Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Appendix B

Graphics, Animation, Drag and Drop

This appendix presents topics related to graphics. The first section discusses how to draw graphs in a control. After you have an understanding of graphic techniques, you will be able to explore topics in animation, which is discussed in Section B.2. Another interesting application of graphics is the drag-and-drop operation, which can provide the user a convenient way to specify operations that can otherwise be fairly complex to describe. This type of application can be exciting and dynamic. Drag and drop is discussed in Section B.3.

B.1. Drawing Graphs

VB 2008 provides support for graphics with several namespaces: The ***System.Drawing*** namespace provides access to GDI+ (improved version of Graphic Device Interface) basic graphics functionality. More advanced functionality is provided in theSystem.Drawing.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text namespaces.

The discussion here is limited to the functionality provided in the System.Drawing namespace. To draw a graph on a control, there must be a graphics object (representing the drawing surface) associated with the control. The graphics object provides various methods to perform drawing. The following table gives a selected list of these methods:

Method	Use
DrawEllipse	Draw an ellipse within a specified rectangle.
DrawLine	Draw a line between two specified points.
DrawPolygon	Draw a polygon with the specified points.
DrawRectangle	Draw a rectangle over two specified points.
DrawString	Write specified text with a specified font at a specified location.

Each method requires a pen that specifies the color and the pen size in pixels. Different methods require different additional parameters. For example, the DrawEllipse and the DrawRectangle methods require a rectangle structure as the second parameter, while the DrawLine method requires two points.

The drawing methods draw an outline for the specified shapes. To paint the interior of the shape with a color, you use the corresponding fill methods. For example, to fill an ellipse, you will use the ***FillEllipse method***. Before discussing these methods in detail, let's first take a look at a few concepts that are fundamental to graphics.

Basic Concepts

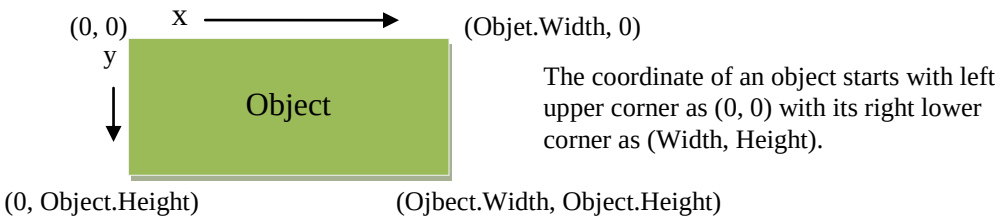
When you are drawing a graph, you will always need to decide the position to begin or end your drawing. The coordinate represents the numeric value of the position. The color

you choose can affect the attractiveness of a graph significantly. The following discussion introduces you to the basics of coordinates and colors.

Coordinates

VB 2008 supports drawing on several controls, including the form, picture box, label, and the panel. The object on which you draw a graph has a coordinate system that is different from the familiar Cartesian coordinate system. In the latter system, the horizontal coordinate, x, increases from left to right, and the vertical coordinate, y, increases from bottom to top. In contrast, with an object, the horizontal coordinate, x, also increases from left to right, with the leftmost margin being 0. The vertical coordinate, y, however, starts from top to bottom, with the topmost margin being 0. (The standard measurement unit is the pixel.) This system is depicted in Figure B-1.

Figure B-1
VB object's coordinate



You need to be aware of this difference so that you can draw on an object correctly.

Colors

In Visual Basic 2008, colors are of type System.Drawing.Color. The System.Drawing namespace provides a list of standard colors, so to specify the blue color, you code:

```
    Color.Blue
```

At run time, you can allow the user to visually choose a color using the ***color dialog,*** a control that behaves similarly to the open file dialog. For example, its ***ShowDialog method*** will display a dialog box with all available basic colors. If the user clicks its Define Custom Color button, the right pane will show a color pane from which the user can select the desired color.

The Graphic Object

To draw any graph on a control, there must be a Graphic object associated with the control. You can use the control's ***CreateGraphic method*** to create the Graphic object. The object is also available as an argument of the Control's ***Paint event***, which occurs when the control is drawn or redrawn. In the case of the form, the event occurs after the form is activated. You can also trigger this event by invoking the control's ***Invalidate method***. To illustrate, suppose you want to draw a horizontal line between Point (0, 100) and Point (100, 100) on the form when the project starts. You can use the Graphics object's ***DrawLine method***, which has the following syntax:

```
    Object.DrawLine(Pen, Point1, Point2)
```

Where Pen = a Pen object specifying the color and an optional parameter specifying the pen width, and
Point1, Point2 = Point object specifying the coordinate of the point; each can also be specified by two numbers representing the coordinate without the explicit Point type.

The following code will draw a blue line with a pen size of three pixels between the specified points on the form:

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
    e.Graphics.DrawLine(New Pen(Color.Blue, 3), New Point(0, 100), New
      Point(100, 100))
End Sub
```

The Pen object specifies the blue color and a draw width of 3 (pixels). The first point specifies a coordinate of (0, 100). The first value gives the X value; the second, the Y value. Note that the statement can also be coded as:

```
    e.Graphics.DrawLine(New Pen(Color.Blue, 3), 0, 100, 100, 100)
```

If you are a reviewer of the code, you may prefer the first format because it gives a clearer meaning of those numbers used in the statement.

Free-Hand Drawing

The DrawLine method can be used to provide the user the free-hand drawing capability with the mouse. For example, the MouseDown event is triggered when the user presses the mouse down. Its event arguments return the button (left, middle, or right) and the coordinate at which the button is pressed. You can use the event to provide the drawing capability as follows:

- When the right mouse button is pressed, the program will start the drawing process. Each time the right mouse button is pressed thereafter, a line is drawn between the current point and the previous point where the mouse was pressed.
- When a different mouse button is pressed, the drawing process discontinues.

Because you will write the code in the MouseDown event (which does not provide a graphic object as an event argument), you must create a graphic object that is associated with the form to be able to perform the drawing. You can create the object as soon as the project starts. The code appears as follows:

```
Dim TheGraph As Graphics
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    'Create a graphic object associated with the form
    TheGraph = Me.CreateGraphics()
End Sub
```

When performing the freehand drawing, the routine needs to decide whether to draw the line when the MouseDown event is triggered. It should draw a line only if the right mouse button was pressed. You will use a Boolean variable, IsDrawing, to track this state. The button pressed can be tested with the following If statement:

```
   If e.Button = MouseButtons.Right Then
```

The event procedure appears as follows:

```
Private Sub Form1_MouseDown(ByVal sender As Object, ByVal e As
  System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseDown
    Static IsDrawing As Boolean
    Static PreviousPoint As New Point()
```

```
    Dim CurrentPoint As New Point()
    Dim ThePen As New Pen(Color.Blue, 2)

    If e.Button = MouseButtons.Right Then
        CurrentPoint = New Point(e.X, e.Y)
        If IsDrawing Then
            'Draw a line when the right mouse button is pressed
            TheGraph.DrawLine(ThePen, PreviousPoint, CurrentPoint)
        Else
            'Set to start drawing
            IsDrawing = True
        End If
        'Keep the current point to draw the next line
        PreviousPoint = CurrentPoint
    Else
        IsDrawing = False
    End If
End Sub
```

As you can see from the code, the procedure first tests whether the button pressed is the right button. If it is, the position of the mouse is kept as CurrentPoint. It then decides whether to draw the line based on the value of IsDrawing, which is turned on by the first right button press and turned off when the button pressed in not the right button. Note that IsDrawing and PreviousPoint are declared to be Static because their values need to be preserved between the event procedure calls.

You may find it more intuitive for the user to perform free-hand drawing by dragging the right mouse with the line starting where the right mouse is pressed and ending at the point where it is let go. This modification is left to you.

The DrawRectangle and DrawEllipse Methods

The DrawRectangle method draws a specified rectangle on the object. If the width and height are the same, the result is a square. The method has the following syntax:

```
    Object.DrawRectagle(Pen, Rectangle)
```

The Rectangle is a structure defined by a point specifying the location of its upper left corner, and a size specifying the width and height.

The DrawEllipse method draws an ellipse within a specified rectangle on the object. If the width and height are the same (and thus the shape is actually a square), the result is a circle. The method has the following syntax:

```
Object.DrawEllipe(Pen, Rectangle)
```

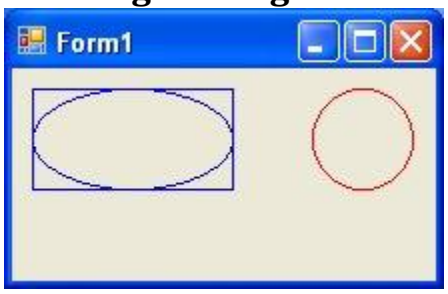
The following code example produces the graphs shown in Figure B-2.

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
    e.Graphics.DrawRectangle(New Pen(Color.Blue), New Rectangle(New
      Point(10, 10), New Size(100, 50)))
    e.Graphics.DrawEllipse(New Pen(Color.Blue), New Rectangle(New
      Point(10, 10), New Size(100, 50)))
    e.Graphics.DrawEllipse(New Pen(Color.Red), New Rectangle(New
      Point(150, 10), New Size(50, 50)))
End Sub
```

The first statement draws a rectangle of 100 x 50. The second statement draws an ellipse of the same size at the same point. As you can see, the ellipse is defined by the boundary

of the rectangle. The third statement draws an ellipse defined by a rectangle of the same width and height of 50. The result is a circle.

