Chapter 14

Drawing and Painting with Visual Basic

One of the most interesting and fun parts of a programming language is its graphics elements. In general, graphics fall into two major categories: vector and bitmap. Vector graphics are images generated by graphics methods such as the DrawLine and DrawEllipse methods. The drawing you create is based on mathematical descriptions of the various shapes. Bitmap graphics are images that can be displayed on various controls and processed on a pixel-by-pixel basis. The difference between vector and bitmap graphics is that vector graphics aren’t tied to a specific monitor resolution; that is, they can be displayed at various resolutions.

Vector graphics can be redrawn at any resolution. Bitmap graphics, on the other hand, have a fixed resolution. An image 600 pixels wide and 400 pixels tall has that specific resolution. If you attempt to use it to fill a monitor that’s 1280 pixels wide and 1024 pixels tall, you’ll have to repeat some pixels. Image-processing software can interpolate between pixels, but when you blow up a bitmap, you see its block-like structure.

Despite their inherent limitations, bitmap graphics are quite useful and much more common than vector graphics. For example, you can’t create the image of a landscape with graphics commands. On the other hand, it doesn’t make sense to display the bitmap of a circle when a simple Circle command can produce the same image faster and more cleanly. Both types of graphics have their place, and you can mix them to produce the desired result.

Text belongs to the vector graphics category, because the characters in various fonts are described mathematically and can be rendered at various sizes with no loss of quality. Figure 14.1 shows a string printed at 96 points (at the top) and the same string printed at 48 points and enlarged 200%. The upper string is as smooth as it can be, while the lower one has too many artifacts around its edges. I could have made the upper string even smoother by turning on the anti-aliasing feature, but then the comparison wouldn’t be fair. The .NET Framework provides rich tools for rendering text, and we’ll examine them along with the other vector drawing methods. If the differences between the two strings aren’t obvious on the printed page, you can open the file VectorBitmap.tif in this chapter’s folder on the CD, which is the electronic version of the same image.
With VB.NET you can draw on just about any control. However, it's quite unusual to draw on a TextBox control and highly unlikely that you will ever draw on a ListBox control. The two objects we usually draw on are the Form object and the PictureBox control. You can place graphics on controls at design time and runtime. To load a graphic (bitmap or icon) on a control at design time, you assign its filename to the Image, or BackgroundImage, property of the control in the Properties window. Or, you can change the setting of the same two properties at runtime.

If the graphic is assigned to a control at design time, it's stored along with the application. Vector drawings aren't loaded; they are generated on-the-fly. Where bitmap graphics are copies of their subjects (pictures of buildings, persons, landscapes, and so on), vector graphics are the descriptions of the objects we want to display (a circle centered at a given point having a certain radius, or a rectangle of certain width and height filled with a specific gradient) and are rendered at runtime.

**Displaying Images**

To load an image to a PictureBox control, locate the Image property in the Properties window and click the button with the ellipsis next to it. An Open dialog box will appear, where you can select the image to be displayed. The image is stored in a hidden file that has the same name as the form plus the extension .resx. As a result, you don’t have to distribute the image with your application.

After the image is loaded, you must make sure it will fill the available space, unless you let the user select the graphic at runtime. The PictureBox control exposes the SizeMode property, which determines how the image will be sized and aligned on the control. Its default setting is Normal, and in this mode the control displays the image at its normal magnification. If the image is larger than the control, part of the image will be invisible. If the image is smaller than the control, part of the control will be empty. If the image is smaller than the control, you can set the SizeMode property to CenterImage to center the image on the control.

The SizeMode property can also be set to StretchImage and AutoSize. The StretchImage setting resizes the image so that it fills the control. If the control’s aspect ratio isn’t the same as the aspect ratio of the image, the image will be distorted in the process. If you want to use the StretchImage setting, you must also resize one of the dimensions of the control, so that the image will be properly resized. You’ll see how to do this in the following sample. The last setting, AutoSize, resizes the control to the image. This is not the most convenient setting, because the control may cover other controls on the form. Figure 14.2 shows a PictureBox control with a small image in all four settings.
The most flexible setting of the SizeMode property is StretchImage. Before letting the Form Designer stretch the image, however, you must make sure that the control has the same aspect ratio as the image it displays. If the image is twice as wide as it is tall, the same should be true for the PictureBox control that hosts the image. If that’s the case, the image can be resized safely. If not, the image will be distorted in the process.

Loading an image to a PictureBox control doesn’t require any code or special handling. For more information on resizing images while maintaining their aspect ratio, see the discussion of the ImageLoad project, later in this chapter. The image itself is an object, and you can also manipulate it from within your code. The following section describes the Image object, its properties, and a few of the methods you can use to manipulate an image.

The Image Object

The Image property of the PictureBox control is an Image object, which contains the current bitmap and exposes properties and methods for manipulating this image. There are several ways to create an Image object. You can declare a variable of the Image type and then assign the Image property of the PictureBox control of the Form object to the variable:

```vbnet
Dim img As Image
img = PictureBox1.Image
```

The `img` Image variable holds the bitmap of the PictureBox1 control. As you will see shortly, you can call the Save method of the Image class to save the image to a disk file.

You can also create a new Image object from an image file, using the Image object's FromFile method:

```vbnet
Dim img As Image
img = Image.FromFile("Butterfly.jpg")
```
Once the `img` variable has been set up, you can assign it to the Image property of a PictureBox control:

```vbnet
PictureBox1.Image = img
```

**Properties**
The Image object exposes more members, some of which are discussed in the following sections. Let’s start with the properties, which are simpler.

*HorizontalResolution, VerticalResolution*
These are read-only properties that return the horizontal and vertical resolution of the image, respectively, in pixels-per-inch.

*Width, Height*
These are read-only properties that return the width and height of the image, respectively, in pixels. If you divide the dimensions of the image (properties `Width` and `Height`) by the corresponding resolutions (properties `HorizontalResolution` and `VerticalResolution`), you’ll get the actual size of the image—the dimensions of the image when printed, for instance.

*PixelFormat*
This is another read-only property that returns the pixel format for this Image object. The `PixelFormat` property determines the quality of the image; there are many pixel formats, which are members of the `PixelFormat` enumeration. For now, I will assume that you’re using a color display with a depth of 24 bits per pixel. Images with 24-bit color are of the `PixelFormat24bppRgb` type. `Rgb` stands for “red green blue” (the three basic colors) and `24bpp` stands for 24 bits per pixel. Each of the basic colors in this format is represented by one byte (8 bits).

**Methods**
In addition to the basic properties, the Image object exposes methods for manipulating images. These are discussed next.

*RotateFlip*
This method rotates and/or flips an image, and its syntax is:

```vbnet
Image.RotateFlip(type)
```

where the `type` argument determines how the image will be rotated. This argument can have one of the values of the `RotateFlipType` enumeration, shown in Table 14.1.

To flip vertically the image displayed on a PictureBox control, use the following statement:

```vbnet
PictureBox1.Image.RotateFlip(RotateFlipType.RotateNoneFlipY)
```

The Refresh method redraws the control, and you must call it to display the new (flipped) image on the control.
### Table 14.1: The RotateFlipType Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate180FlipNone</td>
<td>Rotates image by 180 degrees</td>
</tr>
<tr>
<td>Rotate180FlipX</td>
<td>Rotates image by 180 degrees and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate180FlipXY</td>
<td>Rotates image by 180 degrees and then flips it vertically and horizontally</td>
</tr>
<tr>
<td>Rotate180FlipY</td>
<td>Rotates image by 180 degrees and then flips it vertically</td>
</tr>
<tr>
<td>Rotate270FlipNone</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by –90 degrees)</td>
</tr>
<tr>
<td>Rotate270FlipX</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by –90 degrees) and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate270FlipXY</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by –90 degrees) and then flips it vertically and horizontally</td>
</tr>
<tr>
<td>Rotate270FlipY</td>
<td>Rotates image by 270 degrees (which is equivalent to rotating it by –90 degrees) and then flips it vertically</td>
</tr>
<tr>
<td>Rotate90FlipNone</td>
<td>Rotates image by 90 degrees</td>
</tr>
<tr>
<td>Rotate90FlipX</td>
<td>Rotates image by 90 degrees and then flips it horizontally</td>
</tr>
<tr>
<td>Rotate90FlipXY</td>
<td>Rotates image by 90 degrees and then flips it horizontally and vertically</td>
</tr>
<tr>
<td>Rotate90FlipY</td>
<td>Rotates image by 90 degrees and then flips it vertically</td>
</tr>
<tr>
<td>RotateNoneFlipNone</td>
<td>No rotation and no flipping</td>
</tr>
<tr>
<td>RotateNoneFlipX</td>
<td>Flips image horizontally</td>
</tr>
<tr>
<td>RotateNoneFlipXY</td>
<td>Flips image vertically and horizontally</td>
</tr>
<tr>
<td>RotateNoneFlipY</td>
<td>Flips image vertically</td>
</tr>
</tbody>
</table>

### GetThumbnailImage

This method returns the thumbnail of the specified image. The thumbnail is a miniature version of the image, whose exact dimensions you can specify as arguments. Thumbnail images are used as visual enhancements in selecting an image. The thumbnail takes a small fraction of the space taken by the actual image, and we can display many thumbnails on a form to let the user select the desired one(s). The syntax of the GetThumbnailImage method is:

```csharp
Image.GetThumbnailImage(width, height, Abort, Data)
```

The first two arguments are the dimensions of the thumbnail. The other two arguments are callbacks, which are used when the process is aborted. Since thumbnails don’t take long to generate, we’ll ignore these two arguments for the purposes of this book (we’ll set them both to Nothing). The following statements create a thumbnail of the image selected by the user and display it on a
PictureBox control. To test these statements, place a PictureBox and a Button control on the form. Then place an instance of the Open dialog box on the form and insert the following statements in the Button’s Click event handler:

```vbnet
Dim img As Image
    img = ImageFromFile(OpenFileDialog1.FileName)
    PictureBox1.Image = img.GetThumbnailImage(32, 32, Nothing, Nothing)
```

Using the techniques described in Chapter 13, you can scan a folder, retrieve all the image files, and create a thumbnail for each. As for displaying them, I would suggest you create as many PictureBox controls as there are images in the folder and arrange them horizontally and vertically on a form. Chapter 5 describes how to create instances of Windows controls at runtime and position them on the form from within your code. Since this isn’t a trivial project, I’ve included a sample project on the CD that demonstrates how to display thumbnails on a form. The project is called Thumbnails, and you will find it in this chapter’s folder. I’ve copied the CustomExplorer project of Chapter 11, renamed the Form, and removed the FilesList control (where the names of the files in the selected folder were displayed). In its place, the program displays the PictureBox controls with the thumbnails. When the user clicks an image, the program loads the image on the PictureBox control of another form and displays it. Figure 14.3 shows the two forms of the Thumbnails application. You can see the thumbnails of the images on one of the forms and one image in preview mode on the other form.

**Figure 14.3**
The Thumbnails application displays the images in a folder as thumbnails.

Then I adjusted the code to accommodate the display of thumbnails instead of file names. The ShowFilesInFolder() subroutine of the original application displayed the names of the files in the current folder on a ListBox control. This subroutine was replaced by the ShowImagesInFolder() subroutine, which is shown in Listing 14.1.
LISTING 14.1: THE SHOWIMAGESINFOLDER SUBROUTINE

Sub ShowFilesInFolder()
    Dim file As String
    Dim FI As FileInfo
    Dim PBox As PictureBox, img As Image
    Dim Left As Integer = 280
    Dim Top As Integer = 40
    Dim ctrl As Integer
    ' remove all PictureBox controls on the form
    For ctrl = Me.Controls.Count - 1 To 2 Step -1
        Me.Controls.RemoveAt(Me.Controls(ctrl))
    Next
    Me.Invalidate()
    For Each file In Directory.GetFiles(Directory.GetCurrentDirectory())
        FI = New FileInfo(file)
        If FI.Extension = ".GIF" Or FI.Extension = ".JPG" Or _
            FI.Extension = ".BMP" Then
            PBox = New PictureBox()
            img = Image.FromFile(FI.FullName)
            PBox.Image = img.GetThumbnailImage(64, 64, Nothing, Nothing)
            If Left > 560 Then
                Left = 280
                Top = Top + 74
            End If
            PBox.Left = Left
            PBox.Top = Top
            PBox.Width = 64
            PBox.Height = 64
            PBoxVisible = True
            PBox.Tag = FI.FullName
            Me.Controls.Add(PBox)
            AddHandler PBox.Click, New System.EventHandler(AddressOf OpenImage)
            Left = Left + 74
        End If
    Next
End Sub

The subroutine starts by removing any PictureBox control already on the form. This is necessary because when the user switches to another folder, we want to display this folder’s images on a clean form. Then the code goes through each file in the selected folder and examines its extension. If it’s JPG, GIF, or BMP (you can add more file extensions if you want), it creates a new PictureBox control, sets its size and location, loads the thumbnail of the image, and then adds it to the Controls collection of the form. Each image’s path is stored in the PictureBox control’s Tag property, and it’s retrieved later to load the image on the second form, where it can be previewed.
Notice how the code adds a handler for the Click event of each PictureBox control. All the PictureBox controls share a common handler for their Click event, the OpenImage() subroutine. This subroutine reads the selected image’s path from the Tag property of the control that fired the Click event and displays the corresponding image on the auxiliary form. The implementation of the OpenImage() subroutine is shown here:

Sub OpenImage(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim imgForm As New previewForm()
    imgForm.PictureBox1.Image = Image.FromFile(sender.tag)
    imgForm.Show()
End Sub

previewForm is the name of the second form of the application, where the selected image is previewed. If you need more information about this project, please review the material of the last part of Chapter 5, which explains how to create instances of controls at runtime. This application is a rather advanced example of dynamic forms, rather than a demo of the GetThumbnailImage method, but it’s an interesting application and some readers may have a good use for the techniques demonstrated here. You will notice that all the bitmaps have the same dimensions (64 by 64), which means that the thumbnails will be distorted (most images aren’t square). You must choose the dimensions of the thumbnail for an image, so that the reduced image has the same aspect ratio as the original image. For example, if the original image’s dimensions are 640×480, the thumbnail’s dimensions should be 64×48 (or 32×24, or 128×96, and so on). In the later section on the ImageLoad project, you will learn how to resize an image and maintain its aspect ratio.

The user interface of the Thumbnails application isn’t the most functional either. If you scroll the form to see all the thumbnails that aren’t near the top of the form, the controls with the drives and folder names will be scrolled out of view. Use a different form to display the thumbnails, or add the appropriate menu commands, which can’t be scrolled out of view.

Save

If your application processes the displayed image during the course of its execution and you want to save the image, you can use the Save method of the Image property. The simplest syntax of the Save method accepts a single argument, which is the path of the file where the image will be saved:

Image.Save(path)

To save the contents of the PictureBox1 control to a file, you must use a statement like the following:

PictureBox1.Image.Save("c:\tmpImage.bmp")

The image will be saved in BMP format. Another form of the Save method allows you to specify the format in which the image will be saved:

PictureBox1.Image.Save("c:\tmpImage.bmp", format)

where the format argument’s value can be one of the members of the ImageFormat enumeration. The fully qualified name of the enumeration is System.Drawing.Imaging.ImageFormat, so you should import the library System.Drawing.Imaging into any project that uses the enumerations mentioned in this chapter. This way you won’t have to fully qualify the name of the enumeration.
The ImageFormat enumeration contains members for all common image formats (see Table 14.2). Once you’ve imported the System.Drawing.Imaging class to your project, then to save the image on the PictureBox1 control in GIF format, use the statement:

`PictureBox1.Image.Save("c:\tmpImage.gif", ImageFormat.Gif)`

<table>
<thead>
<tr>
<th>MEMBER</th>
<th>DESCRIPTION</th>
<th>EXTENSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bmp</td>
<td>Bitmap image</td>
<td>BMP</td>
</tr>
<tr>
<td>Emf</td>
<td>Enhanced Windows metafile</td>
<td>EMF</td>
</tr>
<tr>
<td>Exif</td>
<td>Exchangeable Image Format</td>
<td>EXIF</td>
</tr>
<tr>
<td>Gif</td>
<td>Graphics Interchange Format</td>
<td>GIF</td>
</tr>
<tr>
<td>Icon</td>
<td>Windows icon</td>
<td>ICO</td>
</tr>
<tr>
<td>Jpeg</td>
<td>Joint Photographic Experts Group format</td>
<td>JPEG</td>
</tr>
<tr>
<td>MemoryBmp</td>
<td>Saves the image to a memory bitmap</td>
<td></td>
</tr>
<tr>
<td>Png</td>
<td>W3C Portable Network Graphics format</td>
<td>PNG</td>
</tr>
<tr>
<td>Tiff</td>
<td>Tagged Image File Format</td>
<td>TIF</td>
</tr>
<tr>
<td>Wmf</td>
<td>Windows metafile</td>
<td>WMF</td>
</tr>
</tbody>
</table>

**VB.NET AT WORK: THE IMAGELOAD PROJECT**

The ImageLoad application (shown in Figure 14.4) demonstrates how to use the SizeMode property to best fit an image on a PictureBox control. The PictureBox control maintains a constant size, and you won’t have to do anything special about the other controls on the form. If the image fits on the control, the control’s SizeMode property is set to CenterImage—the image is displayed centered on the control. If the image’s dimensions exceed the dimensions of the PictureBox control, the code resizes the larger dimension of the control, according to the image’s aspect ratio. The PictureBox control’s size isn’t drastically different from its initial dimensions, and it never grows to cover other controls.

**FIGURE 14.4**

The ImageLoad project
The application prompts the user to select an image file through the Open dialog box. Then it compares the image’s dimensions to the dimensions of the control, and if the image is smaller than the control in both dimensions, it sets the control’s SizeMode property to CenterImage. If not, it calculates the image’s aspect ratio and resizes the larger dimension accordingly.

The Load command under the Image menu is implemented with the statements in Listing 14.2.

**Listing 14.2: The Image ➤ Load Menu Command**

```vbnet
Private Sub ImageLoad_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ImageLoad.Click
    OpenFileDialog1.Filter = "Images|*.GIF;*.JPG;*.TIF;*.BMP"
    If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
        PictureBox1.Image = Image.FromFile(OpenFileDialog1.FileName)
        ResizeImage()
    End If
End Sub
```

The code displays the image on the control and immediately calls the ResizeImage subroutine to resize the image. This subroutine will display the image on a PictureBox control whose dimensions are 400 by 400 pixels. If one of the image’s dimensions exceeds the corresponding dimension of the control, this dimension will be resized while maintaining the aspect ratio. Listing 14.3 shows the ResizeImage() subroutine used in the LoadImage application.

**Listing 14.3: The ResizeImage() Subroutine**

```vbnet
Private Sub ResizeImage()
    PictureBox1.Width = 400
    PictureBox1.Height = 400
    If PictureBox1.Image.Width < PictureBox1.Width And _
       PictureBox1.Image.Height < PictureBox1.Height Then
        PictureBox1.SizeMode = PictureBoxSizeMode.CenterImage
    Else
        Dim ratio As Single
        If PictureBox1.Image.Width > PictureBox1.Image.Height Then
            ratio = PictureBox1.Image.Width / PictureBox1.Image.Height
            PictureBox1.Height = PictureBox1.Width / ratio
        Else
            ratio = PictureBox1.Image.Height / PictureBox1.Image.Width
            PictureBox1.Width = PictureBox1.Height / ratio
        End If
    End If
End Sub
```
The user can restore the image to its original size with the Zoom ➔ Normal command, whose code is almost trivial:

```vbnet
Private Sub ZoomNormal_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ZoomNormal.Click
    PictureBox1.Width = PictureBox1.Image.Width
    PictureBox1.Height = PictureBox1.Image.Height
End Sub
```

The Auto command of the Zoom menu fits the image on the control by calling the ResizeImage() subroutine. The last two commands in the menu, Zoom In and Zoom Out, enlarge or reduce the magnification of the image by 25%: their implementation is also trivial. The following statements zoom into the image by 25%:

```vbnet
PictureBox1.Width = PictureBox1.Width * 1.25
PictureBox1.Height = PictureBox1.Height * 1.25
```

As you can see, the various zooming commands don’t directly manipulate the Image object. Instead, they control the dimensions of the image’s container (the PictureBox control) and rely on the AutoSize setting of the control’s SizeMode property to resize the image. The code would have been even simpler if we didn’t want to maintain the image’s aspect ratio. We’re going to use the same subroutine in the PrintBitmap project of the following chapter, where you will learn how to print bitmaps at any magnification.

