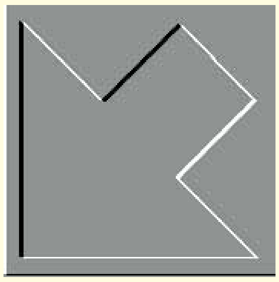


It's time for **take off**

Take a Stealth bomber, apply a working Visual Basic program, and watch it move. Mike Liardet shows you how to make it pitch, roll, yaw, and spin, and explains the important technique of perspective projection.

In last month's column, I demonstrated how some simple matrix algebra operations could be used to twist, turn, move and stretch a computer representation of a three-dimensional object. But these manipulations would be of little interest to anyone unless they could actually be seen; so I also reviewed how a 3D computer representation might be projected onto a 2D screen, enabling the user to actually see a picture of what is happening. So much for the theory — this month we are going to tie up the project with a practical, working Visual Basic program.

Fig 1 shows the program in action. There are three main sets of command buttons held in the three frames labelled "Rotations", "Sizes" and "Moves". Each of the buttons in these frames applies a matrix operation to an underlying data structure. Following this, the image is redrawn from the data structure with its new position or orientation, thus appearing to the user as if he has directly manipulated the object in the desired fashion. The Large and Small movement options determine the amount of movement for each of the command buttons. The "Image Trace On" option button simply prevents the screen being cleared before a new image is drawn, so that the history of previous positions can be seen. "Reset" puts the object back into the middle of the display area at a reasonable distance and in a standard orientation. "Exit" has the usual effect.

The shape which is manipulated by all these operations is supposed to be a Stealth bomber, but it is admittedly rather

1

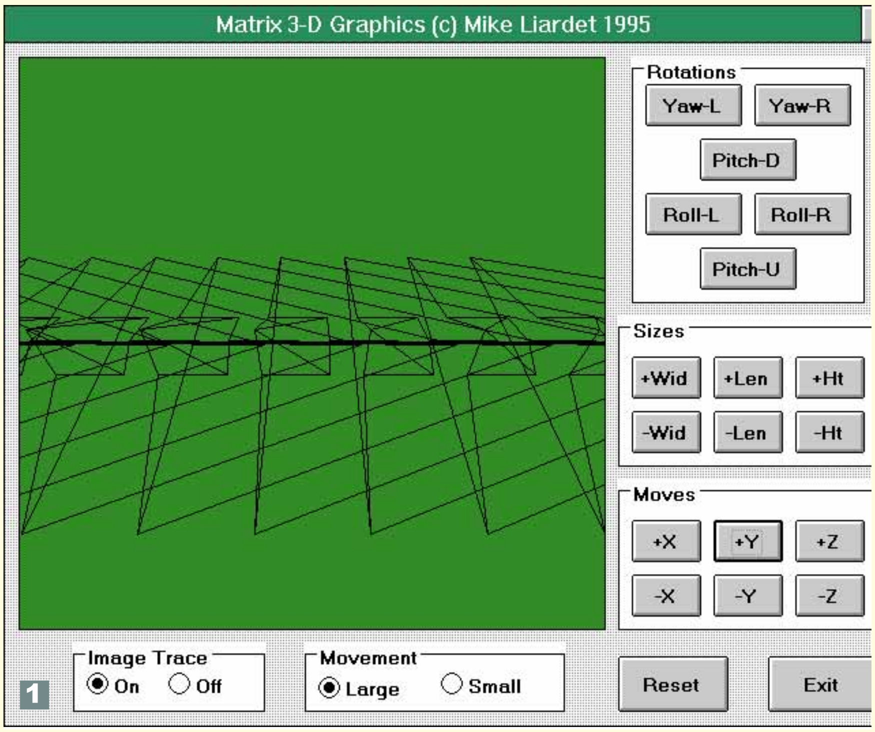


Image Trace: [✓ On] Off
Movement: [✓ Large] Small

2

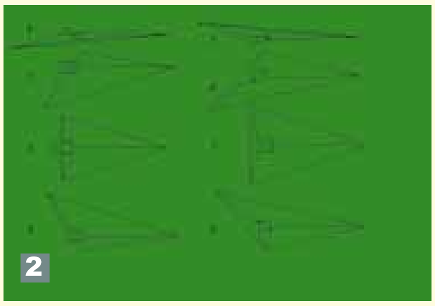


Fig 1 *The 3D graphics program in action. The shape that is flashing across the display area is supposed to be a Stealth bomber, but is unlikely to fool the discerning PC user*