Figure B-2
Drawing rectangle and ellipses



The Fill Methods and the Brushes

As mentioned at the beginning of this appendix, the draw methods draw the outlines of shapes, while the fill methods paint the shapes with colors. For example, the ***FillEllipse method*** fills an ellipse with a color. Instead of using a pen as in the case of the draw methods, the fill methods use a brush. The System.Drawing namespace provides several brush objects, which are listed in the following table:

Type of Brush	Explanation	Code Example (reference)
SolidBrush	Fills the specified area with the color specified for the brush	`SolidBrush(Color.Red)`
Brushes	Provides the specified solid brush with named standard color	`Drawing.Brushes.AliceBlue`
SystemBrushes	Provides the specified brush with named system color (such as Control, Window)	`Drawing.SystemBrushes.ActiveCaption`
TextureBrush	Uses an image to fill the specified area	`Drawing.TextureBrush(TheImage)`

The Brushes object provides brushes for all standard colors. The SolidBrush object allows any color of your choice. If you choose a standard color for the SolidBrush, the effect will be exactly the same as using one of the brushes provided by the Brushes object. The SystemBrushes object provides brushes of the system colors. These are the colors you can find in the System tab for the foreground or background color property in the Properties window.

The FillEllipse method has the following syntax:

```
Object.FillEllipse(Brush, Rectangle)
```

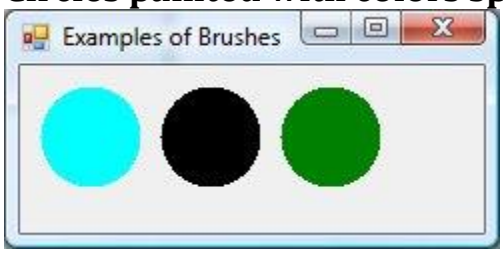
The following code produces the three circles as given in Figure B-3.

```
Dim TheColor As Color
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
```

```
    cdlColor.ShowDialog()
    TheColor = cdlColor.Color
End Sub
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
    With e.Graphics
         .FillEllipse(Drawing.Brushes.Aqua, New Rectangle(New Point(10,
10), New Size(50, 50)))
        .FillEllipse(Drawing.SystemBrushes.ActiveCaptionText, New
Rectangle(New Point(70, 10), New Size(50, 50)))
        .FillEllipse(New SolidBrush(TheColor), New Rectangle(New
Point(130, 10), New Size(50, 50)))
    End With
End Sub
```

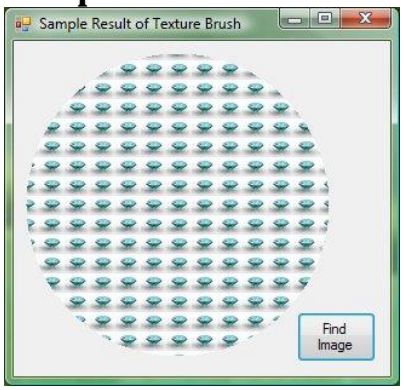
When the project starts, it prompts for a color from the user, using the color dialog. (To test the program, draw a ColorDialog control on the form and name it **cdlColor**.) This color is then used as the argument to the SolidBrush in the third statement in the Form Paint event procedure. The first statement uses the Aqua brush of the Brushes object to paint the circle, while the second statement uses the ActiveCaptionText brush of the SystemBrushes object to paint the second circle.

Figure B-3
Circles painted with colors specified by different types of brushes



The TextureBrush object uses a specified image to fill an area. The following code initially paints a circle with the AliceBlue color when the project starts, but when the user clicks the Find Image button and selects an image, the circle is filled with that image. Figure B-4 shows a sample result using a diamond image as the texture brush.

Figure B-4
Sample result of texture brush



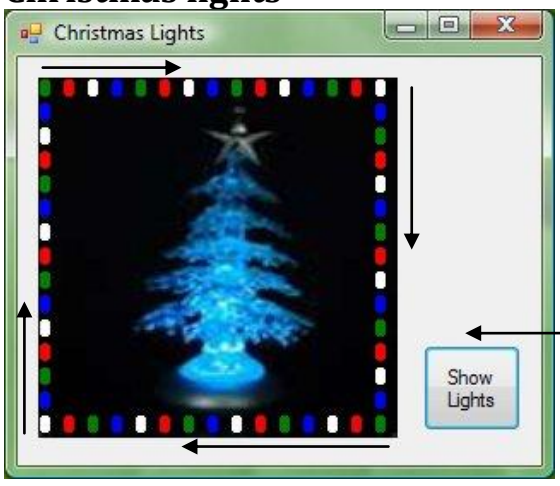
```
Dim TheImage As Image
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
    Dim TheBrush As Brush
    If TheImage Is Nothing Then
        TheBrush = Drawing.Brushes.AliceBlue
    Else
        TheBrush = New Drawing.TextureBrush(TheImage)
    End If
    e.Graphics.FillEllipse(TheBrush, New Rectangle(New Point(10, 10),
New Size(150, 150)))
End Sub
Private Sub btnFind_Click(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles btnFind.Click
    Dim TheImageFile As String
    cdlOpenFile.Title = "Open Image File"
    cdlOpenFile.ShowDialog()
    TheImageFile = cdlOpenFile.FileName
    TheImage = Image.FromFile(TheImageFile)
    'Force the form the repaint
    Me.Invalidate()
End Sub
```

Notice how the form is forced to repaint in the Find Image button's click event procedure. The form's *Invalidate method* is called. The method "invalidates" a region of the form, and therefore causes the form to repaint that region. In this case, no region is specified, so the entire form is repainted when the form's Paint event is triggered. Combined with colors, the different shapes produced by the FillEllipse method can produce an interesting presentation. For example, you can draw ellipses on the sides of a picture box with assorted colors. These ellipses can look like Christmas lights.

Drawing Christmas Lights: An Example

This example develops a project that will draw Christmas lights on the sides of a picture box. When the picture is complete, it will look like Figure B-5.

Figure B-5
Christmas lights



This project draws Christmas lights around the sides of the picture box starting at the upper left corner. Lights are drawn clockwise as the arrows show. The Christmas tree is loaded to the image property of the picture box at design time. You can download a similar picture from the internet although you do not have to have the tree to complete this project.

To create the picture, follow these steps:

1. Draw a picture box on a new form, adjusting its size and location.
2. Name the picture box **picLights** and set its BackColor property to black.
3. Set the picture box's Image property to a Christmas tree picture. You can download one from websites that feature clip art. You do not have to have a tree picture to do this project, although the result will look nicer if you do.
4. Draw a button on the form. Name it **btnShow** and set its Text property to **Show Lights**.
5. Develop the code to draw the lights.

The remainder of this subsection considers the issues pertaining to this project as listed below:

- How to compute the size of the lights and the distance between them?
- How to compute the position of each light and draw it?
- How to alternate the colors for the lights?

Computing the Distance and Size of the Lights

The distance and size of the lights must be computed before these values can be used to determine the position of each light. To make your job easier, you will force the picture box to be square, using the longer of the width and height. You will then draw 14 lights on each side.

