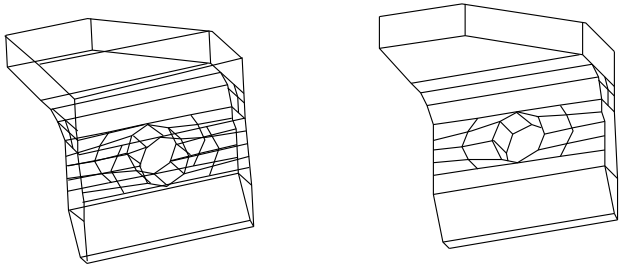
Chapter 3 **Three-dimensional graphics**

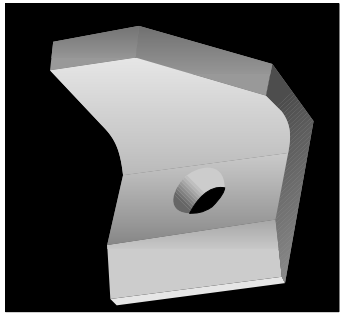
This chapter looks at handling three-dimensional solids in computer graphics systems. Four important aspects are dealt with. First of all we look at the mathematical techniques for transforming (enlarging, rotating etc.) such models and we shall pave the way for this by looking at analogous operations on two-dimensional models. Secondly we consider how to display such solid bodies on a two-dimensional display or screen system. Thirdly we shall look at how models can be generated mathematically, that is how we can specify the three-dimensional shape of, for example, a cylinder to the computer. Finally we shall dip gently into a classic problem in computer graphics - hidden line removal.

This chapter contains a good deal of fairly straightforward mathematics involving matrix manipulation (mainly multiplication). A lot of the mathematical detail is in Appendices 4 and 5. The transformation procedures, particularly those for creating a screen image from a three-dimensional object, can easily be used as a tool without understanding how they were derived. The procedures used in this chapter will give you most of the tools you require to delve into three-dimensional computer graphics. If, however, you wish to develop your own software, or alter the given procedures, then obviously an understanding of the mathematics is desirable.

Currently there are two main methods for representing three-dimensional solids on computer graphics display devices. The first is the wire frame model. Here the model is defined as a list of vertices together with connectivity information. A two-dimensional representation of this model as seen from a particular viewpoint can then be drawn by using line graphics to join the vertices together. Hidden line removal may also be employed in such models. A more elaborate and more realistic display involves surface modelling. This is much closer to visual reality and here an attempt is made to create an image on the screen rather than a vertex model. Hidden surfaces are removed in the representation and visible surfaces are shaded, taking into account such factors as assumed position of the light source, surface textural characteristics that define reflectivity, etc. Colour graphics is used in such modelling and the end result can be uncannily real.



Courtesy Boeing Corporation



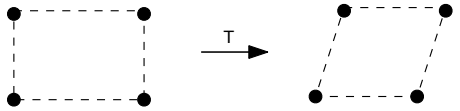
Now with the BBC micro we are limited by practical considerations to wire frame models. In MODE 0 even although the spatial resolution (640x256) is reasonable, each pixel at this resolution is 1 bit in the screen memory. This can only be used to define a surface using lines for its boundaries or by filling it in one uniform colour. To do realistic surface modelling we would need to have say 8 bits/pixel. In MODE 1 four colours are available and this is still insufficient for surface modelling. Colours can be mixed using a technique called 'dithering' or 'super-pixeling' but this results in a further decrease in resolution. However, we can still explore a variety of fascinating techniques used in three-dimensional graphics with wire frame models.

3.1 Two-dimensional transfomations and matrix notation

The easiest two-dimensional transformation to implement and one that you have probably used already is translation or movement in one direction. Translation is one of a set of linear transformations that we can apply to a set of coordinates that specify the vertices or corners of a piecewise linear figure.

In the more general case a set of points S is operated on by a transformation T to produce the set S'. This means that

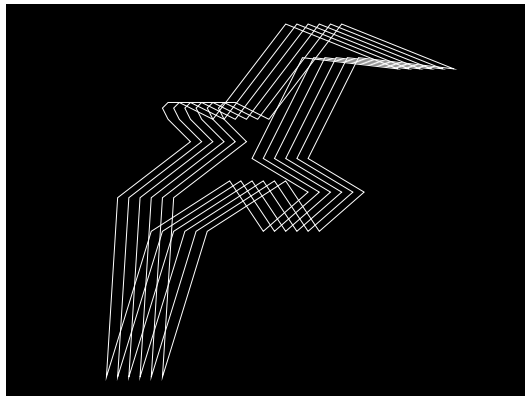
a mathematical operation is carried out on all the points in S, changing their coordinate values to produce points in S'.



S, a set of 4 points

S', the points after transformation *T*

Depending on the operation used this may change the shape of a figure that is defined by drawing straight lines between the points. Consider a set of coordinate points representing a piecewise linear bird. To draw the six seagulls shown below from a single set of (x,y) coordinates in DATA statements we could use the next program.



```
 10  MODE 0
 20  VDU 29, 640; 512;
 30  FOR i = 0 TO 100 STEP 20
 40    PROCdrawseagull(i)
 50  NEXT i
 60  k=GET : MODE 7 : END

 70  DEF PROCdrawseagull(i)
 80    RESTORE
 90    READ noofpoints
100    READ x, y : MOVE x+i, y
110    FOR j = 2 TO noofpoints
120      READ x, y : DRAW x+i, y
130    NEXT j
140  ENDPROC
```

```
150   DATA 17,-110,70
160   DATA -120 ,90, -110,100, -90,100, -30,70
170   DATA 100,240, 300,160, 130,180, 40,0
180   DATA 140,-60, 60,-130, 0,-40, -140,-130
190   DATA -220,-390, -200,-70, -70,30, -110,70
```

The program effects five simple transformations of the coordinate set in the DATA statements. To change the translation such that, for example, a diagonally displaced gull was displaced, we could change lines 100 and 120:

```
100   READ x, y : MOVE x+i, y+i

120     READ x, y : DRAW x+i, y+i
```

This is, however, an approach that would lead us into a different programming notation for each transformation that is used - a rather unsatisfactory state of affairs.

What we will now proceed to develop is a mathematical technique that allows us to specify any transformation as a set of four parameters. We can then have a single procedure that operates on the data set, using these parameters and producing the desired transformation. (We will then find that this system has certain deficiencies that can be overcome by using six parameters.)

To start with we will ignore translation and consider the other two common transformations, rotation and scaling. To rotate a point (x,y) through a clockwise angle, theta, about the origin, the transformation is:

$$xt = x\cos + y\sin$$
$$yt = -x\sin + y\cos$$

and scaing is given by:

$$x1 = xS1$$
$$y1 = yS2$$

The xt and yt are the transformed values of x and y. S1 is the scaling factor in the x direction and S2 the scaling factor in the y direction. For example, to magnify a figure uniformly we would use S1 = S2 = 3, say. Now we can make the two sets of equations look like each other by re-expressing the scaling equations as:

$$xt = xS1 + y \times 0$$
$$yt = x \times 0 + yS2$$

This enables us to write the operations using matrix notation. (Details on matrix notation and matrix

manipulation are to be found in Appendix 4.) Firstly rotation clockwise:

$$(xt, yt) = (x, y)\begin{bmatrix} \cos & -\sin \\ \sin & \cos \end{bmatrix}$$

and scaling:

$$(xt, yt) = (x, y)\begin{bmatrix} S1 & 0 \\ 0 & S2 \end{bmatrix}$$

and this is just a different way of writing the operations expressed as equations above. The notation used for expressing such transformations in matrix form is not standard but is, for better or worse, the de-facto standard in computer graphics. We can now represent various linear transformations as 2x2 matrices:

1) Identity (no effect) $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

2) Rotation (clockwise) $\begin{bmatrix} \cos & -\sin \\ \sin & \cos \end{bmatrix}$

3) Rotation (counter-clockwise) $\begin{bmatrix} \cos & \sin \\ -\sin & \cos \end{bmatrix}$

4) Scaling $\begin{bmatrix} S1 & 0 \\ 0 & S2 \end{bmatrix}$

5) Reflection (about the x-axis) $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

6) Reflection (about the y-axis) $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$

7) Y Shear $\begin{bmatrix} 1 & S \\ 0 & 1 \end{bmatrix}$

8) X Shear $\begin{bmatrix} 1 & 0 \\ S & 1 \end{bmatrix}$

Note that we cannot define translation using this system, a point we shall return to in a moment.

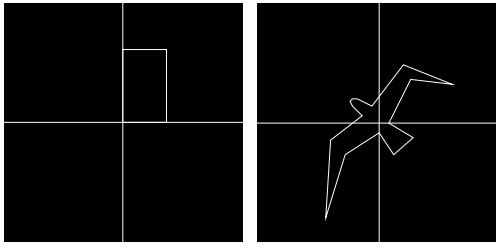
Here is a program that will accept the four parameters defining a transformation, using the terminology:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

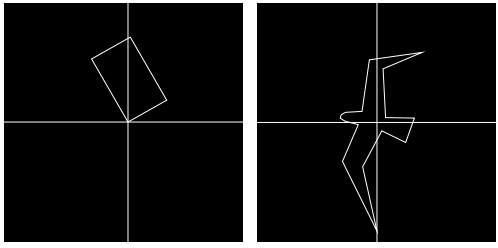
for the transformation matrix.

```
 10   MODE 0
 20   VDU 29, 640; 512;
 30   PROCdrawaxes
 40   PRINT '"a  c"''"b  d"; TAB(6,3); "?"
 50   INPUT TAB(9,2)a, TAB(14,2)c, TAB(9,4)b, TAB(14,4)d
 60   READ noofpoints
 70   READ x, y : PROCtransform(x,y)
 80   MOVE xt, yt
 90   FOR i = 2 TO noofpoints
100     READ x , y
110     PROCtransform(x, y)
120     DRAW xt , yt
130   NEXT i
140   k=GET : MODE 7 : END
150   DATA 5, 0,0, 180,0, 180,300, 0,300, 0,0

200   DEF PROCdrawaxes
210     MOVE -640,0 : DRAW 640,0
220     MOVE 0,-512 : DRAW 0,512
230   ENDPROC

300   DEF PROCtransform(x, y)
310     xt = a*x + b*y
320     yt = c*x + d*y
330   ENDPROC
```

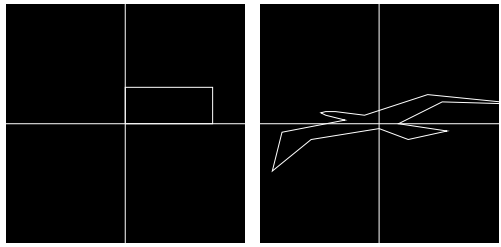
Note that we have just one procedure and two lines of calculation to perform any of the above transformations. The illustrations show, for both a rectangle and the more complex seagull, the effect of the transformations.



