



## Basic introduction to Graphics programming in VB

Written by Mike D Sutton Of EDais

[Http://www.mvps.org/EDais/](http://www.mvps.org/EDais/)

[EDais@mvps.org](mailto:EDais@mvps.org)

- 05.03.2002 -

## **VB graphics programming**

---

I was asked recently where on the site one should go to find a basic introduction to graphics programming in VB, and I realised that there was nothing that really covered the basics; things that once you're messing around with more advanced things you tend to take for granted (and indeed I had done in my other tutorials on the subject.) As such I'm writing this tutorial with no assumed knowledge (hopefully) apart from a very basic knowledge of VB programming in general.

Chapter 1 - The picture box

Chapter 2 - Basic drawing

Chapter 3 - Customising drawing

Chapter 4 - Working with images (Bitmaps)

Chapter 5 - Image manipulation

Chapter 6 - Introduction to the API

Chapter 7 - Migrating to API drawing

Chapter 8 - Customising drawing and Image manipulation with the API

Chapter 9 - System colours

Chapter 10 - Further reading

## The picture box

---

First thing's first, before we can draw any kind of graphics we'll need to draw it onto so I'll introduce you to your best friend when graphics programming – the picture box control.



Drop one on a form now and let's have a look at it

*Note: The Image control is not the same as the picture box control and will not allow you do the same things*

Ok, it doesn't look like much but let's have a quick look at some of its properties. The "Appearance" property is the first one we'll look at. If you go to change its value, you'll see that it turns into a little drop down menu with two options; flat and 3D. Changing the property to flat makes the border of the picture box just a single line rather than the 'bevelled' effect we get with the 3D property. Don't worry that the background colour of the picture box changes when you change this property; it's a known problem.

The next property is "AutoRedraw" but we'll come back to that later as it's quite an important and useful one, the same with the "AutoSize" property.

"BackColor" allows you to change the background colour of the picture box, you can either pick from the system colours or pick from a palette. If you want another colour that doesn't appear on either section then you can opposite-click on any of the bottom two rows of squares on the palette tab and that will bring up a colour picker where you can select or enter your own HSL or RGB colour references.

Next we have "BorderStyle", which will either show or hide the border of the picture box.

Now there's a whole load more properties listed but for the time being we're not interested in most of them so just scroll down until you come to the Picture property. When you go to edit that value, you'll see that a small ellipsis (...) button appears next to the property value, clicking on this will bring up an open dialogue where you can find the picture you want to put into the control. The control natively supports the Bitmap image formats (.BMP, .DIB, .ICO and .CUR), Jpeg (.JPG), Gif (.GIF) and Metafiles (.EMF and .WMF). Once you've chosen an image then it should appear in the picture box background.

Now we have an image embedded in the control we can go back and have a look at the "AutoSize" property, changing this now will automatically scale the picture box to the size of the background image.

At the moment that's all we'll deal with as far as the properties go, there are some other useful ones but you can pick those up as you go along.

## Basic drawing

Now you're familiar with the control we'll be using to draw on, we can dive into the code and actually do some drawing. At this point it's worth mentioning that the drawing methods used in this section aren't very efficient but hopefully very easy to use and learn. At the risk of introducing bad practice into your code at an early level, I'll go through them anyway since a basic knowledge of what the controls have to offer is important. Once you get more adept with graphics programming you'll use more efficient methods of drawing and use these less and less if at all. To this end, at this point you either have the choice of learning the basics then moving on to the more advanced stuff later (highly recommended), or you can just jump straight onto the advanced methods now (or if you've had some prior experience doing graphics programming in VB), which you'll find in chapter 6.

At this point I'll assume you want to learn the basics so let's get stuck in. First up, drop a picture box and command button on the form, arrange them to taste (just make sure you can see both!) then double click the button and you'll be whisked to the code view.

In case you're not familiar with computer graphics, any image displayed on the monitor is made up of lots of small points of light called "Pixels" (picture elements.) When doing any kind of drawing to the computer screen you're using pixels, however a lot of time you don't have to worry about the individual pixels. For instance you don't have to specifically set every single pixel making a line or polygon, instead you can use pre-made functions that do this for you.



Pixels

The picture box contains some basic graphics routines to get us started and it's function to draw a pixel is called PSet() (Pixel set): PSet (X, Y), Colour  
 Lets give it a whirl; type this into your command button's click event:

```
Picture1.PSet (10, 10), 0
```

Run the application (F5) and hit the command button. If you look *\_really\_* closely at the top left hand corner of the picture box then you should see a tiny black dot – Success, you've drawn a pixel!

At this point you may be scratching your head since a couple of things may seem odd at this point depending on how much you've thought about it. First off, we told our pixel to be drawn at point (10,10) (coordinates are measured from the top left of the control starting from (0,0) as the top-left-most pixel) where as the one we just drew was placed at (1,1) instead. Secondly the last parameter is the colour but I put the parameter as 0, how that that be a colour?

Ok, first question first. VB controls have a rather annoying (if you ask me) scale property that changes how the coordinate system and thus the drawing routines work. By default the scale mode on the picture box is set to "Twips" which are roughly equal to one fifteenth of a pixel each. As such when we specified that the pixel be drawn at (10,10), VB interpreted this in the control's scale mode of Twips and converted it to Pixels, which comes out to (0.666,0.666). Pixels coordinates cannot be fractional since they relate to absolute positions on the screen, so they're rounded to the nearest integer coordinates (1,1).

To change this, go back into design mode and change the picture box's "ScaleMode" property to "3 - Pixel". Now when you run the application again, you'll see that the pixel is drawn correctly at pixel (10,10).