**NOTE** The PictureBox control’s Image object doesn’t change when you resize the control. Its dimensions are the dimensions of the image loaded initially, regardless of the current magnification.

The Process menu of the application contains commands for rotating and flipping the image. These commands call the RotateFlip method with different arguments. The only implication worth mentioning here is that when we rotate an image right or left, we’re actually swapping its width with its height. To avoid clipping images that are not square, you must swap the dimensions of the PictureBox control as well. Listing 14.4 shows the code of the Rotate Right command:

**Listing 14.4: Rotating an Image**

```vbnet
Private Sub ProcessRotateRight_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ProcessRotateRight.Click
    PictureBox1.Image.RotateFlip(RotateFlipType.Rotate90FlipNone)
    PictureBox1.Invalidate()
    Dim tmp As Integer
    tmp = PictureBox1.Width
    PictureBox1.Width = PictureBox1.Height
    PictureBox1.Height = tmp
End Sub
```
Exchanging Images through the Clipboard

Whether you use bitmap images or create graphics from scratch with the Visual Basic drawing methods, sooner or later you’ll want to exchange them with other Windows applications. To do so, you use the Clipboard and its GetDataObject and SetDataObject methods. The SetDataObject method accepts the data to be placed on the Clipboard as argument. To copy the bitmap displayed on the PictureBox1 control to the Clipboard, use the following statement:

```
Clipboard.SetDataObject(PictureBox1.Image)
```

A second form of the SetDataObject method accepts an additional argument: a True/False value that specifies whether the contents of the Clipboard will remain on it after the application that placed them there has terminated.

The GetDataObject method is a bit more complicated. This method returns an IDataObject, which in turn exposes three methods:

- **GetData**  Retrieves the clipboard’s contents.
- **GetDataPresent**  Returns True if the Clipboard contains data of a specific type.
- **GetFormats**  Returns all the formats supported by the Clipboard.

The GetData method accepts a single argument, which is the format of the desired data. The Clipboard doesn’t just return any data it may contain; instead, you must specify the type of data you expect to read into your application when you request it. To read the bitmap stored in the Clipboard and display it on the PictureBox1 control, you must use a statement like the following:

```
PictureBox1.Image = Clipboard.GetDataObject.GetData(DataFormats.Bitmap)
```

The DataFormats enumeration contains a member for each type of data it recognizes (it includes types like Text, HTML, WaveAudio, and many more. If you can’t be sure whether the Clipboard contains data of a specific type, use the GetDataPresent method passing as argument the desired type. If the Clipboard’s data are of this type, the GetDataPresent method will return True:

```
Clipboard.GetDataObject.GetDataPresent(dataFormat)
```

where `dataFormat` is a member of the DataFormats enumeration. You can also specify a second argument, which is a True/False value that determines whether the Clipboard should attempt to automatically convert its data to the specified format.

**VB.NET at Work: The ImageClipboard Project**

The ImageClipboard project, whose main form is shown in Figure 14.5, allows you to exchange images between your VB application and any other image-aware application running under Windows through the Clipboard.

The Load Image button prompts the user to select an image with the Open dialog box (you’ve seen this code in the ImageLoad application). The Clear Image button clears the PictureBox control by calling the Clear method of the Graphics object (this object is discussed in the following section). The other two buttons move a bitmap to and from the Clipboard, as explained already. Listing 14.5 shows the event handlers for all four buttons on the form.

---

*Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com*
LISTING 14.5: THE IMAGECLIPBOARD APPLICATION

Private Sub btnPaste_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnPaste.Click
    If Clipboard.GetDataObject.GetDataPresent(DataFormats.Bitmap) Then
        PictureBox1.Image = Clipboard.GetDataObject.GetData(DataFormats.Bitmap)
    Else
        MsgBox("The Clipboard doesn’t contain a bitmap!")
    End If
End Sub
Private Sub btnCopy_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCopy.Click
    Clipboard.SetDataObject(PictureBox1.Image)
End Sub
Private Sub btnLoad_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnLoad.Click
    OpenFileDialog1.Filter = "Images (*.bmp; *.tif; *.jpg; *.gif)"
    If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
        PictureBox1.Image = Image.FromFile(OpenFileDialog1.FileName)
    End If
End Sub
Private Sub btnClear_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnClear.Click
    PictureBox1.CreateGraphics.Clear(Color.Black)
End Sub

This example concludes our discussion of the Image object. So far, you have learned how to add images to the application’s user interface, properly resize bitmaps, and perform simple geometrical transformations like rotating and flipping. In the following section, you’ll learn how to create your own graphics.
Drawing with GDI+

GDI stands for Graphics Design Interface, and it’s a collection of classes that enables you to create graphics, text, and images. In short, GDI is the graphics engine of Windows. GDI has been around for many years, and its latest version is GDI+, which is the only way to create graphics in .NET. All the drawing statements of VB6 are gone, and although it’s more difficult to create graphics with GDI+, the new graphics engine is faster, richer, and common for all .NET languages.

One of the basic characteristics of GDI+ is that it’s stateless. This means that a graphics operation is totally independent of the previous one and can’t affect the following one. To draw a line, you must specify a Pen object and the two endpoints of the line. You must do the same for the next line you’ll draw. You can’t assume that the second line will use the same pen, or that it will start at the point where the previous line ended. There isn’t even a default font for text-drawing methods. Every time you draw some text, you must specify the font in which the text will be rendered, as well as the Brush object that will be used to draw the text.

The GDI+ classes reside in the following namespaces, and you must import one or more of them in your projects: System.Drawing, System.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text. In this chapter we’ll explore all three aspects of GDI+, namely vector drawing, imaging, and typography, starting with the basic drawing objects.

Before you start drawing, you must select the surface you want to draw on, the type of shapes you want to draw, and the instrument you’ll use to draw with. The surface on which you can draw is a Graphics object. This object exposes numerous methods for drawing basic (and not so basic) shapes. To draw on a form, or a control, we request the proper Graphics object, which exposes all the drawing methods.

The next step is to decide what instrument you’ll use to draw with. There are two major drawing instruments, the Pen object and the Brush object. You use pens to draw stroked shapes (lines, rectangles, curves) and brushes to draw filled shapes (any area enclosed by a shape). The main characteristics of the Pen object are its color and its width (the size of the trace left by the pen). The main characteristic of the Brush object is the color or pattern that will fill the shape. An interesting variation of the Brush object is the gradient brushes, which change color as you move from one point of the shape you want to fill to another. You can start filling a shape with red in the middle and specify that as you move toward the edges of the shape, the fill color fades to yellow.

After you have specified the drawing surface and the drawing instrument, you draw an actual shape by calling the appropriate method of the Graphics object. Here’s a simple example of a few statements that draw a line on the form.

```vbnet
Dim redPen As Pen = New Pen(Color.Red, 2)
Dim point1 As Point = New Point(10,10)
Dim point2 As Point = New Point(120,180)
Me.CreateGraphics.DrawLine(redPen, point1, point2)
```

The first statement declares a new Pen object, which is initialized to draw in red with a width of 2 pixels. The following two statements declare and initialize two points, which are the line’s starting and ending points. The coordinates are expressed in pixels, and the origin is at the form’s top-left corner. The last statement draws a line by calling the DrawLine method. The expression `Me.CreateGraphics` retrieves the Graphics object of the form, which exposes all the drawing methods, including the DrawLine method.
method. The Graphics object is the drawing surface, and all drawing methods produce some output on this surface. You can also create a new Graphics object and associate it with the form:

```csharp
dim g as graphics
g = me.creategraphics
  g.drawLine(redpen, point1, point2)
```

The DrawLine method accepts as argument the pen it will use to draw and the line’s starting and ending points. I have used two Point objects to make the code easier to read. The DrawLine method, like all other drawing methods, is heavily overloaded. You can omit the declarations of the two points and pass their coordinates as arguments to the DrawLine method with the following statement:

```csharp
me.creategraphics.drawLine(redpen, 10, 10, 120, 180)
```

You can also omit the declaration of a Pen object variable and initialize it in the same statement that draws the line:

```csharp
me.creategraphics.drawLine(new pen(color.red, 2), 10, 10, 120, 180)
```

All coordinates are expressed by default in pixels. It’s possible to specify coordinates in different units and let GDI+ convert them to pixels before drawing. If you’re drawing molecules, your units will be tiny fractions of a millimeter (microns), while if you’re drawing the trajectories of planets, your units will be millions or billions of miles. For now, we’ll use pixels, which are quite appropriate for simple objects. Once you’ve familiarized yourself with the drawing methods, you’ll learn how to specify different coordinate systems.

**The Basic Drawing Objects**

This is a good point to introduce some of the objects we’ll be using all the time in drawing. Instead of interrupting the discussion of the more interesting drawing methods that will follow, I’d rather discuss here all the auxiliary objects used in drawing. No matter what you draw, or what drawing instrument you’re using, one or more of the objects discussed in this section will be required.

**The Graphics Object**

The Graphics object is the drawing surface. Every control you can draw on exposes a Graphics property, which is an object. The Graphics object exposes all the methods for drawing on the surface of the control. It goes without saying that the PictureBox control exposes a Graphics property, but so does the TextBox control, as well as many controls you wouldn’t expect. It’s not recommended that you draw on a TextBox control, of course, unless you’re coding a peculiar application. Bear in mind that anything you draw on the TextBox control will disappear as you start typing. You must first place the text on the control and then draw on its surface.

To retrieve the Graphics object of a control, call the control’s CreateGraphics method. Because this method returns a Graphics object, it also exposes all the methods and properties you will use to create graphics on the control. If you enter the string `Me.CreateGraphics` and a period, you will see a list of the members of the Graphics object in a drop-down list. The DpiX and DpiY properties, for example, return the horizontal and vertical resolution of the form. On an average monitor, these two properties return a resolution of 96 dots per inch.
To use the Graphics object, you must first import the library Drawing2D into your project with the following statement (if not, you will have to fully qualify the references to the drawing methods):

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

Then, declare a variable of the Graphics type and initialize it to the Graphics object returned by the control's CreateGraphics method:

```vbnet
Dim G As Graphics
G = PictureBox1.CreateGraphics
```

At this point you're ready to start drawing on the PictureBox1 control with the methods we'll discuss in the following sections. If you want to draw on the form, create a Graphics object with the form's CreateGraphics method:

```vbnet
Dim G As Graphics
G = Me.CreateGraphics
```

You can actually draw on any control that provides a CreateGraphics method.

**NOTE** The Graphics object is initialized to the control's drawing surface the moment you create it. If the form is resized at runtime, the Graphics object won't change and part of the drawing surface may not be available for drawing. If you create a Graphics object to represent a form in the form's Load event handler, this object will represent the surface of the control the moment the Graphics object was created. If the form is resized at runtime, the drawing methods you apply to the Graphics object will take effect in part of the form. The most appropriate event for initializing the Graphics object and inserting the painting code is the form's Paint event. This event is fired when the form must be redrawn. Insert your drawing code there and create a Graphics object in the Paint event. Then draw on the Graphics object and release it when you're done.

The Graphics object exposes the following basic properties, in addition to the drawing methods discussed in the following sections.

- **DpiX, DpiY** The horizontal and vertical resolutions of the drawing surface. These properties are expressed in pixels per inch (or dots per inch, if the drawing surface is your printer). A distance of Graphics.DpiX pixels will be exactly one inch on the monitor. If you plan to work with a unit other than pixels, you should take advantage of the PageUnit property.

- **PageUnit** The unit in which you want to express the coordinates on the Graphics object. Its value can be a member of the GraphicsUnit enumeration (Table 14.3).

- **TextRenderingHint** This property specifies how the Graphics object will render text; its value is one of the members of the TextRenderingHint enumeration: AntiAlias, AntiAliasGridFit, ClearTypeGridFit, SingleBitPerPixel, SingleBitPerPixelGridFit, and SystemDefault.

- **SmoothingMode** This property is similar to the TextRenderingHint, but it applies to all shapes, not just text. Its value is one of the members of the SmoothingMode enumeration: AntiAlias, Default, HighQuality, HighSpeed, Invalid, and None.

Figure 14.6 shows an ellipse drawn with the SmoothingMode property set to AntiAlias (the one on the left) and to HighSpeed (on the right). Parts of the two ellipses were blown up with an image-processing application, so that you can see the difference in the two modes. Anti-aliased shapes
(or text, for that matter) are smoother because their edges contain shades between the drawing and background colors. These shades are introduced by GDI+ automatically when you render shapes to lessen the contrast between the two colors. As a result, anti-aliased drawings look smoother.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>The unit is 1/6 of an inch.</td>
</tr>
<tr>
<td>Document</td>
<td>The unit is 1/1000 of an inch.</td>
</tr>
<tr>
<td>Inch</td>
<td>The unit is one inch.</td>
</tr>
<tr>
<td>Millimeter</td>
<td>The unit is one millimeter.</td>
</tr>
<tr>
<td>Pixel</td>
<td>The unit is one pixel (the default value).</td>
</tr>
<tr>
<td>Point</td>
<td>The unit is a printer's point (1/72 of an inch).</td>
</tr>
<tr>
<td>World</td>
<td>The developer specifies the unit to be used.</td>
</tr>
</tbody>
</table>

Figure 14.6
SmoothingMode set to (left) Anti-Alias and (right) High Speed

Figure 14.7 shows the effect of the TextRenderingHint property on text. The anti-aliased text looks much better on the monitor. The ClearType setting has no effect on CRT monitors. You can see the difference only when you render text on LCD monitors, such as the new flat panel monitors or notebook monitors.
**Figure 14.7**
TextRenderingHint set to (top)
ClearType and (bottom)
AntiAlias

---

**The Point Object**
The Point object represents a point on the drawing surface and is expressed as a pair of \((x, y)\) coordinates. The \(x\) coordinate is its horizontal distance from the origin, and the \(y\) coordinate is its vertical distance from the origin. The origin is the point with coordinates \((0, 0)\), and this is the top-left corner of the drawing surface. Figure 14.8 shows the coordinates of the two opposite corners of the Graphics object and a point in its interior.

**Figure 14.8**
The origin of the default coordinate system is at the top-left corner of the drawing surface.

To create a new Point object, you must specify its \(x\) and \(y\) coordinates, represented as \(X\) and \(Y\) properties of the object. The constructor of the Point object is:

```vbnet
Dim P1 As Point
P1 = New Point(X, Y)
```

where \(X\) and \(Y\) are integer values, the point’s horizontal and vertical distances from the origin. Alternatively, you can declare a Point object and then set its \(X\) and \(Y\) properties:

```vbnet
Dim P1 As Point
P1.X = 34
P1.Y = 50
```
As you will see later, coordinates can also be specified as Single numbers (if you choose to use a coordinate system other than pixels). In this case, use the PointF object, which is identical to the Point object with the exception that its coordinates are non-integers (F stands for floating-point, and floating-point numbers are represented by the Single or Double data type).

**The Rectangle Object**

Another object quite common in drawing is the Rectangle object. Its constructor accepts as arguments the coordinates of the rectangle’s top-left corner and its width and height.

```vb
Dim box As Rectangle
box = New Rectangle(X, Y, width, height)
```

The following statement creates a rectangle whose top-left corner is 1 pixel to the right and 1 pixel down from the origin and whose dimensions are 100 and 20 pixels:

```vb
box = New Rectangle(1, 1, 100, 20)
```

The box variable represents a rectangle, but it doesn’t generate any output on the monitor. If you want to draw the rectangle, you can pass it as argument to the DrawRectangle or FillRectangle method, depending on whether you want to draw the outline of the rectangle or a filled rectangle.

Another form of the Rectangle constructor uses the Size object to specify the dimensions of the rectangle:

```vb
box = New Rectangle(point, size)
```

To create the same Rectangle object as in the last example with this form of the constructor, use the following statement:

```vb
Dim P As Point
P.X = 1
P.Y = 1
Dim S As Size
S.Width = 100
S.Height = 20
box = New Rectangle(P, S)
```

Both sets of statements create a rectangle that extends from point (1, 1) to the point ([1 + 100], [1 + 20]) or (101, 21), in the same manner as the ones shown in Figure 14.9.

**Figure 14.9**

Specifying rectangles with the coordinates of their top left corner and their dimensions

Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com
Alternatively, you can declare a Rectangle object and then set its properties, as shown here:

```
Dim box As New Rectangle
box.X = 1
box.Y = 1
box.Width = 100
box.Height = 20
```

**The Color Object**

The Color object represents a color, and there are many ways to specify a color. We'll discuss the Color object in more detail later in this chapter, in the discussion of bitmaps. In the meantime, you can specify colors by name. Declare a variable of the Color type, and initialize it to one of the named colors exposed by the Color object:

```
Dim myColor As Color
myColor = Color.Azure
```

The 128 named members of the Color object will appear in a drop-down list as soon as you enter the period following the keyword Color. You can also use the FromARGB method, which creates a new color from its basic color components (the Red, Green, and Blue components). For more information on specifying colors with this method, see the section “Specifying Colors” later in this chapter.

The Color object is used to set the color of the Pen object you draw with or the color in which a string will be rendered. You can also use the same object to assign values to any color-related property, such as the BackColor property of any control. To set the background color of a TextBox control, use the following statement:

```
TextBox1.BackColor = Color.Beige
```

**The Font Object**

The Font object represents the font to be used when rendering strings with the DrawString method. To specify a font, you must create a new Font object, set its family name, size, and style, and then pass it as argument to the DrawString method. Alternatively, you can prompt the user for a font with the Font common dialog box and use the object returned by the dialog box’s Font property as argument with the DrawString method.

To create a new Font object, use a few statements like the following:

```
Dim drawFont As New Font("Comic Sans MS", FontStyle.Bold)
```

The Font constructor has 13 forms in all. Two of the simpler forms of the constructor, which allow you specify the size and the style of the font, are shown next:

```
Dim drawFont As New Font(name, size)
Dim drawFont As New Font(name, size, style)
```

where size is an integer and style is a member of the FontStyle enumeration (Bold, Italic, Regular, Strikeout, and Underline). To specify multiple styles, combine them with the Or operator:

```
FontStyle.Bold Or FontStyle.Italic
```

Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com
You can also initialize a Font variable to an existing font. The following statement creates a Font object and initializes it to the current font of the form:

```
Dim textFont As New Font
textFont = Me.Font
```

The Font object provides the Size, Bold, and Italic properties. Unfortunately, these properties are read-only and return the attributes of the font in use. You can't turn on the bold attribute by setting the Font.Bold property to True. It would be very convenient to be able to quickly adjust the properties of an existing font and create a new one, but this isn't the case.

Of course, you can use the current settings of an existing font to create a new Font object. The following statements build a new Font object based on the settings of the form's current font. The new font belongs to the same family as the form's current font, is twice the size of this font, and has the same attributes as the form's font plus the bold attribute.

```
Dim textFont As Font
textFont = New Font(Me.Font.FontFamily, 2 * Me.Font.Size, _
                     Me.Font.Style Or FontStyle.Bold)
```

**The Pen Object**

The Pen object represents a virtual pen, which you use to draw on the Graphics object's surface. To construct a Pen object, you must specify a color and the pen's width in pixels. The following statements declare three Pen objects with the same color and different widths:

```
Dim thinPen, mediumPen, thickPen As Pen
thinPen = New Pen(Color.Black, 1)
mediumPen = New Pen(Color.Black, 3)
thickPen = New Pen(Color.Black, 5)
```

If you omit the second argument, a pen with a width of a single pixel will be created by default. Another form of the Pen object's constructor allows you to specify a brush, instead of a color:

```
Dim patternPen as Pen
patternPen = New Pen(brush, width)
```

where `brush` is a Brush object (which is discussed later in this chapter). As you will see, the drawing methods generate two types of drawings: stroked shapes and filled shapes. Stroked shapes (or outlines) are drawn with a pen, while filled shapes are drawn with a brush. To draw the outline of a shape with a pattern, you must create a Pen object based on an existing brush, and then use it with a drawing method.