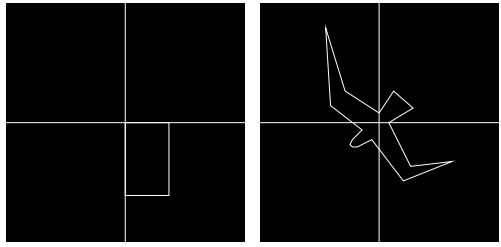
Identity $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$



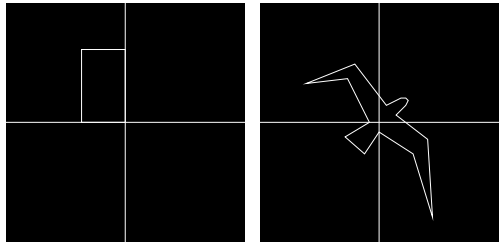
Rotation 30 degrees anticlockwise $\begin{bmatrix} 0.866 & 0.500 \\ -0.500 & 0.866 \end{bmatrix}$



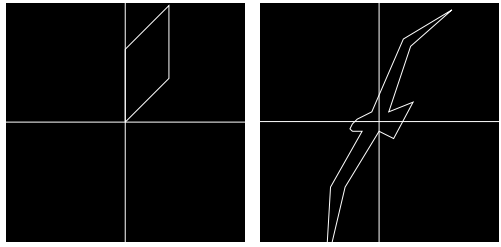
Scaling $\begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix}$



Reflection $\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$



Reflection $\begin{bmatrix} -0 & 0 \\ 0 & 1 \end{bmatrix}$

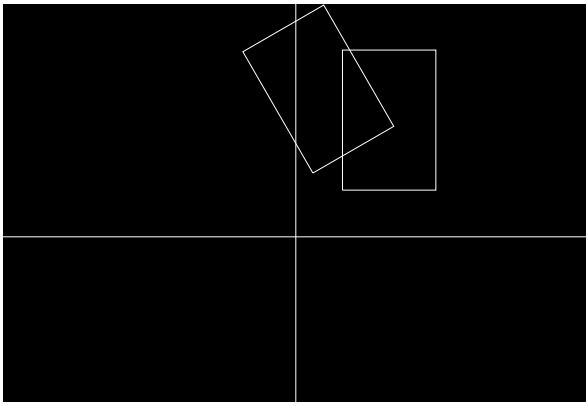


Y shear $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

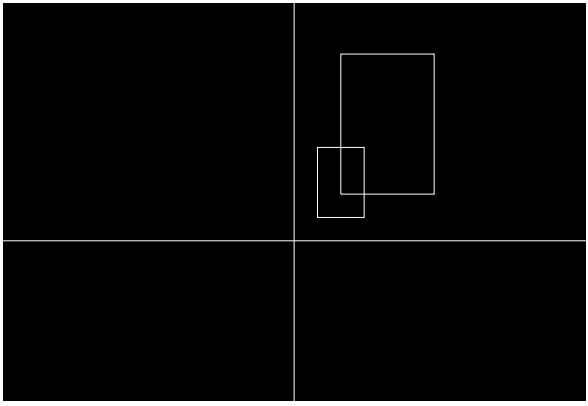
Now bear in mind that this is only a demonstration program and that normally we would not calculate cosines and sines before supplying these as program input. Rather cos(theta) and sin(theta) would be calculated in the course of execution of a program.

Homogeneous coordinates

The above system for effecting two-dimensional transformations has a number of drawbacks. Firstly it excludes important transformation (translation). Secondly all the transformations are centred at the origin, (0,0). This is fine in the above examples where the seagull was centred over the origin and one vertex of the rectangle was centred at the origin. Consider a rectangle not centred at the origin and subject to a 30 degree rotation.



or to a scaling:



Because the rotation is centred on the origin the rectangle

both rotates and translates. When we are rotating a geometrical figure it is far mere likely that we require rotation about an arbitrary point, for example, any one of the vertices of the rectangle, or the intersection of it diagonals. Such a transformation is not available in the above system. Similarly reflections are through the x-axis or the y-axis. Reflections through other lines are not. available.

Perhaps most absurd of all is scaling. This also involves a translation. The second illustration shows the rectangle offset from the origin and subject to a shrinking. Again it, is highly unlikely that we should ever require scaling complicated by a translation proportional to the magnitude of the scaling. Rather we may require 'pure' scaling about a centre point or vertex.

A homogenous coordinate systean is a notation that overcomes these difficulties. In a homogeneous coordinate system a point (x,y) becomes (x/r, y/r, r). It is convenient to make r = 1, avoiding division, giving the representation of a point as (x, y, 1). Because a point is now a three element row matrix, transformation matrices are now 3x3. This system has the immediate advantage that we can now represent the translation transformation with a 3x3 matrix, We can now write down seven common transformation matrices:

1) Translation $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$

where Tx is the translation in the x direction and Ty is the translation in the y direction. To check that this works let us translate the point (1, 1, 1) through a distance of 2 in the x direction:

$$(1, 1, 1)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} = ((1 \times 1 + 1 \times 0 + 1 \times 2), (1 \times 0 \times 1 \times 1 + 1 \times 0), (1 \times 0 + 1 \times 0 + 1 \times 1))$$

In other words

$$(x, y, 1)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix} = (x + Tx, y + Ty, 1)$$

Some of the other transformations in our new system are:

2) Rotation (clockwise) $\begin{bmatrix} \cos & -\sin & 0 \\ \sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix}$

3) Rotation (anti-clockwise)

$$\begin{bmatrix} \cos & \sin & 0 \\ \sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These rotations are still centred on the origin.

4) Scaling

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5) Reflection (x-axis)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

6) X Shear

$$\begin{bmatrix} 1 & 0 & 0 \\ S & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To test that these are exactly the same as we had before we can alter the demonstration program to input 6 parameters and perform the matrix multiplication:

```
 40  PRINT ''"a d"''"b e"''"c f";TAB(6,4); "?"
 50  INPUT TAB(9,2)a, TAB(14,2)d,
       TAB(9,4)b, TAB(14,4)e,
       TAB(9,6)c, TAB(14,6)f

300  DEF PROCtrans£orm(x, y)
310    xt = a*x + b*y + c
320    yt = d*x + e*y + f
330  ENDPROC
```

where the input parameters represent six of the coefficients of a 3x3 transformation matrix:

$$\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

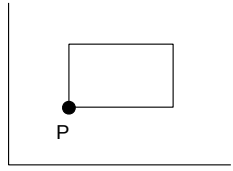
So what have we achieved? Very little as yet! Certainly we have represented translation in a matrix system, but we now need 9 matrix coefficients instead of 4 to achieve the same result. (Actually we have only used 6 of the 9 above, but you will see shortly that nine are convenient when combinations of transformations are considered.) However, the fact that we have included translation in our system now gives us a major advantage - we need no longer restrict ourselves to transformations at the origin. Thus homogeneous coordinates provide a rather roundabout way of adding constants to the transformed x and y values. The advantage of this notation is that transformations can still be

represented as matrices which will be useful when considering combinations of transformations.

Generalized two-dimensional transformations

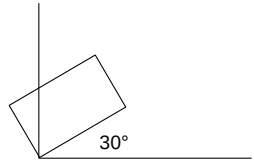
Consider the problem of rotation about any point. Say, for example, we wish to rotate a rectangle, in any position, 30 degrees counter-clockwise about its bottom LH vertex. We can easily do this now:

1) Original rectangle at P(Tx,Ty)

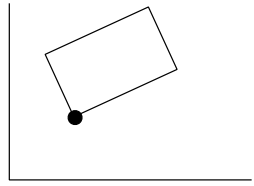


2) Translate to the origin

3) Rotate about the origin



4) Translate to P(Tx,Ty)



The three operations would be:

$$T1 = \text{translate} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Tx & Ty & 1 \end{bmatrix}$$

$$R = \text{rotate} = \begin{bmatrix} \cos 30 & \sin 30 & 0 \\ -\sin 30 & \cos 30 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T2 = \text{translate} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

Now instead of multiplying each point (x,y,1) by three matrices we can multiply the matrices together to obtain a 'net transformation matrix' (Details on matrix multiplication plus a procedure to perform the multiplication are contained in Appendix 4).

net transformation matrix = T1*R*T2

and then multiply each point (x,y,1) by this matrix. A net transformation matrix is always of the form:

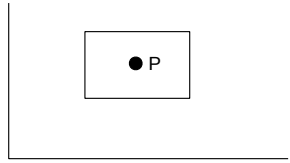
$$\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

and in this case

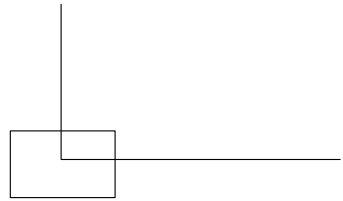
$$T1*R*T2 = \begin{bmatrix} \cos 30 & \sin 30 & 0 \\ -\sin 30 & \cos 30 & 0 \\ Tx(1-\cos 30) + Ty \sin 30 & Ty(1-\cos 30) - Tx \sin 30 & 1 \end{bmatrix}$$

The same approach can be used for scaling. Say we wanted to shrink the rectangle about its centre point:

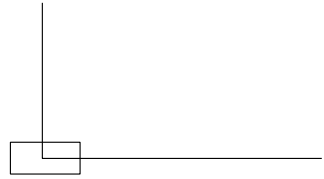
1) Original rectangle at P(Tx,Ty)



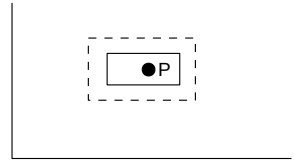
2) Translate to origin



3) Shrink



4) Translate to P(Tx,Ty)

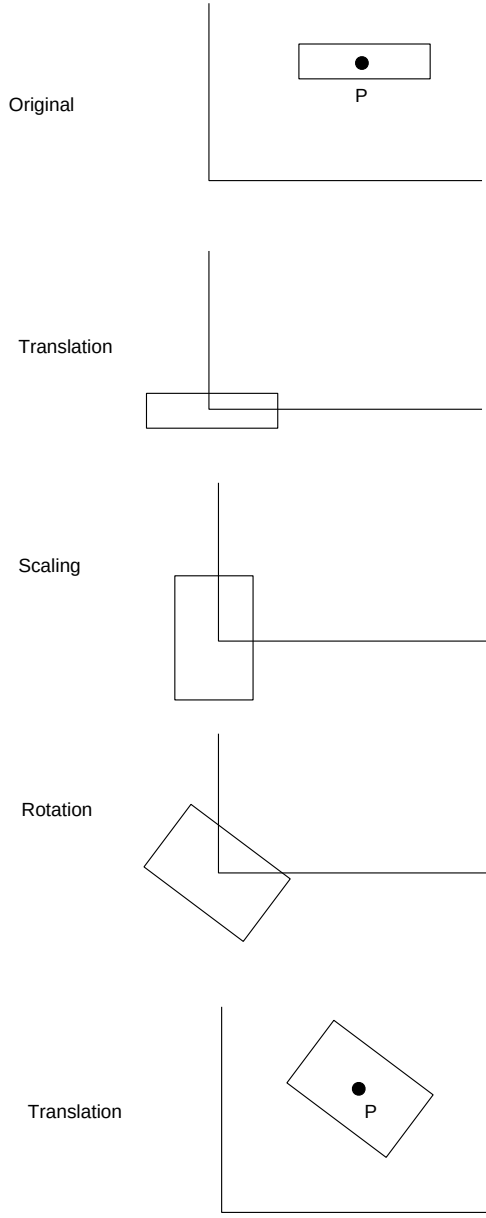


The rectangle has shrunk within itself - the desired transformation. This should be compared with the 2x2 system shrinking illustration (earlier) where the effect of. applying a shrink transformation was also to translate the rectangle 'outside itself'.

The net transform matrix for scaling is:

$$\begin{bmatrix} S1 & 0 & 0 \\ 0 & S2 & 0 \\ Tx(1-S1) & Ty(1-S2) & 1 \end{bmatrix}$$

A similar approach can be adopted for the other transformations. Any arbitrary combination of translation scaling and rotation will result in a net transformation matrix. For example we could combine the following transformations into a net transformation matrix:


Original
P
Translation
Scaling
Rotation
Translation
P

Now in general transformations are not commutative - the order in which the individual matrices are multplied to form a net transformation matrix is important. T1*R*T2 is not the same as T1*T2*R.