Now, let's deal with that colour problem. In VB, colours are passed around as numbers, Long (32-bit) integers to be more accurate, and using them we can define any RGB colour. The easiest way of defining colours are the VB colour constants:

vbRed  
vbGreen  
vbBlue  
vbCyan  
vbMagenta  
vbYellow  
vbBlack  
vbWhite



However if we want other colours then we can use the RGB() function: RGB(Red, Green, Blue) - Each channel value is defined in the range 0 - 255 where 0 indicates the lowest intensity and 255 indicates the highest intensity. I.e. RGB(32, 32, 32) would be a dull grey colour, where as RGB(255, 128, 0) would be **bright orange**.

Let's change the PSet() call to be a bit more colourful:

```
Picture1.PSet (10, 10), vbRed
```

This time you should see a red pixel being drawn instead of the black one. Don't worry for the time being why 0 == Black, you can always experiment with that later.

Now you've mastered pixels, we can move on to drawing simple shapes. The first one we'll deal with is lines which, surprisingly, is done by using the Line() method: Line (Xa, Ya)-(Xb, Yb), Colour, Flags

Basically there's little more to learn here over the PSet() function, it just takes two coordinates (For each end of the line), a colour and some optional flags. These flags

specify what the function should do but we'll come back to these in a second though after trying out the basic function. Stick this in your click event, after the PSet() call:

```
Picture1.Line (5, 15)-(25, 15), vbBlue
```

Now when you run the application and click on the button you'll see a small blue line being drawn beneath the pixel - Cool!.. Well, maybe not, but its progress!

Now back to those flags that I was talking about earlier. You can draw three different types of 'line' with the one function; the flags define which you want. When this property is left blank then the default line is drawn, however the "B" flag indicates that VB should draw a rectangle between the two coordinates rather than a straight line. Finally, the "BF" flag indicates that VB should draw a filled rectangle between the two coordinates.

To demonstrate this, paste these two calls into the bottom of your subroutine, which show both properties:

```
Picture1.Line (5, 20)-(25, 25), vbYellow, B
Picture1.Line (5, 30)-(25, 35), vbMagenta, BF
```

It's a somewhat weird method this one since it uses very unusual calling convention but unfortunately this is just the way they work since they've been carried through since the early versions of BASIC in the same way for compatibilities sake.

In the same way as we have the ability to draw lines on the picture box, we can also draw circles which is accomplished via, yes you guessed it, the Circle() method: Circle (X, Y), Radius, Colour

*Note; It does actually have more properties than just these to define segments of a circle and things like that but we'll not bother with these right now.*

The coordinate passed to the function defines the Centre of the circle rather than the top left as with the other functions, put this into your subroutine and let's see it in action:

```
Picture1.Circle (45, 20), 15, vbGreen
```

This draws a green circle with a 15-pixel radius at the point (45,20).

The last thing to deal with in this chapter is text which is drawn using the unlisted Print() method: Print Text

```
Picture1.Print "Hello, world!"
```

The text is drawn at the 'current point' of the control, which is set via the "CurrentX" and "CurrentY" properties and by default it sets the position that the last drawing operation finished.

```
Picture1.CurrentX = 2
Picture1.CurrentY = 40 ' Set the current position
Picture1.Print "Look, some text!"
```

## EDais graphics programming tutorial

The colour of the text is defined by the “ForeColor” property, while the font size and style can be changed via the “Font” object property or “Font\*” properties at runtime. So now you can draw pixels, lines, boxes, filled boxes, circles and text in any number of colours.

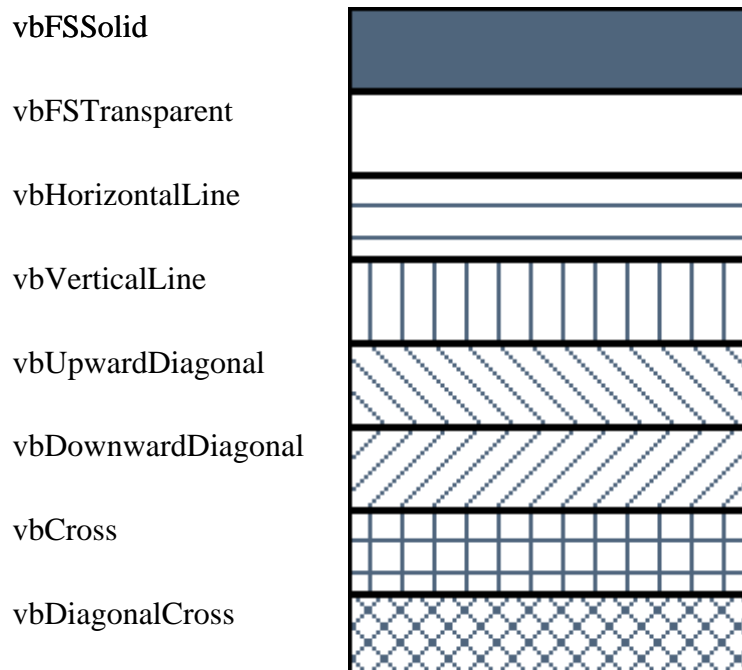
## Customising drawing

Some of the shapes that we can draw can be customised by using them in conjunction with the other properties of the picture box object. For instance, by changing the “FillStyle” property we can create a filled circle.

