6. Write **Do-While** loops to replace these **For-Next** loops:

   a)
   ```
   For x = 1 To 25
      ...
   Next x
   ```

   b)
   ```
   For w = 0.5 To 10 Step 0.1
      ...
   Next w
   ```

7. Where do you find the **Exit Do** statement, and what does it do?
8. What happens when an **Exit Do** statement is executed from the inner of two nested **Do-While** loops? Draw a diagram to explain your answer.
9. Compare the relative merits of using a textbox or **InputBox** to enter values into a program.

The Line Drawing Program

This project uses graphic methods, mouse events, arrays, and binary files to build a simple drawing program. It uses graphics methods and mouse events to build a line drawing program. The coordinates of the endpoints of the lines drawn are stored in arrays. When the drawing is complete, the coordinates are stored in a file. Drawings can be loaded from files.

Starting Out

Before you can build a program that draws circles and lines on a form, you need to explore two methods:

- **Circle** method
- **Line** method

The code for these methods can be placed in several places, including in mouse events. Before you start the Line Drawing program, then, you need to explore the effects of placing the code in different subroutines.

CIRCLE AND LINE METHODS

The **Circle** method is a built-in procedure used to draw circles on an object. The syntax is:

object.**Circle** (x,y),radius

The object, in this case, is the form, a Picture Box control, or the Printer object. The center of the circle is (*x,y*). *X* and *y* are screen coordinates. *Radius* is the radius of the circle. If you omit an object from the syntax, the program draws the circle on the current form.

The **Line** method is a built-in procedure you use to draw lines on an object.

object.**Line** (x,y)-(x',y')
object.**Line** -(x,y)

This method draws a line on an object. As is true of the **Circle** method, the object can be a form, a Picture Box control, or the Printer object. If you omit an object, the program draws the line on the current form. The first version of the **Line** method draws a line between (*x,y*) and (*x',y'*). The second version draws a line from the last point drawn to (*x,y*). The coordinates of this last point are saved in the CurrentX and CurrentY properties of the object. When you use the **Circle** method to draw a circle centered at a point (*x,y*), the last point drawn is set to that point (*x,y*).

MOUSE EVENTS

A MouseDown event occurs when a user presses a mouse button. A MouseUp event occurs when the user releases a mouse button. The parameters of the event procedure tell you which buttons are pressed or released, as well as the coordinates of the point where the press or release occurred. A MouseMove event occurs when the mouse's position changes. The parameters to the MouseMove event tell you the current coordinates of the mouse cursor.

These three mouse events each have four parameters. In this section, only the last two are important: X As Single, Y As Single. These provide the screen coordinates of the mouse pointer.

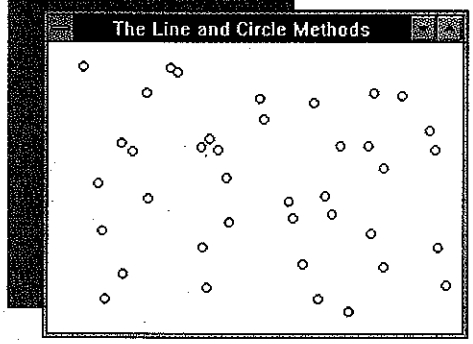
The mouse events let you program the mouse with finer control than the Click event gives you. Click events occur after a mouse button is both pressed and released. When the Click event occurs, you are not told the coordinates of the mouse cursor. In the program you are going to write, you will respond to mouse events by saving the mouse cursor coordinates when a mouse button is pressed, saving the coordinates when a button is released, and drawing a straight line between those two points. Clearly, the Click event does not provide enough information to accomplish this.

To experiment with the **Line** and **Circle** methods, follow these steps.

1. Open Visual Basic. If Visual Basic is already open, select New Project from the File menu.
2. Change the caption of the default form to **The Line and Circle Methods**.
3. Double-click on the form to open the Code window.
4. In the Proc drop-down menu, select the MouseDown event.
5. In the Form_MouseDown (...) event procedure, insert the following code. Here, *X* and *Y* are parameters of the MouseDown event giving the coordinates of the mouse cursor:

```
Const sglRadius = 50
Circle (X,Y), sglRadius
```

Figure 6-24
Drawing circles on the form



6. Run the program. Move and click the mouse on various spots on the form. Every time you press any mouse button, a small circle appears, centered at the point where you pressed the mouse button (see Figure 6-24). No action is taken in response to releasing mouse buttons or to mouse movement.
7. Stop the program.