Let *Distance* = the distance between the lights, and
BulbHeight = the height of each light.
You can then compute the two variables as follows:

```
    Distance = Width / 15
    BulbHeight = 80% of Distance
    BulbSize = 67 % of BulbHeight
```

Although 14 lights will be drawn on each side, 15 lights will actually appear on each side because the first light of each side will start at a corner, which will overlap two sides. Note also that the distance is measured from the upper left corner of the rectangle that defines each light; however, such a value will cause the lights to overlap each other when they are aligned vertically. An 80% of that maximum value appears to keep a proper distance between lights. This is how the preceding formula for the bulb height is derived. The bulb size will have a width of approximately two-thirds of the bulb height.
The following Form Load event procedure will carry out the needed computations.

```
Dim Distance As Integer
Dim BulbHeight As Integer
Dim TheGraph As Graphics
Dim TheBrush As SolidBrush
Dim BulbSize As Size
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    'Set the picture box squared
    With picLights
        If .Height > .Width Then
            .Width = .Height
        Else
            .Height = .Width
        End If
```

```
        'Compute distance between each light bulb
        Distance = CInt(.Width / 15)
    End With
    ' Radius is a bit shorter than half distance
    BulbHeight = CInt(0.8 * Distance)
    BulbSize = New Size(CInt(0.67 * BulbHeight), CInt(BulbHeight))
    TheGraph = picLights.CreateGraphics()
End Sub
```

Notice that both Distance and BulbSize will be used by other procedures; therefore, they are declared as the class-level variables.

In the procedure, the If block compares the picture box's height with its width and changes the shorter one to match the longer to make the picture box square. The Distance variable is then computed by dividing the picture box's Width property by 15. Notice also that a graph object associated with the picture box is created at the end of the procedure. It is important that this statement be executed only after the size of the picture box is fixed. If it is placed before a change in the picture box's size, the object may not perform the graphic method properly.

Tip

When creating a graphic object for a control, be sure that it is created after the location and size of the control is fixed. The graphic object is associated with a particular region at the time it is created. If there is a change, the object may not recognize the change and may not perform properly on the control's new region.

Computing the Position for Each Light

Recall that there are 56 lights. For computation, you will identify each light by its position counting clockwise from the upper left corner starting with number 0. With this scheme, the first 14 lights will be drawn on the top from left to right; the next 14, on the right side, top down; the next 14, at the bottom from right to left; and finally, the last 14 on the left side, bottom up. You can then compute the position of a light based on its number. (There are simpler ways to draw the lights. The reason to draw the lights this way is explained later.)

For example, consider the first light bulb (position 0). Its upper left corner should be placed at upper left corner (0, 0) of the picture box. The second light bulb (position 1) should be *a full distance from the first one*; and the third will be another full distance from the second light. This means the x coordinate will increase by a full distance for each next light, whereas the y coordinate remains the same.

Let X = the x coordinate of the light bulb,
Y = the y coordinate of the light bulb, and
P = the position number of the light (0 = first position).

The positions of the *first 14 light bulbs* can then be computed as follows:

```
    X = P * Distance
    Y = 0
```

The next 14 light bulbs will go on the right side and down vertically. The x coordinate for these light bulbs will remain the same, whereas y will increase by a full distance for each subsequent light. Their positions can be computed as follows:

```
    X = 14 * Distance
```

```
    Y = (P - 14) * Distance
```
Notice that in computing Y, 14 is subtracted from P so that light number 14 (the first light for this side) is on the corner (at the top); and light number 15 is one position below light number 14.

For the bottom side, the lights will go from the right to the left. As the position value increases, the X value decreases. This first light for this side (number 28) should be 14 times the distance from the left, same as all lights on the right side.

Let *P* = position of a light on this side; then, the X value of the light position can be computed as follows:

```
    X = (42 - P) * Distance
    Y = 14 * Distance
```
In a similar fashion, the formula for the light positions on the left side can be developed.

Drawing the Light Bulb

After the position is computed, it is fairly easy to draw the light bulb using the FillEllipse method. Let TheColor be the color to fill in the ellipse; then, the statement should appear as follows:

```
TheGraph.FillEllipse(New SolidBrush(TheColor), New Rectangle(New
  Point(X, Y), BulbSize))
```
Where the point (X, Y) and BulbSize are as computed as above.

The DrawBulb Procedure

Now, you are ready to write the procedure that computes the position and draws a light bulb. You will make it a sub procedure named DrawBulb. This procedure requires two parameters: P, the position number (0 through 55) of the light bulb; and TheColor, the color for the light bulb. The complete procedure appears as follows:

```
Private Sub DrawBulb(ByVal P As Integer, ByVal TheColor As Color)
    Dim X As Integer
    Dim Y As Integer
    'Step 1: Compute the position (x, y) of the light bulb
    Select Case P
        Case 0 To 13
            ' on the top
            X = P * Distance
            Y = 0  'Y is fixed for the top row
        Case 14 To 27
            ' on the right margin
            X = 14 * Distance 'X is fixed on the right
            Y = (P - 14) * Distance
        Case 28 To 41
            ' at the bottom
            X = (42 - P) * Distance
            Y = 14 * Distance 'Y is fixed
        Case 42 To 55
            'on the left
            X = 0 'X is fixed
            Y = (56 - P) * Distance
    End Select
    ' Step 2: Draw the light bulb
TheGraph.FillEllipse(New SolidBrush(TheColor), New Rectangle(New
  Point(X, Y), BulbSize))
End Sub
```

Calling the Procedure

The preceding procedure can be called to draw all 56 light bulbs. It should be called when the user clicks the Draw Lights button. You would like to alternate the light colors among green, red, white, and blue. For simplicity, you will use four lines of code, using a different color for each call. The complete procedure appears as follows:

```
Private Sub btnShow_Click(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    Dim I As Integer
    ' This loop draws 56 light bulbs starting with position 0
    ' The colors are: green, red, white, and blue
    For I = 0 To 52 Step 4
        DrawBulb(I, Color.Green)
        DrawBulb(I + 1, Color.Red)
        DrawBulb(I + 2, Color.White)
        DrawBulb(I + 3, Color.Blue)
    Next I
End Sub
```

The procedure uses a For loop to call the DrawBulb Sub. Because each iteration calls DrawBulb four times using different color parameters, the loop counter is incremented by 4. Notice that the fourth call sets the bulb position to I + 3, so when I reaches 52, the last light bulb (number 55) will be drawn. The value of I should not increase beyond 52.

Why Compute the Bulb Positions That Way?

The DrawBulb procedure appears to be a bit complex. You may have figured out a simpler way to compute the positions for the light bulbs given the program requirements. So, you may wonder why such a complex procedure. There is a reason. The procedure can easily be modified to perform animation, which is discussed in the next section. You will revisit this project when one type of animation is illustrated.

The DrawString Method and Fonts

The DrawString method draws a string (text) with a specified font and a specified brush starting at a specified location. The method has the following syntax:

```
    Object.DrawString(string, font, brush, x, y)
```

Where x, y represent the coordinate for the starting point.

Creating a Font Object

The DrawString method requires a font object, which can be created with the following syntax:

```
New Font(FontName, FontSize, FontStyle)
```

For example, the following statement defines a font named Times New Roman with a size of 24, italic, and boldfaced:

```
MyFont = New Font("Times New Roman", 24, FontStyle.Italic Or
  FontStyle.Bold)
```

The ***FontFamilies object*** provides a list of fonts available in the system. The following routine enumerates the available fonts and populates them in a combo box:

```
    Dim TheFontFamily As FontFamily
    For Each TheFontFamily In Drawing.FontFamily.Families
        cboFont.Items.Add(TheFontFamily.Name)
    Next TheFontFamily
```

Measuring the String Length

Occasionally, you may have a need to compute the length of a string to properly draw it on an object. The ***MeasureString method*** returns a ***SizeF structure*** that gives the width and height of the string. The method has the following syntax:

```
    Object.MeasureString(String, Font)
```

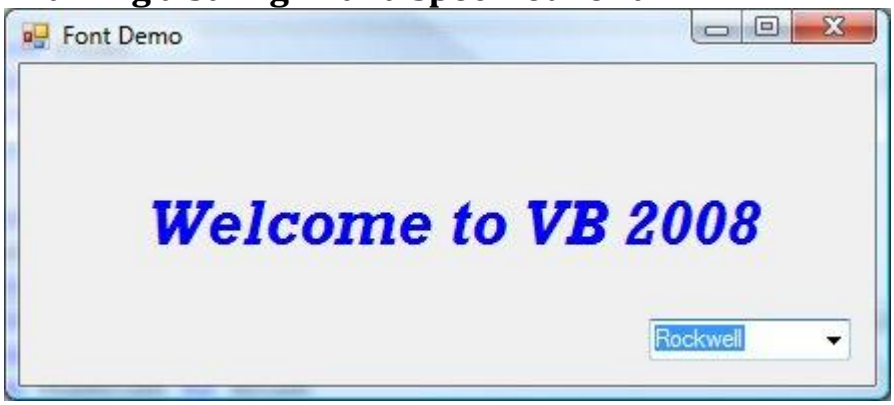
For example, the following expression using the MyFont object as defined previously will return a SizeF structure:

```
    e.Graphics.MeasureString("Welcome to VB 2008", MyFont)
```

A "Welcome" Example

To put together everything about drawing a string discussed above, let's consider a project to draw "Welcome to VB 2008" at the center of the form. In this project, all the fonts available in the system will be populated in a combo box named cboFont. When the user makes a font selection from the combo box, the text drawn on the form will immediately reflect the change. A sample result is shown in Figure B-6.

