Use Bitmap Animation for Real-Time Graphing

Click & Retrieve Source CODE!

A few controls, a few API calls, and voilà! Spinning globes and scrolling graphs in your apps.

by Mark Pruett

Nearly all Windows applications use some kind of animation, from simple ones like command buttons (i.e., two framed animation sequences) and percentage controls (showing the progress of a task) to those little bitmap sequences that web browsers display during HTML document loads.

You can program such animations several ways in VB. Let's start by doing simple frame animation using VB's Picture Clip control, which takes the least amount of program code. It can run a succession of bitmaps to create the computer equivalent of a movie.

You'll probably have more trouble creating the bitmapped pictures themselves than using the control. Fortunately, many graphics apps can help you create a variety of two- and three-dimensional animation sequences. Even Windows Paintbrush lets you create simple and effective animation sequences.

However, for this exercise I used Virtual Reality Lab's VistaPro to create the animation sequence. It helps you create and render computer generated landscapes. With it I created 33 frames showing a medieval castle (see Figure 1). Each frame shows the castle from a slightly different angle, as if you walked around it, taking a photo every few yards, stopping just short of your starting point. With this I could create a continuous animation loop.

My frame animation of the castle just needed one form containing three controls: a picture box, a timer, and the Picture Clip control (add it to your VB3 or VB4 toolbox if you don't have it there by default).

The Picture Clip control lets you store and access the many individual frames needed to create a realistic animation. At design time, the control resembles a conventional picture box control, complete with a Picture property that we'll use to load a bitmap. The Picture Clip control only stores a single bitmap. To display all 33 frames of our animation, we break up a single large bitmap into smaller pieces.

By setting its Rows and Cols properties you tell the control to treat the single bitmap it contains as if it were a two-dimensional array of bitmaps. A Picture Clip control with Rows set to 2 and Cols set to 3 can be accessed, via a property called GraphicCell, as a sequence of six separate bitmaps, numbered zero through 5. For example, Frame 3 would be the first bitmap in the second row.

Mark Pruett is the author of Black Art of Visual Basic Game Programming, *published by Waite Group Press. Reach him on CompuServe at 74133,3406 or on the internet at pruettm@vancpower.com.*

This means you have to take all your separate frame bitmaps and store them in a single large bitmap. You can do this by cutting and pasting the separate frames via Windows Paintbrush or just about any bitmap editor.

My CASTANIM.BMP bitmap holds 33 rows of individual frames in a single column. I could have used other variations, such as 3 rows of 11 columns, just as easily. I assigned this bitmap to the Picture Clip control in my project by setting its Picture property. Then I set its Rows property to 33 and its Cols property to 1. Figure 1 shows how the form looks at design time, with part of the Picture Clip control running off the bottom of the form.

My program takes one frame from the Picture Clip control and displays it in the form's picture box control. Animating it only takes a few lines of code (see Listing 1).