Fig 2 *The effect of the "Roll-R" command button: the shape rotates about its longitudinal access, eventually moving through 360 degrees and returning to its original position*

crude in appearance and somewhat closer to a paper dart than any real aeroplane. It is not too difficult to modify the program to create a more realistic shape, but it is convenient to have a simple shape for the initial development work: it can be drawn quickly and it's reasonably easy to see what is happening with it when testing the code.

The display in *Fig 1* was generated by using various command buttons to achieve the shape with the desired orientation and size, and positioned at the far left of the screen. At this point, the Image Trace option was selected along with Large Movements. Repeated presses of the Moves "+Y" button caused the shape to "fly" across the display area from left to right in several large steps, each time leaving its trace behind.

Fig 1 shows the effect of just one of the command buttons but there are many other possibilities. *Fig 2* shows what happens when the "Roll-R" button is used. This causes the shape to perform what is, in effect, an "aileron roll" to the right. At each press of the button, the shape rotates a little further about its longitudinal access, eventually moving through 360 degrees and returning to its original position.

All the other rotations are possible as well, using the other rotation command buttons. The Pitch buttons cause the shape to rotate about its lateral axis. From normal horizontal flight the "Pitch Up" command makes the shape's nose point up and "Pitch down" makes it point down. The yaw buttons rotate the shape about its normal axis causing it to point to the left or right as appropriate. In a real plane, this sort of movement would be made by the rudder (which is controlled by the pilot's feet on the pedals). The pitching and rolling rotations are effected by pushing, pulling or turning the control column (or joystick) in the pilot's hands.

It's also possible to change the shape of the object — something that is impossible in a real aeroplane. *Fig 3* shows what can happen if the "Sizes" buttons are used. The images numbered 1 to 4 were produced by repeatedly using the "+Len" button. Images 5 and 6 show the shape getting wider (the "+Wid" button). Finally, the shape is made taller with the aid of the "+Ht" button.

The shape can be moved anywhere within the three-dimensional space that it occupies by using the "Moves" buttons. We have already seen it moving from left to right (the Y direction). Up and down motion is possible using the "+Z" or "-Z" buttons. *Fig 4* shows the shape moving in

