**The PictureBox Control**

PictureBox controls are among the most powerful and complex items in the Visual Basic Toolbox window. In a sense, these controls are more similar to forms than to other controls. For example, PictureBox controls support all the properties related to graphic output, including AutoRedraw, ClipControls, HasDC, FontTransparent, CurrentX, CurrentY, and all the Drawxxxx, Fillxxxx, and Scalexxxx properties. PictureBox controls also support all graphic methods, such as Cls, PSet, Point, Line, and Circle and conversion methods, such as ScaleX, ScaleY, TextWidth, and TextHeight. In other words, all the techniques that I described for forms can also be used for PictureBox controls (and therefore won't be covered again in this section).

**Loading images**

Once you place a PictureBox on a form, you might want to load an image in it, which you do by setting the Picture property in the Properties window. You can load images in many different graphic formats, including bitmaps (BMP), device independent bitmaps (DIB), metafiles (WMF), enhanced metafiles (EMF), GIF and JPEG compressed files, and icons (ICO and CUR). You can decide whether a control should display a border, resetting the BorderStyle to 0-None if necessary. Another property that comes handy in this phase is AutoSize: Set it to True and let the control automatically resize itself to fit the assigned image.

You might want to set the Align property of a PictureBox control to something other than the 0-None value. By doing that, you attach the control to one of the four form borders and have Visual Basic automatically move and resize the PictureBox control when the form is resized. PictureBox controls expose a Resize event, so you can trap it if you need to move and resize its child controls too.

You can do more interesting things at run time. To begin with, you can programmatically load any image in the control using the LoadPicture function:

Picture1.Picture = LoadPicture("c:\windows\setup.bmp")

and you can clear the current image using either one of the following statements:

' These are equivalent.  
Picture1.Picture = LoadPicture("")  
Set Picture1.Picture = Nothing

The LoadPicture function has been extended in Visual Basic 6 to support icon files containing multiple icons. The new syntax is the following:

LoadPicture(filename, [size], [colordepth], [x], [y])

where values in square brackets are optional. If filename is an icon file, you can select a particular icon using the size or colordepth arguments. Valid sizes are 0-vbLPSmall, 1-vbLPLarge (system icons whose sizes depend on the video driver), 2-vbLPSmallShell, 3-vbLPLargeShell (shell icons whose dimensions are affected by the Caption Button property as set in the Appearance tab in the screen's Properties dialog box), and 4-vbLPCustom (size is determined by x and y). Valid color depths are 0-vbLPDefault (the icon in the file that best matches current screen settings), 1-vbLPMonochrome, 2-vbLPVGAColor (16 colors), and 3-vbLPColor (256 colors).

You can copy an image from one PictureBox control to another by assigning the target control's Picture property:

Picture2.Picture = Picture1.Picture

**The PaintPicture method**

PictureBox controls are equipped with a very powerful method that enables the programmer to perform a wide variety of graphic effects, including zooming, scrolling, panning, tiling, flipping, and many fading effects: This is the PaintPicture method. (This method is also exposed by form objects, but it's most often used with PictureBox controls.) In a nutshell, this method performs a pixel-by-pixel copy from a source control to a destination control. The complete syntax of this method is complex and rather confusing:

DestPictBox.PaintPicture SrcPictBox.Picture, destX, destY, [destWidth], \_  
[destHeight], [srcX], [srcY2], [srcWidth], [srcHeight], [Opcode])

The only required arguments are the source PictureBox control's Picture property and the coordinates inside the destination control where the image must be copied. The destX / destY arguments are expressed in the ScaleMode of the destination control; by varying them, you can make the image appear exactly where you want. For example, if the source PictureBox control contains a bitmap 3000 twips wide and 2000 twips tall, you can center this image on the destination control with this command:

picDest.PaintPicture picSource.Picture, (picDest.ScaleWidth - 3000) / 2, \_  
(picDest.ScaleHeight - 2000) / 2

In general, Visual Basic doesn't provide a way to determine the size of a bitmap loaded into a PictureBox control. But you can derive this information if you set the control's AutoSize property to True and then read the control's ScaleWidth and ScaleHeight properties. If you don't want to resize a visible control just to learn the dimensions of a bitmap, you can load it into an invisible control, or you can use this trick, based on the fact that the Picture property returns an StdPicture object, which in turn exposes the Height and Width properties:

' StdPicture's Width and Height properties are expressed in  
' Himetric units.   
With Picture1  
width = CInt(.ScaleX(.Picture.Width, vbHimetric, vbPixels))  
height = CInt(.ScaleY(.Picture.Height, vbHimetric, \_  
vbPixels))  
End With

