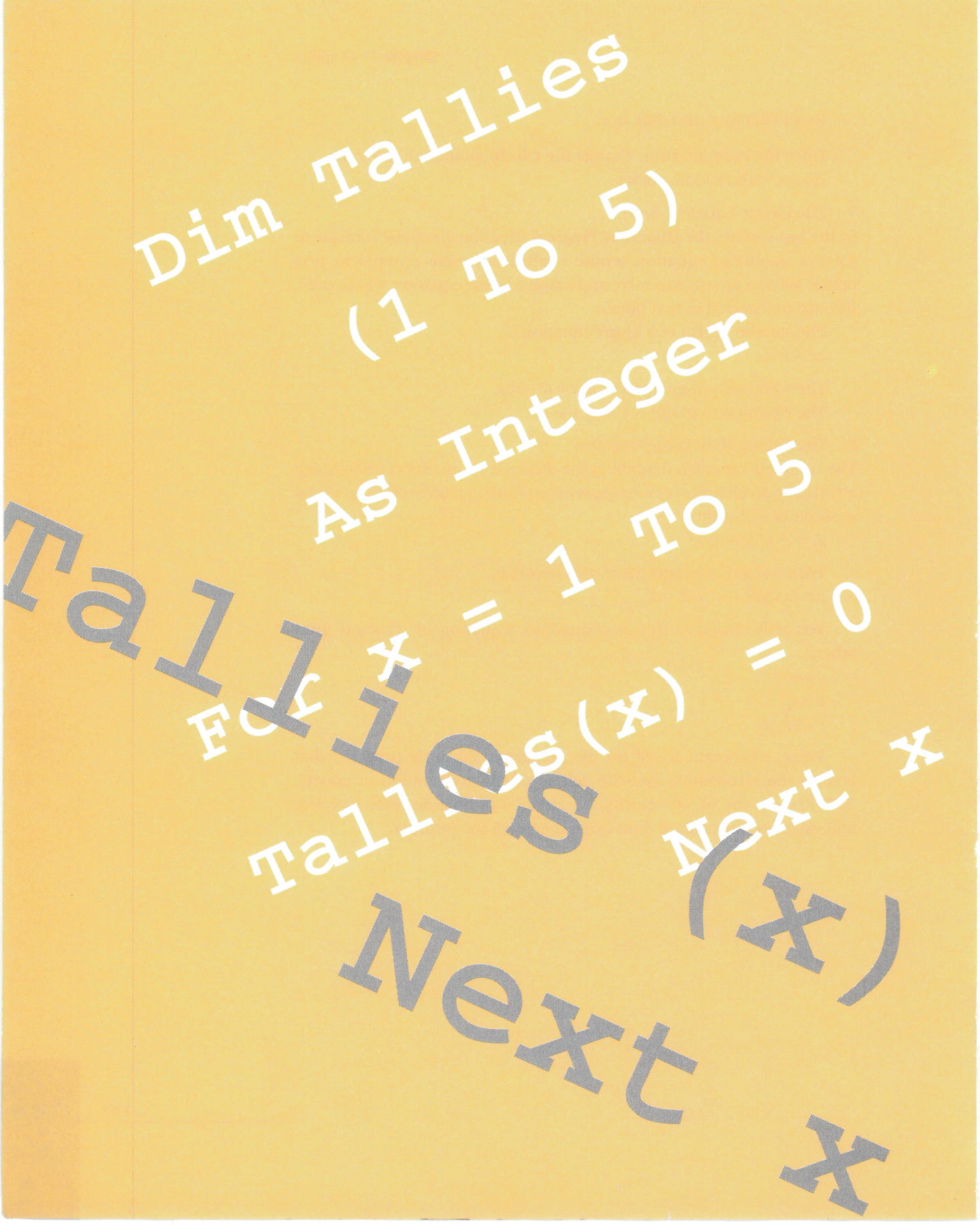
You will recognize this as a quadratic equation in standard form where:

$a = a/2$

$b = v_o$

$c = d_i - d_f$

Write a program based on the Quadratic Formula program that accepts the acceleration (set the default to the acceleration of gravity, 32.2 ft/sec^2), the initial velocity, the initial and final distance traveled, and calculate the elapsed time using the quadratic formula.


Dim Tallies
(1 To 5)
As Integer
For x = 1 To 5
Tallies(x) = 0
Next x
Tallies(x)
Next x