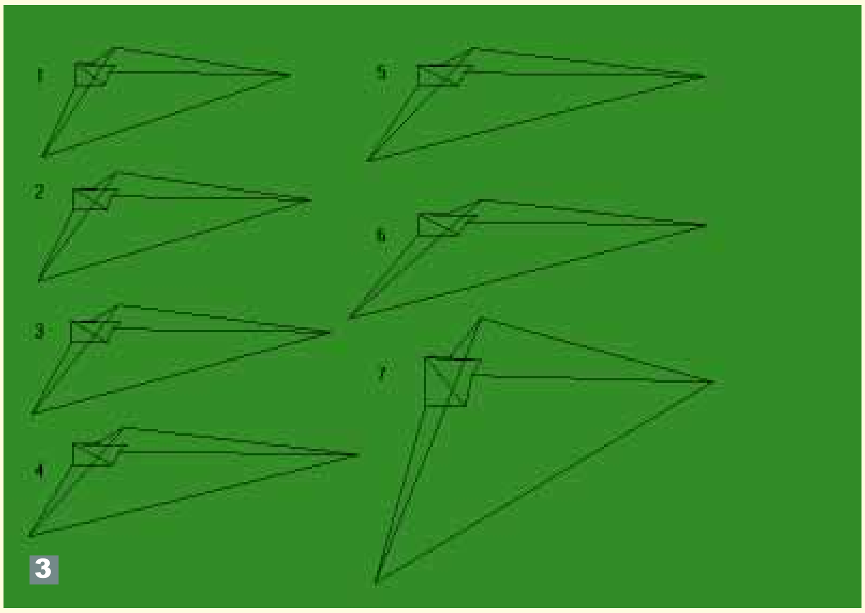


Fig 3 *Unlike a real aeroplane, this one can change its shape: images 1 to 4 show it getting longer; then in 5 and 6 it becomes wider; finally, image 7 shows it suddenly having grown a bit taller*

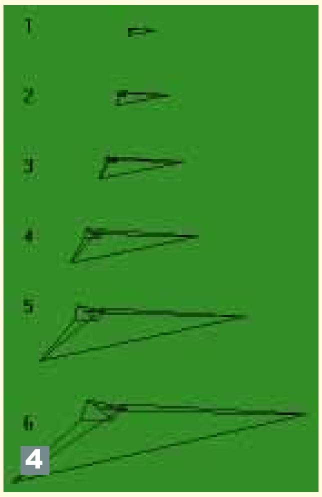
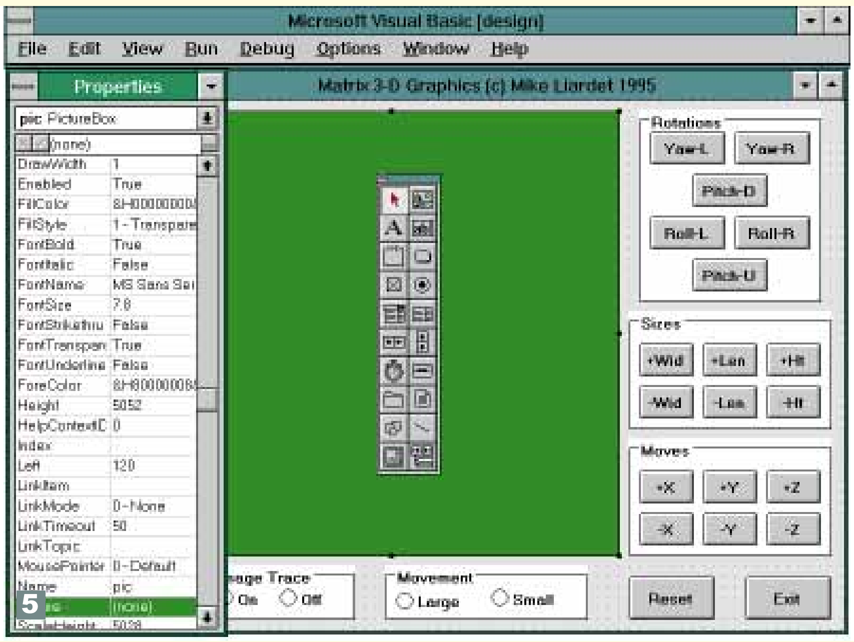


Fig 4 *The effect of the "+X" command button: this brings the shape nearer and nearer to the viewer, with a larger image at each stage*