Figure B-6
Drawing a string with a specified font



The following form load procedure should enumerate the available fonts and populate them in the combo box:

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    Dim TheFontFamily As FontFamily
    For Each TheFontFamily In Drawing.FontFamily.Families
        cboFont.Items.Add(TheFontFamily.Name)
    Next TheFontFamily
    cboFont.SelectedIndex = cboFont.FindString(Me.Font.Name)
End Sub
```

The last statement in the procedure uses the FindString method to find the position of the form font in the combox's item list and assigns the position to the combo box's SelectedIndex property. This statement makes the font used for the form appears as the selection for the combo box.

The following Form Paint event procedure draws the Welcome message at the center of the form:

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
```

```
    Dim MyText As String
    Dim TheBrush As Brush
    Dim MyFont As Font
    Dim MySizeF As SizeF
    MyText = "Welcome to VB 2008"
    TheBrush = Brushes.Blue
    MyFont = New Font(cboFont.Text, 24, FontStyle.Italic Or
      FontStyle.Bold)
    With e.Graphics
        MySizeF = .MeasureString(MyText, MyFont)
        .DrawString(MyText, MyFont, TheBrush, (Me.Width -
    MySizeF.Width) / 2,
    (Me.ClientSize.Height - MySizeF.Height) / 2)
    End With
End Sub
```

As explained previously, the SizeF structure has the Width and Height elements. These elements are used in conjunction with the form's width and height to compute the starting position for the Welcome message. Note that the form's Height property includes its title bar. The area that does not include the title bar is the ClientSize property, which also has the Height and Width properties. Therefore, CleintSize.Height is actually used in the computation for more accurate vertical alignment. Also note that the font name specified in the font is taken from the combo box's Text property. The font is then used by the DrawString method to draw the string.

Tip

Also notice a fine point: The form height includes the title bar, whereas the Y value (for the Top property) starts with a zero at the client region not including the title bar. The height available for drawing does not go as high as the form's height. The area available for drawing is represented by the ClientSize property and its height is referenced as Me.ClientSize.Height.

Because you would like the string to reflect the change as soon as the user makes a font selection, you should force the form to repaint whenever there is a change in the selected index. The following event procedure should take care of this requirement.

```
Private Sub cboFont_SelectedIndexChanged(ByVal Sender As Object, ByVal
  e As System.EventArgs) Handles cboFont.SelectedIndexChanged
    ' Force the form to repaint
    Me.Invalidate()
End Sub
```

B.2. Animation

Animations take various forms. For example, you can show different pictures at a fixed time intervals in the same or different positions. While one picture is displayed, all other pictures of the same objects are made invisible. Usually, a timer is used to regulate the timing for displaying the pictures. This creates an illusion that the object is moving. You can also create animations by drawing instead of using existing pictures. For example, you can draw different colors on the same graph, again at a fixed time interval. This can create an impression that the object is blinking or the lights are moving. Also, you can create a rotating object (or an object with different motions) by drawing some part of the

object at a different relative position while the object moves. This section presents three examples to illustrate how each of these effects can be achieved.

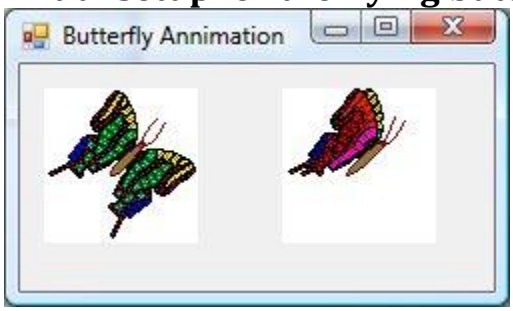
The Flying Butterfly

This example project shows a butterfly flying at a random height and distance (width) from the lower left corner toward the upper right corner of the form. (This example assumes that your computer has the illustrated images. If it does not, download similar images from the internet.) Follow these steps to set up the project:

1. Draw two picture boxes and a timer on a new form.
2. Name the picture boxes **picFly1** and **picFly2**; then set the Image properties to the two illustrated images for the two controls, respectively.
3. Name the timer control **tmrFly**. Set its Interval property to 2**00**. (You can change this setting depending on how fast you want the butterfly to flip and fly.)
4. Develop the code to perform the animation. The following discussion focuses on this aspect.

The initial form setup appears as in Figure B-7.

Figure B-7
Initial setup for the flying butterfly project



The Form Load Event Procedure

As soon the project starts, the program should perform several preparatory steps:

- Set the WindowState property to Maximized so that you can view the butterfly flying over the entire screen.
- Set the form's BackColor property to white so that it will correspond with that of the pictures.
- Move the pictures to the lower left corner so that they cannot be seen; that is, each picture is placed an entire width of itself to the left and entire height of itself below the form. Note that the Top property of the form starts with a value of zero below the title bar, but the form's Height property includes the title bar height. The area below the title bar is measured by the ClientSize property whose height is referenced as Me.ClientSize.Height.
- Randomize the seed for the Rnd function because this function will be used.

The following procedure should accomplish these requirements.

```
Private Sub Form1_Load(ByVal Sender As Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Me.WindowState = FormWindowState.Maximized  'maximize the form
```

```
    Me.BackColor = Color.White 'set background to white
    HideAtLowerLeft()
    Randomize() 'Randomize the seed
    tmrFly.Start()
End Sub
```

Note that the timer is started with its Start method, which sets the timer's Enabled property to True and thus starts the ticks. Notice that the routine to hide the images at the lower left corner of the form is written as a separate procedure named HideAtLowerLeft. The procedure appears as follows:

```
Private Sub HideAtLowerLeft()
    'Move picFly1 to lower left corner
    picFly1.Left = -picFly1.Width
    picFly1.Top = Me.ClientSize.Height
    'Move picFly2 to lower left corner
    picFly2.Left = -picFly2.Width
    picFly2.Top = Me.ClientSize.Height
End Sub
```

Toggling the Images

This animation involves only two pictures, so you can use a Static Boolean variable (IsFlyWingsOpen) to determine which picture should appear. The code structure can appear as follows:

```
    Static IsFlyWingsOpen As Boolean
    IsFlyWingsOpen = Not IsFlyWingsOpen 'Toggle wings open state
    If IsFlyWingsOpen Then
        'Compute the Wings-open picture position
    Else
        'Compute the wings-closed picture position
    End If
    picFly1.Visble = IsFlyWingsOpen
    picFly2.Visible = Not IsFlyWingsOpen
```

The first Not statement will toggle the value of IsFlyWingsOpen. The Visible property of the first picture is set to the value of IsFlyWingsOpen, so if the variable is True, that picture will appear; otherwise, it will disappear. On the other hand, the second picture's Visible property is set to the opposite (Not) value of IsFlyWingsOpen, so when the variable is True, the second picture will not be visible; otherwise, it will be visible. The last two statements will make one picture appear and the other disappear depending on the value of IsFlyWingsOpen, which is toggled (with the Not statement) between procedure calls.

Computing the Picture Position

The current picture will move on average half its own width and a quarter of its own height from the position of the previous picture. That is, picFly1's position (x, y) can be computed as follows:

```
    X = picFly2.Left + CInt(Rnd * picFly1.Width)
    Y = picFly2.Top - CInt(Rnd * 0.5 * picFly1.Height)
```

Notice that the position of picFly1 is based on the previous position of picFly2 because the two images take turns to appear. Notice also that Rnd * picFly1.Width will result in a horizontal move on average by half of the first picture's width because Rnd generates a random number between 0 and 1, whose expected (average) value is 0.5. Multiplying this

expected value by 0.5 should obtain an expected value of one-quarter, so the second formula should result in a vertical move up by one-quarter of the image's height. Similar formulas can be used to compute the new position for picFly2.