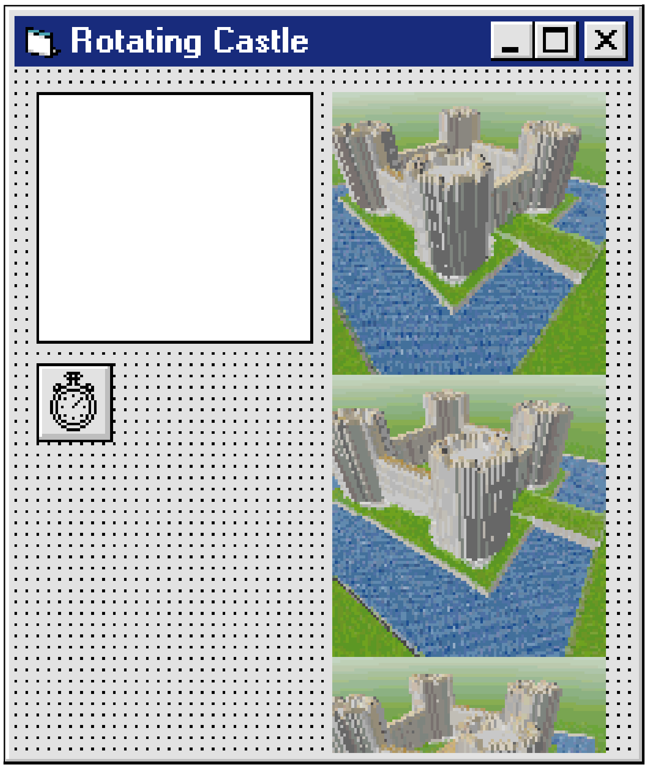

FIGURE 1 ***Going Around and Around.*** *A design-time view of a simple animation program shows the Picture Box where the animation will be displayed and part of the PictureClip control where the actual animation frames are stored. Users don't see the PictureClip control at run-time. And a simple Windows API call lets you achieve the same level of animation without using the PictureClip control.*

ON TO WINDOWS API BITBLTING

Picture Clip works fine for simple animations, but for more speed and scalability we'll turn to the Windows API's Bit-Block Transfer function. Luckily, it only takes a few more lines of code to make the single Windows API call needed to mimic Picture Clip. Then you can use the BitBlt function to copy all or part of a bitmap onto another bitmap.

Don't let the size of BitBlt's function declaration put you off:

```
Declare Function BitBlt Lib "GDI" _
    (ByVal hDestDC As Integer, _
     ByVal x As Integer, _
     ByVal y As Integer, _
     ByVal nWidth As Integer, _
     ByVal nHeight As Integer, _
     ByVal hSrcDC As Integer, _
     ByVal xSrc As Integer, _
     ByVal ySrc As Integer, _
     ByVal dwRop As Long) As Integer
```

Consider VB's Left and Top properties. These usually refer to how far you should place a control from the top and left sides of a form. You can view BitBlt's x and y parameters similarly: x indicates a point x pixels from the left of the bitmap, and y indicates y pixels from the top. So you'd indicate the upper left corner of a bitmap as x-y coordinate (0,0).

This function really says, "Copy to location (x,y) in bitmap hDestDC a portion of bitmap hSrcDC." The portion of hSrcDC is nWidth pixels wide and nHeight pixels high, and starts at location (x,y) in hSrcDC. The final parameter, dwROP, is a flag specifying how to copy the bitmap. You have a number of options. We'll use the most common one, SRCCOPY, to tell BitBlt to replace the destination area with the source bitmap.

Now we can rewrite our first example, using BitBlt instead of a Picture Clip control. We delete the Picture Clip control from the form at design time, replacing it with a second picture box control, Picture2. We assign the this control's Picture property to the same bitmap we used before, CASTANIM.BMP.

To emulate Picture Clip we need to set two properties in the picture box control. First, we set Picture2's AutoRedraw property to True. This property forces Visual Basic to maintain a copy of Picture2's bitmap in memory, making it a *persistent bitmap* in Windows parlance. Next, we set the Visible property of Picture2 to False, rendering the control invisible at run-time. And of course we add the BitBlt function to our form's declaration section, along with the needed SRCCOPY constant:

```
Const SRCCOPY = &HCC0020
```

Then we modify the Timer event of our program to complete the change:

```
Sub Timer1_Timer ()
Static CellNum As Integer
Dim rc As Integer
CellNum = (CellNum + 1) Mod 33
    rc = BitBlt(Picture1.hDC, 0, 0, _
         100, 100, Picture2.hDC, 0, _
         CellNum * 100, SRCCOPY)
End Sub
```

We still use CellNum to cycle through the bitmap's "frames." The Mod function forces CellNum's value back to zero after displaying the last frame to close the animation loop. The BitBlt function cuts a 100-by-100 pixel frame out of the appropriate section of Picture2, and displays it in Picture1. By passing a ySrc parameter of CellNum * 100, we tell BitBlt how far down in Picture2 we should go to find the proper frame.