In case you’ve not come from the last chapter then so far we have a form with a picture box and button on, and the following code:

```
Private Sub Command1_Click()
    Picture1.PSet (10, 10), vbRed
    Picture1.Line (5, 15)-(25, 15), vbBlue
    Picture1.Line (5, 20)-(25, 25), vbYellow, B
    Picture1.Line (5, 30)-(25, 35), vbMagenta, BF
    Picture1.Circle (45, 20), 15, vbGreen
End Sub
```

Change the “FillStyle” property on the picture box to “0 - Solid” and run the application again. Hmm, now the box and circle are filled in black, not the colour we specified. This is because it’s filling using the fill colour of the picture box, this can be changed through the “FillColor” property of the picture box, change that to something more colourful and try it again. All of these properties of the picture box can be changed on the fly at runtime as well as at design time; you just need to specify one of the following constants:



In this case we don’t want the box draw with the “B” flag to be filled, so for that we’ll set the fill mode as transparent but solid for the circle. Add this line to the beginning of the subroutine to set the fill style to transparent.

```
Picture1.FillStyle = vbFSTransparent
```










## EDais graphics programming tutorial














Now add this line before the call to Circle() so it will be filled solidly:

```
Picture1.FillStyle = vbFSSolid
```

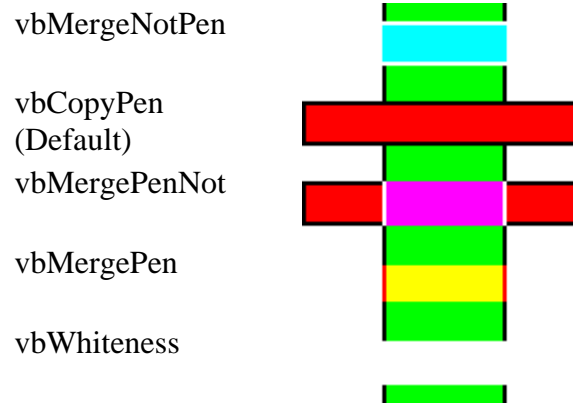
Similarly the “DrawStyle” property will change the way the lines are drawn in the same way the “FillStyle” property changes the way the fills are drawn. The draw style is defined as one of the following constants:

vbSolid	
vbDash	
vbDot	
vbDashDot	
vbDashDotDot	
vbInvisible	
vbInsideSolid	

The “DrawMode” property changes the way the drawings are combined with the existing image. Here you see each of the draw modes when a red filled rectangle is drawn over a green filled rectangle (both have solid black borders):

	+	
vbBlackness		
vbNotMergePen		
vbMaskNotPen		
vbNotCopyPen		
vbMaskPenNot		
vbInvert		
vbXorPen		
vbNotMaskPen		
vbMaskPen		
vbNotXorPen		
vbNop		

## EDais graphics programming tutorial



If this all looks pretty confusing and/or useless then don't worry, most of the time you'll never need to both with this anyway and when you do simply write a test application and cycle through the available draw modes until you find one that does what you want. There is logic behind the seemingly random colours in the above images based on Boolean mathematics.

## Working with images (bitmaps)

In the same way as you can change the drawing styles at runtime, you can also load a picture at design time, this is through the LoadPicture() function. To use it, simply send LoadPicture() the full path to your image and it will put the image into the picture box (it actually creates a copy of the image in memory, then assigns that to the picture box, the original file is not linked to the object in any way)

```
Set Picture1.Picture = LoadPicture("C:\MyPic.bmp")
```

If you've set the AutoSize property to True then this line will automatically scale the control to the size of the picture you've specified. If the file doesn't exist then you'll get a runtime error, you can always check to see if it exists with either the Dir() or FileLen() functions:

```
Private Function FileExist(ByRef inFile As String) As Boolean
    FileExist = CBool(Len(Dir(inFile)))
End Function
```

```
Private Function FileExist(ByRef inFile As String) As Boolean
    On Error Resume Next
    FileExist = CBool(FileLen(inFile) + 1)
End Function
```

Once you've loaded the image into the control you can draw over the top of it using the methods we've covered in the previous chapters including using different draw modes. For instance to invert the entire image you can load the image into the control, set the draw mode to invert and draw a white filled rectangle over the entire image using the Line() function:

```
Picture1.AutoSize = True
Picture1.ScaleMode = vbPixels ' Set the scale mode to pixels
Picture1.Picture = LoadPicture("C:\MyPic.bmp")
Picture1.DrawMode = vbInvert ' Set invert draw mode
Picture1.FillStyle = vbInvisible ' Line() does the fill for us with the "BF" flag
Picture1.Line (0, 0)-(Picture1.ScaleWidth, Picture1.ScaleHeight), vbWhite, BF
```

You may notice at this point that your image is flashing when you load and invert the picture and the image is lost when you move it off the screen. This is because when Windows is told to re-draw your control it simply redraws what it knows about which is just the background picture, everything you draw on top is simply there temporarily as an overlay.

To change this we can enable the "AutoRedraw" property that I mentioned earlier which forces Windows to remember what we've draw on the picture box (Even when it's been covered by another window or moved off the screen), and some other useful things like only redrawing when it won't 'tear' the image in middle of a refresh (this is why the picture was flickering before.)

The down side of auto-redraw is that it does require more memory to use than a standard picture box, so only use it when you absolutely have to. In cases where you want your own drawing to appear in a picture box but don't want to use auto-redraw then put your drawing routine in the Paint() event of the picture box and turn off auto-

redraw, this will fire your drawing code off when Windows is told to re-paint the control.