When the Butterfly Disappears from the Form

As soon as the butterfly disappears from the form, you can start the two images from the lower right corner again. The butterfly disappears when the Left property of either image is greater than or equal to the form's width or when the image's Top property plus its Height is less than zero. For code simplicity you will test only the first image. The code structure will appear as follows:

```
If picFly1.Left >  Me.Width OrElse  picFly1.Top + picFly1.Height < 0
  Then
'butterfly has completely disappeared, start at the lower left corner
'again
    HideAtLowerLeft()
End If
```

The test for the control's position should be done before any computation of the control's new position; therefore, this block of code should appear at the top of the procedure.

The Complete Timer Tick Procedure

The placement and appearance of the butterfly should be regulated by the timer's interval, so the code to perform the animation should be placed in the timer's Tick event. The complete timer Tick procedure appears as follows:

```
Private Sub tmrFly_Tick(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles tmrFly.Tick
    Static IsFlyWingsOpen As Boolean 'Declare a static variable
    If picFly1.Left > Me.Width OrElse picFly1.Top + picFly1.Height < 0
      Then
        'butterfly has completely disappeared, start at the lower left
        'corner again
        HideAtLowerLeft()
    End If
    IsFlyWingsOpen = Not IsFlyWingsOpen 'Toggle the state
    If IsFlyWingsOpen Then
        'Move the wings-open image
        picFly1.Left = picFly2.Left + CInt(Rnd() * picFly1.Width)
        picFly1.Top = picFly2.Top - CInt(Rnd() * 0.5 * picFly1.Height)
    Else
        ' Move the wings-closed image
        picFly2.Left = picFly1.Left + CInt(Rnd() * picFly2.Width)
        picFly2.Top = picFly1.Top - CInt(Rnd() * 0.5 * picFly2.Height)
    End If
    'Show wings-open picture if IsFlyWingsOpen is True
    picFly1.Visible = IsFlyWingsOpen
    'Make wings-closed image appear if IsFlyWingsOpen is False
    picFly2.Visible = Not IsFlyWingsOpen
End Sub
```

This project consists of two event procedures (the Form Load and the Timer Tick event procedures) and the HideAtCorner sub procedure. Run the project, and you should see the butterfly flies from the lower left corner toward the upper right corner until it disappears. It will then reappear from the lower left corner again.

Rotating Light Colors

By rotating the colors for the light bulbs, you can create an impression that the lights are moving. The previous Christmas lights project can easily be modified to show this effect. Follow these steps:

1. Copy the entire ChristmasLight folder, and name the new folder **ChristmasLgiht2**.
2. Draw a timer control on the form. Name it **tmrLights**. Set its Interval property to 250.
3. Add code in the Timer event. The following discussion focuses on this aspect.

Actually, the code in the Timer Tick event should be similar to the one in the previous btnShow Click event. Consider the For loop in that procedure:

```
    For I = 0 To 52 Step 4
        ' Statements to display green, red, white and blue lights
    Next I
```

The loop begins with drawing a green light at position 0 and rotates among the four colors as it proceeds. After the loop is complete, 56 lights are drawn. Suppose the next time the procedure is called, the loop will begin with drawing a green light at position 1 and proceed from there to draw 56 lights. The fifty-sixth light will be drawn at position 56, one position beyond the last (55). Notice, however, position 0 has not yet been drawn, so if you subtract 56 from the position number when it exceeds 55, you make the drawing starting from position 0 again. The loop can then be made to draw a complete rotation from whichever position it begins to draw the lights. That is, if you let BegPos be the position the loop starts, you can modify the loop parameters as follows:

```
    For I = BegPos To 52 + BegPos Step 4
        ' statements to draw the lights
    Next I
```

You can increase the value of BegPos by 1 each time the procedure is called. There are four colors. There is no need to have this variable take on more than four values (from 0 to 3), so you can code the following:

```
    BegPos = BegPos + 1
    If BegPos > 3 Then
        BegPos = 0
    End If
```

The complete Timer Tick event procedure can appear as follows:

```
Private Sub tmrLights_Tick(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles tmrLights.Tick
    Dim I As Integer
    Static BegPos As Integer
    'Draw 56 lights in rotation
    ' The colors are: green, red, white, and blue
    For I = BegPos To 52 + BegPos Step 4
        DrawBulb(I, Color.Green)
        DrawBulb(I + 1, Color.Red)
        DrawBulb(I + 2, Color.White)
        DrawBulb(I + 3, Color.Blue)
    Next I
    BegPos = BegPos + 1
    If BegPos > 3 Then
        BegPos = 0
    End If
End Sub
```

Notice that BegPos is declared as a Static variable so that its value can be preserved between the Timer Tick event procedure calls.
The event procedure does not really take care of the situation in which the light position exceeds 55. This can easily be handled in the DrawBulb procedure as follows:

```
Private Sub DrawBulb(ByVal P As Integer, ByVal TheColor As Color)
    Dim X As Integer
    Dim Y As Integer
    'Step 0: Adjust value for P
    If P > 55 Then
        P -= 56
    End If
    'Step 1: Compute the position of the light bulb
    ' Other statements (remain the same)
End Sub
```

You need to start the timer. One way, of course, is just to set its enabled property to True at design time. For the fun of it, you can place the code in the btnShow click event, replacing the code that you no longer need. You can make the button toggle between Show Lights and Stop. When the button's text shows Show Lights, a click will start the lights and the button will show the Stop text. Clicking on the button again will make the lights stop and the button show the Show Lights text. The code appears as follows:

```
Private Sub btnShow_Click(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    'Toggle between "Show Lights" and "Stop"
    If btnShow.Text = "Show Lights" Then
        tmrLights.Start()
        btnShow.Text = "Stop"
    Else
        tmrLights.Stop()
        btnShow.Text = "Show Lights"
    End If
End Sub
```

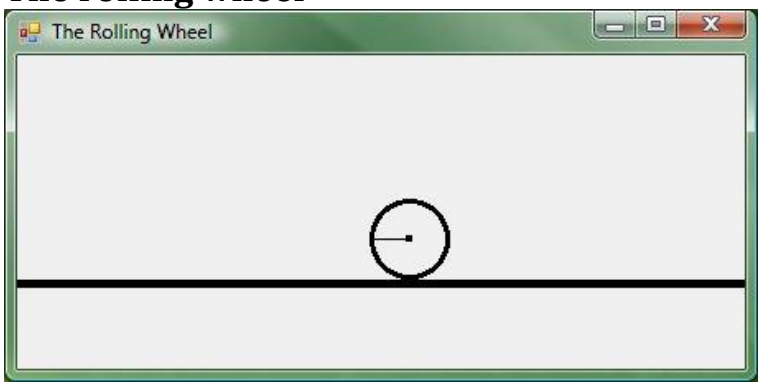
After you modify the project as shown, you are ready to see the effect. Run the program. Click the button, and you should be able to see lights rotating around the picture frame. Click it again, and you should see the lights stop.

The Rolling Wheel

This example project will emulate a rolling wheel on the form. The rolling wheel will appear from the right side and roll across the form. As soon as it disappears, it will appear from the right side again. To set up this project, bring a timer on a new form. Name the timer **tmrWheel** and set its interval to 100.

This project presents two issues. To "move" the wheel, you need not only to draw the wheel at the new position but also to erase the wheel at its previous location. Also, you need to show that the wheel not only moves forward but also rolls. To show the wheel is rolling, you can draw a line similar to the clock hand in the wheel and make it appear at different angles. A glimpse of the resulting form appears in Figure B-8.

Figure B-8
The rolling wheel



This project draws a wheel rolling across the form. As the wheel moves from right to left, its hand rotates, making the wheel appear to roll.

As the wheel "turns," the hand will rotate. The distance that the wheel travels and the angle that the hand rotates should be consistent.
Let *WheelRadius* equal the wheel radius, and *AngleTurned* equal the angle (in radians) that the hand rotates.

The distance the wheel travels can then be computed as follows:

```
Distance = WheelRadius * AngleTurned
```

Here is a roundabout way of explaining how the preceding formula is derived:
The length of the whole circle is 2 π WheelRadius. The ratio of the distance traveled to the circle for an angle, AngleTurned (in radians), is AngledTurned/2 π; therefore:

```
Distance = Circle × Ratio
         = (2 π WheelRadius) (AngleTurned / 2 π)
         = WheelRadius × AngleTurned
```

Assume that you will move the wheel 16 times to complete a full rotation (whole circle). The AngleTurned can be computed as follows:

```
AngleTurned = 2 π / 16; that is,
AngleTurned = π/8
```

There should also be a "road" below the wheel. To show the road, you will draw a box at 60% of the form's height. You can use the ***FillRectangle method*** to draw the box. The method has the following syntax:

```
Object.FillRectangle(Brush, Retangle)
```

where *Object* = the graphic object,
Brush = the brush (for example, SolidBrush) to use to fill the rectangle, and
Rectangle = a structure giving the location and size of the rectangle.