Therefore, half a dozen extra lines of code let us mimic the Picture Clip control's abilities and avoid having to distribute yet another VBX or OCX with our apps.

DRAWING ON A PICTURE BOX

Picture Boxes display bitmaps nicely, and provide dynamic drawing surfaces as well. You can draw lines and shapes directly onto a Picture Box, using either built-in methods such as Circle, Line, and PSet, or Windows API graphics functions, which usually run faster. Let's use them to build a realtime graphing control within a Picture Box.

Realtime graphing lets you display data from a source as the data is acquired. For example, you might want to graph temperature changes that your app is receiving through a PC's serial port, connected in turn to an external device.

Let's say that as the program receives new data, it graphs the data on the Picture Box's y axis. With each new data reading, the currently displayed data scrolls to the left, and the new data is plotted at the right side of the Picture Box. Old data eventually disappears off the left side of the Picture Box. Users see a smoothly scrolling line graph (see Figure 2).

We create the appearance of scrolling with a little BitBlt trick. We previously used BitBlt to copy from one Picture Box to another. This time we'll use BitBlt to copy a piece of a bitmap within the *same* Picture Box. Actually we'll be copying the right side of the graph in a Picture Box to the left side of the same Picture Box. This will leave us a narrow area on the right side on which we can draw our new incoming data, using the simple Windows API calls MoveTo and LineTo:

```
Declare Function MoveTo Lib "GDI" _
    (ByVal hDC As Integer, _
     ByVal x As Integer, _
     ByVal y As Integer) As Long
Declare Function LineTo Lib "GDI" _
    (ByVal hDC As Integer, _
     ByVal x As Integer, _
     ByVal y As Integer) As Long
```

VB4

```
Sub Form_Load ()
' Make sure picture box is same size as animation
' frame
    Picture1.AutoSize = True
' Set and turn on the timer
    Timer1.Interval = 50
    Timer1.Enabled = True
End Sub
Sub Picture1_Click ()
' Toggle timer on and off by clicking on picture box
    Timer1.Enabled = Not Timer1.Enabled
End Sub
Sub Timer1_Timer ()
Static CellNum As Integer
' Display next frame in the animation, looping back
CellNum = (CellNum + 1) Mod (PicClip1.Rows *
PicClip1.Cols)
    Picture1.Picture = PicClip1.GraphicCell(CellNum)
End Sub
```

LISTING 1 ***Create a Simple Frame Animation.*** *Use a PictureClip, Picture Box and Timer control to create an animation sequence. Applications like Mosaic and Netscape use animation sequences to provide a visually appealing method of user feedback.*

property and an x-y coordinate inside it. MoveTo doesn't actually draw. Just as you often lift and move your hand when drawing on a piece of paper, MoveTo moves a "virtual pen" to a point on the Picture Box drawing surface. When you provide a subsequent call to LineTo, VB draws a line from the point of the previous MoveTo to the point specified by LineTo.

Think of a line graph as a collection of short line segments. You use MoveTo and LineTo to draw a new segment each time the app gets new data. By combining this technique with the scrolling BitBlt technique, we can graph virtually any x-y data in a Picture Box (see Listing 2).

I encapsulated all the graphing work into the single reusable function UpdateGraph. This function supports up to 10 different graphs, each in its own Picture Box. We assign each graph its own logical "channel" because UpdateGraph needs to remember the previous y value of each graph. The MoveTo function