*Note; This can sometimes cause flickering if it's called in quick succession, the only way of getting round this (apart from enabling auto-redraw) is to subclass the window and catch it's "WM\_PAINT" message, but that's a somewhat more advanced topic, beyond the scope of this tutorial. If your drawing is quite complex and takes too long to draw by calling it every time the window is refreshed then in these cases it's best to simply use auto-redraw.*

To clear this drawing when auto-redraw is enabled, you can use the Cls() method of the picture box, which will revert the image back to its background picture only:

```
' Clear the drawing
Call Picture1.Cls
```

On the other hand if you want to lock the drawing so it remains even when Cls() is called then you can set the background picture to be the current screen image:

```
' Lock the drawing
Set Picture1.Picture = Picture1.Image
```

Finally, if you need to save the image back out to disk then you can use the SavePicture() method: SavePicture(Picture, FileName)

So you could add this line to the end of the current subroutine:

```
' Write the inverted image to disk
Call SavePicture(Picture1.Image, "C:\InvPic.bmp")
```

If you've already locked the image then you can use the Picture property rather than the Image property but it's up to you, the function will accept either.

*Note; This method will only allow you to save uncompressed .BMP images to disk, for compressed/palettred Bitmaps you have to write the images out to disk manually but that's beyond the scope of this tutorial. If you want to pursue this later then have a look at my "Basic introduction to DIB's (Device Independent Bitmaps)" tutorial after you're more familiar with graphics programming in general.*

*To save a .JPG image you can use the IJL, which is over on [www.Intel.com](http://www.Intel.com) and for other formats you'll need to either use a third party control or write your own file I/O routines for that format, you can usually find all the information you need on other image formats over on [www.Wotsit.org](http://www.Wotsit.org).*

So now you know how to load an image into a picture box, edit it, display it properly and finally save the edited version out to disk again.

## Image manipulation

---

We've covered how to draw a pixel into a picture box with PSet() and so now it's time to introduce it's partner in crime; Point() which retrieves the colour at a pixel: Point (X, Y)

In the same way we defined colours before, the colour returned will be as a Long integer specifying the RGB colour. At this point it's probably best to into a little more detail as to how the colour is stored and thus how we can get it back into a format that we can deal with. A Long integer is 4 bytes long (32-bit) in memory and each of the lower three bytes are used to store the colour channels. You can see this by using the Hex() function:

**Call** MsgBox(Hex(RGB(&H11, &H22, &H33)))

This will display "332211" which shows us that Red is being stored in the low byte, green in the next and finally blue in the next. The high byte is not often used apart from in special situations, which I'll cover later.

From this little evaluation we can then go about getting the colour values back from the long, while other languages have bit shifting functions that would make retrieving the colour values easy, VB does not but we can 'fake it' with a little understanding of bitwise mathematics.

Take for instance the number 1; in binary this is written as 00000001 as only the lowest bit place is in use (The bit places representing 128, 64, 32, 16, 8, 4, 2 and 1 respectively, to get the value a binary value holds simply add up the bit places marked with 1's)

Now, let's shift that value left once and see what we get: 00000010 and in decimal this is 2. Hmm, interesting, it's exactly twice the original value. Let's see if this rule continues if we shift left again: 00000100 which in decimal as 4 so the rule's worked. Now if we shift this value right once we get 00000010 which is half the original value, and again this rule sticks for any right bit shift.

So we now know that bit shifting is simply multiplying or dividing a value by 2, if we want to shift 2 places it's (2\*2), 3 places (2\*2\*2) etc, this can be generalised into saying to shift left or right we either multiply or divide the value by (2^Places) (^ = Raised to the power) respectively.

It's also worth noting at this point that it's faster to use VB's integer divide rather than the floating point version for this and it also crops and values that may drop beneath the decimal point. For instance; 00001111 (15) bit shifted right 2 places would equal 00000011.11 (3.75) with a floating-point divide and would be rounded to the nearest integer 00000100 (4), which is the wrong value. Using the integer divide instead we get the desired value of 00000011.

This has covered basic bit shifting (Don't worry if it's a bit confusing right now, especially if you've not worked with binary before) and now to move on to the other principle of bit masking. To bit mask, we simply take a value and mask out the part that we want using a Boolean And operation.

The way the Boolean And operation works is that it compares the bits of the two values and only returns the subset of those bits that are set in both values, i.e.:

0 And 0 = 0  
 0 And 1 = 0  
 1 And 0 = 0  
 1 And 1 = 1

So when working with a full byte that process is just repeated for each bit within the bytes: 00110011 And 00001111 = 00000011

Now, the Long colour value is stored as:

00000000BBBBBBBBGGGGGGGGRRRRRRRR

Since these values are getting quite large, I'll now start to use Hex (base 16) rather than Binary (base 2) instead otherwise it will be unreadable. As such, this value can also be defined as:

00BBGGRR

Where:

R = Red

G = Green

B = Blue

0 = Not used

If you're unfamiliar with Hex then don't worry, for this section all you'll need to know is that Binary 00000000 == Hex 00 and Binary 11111111 == Hex FF.

So to get the red part only we need to mask out the lower 8 bits, for which we can use this mask:

000000FF

00BBGGRR And 000000FF = 000000RR = RR

Great, that's got Red sorted out so lets try the same thing with Green:

00BBGGRR And 0000FF00 = 0000GG00 = GG00

Unfortunately this has left us with some trailing zero's, which when converted to decimal change the value completely. As such we must first perform a bit shift right 8 places on the colour value, then mask it:

00BBGGRR >> 8 = 0000BBGG

0000BBGG And 000000FF = 000000GG = GG

And similarly with the blue channel, we must first bit shift it right 16 places and then mask it:

```
00BBGRR >> 16 = 000000BB
000000BB And 000000FF = 000000BB = BB
```

Wow, you'd never have thought converting a colour would be so difficult since they're so easy to make!