Now, you will make the program more general by allowing the user to specify a different radius each time the program is run. The user can enter a new radius, or just accept the default value of 50:

1. Reopen the Code window.
2. Remove **Const sglRadius = 50** from the MouseDown event procedure.
3. Add this line to the MouseDown event:

```
Debug.Print X,Y
```

4. Add this line to the general declarations section of the form:

```
Dim gsglRadius As Single
```

5. Add this code to the Form_Load procedure (select Form from the Object drop-down list and Load from the Proc drop-down list):

```
gsglRadius = Val(InputBox("Enter a value for the radius",,"50"))
```

6 Change the name of the radius variable in the Form_MouseDown event from *sglRadius* to *gsglRadius*, so that the statement that draws the circle looks like this:

```
Circle (X,Y), gsglRadius
```

7 Run the program. An InputBox appears, prompting you to enter a radius. Enter a radius and click OK. Now all the circles that appear will be of that radius.

8 Click on the form in various places. Each time you press a mouse button, a circle appears with the radius you selected.

9 Click on the Debug window. The *X* and *Y* coordinates are displayed of the center of each circle you drew.

10 Stop the program, then run it again. In the InputBox, enter a different radius, and repeat the steps. When you are finished, stop the program.

Now, you will respond to the MouseUp event by using the **Line** method to draw a line between the points at which a mouse button was pressed and released. To do so:

1 Reopen the Code window.

2 Insert this line of code in the MouseUp event procedure:

```
Line -(X,Y)
```

3 Run the program. Enter a radius, as prompted.

4 Press a mouse button. Due to the instructions in the MouseDown event, the **Circle** method draws a circle on the form with the radius you entered, centered at the point where you pressed the button.

5 Now, press a mouse button *and drag* before you release the mouse button. The **Circle** method draws a circle due to the MouseDown event. When you release the mouse button, a MouseUp event occurs. As a result, the instructions in that event procedure are executed and a line appears.

6 Stop the program.

7 Save the project and form files.

If you press and then release a mouse button without moving the mouse, a line is still drawn. However, the beginning and endpoints of the line are identical, so the line is not visible.

Setting Up the Form

Setting up the appearance of this simple drawing program is easy. As you can see in Figure 6-25, you need a menu bar with a single entry (File). This menu opens a drop-down list of four commands. There are no buttons or other objects on the form.

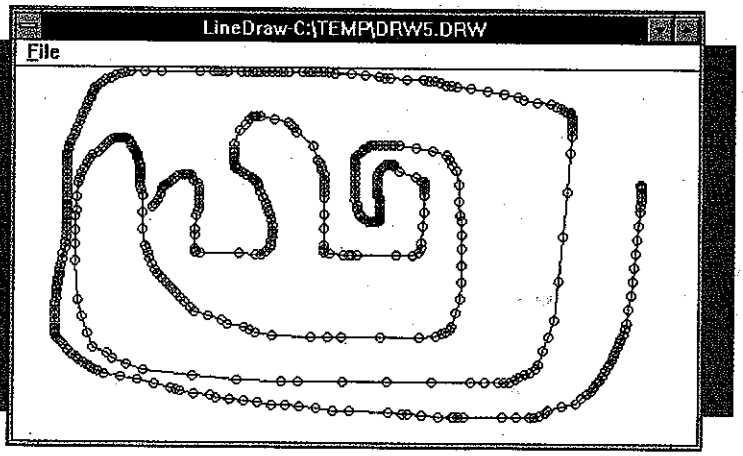


Figure 6-25
A simple drawing program

To set up the form:

1. Create a directory to store the Line Drawing project.
2. Start Visual Basic, or choose New Project from the File menu.
3. Change the caption of the form to **LineDraw**. Change the name of the form to **frmDrawLine**. Change the AutoRedraw property of the form to True. This redraws the contents of the form after a dialog box has appeared and vanished from the form, erasing part of its contents.

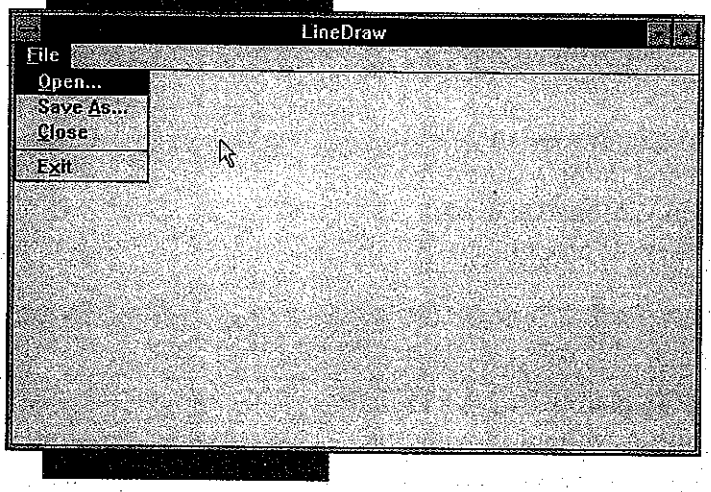


Figure 6-26
File menu

4. Create a menu for the form with the structure shown in Figure 6-26 (see Chapter 5, section 2 for a discussion of how to set up a menu):