The quickest method of creating a new Pen object is to use the built-in Pens collection, which creates a Pen with a width of one pixel and the color you specify. The following statement can appear anywhere a Pen object is required and will draw shapes in blue color:

```
Pen = Blue
```

**NOTE** There's an important distinction between Pens and Brushes you should bear in mind as you draw with VB.NET. You can't draw a shape with a Brush object, and you can't fill a closed shape with a Pen. It is possible, however, to draw a shape with a pattern, as long as you assign the pattern to the Pen object. Likewise, you can fill a shape with a solid color, as long as you set the color of the Brush object you're using to fill with.
The Pen object exposes these properties:

**LineJoin**  
Determines how two consecutive line segments will be joined. Its value is one of the members of the LineJoin enumeration: Bevel, Miter, MiterClipped, and Round.

**StartCap, EndCap**  
Determine the caps at the beginning and end of a line segment respectively. Their value is one of the members of the LineCap enumeration: Round, Square, Flat, Diamond, and so on.

**DashCap**  
Determines the cap to be used at the beginning and end of a dashed line. Its value is one of the members of the DashCap enumeration: Flat, Round, and Triangle.

**DashStyle**  
Determines the style of the dashed lines drawn with the specific Pen. Its value is one of the members of the DashStyle enumeration (Solid, Dash, DashDot, DashDotDot, Dot, and Custom).

**PenType**  
Determines the style of the Pen. Its value is one of the members of the PenType enumeration: HatchFilled, LinearGradient, PathGradient, SolidColor, and TextureFill.

**The Path Object**

The Path object is a combination of the various drawing entities, like lines, rectangles, and curves. You can create as many of these drawing entities and build a new entity, which is called Path. Paths are usually closed and filled with a color, a gradient or a bitmap. You can create a path in several ways. The simplest method is to create a new Path object and then use one of these methods to append the appropriate item to the path:

- `AddArc`
- `AddEllipticalArc`
- `AddLine`
- `AddPie`
- `AddPolygon`
- `AddRectangle`
- `AddString`

These methods add to the path the same shapes you can draw on the Graphics object with the methods discussed in the later section “Drawing Shapes with the Graphics Object.” There’s even an AddPath method, which adds an existing path to the current one. The syntax of the various methods that add shapes to a path is identical to the corresponding methods that draw. We simply omit the first argument (the Pen object), because all the shapes will be rendered with the same pen. The following method draws an ellipse:

```vbnet
Me.CreateGraphics.DrawEllipse(pen, 10, 30, 40, 50)
```

To add the same ellipse to a Path object, use the following statement:

```vbnet
Dim myPath As New Path
myPath.AddEllipse(10, 30, 40, 50)
```

To display the path, call the DrawPath method passing a Pen and the Path object as arguments:

```vbnet
Me.CreateGraphics.DrawPath(pen, myPath)
```
Why combine shapes into paths instead of drawing individual shapes? There are many reasons for maintaining multiple shapes as a single entity. Once the shape has been defined, you can draw multiple instances of it on the monitor, draw the same path with a different pen, or fill the path’s constituent shapes with the same bitmap or gradient. Paths are also used to create the ultimate type of gradient, the PathGradient, as you will see in the section “Path Gradients,” later in this chapter.

Later in this chapter, we’ll build an application for plotting functions. To plot a function, we’ll create a shape with all the points along the curve and draw it with a single call the DrawPath method.

**The Brush Object**
The Brush object is the instrument for filling shapes, and you can create brushes that fill with a solid color, a pattern, or a bitmap. In reality, there’s no Brush object. The Brush class is actually an abstract class that is inherited by all the objects that implement a brush, but you can’t declare a variable of the Brush type in your code. The brush objects are:

<table>
<thead>
<tr>
<th>Brush Object Type</th>
<th>Fills Shapes With</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidBrush</td>
<td>A solid color</td>
</tr>
<tr>
<td>HatchBrush</td>
<td>A hatched pattern</td>
</tr>
<tr>
<td>LinearGradientBrush</td>
<td>A linear gradient</td>
</tr>
<tr>
<td>PathGradientBrush</td>
<td>A gradient that has one starting color and many ending colors</td>
</tr>
<tr>
<td>TextureBrush</td>
<td>A bitmap</td>
</tr>
</tbody>
</table>

**Solid Brushes**
To fill a shape with a solid color, you must create a SolidBrush object with the following constructor:

```vbscript
Dim sBrush As SolidBrush
sBrush = New SolidBrush(brushColor)
```

where `brushColor` is a color value, specified with the help of the Color object. Every filled object you draw with the `sBrush` object will be filled with the color of the brush.

**Hatched Brushes**
To fill a shape with a hatch pattern, you must create a HatchBrush object with the following constructor:

```vbscript
Dim hBrush As HatchBrush
hBrush = New HatchBrush(hatchStyle, hatchColor, backColor)
```

The first argument is the style of the hatch, and it can have one of the values shown in Table 14.4. The other two arguments are the colors to be used in the hatch. The hatch is a pattern of lines drawn on a background, and the two color arguments are the color of the hatch lines and the color of the background on which the hatch is drawn.
**Table 14.4: The HatchStyle Enumeration**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackwardDiagonal</td>
<td>Diagonal lines from top-right to bottom-left</td>
</tr>
<tr>
<td>Cross</td>
<td>Vertical and horizontal crossing lines</td>
</tr>
<tr>
<td>DiagonalCross</td>
<td>Diagonally crossing lines</td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>Diagonal lines from top-left to bottom-right</td>
</tr>
<tr>
<td>Horizontal</td>
<td>Horizontal lines</td>
</tr>
<tr>
<td>Vertical</td>
<td>Vertical lines</td>
</tr>
</tbody>
</table>

**Gradient Brushes**

A gradient brush fills a shape with a specified gradient. The LinearGradientBrush fills a shape with a linear gradient, and the PathGradientBrush fills a shape with a gradient that has one starting color and many ending colors. Gradient brushes are discussed in detail in the section “Gradients” later in this chapter.

**Textured Brushes**

In addition to solid and hatched shapes, you can fill a shape with a texture using a TextureBrush object. The texture is a bitmap that is tiled as needed to fill the shape. Textured brushes are used in creating rather fancy graphics, and we’ll not explore them in this chapter.

**Drawing Shapes**

In this section, you will learn the drawing methods of the Graphics object. Before getting into the details of the drawing methods, however, let’s write a simple application that draws a couple of simple shapes on a form. First, we must create a Graphics object with the following statement:

```vbnet
Dim G As Graphics
G = Me.CreateGraphics
```

Everything you draw on the surface represented by the G object will appear on the form. Then, we must create a Pen object to draw with. The following statement creates a Pen object that draws in blue:

```vbnet
Dim P As New Pen(Color.Blue)
```

You’ve created the two basic objects for drawing: the drawing surface and the drawing instrument. Now you can draw shapes by calling the Graphics object’s methods. The following statement will print a rectangle with its top-left corner near the top-left corner of the form (at a point that’s 10 pixels to the right and 10 pixels down from the form’s corner) and is 200 pixels wide and 150 pixels tall. These are the values you must pass to the DrawRectangle method as arguments, along with the Pen object that will be used to render the rectangle:

```vbnet
G.DrawRectangle(P, 10, 10, 200, 150)
```
Let's add the two diagonals of the rectangle. The diagonals are two lines, one from the top-left to the bottom-right corner of the rectangle and another from top-right to bottom-left. Here are the two statements that draw the diagonals:

G.DrawLine(P, 10, 10, 210, 160)
G.DrawLine(P, 210, 10, 10, 160)

We've written all the statements to create a shape on the form, but where do we insert them? Let's try a Button. Start a new project, place a button on it, and then insert the statements of Listing 14.6 in the Button's Click event handler.

**Listing 14.6: Drawing Simple Shapes**

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    Dim P As New Pen(Color.Blue)
    G.DrawRectangle(P, 10, 10, 200, 150)
    G.DrawLine(P, 10, 10, 210, 160)
    G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

Run the application and click the button. You will see the shape shown in Figure 14.10. This figure was created by the SimpleShapes application on the CD.

**Figure 14.10**
The output of Listing 14.6

**Persistent Drawing**

If you switch to the Visual Studio IDE, or any other window, and then return to the form of the SimpleShapes application, you'll see that the drawing has disappeared! If you're a VB6 programmer, you've recognized that the form's AutoRedraw property isn't True. But there's no AutoRedraw property in VB.NET. Only the bitmap of the PictureBox is persistent. Everything you draw on the Graphics object is temporary. It doesn't become part of the Graphics object and is visible only while the control, or the form, need not be redrawn. As soon as the form is redrawn, the shapes disappear.
So, how do we make the output of the various drawing methods permanent on the form? Microsoft suggests placing all the graphics statements in the OnPaint method, which is activated automatically when the form is redrawn. OnPaint is a method of the form, which is invoked automatically by the operating system and, in turn, it invokes the Paint event. To draw something every time the form is redrawn, place the necessary statements in the OnPaint method.

The OnPaint method accepts a single argument, the e argument, which, among other properties, exposes the form’s Graphics object. You can create a Graphics object in the OnPaint method and then draw on this object. Listing 14.7 is the OnPaint event handler that creates the shape shown in Figure 14.10 and refreshes the form every time it’s totally or partially covered by another form. Delete the code in the button’s Click event handler and insert the subroutine from the listing into the form’s code window.

**Listing 14.7: Programming the OnPaint Event**

```vbscript
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
    Dim G As Graphics
    G = e.Graphics
    Dim P As New Pen(Color.Blue)
    G.DrawRectangle(P, 10, 10, 200, 150)
    G.DrawLine(P, 10, 10, 210, 160)
    G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

If you run the application now, it works like a charm. The shapes appear to be permanent, even though they’re redrawn every time you switch to the form. All the samples that come with Visual Studio place the graphics statements in the OnPaint method, so that they’re executed every time the form is redrawn.

This technique is fine for a few graphics elements you want to place on the form to enhance its appearance. But many applications draw something on the form in response to user actions, like the click of a button or a menu commands. Using the OnPaint method in a similar application is out of the question. The drawing isn’t the same, and you must figure out from within your code which shapes you have to redraw at any given time. Consider a drawing application. The current drawing evolves according to the commands you issue. The code in the OnPaint method can’t execute a few drawing commands to regenerate the drawing. Keeping track of the drawing commands that were executed and the order in which they were executed is quite a task. The solution is to make the drawing permanent on the Graphics object, so it won’t have to be redrawn every time the form is hidden or resized.

It is possible to make the graphics permanent by drawing not on the Graphics object, but directly on the control’s (or the form’s) bitmap. The Bitmap object contains the pixels that make up the image and is very similar to the Image object. As you will see, you can create a Bitmap object and assign it to an Image object. In the image-processing application we’ll develop toward the end of this chapter,

---

*Copyright ©2002 SYBEX, Inc., Alameda, CA [www.sybex.com](http://www.sybex.com)*
you'll learn how to extract the bitmap from a PictureBox control, process the pixels of a bitmap, and then assign the processed bitmap back to the control's Image property. In the meantime, you can use the code of Listing 14.7 to create a drawing surface that doesn't have to be constantly redrawn.

To create this "permanent" drawing surface, you must first create a Bitmap object that has the same dimensions as the PictureBox control you want to draw on:

```vba
Dim bmp As Bitmap
bmp = New Bitmap(PictureBox1.Width, PictureBox1.Height)
```

The `bmp` variable represents an empty bitmap. Then, we set the control's Image property to this bitmap with the following statement:

```vba
PictureBox1.Image = bmp
```

Immediately after that, you must set the bitmap to the control's background color with the Clear method:

```vba
G.C lear(PictureBox1.BackGroundColor) ;
```

After the execution of this statement, anything we draw on the `bmp` bitmap is shown on the surface of the PictureBox control and is permanent. All we need is a Graphics object that represents the bitmap, so that we can draw on the control. The following statement creates a Graphics object based on the `bmp` variable:

```vba
Dim G As Graphics
G = Graphics.FromImage(bmp)
```

Now, we're in business. We can call the `G` object's drawing methods to draw and create permanent graphics on the PictureBox control. You can put all the statements presented so far in a function that returns a Graphics object (Listing 14.8) and use it in your applications.

**LISTING 14.8: RETRIEVING A GRAPHICS OBJECT FROM A PICTUREBOX'S BITMAP**

```vba
Function GetGraphicsObject(ByVal PBox As PictureBox) As Graphics
Dim bmp As Bitmap
bmp = New Bitmap(PBox.Width, PBox.Height)
PBox.Image = bmp
Dim G As Graphics
G = Graphics.FromImage(bmp)
Return G
End Function
```

To create permanent drawings on the surface of the PictureBox control, you must call the `GetGraphicsObject()` function to obtain a Graphics object from the control's bitmap. The Form object doesn't expose an Image property, so you must use its BackgroundImage property. Listing 14.9 is the revised `GetGraphicsObject()` function for the Form object.
LISTING 14.9: RETRIEVING A GRAPhICS OBJECT FROM A FORM’S BITMAP

Function GetGraphicsObject() As Graphics
    Dim bmp As Bitmap
    bmp = New Bitmap(Me.Width, Me.Height)
    Dim G As Graphics
    Me.BackgroundImage = bmp
    G = Graphics.FromImage(bmp)
    Return G
End Function

Let’s revise the SimpleShapes application so that it draws permanent shapes on the form. Create a new project and place two Button controls on it. Insert the GetGraphicsObject() function in the form’s code window and then the statements shown in Listing 14.10 behind each button. The listing shows the entire code of the SimpleGraphics application, which is on the CD.

LISTING 14.10: THE SIMPLEGRAPHICS APPLICATION

Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    G.Clear(Color.Silver)
    Dim P As New Pen(Color.Blue)
    G.DrawRectangle(P, 10, 10, 200, 150)
    G.DrawLine(P, 10, 10, 210, 160)
    G.DrawLine(P, 10, 10, 160)
    Me.Invalidate()
End Sub

Private Sub Button2_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button2.Click
    Me.CreateGraphics.DrawEllipse(Pens.Red, 10, 10, 200, 150)
End Sub

Function GetGraphicsObject() As Graphics
    Dim bmp As Bitmap
    bmp = New Bitmap(Me.Width, Me.Height)
    Dim G As Graphics
    Me.BackgroundImage = bmp
    G = Graphics.FromImage(bmp)
    Return G
End Function
The first button (the Draw On Bitmap button of the SimpleGraphics application) draws on the Graphics object derived from the form's background bitmap. Anything drawn on this object is permanent. The second button (the Draw On Graphics button) uses the Graphics object returned by the form's CreateGraphics method to draw an ellipse in red color, inscribed in the rectangle. The ellipse isn't permanent. If you click both buttons, you will see the rectangle and its two diagonals, as well as the ellipse. Switch to another window and then bring the application to the foreground. The ellipse will not be there, because it wasn't drawn permanently on the form.

As you can guess, it's possible to combine the two methods and draw shapes that are permanent and shapes that are not. To erase the non-permanent shapes, call the control's Invalidate method, which redraws the control. Anything drawn on the Graphics object returned by the control's CreateGraphics method will disappear. The Invalidate method can be called without an argument to refresh (invalidate) the entire control. Or it can be called with a Rectangle object as argument, in which case it will invalidate the area of the control specified by the Rectangle object.

Now that you know how to draw on the Graphics object and you're familiar with the basic drawing objects, we can discuss the drawing methods in detail. In the following sections, I use the CreateGraphics method to retrieve the drawing surface of a PictureBox or form to keep the examples short. You can modify any of the projects to draw on the Graphics object derived from a bitmap. All you have to do is change the statements that create the G variable.

**Drawing Methods**

With basic objects out of the way, we can now focus on the drawing methods themselves. There are many drawing methods, one for each basic shape. You can create much more elaborate shapes by combining the methods described in the following sections.

All drawing methods have a few things in common. The first argument is always a Pen object, which will be used to render the shape on the Graphics object. The following arguments are the parameters of a shape: they determine the location and dimensions of the shape. The DrawLine method, for example, needs to know the endpoints of the line to draw, while the DrawRectangle method needs to know the origin and dimensions of the rectangle to draw. The parameters needed to render the shape are passed as arguments to each drawing method, following the Pen object.

The drawing methods can also be categorized in two major groups: the methods that draw stroked shapes (outlines) and the methods that draw filled shapes. The methods in the first group start with the "Draw" prefix (DrawRectangle, DrawEllipse, and so on). The methods of the second group start with the "Fill" prefix (FillRectangle, FillEllipse, and so on). Of course, some DrawXXX methods don't have an equivalent FillXXX method. For example, you can't fill a line or an open curve, so there are no FillLine or FillCurve methods.

Another difference between the drawing and filling methods is that the filling methods use a Brush object to fill the shape—you can't fill a shape with a pen. So, the first argument of the methods that draw filled shapes is a Brush object, not a Pen object. The remaining arguments are the same, because you must still specify the shape to be filled, just as you would specify the shape to be drawn. In the following sections, I will present in detail the shape-drawing methods but not the shape-filling methods. If you can use a drawing method, you can just as easily use its filling counterpart.

Table 14.5 shows the names of the drawing methods. The first column contains the methods for drawing stroked shapes and the second column contains the corresponding methods for drawing filled shapes (if there's a matching method).
<table>
<thead>
<tr>
<th>DRAWING METHOD</th>
<th>FILLING METHOD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawArc</td>
<td></td>
<td>Draws an arc</td>
</tr>
<tr>
<td>DrawBezier</td>
<td></td>
<td>Draws very smooth curves with fixed endpoints, whose exact shape is determined by two control points</td>
</tr>
<tr>
<td>DrawBeziers</td>
<td></td>
<td>Draws multiple Bezier curves in a single call</td>
</tr>
<tr>
<td>DrawClosedCurve</td>
<td>FillClosedCurve</td>
<td>Draws a closed curve</td>
</tr>
<tr>
<td>DrawCurve</td>
<td></td>
<td>Draws curves that pass through certain points</td>
</tr>
<tr>
<td>DrawEllipse</td>
<td>FillEllipse</td>
<td>Draws an ellipse</td>
</tr>
<tr>
<td>DrawIcon</td>
<td></td>
<td>Renders an icon on the Graphics object</td>
</tr>
<tr>
<td>DrawImage</td>
<td></td>
<td>Renders an image on the Graphics object</td>
</tr>
<tr>
<td>DrawLine</td>
<td></td>
<td>Draws a line segment</td>
</tr>
<tr>
<td>DrawLines</td>
<td></td>
<td>Draws multiple line segments in a single call</td>
</tr>
<tr>
<td>DrawPath</td>
<td>FillPath</td>
<td>Draws a GraphicsPath object</td>
</tr>
<tr>
<td>DrawPie</td>
<td>FillPie</td>
<td>Draws a pie section</td>
</tr>
<tr>
<td>DrawPolygon</td>
<td>FillPolygon</td>
<td>Draws a polygon (a series of line segments between points)</td>
</tr>
<tr>
<td>DrawRectangle</td>
<td>FillRectangle</td>
<td>Draws a rectangle</td>
</tr>
<tr>
<td>DrawRectangles</td>
<td>FillRectangles</td>
<td>Draws multiple rectangles in a single call</td>
</tr>
<tr>
<td>DrawString</td>
<td>FillRegion</td>
<td>Draws a string in the specified font on the drawing surface</td>
</tr>
</tbody>
</table>

Some of the drawing methods allow you to draw multiple shapes of the same type, and they're properly named DrawLines, DrawRectangles, and DrawBeziers. We simply supply more shapes as arguments, and they're drawn one after the other with a single call to the corresponding method. The multiple shapes are stored in arrays of the same type, as the individual shapes. The DrawRectangle method, for example, accepts as argument the Rectangle object to be drawn. The DrawRectangles method accepts as argument an array of Rectangle objects and draws them in a single call.

DrawLine

The DrawLine method draws straight line-segments between two points with a pen supplied as argument. The simplest forms of the DrawLine method are the following:

```csharp
Graphics.DrawLine(pen, x1, y1, x2, y2)
Graphics.DrawLine(pen, point1, point2)
```

where the coordinates are expressed in pixels (or the current coordinate system) and point1 and point2 are either Point or PointF objects, depending on the coordinate system in use.
**DrawRectangle**

The DrawRectangle method draws a stroked rectangle and has two forms:

```csharp
Graphics.DrawRectangle(pen, rectangle)
Graphics.DrawRectangle(pen, X1, Y1, width, height)
```

The `rectangle` argument is a Rectangle object that specifies the shape to be drawn. In the second form of the method, the arguments `X1` and `Y1` are the coordinates of the rectangle’s top-left corner and the other two arguments are the dimensions of the rectangle. All these arguments can be integers or singles, depending on the coordinate system in use. However, they must all be of the same type.

The following statements draw two rectangles, one inside the other. The outer rectangle is drawn with a red pen with the default width, while the inner rectangle is drawn with a 3-pixel-wide green pen.