Putting this into VB code looks something like this:

```
Red = Colour And &HFF
Green = (Colour \ (2 ^ 8)) And &HFF
Blue = (Colour \ (2 ^ 16)) And &HFF
```

However, since the ^ operation is quite slow, we can speed to routine up by simply using the results of these calculations instead, and so we get this faster version:

```
Red = Colour And &HFF
Green = (Colour \ &H100) And &HFF
Blue = (Colour \ &H10000) And &HFF
```

Phew, two pages of text for those simple three line. Hopefully it made some sense though, if not then look it over when you're more familiar with Boolean and bitwise manipulation.

Ok, now we have the ability to extract these values, let's put it into practice in a simple demonstration to adjust the brightness of an image. If you haven't already then put a picture box and command button on the form, and put an image into the picture box (you don't have to, but it will be more interesting than just a flat colour.) Make sure the picture box's scale mode is set to pixels then double click the button to drop you into its click event handler.

We'll need a couple of loop variables to iterate through all the pixels in the image, the first loop to go from the left to right and the second loop within the first to go top to bottom, so declare two loop variables:

```
Dim ScanX As Long, ScanY As Long
```

Now, for each pixel we're going to need to extract the colour so we'll need a Long integer for that, and then we'll in turn need to break that down into 3 channel values so for that we'll use Short Integers (we could also use bytes, but they require more work as they overflow easily causing runtime errors when we adjust their values later):

```
Dim TempCol As Long
Dim cRed As Integer, cGreen As Integer, cBlue As Integer
```

Now we're ready to go, so start by creating the two loops:

```
For ScanX = 0 To Picture1.ScaleWidth - 1
  For ScanY = 0 To Picture1.ScaleHeight - 1
    Next ScanY
  Next ScanX
```

Ok, so far so good. Now we must get the colour at this pixel, we can use Point() for this:

```
TempCol = Picture1.Point(ScanX, ScanY)
```

And now we can use our new found bit manipulation knowledge to break this down into its source channel components:

```
cRed = TempCol And &HFF
cGreen = (TempCol \ &H100) And &HFF
cBlue = (TempCol \ &H10000) And &HFF
```

To brighten the colour all we need do is add a small amount to the existing colour value but we must also check to see that the resulting value doesn't get too big (overflow):

```
cRed = cRed + 50
If (cRed > &HFF) Then cRed = &HFF
```

You can do the same on the other two channels now too.

We must now pack these new values together into a long colour value, we'll use the RGB() function for this:

```
TempCol = RGB(cRed, cGreen, cBlue)
```

And finally draw it back to the picture box with PSet():

```
Picture1.PSet(ScanX, ScanY), TempCol
```

And that's all there is to it! If you run the application now and hit the button you'll either see the picture box updating gradually or it will freeze while the work is being done and only update the entire image once depending on if auto-redraw is enabled or not. You can force a redraw at any time by calling the picture box's Refresh method, a common place to put that is in the first loop either before or after the second loop so it only updates the picture box when a whole line is drawn.

Here's the final source code:

```
Private Sub Command1_Click()
    Dim ScanX As Long, ScanY As Long
    Dim TempCol As Long
    Dim cRed As Integer, cGreen As Integer, cBlue As Integer

    Picture1.ScaleMode = vbPixels
    Picture1.AutoRedraw = True

    For ScanX = 0 To Picture1.ScaleWidth - 1
        For ScanY = 0 To Picture1.ScaleHeight - 1
            TempCol = Picture1.Point(ScanX, ScanY)

            cRed = TempCol And &HFF
            cGreen = (TempCol \ &H100) And &HFF
            cBlue = (TempCol \ &H10000) And &HFF

            cRed = cRed + 50
            If (cRed > &HFF) Then cRed = &HFF
```



```
cGreen = cGreen + 50
If (cGreen > &HFF) Then cGreen = &HFF

cBlue = cBlue + 50
If (cBlue > &HFF) Then cBlue = &HFF

TempCol = RGB(cRed, cGreen, cBlue)
Picture1.PSet (ScanX, ScanY), TempCol
Next ScanY

Call Picture1.Refresh
Next ScanX
End Sub
```

At this point, we've covered the basics of graphics programming in VB and you now have all you'll need to do pretty much anything you want to. The problem with these methods though is that they are so slow and so we can turn to the Win32 API drawing methods to give us better performance, the next chapters will deal with converting your existing VB6 drawing code to use the API methods and hopefully get us better performance.

## Introduction to the API

---

Before I start going into any code, it's probably best to give you a brief explanation of some of the terminology and concepts behind the API.

What is the API?

API stands for "Application Programming Interface" and is basically a library of useful functions that sit behind Windows that we can tap into and use them from our own applications. To use one of these API functions we must first add a "Declare" to that function, telling our application where to find the function and what parameters it's expecting us to send it, much in the same way as each Function or Subroutine we create in VB has a header line with the list of parameters.

A VB Function header would look something like this:

```
Function MyFunc(ByVal inA As Long) As Long
```

Where as an API Function declaration looks like this:

```
Declare Function MyFunc Lib "SomeLib.dll" (ByVal inA As Long) As Long
```

As you can see there is not a huge amount of difference, it's just prefixed with the "Declare" keyword since it's an external function, and we have to give it a "Lib" or library name which is simply where the function is located so VB can find it.