For your purpose, you can draw the road as soon the form is ready, so you can place the code in the form Paint event as follows:

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
    e.Graphics.FillRectangle(Brushes.Black, _
      New Rectangle(New Point(0, CInt(0.6 * Me.Height)), New
Size(Me.Width, 5)))
End Sub
```

The code specifies a black brush. The rectangle will start at the coordinate (0, 60% of form height) with a width of the entire form and height of five pixels.

Setting the Initial Position of the Wheel

The wheel will go from right to left on the form. Initially, it will not be seen, so the center of its X coordinate should be the form's width plus its radius. The center of its Y coordinate should be immediately above the road. The wheel itself should not be drawn on the road to avoid drawing over it. Let WheelX and WheelY be the coordinate for the wheel's center. They can then be computed as follows:

```
    ' Compute the intial coordinate for wheel center
    WheelX = Me.Width + WheelRadius 'On the form's right margin
    'Place the bottom of the wheel two pixels above the road
    WheelY = CInt(0.6 * Me.Height) - WheelRadius - 2
```

The formula for WheelY indicates that the lowest portion of the wheel will be drawn two pixels above the road. This should make the wheel barely touch the road surface since the wheel itself will be drawn with a pen size of three pixels.

The Form Load Procedure

All the preceding discussion pertains to the computations required as soon as the program starts. The code can be placed in the Form Load event as follows:

```
Dim TheGraph As Graphics
Dim Distance As Integer
Dim WheelRadius As Integer
Dim AngleTurned As Single 'angle turned in each period
Dim WheelX As Integer
Dim WheelY As Integer

Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    ' Compute parameter values
    WheelRadius = CInt(Me.Height / 10)
    AngleTurned = Math.PI / 8
    Distance = CInt(WheelRadius * AngleTurned)
    ' Compute the intial coordinate for wheel center
    WheelX = Me.Width + WheelRadius 'On the form's right margin
    'Place the bottom of the wheel two pixels above the road
    WheelY = CInt(0.6 * Me.Height) - WheelRadius - 2
    TheGraph = Me.CreateGraphics()
    tmrWheel.Start()
End Sub
```

Notice that all five variables have been declared as class-level variables because they will be used in other procedures. Notice also that a statement has been inserted to create the graph object and to start the timer. Finally, notice that the procedure uses the constant, PI of the Math object that gives the value of π.

Drawing the Wheel

The wheel has three parts: the circle, the center point, and the hand. Given the wheel center at (Xc,Yc) with a Radius and a Color, the circle for the wheel can be drawn on the form with the following code:

```
    'Draw the wheel
    TheGraph.DrawEllipse(New Pen(TheColor, 3), _
    New Rectangle(New Point(Xc - Radius, Yc - Radius), New Size(Radius
+ Radius, Radius + Radius)))
```

Recall that the rectangle to enclose the circle is defined with a point representing its upper left corner and a size of width by height. Both the width and height should be twice of the radius.
You would also like to draw a point at the center. This can be done with the following code:

```
    ' Draw the center point
    TheGraph.FillEllipse(New SolidBrush(TheColor), New Rectangle(New Point(Xc -
3, Yc - 3), New Size(5, 5)))
```

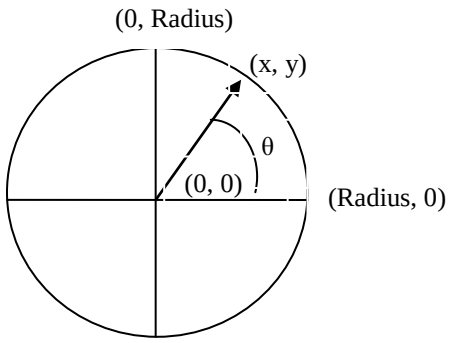
The center point is drawn with the FillEllipse method (instead of the DrawEllipse method) because you would like for it to show as a solid shape rather than a hollow circle.

The hand will connect the center point to the circle with a line. The hand's position on the circle depends on the angle. Given an angle, Theta (in radians), its coordinate, can be computed with the following formula:

```
    X = Xc + Cos(Theta) * Radius
    Y = Yc - Sin(Theta) * Radius
```

Recall that by definition, at the point of origin (0, 0), Sin(Theta) = Y / Radius. (Refer to Figure B-9 for a visual hint.) At that point, Y = Sin(Theta) × Radius. Notice, however, that the Y value computed in this formula is based on the Cartesian coordinate. The Y coordinate of an object in VB goes the other way: the closer a position is to the top, the smaller the value is. The Y value should be subtracted from (rather than added to) Yc. This is how the formula to compute Y is derived.

Figure B-9
(x, y) with an angle θ



Given this chart in Cartesian coordinate, the values of x and y can be computed as follows:
Sin(θ) = y / radius; thus, y = Sin(θ) x radius
θ = Cos(θ) x /radius; thus, x = Cos(θ) x radius

Similarly, at the point of origin, Cos(Theta) = X / Radius. Thus, X = Cos(Theta) × Radius. The X coordinate under the Cartesian system is consistent with that for the object in VB, so this value is added to Xc, the X coordinate of the wheel center. The hand then can be drawn with the DrawLine method as shown in following statement:

```
    'Draw the hand from the center
    TheGraph.DrawLine(New Pen(TheColor, 1), New Point(Xc, Yc), _
      New Point(CInt(Xc + Math.Cos(Theta) * Radius), CInt(Yc -
        Math.Sin(Theta) * Radius)))
```

The Sub procedure to draw a wheel (DrawWheel) appears as follows:

```
Sub DrawWheel(ByVal Xc As Integer, ByVal Yc As Integer, ByVal Radius As
  Integer, ByVal Theta As Single, ByVal TheColor As Color)
    'Draw the wheel
    TheGraph.DrawEllipse(New Pen(TheColor, 3), _
      New Rectangle(New Point(Xc - Radius, Yc - Radius), New
        Size(Radius + Radius, Radius + Radius)))
    ' Draw the center
    TheGraph.FillEllipse(New SolidBrush(TheColor), New Rectangle(New
      Point(Xc - 3, Yc - 3), New Size(5, 5)))
        'Draw the hand from the center
    TheGraph.DrawLine(New Pen(TheColor, 1), New Point(Xc, Yc),
      New Point(CInt(Xc + Math.Cos(Theta) * Radius), CInt(Yc -
        Math.Sin(Theta) * Radius)))
End Sub
```

Rolling the Wheel

The DrawWheel Sub procedure can be used to roll the wheel across the form. This should be done in the Timer Tick event procedure. In each event call, the original wheel should be erased; then another wheel should be drawn at a new location.

To erase the wheel, you can draw it at the same location with the BackColor. This will restore every part of the wheel to the BackColor, making it disappear. To draw another wheel at a new location, you will need to know the wheel's current coordinate. Recall that you have calculated the original coordinate for the wheel in the Form Load procedure. There, you also calculated the distance that the wheel would travel each time. The class-level variables for the coordinate are WheelX and WheelY, and the variable for the distance traveled is named Distance. WheelY should remain the same; however, WheelX should be decreased (because the wheel is going from right to left) by Distance for the distance the wheel travels; that is:

```
    WheelX = WheelX - Distance
```

When the wheel completely disappears from the form, it will reappear from the right. If the wheel has disappeared from the left side, the WheelX value should be less than the negative value of the radius. The code should appear:

```
    If WheelX < -WheelRadius Then
        ' wheel has disappeared; start from right again
        WheelX = Me.Width + WheelRadius
    End If
```

The wheel rolls counter-clockwise. This means that as the wheel rolls, the angle Theta is increased by the angle turned, which is represented by the variable AngleTurned and was also computed in the form load procedure; therefore,

```
    Theta = Theta + AngleTurned
```

You can start the value for Theta at zero and then check whether the value is greater than or equal to 2π. If so, the wheel has turned a complete round and Theta can be reset to 0. The code will appear as follows:

```
    If Theta >= 2 * Math.Pi Then
        ' Set Theta to 0
        Theta = 0
    End If
```

To put everything together, the complete Timer Tick event procedure should appear as follows:

```
Private Sub tmrWheel_Tick(ByVal Sender As Object, ByVal e As
System.EventArgs)
```