```csharp
Graphics.DrawRectangle(Pens.Red, 100, 100, 200, 100)
Graphics.DrawRectangle(new Pen(Color.Green, 3), 125, 125, 75, 50)
```

**DrawEllipse**

An ellipse is an oval or circular shape, determined by the rectangle that encloses it. The two dimensions of this rectangle are the ellipse’s major and minor diameters. Instead of giving you a mathematically correct definition of an ellipse, I’ve prepared a few ellipses with different ratios of their two diameters. These ellipses are shown in Figure 14.11. The figure was prepared with the GDIPlus application, which demonstrates a few more graphics operations; you will find it in this chapter’s folder on the CD. The ellipse is oblong along the direction of the major diameter and squashed along the direction of the minor diameter. If the two diameters are exactly equal, the ellipse becomes a circle. Indeed, the circle is just a special case of the ellipse.

To draw an ellipse, call the DrawEllipse method, which has two basic forms:

```csharp
Graphics.DrawEllipse(pen, rectangle)
Graphics.DrawEllipse(pen, X1, Y1, width, height)
```

**Figure 14.11**

Two ellipses with their enclosing rectangles
The arguments are the same as with the DrawRectangle method, because an ellipse is basically a circle deformed to fit in a rectangle. The two ellipses and their enclosing rectangles shown in Figure 14.11 were generated with the statements of Listing 14.11.

**Listing 14.11: Drawing Ellipses and Their Enclosing Rectangles**

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
    G.FillRectangle(Brushes.Silver, ClientRectangle)
    Dim R1, R2 As Rectangle
    R1 = New Rectangle(10, 10, 160, 320)
    R2 = New Rectangle(200, 85, 320, 160)
    G.DrawEllipse(New Pen(Color.Black, 3), R1)
    G.DrawRectangle(Pens.Black, R1)
    G.DrawEllipse(New Pen(Color.Black, 3), R2)
    G.DrawRectangle(Pens.Red, R2)
End Sub
```

The ellipses were drawn with a pen that is three pixels wide. As you can see in the figure, the width of the ellipse is split to the inside and outside of the enclosing rectangle, which is drawn with a 1-pixel-wide pen.

**DrawPie**

A pie is a shape similar to a slice of a pie: an arc along with the two line segments that connect its endpoints to the center of the circle, or the ellipse, to which the arc belongs. The DrawPie method accepts as arguments the pen with which it will draw the shape, the circle to which the pie belongs, the arc's starting angle, and its sweep angle. The circle (or the ellipse) of the pie is defined with a rectangle. The starting and sweeping angles are measured clockwise. The DrawPie method has two forms, which are:

```vbnet
Graphics.DrawPie(pen, rectangle, start, sweep)
Graphics.DrawPie(pen, X, Y, width, height, start, sweep)
```

The two forms of the method differ in how the rectangle is defined (a Rectangle object versus its coordinates and dimensions). The `start` argument is the pie's starting angle, and `sweep` is the angle of the pie. The ending angle is `start+sweep`. Angles are measured in degrees (there are 360 degrees in a circle) and increase in a clockwise direction. The 0 angle corresponds to the horizontal axis, and the vertical axis forms a 90-degree angle with the horizontal axis.

The following statements create a pie chart by drawing individual pie slices. Each pie starts where the previous one ends, and the sweeping angles of all pies add up to 360 degrees, which corresponds to a full rotation (a full circle). Figure 14.12 shows the output produced by the Listing 14.12.
Figure 14.12
A simple pie chart generated with the PieChart method.

Listing 14.12: Drawing a Simple Pie Chart with the FillPie Methods

Private Sub Button2.Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
    Dim G As System.Drawing.Graphics
    G = Me.CreateGraphics
    Dim brush As System.Drawing.SolidBrush
    Dim rect As Rectangle
    brush = New System.Drawing.SolidBrush(Color.Green)
    Dim Angles() As Single = {0, 43, 79, 124, 169, 252, 331, 360}
    Dim Colors() As Color = {Color.Red, Color.Cornsilk, Color.Firebrick, _
                              Color.OliveDrab, Color.LawnGreen, _
                              Color.SandyBrown, Color.MidnightBlue}
    G.Clear(Color.Ivory)
    rect = New Rectangle(100, 10, 300, 300)
    Dim angle As Integer
    For angle = 1 To Angles.GetUpperBound(0)
        brush.Color = Colors(angle - 1)
        G.FillPie(brush, rect, Angles(angle - 1), _
                   Angles(angle) - Angles(angle - 1))
    Next
    G.DrawEllipse(Pens.Black, rect)
End Sub

The code sets up two arrays, one with angles and another with colors. The Angles array holds the starting angle of each pie. The sweep angle of each pie is the difference between its own starting angle and the starting angle of the following pie. The sweep angle of the first pie is Angles(1) - Angles(0), which is 43 degrees. The loop goes through each pie and draws it with a color it picks from the Colors array, based on the angles stored in the Angles array.

The second button on the PieChart project's form draws the same pie chart, but it also connects each slice's endpoints to the center of the circle. The code behind this button is identical to the code.
shown in Listing 14.6 with the exception that after calling the FillPie method (which draws the
filled pie shape), it calls the DrawPie method to draw the outline of the pie.

Notice that the FillPie method doesn’t connect the pie’s endpoints to the center of the ellipse.
Use the DrawEllipse method to draw the complete outline of the pie.

### DrawPolygon

This method draws an arbitrary polygon. It accepts two arguments, which are the Pen that it will use
to render the polygon and an array of points that define the polygon. The polygon has as many sides
(or vertices) as there are points in the array, and it’s always closed, even if the first and last points are
not identical. In fact, you need not repeat the starting point at the end, because the polygon will be
automatically closed. The syntax of the DrawPolygon method is:

```vba
Graphics.DrawPolygon(pen, points())
```

where `points` is an array of points, which can be declared with a statement like the following:

```vba
Dim points() As Point = {New Point(x1, y1), New Point(x2, y2), ...}
```

### DrawCurve

Curves are smooth lines drawn as **cardinal splines**. A real spline was a flexible object (made of soft
wood) that designers used to flex on the drawing surface with spikes. The spline would go through
all the fixed points, but the shape between the fixed points was a smooth curve. The entire spline
yielded a smooth curve that passed through all the spikes that held it in place. If the spline weren’t
flexible enough, it would break. In modern computer graphics, there are mathematical formulas that
describe the path of the spline through the fixed points and take into consideration the tension (the
degree of flexibility) of the spline. A more flexible spline yields a curve that bends easily. Less flexible
splines do not bend easily around their fixed points. Computer-generated splines do not break, but
they can take unexpected shapes.

To draw a curve with the DrawCurve method, you specify the locations of the spikes (the points
which the spline must go through) and the spline’s tension. If the tension is 0, the spline is totally
flexible, like a rubber band: all the segments between points are straight lines. The higher the tension,
the smoother the curve will be. Figure 14.13 shows four curves passing through the same points, but
each curve is drawn with a different tension value. The curves shown in the figure were drawn with
the GDIPPlus project (using the Ordinal Curves button).

The simplest form of the DrawCurve method has the following syntax:

```vba
Graphics.DrawCurve(pen, points, tension)
```

where `points` is an array of points. The first and last elements of the array are the curve’s endpoints,
and the curve will go through the remaining points.

An alternate form of the method lets you specify the curve’s first fixed point in the array, as well as
the number of segments that make up the curve:

```vba
Graphics.DrawCurve.(pen, points, offset, segments, tension)
```

The `offset` and `segments` arguments allow you to work with a portion of the `points` array, rather than
with the entire array. The `points` array must contain at least four points—the two endpoints and two
more control points along the curve. The `tension` argument is optional, and if you omit it, the curve
will be drawn with a tension of 1.
**Figure 14.13**
These curves go through the same points, but they have different tensions.

The curves shown in Figure 14.13 were produced by the code shown in Listing 14.13. Notice that a tension of 0.5 is practically the same as 0 (the spline bends around the fixed points like a rubber band). If you drew the same curve with a tension of 5, you'd get an odd curve indeed. The reason is, although a physical spline would break, the mathematical spline takes an unusual shape to accommodate the fixed points.

**Listing 14.13: Curves with Common Fixed Points and Different Tensions**

```vbc
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.HighQuality
    Dim points() As Point = {New Point(20, 50), New Point(220, 190), _
                               New Point(330, 80), New Point(450, 280)}
    G.DrawCurve(Pens.Blue, points, 0.1)
    G.DrawCurve(Pens.Red, points, 0.5)
    G.DrawCurve(Pens.Green, points, 1)
    G.DrawCurve(Pens.Black, points, 2)
End Sub
```

**DrawBezier**
The DrawBezier method draws Bezier curves, which are smoother than cardinal splines. A Bezier curve is defined by two endpoints and two control points. The control points act as magnets. The curve is the trace of a point that starts at one of the endpoints and moves toward the second one. As it moves, the point is attracted by the two control points. Initially, the first control point's influence is predominant. Gradually, the curve comes into the second control point's field, and it ends at the second endpoint.
The DrawBezier method accepts a pen and four points as arguments:

```vbnet
Graphics.DrawBezier(pen, X1, Y1, X2, Y2, X3, Y3, X4, Y4)

Graphics.DrawBezier(pen, point1, point2, point3, point4)
```

Figure 14.14 shows four Bezier curves, which differ in the y coordinate of the third control point. All control points are marked with little squares: one each for the three points that are common to all curves and four in a vertical column for the point that differs in each curve. The code, shown in Listing 14.14, draws the little squares at the control points and then draws the four Bezier curves. The endpoints and one control point (P1, P2, and P4) remain the same, while the other control point (P3) is set to four different values. Notice how far the control point must go to have a significant effect on the curve’s shape.

**Figure 14.14**
Bezier curves and their control points

---

**Listing 14.14: Drawing Bezier Curves and Their Control Points**

```vbnet
Private Sub Button3.Click(ByVal sender As System.Object, 
ByVal e As System.EventArgs) Handles Button3.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
    G.FillRectangle(Brushes.Silver, ClientRectangle)
    Dim P1 As New Point(120, 150)
    Dim P2 As New Point(220, 90)
    Dim P3 As New Point(330, 30)
    Dim P4 As New Point(410, 110)
    Dim sqrSize As New Size(6, 6)
    G.FillRectangle(Brushes.Black, New Rectangle(P1, sqrSize))
    G.FillRectangle(Brushes.Black, New Rectangle(P2, sqrSize))
    G.FillRectangle(Brushes.Red, New Rectangle(P3, sqrSize))
    G.FillRectangle(Brushes.Black, New Rectangle(P4, sqrSize))
    G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
```

*Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com*
P3 = New Point(330, 130)
G.FillRectangle(Brushes.Red, New Rectangle(P3, scrSize))
G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
P3 = New Point(330, 330)
G.FillRectangle(Brushes.Red, New Rectangle(P3, scrSize))
G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
P3 = New Point(330, 330)
G.FillRectangle(Brushes.Red, New Rectangle(P3, scrSize))
G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
End Sub

The calls to the FillRectangle method draw the little boxes that represent the control points. To draw the curve, all you need is to specify the four control points and pass them along with a Pen object to the DrawBezier method.

**DrawPath**
This method accepts a Pen object and a Path object as arguments and renders the specified path on the screen:

```
Graphics.DrawPath(pen, path)
```

To construct the Path object, use the AddXXX methods (AddLine, AddRectangle, and so on), as discussed in the section “The Path Object,” earlier in this chapter. You will find an example of how to use the Path object later in this chapter, when you’ll learn how to plot functions.

**DrawString**
The DrawString method renders a string on the drawing surface. The string may be rendered on a single line or multiple lines (there are different forms of the DrawString method for each type of text rendering). As a reminder, the TextRenderingHint property of the Graphics object allows you to specify the quality of the rendered text.

The simplest form of the DrawString method is:

```
Graphics.DrawString(string, font, brush, X, Y)
```

The first argument is the string to be rendered in the font specified with the second argument. The text will be rendered with the Brush object specified with the brush argument. Here, X and Y are the coordinates of the string’s top-left corner when it will be rendered.

While working with strings, you frequently need to know the actual dimensions of the string when rendered with the DrawString method in a specific font. The MeasureString method allows you to retrieve the metrics of a string before actually drawing it. This method returns a SizeF structure with the width and height of the string when rendered. Having this information allows you to align your strings on the drawing surface. You can also pass a Rectangle object as argument to the MeasureString method to find out how many lines it will take to render the string on the rectangle.

The simplest form of the MeasureString method is:

```
Dim textSize As SizeF
textSize = Me.Graphics.MeasureString(string, font)
```
where \textit{string} is the string to be rendered and \textit{font} is the font in which the string will be rendered. To center a string on the form, use the X coordinate returned by the following expression:

\[
\begin{align*}
\text{Dim } & \text{ textSize As SizeF} \\
\text{Dim } & \text{ X As Integer, Y As Integer } = 0 \\
\text{textSize } & = \text{Me.Graphics.MeasureString(string, font)} \\
\text{X } & = (\text{Me.Width } - \text{textSize.Width}) / 2 \\
\text{G.DrawString( Centered string, font, brush, X, Y)}
\end{align*}
\]

We subtract the rendered string's length from the form's width, and we split the difference in half at the two sides of the string.

Figure 14.15 shows a string printed at the center of the form (by the Draw Centered String button of the TextEffects project), and the two lines pass through the same point. Listing 14.15 shows the statements that produced the string at the middle of this form. This listing is part of the TextEffects project, which you will find in this chapter's folder on the CD.

\textbf{Figure 14.15}

Centering a string on a form

\textbf{Listing 14.15: Print a String Centered on the Form}

```vbnet
Private Sub Center(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCentered.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
    G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
    FontDialog1.Font = Me.Font
    FontDialog1.ShowDialog()
    Dim txtFont As Font
    txtFont = FontDialog1.Font
    G.DrawLine(New Pen(Color.Green), CInt(Me.Width / 2), CInt(0), _
               CInt(Me.Width / 2), CInt(Me.Height))
    G.DrawLine(New Pen(Color.Green), 0, CInt(Me.Height / 2), _
               CInt(Me.Width), CInt(Me.Height / 2))
    Dim txtLen, txtHeight As Integer
    Dim textSize As SizeF
```

Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com
txtSize = CMeasureString('Visual Basic.NET', txtFont)
Dim txtX, txtY As Integer
txtX = (Me.Width - txtSize.Width) / 2
txtY = (Me.Height - txtSize.Height) / 2
    txtX, txtY)
End Sub

As you can see, the coordinates passed to the DrawString method (variables txtX and txtY) are the
coordinates of the top-left corner of the rectangle that encloses the first character of the string. After
drawing the string, the code calls the MeasureString method to retrieve the rectangle that encloses
the string (the boxSize variable) and prints this rectangle on the form.

Another form of the DrawString method accepts a rectangle as argument and draws the string in
this rectangle, breaking the text into multiple lines if needed. The syntax of this form of the method is:

Graphics.DrawString(string, font, brush, rectanglef)
Graphics.DrawString(string, font, brush, rectanglef, stringFormat)

If you want to render text in a box, you will most likely use the equivalent form of the Measure-
String method to retrieve the metrics of the text on the rectangle. This form of the MeasureString
method returns the number of lines it will take to render the string on the rectangle, and it has the
following syntax:

e.Graphics.MeasureString(string, Font, fitSize, StringFormat, lines, cols)

string is the text to be rendered, and Font is the font in which the string will be rendered. The fitSize
argument is a SizeF object that represents the width and height of a rectangle, where the string must fit.
The lines and cols variables are passed by reference, and they are set by the MeasureString method
to the number of lines and number of characters that will fit in the specified rectangle. The exact
location of the rectangle doesn’t make any difference—only its dimensions matter, and that’s why
the third argument is a SizeF object and not a Rectangle object.

Figure 14.16 shows a string printed in two different rectangles. The sample code can be found in
the TextEffects project on the CD, and Figure 14.16 was created with the Draw Boxed Text button.
The code that produced the figure is shown in Listing 14.16.
LISTING 14.16: PRINTING TEXT IN A RECTANGLE

Private Sub BoxedText(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnBoxed.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
    Dim txt As String = "This text was rendered in a rectangle with the ' & _
    'DrawString method of the form's Graphics object."
    ' Make the string longer
    txt = txt & txt & txt & txt & txt
    G.DrawString(txt, New Font("verdana", 12, FontStyle.Regular), _
        Brushes.Black, New RectangleF(100, 80, 180, 250))
    G.DrawRectangle(Pens.Red, 100, 80, 180, 250)
    G.DrawString(txt, New Font("verdana", 12, FontStyle.Regular), _
        Brushes.Black, New RectangleF(350, 100, 400, 150))
    G.DrawRectangle(Pens.Red, 350, 100, 400, 150)
End Sub

Some of the overloaded forms of the DrawString method accept an argument of the StringFormat type. This argument determines characteristics of the text and exposes a few properties of its own, which include the following:

Alignment  Determines the alignment of the text. Its value is one of the members of the StringAlignment enumeration: Center (text is aligned in the center of the layout rectangle), Far (text is aligned far from the origin of the layout rectangle) and Near (text is aligned near the origin of the layout rectangle).

Trimming  Determines how text will be trimmed if it doesn’t fit in the layout rectangle. Its value is one of the members of the StringTrimming enumeration: Character (text is trimmed to the nearest character), EllipsisCharacter (trimmed to the nearest character and an ellipsis is inserted at the end to indicate that some of the text is missing), EllipsisPath (text at the middle of the string is removed and replaced by an ellipsis), EllipsisWord (trimmed to the nearest word and an ellipsis is inserted at the end), None (no trimming), and Word (trimmed to the nearest word).

FormatFlags  Specifies layout information for the string. Its value can be one of the members of the StringFormatFlags enumeration. The two members of this enumeration you may need often are DirectionRightToLeft (prints to the left of the specified point) and DirectionVertical.

To use the stringFormat argument of the DrawString method, instantiate a variable of this type, set the desired properties, and then pass it as argument to the DrawString method, as shown here:

Dim G As Graphics = Me.CreateGraphics
Dim SF As New StringFormat()
SF.FormatFlags = StringFormatFlags.DirectionVertical
G.DrawString("Visual Basic", Me.Font, Brushes.Red, 80, 80, SF)
The call to the DrawString method will print the string from top to bottom. It will also rotate the characters. The DirectionRightToLeft will print the string to the left of the specified point, but it will not mirror the characters. In effect, it shifts the string to the left of the point specified with the DrawString method, by the length of the string, and then prints it.

You can find additional examples of the MeasureString method in Chapter 15, where we’ll use this method to fit strings on the width of the page. The third button on the form of the TextEffect project draws text with a three-dimensional look by overlaying a semitransparent string over an opaque string. This technique is explained in the section “Alpha Blending,” later in this chapter, where you’ll learn how to use transparency. You may also wonder why none of the DrawString methods’ forms accept as argument an angle of rotation for the text. You can draw text, or any shape, at any orientation as long as you set up the proper rotation transformation. This topic is discussed in the section “Coordinate Transformations,” also later in this chapter.