VB4

```
Option Explicit
' A Realtime Graphing Demonstration by Mark Pruett
Const RED = &HFF&
' Constants and Windows API Calls
Const GREEN = &HFF00&
Const SRCCOPY = &HCC0020
' Windows GDI Bitmap API constants and functions
Const SRCBLACK = &H42
Declare Function BitBlt Lib "GDI" (ByVal hDestDC As _
     Integer, ByVal x As Integer, ByVal y As _
     Integer, ByVal nWidth As Integer, ByVal nHeight _
     As Integer, ByVal hSrcDC As Integer, ByVal _
     XSrc As Integer, ByVal YSrc As Integer, ByVal _
     dwRop As Long) As Integer
Declare Function LineTo Lib "GDI" (ByVal hDC As _
   Integer, ByVal x As Integer, ByVal y As Integer) _
   As Long
Declare Function MoveTo Lib "GDI" (ByVal hDC As _
   Integer, ByVal x As Integer, ByVal y As Integer) _
   As Long
Sub btnOnOff_Click ()
' This subroutine feeds new "data" to the graphs
Static x1 As Integer, y1 As Integer
Static x2 As Integer, y2 As Integer
Static Running As Integer
Dim fps As Integer, sec As Single
Dim i As Integer
Dim rc As Integer
    Running = Not Running
     ' Turn graph on or off If Not Running Then
        btnOnOff.Caption = "Start"
        Exit Sub
    End If
    btnOnOff.Caption = "Stop"
        sec = Timer
    While Running
       ' Scroll and update
       x1 = x1 + 1
       ' Calculate the new y1 position
       y1 = (Sin(x1 * 132) * 60) + _
          (picGraph1.ScaleHeight / 2
       If x1 >= 118 Then x1 = 0)
       ' Reset x to avoid overflow in y calculation
        x2 = x2 + 1
       ' Calculate the new y2 position
        y2 = Rnd * picGraph2.ScaleHeight
        For i = 1 To scrSpeed.Value
           ' Pause briefly
           DoEvents
        Next
        UpdateGraph picWork, 2, x2, y2
       ' Update the two graphs
        UpdateGraph picGraph1, 1, x1, y1
        rc = BitBlt(picGraph2.hDC, 0, 0, _
          picWork.ScaleWidth, _
            picWork.ScaleHeight, picWork.hDC, _
             0, 0, SRCCOPY)
        fps = fps + 1
       ' Show statistics for frames-per-second
        If Timer - sec > 1 Then
            Me.Caption = fps & " fps"
                fps = 0
                sec = Timer
            End If
    Wend
End Sub
Sub Form_Load ()
' Make sure all Picture Box ScaleModes are set
' to PIXEL, the mode used by Windows API calls
Const PIXEL = 3
picGraph1.ScaleMode = PIXEL
    picGraph2.ScaleMode = PIXEL
    picWork.ScaleMode = PIXEL
picWork.Width = picGraph2.Width
' Make picWork a persistent bitmap
    picWork.Height = picWork.Height
    picWork.Visible = False
    picWork.AutoRedraw = True
End Sub
Sub Form_Unload (Cancel As Integer)
' Terminate program when form unloads
End
End Sub
Sub UpdateGraph (PicBox As PictureBox, ByVal Channel _
  As Integer, XValue As Integer, YValue As Integer)
' Update graph in PicBox (which is associated with
' Channel) using new x-y coordinates from XValue and
' YValue
Static yprev(1 To 10) As Integer
Dim rc As Long
Dim ScrollAmt As Integer
ScrollAmt = 2
  ' Shift Bitmap to the left
    rc = BitBlt(PicBox.hDC, 0, 0, PicBox.ScaleWidth - _
        ScrollAmt, PicBox.ScaleHeight, PicBox.hDC, _
        ScrollAmt, 0, SRCCOPY)
     ' Clear area where we will draw new plot (right
     ' side of PictureBox)
     rc = BitBlt(PicBox.hDC, PicBox.ScaleWidth - _
     ScrollAmt, 0, ScrollAmt, PicBox.ScaleHeight,
        PicBox.hDC, PicBox.ScaleWidth - 1, 0, SRCBLACK)
PicBox.ForeColor = GREEN
    If (XValue Mod 20) = 0 Then
     ' Draw Vertical Grid Line
        rc = MoveTo(PicBox.hDC, PicBox.ScaleWidth - _
           1, 0)
     rc = LineTo(PicBox.hDC, PicBox.ScaleWidth - 1, _
        PicBox.ScaleHeight)
    End If
    rc = MoveTo(PicBox.hDC, PicBox.ScaleWidth -
ScrollAmt, _
    PicBox.ScaleHeight \ 2)
    ' Draw Horizontal Grid Line
    rc = LineTo(PicBox.hDC, PicBox.ScaleWidth, _
        PicBox.ScaleHeight \ 2)
PicBox.ForeColor = RED
     ' Plot the new point
    rc = MoveTo(PicBox.hDC, PicBox.ScaleWidth - _
        ScrollAmt - 1, yprev(Channel))
    rc = LineTo(PicBox.hDC, PicBox.ScaleWidth - 1, _
        YValue)
yprev(Channel) = YValue
    DoEvents
End Sub
```

