How to…
Write applications using Visual Basic

Last month, we provided the user with a way to select a drawing colour using the *lstColours* listbox. Now, we'll add the code that performs the actual drawing itself and you'll see how everything fits together.

Keep your options open

Before we can begin writing the drawing code, we need a way to determine exactly which shape is to be drawn. The user will click one of the option buttons in the control array *optShape* to indicate what shape is to be used. Our program will make a note of which shape the user chose and then consult this whenever any drawing needs to occur. Add the following code to the *(General) (Declarations)* section, right at the top of the form's code above *Form_Load*:

```
Private mlngChosenShape As Long
```

This module-level variable will hold the index of the currently-selected option button within the control array, reflecting which shape the user wants to draw with. Now we need to add some code to the option buttons to update *mlngChosenShape* whenever they are clicked. Double-click on any of the option buttons already on the form. Visual Basic responds by adding the following empty event handler:

```
Private Sub optShape_Click(Index As Integer)

End Sub
```

Notice that unlike the other *Click* event handlers you've encountered so far, this one has a variable passed to it called *Index*. This is because the option buttons are part of a control array. Since all controls within a control array share a common event handler, entering code here will serve all four option buttons. Refer to figure 1 to see this how this works. The *Index* variable is provided by VB to tell us which of the option buttons was clicked. Add the following code in the event handler you've just created:

```
mlngChosenShape = Index
```

This stores the index of the option button that was clicked into our module-level variable, *mlngChosenShape* so that we can refer to it later.

Square peg in a round hole?

The more observant amongst you might have noticed that *Index* is an *Integer* whilst *mlngChosenShape* is a *Long*. Assigning an *Integer* value to a *Long* variable is fine; they are both numerical types. Variables declared as *Long* can hold larger numbers than

Integer variables, so it is perfectly valid to assign an *Integer* value to a variable declared as *Long*. The reverse is not necessarily true however; you'd run into problems assigning a *Long* value to an *Integer* variable if the *Long* value was too large for the *Integer* variable to hold. See the box entitled "32-bit power" elsewhere in this article for a description of *Long* and *Integer* variables.

Graphics, VB style

Now it's time to add the drawing code. Double-click on *picDrawingArea*, choose the *MouseDown* event from the right-hand combo box, and enter the following code:

```
Dim lngColour As Long

lngColour = lstColours.ItemData(lstColours.ListIndex)

Select Case optShape(mlngChosenShape).Caption
   Case "Circle"
       picDrawingArea.Circle (X, Y), 300, lngColour
   Case "Square"
       picDrawingArea.Line (X - 300, Y - 300)-(X + 300,
Y + 300), lngColour, B
   Case "Horizontal Line"
       picDrawingArea.Line (X - 300, Y)-(X + 300, Y),
lngColour
   Case "Vertical Line"
       picDrawingArea.Line (X, Y - 300)-(X, Y + 300),
lngColour
   Case Else
       MsgBox "Encountered unexpected shape """ &
optShape(mlngChosenShape).Caption & """",
vbExclamation, "Warning"
End Select
```

This does a variety of things. First of all, it works out which colour is to be used by examining the *ItemData* property of this currently-selected item in the list of colours. Then, it examines the *Caption* property of the option button that was most recently chosen and decides which shape to draw as appropriate.

Readers accustomed to older versions of Basic might not recognise the *Select Case…End Select* block. These two statements provide an easy way of repeatedly comparing an expression against a known set of values. VB compares each "Case" value given, to the value at the top of the *Select Case* block. If they match, then the code immediately following the *Case* statement that matched is executed until the next *Case* statement is encountered, at which point, VB will drop out of the *Select Case…End Select* block and continue running the code below it.

Prepare for the unexpected

There is a special case (pun intended) – the *Case Else* clause. This is optional; if provided, it tells VB what to do if no match can be found. I have included a *Case Else* in the routine above as a safety feature. The shape names in the *Case "Circle"* et al must exactly match the *Caption* properties of the option buttons. If they differed in the slightest way, no match will be made and nothing would be drawn. However, the program would continue as if nothing had gone wrong, bar the fact that nothing would be drawn. Therefore, I've added some extra code in here to alert you to the fact that something is amiss should this ever happen. Of course, the program as it is printed here is fine, but if you are typing this in rather than loading the project files from the cover CD, the possibility exists for you to mistype something, hence the extra code to deal with this situation.

I recommend that you always check for unexpected eventualities such as this happening in your programs. The little effort involved will pay off later if something goes wrong in your program, since you'll get to know about the problem straight away.

More Graphics

Examine the statements that perform the actual drawing. Notice that graphical operations are implemented in Visual Basic as methods of some drawing surface, in this case, *picDrawingArea*. You can't just say "Circle" or "Line" as in older versions of BASIC because we are working in a windowed environment and we have to say which window or control we want to draw on. In addition, the Y axis in VB goes from top to bottom, i.e. larger values move you further down the screen, so the mathematicians or Sinclair Spectrum programmers amongst you will have to think upside down. For more information, see Figure 2.