**DRAWIMAGE**

The DrawImage method renders an image on the Graphics object, at a specified location. The DrawImage method is heavily overloaded and quite flexible. The following form of the method draws the image at its original magnification at the specified location. Both the image and the location of its top-left corner are passed to the method as arguments:

```csharp
Graphics.DrawImage(img, point)
```

*img* is an Image object, and *point* is a Point object that specifies the location of the image’s top-left corner on the drawing surface.

Another form of the method draws the specified image within the specified rectangle. If the rectangle doesn’t match the original dimensions of the image, the image will be resized to fit in the rectangle. The rectangle should have the same aspect ratio as the Image object, so that the image won’t be distorted in the process.

```csharp
Graphics.DrawImage(img, rectangle)
```

Another form of the method allows you to change not only the magnification of the image, but its shape as well. This method accepts as argument not a rectangle, but an array of three points that specify a parallelogram. The image will be sheared to fit in the parallelogram.

```csharp
Graphics.DrawImage(img, points())
```

where *points* is an array of points that define a parallelogram. The array holds three points, which are the top-left, top-right, and bottom-left corners of the parallelogram. The fourth point is determined uniquely by the other three, and you need not supply it.

The last important form of the method allows you to set the attributes of the image:

```csharp
Graphics.DrawImage(image, points(), srcRect, units, attributes)
```

The first two arguments are the same as in the previous version of the method. The *srcRect* argument is a rectangle that specifies the portion of image to draw, and *units* is a constant of the GraphicsUnit enumeration. It determines how the units of the rectangle are measured (pixels, inches, and so on). The last argument is an ImageAttributes object that contains information about the attributes of the image you want to change (such attributes include the gamma value, and a transparent color value, or
color key). The ImageAttributes object provides methods for setting image attributes, and they’re discussed shortly.

The DrawImage method is quite flexible, and you can use it for many special effects, including wipes. A *wipe* is the gradual appearance of an image on a form or PictureBox control. You can use this method to draw stripes of the original image, or start with a small rectangle in the middle and enlarge it until the entire image is covered.

You can also correct the color of the image by specifying the *attributes* argument. To specify the *attributes* argument, create an ImageAttributes object, with a statement like the following:

```
Dim attr As New System.Drawing.Imaging.ImageAttributes
```

The ImageAttributes object provides the following methods.

**SetWrapMode**
Specifies the wrap mode that is used to decide how to tile a texture across a shape. This attribute is used with textured brushes, and I don’t discuss it in this book.

**Set Gamma**
This method sets the gamma value for the image’s colors and accepts a Single value, which is the gamma value to be applied. A gamma value of 1 doesn’t affect the colors of the image. A smaller value darkens the colors, while a larger value makes the image colors brighter. Notice that the gamma correction isn’t the same as manipulating the brightness of the colors. The gamma correction takes into consideration the entire range of values in the image; it doesn’t apply equally to all the colors. In effect, it takes into consideration both the brightness and the contrast and corrects them in tandem, with a fairly complicated algorithm. The syntax of the SetGamma method is:

```
ImageAttributes.SetGamma(gamma)
```

The following statements render the image stored in the *img* Image object on the G Graphics object, and they gamma-correct the image in the process by a factor of 1.25:

```
Dim attrs As New System.Drawing.Imaging.ImageAttributes()
attrs.SetGamma(1.25)
Dim dest As New Rectangle(0, 0, PictureBox1.Width, PictureBox1.Height)
G.DrawImage(img, dest, 0, 0, img.Width, img.Height, GraphicsUnit.Pixel, attrs)
```

**SetOutputChannel**
If you plan to create high-quality printouts of your images, you must separate them into four different channels. Each channel represents a different color, but these colors aren’t red, green and blue. Typographers use four different basic colors, which are cyan, magenta, yellow, and black. The process of breaking an image into four channels is known as *color separation*, and you can separate your images with SetOutputChannel. Call this method four times, each time with a different channel. The syntax of the SetOutputChannel method is:

```
ImageAttributes.SetOutputChannel(colorChannel)
```
where the `colorChannel` argument can have one of the following values: `ColorChannelC` (cyan channel), `ColorChannelM` (magenta channel), `ColorChannelY` (yellow channel), `ColorChannelK` (black channel), and `ColorChannelLast` (the same channel as in the last time you called the method). The four channels produced by the `SetOutputChannel` method are monochrome (grayscale), and each one is printed with a different ink. All four channels, however, are printed on the same page, and the result is the original image’s colors.

**Gradients**
In this section we’ll look at the tools for creating gradients. The techniques for gradients can get quite complicated, but I will limit the discussion to the types of gradients you’ll need for business or simple graphics applications.

**Linear Gradients**
Let’s start with linear gradients. Like all other gradients, they’re part of the `System.Drawing` class and are implemented as brushes. To use a gradient, you must create the appropriate brush with the appropriate constructor. To draw a linear gradient, you must create a `LinearGradientBrush` with a statement like

```vbnet
Dim lgBrush As LinearGradientBrush
lgBrush = New LinearGradientBrush(rect, startColor, endColor, gradientMode)
```

To understand how to use the arguments, you must understand how the linear gradient works. This method creates a gradient that fills a rectangle, specified by the `rect` object passed as the first argument. This rectangle isn’t filled with any gradient; it simply tells the method how long (or how tall) the gradient should be. The gradient starts with the `startColor` at the left side of the rectangle and ends with the `endColor` at the opposite side. The gradient changes color slowly as it moves from one end to the other. The last argument, `gradientMode`, specifies the direction of the gradient and can have one of the values shown in Table 14.6.

<table>
<thead>
<tr>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackwardDiagonal</td>
<td>The gradient fills the rectangle diagonally, from the top-right corner (startColor) to the bottom-left corner (endColor).</td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>The gradient fills the rectangle diagonally, from the top-left corner (startColor) to the bottom-right corner (endColor).</td>
</tr>
<tr>
<td>Horizontal</td>
<td>The gradient fills the rectangle from left (startColor) to right (endColor).</td>
</tr>
<tr>
<td>Vertical</td>
<td>The gradient fills the rectangle from top (startColor) to bottom (endColor).</td>
</tr>
</tbody>
</table>
Notice that in the descriptions of the various modes, I stated that the gradient fills the rectangle, not the shape. The gradient is calculated according to the dimensions of the rectangle specified with the first argument. If the actual shape is smaller than this rectangle, only a section of the gradient will be used to fill the shape. If the shape is larger than this rectangle, the gradient will repeat as many times as necessary to fill the shape.

Let’s say you want to use the same gradient that extends 300 pixels horizontally to fill two rectangles, one that’s 200 pixels wide and another one that’s 600 pixels wide. We’ll fill this shape with two similar LinearGradientBrushes that differ only in the size of the rectangle specified with the first argument. The first brush will use a rectangle 200 pixels wide, filled with two thirds of the gradient; the second will use a rectangle 600 pixels wide and be filled with a gradient that’s repeated twice. The code in Listing 14.17 corresponds to the GDIPlusGradients project on the CD.

**Listing 14.17: Filling Rectangles with a Linear Gradient**

```vbscript
Private Sub LinearGradient_Click(ByVal sender As System.Object,
                               ByVal e As System.EventArgs) Handles btnLinearGradient.Click
    Dim G As Graphics
    G = Me.CreateGraphics
    Dim R As New RectangleF(20, 20, 300, 100)
    Dim startColor As Color = Color.BlueViolet
    Dim EndColor As Color = Color.LightYellow
    Dim LBrush As New System.Drawing.Drawing2D.LinearGradientBrush
        (R, startColor, EndColor, LinearGradientMode.Horizontal)
    G.FillRectangle(LBrush, New Rectangle(20, 20, 200, 100))
    G.FillRectangle(LBrush, New Rectangle(20, 150, 600, 100))
End Sub
```

For a horizontal gradient, only the width of the rectangle is used; the height is irrelevant. For a vertical gradient, only the height of the rectangle matters. When you draw a diagonal gradient, then both dimensions are taken into consideration.

You can also use a LinearGradientBrush to fill any shape, including closed polygons and closed curves. How does the brush handle irregular shapes? It doesn’t, really. It fills, with the specified gradient, a rectangle that completely encloses the shape, and it shows only the pixels that fall within the shape. It’s like building a larger gradient and looking at it through an irregularly shaped (nonrectangular) window.

You can create gradients at any direction by setting the `gradientMode` argument of the LinearGradientBrush object’s constructor. The Diagonal Linear Gradient button on the GDIPlusGradients project does exactly that.

The button Gradient Text on the form of the GDIPlusGradients project on the CD renders some text with a linear gradient. As you recall from our discussion of the DrawString method, strings are rendered with a Brush object, not a Pen object. If you specify a LinearGradientBrush object, the text will be rendered with a linear gradient. The text shown in Figure 14.17 was produced by the Gradient Text button, whose code is shown in Listing 14.18.
**Figure 14.17**
Drawing a string filled with a gradient

**Listing 14.18: Rendering Strings with a Linear Gradient**

```vbnet
Private Sub btnGradientText_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnGradientText.Click

    Dim G As Graphics
    G = Me.CreateGraphics
    G.Clear (me.BackColor)
    G.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias
    Dim largeFont As New Font('Comic Sans MS', 48,
                               FontStyle.Bold, GraphicsUnit.Point)
    Dim gradientStart As New Point(0, 0)
    Dim txt As String = 'Gradient Text'
    Dim txtSize As New SizeF()
    txtSize = G.MeasureString(txt, largeFont)
    Dim gradientEnd As New PointF()
    gradientEnd.X = txtSize.Width
    gradientEnd.Y = txtSize.Height
    Dim grBrush As New LinearGradientBrush(gradientStart, gradientEnd, _,
                                           Color.Yellow, Color.Blue)
    G.DrawString(txt, largeFont, grBrush, 20, 20)

    End Sub
```

The code of Listing 14.18 is a little longer than it could be (or than you might expect). Because linear gradients have a fixed size and don’t expand or shrink to fill the shape, you must call the MeasureString method to calculate the width of the string and then create a linear gradient with the exact same width. This way, the characters will be filled exactly with the specified gradient.

**Path Gradients**

This is the ultimate gradient tool. Using a PathGradientBrush, you can create a gradient that starts at a single point and fades into multiple different colors in different directions. You can fill a rectangle
starting from a point in the interior of the rectangle, which is colored, say, black. Each corner of the rectangle may have a different ending color. The PathGradientBrush will change color in the interior of the shape and will generate a gradient that’s smooth in all directions. Figure 14.18 shows a rectangle filled with a path gradient, but the gray shades on the printed page won’t show the full impact of the gradient. Open GDIPlusGradients project on the CD to see the same figure in color (button Path Gradient).

**Figure 14.18**
A path gradient starting at the middle of the rectangle.

To fill a shape with a path gradient, you must first create a path object. The PathBrush will be created for the specific path and can be used to fill this path—but not any other shape. Actually, you can fill any other shape with the PathBrush created for a specific path, but the gradient won’t fit the new shape. A path gradient must be applied only to the Path object for which it was created. To create a PathGradientBrush, use the following syntax:

```vbnet
dim pBrush as pathGradientBrush
pBrush = new LinearGradientBrush(path)
```

where *path* is a properly initialized Path object.

The *pBrush* object provides properties that determine the exact coloring of the gradient. First, you must specify color of the gradient at the center of the shape, using the CenterColor property. The SurroundColors property is an array, with as many elements as there are vertices (corners) in the Path object. Each element of the SurroundColors array must be set to a color value, and the resulting gradient will have the color of the equivalent element of the SurroundColors array.

The following declaration creates an array of three different colors and assigns them to the SurroundColors property of a *PathGradientBrush*:

```vbnet
dim colors() as color = {color.yellow, color.green, color.blue}
pBrush.SurroundColors = Colors
```

After setting the *PathGradientBrush*, you can fill the corresponding Path object by calling the FillPath method. The Path Gradient button on the form of GDIPlusGradient creates a rectangle filled with a gradient that’s red in the middle of the rectangle and has a different color at each corner. Listing 14.19 shows the code behind the Path Gradient button.
Listing 14.19: Filling a Rectangle with a Path Gradient

Private Sub btnPathGradient_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnPathGradient.Click
Dim G As Graphics
G = Me.CreateGraphics
Dim path As New System.Drawing.Drawing2D.GraphicsPath()
path.AddLine(New Point(10, 10), New Point(400, 10))
path.AddLine(New Point(400, 10), New Point(400, 250))
path.AddLine(New Point(400, 250), New Point(10, 250))
Dim pathBrush As New System.Drawing.Drawing2D.PathGradientBrush(path)
pathBrush.CenterColor = Color.Red
Dim surroundColors() As Color = _
{Color.Yellow, Color.Green, Color.Blue, Color.Cyan}
pathBrush.SurroundColors = surroundColors
G.FillPath(pathBrush, path)
End Sub

The gradient’s center point is, by default, the center of the shape. You can also specify the center of the gradient (the point that will be colored according to the CenterColor property). You can place the center point of the gradient anywhere by setting its CenterPoint property to a Point or PointF value.

The GDIPlusGradients application has a few more buttons that create interesting gradients, which you can examine on your own. The Rectangle Gradient button fills a rectangle with a gradient that has a single ending color all around. All the elements of the SurroundColors property are set to the same color. The Animated Gradient animates the same gradient by changing the coordinates of the PathGradientBrush object’s CenterPoint property.

Clipping

Anyone who has used drawing or image-processing applications already knows that many of the tools of a similar application make use of masks. A mask is any shape that limits the area in which you can draw. If you want to place a star or heart on an image and print something in it, you create the shape in which you want to limit your drawing tools, then convert this shape into a mask. When you draw with the mask, you can start and end your strokes anywhere on the image. Your actions will have no effect outside the mask, however.

The mask of the various image-processing applications is a clipping region. A clipping region can be anything, as long as it’s a closed shape. While the clipping region is activated, drawing takes place in the area of the clipping region. To specify a clipping area, you must call the SetClip method of the Graphics object. The SetClip method accepts the clipping area as argument, and the clipping area can be the Graphics object itself (no clipping), a Rectangle, a Path, or a Region.

NOTE A Region is a structure made up of simple shapes. There many methods to create a Region object—you can combine and intersect shapes, or exclude shapes from a region—but we aren’t going to discuss the Region object in this chapter, because it’s not among the common objects we use to generate the type of graphics discussed in the context of this book.
The SetClip method has the following forms:

Graphics.SetClip(Graphics)
Graphics.SetClip(Rectangle)
Graphics.SetClip(GraphicsPath)
Graphics.SetClip(Region)

All methods accept a second optional argument, which determines how the new clipping area will be combined with the existing one. The second argument is the combineMode argument, and its value can be one of the members of the CombineMode enumeration: Complement, Exclude, Intersect, Replace, Union, and XOR.

Once a clipping area has been set for the Graphics object, drawing is limited to that area. You can specify any coordinates, but only the part of the drawing that falls inside the clipping area is visible. The clipping project demonstrates how to clip text and images within an elliptical area (see Figure 14.19). The button Boxed Text draws a string in a rectangle. The button Clipped Text draws the same text but first applies a clipping area. The clipping area is an ellipse. The Clipped Image button uses the same rectangle to clip an image. Since there’s no form of the SetClip method that accepts an ellipse as argument, we must construct a Path object, add the ellipse to the path, and then create a clipping area based on the path.

**Figure 14.19**
Clipping text and images in an ellipse

The following statements create the clipping area for the text, which is an ellipse. The path is created by calling the AddEllipse method of the GraphicsPath object. This path is then passed as argument to the Graphics object’s SetClip method:

```
Dim P As New System.Drawing.Drawing2D.GraphicsPath()
Dim clipRect As New RectangleF(30, 30, 250, 150)
P.AddEllipse(clipRect)
Dim G As Graphics
G = PictureBox1.CreateGraphics
G.SetClip(P)
```

The listing behind the Clipped Text and Clipped Image buttons is shown next. The first button prints some text in a rectangular area that is centered over the clipping area. The first button, Boxed Text, shows how the text is printed within the rectangle. The same rectangle and its text are then...
printed at a different location, right behind the clipping area. Both the rectangle and the ellipse are based on the same Rectangle object. Listing 14.20 shows the code behind the Boxed Text and the Clipped Text buttons.

**Listing 14.20: The Boxed Text and Clipped Text Buttons**

```vbnet
Private Sub btnBoxedText_Click(ByVal sender As System.Object,
                              ByVal e As System.EventArgs) Handles btnBoxedText.Click
    Dim Rect As New RectangleF(30, 30, 250, 150)
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    Dim format As StringFormat = New StringFormat()
    format.Alignment = StringAlignment.Center
    G.DrawString(txt & txt, New Font("Verdana", 11, FontStyle.Regular), _
                  Brushes.Coral, Rect, format)
End Sub
Private Sub btnClippedText_Click(ByVal sender As System.Object,
                                  ByVal e As System.EventArgs) Handles btnClippedText.Click
    Dim P As New System.Drawing.Drawing2D.GraphicsPath()
    Dim clipRect As New RectangleF(30, 30, 250, 150)
    P.AddEllipse(clipRect)
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.DrawEllipse(Pens.Red, clipRect)
    G.Clip(P)
    Dim format As StringFormat = New StringFormat()
    format.Alignment = StringAlignment.Center
    G.DrawString(txt & txt, New Font("Verdana", 11, FontStyle.Regular), _
                  Brushes.Coral, clipRect, format)
End Sub
```

The difference between the two subroutines is that the second sets an ellipse as the clipping area and draws the same ellipse. Because the Graphics object has a clipping area, anything we draw on it is automatically clipped.

The Clipped Image button sets up a similar clipping area and then draws an image centered behind the clipping ellipse. As you saw in Figure 14.19, only the segment of the image that’s inside the clipping area is visible. The code behind the Clipped Image button is shown in Listing 14.21.

**Listing 14.21: The Clipped Image Button**

```vbnet
Private Sub btnClippedImage_Click(ByVal sender As System.Object,
                                  ByVal e As System.EventArgs) Handles btnClippedImage.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.TranslateTransform(200, 200)
```
Coordinate Transformations

So far, we’ve been specifying our coordinates in pixels. This is a convenient coordinate system for drawing simple shapes and experimenting with the various drawing methods, but in real applications you need a more familiar method of specifying coordinates. No physical object’s dimensions are declared in pixels; we measure objects in inches, meters, or even miles. Other objects are measured in millionths of a centimeter. Since pixels are the natural units for displays, we must map the actual units to pixels. In this section, we’ll look at techniques for mapping real-world coordinates to pixels.

The coordinate system is similar to a city map. Each square on the map has its own unique address: a combination of a column and a row number. The row number is the vertical coordinate, or y coordinate. The column number is the horizontal coordinate, or x coordinate. Any point on the form can be identified by its x and y coordinates, and we refer to it as the point at coordinates (x, y) or simply the point (x, y).

The point with the smallest coordinates is the origin of the coordinate system. The origin of a coordinate system is the point (0, 0), and it’s the top-left point of the Graphics object. The x coordinates increase to the right, and the y coordinates increase downward. Each coordinate is a number that may or may not correspond to a meaningful unit. For example, the letter and number coordinates on a city map don’t correspond to meaningful units; they are arbitrary. The coordinates on a topological map, though, correspond to physical distances (e.g., kilometers or miles). The physical interpretation of the coordinates depends on the intended application.

If you want to draw a plan for your new house, you need to use a coordinate system in inches or centimeters so that there will be some relation between units and the objects you draw. If you’re going to draw some nice geometrical shapes, any coordinate system will do. Finally, if you’re going to display and process images, you’ll want a coordinate system that uses pixels as units. Actually, pixels aren’t the best units for any application, except for image-processing applications.

GDI+ lets you define your own coordinate system. All you have to do is set the PageUnit property to the appropriate constant. If you set PageUnit property to Inches, the dimensions of your shapes, as well as their locations on the Graphics object, must be specified in inches. In this case, two points that are one unit apart are one inch from each other. You can also specify decimal distances such as 0.1, which corresponds to 1/10 of an inch. Changing the PageUnit property doesn’t resize or otherwise affect the form or the printer’s page. It simply changes the density of the grid you use to address the points on the control. The benefit of using the PageUnit property is that you don’t have to map your coordinates to pixels on the monitor or to dots on the printed page. GDI+ knows how many pixels are in an inch on the monitor, and it will scale the coordinates accordingly. A statement
that draws a rectangle one inch tall and three inches wide will produce the correct shape on any
monitor or printer, regardless of its resolution.