Entering Code: Stage 1

Now, add the following code and test the program:

1. In the general declarations section of the form, put the following statements:

```
Option Explicit
Dim gfDrawing As Integer
```

The *gfDrawing* variable is used as a flag. For more detail on flags, see the textbox.

2 Open the form's MouseDown event Code window and enter the code:

```
gfDrawing = True  '—used by MouseMove handler
Circle (X,Y),50
'—Set form's CurrentX, CurrentY properties
CurrentX = X
CurrentY = Y
```

The CurrentX and CurrentY properties provide coordinates used by the next drawing method. The **Line** method uses these coordinates as the first point of a line. True is a constant in Visual Basic equal to -1.

3 Run the program.

4 Press and release a mouse button on the form several times. The **Circle** method draws a circle every time a MouseDown event occurs. That circle has a radius of 50.

5 Stop the program.

6 Save the project and form files in the directory created from the project.

The changes you just made have produced a less complex program than you had before. You have laid the foundation for the next stage, in which you use the flag *gfDrawing* to determine whether or not to draw anything. In the next section, you will clear the *gfDrawing* flag in the MouseUp event. You will also draw in the MouseMove event, only if the flag is set.

Entering Code: Stage 2

This section adds the code that draws the images. You'll find the code surprisingly simple. The **Line** and **Circle** methods are used. Nothing at all is drawn if the flag, *gfDrawing*, is set to **False**.

1 Enter the following code in the MouseMove event procedure:

```
If gfDrawing Then
   Line -(X,Y)
   Circle (X,Y),50
End If
```

FLAGS

…or … Then … usually … (of Integer type) which are set and cleared in certain procedures to signal that some condition does or does not hold true. The flag is then tested in other procedures, and actions are taken depending on the state of the flag. A common example is the use of a flag to indicate whether a mouse button is down or not.

In this exercise, the variable's prefix, *gf*, indicates that the variable is visible to any procedure in the form. Visual Basic does not automatically make the variable global because of the "g" in the prefix. You made it global by placing it within the general declarations area of the form. When you use the variable in an event procedure, its prefix makes it easy to remember that you are using a global variable. The "f" portion of the prefix indicates that the variable contains a flag.

2 Enter the following code in the MouseUp event procedure:

```
gfDrawing = False
```

3 Enter the following command in the mnuExit_Click() event procedure:

```
End
```

4 Run the program. Press a button, drag, and release the button. Also try pressing and releasing buttons without dragging.

5 Stop the program, then save the project and form files.
The **Circle** method draws a circle whenever either a MouseMove or MouseDown event occurs. You don't, however, get a MouseMove event for every point that the mouse moves over. The faster you move the mouse, the more sparse the circles become.

Entering Code: Stage 3

Your next task is to deal with file input/output (i/o). Suppose, for example, that the user of your program has created a drawing and now wants to save it. How do you save a drawing?

Remember the list of *x* and *y* coordinates you saw displayed in the Debug window? You can identify lines by the *x* and *y* coordinates of their endpoints. You can identify circles by the *x* and *y* coordinates of their centers. Therefore, you need to set up two arrays, one to record *x* coordinates and one to record *y* coordinates. Procedures can then save the lists of coordinates to a file and read lists of coordinates from files. After reading a list of coordinates from a file, the drawing that those coordinates describe can be recreated.

To set up an array, you need to pick a maximum number of entries. In this program, you use two arrays, both of the same size. An adequate number for the maximum number of entries is 2000.

As you work through these steps, you will also experiment with creating procedures that are not linked to particular events. One of these procedures (Sub DrawCircle) replaces calls to the **Circle** method in the MouseDown event. In addition to drawing circles, this procedure saves coordinates in the arrays.