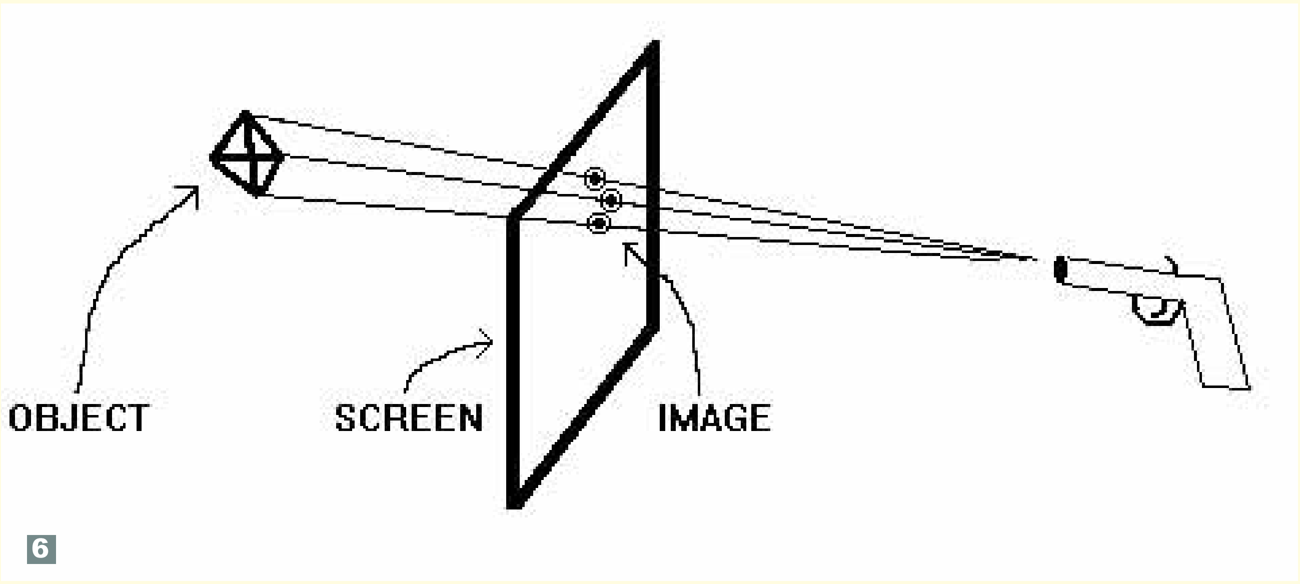
Fig 5 *Form design for the program: there is only one form, with no special hidden tricks behind the scenes other than the control arrays which are used to simplify the processing of most of the command buttons*



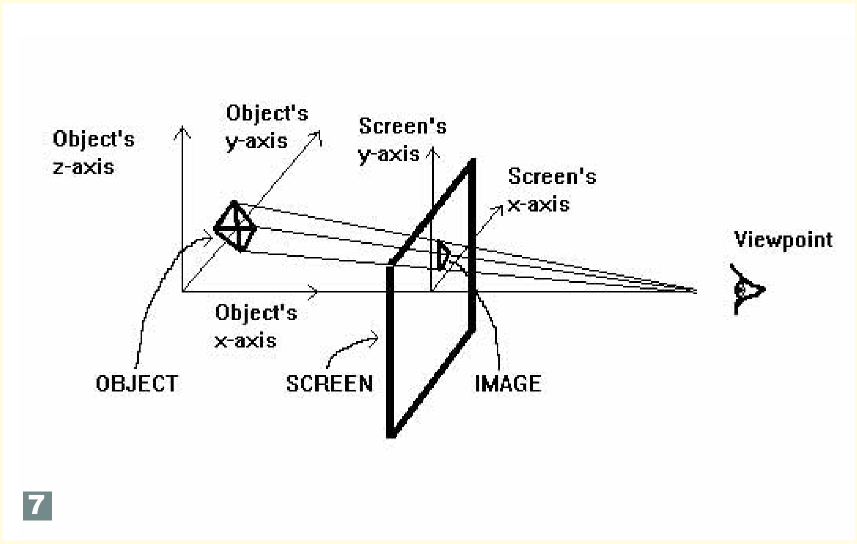
pic PictureBox

(none)