Because net transformation matrices are always of the form:

$$\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

The multiplication

$$(xt, yt, 1) = (x, y, 1) \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

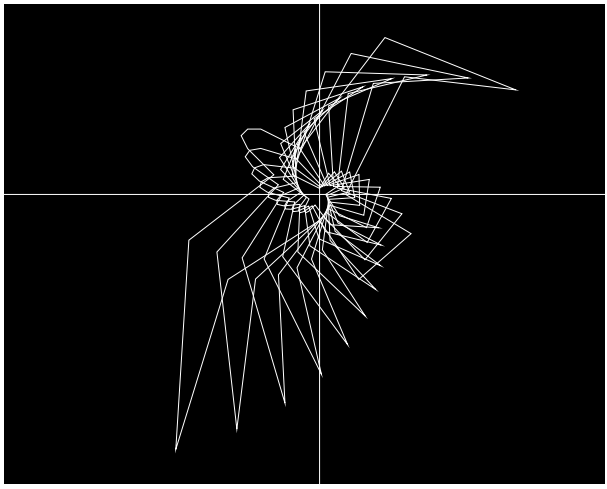
and this is implemented as:

$$(xt, yt) = (x, y, 1) \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

```
300   DEF PROCtransform(x, y)
310     xt = a*x + b*y + c
320     yt = d*x + e*y + f
330   ENDPROC
```

which is identical to the procedure we used above to test the basic transformation matrices. Thus what we have achieved with our 3x3 system is a method for producing a net transformation matrix from a series of 3x3 transformation natrices. The process is sometimes called concatenating (joining together) transformations. We need the 3x3 system to perform the concatenation. Concatenation is clearly advantageous because we need only multiply the matrices together once to obtain the net transformation matrix and then multiply each point in the data set by this matrix. As we have seen this final multiplication reduces to just 4 products and 4 additions. Such efficiency considerations are critically important in real time animation computers such as flight simulators.

Finally as an example of the use of these transformations we return to our seagull. Say we have a seagull drawn at (0,0) and wish to rotate it in steps of 10 degrees, and at the same time shrink it into itself.



The net transformation matrix for a 10 degree rotation is given by:

$$S*R = \begin{bmatrix} S & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 10 & \sin 10 & 0 \\ -\sin 10 & \cos 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} S\cos 10 & S\sin 10 & 0 \\ -S\sin 10 & S\cos 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

giving the implementation:

```
a = s*COS(RAD(10)) : d = s*SIN(RAD(10))
b=-d : e=a : c=0 : f=0
```

Remember that our seagull data set is already centred on the origin. If it were not centred at the origin we would apply the net transformation matrix T1*S*R*T2, to scale and rotate the figure.

The program makes repeated application of the rotation and scaling transformation, for different angles and scaling factors. Note that within the main program loop we reduce the scaling and increment 'theta'. The reduction in scale gives the spiral effect and and makes the image easier to interpret.

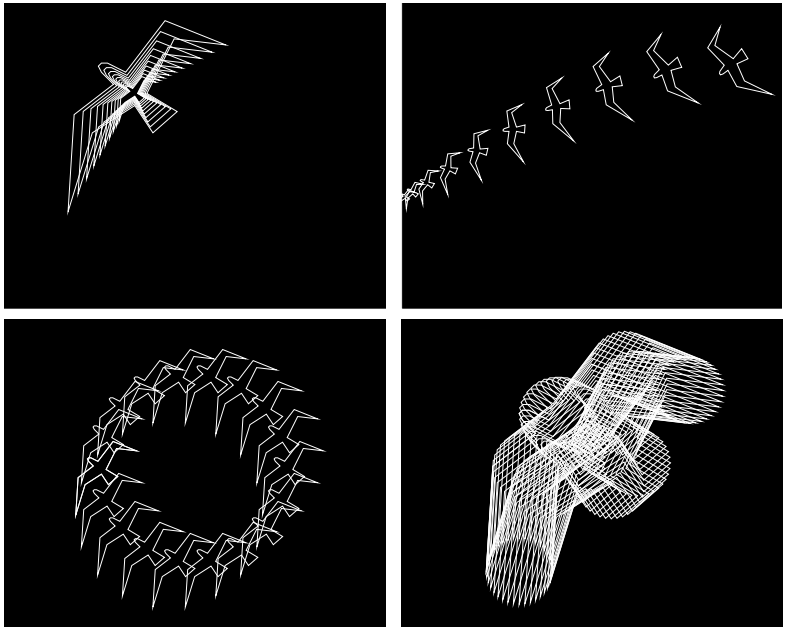
Now this program contains the three main elements of mathematically generated pictures: a process (FOR loop) to control the repetition of a drawing process; a drawing procedure that can either generate a data set mathematically, or operate with a supplied data set; and a net transformation matrix operating on the data set.

```
 10  MODE 0
 20  VDU 29, 640; 512;
 30  PROCdrawaxes
 40  s = 1 : theta = 0
 50  FOR bird = 1 TO 10
 60    RESTORE 200
 70    a = s*COS(RAD(theta)) : d = s*SIN(RAD(theta))
 80    b=-d : e=a : c=0 : f=0
 90    READ noofpoints
100    READ x, y : PROCtransform(x, y)
110    MOVE xt, yt
120    FOR p= 2 TO noofpoints
130      READ x, y
140      PROCtransform(x, y)
150      DRAW xt, yt
160    NEXT p
170    s = s*0.85: theta = theta + 10
180  NEXT bird
190  k=GET : MODE 7 : END

200  DEF PROCdrawaxes
210    MOVE -640,0 : DRAW 640,0
220    MOVE 0,-512 : DRAW 0,512
230  ENDPROC

300  DEF PROCtransform(x, y)
310    xt = a*x + b*y + c
320    yt = d*x + e*y + f
330  ENDPROC

340  DATA ...
```

Exercises

1 Write a utility program for constructing net transformation matrices. The individual transformations to be concatenated could be specified as matrices, or by a sequence of commands specifying translation, scaling, rotation and so on.

2 Write a series of programs that generate designs involving the seagull. The illustrations show some suggestions. The first illustration involves scaling only. The second uses scaling, rotation and (non-linear) translation. The third and fourth illustrations just involve translation and a single scale adjustment. They were produced by drawing instances of the seagull at equal angular increments around the circumference of a circle. Changing the radius of the circle, the scale of the seagull and the size of the angular increment will produce literally hundreds of different designs.



3 Repeat these programs but this time use a motif of your own design. This is most conveniently planned out using graph paper.

3.2. Three-dimensional graphics - general transformations

In this section we extend the techniques used in the two-dimensional transformations for translation, scaling, rotation etc. to deal with three-dimensional objects. Bear in mind that we are dealing with three-dimensional coordinates (x,y,z) that transform under scaling, rotation or whatever to other three-dimensional coordinates (xt,yt,zt) and that we are not yet ready to display a two-dimonsional image of a three-dimensional object on the screen. We will thus restrict this section to a short discussion on extending two-dimensional transformations to three-dimensional transformations without developing any programs. The three-dimensional transformation techniques are required before we can deal with the transformation into two dimensions for viewing.

The three-dimensional transformations using homogeneous cordinates (to enable the calculation of net transformation

matrices) are all of the form:

$$(xt, yt, zt, 1) = (x, y, z, 1) \begin{bmatrix} a & e & i & 0 \\ b & f & j & 0 \\ c & g & k & 0 \\ d & h & l & 1 \end{bmatrix}$$

where the basic transformation matrices are:

1) Scaling $\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Overall magnification or shrinking is achieved by:

Magnification $\begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & S & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotation is easily arrived at by again extending the two-dimensional rotation matrix. To rotate counter-clockwise in two-dimensions about the origin, we used:

$$\begin{bmatrix} \cos & \sin & 0 \\ -\sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and in the three-dimensional case if we consider the z-axis coming out of the paper, we have:

2) Rotation about th z-axis $\begin{bmatrix} \cos & \sin & 0 & 0 \\ -\sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

which produces rotation counter-clockwise about the z-axis. That this is a rotation about the z-axis can be seen, informally, from the fact that the matrix multiplication only affects the x and y coefficients. This observation can then be used to write down the other 2 rotation transformations:

3) Rotation about the x-axis $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos & \sin & 0 \\ 0 & -\sin & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

4) Rotation about the y-axis $\begin{bmatrix} \cos & 0 & -\sin & 0 \\ 0 & 1 & 0 & 0 \\ \sin & 0 & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

5) Translation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

Again we can derive net transformation matrices and, for example, if we wanted to rotate a body about a line parallel to the z-axis which passes through point (Tx, Ty, 0) we would use:

$$T1*R*T2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Tx & -Ty & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos & \sin & 0 & 0 \\ -\sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos & \sin & 0 & 0 \\ -\sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Tx\cos + Ty\sin & -Tx\sin - Ty\cos & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos & \sin & 0 & 0 \\ -\sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Tx\cos + Ty\sin + Tx & -Tx\sin - Ty\cos + Ty & 0 & 1 \end{bmatrix}$$

This would be implemented as:

```
xt = x*costheta-y*sintheta-Tx* costheta+Ty*sintheta+Tx
yt = x*sintheta+y*costheta-Tx *sintheta-Ty*costheta+Ty
zt = z
```

where of course the third equation is redundant because the rotation is about the z-axis and the z coordinate is unchanged for all points.

This transformation is effected for the case shown in the following illustration. Since we are not yet ready to actually plot these objects on a display screen the output from the program is shown as a coordinate list.

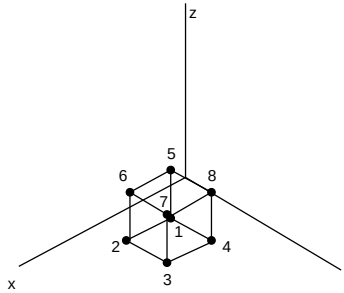
The transformation is:

```
theta = 45, Tx = Ty = 3
```

i.e. rotate 45 degrees counter-clockwise about the vertical line (3,3)

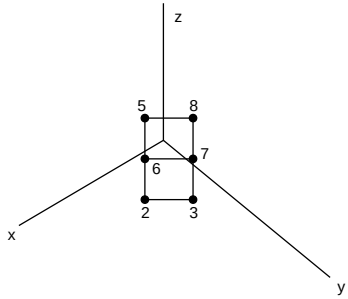
Input coordinates

vertex	x	y	z
1	3	3	3
2	4	3	3
3	4	4	3
4	3	4	3
5	3	3	4
6	4	3	4
7	4	4	4
8	3	4	4



Output coordinates

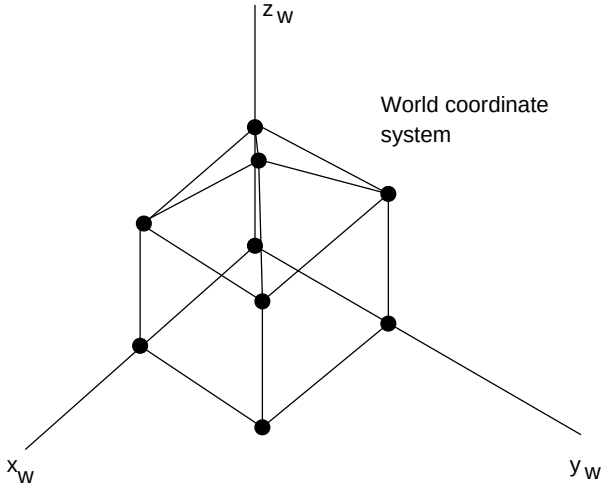
vertex	x	y	z
1	3.000	3.000	3.000
2	3.707	3.707	3.000
3	3.000	4.414	3.000
4	2.293	3.707	3.000
5	3.000	3.000	4.000
6	3.707	3.707	4.000
7	3.000	4.414	4.000
8	2.293	3.707	4.000



3.3 Three-dimesional graphics - viewing and perspective transformations

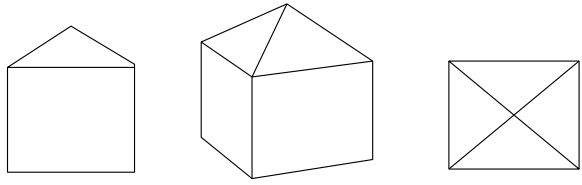
Three-dimensional graphics would be a simple extension of two-dimensional graphics - all our 3x3 transformations would become 4x4 transformations and that would be the end of the matter - if we had access to a three-dimensional display device. Of course the display device is two-dimensional and the external complication in three-dimensional graphics comes frcun the need to map the three-dimensional coordinates of the object to be displayed into the two-dimensional coordinates of the display system. It is convenient to express this as a combination of two transformations - the viewing transformation and the perspective transformation. This is needed in addition to any other of the transformations described above such as scaling, rotation and translation of the three-dimensional object.

We shall specify the coordinates of a three-dimensional object in a so called world coordinate system:



A house with a pyramidal roof could be specified in this system as a list of 9 (x,y,z) points in the world coordinate system.