The *Circle* method is fairly straightforward, it takes at least three parameters; the first and second parameters give the centre co-ordinates of the circle and the third specifies the radius of the circle to be drawn. I say "at least" three parameters because some functions in Visual Basic have optional arguments. This means that if you don't specify a value, Visual Basic supplies its own default value instead. The rest of the arguments for the *Circle* method are optional. If a fourth argument is given (as we have done), VB takes this to mean the colour that we want to draw with. Had we left this out, the default pen colour would have been used instead.

The *Line* method is a little less obvious, you specify two sets of co-ordinates specifying the starting and ending co-ordinates of the line respectively. As with the *Circle* method, you can specify the colour to be used as I have done. Take note of the strange "B" option used when drawing a box. This tells Visual Basic that we want to draw a **B**ox instead of a line and so it takes the co-ordinates to mean the upper-left and lower right-hand corners of the box to be drawn. Using *BF* instead of *B* will draw a **B**ox that is **F**illed. AmigaBasic programmers will have encountered this syntax before since Microsoft worked on AmigaBasic as well.

Okay, that's enough theory. Run the program and you should find that you can draw within the drawing area by clicking in it, changing shapes and colours as you see fit. However, there are two things missing from this program as it stands:

i) The Clear button has no effect.
ii) Continuous drawing doesn't work – if you click the checkbox entitled "Continuous Drawing", you should be able to drag inside the drawing area and draw a continuous trail.

We need to add the code to support these two facilities. End your program, double-click on *cmdClear* and enter the line:

```
picDrawingArea.Cls
```

That simply clears the drawing area. Incidentally, the *Cls* method is so-named because it is a hangover from older versions of BASIC such as that used on the Sinclair Spectrum. It used to stand for **CL**ear **S**creen, but clearing the entire screen wouldn't make much sense in a windowed environment since you'd only want to clear a single window.

In Closing

That's all for this month – as usual, you can find the project files that accompany this tutorial on the cover disc. Next month, we'll add the code to support the continuous drawing. In the meantime, why not have a go yourself?

Hint : You'll want to draw the relevant shape when the mouse moves provided that the check box is checked and the left mouse button is being held down. You'll have to copy the drawing code to another event handler for this to work. Incidentally, copying code in this way isn't good programming practice so I'll show you a more efficient way to achieve this next month.

Good luck,
Nick.

Nicholas Scott is a freelance columnist who currently works for MIS Computer Services in Northwich. Nick can be contacted via email at nicks@miscs.com.

32-bit Power

What is the difference between *Long* and *Integer* variables? *Integer* variables are really 16-bit numbers and therefore, hold whole (i.e. non-decimal) numbers in the range -32,768 to 32,767. *Long* values are like *Integers*, except that they are 32-bit numbers and can hold numbers ranging from -2,147,483,648 to 2,147,483,647.

You may wonder why I use *Long* variables as opposed to *Integers* if I don't need the larger range of values offered by the *Long* datatype. After all, a variable declared as a *Long* will take up twice the space than the equivalent . *Integer* and you'd expect it to be twice as slow. Well, the answer is that the 32-bit processor driving your PC is able to deal with 32-bit values faster than 16-bit values, so using *Long* variables instead will actually make your programs faster

(ED: The filename for this image is SamplePic.BMP)