By the way, in all subsequent code examples I assume that the source PictureBox control's ScaleWidth and ScaleHeight properties match the actual bitmap's size. By default, the PaintPicture method copies the entire source bitmap. But you can copy just a portion of it, passing a value for srcWidth and srcHeight:

' Copy the upper left portion of the source image.  
picDest.PaintPicture picSource.Picture, 0, 0, , , , , \_  
picSource.ScaleWidth / 2, picSource.ScaleHeight / 2

If you're copying just a portion of the source image, you probably want to pass a specific value for the srcX and srcY values as well, which correspond to the coordinates of the top-left corner of the area that will be copied from the source control:

' Copy the bottom-right portion of the source image  
' in the corresponding corner in the destination.   
wi = picSource.ScaleWidth / 2  
he = picSource.ScaleHeight / 2  
picDest.PaintPicture picSource.Picture, wi, he, , , wi, he, wi, he

You can use this method to tile a target PictureBox control (or form) with multiple copies of an image stored in another control:

' Start with the leftmost column.  
x = 0  
Do While x < picDest.ScaleWidth  
y = 0  
' For each column, start at the top and work downward.  
Do While y < picDest.ScaleHeight  
picDest.PaintPicture picSource.Picture, x, y, , , 0, 0  
' Next row  
y = y + picSource.ScaleHeight  
Loop  
' Next column  
x = x + picSource.ScaleWidth  
Loop

Another great feature of the PaintPicture method lets you resize the image while you transfer it, and you can even specify different zoom-in and zoom-out factors for the x- and y-axes independently. You just have to pass a value to the destWidth and destHeight arguments: If these values are greater than the source image's corresponding dimensions, you achieve a zoom-in effect, and if they are less you get a zoom-out effect. For example, see how you can double the size of the original image:

picDest.PaintPicture picSource.Picture, 0, 0, \_  
picSource.ScaleWidth \* 2, picSource.ScaleHeight \* 2

As a special case of the syntax of the PaintPicture method, the source image can even be flipped along its x-axis, y-axis, or both by passing negative values for these arguments:

' Flip horizontally.  
picDest.PaintPicture picSource.Picture, \_  
picSource.ScaleWidth, 0, -picSource.ScaleWidth  
' Flip vertically.  
picDest.PaintPicture picSource.Picture, 0, \_  
picSource.ScaleHeight, , -picSource.ScaleHeight  
' Flip the image on both axes.  
picDest.PaintPicture picSource.Picture, picSource.ScaleWidth, \_  
picSource.ScaleHeight, -picSource.ScaleWidth, -picSource.ScaleHeight

As you might expect, you can combine all these effects together, magnifying, reducing, or flipping just a portion of the source image, and have the result appear in any point of the destination PictureBox control (or form). You should find no problem in reusing all those routines in your own applications.

As if all these capabilities weren't enough, we haven't covered the last argument of the PaintPicture method yet. The opcode argument lets you specify which kind of Boolean operation must be performed on pixel bits as they're transferred from the source image to the destination. The values you can pass to this argument are the same that you assign to the DrawMode property. The default value is 13-vbCopyPen, which simply copies the source pixels in the destination control. By playing with the other settings, you can achieve many interesting graphical effects, including simple animations.

**The Image Control**

Image controls are far less complex than PictureBox controls. They don't support graphical methods or the AutoRedraw and the ClipControls properties, and they can't work as containers, just to hint at their biggest limitations. Nevertheless, you should always strive to use Image controls instead of PictureBox controls because they load faster and consume less memory and system resources. Remember that Image controls are windowless objects that are actually managed by Visual Basic without creating a Windows object. Image controls can load bitmaps and JPEG and GIF images.

When you're working with an Image control, you typically load a bitmap into its Picture property either at design time or at run time using the LoadPicture function. Image controls don't expose the AutoSize property because by default they resize to display the contained image (as it happens with PictureBox controls set at AutoSize = True). On the other hand, Image controls support a Stretch property that, if True, resizes the image (distorting it if necessary) to fit the control. In a sense, the Stretch property somewhat remedies the lack of the PaintPicture method for this control. In fact, you can zoom in to or reduce an image by loading it in an Image control and then setting its Stretch property to True to change its width and height:

' Load a bitmap.  
Image1.Stretch = False  
Image1.Picture = LoadPicture("c:\windows\setup.bmp")  
' Reduce it by a factor of two.  
Image1.Stretch = True  
Image1.Move 0, 0, Image1.Width / 2, Image1.Width / 2

Image controls support all the usual mouse events. For this reason, many Visual Basic developers have used Image controls to simulate graphical buttons and toolbars. Now that Visual Basic natively supports these controls, you'd probably better use Image controls only for what they were originally intended.