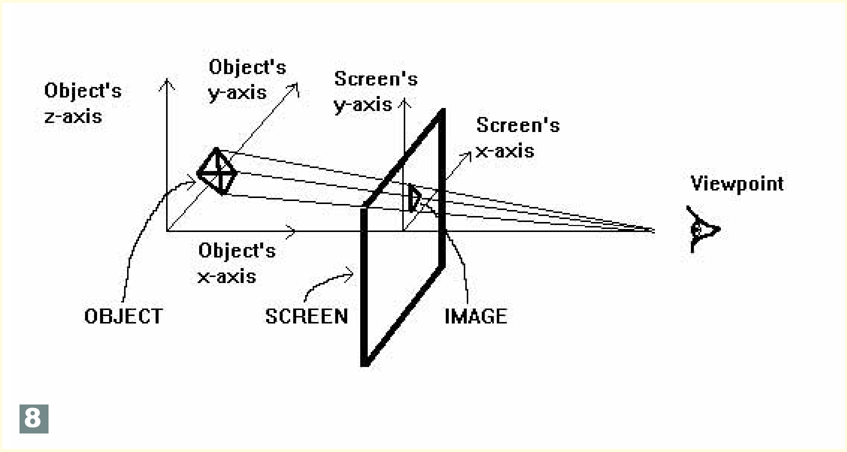
DrawWidth	1
Enabled	True
FillColor	&H0000000(
FillStyle	1 - Transpare
FontBold	True
FontItalic	False
FontName	MS Sans Se
FontSize	7.8
FontStrikethru	False
FontTranspar	True
FontUnderline	False
ForeColor	&H8000000(
Height	5052
HelpContextID	0
Index	
Left	120
LinkItem	
LinkMode	0 - None
LinkTimeout	50
LinkTopic	
MousePointer	0 - Default
Name	pic
[illegible]	(none)
ScaleHeight	5028



6



7



8

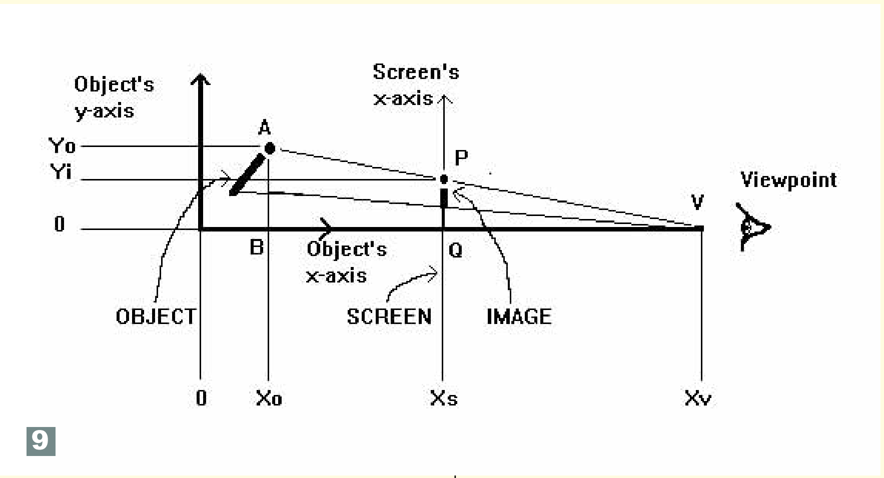
Fig 6 *How to produce a perspective projection: simply shoot through a screen at prominent features on the object and the bullet holes will create the drawing for you!* **Fig 7** *Putting the perspective projection on a less destructive and more mathematical footing.* **Fig 8** *Looking straight down on the scene in Fig 7, from a position on the object's negative Z axis. This view shows how the (horizontal) X co-ordinate of the perspective projection is calculated*

the X direction, which moves it towards the viewing window. At first (image 1), it is at some distance from the window, and it therefore appears to be fairly small. Repeatedly pressing the "+X" button gradually brings it nearer, and consequently it appears larger.

If the button is pressed for long enough, the shape eventually fills the entire screen as it gets very near the window. Ultimately it will crash into the window, ironically crashing the system at the same time. This is caused by an arithmetic overflow error resulting from the enormous co-ordinate values that are generated as part of the calculation. Visual Basic ought to fail gracefully when this happens, with an appropriate error message, but on the development machine (a 486 with numeric processor) it simply falls over completely, sometimes necessitating a reboot. There is a fairly simple cure for this problem: do not let the shape get too near the window, for instance, by disabling the "+X" button when the shape's X co-ordinate exceeds a certain value.