Don't worry if this seems a little confusing, luckily there's an easy way to get these function declarations, Visual Studio (and I presume stand-alone versions of Visual Basic too?) is shipped with a little helper application called the API viewer. To use this application you can either launch it from the start menu, or in VB go to the Add-Ins menu and select "Add-In Manager". You should see a list of available Add-Ins and somewhere on there should be the "VB 6 API-Viewer". If not then luckily all is not lost - Head over to [www.AllAPI.net](http://www.AllAPI.net) and grab a copy of their API-Viewer application, which is pretty much an equivalent of the standard Microsoft one.

If it does appear on the menu however then select it and check the "Loaded/Unloaded" and "Load on Startup" options then hit Ok to close the dialogue. You should now see the API viewer in the Add-Ins menu, and clicking its icon will fire it up. Once in the application you'll need to locate your WIN32API.txt file which holds all the declares, constants and UDT's for the Win32 API. Select "File" -> "Load Text File" then find "WIN32API.txt". The viewer will take a second or two depending on the speed of your machine to parse the file and fill the list box, but you should see a list of the declares now in the list box. At this point, select "Load Last File" from the "View" menu, which will mean that this file will be opened automatically next time you load the application.

In the API-Viewer from AllAPI.net, you can do the same thing, just select "Win32api.apv" as the file then in "File" -> "Options" click the "Startup" tab and select "Load last opened file".

Hopefully now that should have demystified the API a little in case you were unsure of it earlier, and if not then after you've used it a few times you'll find that it's really not as bad or as difficult as you may have previously thought, just different.

In VB we are in an environment comprised of objects such as forms and picture box's that have their drawing functions tied to them through functions such as PSet, Line and Circle. When working with the API however, we're not just in VB indeed the API was actually written in C and are designed to function without and required UI. As such we have to tell them that we want them to draw on the controls and this is achieved through their hDC's. A DC (the h prefix simply indicates it's a handle or 'name' for the object) stands for Device Context but I'll not get into much detail about them there since I've written a whole other tutorial on the subject, which you can find elsewhere on this site if your still thirsty for information after this one. Think of the DC of a control as its "Drawing Canvas" (pun intended), which holds the image that we see on screen and most of the default controls have them (if not directly exposed).

Luckily for us, the hDC property of the picture box is exposed for us without having to do any extra work so it's immediately compatible with the API drawing functions.

## Migrating to API drawing

I'll first go through and show how each of the VB drawing methods are accomplished using the API then go on to show some more things that are possible with the API.

Before we get started though, it would be best to start off fresh but again drop a picture box and button on the form.

As we started with Pixels with the VB routines, I'll start off here with drawing pixels too. The API we use for drawing pixels is SetPixelV() but before we can use this call we'll need to copy the declare into our form. Open your API viewer and find the call, making sure the declare scope is "Private" then copy it to the clipboard buffer and paste it into the top of your form. All API declarations go at the very top of your code before any subroutines or functions, just to make sure the declaration should look something like this:

```
Private Declare Function SetPixelV Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal X As Long, ByVal Y As Long, _
    ByVal crColor As Long) As Long
```

Don't worry if yours looks different to this, I've just added some line breaks to mine so it doesn't get killed by the word-wrap on this window.

The call itself is little more complicated than the call to PSet() before:

```
' VB version
Picture1.PSet (10, 10), vbRed
```

```
' API version
Call SetPixelV(Picture1.hDC, 10, 10, vbRed)
```

There's very little that changes here so we'll now move on to drawing lines.

Here's where things get a little more complicated though, I've no idea why, but there is no simple API for drawing a line between two points, instead we have to use a combination of two API's to change the current point and drawing a line for there to our second position, the API calls we'll need here are MoveToEx() and LineTo():

```
Private Declare Function MoveToEx Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal X As Long, ByVal Y As Long, _
    ByVal lpPoint As PointAPI) As Long
```

```
Private Declare Function LineTo Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal X As Long, ByVal Y As Long) As Long
```

As you can see here though, the MoveToEx() declaration also includes one parameter as a POINTAPI, so what's that all about then? The API doesn't just have declarations to functions, but also to Types and Constants, and POINTAPI is one of these Types.

As such we'll need to go back into the API viewer and select Types from the drop down list then copy the type declaration into your code view beneath the three function declarations:

```
Private Type PointAPI
    X As Long
    Y As Long
End Type
```

Now we finally have all the declarations we can try them out. First off we'll need to declare a dummy point type for the MoveToEx() call (It's simply to hold the last current point, but we're not interested in what it was):

```
Dim DummyPt As PointAPI
```

Now we'll need to move the current point to the starting point of the line:

```
Call MoveToEx(Picture1.hDC, 5, 15, DummyPt)
```

And finally draw a line to the end point:

```
Call LineTo(Picture1.hDC, 25, 15)
```

Hmm, there was nowhere to put in a colour for the line in there and indeed if you run the application now you'll see a black line being drawn. So how do we draw in colour? This is where one of the biggest changes with migrating your code to use the API comes in, none of the API drawing routines save for the pixel drawing routines take an input colour as a parameter, instead they use something called a "Pen object" of the DC which specifies what colour, thickness and style the drawing operation uses. For the time being though we'll bypass this by simply using the properties of the picture box, one step at a time. We can do this since a picture box as a DC sitting behind it and it actually changes its Pen object when we change some of its properties, the API drawing routines will then take these into account when they draw on the picture box.

To demonstrate this, before you make the call to LineTo(), add this line:

```
Picture1.ForeColor = vbBlue
```

Great, we've got colours back. If you want to experiment with making your own pen objects rather than relying on the picture box then this is also covered in the DC tutorial on this site.