To create the arrays:

1 Add the following lines to the general declarations section.

```
Const gnMaxPoints = 3000     ' gn = global number
'--gsng = global single
Dim gsngX(1 To gnMaxPoints) As Single
Dim gsngY(1 To gnMaxPoints) As Single
```

```
'—Actual number of points recorded
Dim gnNumPoints As Integer
'—gstr = global string
Const gstrTitle = "LineDraw"
```

2 Create a new procedure by entering the following lines in the general section:

```
Sub DrawCircle (X As Single, Y As Single)
'—Draw the circle
   Circle (X, Y), 50
'—Too many points?
   If gnNumPoints < gnMaxPoints Then
'—Add one more point
      gnNumPoints = gnNumPoints + 1
'—Save X and Y coordinates
      gsngX(gnNumPoints) = X
      gsngY(gnNumPoints) = Y
   End If
End Sub
```

3 In the MouseDown and the MouseMove procedures, replace

```
Circle (X,Y),50
```

with

```
DrawCircle X,Y
```

4 In the Form_Load procedure, enter the following line:

```
gnNumPoints = 0
```

5 Select mnuClose from the **Object** drop-down list in the Code window. In the mnuClose_Click procedure, enter the following code.

```
gnNumPoints = 0
frmDrawLine.Cls
frmDrawLine.Caption = gstrTitle
```

6 Select mnuSaveAs from the Object drop-down list. In the mnuSaveAs_Click procedure, enter the following code.

```
'—Collect filename from user
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "OpenFile")))
'—Open binary file for output
Open strFn For Binary Access Write As #1
'—Save actual number of points
```

continued

PROC

a f
mod
proce **ect**
New P **om the**
View m **hen, you choose Sub or Function and supply a name.**

Another option is to open the general declarations section of a form's code. Under the declarations, enter the top line of the new procedure. As soon as you press Enter at the end of that line, Visual Basic creates the procedure, automatically adding the last line (End Sub). (To see any declarations in the general declarations area, you now have to select declarations from the Proc drop-down list.) Visual Basic will now list this procedure in the Proc list.

```
Put #1, , gnNumPoints
'—Local variable for loop
Dim i As Integer
'—Loop to actual number of points
For i = 1 To gnNumPoints
   '—Save coordinates
   Put #1, , gsngX(i)
   Put #1, , gsngY(i)
Next i
Close #1
'—Reset form caption
frmDrawLine.Caption = gstrTitle & "-" & strFn
```

7 The Open command collects the coordinates of the points and loads those coordinates in two arrays. Once the points are loaded, the DrawLines procedure draws the lines and circles. In the general section of the form, enter the following code.

```
Sub DrawLines ()
   Dim i As Integer
   CurrentX = gsngX(1)
   CurrentY = gsngY(1)
   '—Draw the first circle
   Circle (gsngX(1), gsngY(1)), 50
   '—Plot the rest of the lines and circles
   For i = 2 To gnNumPoints
      Line -(gsngX(i), gsngY(i))
      Circle (gsngX(i), gsngY(i)), 50
   Next i
End Sub
```

8 Select mnuOpen from the Object drop-down list. In the mnuOpen_Click procedure, enter the following code.

```
'—Collect filename from user
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "OpenFile")))
'—Open binary file for input
Open strFn For Binary Access Read As #1
'—Save actual number of points
Get #1, , gnNumPoints
'—Local variable for loop
Dim i As Integer
'—Loop to actual number of points
```

```
For i = 1 To gnNumPoints
'-Collect coordinates from file
   Get #1, , gsngX(i)
   Get #1, , gsngY(i)
Next i
Close #1
'-Reset form caption
frmDrawLine.Caption = gstrTitle & "-" & strFn
frmDrawLine.Cls
DrawLines
```

9 Run the program. Click and drag, then release.

10 Choose Save As from the File menu. Supply a complete pathname.

11 Choose Close from the File menu.

12 Choose Open from the File menu. Supply the same complete pathname.

The code for Open is almost identical to the code for Save As. Get is substituted for Put. At the end of the routine, the form is cleared and the circles and points are drawn with the DrawLines procedure.

Finishing the Program

Now, use the program:

1 Experiment with drawing, saving, opening, and closing.

2 Stop the program.

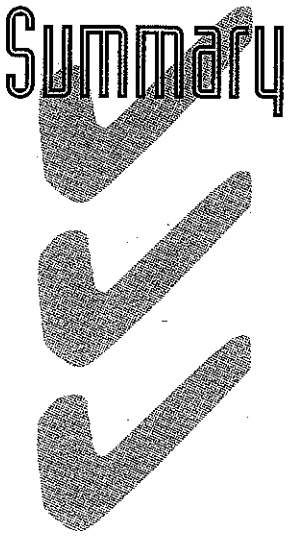
3 Save the project and form files.