Perspective projection

Three-dimensional objects can be shown on the screen using plan and elevational drawings, or with axonometric and isometric projections. These techniques are primarily used for technical drawings, and in architecture. They are useful because they preserve much of the original 3D information in the 2D image, including many of the angles and lengths. Unfortunately, they do this at the expense of realism, and for recreational purposes the perspective projection is



9

Fig 9 *Looking from the side of the scene in Fig 7, from a position on the object's negative Y axis. Here we see how the (vertical) Y co-ordinate of the perspective projection is calculated*

preferred. This is the technique used in the program presented here, and since it is so important we shall take a close look at the way in which it works.

Perspective projection provides a view of an object which is identical to the view that might be obtained on photographic film, using a camera. Although art teachers generally teach perspective drawing in terms of parallel lines meeting at a single point, using illustrations of long, straight, railway tracks converging as they near the horizon, this methodology is only necessary to help human artists get their perspectives right. Computer-generated perspectives are not built in this way.

Fig 6 shows the general principle behind a computer-generated perspective. Imagine an object (in this instance a tetrahedron, or four-faced "pyramid") lying behind a transparent screen, with a gun pointing at it from the other side. Imagine firing the gun through that screen at a prominent feature in the object: the bullet hole in the screen marks the point where that feature is to be located in a perspective projection of the object. But if the bullet misses the screen altogether, then this is an indication that part of the object would not normally be visible through the screen "window". In *Fig 6*, the whole object can be seen in the top right quarter of the screen and the three bullet holes represent the location of three of the tetrahedron's vertices.

To put the method onto a more mathematical footing we must include axes for the object and for the screen *(Fig 7)*. The object is located in 3D space relative to its X, Y and Z axes, and its 2D image is drawn relative to the screen's X and Y axes. The gun is replaced by an eyeball "viewpoint", and the bullet paths are replaced by straight lines drawn from the viewpoint to points on the chosen object.

Notice that the screen axes are parallel to two of the object's axes and that the third object axis (the X axis) is perpendicular to the screen, hitting it right in the middle and then continuing on to the viewpoint. The precise location and orientation of these axes is arbitrary. For example, we could have an angled line from the object's origin going through the screen's mid-point, and we need not even have the screen perpendicular to it (although this would produce distorted images). I have set up the axes in this way simply to make the maths as easy as possible.

Fig 8 shows the same scene as *Fig 7* but in just two dimensions; looking directly down from the direction of the object's Z axis. It shows how to calculate the horizontal co-ordinate for the projection. We see that one of the vertices (labelled A) in the original object has co-ordinates (Xo, Yo) (its Z co-ordinate is irrelevant here). The screen and viewpoint are located at a distance of Xs and Xv respectively from the object origin. The vertex A is projected onto the screen at point P, and for the horizontal co-ordinate of the projection we need to determine the value Yi for this point in terms of the other values.

The calculation turns out to be fairly easy. The gradient of the line VP is the same as the gradient of VA, so we have:

```
AB/BV = PQ/QV
```

Plugging in the co-ordinate values of the points A, B, P and so on, we get:

```
Yo/(Xv-Xo) = Yi/(Xv-Xs)
```

Some simple rearrangement of this equation gives us:

```
Yi = Yo*(Xv-Xs)/(Xv-Xo)
```

which is the horizontal value (on the screen) for the perspective projection. *Fig 9* shows how to work out the vertical screen value. A similar line of reasoning produces:

`Zi = Zo*(Xv-Xs)/(Xv-Xo)`

When programming the projection we have to decide on appropriate values for the screen position and viewpoint (the values Xv and Xs). The program contains reasonable values for these variables, hard-coded as constants, but there is nothing absolute about them. It would even be possible to arrange for these values to change dynamically if that was appropriate to the application.

It's important to keep the object, screen and viewpoint in the right order, as strange effects will be produced if the object and viewpoint ever end up on the same side of the screen. Assuming they are in the right order, then the general rule is that the further away the screen is from the object, the smaller that object will appear. This corresponds to the real life observation that distant objects look small.