Our intuition and visual experience with such abstractions as the wire frame model shown above enables us to realise the shape of the solid body that the model represents. Just as we would see different views of the real object as we moved our viewpoint around the sides and over the top, so we can construct two-dimensional representations of these different views from the original wire frame model.



The views can be constructed from the original object by specifying a viewpoint in relation to the object. The so-called 'viewpoint transformation' converts the coordinates expressed in the world coordinate system into 'eye' coordinates expressed in a coordinate system centred at the viewpoint.

Having applied the viewpoint transformation we are stillleft with a list of three-dimensional coordinates. The transformation that produces a list of two-dimensional or screen coordinates from the viewpoint list is called the perspective transformation.

The process can be illustrated:

World coordinates

vertex	xw	yw	zw
1	100	0	0
2	100	100	0
3	0	100	0
4	0	0	0
5	100	0	100
6	100	100	100
7	0	100	100
8	0	0	100
9	50	50	150

This is a coordinate list for the wire frame house shown above. After application of a particular viewpoint transformation:

Eye coordinates

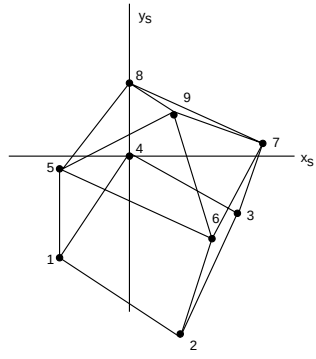
vertex	xe	ye	ze
1	-42	-69	1442
2	48	-102	1415
3	91	-32	1473
4	0	0	1500
5	-42	-5	1365
6	48	-38	1338
7	91	32	1396
8	0	64	1423
9	24	46	1342

This transformation takes us from world coordinates to eye coordinates and specifies the object in three dimensions as it would be seen from the specified viewpoint.

After application of a perspective transformation we have:

Screen coordinates

vertex	xs	ys
1	-88	-144
2	103	-216
3	185	-66
4	0	0
5	-93	-11
6	108	-84
7	195	69
8	0	135
9	54	102



Now it is important to bear in mind that both the the viewpoint transformation and the perspective transformation are particular transformations, arbitrarily chosen for this example. There is an infinity of viewpoint transformations because there is an infinity of viewpoints. There is a large set of perspective transformations depending on the particular criteria that we wish to adopt to transform or map a three-dimensional point into a two-dimensional point.

The viewing transformation

The viewing transformation, V, transforms points in the world coordinate system into the eye coordinate system. The required operation is:

$$(xe, ye, ze, 1) = (xw, yw, zw, 1)V$$

where

$$V = \begin{bmatrix} -\sin & -\cos\cos & -\cos\sin & 0 \\ \cos & -\sin\cos & -\sin\sin & 0 \\ 0 & \sin & -\cos & 0 \\ 0 & 0 & & 1 \end{bmatrix}$$

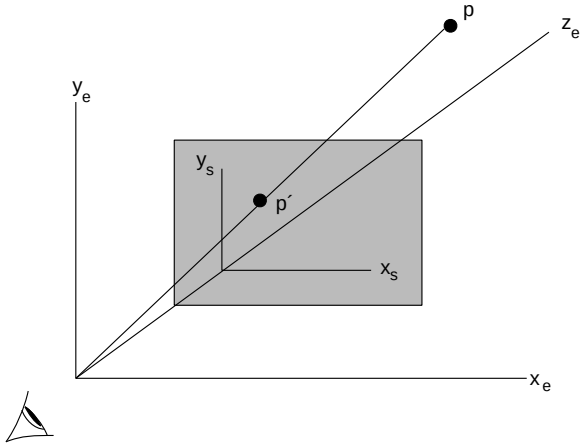
where rho, theta and phi together specify the viewpoint. The mathematics although not difficult requires a firm appreciation of movements (rotations etc.) in three-dimensional space. The mathematics that lead to this result, together with diagrams illustrating the significance of rho, theta and phi, is given in Appendix 5. If you wish, you can simply accept this result and use the following BASIC implementation:

```
1000  DEF PROCinitviewtransform(rho, theta, phi)
1010    LOCAL sintheta,costheta,sinphi,cosphi
1020    sintheta=SIN(RAD(theta)):costheta=COS(RAD(theta))
1030    sinphi =SIN(RAD(phi))   : cosphi =COS(RAD(phi))
1040    va = -sintheta : vb= costheta
1050    ve = -costheta*cosphi : vf = -sintheta*cosphi
1060    vg = sinphi
1070    vi = -costheta*sinphi : vj = -sintheta*sinphi
1080    vk = -cosphi : vl = rho
1090  ENDPROC

1100  DEF PROCviewtransform(x, y, z)
1110    xe = va*x + vb*y
1120    ye = ve*x + vf*y + vg*z
1130    ze = vi*x + vj*y + vk*z +vl
1140  ENDPROC
```

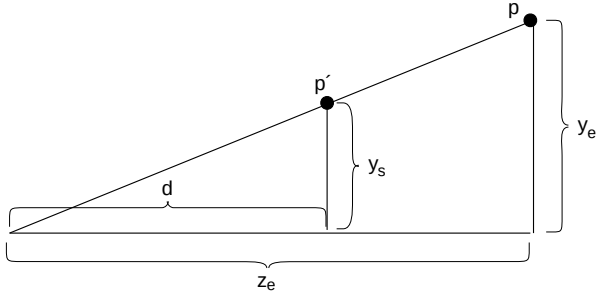
PROCinitviewtransform would be called once for a given viewpoint and PROCviewtransform would then be called once for each vertex in the object being viewed.

Now as we have already mentioned this transformation will give us a list of three-dimensional coordinates in the eye coordinate system. What we now need is a perspective: transformation that will produce screen coordinates. This considerably easier to derive than the viewing transformation.

Perspective transformation
The perspective transformation from the eye coordinate system to the screen coordinate system can be illustrated:



Point p is a point in the eye system, p' is its mapping into the screen coordinate system and d is the distance of the eye from screen. If we look at the illustration normal to the ye,ze plane we have:



and it is easily seen that:

ys = d*ye/ze

Similarly

xs = d*xe/ze

We can imagine the process an follows The screen is a plane parallel to the (xe,ye) plane. For every vertex in the object - a collection of points in the eye coordinate system - we can use a line to join the vertex to the origin (the eye). Where each line intersects the screen plane gives us the mapping from the vertex into the screen plane.

There is a family of perspective transformations available but this particular one is the easiest to compute and use, so we will restrict ourselves to it. It is categorised by having a single vanishing point and the xs and ys axes are parallel to the xe and ye axes. The perspective transformation is implemented by:

```
1150  DEF PROCperspecttransform(xe, ye, ze, d)
1160    xs = d*xe/ze
1170    ys = d*ye/ze
1180  ENDPROC
```

The procedures for carrying out the viewing and perspective transformations will be needed by all the remaining programs in this chapter.

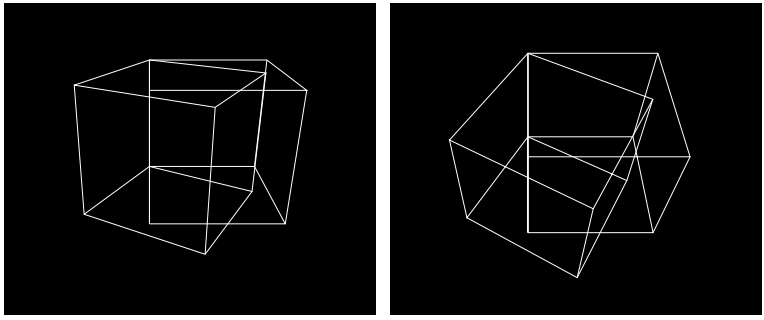
An example using using and perspective transformations

Here we turn to that hoary old chestnut - the wire frame cube. This is not so much lack of imagination, as the fact that a wire frame cube is such an intuitively obvious shape. The way in which the parameters input to the program affect the outcome of the program is then easier to understand.

The variables controlling the view are 'd' - the viewing distance, 'rho', 'theta' and 'phi' the spherical coordinates specifying a viewpoint (see Appendix 5). 'theta' is controlled by a FOR loop and the other three parameters are typed in. This means that we 'fly round' the cube at a constant elevation.

```
  10  DIM sx(8), sy(8)
  20  INPUT "rho",rho, "phi",phi, "screen dist, d",d
  30  MODE 0
  40  VDU 29, 640; 512;
  50  FOR theta = 0 TO 90 STEP 10
  60    PROCinitviewtransform(rho, theta, phi)
  70    RESTORE
  80    FOR vertex = 1 TO 8
  90      READ xw, yw, zw
 100      PROCviewtransform( xw, yw, zw)
 110      PROCperspecttransform(xe, ye, ze, d)
 120      sx(vertex) = xs : sy(vertex) = ys
 130    NEXT vertex
 140    CLG
 150    PROCdrawcube
 160  NEXT theta
 170  k=GET : MODE 7 : END
```

```
180  DATA 100,0,0, 100,100,0, 0,100,0, 0,0,0
190  DATA 100,0,100, 100,100,100, 0,100,100, 0,0,100
200  DEF PROCdrawcube
210    MOVE sx(1), sy(1)
220    FOR vertex = 2 TO 4
230      DRAW sx(vertex), sy(vertex)
240    NEXT vertex
250    DRAW sx(1), sy(1)
260    DRAW sx(5), sy(5)
270    FOR vertex = 6 TO 8
280      DRAW sx(vertex), sy(vertex)
290    NEXT vertex
300    DRAW sx(5), sy(5)
310    FOR vertex = 2 TO 4
320      MOVE sx(vertex+4), sy(vertex+4)
330      DRAW sx(vertex), sy(vertex)
340    NEXT vertex
350  ENDPROC
```



The illustration shows the cube from two different viewpoints. For each, the cube is displayed for two consecutive values of 'theta'. Now admittedly it's not a flight through a wire frame model of Chicago (a recent unbitious example of computer graphics), but you can begin to see the principles involved. The extra complexity in a flight through Chicago program would reside in the techniques necessary to cope with the vast number of vertices necessary to specify the model.

Now there are no checks in this program and certain input values will produce nonsense. Suggested values to start with are:

```
phi = 50 degrees
rho = 400 units
d = 900 units
```

Now make rho = 300 and see how the views increase in size because the viewpoint has moved closer to the world coordinnate system origin. (Incidentally note what happens if you move the viewpoint inside figure.) Changing the value of 'phi' will alter the elevation of the view. For example try 'phi = 0'. You should be able to understand why we do not get a rotating square from this viewpoint. Now, keeping 'rho' constant, decrease the value of 'd'. This not only makes the object smaller but increases the perspective.

Exercises

1 Using the two-dimensional seagull of Section 3.1, assume that each vertex of the bird has a zero z-coordinate. We can then treat it as if it were a cardboard cut-out hanging in three-dimensional space. Write a program that will generate displays of the bird as seen from different viewing positions.

2 Organise a 'fly round' a house shape. Include a few windows and doors in the three-dimensional specification of the house.

3 Organise a 'fly round' any other three-dimensional shape with which you are familiar.

4 Select a particular viewpoint and modify the wire frame cube program so that it draws only three 'visible' faces. Ornament each face of this display with an appropriately scaled two-dimensional seagull. The seagull must of course be in the same plane as the face it is ornamenting.

3.4 Constructional techniques

Now that we have developed a viewing and perspective transform that allows us to display, on a two-dimensional screen, a mapping of a three-dimensional object, we can turn our attention to simple techniques for constructing wire frame and other models. Of course not all models that we may wish to construct can be defined mathematically, or at least it may be exceedingly difficult to so specify them. Such models, car body shapes for example, may be built up as data files from perhaps diverse sources that may include some mathematical modelling together with interaction from a device such as a light pen or a graphics tablet.