Unlike the VB line method we can't draw rectangles with the same call, for this we'll need to use a new API call, Rectangle():

```
Private Declare Function Rectangle Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal X1 As Long, ByVal Y1 As Long, _
    ByVal X2 As Long, ByVal Y2 As Long) As Long
```

So lets give this one a try:

```
Call Rectangle(Picture1.hDC, 5, 20, 25, 25)
```

Hmm, this is drawing in blue again since we set that before the previous call so we'll need to add another line to change the colour before calling rectangle:

```
Picture1.ForeColor = vbYellow
```

So far so good, but the next example we did back in the second chapter was to draw a filled rectangle, but how do we accomplish this using the API? In the same way that

the Pen objects control how the lines of shapes get drawn, the Brush objects control how they're filled but again we can just bypass this by using the picture box's properties instead:

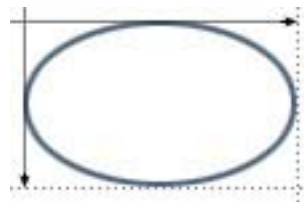
```
Picture1.ForeColor = vbMagenta
Picture1.FillColor = vbMagenta
Picture1.FillStyle = vbFSSolid
Call Rectangle(Picture1.hDC, 5, 30, 25, 35)
```

Again, brush objects are covered in the DC tutorial if you want to read up on them later.

When it comes to drawing circles, again there's a difference, this time in the coordinates we have to pass it. The VB circle drawing routine takes a single centre point and a radius as its parameters:



Where as the API uses a corner-to-corner method instead:



This makes it possible to draw ellipses as well as perfect circles and as such the call is Ellipse():

```
Private Declare Function Ellipse Lib "GDI32" ( _
    ByVal hDC As Long, ByVal X1 As Long, ByVal Y1 As Long, _
    ByVal X2 As Long, ByVal Y2 As Long) As Long
```

We can convert the centre-out to corner-to-corner coordinates using this simple calculation:

```
X1 = Centre.X - Radius
Y1 = Centre.Y - Radius
X2 = Centre.X + Radius
Y2 = Centre.Y + Radius
```

So to draw the original circle we can use this:

```
Picture1.ForeColor = vbGreen
Picture1.FillColor = vbCyan
Call Ellipse(Picture1.hDC, 30, 5, 60, 35)
```

Finally we need to convert the text drawing to the API, and here we have the choice of a number of API's each offering different things. For the sake of this tutorial though I'll just choose the easiest as it matches the one we were using in VB closest. To this end, grab the TextOut() API declaration from the API viewer and paste it into your code:

```
Private Declare Function TextOut Lib "GDI32.dll" ( _  
    ByVal hDC As Long, ByVal X As Long, ByVal Y As Long, _  
    ByVal lpString As String, ByVal nCount As Long) As Long
```

We'll need a string variable to hold the text we want to draw, so declare that now, and set it to the text we want to draw:

```
Const PrintText As String = "Look, some text!"
```

The call itself is no more difficult than the others, but unlike print we also have to specify the length of the string, this is simply because C deals with string variables in a different way than VB:

```
Call TextOut(Picture1.hDC, 2, 40, PrintText, Len(PrintText))
```

Again the text will be drawn with the font that picture box is currently set to, and its colour is set to the ForeColor property of the picture box. If you were writing a completely API version of this then the font is dealt with by using a Font object, like the Pen and Brush objects with the other routines and the text colour is defined with the SetTextColor() API call.

I won't go into dealing with images using the API since again this has been covered in the DIB tutorial, which you'll find elsewhere on this page.

## Customising drawing and image manipulation with the API

The API drawing methods use the Pen and Brush objects bound to the DC and we've seen that by altering some of the properties of the control, it will change the internal Pen and Brush object and thus the way the API call's will draw on the control. Below I've made a quick-convert table that shows how each of the properties are accomplished using just the API:

Property	API equivalent
DrawMode	SetROP2()
DrawStyle	Pen object's Style property
DrawWidth	Pen object's Width property
FillColor	Brush object's Colour property
FillStyle	Brush object's Hatch and Style properties
Font	Font object
ForeColor	Pen object's Colour property for Drawing, and SetTextColor() for Text.

The only one I've not really dealt with here so far is the SetROP2() API call which is declared as follows:

```
Private Declare Function SetROP2 Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal nDrawMode As Long) As Long
```

With the nDrawMode property being one of the following values:

```
Private Const R2_BLACK As Long = &H1
Private Const R2_NOTMERCYPEN As Long = &H2
Private Const R2_MASKNOTPEN As Long = &H3
Private Const R2_NOTCOPYPEN As Long = &H4
Private Const R2_MASKPENNOT As Long = &H5
Private Const R2_NOT As Long = &H6
Private Const R2_XORPEN As Long = &H7
Private Const R2_NOTMASKPEN As Long = &H8
Private Const R2_MASKPEN As Long = &H9
Private Const R2_NOTXORPEN As Long = &HA
Private Const R2_NOP As Long = &HB
Private Const R2_MERGENOTPEN As Long = &HC
Private Const R2_COPYPEN As Long = &HD
Private Const R2_MERGEENNOT As Long = &HE
Private Const R2_MERGEEN As Long = &HF
Private Const R2_WHITE As Long = &H10
```

As in the VB only section of this tutorial I mentioned that VB had a method to return the value of a pixel at a location and the API does too via the GetPixel() API call:

```
Private Declare Function GetPixel Lib "GDI32.dll" ( _
    ByVal hDC As Long, ByVal X As Long, ByVal Y As Long) As Long
```

This works in much the same way as Point() does in VB and so we can easily re-write the brightness adjustment application using the API methods instead by simply adding the function declarations to GetPixel() and SetPixelV() and changing two lines.