On the other hand, moving the viewpoint closer to the screen allows more of the scene to be viewed, thereby corresponding to the real life observation that one can see more out of a window if one stands close up to it. Moving the viewpoint can also be thought of as being like the operation of a zoom-lens on a camera: when the viewpoint is close to the screen one gets a wide-angle view, and when it is distant it becomes like a telephoto image. Theoretically, we can allow the viewpoint to go all the way back to +infinity, and with the aid of some mathematical chicanery with the aforementioned calculations, the perspective projection then turns into a simple elevational view.

There are plenty of possibilities for improving the program presented here. There is no validation of user input so it is comparatively easy to crash the program, for example by flying the shape slap into the viewing screen. It is also easy to lose track of the shape altogether by sending it out of sight of the screen. Visual Basic itself handles the clipping of the image as the shape moves out of range, so this is not an error condition as such, but implementing "Wraparound" space could help here, so that the shape reappears at the left having gone off the right-hand side and reappears at the bottom, having vanished from the top.

It is a fairly straightforward matter to set up a more interesting object by modifying the code in the routine PolyObjectMake. Indeed there is no reason why several objects should not be set up and dealt with simultaneously. This is not very difficult to program, but of course the more lines, faces and vertices in a scene, the longer it takes to draw. So if the scene gets too complicated, it may not be possible to see it without frustratingly long delays.

When designing a new object it is a good idea to first sketch it fairly accurately on paper, labelling the vertices and deciding on their co-ordinate values. For very complex objects it may be worthwhile implementing some "service" routines that can set up subcomponents of the object in one go. For example, if the object contains a number of cube shapes, then it might be useful to have a subroutine that can create the vertices, edges and faces of a cube, located in any position.

Curved surfaces can be approximated by a number of flat faces and these are best built up by using special purpose routines rather than by trying to work out the co-ordinates by hand. For instance, a circular cylindrical shape might be set up as a hexagonal (or decagonal, etc) cylinder and it is a fairly straightforward matter to let the computer calculate the vertex co-ordinates for these approximations.

The program does not perform any hidden-line removal so it always displays all the lines in the object, whether or not they would normally be visible from that angle of view. With more complicated objects this can be confusing. It is sur-

Fig 10 Visual Basic Code from the MATRIX.FRM module

```
Option Explicit
Dim poly As PolyXD

Sub cmdExit_Click ()
  End
End Sub

Sub cmdReset_Click ()
  ‘Make a 1000 by 1000 by 1000 cube centred on origin
    ‘CuboidMake poly, 1000, 1000, 1000
    PolyObjectMake poly
  ‘Set pic with 4000 x 4000 area and origin in middle
    optOff = True
    optSmall = True
    pic.Scale (-2000, 2000)-(2000, -2000)
    PolyDraw poly, pic, optOff
End Sub
Sub cmdRot_Click (Index As Integer)
Dim ang As Single
Static r(3, 3) As Single
    If optSmall Then
        ang = pi / 8
    Else
        ang = pi / 4
    End If
    Select Case Index
    Case AXIS_X
        MatRotate r(), ang, AXIS_X
    Case AXIS_Y
        MatRotate r(), ang, AXIS_Y
    Case AXIS_Z
        MatRotate r(), ang, AXIS_Z
    Case AXIS_MX
        MatRotate r(), -ang, AXIS_X
    Case AXIS_MY
        MatRotate r(), -ang, AXIS_Y
    Case AXIS_MZ
        MatRotate r(), -ang, AXIS_Z
    End Select
    MatApply r(), poly
    PolyDraw poly, pic, optOff
End Sub
```