```
  Handles tmrWheel.Tick
    Static Theta As Single
    ' Erase wheel drawn previously
    DrawWheel(WheelX, WheelY, WheelRadius, Theta, Me.BackColor)
    If WheelX < -WheelRadius Then
        ' wheel has disappeared; start from right again
        WheelX = Me.Width + WheelRadius
    End If
    'Change the angle (turn)
    Theta = Theta + AngleTurned
    If Theta >= 2 * Math.PI Then
        ' Set theta to 0 to start over
        Theta = 0
    End If
    ' Move the wheel by a fixed distance
    WheelX = WheelX - Distance
    DrawWheel(WheelX, WheelY, WheelRadius, Theta, Me.ForeColor)
End Sub
```

Notice that Theta is declared to be a Static variable in the procedure so that its value between event calls can be preserved. Notice also the DrawWheel Sub is called twice. The first time, the form's BackColor is used as the color parameter so that the previous wheel can be erased. The second time, the new wheel position, the new angle, and the ForeColor are passed to draw the wheel.

This rolling wheel project includes the Form Paint event procedure (to draw the road), Form Load event procedure (to initialize various variables), the DrawWheel Sub procedure, and the tmrWheel Tick event procedure. Try the project and get a feel for how the wheel turns. You can change the timer interval and the denominator for the AngleTurned formula to get a different speed for the wheel.

B.3. Drag and Drop

Drag and drop provides an interesting and easy way for the user to specify operations that can be too complex to express by some other means. A typical drag-and-drop operation involves dragging an icon to another icon at another location to initiate certain activities. For example, a file can be dragged into a trashcan to delete the file. Typically, in this type of operation, three icons are involved. In the case of dumping a file, one icon will represent the file (the ***source***), another will represent an empty trashcan (the ***target***; before the file is dragged), and the other will represent the trashcan containing the dumped file (after the file is dropped). Only one of the two trashcans will be visible at a given time.

Before a drag and drop operation can be performed, *the target control's* ***AllowDrop property*** *must be set to True* (False is the default). Each drag and drop involves three steps:

1. Set up what is to be dragged. This is done in the source control using the DoDragDrop method, typically in the control's MouseDown event, which occurs when the user presses the mouse on the source control.
2. When the drag enters the target control, examine the data type and decide what type of drag-drop effect will be allowed. This is done in the target control's ***DragEnter event***.

3. Accept the data or perform the desired operation in the target control's ***DragDrop event***, which occurs when the user drops the object on the target control by releasing the mouse.

Tip

When testing your drag and drop code, if nothing seems to be active, check the AllowDrop property of your target control first. Make sure your code sets the property to True or it is set to True if it is accessible in the Properties window.

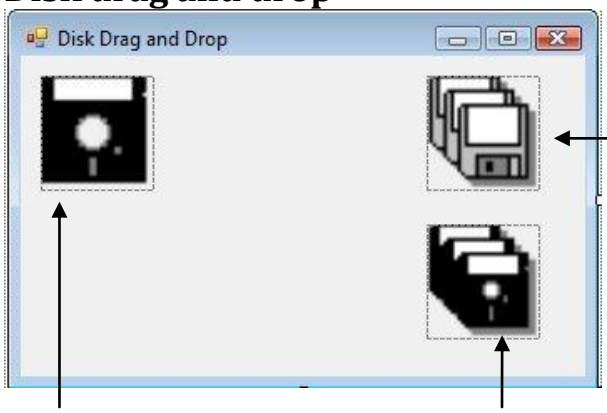
The following two examples illustrate some of the basics of the drag and drop operation.

Keeping a Disk in the Holder: An Example

This example project will allow the user to drag the disk and drop it in a disk holder. When the operation is complete, the disk will disappear as if it had been placed in the holder. The disk holder will change colors to indicate that it is holding the disk. Follow these steps to setup the project:

1. Draw three picture boxes on the new form. Set all the controls' *SizeMode property* to StretchImage so that the icon will fill the box.
2. Name the first control **picDisk**. Set its Image property to a disk icon. (See Figure B-10.) If you do not have one in your computer, search the internet for one.
3. Name the second picture box **picDiskHolder1**. Name the third one **picDiskHolder2** and set its Visible property to **False**.
4. Set picDiskHolder1's Image property to an icon that looks like an empty disk holder. Set picDiskHolder2's Image property to an icon that looks like a disk holder holding some disks. (See Figure B-10). If you do not have any similar images, search the internet for similar items.
5. Position the picture boxes so that they will appear in the form similar to those in Figure B-10.

Figure B-10
Disk drag and drop



Place picDiskHolder1 here. When the program starts, this holder appears. But after the disk is dragged and dropped here, the second holder will appear here in place.

Place picDisk here. Once this disk is dragged to the empty holder and dropped, it will disappear.

Place picDiskHolder2 here. When your program starts, this holder will not be visible.

You'd like for all three picture boxes to be of the same size. This can be done with code. Also, when the project starts, the disk on the left can be dragged to the disk holder on the right, so picDiskHolder1 is the target control and its AllowDrop property should be set to True. The Form Load event procedure appears as follows:

```
Private Sub Form1_Load(ByVal Sender As Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    picDiskHolder1.AllowDrop = True
    'Make all picture boxes of the same size
    picDiskHolder1.Size = picDisk.Size
    picDiskHolder2.Size = picDisk.Size
End Sub
```

As explained previously, when the drag operation starts, you need to indicate (with the DoDragDrop method) what types of drag-drop effects to have. The method has the following syntax:

```
    Object.DoDragDrop(Data, DragDropEffect)
```

Where *Data* = the data to drag, and
DragDropEffect = Copy, Move, Link, None, or All.

The code should be placed in the MouseDown event and appears as follows:

```
Private Sub picDisk_MouseDown(ByVal sender As Object, ByVal e As
  System.Windows.Forms.MouseEventArgs) Handles picDisk.MouseDown
    picDisk.DoDragDrop(picDisk, DragDropEffects.Move)
End Sub
```

The code indicates that the data to be worked with is the control itself and the drag-drop effect is "move".

The next step is to indicate what kind of effect to allow in the target control's DragEnter event. It should allow the move effect. The event procedure appears as follows:

```
Private Sub picDiskHolder1_DragEnter(ByVal sender As Object, ByVal e As
  System.Windows.Forms.DragEventArgs) Handles picDiskHolder1.DragEnter
    e.Effect = DragDropEffects.Move
End Sub
```

Finally, when the DragDrop event occurs in picDiskHolder1, picDisk should disappear. In addition, picDiskHolder2 should take the place of picDiskHolder1; that is, picDiskHolder2 should appear at the location of picDiskHolder1, which should then disappear. This description yields the following code:

```
Private Sub picDiskHolder1_DragDrop(ByVal sender As Object, ByVal e As
  System.Windows.Forms.DragEventArgs) Handles picDiskHolder1.DragDrop
    picDisk.Visible = False
    picDiskHolder2.Location = picDiskHolder1.Location
    picDiskHolder2.Visible = True
    picDiskHolder1.Visible = False
End Sub
```

You can now test the program. When you drag and drop the disk on the disk holder, the disk should disappear. In addition, the disk holder changes color (disk holder 1 is replaced by disk holder 2).

The drag-and-drop operation is not limited to images. It can also involve other kind of data. For example, the text in text boxes can be dragged and dropped into a list box. The next example shows how this can be done.

Dragging Texts Among Controls

This example project illustrates how to perform drag-and-drop operation on the text. You can set up the visual interface with the following steps:

1. Draw two text boxes and a list box on a new form.
2. Name the two text boxes **txtSender1** and **txtSender2**.
3. Name the list box **lstRecipient** and set its AllowDrop property to **True**. (Note that some controls do not have this property accessible at design time.)

As discussed previously, the first step is to indicate in the source control's MouseDown event the data to be dragged and the type(s) of allowed effect(s). You'd like to drag the text to the list box, so the data to operate on is the Text property of the text box and the type of effect is "Copy." You need to do this for both text boxes. The two event procedures appear as follows:

```
Private Sub txtSender1_MouseDown(ByVal sender As Object, ByVal e As
  System.Windows.Forms.MouseEventArgs) Handles txtSender1.MouseDown
    txtSender1.DoDragDrop(txtSender1.Text, DragDropEffects.Copy)
End Sub

Private Sub txtsender2_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles txtSender2.MouseDown
    txtSender2.DoDragDrop(txtSender2.Text, DragDropEffects.Copy)
End Sub
```

The DoDragDrop method in effect sends the text to the ***ClipBoard***. In subsequent events, the event argument passes the data in the Clipboard as the Data object.

The second step is to allow the drop effect when the drag operation enters the target control. The drop effect should be allowed only if the data format is as expected. The code appears as follows:

```
Private Sub lstRecipient DragEnter(ByVal sender As Object, ByVal e As
  System.Windows.Forms.DragEventArgs) Handles lstRecipient.DragEnter
    'Accept only text to drop
    If (e.Data.GetDataPresent(DataFormats.Text)) Then
        e.Effect = DragDropEffects.Copy
    Else
        e.Effect = DragDropEffects.None
    End If
End Sub
```

Notice how the If statement is coded. The event argument passes the data dragged (Data) and the data's GetDataPresent method tests whether the data dragged (in the Clipboard) is of the data format specified in the parameter, which is specified to be DataFormats.Text. (Other possible data formats include Bitmap, Html, UnicodeText, WaveAudio, and Tiff.) If the method returns a value of True, the copy effect of the drop operation will be allowed; otherwise, there will be no effect (none).

The third step is to handle the data when the Drag-Drop event occurs. The code appears as follows:

```
Private Sub lstRecipient_DragDrop(ByVal sender As Object, ByVal e As
  System.Windows.Forms.DragEventArgs) Handles lstRecipient.DragDrop
    lstRecipient.Items.Add(e.Data.GetData(DataFormats.Text))
End Sub
```

Notice what is added to the list box's Items list. The data (text) is not obtained directly from either of the text boxes but from the event's Data argument, using the Data object's GetData method with a specified format of Text. Recall that the text was placed into the

Clipboard in the first step of the drag-drop operation. With this approach, there is no need to identify the control that provides the source data.

Summary

- Various draw methods draw the outline of a shape. For example, the DrawRectangle method draws a rectangle.
- Various fill methods paint the entire shape. For example, the FillReactangle method fills the rectangle with a color.
- To draw a graph on a control, such as label, picture box, and form, there must be a graphic object associated with the control. The graphic object is one of the event arguments of the control's paint event. If you want draw something on the control when it first appears, you can perform the drawing in the paint event and use that graphic object. You can also use the control's CreateGraphics method to create graphic object.
- The freehand drawing example shows some of the basics about drawing using the DrawLine method.
- The DrawEllipse and FillEllipse methods draw ellipses and circles. The Christmas lights example illustrates the use of the FillEllipse method. It also suggests the potential computational complexity involved in drawing.
- Animation can be done using existing images or by drawing graphics. Typically, some code is placed in a Timer Tick event to move or rotate the images or draw shapes across the control surface to cause the animation effect.
- Three examples (flying butterfly, rotating Christmas lights, and rolling wheel) were used to illustrate how animation can be accomplished.
- Drag and drop operations are carried out in three steps. The first step involves coding in the source control's MouseDown event to place the data in the Clipboard based on the specified type of effect the drag is to have. The second step involves coding in the target control's DragEnter event to determine what type of drop effect is allowed. The third step involves coding in the target control's DragDrop event to take the data and put it in the proper place.
- Two examples (keeping a disk in the holder and dragging the text into a list box) were used to illustrate the basics of the drag and drop operation.

Explore and Discover

B-1. The Pens Class. Enter the following code in a new form:

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim ThePen As Pen
    ThePen = Pens.Beige
End Sub
```

When you typed the period (.)after Pens, what did you notice? The IntelliSense provides you with a long list of standard colors, which are the same as those in the Brushes list. You can use a pen from the Pens list without using the Pen constructor.

B-2. The Graphics Clear Method. Draw a button on the form. Name the button **btnClear** and enter the following code:

```
Private Sub btnClear_Click(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles btnClear.Click
    Dim TheGraph As Graphics
    TheGraph = Me.CreateGraphics
    TheGraph.Clear(Color.Red)
End Sub
```

Run the project and then click the form. What do you see? The Clear method paints the entire region of a control with the specified color. To actually clear the surface of the control, use the control's BackColor property.

B-3. Additional Font Styles. Draw a button on the form. Name it **btnWrite** and enter the following code:

```
Private Sub btnWrite_Click(ByVal Sender As Object, ByVal e As
  System.EventArgs) Handles btnWrite.Click
    Dim TheGraph As Graphics
    Dim TheFont As Font
    TheGraph = Me.CreateGraphics
    TheGraph.Clear(Me.BackColor)
    TheFont = New Font("Times New Roman", 16, FontStyle.Strikeout Or
      FontStyle.Underline)
    TheGraph.DrawString("Done!", TheFont, Drawing.Brushes.Black, 10,
10)
End Sub
```

Run the project and then click the button. What do you see on the form? Strikeout and Underline are the two additional font styles that you can use with a font.

B-4. The Linear Gradient Brush. Draw a panel on a new form. Place the following line above the Class statement in the code window.

```
Imports System.Drawing.Drawing2D
```

Enter the following code:

```
Private Sub Panel1_Paint(ByVal sender As Object, ByVal e As
  System.Windows.Forms.PaintEventArgs) Handles Panel1.Paint
    Dim TheGraph As Graphics
    Dim LGBrush As LinearGradientBrush
    Dim Rect As Rectangle
    TheGraph = e.Graphics()

    Rect = New Rectangle(100, 10, 40, 80)
    LGBrush = New LinearGradientBrush(Rect, Color.Blue, Color.White,
      LinearGradientMode.Vertical)
    LGBrush.SetSigmaBellShape(0.5)
    TheGraph.FillEllipse(LGBrush, Rect)
End Sub
```

Run the project. What do you see in the panel? The LinearGradient brush can be used to create various effects with different modes. For additional information, use LinearGradientBrush class as the keyword in the index tab of the help menu to find the relevant pages.

Exercises

B-5. Draw Diagonal Cross. Draw a picture box on a new form. Write the code so that as soon as the program starts, the picture box will appear with a blue diagonal cross with a width of five pixels.

B-6. Draw a Circle. Draw a picture box on the form. Write the code to draw a circle that touches all sides of the picture box. The circle should be drawn with a green pen of thee pixels width and should appear as soon as the program starts. (*Hint:* Make sure the picture box's width and height have the same value.)

B-7. Revised Free Hand Drawing. As suggested in this appendix, a more intuitive way for free hand drawing is to draw the line starting at the point where the right mouse is pressed and ending at the point where the mouse is let go (the MouseUp event). Develop a project that will implement this method.

B-8. Rotating Christmas Lights. Modify the first Christmas lights project in this appendix so that every fourth light will go out in rotation every quarter of a second. That is, the first quarter of a second, lights 0, 4, 8, and so on will go out; the second quarter of a second, lights 1, 5, 9, and so on will go out, whereas lights 0, 4, 8, and so on will come back on. (*Hint:* Fill the light bulb with the black color to make it "go out." Fill the light bulb with its original color to make it "come back on." Note that all lights to be restored are the same color.)

B-9. Add Color to the Rolling Wheel. Modify the rolling wheel project in this appendix so that the wheel's color is white. Make sure the previous wheel is properly erased as the new one moves forward.

B-10. Emulating the Clock. Draw a clock with a second hand. The hand will tick every second and complete a full rotation on the clock every minute.

B-11. Take the Sad Face to the Happy Home. Draw two picture boxes and a group box on a new form. Name the two picture boxes **picSad** and **picHappy**. Set the Image properties of the two controls to some sad face and happy face images. Set the group box's Text property to **Happy Home** and name it **grpHappyHome**.

Provide the code so that at run time the sad face will appear but the happy face is not visible. When it is dragged into the Happy Home group box, it turns into the happy face.

B-12. Drag and Drop Between Two List Boxes. Draw two list boxes onto a form. Populate both boxes with your friends' names. Provide the code so that at run time you can drag any name from one control and drop it into the other control.

B-13. Print Text on the Form. Write a sub procedure that will take as its parameters a control, a string, and a pair of numbers (representing the location) and will print the string on the control at the specified location. The brush should have a black color. The font

should be Times New Roman with a size of 16 (regular font style). Test the sub by writing “Hello” on the form.

B-14. Printing the Title. Modify the sub in the preceding exercise (B-12) so that it will print the text at the center near the top of the “page.” Note that the sub now needs only one number (y) (instead of a pair) in the parameter list. It should compute the value of x so that the title is printed at the center horizontally. Test the sub by writing **Annual Report** on a picture box.

B-15. Fancy Christmas Lights. (Refer to Exercise B-4.) Now you know how to draw fancier light bulbs. Modify the first Christmas lights example so that the light bulbs look really fancy.