If you want to know the density of the pixels on the current monitor, read the DpiX and DpiY
properties of the Graphics object. These properties return the pixels per inch in x and y directions of
the monitor. If the Graphics object is the printer’s page, they will return the dots per inch for the
specific printer. These properties are determined by the device’s capabilities and are read-only.

The PageUnit property affects all the entities drawn on the control, even the width of the Pen. A
Pen with a width of 1 will draw lines one inch wide. To specify pen widths in pixels, use the inverse
of the property DpiX, which is one pixel. To specify a 2-pixel-wide pen, use a statement like this one:

```
Dim myPen = New Pen(Color.Black, 2 * (1 / Graphics.DpiX))
```

Using coordinates that correspond to physical units of length—such as inches, points, and millimeters—is straightforward. The most interesting, and flexible, coordinate system is one that suits
your needs—in other words, a custom coordinate system. A custom coordinate system is dictated by the
application and can be anything, from a fraction of a millimeter, to a mile, or a light year. Custom
coordinate systems need not correspond to length units. When you plot the number of users hitting
your site each hour, you need a coordinate system to represent the hour of the day along the
x axis and the number of visitors (or the number of hits, or any other quantity you care to measure)
along the y axis. This coordinate system goes from 1 to 24 along the horizontal axis and from 0 to a
large value like 100, or 10,000, in the vertical axis. Another chart might involve month numbers and
units sold. If you want to draw the trajectory of the earth around the sun, and the sun is a circle at
the middle of the drawing surface, the coordinates along the two axes must go from a very large nega-
tive value to a very large positive value.

To summarize, your starting point is a coordinate system that represents the physical dimensions
of the entity you want to plot on the form. This is the world coordinate system. The dimensions of the
form are also known: they’re the form’s Width and Height properties. This is the page coordinate system.
The world coordinates must be mapped onto page coordinates, and this mapping is known as a
transformation.

**Specifying Transformations**

In computer graphics, there are three types of transformations: scaling, translation, and rotation. The
scaling transformation changes the dimensions of a shape but not its form. If you scale an ellipse by
0.5, you’ll get another ellipse that’s half as wide and half as tall as the original one. The translation
transformation moves a shape by a specified distance. If you translate a rectangle by 30 pixels along
the x axis and 90 pixels along the y axis, the new origin will be 30 pixels the right and 90 pixels down
from the original rectangle’s top-left corner. The rotation transformation rotates a shape by a speci-
fied angle, expressed in degrees. 360 degrees correspond to a full rotation, and the shape appears the
same. A rotation by 180 degrees is equivalent to flipping the shape vertically and horizontally.

Transformations are stored in a 5×5 matrix, but you need not set it up yourself. The Graphic
object provides the ScaleTransform, TranslateTransform, and RotateTransform methods, and you
can specify the transformation to be applied to the shape by calling one or more of these methods
and passing the appropriate argument(s). The ScaleTransform accepts as arguments scaling factors
for the horizontal and vertical directions:

```
Graphics.ScaleTransform(Sx, Sy)
```
If an argument is smaller than one, the shape will be reduced in the corresponding direction; if it's larger than one, the shape will be enlarged in the corresponding direction. We usually scale both directions by the same factor to retain the shape's aspect ratio. If you scale a circle by different factors in the two dimensions, the result will be an ellipse, and not a smaller or larger circle.

The TranslateTransform method accepts two arguments, which are the displacements along the horizontal and vertical directions:

```
Graphics.TranslateTransform(Tx, Ty)
```

The `Tx` and `Ty` arguments are expressed in the coordinates of the current coordinate system. The shape is moved to the right by `Tx` units and down by `Ty` units. If one of the arguments is negative, the shape is moved in the opposite direction (to the left or up).

The RotateTransform method accepts a single argument, which is the angle of rotation, and it's expressed in degrees:

```
Graphics.RotateTransform(rotation)
```

If the `rotation` argument is 360, the shape is rotated a full circle—no change at all. The rotation takes place about the origin. As you will see, the final position and orientation of a shape is different if two identical rotation and translation transformations are applied in different order.

Every time you call one of these methods, the elements of the transformation matrix are set accordingly. All transformations are stored in this matrix, and they have a cumulative effect. If you specify two translation transformations, for example, the shape will be translated by the sum of the corresponding arguments in either direction. The following two transformations:

```
Graphics.TranslateTransform(10, 40)
Graphics.TranslateTransform(20, 20)
```

are equivalent to the following one:

```
Graphics.TranslateTransform(30, 60)
```

To start a new transformation after drawing some shapes on the Graphics object, call the ResetTransform method, which clears the transformation matrix.

The effect of multiple transformations may be cumulative, but the order in which transformations are performed makes a big difference. The GDIPlusTransformations project allows you to experiment with the various transformations. The shape being transformed is a rectangle that contains a string and a small bitmap, as shown in Figure 14.20. Each button on the right performs a different transformation or combination of transformations. The code is quite short, and you can easily add additional transformations, or change their order, and see how the shape is transformed. Keep in mind that some transformations may bring the shape entirely outside the form. In this case, just apply a translation transformation in the opposite direction.

The code behind the buttons Translate, Rotate, and Scale is shown in Listing 14.22. The buttons set the appropriate transformations and then call the DrawShape() subroutine, passing the current Graphic object as argument:
**Listing 14.22: The Buttons of the GDI+Transform Project**

```vbnet
Private Sub btnTranslate_Click(ByVal sender As System.Object,
                               ByVal e As System.EventArgs) Handles btnTranslate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.TranslateTransform(200, 90)
    DrawShape(G)
End Sub

Private Sub btnRotate_Click(ByVal sender As System.Object,
                             ByVal e As System.EventArgs) Handles btnRotate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.RotateTransform(45)
    DrawShape(G)
End Sub

Private Sub btnTranslateRotate_Click(ByVal sender As System.Object,
                                      ByVal e As System.EventArgs) Handles btnTranslateRotate.Click
    Dim G As Graphics = PictureBox1.CreateGraphics
    G.TranslateTransform(200, 90)
    G.RotateTransform(45)
    DrawShape(G)
End Sub
```

The `DrawShape()` subroutine, which draws the rectangle, the string, and the bitmap, is called by all the buttons after setting up the appropriate transformation(s). Listing 14.23 shows the `DrawShape()` subroutine.
LISTING 14.23: THE DRAWSHAPE() SUBROUTINE

Sub DrawShape(ByVal GraphicObject As Graphics)
    Dim Font As Font = New Font("Comic Sans MS", 36, FontStyle.Bold, GraphicsUnit.Pixel)
    Dim Pen As Pen = New Pen(Color.Red, 2)
    GraphicObject.DrawRectangle(Pen, New Rectangle(1, 1, 200, 120))
    GraphicObject.DrawRectangle(Pen, New Rectangle(1, 1, 200, 120))
    GraphicObject.DrawString("VB.NET", Font, Brushes.Violet, 25, 5)
End Sub

The reason we pass the Graphics object as argument to the DrawShape() subroutine (as opposed to creating the appropriate Graphics object in the subroutine’s code) is because the transformations we defined earlier apply to this object. By passing the G argument to the DrawShape() subroutine, we’re actually passing all the transformations applied to the Graphics object. The shapes drawn on the Graphics objects by the code in the DrawShape() subroutine will undergo the specified transformation.

Run the GDI+Transformations project and examine its code. You can add more buttons with your own transformations to the form, or change the parameters of the various transformations. Notice how the order of the various transformations affects the placement, orientation, and size of the final image.

In the following section, we’ll look at an interesting example of the DrawImage method combined with coordinate transformations. The example you’ll build in the following section will render a cube with different images plastered on each side of the cube. Later in this chapter we’ll use transformations to plot a function, in an interesting and practical application. Even if you’re not interested in graphics or math, you should understand how to use the basic transformations, because in the following chapter we’ll use them to create printouts.

VB.NET AT WORK: THE IMAGE_CUBE PROJECT

As you recall, the DrawImage method can render images on any parallelogram, not just a rectangle, with the necessary distortion. A way to look at these images is not as distorted, but as perspective images. Looking at a printout from an unusual angle is equivalent to rendering an image within a parallelogram. Imagine a cube with a different image glued on each side. To display such a cube on your monitor, you must calculate the coordinates of the cube’s edges and then use these coordinates to define the parallelograms on which each image will be displayed. Figure 14.21 shows a cube with a different image on each side.

If you’re good at math, you can rotate a cube around its vertical and horizontal axes and then map the rotated cube on the drawing surface. You can even apply a perspective transformation, which will make the image look more like the rendering of a three-dimensional cube. This is more involved than the topics discussed in this book; actually, it’s a good topic for a book on 3D graphics, but not for a general programming book. Instead of doing all the calculations, I’ve come up with a set of coordinates for the parallelogram that represents each vertex (corner) of the cube. For a different orientation, you can draw a perspective view of a cube on paper and measure the coordinates of its vertices. Once you
can define the parallelogram that corresponds to each visible side, you can draw an image with the DrawImage method on each parallelogram. The DrawImage method will shear the image as necessary to fill the specified area. The result is the representation of a cube covered with images on the flat surface of your monitor.

**Figure 14.21**
This cube was created with a call to the DrawImage method for each visible side.

The ImageCube project on the CD does exactly that. It sets up the coordinates of the vertices of a cube projected onto a two-dimensional drawing surface. Then, it calls the DrawImage method once for each side of the cube, passing as arguments the image to be rendered on the corresponding side of the cube and an array with the coordinates of the three out of the four corners of the side.

The code that produced Figure 14.21 starts by setting up the coordinates of each side. Notice that we only need the coordinates of three points to identify each parallelogram, and the arrays `face1`, `face2`, and `face3` correspond to the three visible faces of the cube. Each of these arrays contains three elements, all of the Point type, which are the coordinates of the three vertices of the corresponding side of the cube. Then the DrawImage method is called for each face, with the appropriate coordinates and a different image. The complete listing behind the Draw Cube button is shown in Listing 14.24. I'm using the random-number generator to randomly assign images to each of the cube's side, so that each time you click the Draw Cube button, a new cube is drawn. To keep the code simple, I'm not checking each random number to make sure a different image is mapped to each side of the cube, but you can add the necessary logic to avoid reusing the same image for more than one side.

**Listing 14.24: Rendering the ImageCube**

```vba
Private Sub btnnDrawCube_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnnDrawCube.Click
    Dim G As Graphics
    G = GetGraphicsObject()
```

Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com
Clear(Color.SandyBrown)
G.TranslateTransform(250, 200)

Dim images(2) As Image
Dim path As String
path = Application.StartupPath.Remove(Application.StartupPath.Length - 4, 4)
images(0) = Image.FromFile(path & "\image1.jpg")
images(1) = Image.FromFile(path & "\image2.jpg")
images(2) = Image.FromFile(path & "\image3.jpg")

Dim face1(2) As Point
face1(0) = New Point(-150, -20)
face1(1) = New Point(150, -20)
face1(2) = New Point(-150, 230)

Dim face2(2) As Point
face2(0) = New Point(-30, -140)
face2(1) = New Point(270, -140)
face2(2) = New Point(-150, -20)

Dim face3(2) As Point
face3(0) = New Point(150, -20)
face3(1) = New Point(270, -140)
face3(2) = New Point(150, 230)

Dim imgIndex As Integer
Dim rnd As New System.Random()
imgIndex = rnd.Next(0, 3)
G.DrawImage(images(imgIndex), face1)
imgIndex = rnd.Next(0, 3)
G.DrawImage(images(imgIndex), face2)
imgIndex = rnd.Next(0, 3)
G.DrawImage(images(imgIndex), face3)

End Sub

The GetGraphicsObject() function retrieves a Graphics object from the Picturebox control's bitmap, and its code was shown earlier in Listing 14.8. This is necessary if you want to draw persistently, instead of having to redraw the cube from within the OnPaint event. If you want to use the OnPaint method, you must also store the images (or the random numbers that correspond to the three images) in global variables; otherwise every time the OnPaint event handler is executed, a new cube will be displayed.

Notice that the cube is (approximately) centered at the origin of the form, so I've introduced a translation transformation to push the cube toward the middle of the form. You can add a rotation transformation too, but you may have to modify the translation transformation, should the image end up partially outside the form.
### VB.NET at Work: Plotting Functions

Many programmers will use graphics to plot functions or user-supplied data sets. A plot is a visual representation of the values of a function over a range of an independent variable. Figure 14.22 shows the following function plotted against time in the range from –0.5 to 5:

$$10 + 35 \sin(2 \pi X) + \sin(0.80 / X)$$

**Figure 14.22**

Plotting math functions with VB.NET

The plot of Figure 14.22 was created with the Plotting project, which is described in this section. The variable $x$ represents time and goes from –0.5 to 5. The time is mapped to the horizontal axis, and the vertical axis is the magnitude of the function. For each pixel along the horizontal axis, we calculate the value of the function and turn on the pixel that corresponds to the calculated value.

Functions can be plotted in very small, or quite large, ranges. The same is true for their values. One function may extend from –2 to –1, while another function may extend from 0 to 10,000. Obviously, we can’t use pixels as our units because most plots will not even fall in the range covered by the pixels of the bitmap. Somehow, we must map the function values to pixels and scale them, so that the function we’re plotting fills the available area. This mapping takes place through two transformations: a scaling transformation and a translation transformation.

A scaling transformation changes the size of a shape or curve. If the curve extends vertically from –1 to 5 and the area on which you’re plotting goes from 0 to 399 pixels, you must scale the plot up so that it fills the available area. If the curve extends from –1,000 to 500 vertically, you must also scale it to fill the available area, only this time you must reduce the size of the plot. In the first example, you must fit 6 units along the $x$ axis to 400 units. The scaling factor is 400 / 6, or 67 approximately. If the function extends vertically from 1 to 3 and the control’s drawing surface’s height is 250 pixels, the scaling factor along the vertical axis is 250 / 2, or 125.

The scaling will make the area of the plot equal to the drawing area. But the plot may not fall in the range of pixels on the control. If you’re plotting the function in the range from –10 to 0, the curve falls to the left of the drawing surface. The proper translation transformation must push all values to the right so that the smallest value (–10) corresponds to the leftmost pixel on the control and the largest value corresponds to the rightmost pixels on the control. Likewise, you must translate the plot vertically to bring it within the control’s visible area.
GDI+ allows you to define global transformations, which apply to all drawing actions. The global transformations apply to all graphics objects, including the pen you’re drawing with, so you can’t use them to map the function values onto pixels. Another type of transformation applies to individual shapes, and this is what we want. We want to transform the curve, but not the pen we draw with.

To plot a function on a PictureBox, you can either color individual pixels, or draw line segments between consecutive points of the function. In this example, we’ll use a path to represent the plot of a function. We’ll calculate the function along the values of the independent variable, and we’ll form a point defined by its two coordinates, \( x \) and \( y \). \( x \) is the value of the independent variable and \( y \) is the function’s value at this point. Once we have the path, we’ll apply a transformation to the entire path and render it with a Pen object on the drawing surface.

One advantage of this approach is that you can store the Path object that represents the plot to a global variable and reuse it to redraw the plot, or even apply a new transformation and plot the function at a different scale or location. If you discard the path after rendering it, you must recalculate all the points of the function along the \( x \) axis. The sample project doesn’t reuse the path, but you may have to do so in your applications.

The quantity represented on the horizontal axis is the independent variable. You can select any range to plot by setting a minimum and a maximum value for the \( x \) variable. The function must be evaluated along the horizontal axis, and it’s determined by the current value of the independent variable. When you’re plotting a function like \( \sin(x) \cos(x) \), you must evaluate the function for each point in the range of values of the independent variable \( x \). You may choose to plot the function in the range from \(-1\) to \(1\), or in the range from \(-1,000\) to \(1,000\).

Depending on the range of the independent variable, the function’s range may also change. In the range \(-1\) to \(1\), the function may extend from \(-0.5\) to \(2.5\), but in a different range, it may extend over a much larger range of values. The range of values of the function determines the units along the \( y \) axis. The vertical size of the plot doesn’t change; it’s the height of the control on which you’re drawing. However, you must map the physical values of the function you’re plotting to pixel coordinates.

Let me explain how the code of the Plotting application sets up the appropriate transformation. The function’s plot extends over a user-specified area in the horizontal direction (from \(-0.5\) to \(5\) in our case) and over a different range in the vertical axis. The vertical range is calculated by the program—unless you want to specify the vertical range as well. First, we must scale the plot so that the range from \(-0.5\) to \(5\) is mapped to the width of the PictureBox. The function is plotted over a range of \(5.5\) units horizontally, and they must be mapped to \(\text{PictureBox}.Width\) pixels. The function must be scaled horizontally by the factor \(\text{PictureBox}.Width/5.5\). The vertical scaling factor is calculated in a similar manner. You must first iterate through all the function values you’re going to plot and find out the smallest and largest values. Let’s say the function extends vertically from \(-4\) to \(20\). The vertical scaling factor is \(\text{PictureBox.Height}/24\).

The origin of the drawing area is at the top-left corner of the form or control. The origin of the plot, the point \((0,0)\), is at a different location, so we must translate the drawing. The physical coordinate \(-0.5\) must be mapped to a pixel with a \( x \) coordinate \(0\). The translation transformation along the horizontal axis is the negative of the smallest \( x \) value. In our example, it is \((-0.5)\), or \(0.5\).

Let’s see how the two transformations are combined. The first endpoint’s \( x \) coordinate is \(-0.5\). I will assume that the width of the PictureBox is \(500\) pixels. The first point must be translated horizontally by \(0.5\) and then scaled horizontally by \(500/5.5\):

\[
(-0.5 + 0.5) \times 500 / 5.5 = 0
\]
So the first endpoint’s x coordinate is 0. The last point’s x coordinate is:

\[(5 + 0.5) \times 500 / 5.5 = 500\]

The middle physical coordinate along the horizontal axis is 2.25, and it will be mapped to the following pixel coordinate:

\[(2.25 + 0.5) \times 500 / 5.5 = 250\]

which is the middle point of the horizontal axis. The exact same calculation will be performed for the vertical axis. The translation and scaling transformations map the box that encloses the function to the corners of the PictureBox control.

**NOTE** As you may have noticed, the translation transformation is applied first, and then the scaling transformation. If you reverse the order of the transformations, the result won’t be the same. If you first scale the coordinate 1 and then translate it, it will end up at the following x coordinate: \[1 \times 500 / 5.5 + 0.5 = 91.5\], which isn’t the coordinate of the middle point along the x axis.

To specify the transformations we want to perform on each point, we set up a transformation matrix, the World transformation matrix:

```csharp
World = New System.Drawing.Drawing2D.Matrix()
```

Then, we apply the two transformations:

```csharp
World.Scale(((PictureBox1.Width - 4) / (Xmax - Xmin)), _
-(PictureBox1.Height - 4) / (Ymax - Ymin))
World.Translate(-Xmin, -Ymax)
```

(We subtract 4 pixels from the control’s dimensions to make up for the PictureBox control’s border and leave a tiny margin between the extremes of the plot and the border.) At this point, the World transformation contains the definitions of the required transformations, and it will apply them to each point drawn on the Graphics object.

Now that you have seen how the function’s points are mapped to pixels, let’s look at the actual code. First, we calculate the range of y values in the specified range of x values. Since we can’t address sub-pixels, it makes no sense to calculate the function value at more points than there are pixels along the x axis; so the value is calculated at as many points as there are pixels. This is done in a loop, and we keep track of the minimum and maximum values. The same calculations must be repeated for all the functions to be plotted and then we keep the minimum and maximum value over all functions. These values are stored in YMin and YMax variables, and they’re used later in the code to set up the appropriate transformation. If the function returns a nonnumeric value (a value like NaN or Infinity), the program aborts.

Then, the program sets up the scaling and translation transformations as explained already. The transformations are stored in the World matrix, which is then used to apply them to a Path object.