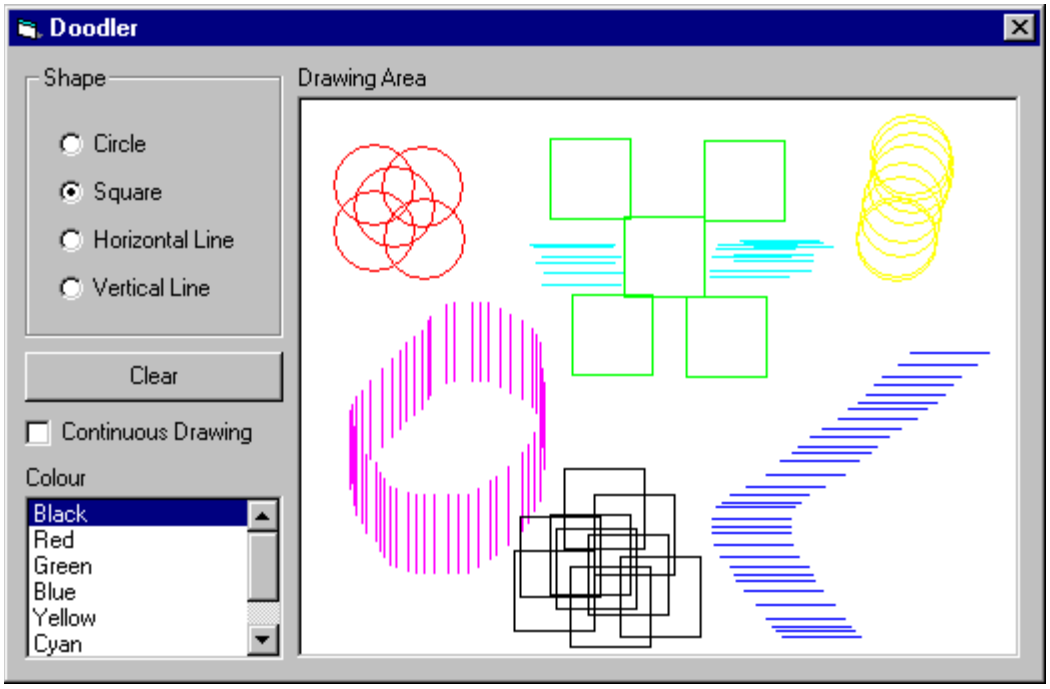
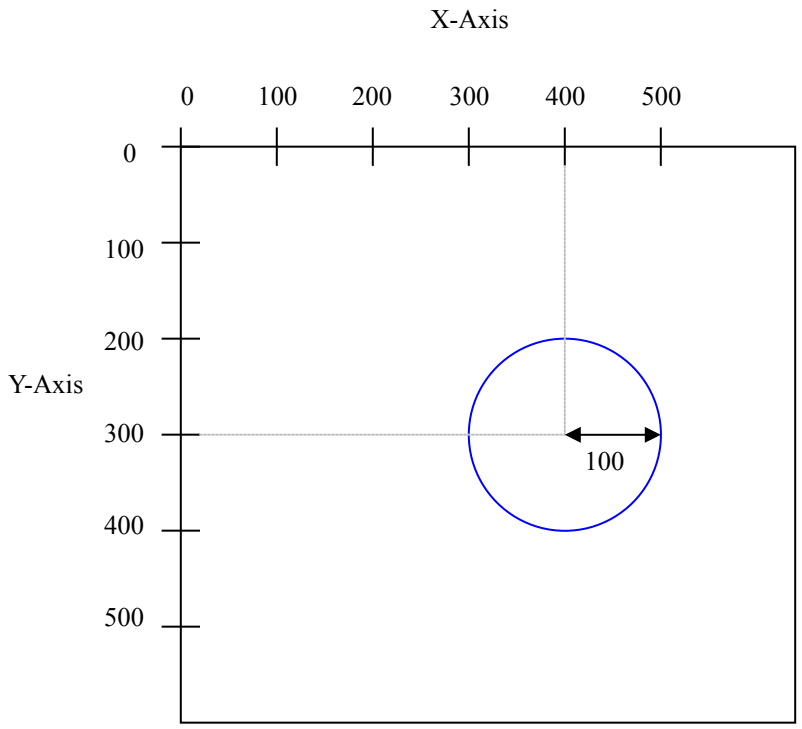


Figure 3 - The Doodler program in action.

Figure 2 – Using Visual Basic's Graphics Methods

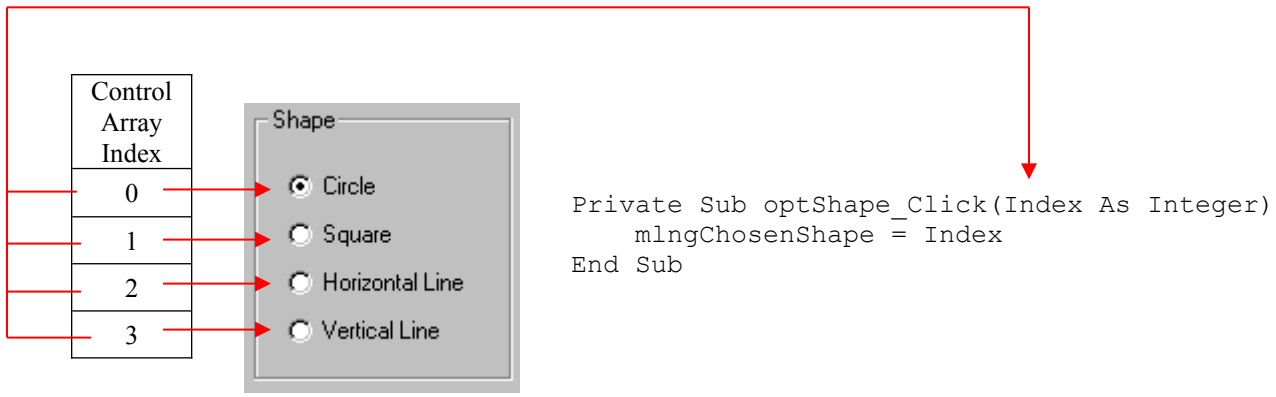


This blue circle is drawn using the statement:

```
picDrawingArea.Circle (400, 300), 100, RGB(0, 0, 255)
```

(ED: The filename for this image is ControlArray.bmp)

Figure 1 – Providing centralised event management using a control array



All the controls in a control array share the same event handler for each type of event. This provides a useful mechanism for sharing event-handling code that should be common to a number of similar controls. If you make any changes to the code in the event handler, all the controls within the control array will benefit from the changes that you make.