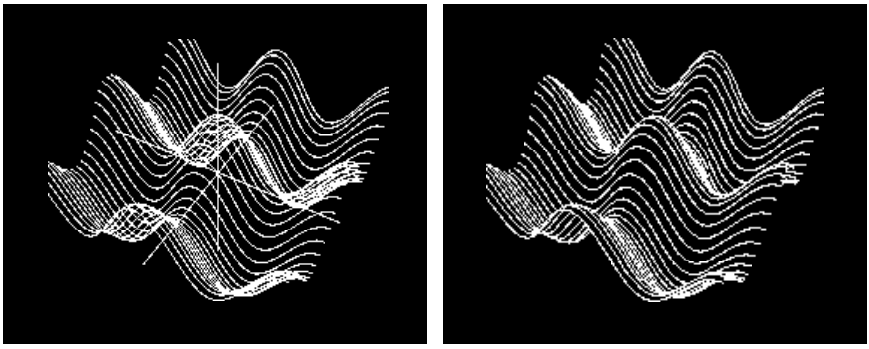
In this section we will consider generating surfaces specified mathematically as functions of two variables (f(x,y)). We will also look at generating simple convex bodies made up of plane faces.

Display of f(x,y)

Functions of two variables can be generated and displayed plotting variations along lines of constant x, constant y, or both. The illustration shows the function

```
f(x,y) = cos(x) + cos(y)
```

plotted along lines of constant x:



The program is:

```
 10  INPUT rho, theta, phi, d
 20  PROCinitviewtransform(rho, theta, phi)
 30  MODE0 : VDU 29, 640; 512;
 40  FOR xw = 360 TO -360 STEP -20
 50    PROCscreen(xw, -360, FNf(xw,-360)) : MOVE xs,ys
 60    FOR yw= -340 TO 360 STEP 20
 70      PROCscreen(xw, yw, FNf(xw,yw)) : DRAW xs,ys
 80    NEXT yw
 90  NEXT xw
100  k=GET : MODE 7 : END

110  DEF FNf(x,y) = 100*(COS(RAD(x)) + COS(RAD(y)))

120  DEF PROCscreen(x,y,z)
130  LOCAL xe ,ye ,ze
140    PROCviewtransform(x, y, z)
150    PROCperspecttransform(xe, ye, ze, d)
160  ENDPROC
```

To plot in the other direction the control variables in the FOR loops can be reversed. Note that when displaying functions of two variables the viewpoint position can be critical if the features of the function are to be visually

recognisable. The illustrations ahown were produced with

```
rho   = 1500
theta = 30
phi   = 45
    d = 2000
```

The high values of 'rho' and 'd' are necessary because we have arbitrarily scaled the function by 100 (otherwise a value in the range -2 to 2 would have been produced). The function itself is composed of humps and valleys symmetrically disposed about the origin. You can see that the tips of the peaks and valleys are confused by 'crossovers'. These can be removed by a fairly easy hidden line removal algorithm for plots of f(x,y) (see the last section in this chapter). The result of applying a hidden line algorithm is shown in the second photograph. If such a hidden line removal algorithm is to be applied then the function lines must be plotted in the order of decreasing x. (i.e. the one nearest to the viewpoint is plotted first.)

Finally the interpretation of f(x,y) may be assisted by drawing the world coordinate axes on the plot. This is easily accomplished by:

```
      35 PROCdrawaxes

  170  DEF PROCdrawaxes
  180    PROCscreen(-360,0,0) : MOVE xs,ys
  190    PROCscreen( 360,0,0) : DRAW xs,ys
  200    PROCscreen(0,-360,0) : MOVE xs,ys
  210    PROCscreen(0, 360,0) : DRAW xs,ys
  220    PROCscreen(0,0,-360) : MOVE xs,ys
  230    PROCscreen(0,0, 360) : DRAW xs,ys
  240  ENDPROC
```

Generating wire frame models

A number of commonly used convex bodies can be generated by sweeping a plane or a line through 360 degrees. The simplest figure in this class is a cylinder. In the next program 74 vertices on the top and bottom circumferential edges of a cylinder are generated by sweeping the line (i.e the line end-points)

```
-100, 0, 100
.-100, 0, 0
```

through 360 degrees at 10 degree increments.

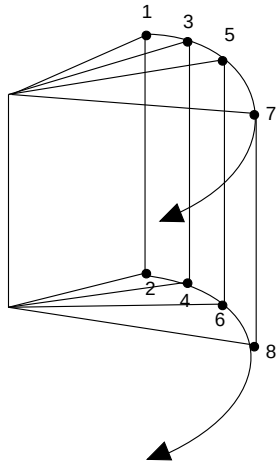
```
10  noofpoints=2 : nootvertices*37*noofpoints
20  DIM object(3,noofvertices),cylinder(2,noofvertices)
30  MODE 0 : VDU 29, 640; 512;
```

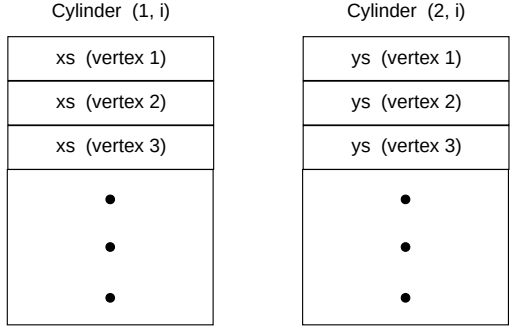
```
 40  INPUT "rho" ,rho, "theta" ,viewtheta,
       "phi" ,phi, "screen dist ,d" ,d
 50  PROCinitialise
 60  PROCinitviewtransform(rho, viewtheta, phi)
 70  PROCworldtoscreen : PROCplotcylinder
 80  k=GET : MODE 7 : END

200  DEF PROCinitialise
210  LOCAL sintheta,costheta, theta, p,v
220    v = 0
230    FOR theta = 0 TO 360 STEP 10
240      RESTORE
250      sintheta = SIN(RAD(theta))
260      costheta =COS(RAD(theta))
270      FOR p = 1 TO noofpoints
280        v = v + 1
290        READ x, y, z
300        object(1, v) = x*costheta + y*sintheta
310        object(2, v) = -x*sintheta + y*costheta
320        object(3, v) = z
330      NEXT p
340    NEXT theta
350  ENDPROC
360  DATA -100,0,0, -100,0,100

400  DEF PROCworldtoscreen
410  LOCAL v
420    FOR v = 1 TO noofvertices
430      PROCviewtransform(object(1,v),
           object(2,v),object(3,v))
440      PROCperspecttransform(xe,ye,ze,d)
450      cylinder(1,v) = xs
460      cylinder(2,v) = ys
470    NEXT v
480  ENDPROC

500  DEFPROCplotcylinder
510  LOCAL v
520    MOVE cylinder(1, 1), cylinder(2, 1)
530    FOR v = 3 TO noofvertices STEP 2
540      DRAW cylinder(1,v), cylinder(2,v)
550    NEXT v
560    MOVE cylinder(1,2), cylinder(2,2)
570    FOR v = 4 TO noofvertices STEP 2
580      DRAW cylinder(1,v), cylinder(2,v)
590    NEXT v
500    FOR v = 1 TO noofvertices STEP 2
610    MOVE cylinder(1,v), cylinder(2,v)
620      DRAW cy1inder(1,v+1), cylinder(2,v+1)
630    NEXT v
640  ENDPROC
```

The transformation required to rotate the line about the zw axis is

$$\begin{bmatrix} \cos & -\sin & 0 & 0 \\ \sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

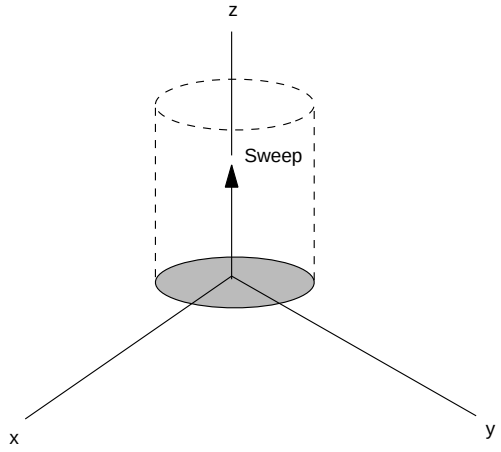
and this is applied 37 times in the program for values of 'theta' of 0, 10, 20,...,360. Thus 74 vertices are generated and stored in the array 'object'. This array is then processed by PROCworldtoscreen that produces the screen coordinates. PROCworldtoscreen repeatedly uses the two procedures PROCviewtransform and PROCperspecttransform. The screen coordinates are stored in array 'cylinder' and this array is plotted by PROCplotcylinder. The cylinder vertices we loaded into array 'cylinder' as follows:



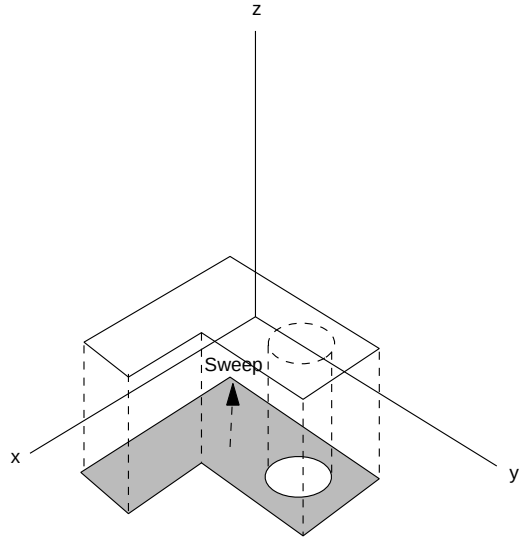


This structure is reflected in the structure of the procedure that plots the cylinder.

It is easy to see that the same effect could be achieved by sweeping a circle (a regular polygon) along the long axle of the cylinder.



This method is called translational sweeping to distinguish from the previous method - rotational sweeping. Just as rotational sweeping can be used to generate any solid with rotational symmetry (a cooling tower for example) translational sweeping can be used to generate any solid with translational symmetry.



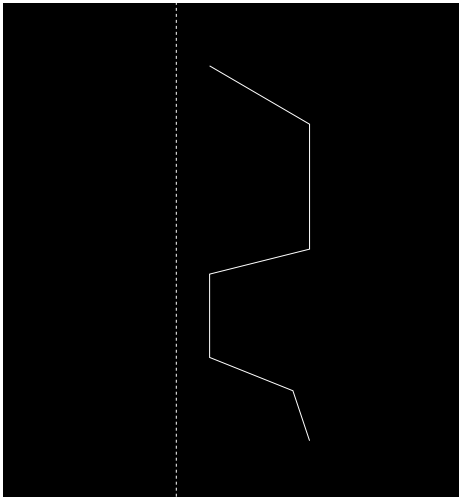
When you execute the last program note the speed of plotting from the calculated screen coordinate array, compared with the speed of calculation of these coordinates. You can genera models and store the vortex arrays in files of DATA statements, but this will, of course, give an instance of the model from one viewpoint only.

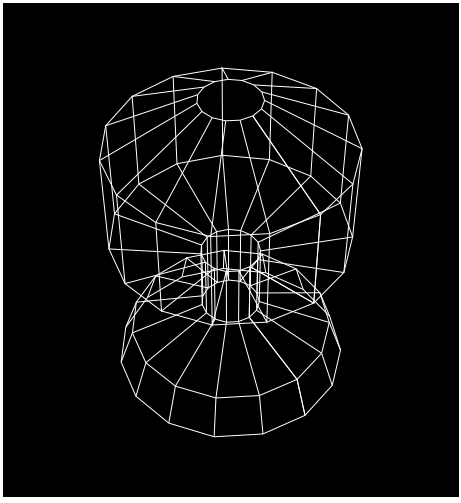
This structure can be used to generate related figures. Changing one vertex in the DATA statement will cause the program to generate cones instead of cylinders. Using a function definition rather than DATA statements can enable spheres (rotating a semicircle) and cooling towers (rotating segment of a hyperbola) for example, to be generated.

Exercises

1 Edit the above program so that it generates the other common 'mathematical solids' such as a cone, a truncated cone and a sphere.