LISTING 2 ***Graphing in Real Time.*** *This example program defines the reusable subroutine UpdateGraph, which can be used to display up to 10 real-time graphs simultaneously.*

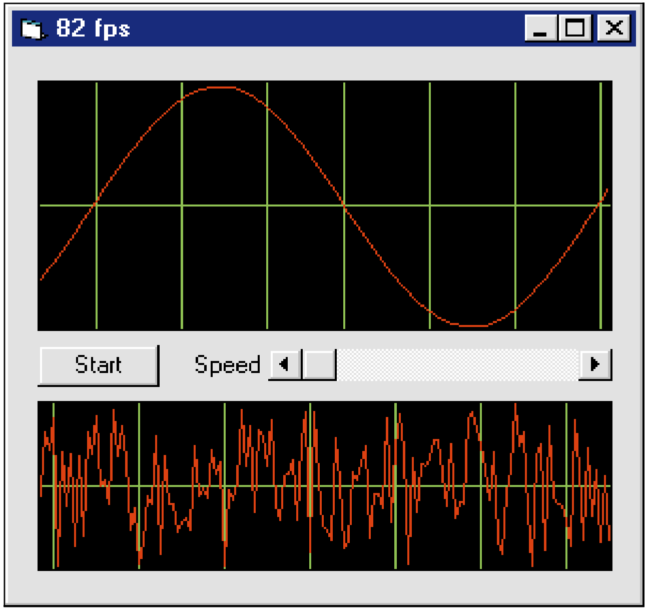

FIGURE 2 ***Realtime Graphing.*** *This example program shows how to display scrolling, realtime graphs with a few Windows API calls and a standard Picture Box control. You can package all the necessary code in a single, reusable, Visual Basic subroutine. The caption bar displays how many frames are being displayed per second.*

uses this to move to the appropriate starting point for the next line segment.

One warning: since the UpdateGraph, as written, draws directly onto on-screen Picture Boxes, VB doesn't maintain a persistent bitmap in memory. No problem—until another window partially obscures your graph. BitBlt is using the actual contents of the screen as the source for its bitmap. So when another window takes that location, BitBlt does its job, copying a rectangular area at a particular location, copying what it "sees." You will then see your graph looking like a section of the other window.

You can get around this with the BitBlt technique I used to display my animated castle. Instead of rendering directly onto an onscreen Picture Box, draw your graph onto an invisible, persistent Picture Box, then copy it to an on-screen Picture Box as if it were an animation frame. I use this technique in Listing 2 for the bottom graph. Try running the program, then partially obscure the right side of the form with another window. The bottom graph continues to display normally while the upper graph is trashed.

Or try setting the on-screen Picture Box to True. Then, after each call to UpdateGraph (or better yet, as the last statement within UpdateGraph), call the Picture Box's Refresh method. The graph is drawn into the Picture Box's persistent bitmap, then displayed on-screen with a call to Refresh. You can modify UpdateGraph to include this behavior.

Now you can add realtime graphing to any VB3 program. For VB4 just modify the BitBlt, LineTo, and MoveTo calls to their GDI32 equivalents.