The first line we'll need to change is where the current colour of the pixel is returned with the `Point()` function, so change this:

```
TempCol = Picture1.Point(ScanX, ScanY)
```

To this:

```
TempCol = GetPixel(Picture1.hDC, ScanX, ScanY)
```

Then the other line to change is where we draw the altered pixel back to the picture box using the `PSet()` function, so change this:

```
Picture1.PSet (ScanX, ScanY), TempCol
```

To this:

```
Call SetPixelV(Picture1.hDC, ScanX, ScanY, TempCol)
```

It won't look much different when you run the application, but it should run a bit faster even when compiled. It's also worth mentioning at this point that most graphics code will perform better when compiled than from running in the IDE.

## System colours

---

As a quick side-note before I sign this tutorial off, I wanted to briefly mention system colours since they're quite useful to know and not that much different to use in VB than standard colours. Again they're sent around using the 32-bit Long integer, however unlike the other colour values they don't contain the RGB value of the colour, but an index in the lower byte. The constants used to reference these colours in VB are defined as:

```
vbScrollBars
vbDesktop
vbActiveTitleBar
vbInactiveTitleBar
vbMenuBar
vbWindowBackground
vbWindowFrame
vbMenuText
vbWindowText
vbTitleBarText
vbActiveBorder
vbInactiveBorder
vbApplicationWorkspace
vbHighlight
vbHighlightText
vbButtonFace
vbButtonShadow
vbGrayText
vbButtonText
vbInactiveCaptionText
vb3DHighlight
```

When using the VB drawing methods, these colours are automatically distinguished and converted to their RGB versions so we never have to worry about them. However when using the API, these values are not recognised and as such if you send them off to any of the drawing methods they'll all be rendered as very dull tones of red or black. The way we can distinguish between a standard RGB value and a system colour constant is by examining the high bit of the value, if this is set then the low byte is assumed to be an index and we can send this value off to the `GetSysColor()` API call to retrieve it's RGB value:

```
Private Declare Function GetSysColor Lib "User32.dll" ( _
    ByVal nIndex As Long) As Long
```

To make this easier for you, I've converted it to a function that will evaluate any colour and return it's RGB value be it an RGB value already or a system colour:

```
Private Function EvalCol(ByVal inCol As Long) As Long
    If ((inCol And &HFFFFFF00) = &H80000000) Then EvalCol = _
        GetSysColor(inCol And &H1F) Else EvalCol = inCol
End Function
```

If you were doing this completely using the API instead, you can use the following API constants for defining the colour rather than the VB ones:

## EDais graphics programming tutorial

```
Private Const COLOR_SCROLLBAR As Long = &H0
Private Const COLOR_BACKGROUND As Long = &H1
Private Const COLOR_ACTIVECAPTION As Long = &H2
Private Const COLOR_INACTIVECAPTION As Long = &H3
Private Const COLOR_MENU As Long = &H4
Private Const COLOR_WINDOW As Long = &H5
Private Const COLOR_WINDOWFRAME As Long = &H6
Private Const COLOR_MENUTEXT As Long = &H7
Private Const COLOR_WINDOWTEXT As Long = &H8
Private Const COLOR_CAPTIONTEXT As Long = &H9
Private Const COLOR_ACTIVEBORDER As Long = &HA
Private Const COLOR_INACTIVEBORDER As Long = &HB
Private Const COLOR_APPWORKSPACE As Long = &HC
Private Const COLOR_HIGHLIGHT As Long = &HD
Private Const COLOR_HIGHLIGHTTEXT As Long = &HE
Private Const COLOR_BTNFACE As Long = &HF
Private Const COLOR_BTNSHADOW As Long = &H10
Private Const COLOR_GRAYTEXT As Long = &H11
Private Const COLOR_BTNTEXT As Long = &H12
Private Const COLOR_INACTIVECAPTIONTEXT As Long = &H13
Private Const COLOR_BTNHIGHLIGHT As Long = &H14
```

The high byte is also sometimes used to store the alpha or opacity of the colour for images with per-pixel alpha, which method to use is up to the application and/or the situation.

## Further reading

---

As mentioned throughout the API section of this tutorial, a number of aspects of using the API drawing methods have been omitted since they've been covered in detail in other tutorials on this site.

If you've not worked with classes before then I suggest that you have a look at the "Basic introduction to classes" tutorial before tackling anything more with graphics since it's a somewhat core principle of VB and very useful especially when working with graphics. Indeed, both the more advanced graphics tutorials make extensive use of classes.

After that then it really depends what you're interested in pursuing, the DC tutorial is probably the logical next step since it has some overlap with this tutorial and covers how to take what you've learned so far and take it a step further, encapsulating it into a class object that means you don't have to deal with all the API calls every time you want to do some drawing but you still have access to their functionality and speed. Alternatively, the DIB tutorial will teach you how to add fast graphics manipulation code to your application and again encapsulates that into a class object for easy re-use.

On the other hand if your head is still spinning from all the information in this tutorial and for the time being you just want to get more experience with what you've learned so far, then you may want to check out some of the graphics libraries on this page, which offer some useful helper function that you can simply call.

For graphics manipulation the Pixel library will provide you with a lot of colour functionality and also makes converting between Long and RGB values incredibly simple.

The API Draw library encapsulates all the messy code to deal with the API pen and brush objects and allows you to draw a number of fully styled shapes using just a single line including line, rectangles, circles, polygons and even stars!

I hope this tutorial helped make graphics seem more approachable in VB and as always I'd appreciate any feedback good or bad that anyone has about this.