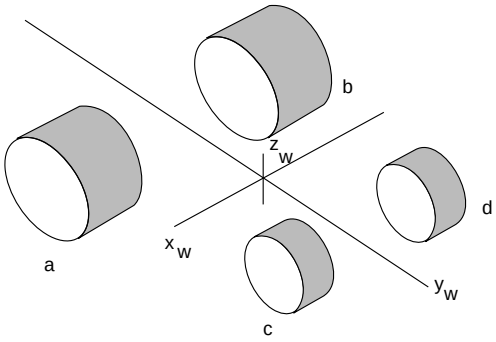
2 Write a program to generate a wire frame model of a body with rotational symmetry. The program should allow the user to generate a profile using the rubberband techniques described in Chapter 2. (An example of this process is shown in the first illustration.) By recording each vertex in the rubber band profile the program should then generate a rotationally symmetric wire framemcxiel. This is simply a matter of generating circles of appropriate radius at each vertex. The 'vertical' wire frame lines are constructed by joining points on the circumferences of these circles at equal angular increments.



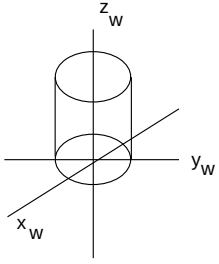


Three dimensional transformations - composite bodies

Composite models can be built up using instances of already generated structures. Say, for example, that we wanted to generate a model consisting of four cylinders as shown, perhaps to represent the four wheels of a motor car.

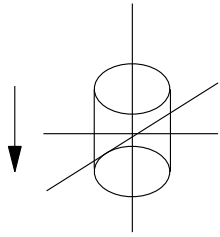


We could start by generating the cylinder in the correct orientation (long axis parallel to the (xw, yw) plane, but let us consider instancing the cylinders in their correct orientation and position from data generated by the previous program - an upright cylinder.

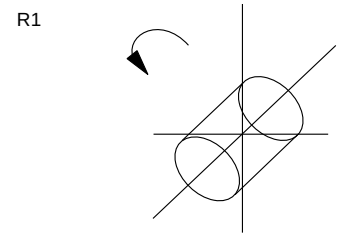


To instance the four cylinders in their required positions and orientations we could use various combinations of the following transformations:

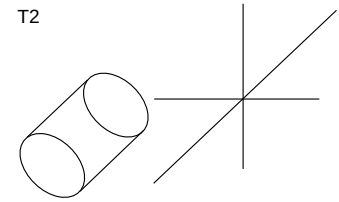
T1 move the cylinder down so that its long axis is centred at the origin



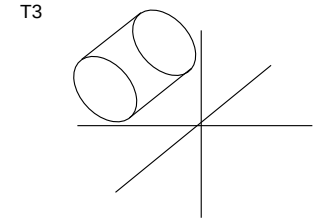
R1 rotate 90 degress about the yw-axis - the cylinder in this position can now be used in each of the subsequent transformations



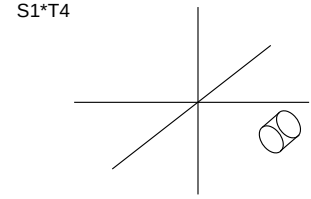
T2 translate 100, -100, 0 and DRAW an instance of the cylinder



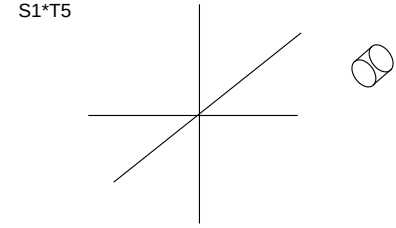
T3 translate (-100, -100, 0) and DRAW an instance of the cylinder



S1*T4 shrink by half and translate (75, 150, 0); DRAW an instance of the cylinder



S1*T5 shrink by half and translate (-75, 150, 0); DRAW an instance of the cylinder

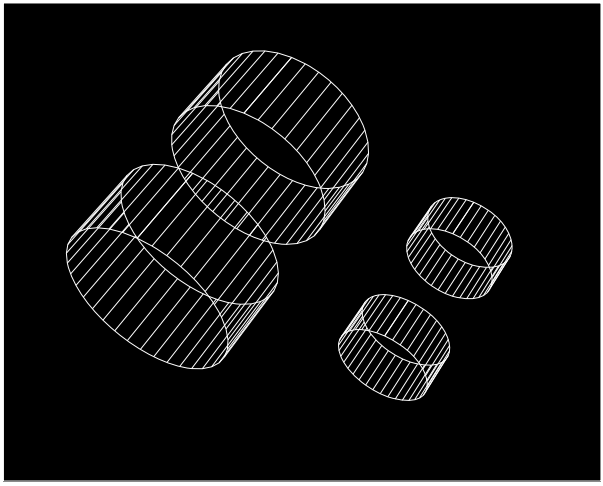


Thus the following transformations are required for each instance:

```
instance A T1*R1*T2
instance B T1*R1*T3
instance C T1*R1*S1*T4
instance D T1*R1*S1*T5
```

T1*R1 is common to all transformations. We can organise the program by having two arrays, one in which to store tho original cylinder in world coordinates, after the common transformation T1*R1. The other array would store the result of the transformations T2, T3, S1*T4 and S1*T5 prior to plotting. The program structure would be:

```
read original vertex file into the vertex array 'object'

PROCtransformT1R1

PROCtransformT2 : PROCworldtoscreen : PROCplotcylinder

PROCtransformT3 : PROCworldtoscreen : PROCplotcylinder

PROCtransformS1T4 : PROCworldtoscreen : PROCplotcylinder

PROCtransformS1T5 : PROCworldtoscreen : PROCplotcylinder
```



PROCtransformT1R1 operates on and outputs to the array 'object' originally initialised with the vertex data produced by the previous program. All the other procedures output to the array 'target' and can be the same procedure supplied with different parameters according to the

transformation being carried out. The array 'target' is the input to PROCworldtoscreen and this produces the array 'cylinder' containing the screen coordinates for PROCplotcylinder. Thus we are using three arrays - 'object', which stores the original cylinder and the cylinder after the cmmon transform T1*R1, 'target', which stores each cylinder after it is subject to the remaining transforms and 'cylinder', which stores the final screen coordinates for each generated cylinder. The transformations are:

$$T1*R1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -50 & 1 \end{bmatrix} \begin{bmatrix} \cos 90 & 0 & -\sin 90 & 0 \\ 0 & 1 & 0 & 0 \\ \sin 90 & 0 & \cos 90 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -50 & 0 & 0 & 1 \end{bmatrix}$$

$$T2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 100 & -100 & 0 & 1 \end{bmatrix}$$

$$T3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -100 & -100 & 0 & 1 \end{bmatrix}$$

$$S1*T4 = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 75 & 150 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 75 & 150 & 0 & 1 \end{bmatrix}$$

$$S1*T5 = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -75 & 150 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ -75 & 150 & 0 & 1 \end{bmatrix}$$

Examination of the transformation shows that only 12 parameters need be specified:

$$\begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix}$$

and this is implemented as the procedure PROCtransform in the program below. PROCinitialise can initialise array 'object' either by generating the cylinder as described previously, or by reading previously generated vertices.

```
 10  noofpoints=2 : noofvertices=noofpoints*37
 20  DIM object(3,noofvertices),
         target(3,noofvertices),
         cylinder(2,noofvertices)
 30  MODE0 : VDU 29, 640; 512;
 40  INPUT "rho" ,rho, "theta" ,viewtheta,
           "phi " ,phi, "screen dist ,d" ,d
 50  PROCinitviewtransform(rho,viewtheta,phi)
 60  PROCinitialise : PROCtransformT1R1
 80  PROCtransform(1,0,0, 100,0,1,0,-100,0,0,1,0)
 90  PROCworldtoscreen : PROCplotcylinder
100  PROCtransform(1,0,0, -100,0,1,0,-100,0,0,1,0)
110  PROCworldtoscreen : PROCplotcylinder
120  PROCtransform(0.5,0,0,75,0,0.5,0,150,0,0,0.5,0)
130  PROCworldtoscreen : PROCplotcylinder
140  PROCtransform(0.5,0,0,-75,0,0.5,0,150,0,0,0.5,0)
150  PROCworldtoscreen : PROCplotcylinder
160  k=GET : MODE 7 : END

400  DEF PROCworldtoscreen
410    LOCAL v
420    FOR v = 1 TO noofvertices
425  REM ***** note change in next line *****
430      PROCviewtransform(target(1,v),
           target(2,v),target(3,v))
440      PROCperspecttransform(xe,ye,ze,d)
450      cylinder(1,v) = xs
460      cylinder(2,v) =ys
470    NEXT v
480  ENDPROC

700  DEF PROCtransformT1R1
710  LOCAL v, x,z
720    FOR v = 1 TO noofvertices
730      x= object(1,v) : z = object(3,v)
740      object(1 ,v) = z - 50
750      object(3,v) = -x
760    NEXT v
770  ENDPROC
```

```
800  DEF PROCtransform(a,b,c,d,e,f,g,h,i,j,k,l)
810    LOCAL v, x,y,z
820    FOR v = 1 TO noofvertices
830      x = object(1,v) : y = object(2,v)
840      z = object(3,v)
850      target(1,v) = a*x + b*y + c*z + d
860      target(2,v) = e*x + f*y + g*z + h
870      target(3,v) = i*x + j*y + k*z + 1
880    NEXT v
890  ENDPROC
```

Off-line animation - moving modelled objects

You can see from the time taken to execute the above example limit real-time animation of such mathematically generated objects is impossible on the BBC micro. Some graphics processors are designed to perform real time animation of three-dimensional scenes - notably flight simulators - but most animation of this kind is generated off-line. A frame us built up and stored and quickly transformed into a screen Image. As far as the BBC micro is concerned even storing frames in the form of final position vertex arrays in disc files will still not facilitate animation in BASIC. The MOVE and DRAW utilities still take too long to construct a picture. Off-line animation could however be performed in BASIC by single shotting with a cine camera and the techniques used are worthy of study.

Let us return to the four wheel model. Say that we want to generate a sequence of 10 frames with the wheel set moving from left to right. Of course it is not just a matter of taking a two-dimensional image and moving it to the right (two-dimensional animation) because if we are animating with respect to a stationary observer, each frame will be different because of the changing position of the object with respect to the stationary viewpoint.

To make the wheels move forward in a straight line (along say the y-axis) we have to apply an increasing yw translation to each wheel object in turn, and keep the viewpoint stationary. This could be accomplished in the above program by inserting another common transformation prior to the individual wheel transformations and drawing sequences. The required transformation is:

$$Ta = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & d & 0 & 1 \end{bmatrix}$$

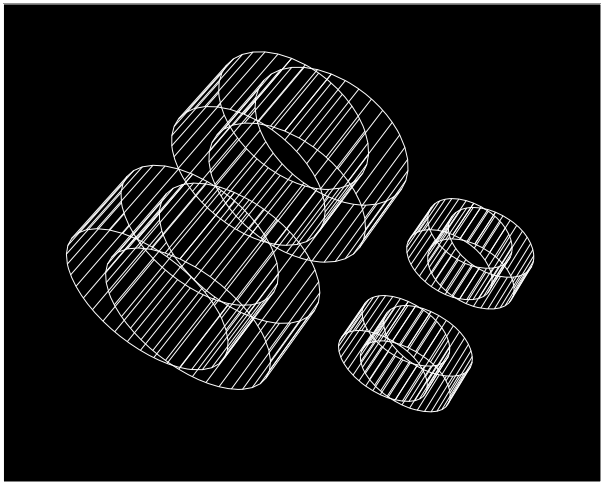
where d is the inter-frame displacement along the yw-axis. The procedure can output into array 'object' so that the displacements accumulate:

```
900 DEF PROCdisplace
910 LOCAL v
920 FOR v = 1 TO 74
930  object(2,v) = object(2,v) + displacement
940 NEXT v
950 ENDPROC
```