Fig 11 Visual Basic code from MATRIX.BAS module

```
Option Explicit

Const MAX_DIM = 4  'allows for homogeneous coords
Const LAST_DIM = MAX_DIM - 1
Const MAX_EDGES = 1000
Const LAST_EDGE = MAX_EDGES - 1
Const MAX_VERTS = 200
Const LAST_VERT = MAX_VERTS - 1

'ID for rotations/translations, etc about the various
axes
Global Const AXIS_X = 0
Global Const AXIS_Y = 1
Global Const AXIS_Z = 2
Global Const AXIS_MX = 3
Global Const AXIS_MY = 4
Global Const AXIS_MZ = 5

Global Const PI = 3.141593

Type vertex
  coord(LAST_DIM) As Single
  xProj As Single
  yProj As Single
End Type

Type edge
  pt1 As Integer 'index to start point of edge
  pt2 As Integer 'index of end point of edge
  partof As Integer  'ID of part that it belongs to
  mark As Integer  'used in edge elimination
End Type

Type polyXD
    numDims As Integer   'number of dimensions in use
    numParts As Integer  'number of parts in poly
    numVerts As Integer
    vert(LAST_VERT) As vertex
    numEdges As Integer
    edge(LAST_EDGE) As edge
End Type

Global picBox As Control

Sub CuboidMake (poly As polyXD, x As Single, y As Single,
z As Single)
'Make a Cuboid centred on the origin, sized x by y by z

  'Start with an x by y rectangle in the xy plane
    poly.numDims = 2
  'poly has four 'parts' (the four edges)
    poly.numParts = 4
  'Set up the four vertices
    poly.numVerts = 4
    poly.vert(0).coord(0) = -x / 2
    poly.vert(0).coord(1) = -y / 2
    poly.vert(1).coord(0) = x / 2
    poly.vert(1).coord(1) = -y / 2
    poly.vert(2).coord(0) = x / 2
    poly.vert(2).coord(1) = y / 2
    poly.vert(3).coord(0) = -x / 2
    poly.vert(3).coord(1) = y / 2
  'Set up the four edges
    poly.numEdges = 4
    poly.edge(0).pt1 = 0
    poly.edge(0).pt2 = 1
    poly.edge(0).partof = 0
    poly.edge(1).pt1 = 1
    poly.edge(1).pt2 = 2
    poly.edge(1).partof = 1
    poly.edge(2).pt1 = 2
    poly.edge(2).pt2 = 3
    poly.edge(2).partof = 2
    poly.edge(3).pt1 = 3
    poly.edge(3).pt2 = 0
    poly.edge(3).partof = 3
  'Add the third dimension
    DimensionAdd poly, -z / 2, z / 2
  'Change verts to homogeneous coords
    HomogCoords poly
End Sub
```

prisingly difficult to implement hidden-line removal, and it is computationally expensive, so any implementation is likely to be slow with a semi-interpreted language like Visual Basic.

Adventurous souls are welcome to try the hidden-line problem. It's a good idea to make all the components in the scene convex (that is, with no cavities) as this makes the problem slightly easier. Concave objects can always be broken up into two or more convex objects (for example, the single polyhedron shape in the program here could be made convex if the tail and main body were treated separately). Once there is hidden line capability in place, the object can be made more interesting with some clever use of Windows APIs (Application Programmer Interfaces) which allow the faces to be filled with colour, rather than displaying a simple outline drawing, as at present.

As it stands, the program makes a rather poor flight simulator and with some mathematical effort could be made to work more like a real flying aeroplane. The intention here is to simulate what is in effect a radio-controlled plane, since the view in the window is of the plane rather than from the plane. Some extra effort is needed with the matrix transformations so that the plane always rotates about its own axes rather than the invisible, fixed, co-ordinate axes.

Once this has been done, it is only necessary to move the plane in the direction in which it is pointing. Obviously, real planes (Harriers excepted) don't fly sideways, backwards or vertically under normal flying conditions; so this plane could be made to simply follow its nose, possibly in conjunction with a throttle, to determine how much it moves after each redraw.

PCW Cover Disk

The full code for this month's Low Level is on the cover disk given with this issue of *PCW*.

PCW *Contacts*

Mike Liardet is a freelance programmer and writer. He can be contacted via the *PCW* editorial office or on email as **mliardet@cix.compulink.co.uk**