4 Choose Make EXE File to create a stand-alone application.

QUESTIONS AND ACTIVITIES

1. Comment out the **Circle** statement from MouseDown, DrawCircle, and DrawLines. Run the program and observe the effect. Restore the **Circle** statements.
2. Change the declaration for *strFn* from local to global. Move the **Dim** statement to general declarations. Initialize *strFn* in the Form_Load procedure to "". Rewrite the **InputBox** functions to use *strFn* as a default value.
3. Although drawing starts and stops in response to mouse events, once they are saved and reloaded, the breaks between the connected lines are lost. Currently there is no record of where drawing

stops and starts. Modify the MouseUp event procedure to record coordinates (-1,-1) when the mouse button is released. Modify DrawLines to stop drawing when the (-1,-1) coordinates are encountered.



The **Format$** function converts values into strings according to certain fixed patterns or according to patterns you provide. The syntax is:

variable = **Format$**(*value, format string*)

Visual Basic shares code through

- Custom controls
- Code modules

To use a custom control, you must place the control's VBX file in the System folder of Windows. Visual Basic will add the filename to the project file list that is displayed in the Project window. You can place procedures you want available to more than one form in a code module.

The scope of a variable is determined by its visibility and lifetime. Visibility is a question of what procedures can use (or "see") a variable. The lifetime of a variable depends on whether the value of the variable is available apart from the limited lifetime of an event procedure. A variable declared and used in one event procedure is not available in another. Variables declared in the form module are available to all event procedures within the form. Variables declared with the **Global** statement in a code module are available to any form in the project and its event procedures.

A static variable, declared within an event procedure, isn't available to other procedures, but its value does persist beyond the end of the procedure in which it is declared. When the procedure is reentered, the static variable is there, with its old value, available for use.

The **Option Explicit** statement is entered in the general declarations section of either a form module or a code module. By using this statement, you require that each variable be declared before use.

You use arrays so that you can save a list of values. The list has a single name. Each element of the list has a unique whole number subscript. For instance, an array of names can be declared as follows:

```
Dim Names(100) As String
```

Another way to declare the same array:

```
Const MaxNames = 100
Dim gstrNames(1 To MaxNames) As String
```

Using 1 To MaxNames lets the programmer specify the starting subscript and use a constant as a maximum subscipt. Using a constant makes it easy to change the references to that value throughout the program with a single change.

The first characters of the name, *gstr*, shows the data type of the variable. The prefix *g* indicates the variable is global; the *str* shows that the variable contains strings.

The backslash (\) operator divides two numbers and gives the whole number result.

Scrollbars can be used to provide a subscript value to access the elements of an array.

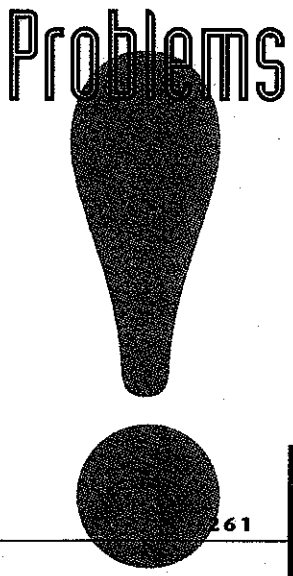
The **Do-Loop** statement, one of a number of indefinite loop statements, allows a loop to execute an indefinite number of times. The number of times the loop is executed is controlled by factors within the loop. A sentinel-controlled loop continues until a certain value, called a sentinel, is entered or generated. The syntax of this statement is:

Do While *condition*
    statements to execute while condition is true
Loop

The **InputBox** function allows the entry of values without textboxes. The syntax is:

variable = **InputBox** *(Prompt, Caption, Default value)*

Prompt is a string containing the instruction to the user. *Caption* is the caption of the window. *Default value* is a string representing a default value for the variable. If a user presses Enter without changing the default value, this value is assigned to the variable.



1. The How-Long-in-School Problem

Write a program using textboxes for input, prompting the user to enter the number of hours a day spent in school and the number of daylight hours in a day. Calculate and display the percentage of daylight time spent in school. Use the **Format$** function to display the percentage.

2. The Sales Tax Problem

Write a program using InputBoxes to enter the amount of a purchase and the percentage sales tax charged. In Illinois the sales tax is 6.25%. You would enter that percentage as 6.25, omitting the percentage sign. Using the Currency and Percent format strings, display the original amount of the purchase (as currency), the sales tax percentage, the sales tax charged (as currency), and the total price including sales tax (as currency).