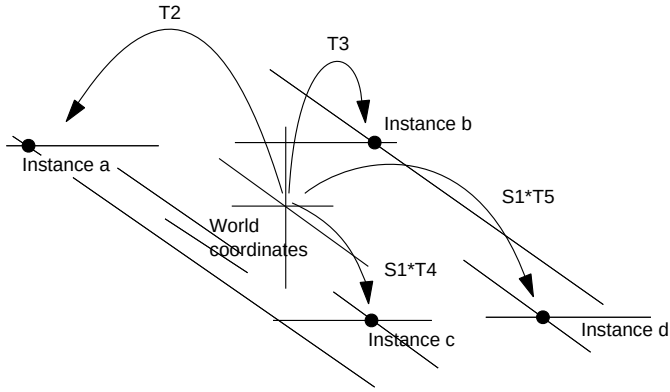
The whole program would then be looped, providing one frame for each execution of the loop.

```
        :
        :
 55   INPUT "displacement", displacement
        :
        :
 75   FOR frame=1 TO 10
 76     CLG
          :
          :
150     PROCworldtoscreen : PROCplotcylinder
155     PROCdisplace : k = GET
156   NEXT frame
```

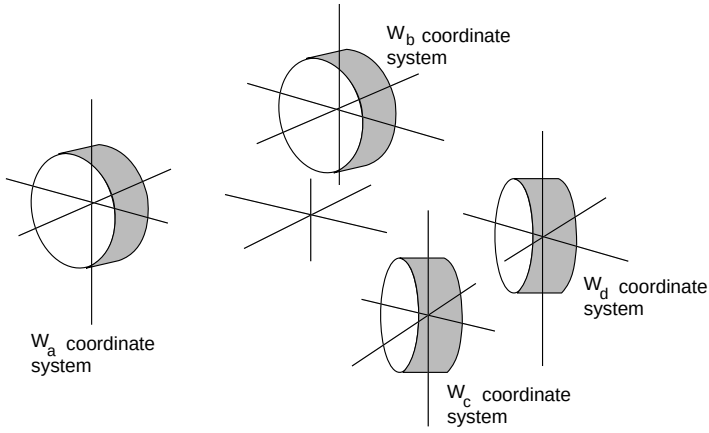
The illustration shows just two frames (superimposed) that are produced by the first two executions of the frame loop in the program.



Local coordinate systems
The method used above to instance the four wheels is somewhat less than satisfactory. What we are doing is taking one cylinder, subjecting it to increasing yw displacements and then translating it into four wheel positions.



What if the car was turning? Then not only would the front actuals have to be rotated about the zw-axis, but each wheel's (x,y) displacement would be different. In general it makes more sense to divide the problem into four sub-problems with a separate local coordinate system for each wheel.



We can then start by having an instance of a cylinder in each of the four coordinate systems. The set of points in Wa, representing a wheel, would be exactly the same as the set of points in Wb. Similarly the set in We would be identical to those in Wd. To start off with a frame in which all the wheels were pointing straight ahead, we would transform the wheel points into the world coordinate system (from their own local system) by transforming each sub-coordinate system into the world coordinate system. Remember that to transform a coordinate system we apply the inverse

translation transformation. Previously we had a cylinder at the centre of the world system and instanced it four times as shown.

Now we have four cylinders each with their own coordinate system and a transformation to give a set of points in theworld coordinate system.

You can see that the two approaches are exactly equivalent by working through a two-dimensional example. Now prior to taking each wheel set into the world system we can operate on each wheel in its own coordinate system. For example, we may want to rotate each front wheel about its own z-axis, by applying the same rotational transformation to each front wheel in turn (to simulate a car turning). After each object has been moved in its own coordinate system, we transform the points into the world coordinate system using:

$$(xwi, ywi, zwi, 1) \begin{bmatrix} -Tx & 0 & 0 & 0 \\ 0 & -Ty & 0 & 0 \\ 0 & 0 & -Tz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where i = a, b, c or d.

The inter-frame x and y increments for the front wheels are identical, the wheels following parallel circumferential tracks. The x and y increments for the rear wheels are different, the differential drive in the rear axle facilitating this discrepancy. In such a context the convenience of separate coordinate systems is apparent.

3.5 Hidden line removal

Now as mentioned in the introduction we are practically constrained on the BBC micro to wire frame models and this means that we are interested in removing hidden lines or edges rather than hidden surfaces. Such an operation tends to enhance the interpretation of the displayed model and diminish the effect of such ambiguities as the Necker cube illusion. There is a large variety of hidden line removal algorithms used in different contexts. There is no standard approach or algorithm, the algorithm used depends on the application or context. One thing is certain, hidden line removal will add an execution time penalty to your program.

Algorithms can either operate in object space where the calculations are carried out in the world coordinate system or in display space where the calculations are carried out ta the display coordinate system. Hybrid algorithms use information frem both the object and the image space domain. One of the most direct approaches to the problem of hidden surface removal is the depth buffer or z-buffer algorithm. In this algorithm we keep a record of the intensity of a

pixel together with its depth or value of its 'ze' coordinate (depth information comes from object space and the algorithm, although primarily a display space algorithm, needs some information from object domain. The need for a two-dimensional buffer to store the 'depth' of each pixel gives the algorithm its name and of course is the major disadvantage of the technique - it requires considerable extra storage. The algorithm operates on the polygons that delineate the boundary of a surface that has been mapped into the screen coordinate system. It can be stated as:

(1) Initially set depth(x,y) to a large value and intensity(x,y) to the background.

(2) For every polygon in the scene find all the pixels that lie within the polygon

    For each pixel:

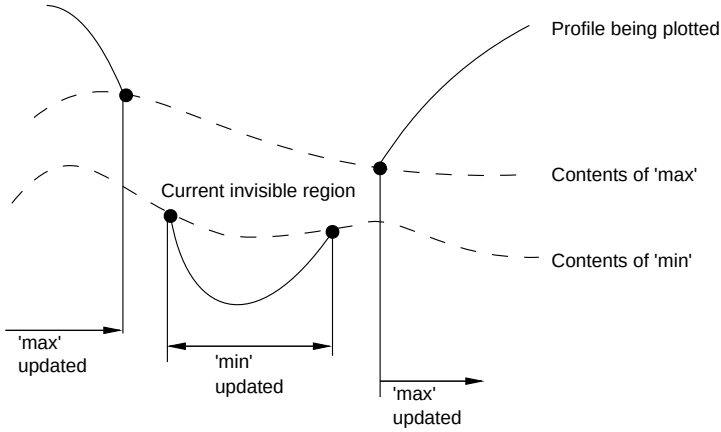
    (a) Calculate its z value

    (b) If the z value < depth(x,y) (eye coordinates) then depth(x,y) becomes the z value and intensity(x,y) becomes the intensity of the pixel, otherwise leave depth(x,y) and intensity(x,y) unaltered.

Now the point of this digression is firstly to point out that it is completely general and will work for any group of objects or scene. Secondly it operates on individual pixels. On the BBC micro if we are restricting ourselves to BASIC then we have no easy access in a wire frame model to the 'bright' pixels joining two vertices - we only have access to the end points. Efficient general purpose hidden line removal algorithms need to be integrated with the scan conversion process - the process that converts for example the statement 'DRAW x,y' into a line of the required bright pixels. We will now look at two special cases of hidden line removal. Firstly consider the plotting of a function of two variables.

Hidden line remval - f(x,y)

This is a display space or image space algorithm and depends on the fact that the function is plotted along lines of constant x starting with the profile nearest to the observer. The algorithm needs two arrays at the horizontal resolution of the display (640 in MODE 0). In these arrays, say 'max(ys)' and 'min(ys)', we keep the current highest and lowest values of ys - the screen y value. When plotting a new function or profile along a line of constant x it is deemed to be visible when above the profile stored in 'max' and below the profile stored in 'min'.



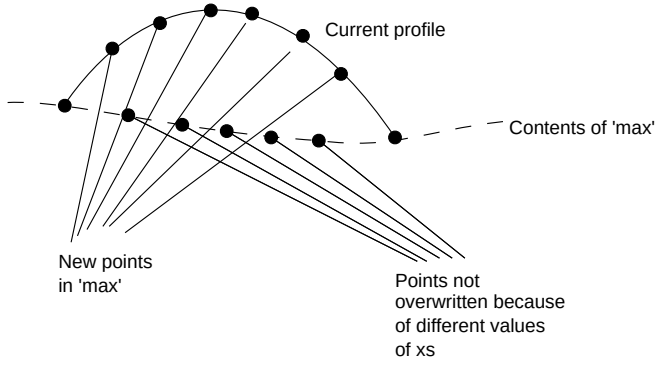
The algorithm is:

(1) Initialise 'max' to the minimum y screen coordinate and 'min' to the maximum y screen coordinate.

(2) For all x profiles or contours
For each y along the profile
Calculate xs and ys

```
IF ys>max(xs)       THEN DRAW xs,ys : max(xs)=ys
ELSE IF ys<min(xs)  THEN DRAW xs,ys : min(xs)=ys
ELSE MOVE xs,ys
```

This simple algorithm however has two drawbacks. The arrays 'max' and 'min' must each have 640 elements. However, if we increment y by 20 degrees in the object space (as we did in the original program) the algorithm will produce only 37 xs values for each profile where the size of the intervals between values depends on the viewpoint. When the arrays are updated with new information gaps will be left. Another effect of these gaps is that when plotting does start it may not necessarily coincide with the previous x profile.



There are two possible solutions to this prolem. We could reduce the increment in y to a point where all the elements in 'max' and 'min' arc guaranteed to be accessed. For the function and viewpoint used earlier this would have to be about 1/2 degree. The plotting time would then be inordinately long, because of excessive use of the trigonometric functions. Another approach is to 'interpolate' between successive screen points generated by the program, i.e. to examine each pixel on a line between two successive screen points. It is the latter method that is implemented in the next program. The output from the program was displayed earlier alongside the same function plotted without hidden line removal. Note that, with the long variable names and readable layout used here, there is no room to run the program in MODE 0. As presented, the program runs in MODE 4. The photograph was taken by running a compacted version of the program in MODE 0. Techniques for compacting programs are given at the end of Chapter 10.

```
 10  INPUT rho, theta, phi, d
 20  PROCinitviewtransform(rho, theta, phi)
 30  MODE 4 : VDU 29, 640; 512;
 32  PROCinithiddenlineremoval
 40  FOR xw= 360 TO -360 STEP -20
 50    PROCscreen(xw, -360, FNf(xw,-360)) : MOVE xs,ys
 55    prevxcell = (640+xs) DIV xstep : prevys =ys
 60    FOR yw = -340 TO 360 STEP 20
 70      PROCscreen(xw,yw,FNf(xw,yw)) : PROCcheckplot
 80    NEXT yw
 90  NEXT xw
100  k=GET : MODE 7 : END

110  DEF FNf(x,y) = 100*(COS(RAD(x)) + COS(RAD(y)))

120  DEF PROCscreen(x,y,z)
130  LOCAL xe, ye,ze
140    PROCviewtransform(x, y, z)
150    PROCperspecttransform(xe, ye, ze, d)
160  ENDPROC

300  DEF PROCinithiddenlineremoval
310  LOCAL c
320    xstep = 4 : cells = 1279 DIV xstep
330    DIM min(cells), max(cells)
340    FOR c = 0 TO cells
350      min(c) = 512 : max(c) = -512
360    NEXT c
370  ENDPROC
```

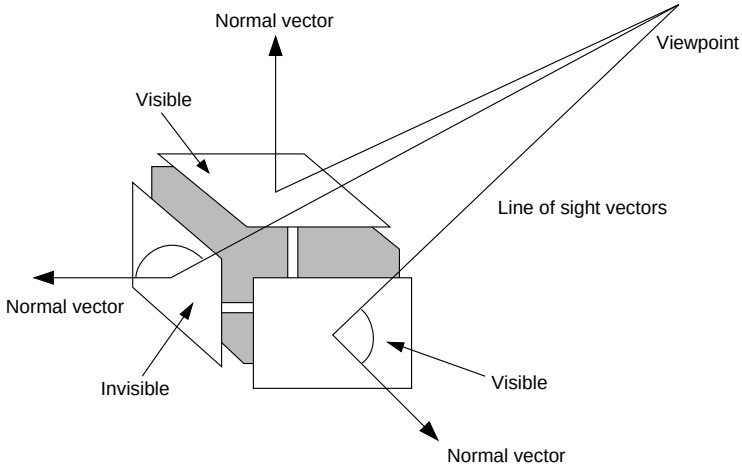
```
100
  400  DEF PROCcheckplot
  410  LOCAL xcell, yinc,nextys, c
  420    xcell = (xs+640) DIV xstep
  430    IF xcell=prevxcell THEN PROCpoint(xcell,ys)
  440    yine = (ys-prevys)/(xcell-prevxcell)
  450    nextys = prevys
  460    FOR c = prevxcell TO xcell
  470      nextys = nextys + yine
  480      PROCpoint(c, nextys)
  490    NEXT c
  500    prevxcell = xcell : prevys = ys
  510  ENDPROC

  520  DEF PROCpoint(c,y)
  530    LOCAL x : x= c*xstep- 640
  540    IF c<0 OR c>cells THEN MOVE x,y :ENDPROC
  550    IF y<max(c) AND y>min(c) THEN MOVE x,y : ENDPROC
  560    IF y<min(c) THEN min(c) = y
  570    IF y>max(c) THEN max(c) = y
  580    DRAW x ,y
  590  ENDPROC
```

All in all you can see that great care must be taken when setting up bidden line removal if inordinately long processing is to be avoided.