In the last section, the code builds a path for the function to be plotted by adding line segments that connect the last point to the current one. The last step is to draw the path, when it’s complete. Before drawing the path, we apply the World transformation to it, so that the world coordinates will be mapped correctly to pixel coordinates:

```csharp
plot1.Transform(World)
G.DrawPath(plotPen, plot1)
```
where \( plot() \) is the name of the Path object that contains the points to be plotted. Listing 14.25 is the complete code of the \( Plot() \) subroutine, which is called from within the Plot button’s Click event handler.

**Listing 14.25: Plotting Functions**

```vba
Private Sub pPlot()
    If Not (CheckBox1.Checked Or CheckBox2.Checked) Then Exit Sub
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.Clear(PictureBox1.BackColor)

    Dim t As Double
    Dim Xmin, Xmax As Single
    Dim Ymax, Ymin As Single
    Ymin = System.Single.MaxValue
    Ymax = -System.Single.MaxValue
    Xmin = txtXMin.Text
    Xmax = txtXMax.Text
    Dim val As Single
    Dim XPixels As Integer = PictureBox1.Width - 1

    For t = Xmin To Xmax Step (Xmax - Xmin) / (PictureBox1.Width - 2)
        If CheckBox1.Checked Then
            val = Function1Eval(t)
            If System.Double.IsInfinity(val) Or 
                System.Double.IsNaN(val) Then
                MsgBox("Can't plot this function in " & _
                "the specified range!")
            Exit Sub
        End If
        Ymax = Math.Max(val, Ymax)
        Ymin = Math.Min(val, Ymin)
    End If
    If CheckBox2.Checked Then
        val = Function2Eval(t)
        If System.Double.IsInfinity(val) Or 
            System.Double.IsNaN(val) Then
            MsgBox("Can't plot this function in " & _
            "the specified range!")
        Exit Sub
    End If
    Ymax = Math.Max(val, Ymax)
    Ymin = Math.Min(val, Ymin)
    End If
    Next
    World = New System.Drawing.Drawing2D.Matrix()
```

*Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com*
World.Scale(((PictureBox1.Width - 2) / (Xmax - Xmin)), - _
            ((PictureBox1.Height - 2) / (Ymax - Ymin)))
        World.Translate(-Xmin, -Ymax)
' the following paths correspond to the two axes
Dim Xaxis As New System.Drawing.Drawing2D.GraphicsPath()
Dim Yaxis As New System.Drawing.Drawing2D.GraphicsPath()
Xaxis.AddLine(New PointF(Xmin, 0), New PointF(Xmax, 0))
Yaxis.AddLine(New PointF(0, Ymax), New PointF(0, Ymin))
Dim oldX, oldY As Single
Dim X, Y As Single
' Each segment in the path goes from (oldX, oldY) to (X, Y)
' At each iteration (X, Y) becomes (oldX, oldY) for the next point
' the following two paths correspond to the functions to be plotted
Dim plot1 As New System.Drawing.Drawing2D.GraphicsPath()
Dim plot2 As New System.Drawing.Drawing2D.GraphicsPath()
Dim plotPen As Pen = New Pen(Color.BlueViolet, 1)
' Calculate the min and max points of the plot
If CheckBox1.Checked Then
    oldX = Xmin
    oldY = Function1Eval(Xmin)
    For t = Xmin To Xmax Step (Xmax - Xmin) / _
        (PictureBox1.Width - 1)
        X = t
        Y = Function1Eval(t)
        plot1.AddLine(oldX, oldY, X, Y)
        oldX = X
        oldY = Y
    Next
End If
If CheckBox2.Checked Then
    oldX = Xmin
    oldY = Function2Eval(Xmin)
    For t = Xmin To Xmax Step (Xmax - Xmin) / _
        (PictureBox1.Width - 1)
        X = t
        Y = Function2Eval(t)
        plot2.AddLine(oldX, oldY, X, Y)
        oldX = X
        oldY = Y
    Next
End If
' create the plot1 and plot2 paths
G.Clear(PictureBox1.BackColor)
If RadioButton1.Checked Then SmoothingMode = _
    Drawing.Drawing2D.SmoothingMode.AntiAlias
If RadioButton2.Checked Then SmoothingMode = _
    Drawing.Drawing2D.SmoothingMode.Default
If RadioButton3.Checked Then SmoothingMode = _
```vbc
    Drawing.Drawing2D.SmoothingMode.HighQuality
    If RadioButton4.Checked Then SmoothingMode =
        Drawing.Drawing2D.SmoothingMode.HighSpeed
    G.SmoothingMode = SmoothingMode
    ' and finally draw everything
    Xaxis.Transform(World)
    plotPen.Color = Color.Red
    G.DrawPath(plotPen, Xaxis) ' The X axis
    Yaxis.Transform(World)
    plotPen.Color = Color.Red
    G.DrawPath(plotPen, Yaxis) ' The Y axis
    If CheckBox1.Checked Then
        plotPen.Color = Color.DarkMagenta
        plot1.Transform(World) ' The first function
        G.DrawPath(plotPen, plot1)
    End If
    If CheckBox2.Checked Then
        plotPen.Color = Color.DarkGreen
        plot2.Transform(World)
        G.DrawPath(plotPen, plot2) ' The second function
    End If
End Sub
```

Notice that the `Plot()` subroutine draws on the Graphics object returned by the CreateGraphics method of the PictureBox control. You can click the Plot button to redraw the function(s) at any time. You can draw the function on the control's bitmap and not have to worry about redrawing the function when the form is covered by another form. You should also call the `Plot()` subroutine from within the form's Resize event to redraw the function when the form is resized, because the PictureBox control is anchored on all four sides of the form.

The functions `Function1Eval()` and `Function2Eval()` calculate the value of each function for any value of the independent variable, and their implementation is shown here:

```vbc
    Function Function1Eval(ByVal X As Double) As Double
        Function1Eval = 5 + 20 * Cos(X * 3) * Cos(X * 5) / Sin(X / 3)
    End Function
    Function Function2Eval(ByVal X As Double) As Double
        Function2Eval = 10 * 35 * Sin(2 * X) * Sin(0.8 / X)
    End Function
```

The `FunctionPlot` application plots always the same two functions. You can specify a different range, but this doesn't make the application any more flexible. Allowing users to specify their own functions to plot isn't trivial, unless you use the Script ActiveX control. This control can evaluate any expression written in VBScript (a variation of VB), and you can bring this functionality in your application by adding an instance of the Script control to your form.

To add an instance of the Script control to your project, select Project ➤ Add Reference. When the Add Reference dialog box appears, click the COM tab and locate the item Microsoft Script

---

Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com

You created this PDF from an application that is not licensed to print to novaPDF printer (http://www.novapdf.com)
Control 1.0. If the control isn’t there, you must download from http://msdn.microsoft.com/scripting and install it on your computer. Once installed, it will appear in the Add Reference dialog box and you can add it to your project.

Using the Script control to evaluate a function is quite simple. All you have to do to evaluate a function for a value of its independent variable is to execute a script like the following:

\[
X = 0.04 \\
Y = 5 + 20 \times \cos(X \times 3) \times \cos(X \times 5) / \log(\text{Abs}(\sin(X))) / 10
\]

You can execute these statements with the help of the Script control and retrieve the value of the function for the specified value of the X variable. Revise the Function1Eval() and Function2Eval() functions as shown in Listing 14.26.

### Listing 14.26: Evaluating Arbitrary Math Expressions at Runtime

```vbs
Function Function1Eval(ByVal X As Double) As Double
    Try
        AxScriptControl1.ExecuteStatement("X=' & X)
        Function1Eval = AxScriptControl1.Eval(txtFunction1.Text)
    Catch exc As Exception
        Throw New Exception("Can't evaluate function at X=' & X")
    End Try
End Function

Function Function2Eval(ByVal X As Double) As Double
    Try
        AxScriptControl1.ExecuteStatement("X=' & X)
        Function2Eval = AxScriptControl1.Eval(txtFunction2.Text)
    Catch exc As Exception
        Throw New Exception("Can't evaluate function at X=' & X")
    End Try
End Function
```

If an error occurs in the script (division by zero, an attempt to calculate the logarithm of a negative value, and so on), a custom exception is raised. This exception must be caught by the Plot() subroutine and abort the plotting of the function.

The revised function is called FunctionPlotting, and you will find it in this chapter’s folder on the CD. The only difference is how the function is evaluated at each point along the x axis and how the program handles math errors. The Script control is an old one and can’t handle values like NaN and Infinity.

### Bitmaps

The type of graphics we explored so far are called vector graphics, because they can be scaled to any extent. Since they’re based on mathematical equations, you can draw any details of the picture without losing any accuracy. Vector graphics, however, can’t be used to describe the type of images you
capture with your digital camera. These images belong to a different category of graphics, the *bitmap graphics*. A bitmap is a collection of colored pixels, arranged in rows and columns.

So, what’s the difference between a Bitmap object and an Image object? The Image object is static. It provides properties that retrieve the attributes of the image stored in the object, but you can’t edit the image’s pixels. The Bitmap object, on the other hand, provides methods that allow you to read and set its pixels. In the last section of this chapter, you’re going to build an image-processing application. But first, let’s look at the information stored in a Bitmap or Image object. Both bitmaps and image are made up of color values (the color of each pixel), so it’s time to look at how GDI+ stores and handles colors in detail.

**Specifying Colors**

You’re already familiar with the Color common dialog box, which lets you specify colors by manipulating their basic components. If you attempt to specify a Color value through the common dialog box, you’ll see three boxes—Red, Green, and Blue (RGB)—whose values change as you move the cross-shaped pointer over the color spectrum. These are the values of the three basic colors that computers use to specify colors. Any color that can be represented on a computer monitor is specified by means of the RGB colors. By mixing percentages of these basic colors, you can design almost any color.

The model of designing colors based on the intensities of their RGB components is called the *RGB model*, and it’s a fundamental concept in computer graphics. If you aren’t familiar with this model, this section is well worth reading. Every color you can imagine can be constructed by mixing the appropriate percentages of the three basic colors. Each color, therefore, is represented by a triplet in which red, green, and blue are three bytes that represent the basic color components. The smallest value, 0, indicates the absence of color. The largest value, 255, indicates full intensity, or saturation. The triplet (0, 0, 0) is black, because all colors are missing, and the triplet (255, 255, 255) is white. Other colors have various combinations: (255, 0, 0) is a pure red, (0, 255, 255) is a pure cyan (what you get when you mix green and blue), and (0, 128, 128) is a mid-cyan (a mix of mid-green and mid-blue tones). The possible combinations of the three basic color components are $256 \times 256 \times 256$, or 16,777,216 colors.

**NOTE** Each color you can display on a computer monitor can be defined in terms of three basic components: red, green, and blue.

Notice that we use the term *basic colors* and not *primary colors*, which are the three colors used in designing colors with paint. The concept is the same; you mix the primary colors until you get the desired result. The primary colors used in painting, however, are different. They are the colors red, yellow, and blue. Painters can get any shade imaginable by mixing the appropriate percentages of red, yellow, and blue paint. On a computer monitor, you can design any color by mixing the appropriate percentages of red, green, and blue.

**NOTE** Just as painters don’t work with three colors only, you’re not limited to the three basic colors. The Color object exposes the names of 128 colors, and you can specify colors by name.

The process of generating colors with three basic components is based on the RGB color cube, shown in Figure 14.23. The three dimensions of the color cube correspond to the three basic colors.
The cube’s corners are assigned each of the three primary colors, their complements, and the colors black and white. Complementary colors are easily calculated by subtracting the Color values from 255. For example, the color (0, 0, 255) is a pure blue tone. Its complementary color is (255 – 0, 255 – 0, 255 – 255), or (255, 255, 0), which is a pure yellow tone. Blue and yellow are complementary colors, and they are mapped to opposite corners of the cube. The same is true for red and cyan, green and magenta, and black and white. If you add a color to its complement, you get white.

**Figure 14.23**
Color specification of the RGB color cube

Notice that the components of the colors at the corners of the cube have either zero or full intensity. As you move from one corner to another along the same edge of the cube, only one of its components changes value. For example, as you move from the green to the yellow corner, the red component changes from 0 to 255. The other two components remain the same. As you move between these two corners, you get all the available tones from green to yellow (256 in all). Similarly, as you move from the yellow to the red corner, the only component that changes is the green, and you get all the available shades from yellow to red. As you can guess, this is how GDI+ calculates the gradients: it draw a (imaginary) line between the two points that represent the starting and ending colors of the gradient and picks the colors along this line.

Although you can specify a little more than 16 million colors, you can’t have more than 256 shades of gray. The reason is that a gray tone, including the two extremes (black and white), is made up of equal values of all three primary colors. You can see this on the RGB cube. Gray shades lie on the cube’s diagonal that goes from black to white. As you move along this path, all three basic components change value, but they are always equal. The value (128, 128, 128) is a mid-gray tone, but the values (127, 128, 128) and (129, 128, 128) aren’t gray tones, although they are too close for the human eye to distinguish. That’s why it’s wasteful to store grayscale pictures using a 3-bytes-per-pixel format. A 256-color file format stores a grayscale just as accurately and more compactly. Once you know an image is grayscale, you needn’t store all three bytes per pixel—one value is adequate (the other two components have the same value).

**Defining Colors**
For defining colors, the Color object provides the FromARGB method, which accepts three arguments:

```csharp
Color.FromARGB(Red, Green, Blue)
```
The FromARGB method can produce any color imaginable. I mentioned earlier that the triplet (255, 255, 0) is a pure yellow tone. To specify this Color value with the FromARGB method, you can use a statement such as:

```
newColor = Color.FromARGB(255, 255, 0)
```

The `newColor` variable is a Color value, and you can use it anywhere you could use a color value. To change the form's background color to yellow, you can assign the `newColor` variable to the BackColor property, like this:

```
Form1.BackColor = newColor
```

or you can combine both statements into one like this:

```
Form1.BackColor = Color.FromARGB(255, 255, 0)
```

There another form of the FromARGB method that accepts four arguments. The first argument in the method is the transparency of the color, and it can be a value from 0 (totally transparent) to 255 (totally opaque). The other three arguments are the usual red, green, and blue color components. For more information on transparent colors, see the following section, “Alpha Blending.”

You can also retrieve the three basic components of a Color value with the R, G, and B methods. The following statements print in the Output window the values of the three components of one of the named colors:

```
Dim C As Color = Color.Beige
Console.WriteLine('Red Component = ' & C.R.ToString)
Console.WriteLine('Green Component = ' & C.G.ToString)
Console.WriteLine('Blue Component = ' & C.B.ToString)
```

In an image-processing application, such as the one we’ll develop later in this chapter, we want to read pixel values, isolate their color components, and then process them separately.

**Alpha Blending**

Besides the red, green, and blue components, a color value may also contain a transparency component. This value determines whether the color is opaque, or transparent. In the case of transparent colors, you can specify the degree of transparency. This component is the *alpha component*. The following statement creates a new color value, which is yellow and 25 percent transparent:

```
Dim trYellow As Color
trYellow = Color.FromARGB(192, Color.Yellow)
```

If you want to "wash out" the colors of an image on a form, draw a white rectangle with a transparency of 50 percent or more. The size of the rectangle must be the same as the size of the form, so you can use the ClientRectangle object to retrieve the area taken by the form. Then create a solid brush with a semitransparent color with the `Color.FromARGB` method. The following code segment does exactly that:

```
Dim brush As New SolidBrush(Color.FromARGB(128, Color.White))
Me.CreateGraphics.FillRectangle(brush, ClientRectangle)
```
If you execute these statements repeatedly, the form will eventually become white. Another use of transparent drawing is to place watermarks on images you’re going to publish on the Web. A watermark is a string or logo that’s drawn transparently on the image. It doesn’t really disturb the viewers, but it makes the image unusable on another site. It’s a crude but effective way to protect your images on the Web.

The following statements place a watermark with the string “MySite.Com” on an image. The font is fairly large and bold, and the code assumes that the text fits in the width of the image.

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim WMFont As New Font("Comic Sans MS", 20, FontStyle.Bold)
    Dim WMBrush As New SolidBrush(Color.FromRGB(92, 230, 80, 120))
    PictureBox1.CreateGraphics.DrawString("MySite.com", WMFont, WMBrush, 100, 40)
End Sub
```

You can combine these statements with a simple program that scans all the images in a folder (you’ll find this information in Chapter 13) to write an application that watermarks a large number of files en masse. Figure 14.24 shows the watermarked image produced by the previous code segment.

**Figure 14.24**

Watermarking an image with a semi-transparent string

Another interesting application of transparency is to superimpose a semitransparent drawing over an opaque one. Figure 14.25 shows some text with a 3D look. To achieve this effect, you render a string with a totally opaque brush. Then you superimpose the same string drawn with a partially transparent brush. The superimposed string is displaced by a few pixels in relation to the first one. The amount of displacement, its direction, and the colors you use determine the type of 3D effect (raised or depressed). The second brush can have any color, as long as the color combination produces a pleasant effect. The strings shown in Figure 14.25 were generated with the TextEffects project (button Draw Semi-Transparent Text). If you run the application and look at the rendered strings carefully, you’ll see that they’re made up of three colors. The two original colors appear around the edges. The inner area of each character is what the transparency of the second color allows us to see.

The code behind the Draw Semi-Transparent Text button is quite simple, really. It’s a bit lengthy, but I will include its listing anyway. The code draws the first string with the solid blue brush:

```vbnet
brush = New SolidBrush(Color.FromArgb(255, 0, 0, 255))
```

Then another instance of the same string is drawn, this time with a different brush:

```vbnet
brush.Color = Color.FromArgb(192, 0, 255, 255)
```
This is a semitransparent shade of cyan. The two superimposed strings are displaced a little with respect to one another. The statements in Listing 14.26 produced the two upper strings of Figure 14.25.

**Listing 14.27: Simple Text Effects with Transparent Brushes**

```vbnet
brush = New SolidBrush(Color.FromRGB(255, 0, 0, 255))
drawFont = New Font('Comic Sans MS', 72, Drawing.FontStyle.Bold)
G.DrawString('Visual Basic.NET', drawFont, brush, 10, 30)
brush.Color = Color.FromRGB(192, 0, 255, 255)
G.DrawString('Visual Basic.NET', drawFont, brush, 7, 27)
brush.Color = Color.FromRGB(255, 0, 0, 255)
drawFont = New Font('Comic Sans MS', 72, Drawing.FontStyle.Bold)
G.DrawString('Visual Basic.NET', drawFont, brush, 10, 130)
brush.Color = Color.FromRGB(128, 0, 255, 255)
G.DrawString('Visual Basic.NET', drawFont, brush, 7, 127)
```

**Processing Bitmaps**

Images are arrays of color values. These values are stored in disk files, and when an image is displayed on a PictureBox or Form control, each of its color values is mapped to a pixel on the PictureBox or form. As you’ll see, image processing is nothing more than simple arithmetic operations on the values of the image’s pixels. The ImageProcessing application we’ll build to demonstrate the various image-processing techniques is slow compared with professional applications, but it demonstrates the principles of image-processing techniques and can be used as a starting point for custom applications.

We’ll build a simple image-processing application that can read all the image types VB can handle (BMP, GIF, TIFF, JPEG, and so on), process them, and then display the processed images. There are simpler ways to demonstrate Visual Basic pixel-handling methods, but image processing is an intriguing topic, and I hope many readers will experiment with its techniques in the ImageProcessing application.

*Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com*
An image is a two-dimensional array of pixels in which each pixel is represented by one or more bits. In a black-and-white image, each pixel is represented by a single bit. Image formats that use 256 colors store each pixel in a byte. The best quality images, however, use three bytes per pixel, one for each RGB color component. (For this example, we’ll ignore the alpha channel.)

Let’s look at a simple technique, the inversion of an image’s colors. To invert an image, you must change all pixels to their complementary colors—black to white, green to magenta, and so on (the complementary colors are on opposite corners of the RGB cube, shown in Figure 14.23, earlier in this chapter).

To calculate complementary colors, you subtract each of the three color components from 255. For example, a pure green pixel whose value is \((0, 255, 0)\) will be converted to \((255 – 0, 255 – 255, 255 – 0)\) or \((255, 0, 255)\), which is magenta. Similarly, a mid-yellow tone \((0, 128, 128)\) will be converted to \((255 – 0, 255 – 128, 255 – 128)\) or \((255, 127, 127)\), which is a mid-brown tone. If you apply this color transformation to all the pixels of an image, the result will be the negative of the original image (what you’d see if you looked at the negative from which the picture was obtained).

Other image-processing techniques aren’t as simple, but the important thing to understand is that, in general, image processing is as straightforward as a few arithmetic operations on the image’s pixels. After we go through the ImageProcessing application, you’ll probably come up with your own techniques and be able to implement them.

**VB.NET at Work: The ImageProcessing Project**

The application we’ll develop in this section is called ImageProcessing; it’s shown in Figure 14.26. It’s not a professional tool, but it can be easily implemented in Visual Basic, and it will give you the opportunity to explore various image-processing techniques on your own. To process an image with the application, choose File ➤ Open to load it to the PictureBox control and then select the type of action from the Process menu. Using the ImageProcessing application, you can apply the following effects to an image:

- **Smooth** Reduces the amount of detail in the image by smoothing areas with abrupt changes in color and/or intensity.
- **Sharpen** Brings out the detail in the image by amplifying the differences between similarly colored pixels.
- **Emboss** Adds a raised (embossed) look to the image.
- **Diffuse** Gives the image a “painterly” look.

Next, let’s look at how each algorithm works and how it’s implemented in Visual Basic.

**How the Application Works**

Let’s start with a general discussion of the application’s operation before we get down to the actual code. Once the image is loaded on a PictureBox control, you can access the values of its pixels with the GetPixel method of the bitmap object that holds the image. The GetPixel method returns a Color value, and you can use the R, G, and B methods of the Color object to extract the basic color components. This is a time-consuming step, and for most algorithms, it must be performed more than once for each pixel.
Figure 14.26
The ImageProcessing application demonstrates several image-processing techniques that can be implemented with VB.

All image-processing algorithms read a few pixel values and process them to calculate the new value of a pixel. This value is then written into the new bitmap with the SetPixel method. The syntax of the SetPixel and GetPixel methods of the Bitmap object are as follows:

```vba
color = Bitmap.GetPixel(X, Y)
Bitmap.SetPixel(X, Y, color)
```

where X and Y are the coordinates of the pixel whose value you're reading, or setting. The GetPixel method returns the color of the specified pixel, while the SetPixel method sets the pixel's color to the specified value.

All image-processing algorithms share a common structure as well. We set up two nested loops, one that scans the rows of pixels and an inner loop that scans the pixels in each row. In the inner loop's body, we calculate the current pixel's new value, taking into consideration the values of the surrounding pixels. Because of this, we can't save the new pixel values to the original bitmap. When processing the next pixel, some of the surrounding pixels will have their original values, while some other will have the new values. As a result, we must create a copy of the original bitmap and use this bitmap to retrieve the original values of the pixels. The processed values are displayed on the bitmap of the PictureBox control, so that you can watch the progress of the processing.

The following is the outline of all the algorithms we'll implement shortly:

```vba
bmap = New Bitmap(PictureBox1.Image)
PictureBox1.Image = bmap
Dim tempbmp As New Bitmap(PictureBox1.Image)
Dim i, j As Integer
With tempbmp
    For i = DX To .Height - DX - 1
        For j = DY To .Width - DY - 1
            'calculate new pixel value'
            bmap.SetPixel(j, i, new_pixel_value)
        Next
        If i Mod 10 = 0 Then
            PictureBox1.Invalidate()
        End If
    Next
```
Here's how it works. First, we create a Bitmap object from the image on the PictureBox control. This is the bmap variable, which is then assigned back to the Image property of the control. Everything you draw on the bmap object will appear on the control's surface. We then create another identical Bitmap object, the tempbmp variable. This object holds the original values of all the pixels of the image.

The two nested loops go through every pixel in the image. In the inner loop's body, we calculate the new value of the current pixel and then write this value to the matching location of the bmap object. The new pixel will appear on the control when we refresh it, by calling the control's Invalidate method. This method isn't called every time we display a new pixel. It would introduce a significant delay, so we invalidate the control after processing 10 rows of pixels. This is a good balance between performance and a constant visual feedback.

**Applying Effects**
In the following sections, you'll find a short description and the implementation of a few image-processing techniques (the ones you can apply to the image with the commands of the Process menu).

**Smoothing Images**
One of the simplest and most common operations in all image-processing programs is the smoothing (or blurring) operation. The smoothing operation is equivalent to low-pass filtering: just as you can cut off a stereo's high-frequency sounds with the help of an equalizer, you can cut off the high frequencies of an image. If you're wondering what the high frequencies of an image are, think of them as the areas with abrupt changes in the image's intensity. These are the areas that are mainly affected by the blurring filter.

The smoothed image contains less abrupt changes than the original and looks a lot like the original image seen through a semitransparent glass. Figure 14.27 shows a smoothed image, obtained with the ImageProcessing application.

**Figure 14.27**
Smoothing an image reduces its detail, but can make the image less “noisy” and “busy.”

---

Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com
To smooth an image, you must reduce the large differences between adjacent pixels. Let’s take a block of nine pixels, centered on the pixel we want to blur. This block contains the pixel to be blurred and its eight immediate neighbors. Let’s assume that all the pixels in this block are green, except for the middle one, which is red. This pixel is drastically different from its neighbors, and for it to be blurred, it must be pulled toward the average value of the other pixels. Taking the average of a block of pixels is, therefore, a good choice for a blurring operation. If the current pixel’s value is similar to the values of its neighbors, the average won’t affect it significantly. If its value is different, the remaining pixels will pull the current pixel’s value toward them. In other words, if the middle pixel was green, the average wouldn’t affect it. Being the only red pixel in the block, though, it’s going to come closer to the average value of the remaining pixels. It’s going to assume a green tone.

Here’s an example with numbers: if the value of the current pixel is 10 and the values of its eight immediate neighbors are 8, 11, 9, 10, 12, 10, 11, and 9, the average value of all pixels will be

\[(8 + 11 + 9 + 10 + 12 + 10 + 11 + 9 + 10) / 9 = 10\]

The pixel under consideration happens to be right on the average of its neighboring pixels. The results would be quite different if the value of the center pixel was drastically different. If the center pixel’s value was 20, the new average would be 11. Because the neighboring pixels have values close to 10, they would pull the “outlier” toward them. This is how blurring works. By taking the average of a number of pixels, you force the pixels with values drastically different from their neighbors to get closer to them.

Another factor affecting the amount of blurring is the size of the block over which the average is calculated. We used a 3×3 block in our example, which yields an average blur. To blur the image even more, use a 5×5 block. Even larger blocks will blur the image to the point that useful information will be lost. The actual code of the Smooth operation scans all the pixels of the image (excluding the edge pixels that don’t have neighbors all around them) and takes the average of their RGB components. It then combines the three values with the method Color.FromRGB to produce the new value of the pixel.

The code that implements the smoothing operation (shown in Listing 14.28) is lengthy, but it’s not really complicated. The long statements combine the color components of all nine neighboring pixels. The current pixel has coordinates \((i, j)\), and the neighboring pixels have indices from \(i-1\) to \(i+1\) and from \(j-1\) to \(j+1\). A more elegant approach would be to use two nested loops to iterate through the nine pixels, but this would make the code a little harder to follow.

**LISTING 14.28: SMOOTHING AN IMAGE**

```
Private Sub ProcessSmooth_Click(ByVal sender As System.Object,
                                ByVal e As System.EventArgs) Handles ProcessSmooth.Click

    bmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = bmap

    Dim tempbmap As New Bitmap(PictureBox1.Image)
    Dim DX As Integer = 1
    Dim DY As Integer = 1
    Dim red, green, blue As Integer
    Dim i, j As Integer
```

*Copyright ©2002 SYBEX, Inc., Alameda, CA  www.sybex.com*
With tempbmp
For i = DX To .Height - DX - 1
  For j = DY To .Width - DY - 1
    red = CInt((CInt(.GetPixel(j - 1, i - 1).R) + _
               CInt(.GetPixel(j - 1, i).R) + _
               CInt(.GetPixel(j - 1, i + 1).R) + _
               CInt(.GetPixel(j, i - 1).R) + _
               CInt(.GetPixel(j, i).R) + _
               CInt(.GetPixel(j, i + 1).R) + _
               CInt(.GetPixel(j + 1, i - 1).R) + _
               CInt(.GetPixel(j + 1, i).R) + _
               CInt(.GetPixel(j + 1, i + 1).R)) / 9)
    green = CInt((CInt(.GetPixel(j - 1, i - 1).G) + _
                   CInt(.GetPixel(j - 1, i).G) + _
                   CInt(.GetPixel(j - 1, i + 1).G) + _
                   CInt(.GetPixel(j, i - 1).G) + _
                   CInt(.GetPixel(j, i).G) + _
                   CInt(.GetPixel(j, i + 1).G) + _
                   CInt(.GetPixel(j + 1, i - 1).G) + _
                   CInt(.GetPixel(j + 1, i).G) + _
                   CInt(.GetPixel(j + 1, i + 1).G)) / 9)
    blue = CInt((CInt(.GetPixel(j - 1, i - 1).B) + _
                  CInt(.GetPixel(j - 1, i).B) + _
                  CInt(.GetPixel(j - 1, i + 1).B) + _
                  CInt(.GetPixel(j, i - 1).B) + _
                  CInt(.GetPixel(j, i).B) + _
                  CInt(.GetPixel(j, i + 1).B) + _
                  CInt(.GetPixel(j + 1, i - 1).B) + _
                  CInt(.GetPixel(j + 1, i).B) + _
                  CInt(.GetPixel(j + 1, i + 1).B)) / 9)
    red = Math.Min(Math.Max(red, 0), 255)
    green = Math.Min(Math.Max(green, 0), 255)
    blue = Math.Min(Math.Max(blue, 0), 255)
    bmap.SetPixel(j, i, Color.FromArgb(red, green, blue))
  Next
Next
If i Mod 10 = 0 Then
  PictureBox1.Invalidate()
  PictureBox1.Refresh()
  Me.Text = Int(_
                100 * i / (PictureBox1.Image.Height - 2)).ToString & "%"
End If
Sharpening Images

Since the basic operation for smoothing an image is addition, the opposite operation will result in sharpening the image. The sharpening effect is more subtle than smoothing, but also more common and more useful. Nearly every image published, especially in monochrome (“one-color”) publications, must be sharpened to some extent. For an example of a sharpened image, see Figure 14.28. Sharpening an image consists of highlighting the edges of the objects in it, which are the very same pixels blurred by the previous algorithm. Edges are areas of an image with sharp changes in intensity between adjacent pixels. The smoothing algorithm smoothed out these areas; now we want to emphasize them.

Figure 14.28
The sharpening operation brings out detail that isn’t evident in the original image.

In a smooth area of an image, the difference between two adjacent pixels will be zero or a very small number. If the pixels are on an edge, the difference between two adjacent pixels will be a large value (perhaps negative). This is an area of the image with some degree of detail that can be sharpened. If the difference is zero, the two pixels are nearly identical, which means that there’s nothing to sharpen there. This is called a “flat” area of the image. (Think of an image with a constant background. There’s no detail to bring out on the background.)

The difference between adjacent pixels isolates the areas with detail and completely flattens out the smooth areas. The question now is how to bring out the detail without leveling the rest of the image. How about adding the difference to the original pixel? Where the image is flat, the difference is negligible, and the processed pixel practically will be the same as the original one. If the difference is significant, the processed pixel will be the original plus a value that’s proportional to the magnitude of the detail. The sharpening algorithm can be expressed as follows:

\[ \text{new\_value} = \text{original\_value} + 0.5 \cdot \text{difference} \]

If you simply add the difference to the original pixel, the algorithm brings out too much detail. You usually add a fraction of the difference; a 50% factor is common. The code that implements the Sharpen command is shown in Listing 14.29.
LISTING 14.29: SHARPENING AN IMAGE

Private Sub ProcessSharpen(Click(ByVal sender As System.Object, 
    ByVal e As System.EventArgs) Handles ProcessSharpen.Click
    bmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = bmap
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim DX As Integer = 1
    Dim DY As Integer = 1
    Dim red, green, blue As Integer
    Dim i, j As Integer
    With tempbmp
        For i = DX To .Height - DX - 1
            For j = DY To .Width - DY - 1
                red = CInt(.GetPixel(j, i).R) + 0.5 * 
                    CInt(bmap.GetPixel(j - DX, i - DY).R) - 
                    CInt(bmap.GetPixel(j - DX, i - DY).R))
                green = CInt(.GetPixel(j, i).G) + 0.5 * 
                    CInt(bmap.GetPixel(j - DX, i - DY).G) - 
                    CInt(bmap.GetPixel(j - DX, i - DY).G))
                blue = CInt(.GetPixel(j, i).B) + 0.5 * 
                    CInt(bmap.GetPixel(j - DX, i - DY).B) - 
                    CInt(bmap.GetPixel(j - DX, i - DY).B)))
                red = Math.Min(Math.Max(red, 0), 255)
                green = Math.Min(Math.Max(green, 0), 255)
                blue = Math.Min(Math.Max(blue, 0), 255)
                bmap.SetPixel(j, i, Color.FromArgb(red, green, blue))
            Next
        Next
        If i Mod 10 = 0 Then
            PictureBox1.Invalidate()
            PictureBox1.Refresh()
            Me.Text = Int( 
                100 * i / (PictureBox1.Image.Height - 2)).ToString & "%"
        End If
    End With
    PictureBox1.Refresh()
    Me.Text = "Done sharpening image"
End Sub

The variables DX and DY express the distances between the two pixels being subtracted. You can subtract adjacent pixels on the same row, adjacent pixels in the same column, or diagonally adjacent pixels, which is what I did in this subroutine. Besides adding the difference to the original pixel value, this subroutine must check the result for validity. The result of the calculations may exceed the valid value range for a color component, which is 0 to 255. That’s why you must clip the value if it falls outside the valid range.
**Embossing Images**

To sharpen an image, we add the difference between adjacent pixels to the pixel value. What do you think would happen to a processed image if you took the difference between adjacent pixels only? The flat areas of the image would be totally leveled, and only the edges would remain visible. The result would be an image like the image in Figure 14.29. This effect clearly sharpens the edges and flattens the smooth areas of the image. By doing so, it gives the image depth. The processed image looks as if it’s raised and illuminated from the right side. This effect is known as emboss or bas relief.

**Figure 14.29**
The Emboss special effect

The actual algorithm is based on the difference between adjacent pixels. For most of the image, however, the difference between adjacent pixels is a small number, and the image will turn black. The Emboss algorithm adds a constant to the difference to bring some brightness to areas of the image that would otherwise be dark. The algorithm can be expressed as follows:

\[ \text{new\_value} = \text{difference} + 128 \]

As usual, you can take the difference between adjacent pixels in the same row, adjacent pixels in the same column, or diagonally adjacent pixels. The code that implements the Emboss filter in the ImageProcessing application uses differences in the x and y directions (set the values of the variables `DispX` or `DispY` to 0 to take the difference in one direction only). The Emboss filter’s code is shown in Listing 14.30.

**Listing 14.30: Embossing an Image**

```vba
Private Sub ProcessEmbass_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles ProcessEmbass.Click
    bmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = bmap
```

Copyright ©2002 SYBEX, Inc., Alameda, CA   www.sybex.com
Dim tempbmp As New Bitmap(TextBox1.Image)
Dim i, j As Integer
Dim DispX As Integer = 1, DispY As Integer = 1
Dim red, green, blue As Integer
With tempbmp
  For i = 0 To .Height - 2
    For j = 0 To .Width - 2
      Dim pixel1, pixel2 As System.Drawing.Color
      pixel1 = .GetPixel(j, i)
      pixel2 = .GetPixel(j + DispX, i + DispY)
      red = Math.Min(Math.Abs(CInt(pixel1.R) - CInt(pixel2.R)) + 128, 255)
      blue = Math.Min(Math.Abs(CInt(pixel1.B) - CInt(pixel2.B)) + 128, 255)
      bmp.SetPixel(j, i, Color.FromRGB(red, green, blue))
    Next
    If i Mod 10 = 0 Then
      PictureBox1.Invalidate()
      PictureBox1.Refresh()
      Me.Text = Int(100 * i / (TextBox1.Image.Height - 2)).ToString & "%"
    End If
  Next
End With
PictureBox1.Refresh()
Me.Text = 'Done embossing image''
End Sub

The variables DispX and DispY determine the location of the pixel being subtracted from the one being processed. Notice that the pixel being subtracted is behind and above the current pixel. If you set the DispX and DispY variables to −1, the result is similar, but the processed image looks engraved rather than embossed.

**Diffusing Images**

The Diffuse special effect is different from the previous ones, in the sense that it's not based on the sums or the differences of pixel values. This effect uses the Rnd() function to introduce some randomness to the image and give it a "painterly" look, as demonstrated in Figure 14.30.

This time we won’t manipulate the values of the pixels. Instead, the current pixel will assume the value of another one, selected randomly in its 5×5 neighborhood with the help of the Random class.

The Diffuse algorithm is the simplest one. It generates two random variables, DX and DY, in the range −3 to 3. These two variables are added to the coordinates of the current pixel to yield the coordinates of another pixel in the neighborhood. The original pixel is replaced by the value of the pixel that (DX, DY) pixels away. The code that implements the Diffuse operation is shown in Listing 14.31.
**Figure 14.30**
The Diffuse special effect gives the image a painterly look.

**Listing 14.31: Diffusing an Image**

```vbnet
Private Sub ProcessDiffuse(Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles ProcessDiffuse.Click
    Dim bmp As New Bitmap(PictureBox1.Image)
    PictureBox1.Image = bmp
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim i As Integer, j As Integer
    Dim DX As Integer
    Dim DY As Integer
    Dim red As Integer, green As Integer, blue As Integer
    With tempbmp
        For i = 3 To .Height - 3
            For j = 3 To .Width - 3
                DX = Rnd() * 4 - 2
                DY = Rnd() * 4 - 2
                red = .GetPixel(j + DX, i + DY).R
                green = .GetPixel(j + DX, i + DY).G
                blue = .GetPixel(j + DX, i + DY).B
                bmp.SetPixel(j, i, Color.FromArgb(red, green, blue))
            Next
        Next
    End With
    Me.Text = Int(100 * i / (.Height - 2)).ToString & "%"
    If i Mod 10 = 0 Then
        PictureBox1.Invalidate()
    End If
```

Copyright ©2002 SYBEX, Inc., Alameda, CA www.sybex.com
Open the ImageProcessing application and experiment with the algorithms described in this chapter. Change the parameters of the various algorithms and see how they affect the processed image. You can easily implement new algorithms by inserting the appropriate code in the inner loop’s body. The rest of the code remains the same. Some simple ideas include clipping one or more colors (force the red color component of each pixel to remain within a range of values), substituting one component for another (replace the red component of each pixel with the green or blue component of the same pixel), inverting the colors of the image (subtract all three color components of each pixel from 255), and so on. With a little imagination, you can create interesting effects for your images.

**Summary**

It’s been a long chapter, but graphics have never been simple. This may explain why they’re not among the most favorite programming topics, but they sure are fun. GDI+ brings VB graphics to a new level. GDI+ exposes a whole lot of functionality, and you have seen most of it in this chapter.

The basic object you’ll be using in your code to generate graphics is the Graphics object, and you can retrieve the Graphics object of a form or control with the CreateGraphics method. Then you can call this object’s drawing methods to generate graphics. If you’re going to display a few graphics elements on your form, you can put the corresponding statements in a subroutine that overwrites the OnPaint method, so that the form is redrawn every time it’s refreshed.

If you want to create graphics in response to user actions, keep in mind that anything you draw on the Graphics object returned by the CreateGraphics method is not permanent; it will be ignored when the form is refreshed. To generate permanent graphics on a form or PictureBox control, you must create a Graphics object based on the bitmap of the control. The bitmap is permanent, and it’s refreshed properly when the form is resized or temporarily covered by another form. As far as the drawing methods go, they’re the same no matter how you created the Graphics object.