Hidden line removal - back surface elimination

This is another special case method that works for single convex polyhedrons. It is not really an algorithm but a straightforward application of vector mathematics. It can be used as the basis of a more general algorithm that will deal with scenes containing many convex polyhedral objects. It is an object space procedure and although we are removing edges, paradoxically it is the surfaces defined by the edges that we consider.

Given a viewpoint we determine whether or not a face is visible from that viewpoint. Consider the next illustration showing a cube decomposed into 6 surfaces. Pointing out of each surface we have a line that is normal or perpendicular to the surface. This line or vector determines the orientation of the plane. It is called a surface normal vector. From the viewpoint we can construct 'line of sight' lines or vectors to any point on each surface. If we construct a line of sight vector to meet the surface vector, then the angle between these two vectors gives us a visibility test. The surface is visible from the viewpoint if, and only if, the angle between these two vectors is less than 90 degrees.

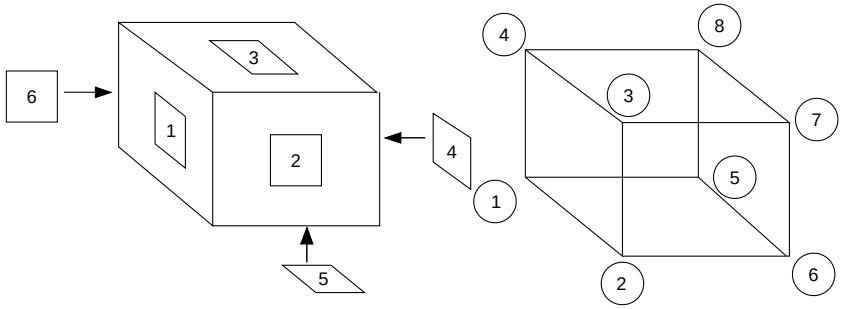


In the next program we have implemented back surface elimination for the wire frame cube. PROChidden line remove tests each of the 6 surfaces for visibility. If surface is visible its edges are plotted. The main program is:

```
 10  DIM vertex(3,8), surface(6,4)
 20  DIM vector1(3), vector2(3)
 30  DIM visible(6)
     35rho=1500:theta=50:phi=30:d=8000:GOTO50
 40  INPUT "rho" ,rho, "theta" ,theta, "phi " ,phi,
           "screen dist ,d" ,d
 50  costheta=COS(RAD(theta)) : sintheta=SIN(RAD(theta))
 60  cosphi=COS(RAD(phi)) : sinphi=SIN(RAD(phi))
 70  xview = rho*sinphi*costheta
 80  yview = rho*sinphi*sintheta
 90  zview = rho*cosphi
100  PROCinitviewtransform(rho,theta,phi)
110  MODE 0 : VDU 29, 640; 512;
120  PROCinitialise
130  PROChidden_line_remove
140  FOR surfaceno = 1 TO 6
150    IF visible(surfaceno) THEN
         PROCtransform_and_plot(surfaceno)
160  NEXT surfaceno
170  k=GET : MODE 7 : END
```

We need to set up a data structure to associate a surface with its edges and this is implemented by using two arrays 'surface' and 'vertex'. The array 'vertex' contains a list of the coordinates of the 8 vertices in the cube, and 'surface' contains for each surface a list of vertex numbers that define the surface. This data structure means that we can deal with surfaces as entities rather than vertices.

	vertex number					coordinates		
surface(1)	1	2	3	4	vertex(1)	100	0	0
surface(2)	2	6	7	3	vertex(2)	100	100	0
surface(3)	3	7	8	4	vertex(3)	100	100	100
surface(4)	6	5	8	7	vertex(4)	100	0	100
surface(5)	1	5	6	2	vertex(5)	0	0	0
surface(6)	1	4	8	5	vertex(6)	0	100	0
					vertex(7)	0	100	100
					vertex(8)	0	0	100



A surface must contain (for the vector mathematics) vertices listed in counter-clockwise order as seen frenn outside the object. Thus the surface number 2 is specified in the second row of array 'surface' which will contain the vertex numbers 2, 6, 7 and 3. Thus to access surface number 2 for calculation or plotting we would indirectly access the array 'vertex' as follows:

```
vertex(surface(2,1))
vertex(surface(2,2))
vertex(surface(2,3))
vertex(surface(2,4))
```

PROCinitialise sets up this DATA structure. Note that this is another disadvantage of the method - either the construction of surfaces in the object needs to be explicitly stated as here, or the method that constructs the solid must do so in such way that consecutive vertices

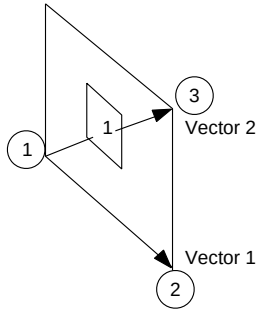
relating to one surface are listed in counterclockwise order (looking in from outside the object).

```
180  DEF PROCinitialise
190  LOCAL v, vertexno,surfaceno
200    FOR v = 1 TO 8
210      READ vertex(1,v), vertex(2,v), vertex(3,v)
220    NEXT v
230    FOR surfaceno= 1 TO 6
240      FOR vertexno = 1 TO 4
250        READ surface(surfaceno, vertexno)
260      NEXT vertexno
270    NEXT surfaceno
280  ENDPROC

290  DATA 100,0,0, 100,100,0, 100,100,100, 100,0,100
300  DATA 0,0,0, 0,100,0, 0,100,100, 0,0,100
310  DATA 1,2,3,4, 2,6,7,3, 3,7,8,4
320  DATA 6,5,8,7, 1,5,6,2, 1,4,8,5
```

PROChidden_line_remove lists the method broken down into further procedure calls.

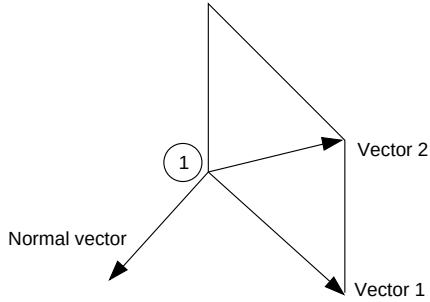
```
330  DEF PROChidden_line_remove
340    FOR surfaceno = 1 TO 6
350      PROCcalc_surface_vectors(surfaceno)
360      PROCcalc_normal_vector
370      PROCcalc_line_of_sight_vector(surfaceno)
380      PROCvisibility_test(surfaceno)
390    NEXT surfaceno
400  ENDPROC
```

The first thing that we do is to calculate the components of a pair of vectors lying in the surface. These are two vectors emanating from the first vertex. We do this for each surface and calculate the components of vector1 and vector2 storing them in arrays 'vector1' and 'vector2'.



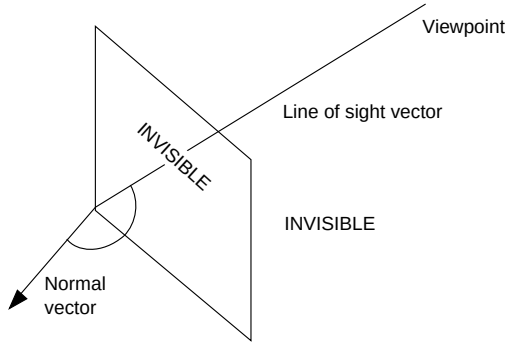
```
410  DEF PROCcalc_surface_vectors(surfaceno)
420    FOR i = 1 TO 3
430      vector1(i) = vertex(i, surface(surfaceno,2))
           - vertex(i,surface(surfaceno,1))
440      vector2(i) = vertex(i, surface(surfaceno,3))
           - vertex(i,surface(surfaceno,1))
450    NEXT i
460  ENDPROC
```

The 'cross product' of vector1 and vector2 gives a normal vector or a vector perpendicular to the surface and joining the surface at the first vertex. This has components 'normalx', 'normaly' and 'normalz'.




```
470  DEF PROCcalc_normal_vector
480  normalx = vector1(2)*vector2(3) -
       vector2(2)*vector1(3)
490  normaly = vector1(3)*vector2(1) -
       vector2(3)*vector1(1)
500  normalz = vector1(1)*vector2(2) -
       vector2(1)*vector1(2)
510  ENDPROC
```

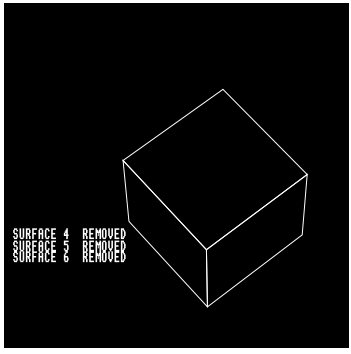
We can then calculate the components of the vector that joins the viewpoint to the vertex containing the normal vector and apply the visibility test.

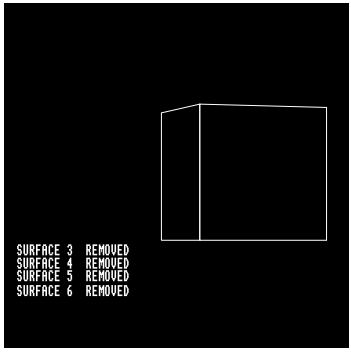


```
520   DEF PROCcalc_line_of_sight_vector(surfaceno)
530     lineofsightx = xview -
          vertex(1,surface(surfaceno,1))
540     lineofsighty = yview -
          vertex(2,surface(surfaceno,1))
550     lineofsightz = zview -
          vertex(3,surface(surfaceno,1))
560   ENDPROC

570   DEF PROCvisibility_test(surfaceno)
580     visible(surfaceno) = normalx*lineofsightx +
          normaly*lineofsighty +
          normalz*lineofsightz > 0
590   ENDPROC
```

The visibility test calculates the 'dot product' of the line of sight and normal vectors. If the magnitude of the dot product is less than zero the the angle between the two vectors is greater than 90 degrees. Finally we need a standard transformation and plotting procedure for a surface:

```
600   DEF PROCtransform_and_plot(surfaceno)
610   LOCAL vertexno, startx,starty
620     PROCscreenvertex(surfaceno,1)
630     MOVE xs,ys : startx=xs : starty=ys
640     FOR vertexno = 2 TO 4
650     PROCscreenvertex(surfaceno,vertexno)
660       DRAW xs ,ys
670     NEXT vertexno
680     DRAW startx,starty
690   ENDPROC

700   DEF PROCscreenvertex(s,v)
710     PROCviewtransform(vertex(1, surface(s,v)),
          vertex(2, surface(s,v)),
          vertex(3, surface(s,v)))
720     PROCperspecttransform(xe,ye,ze,d)
730   ENDPROC
```





The illustration shows views of the cube together with a list of the surfaces removed for two different viewpoints.

Exercises

1 Apply hidden line removal to plots of a variety of three-dimensional functions f(x,y).

2 Apply the hidden surface removal algorithm to a variety of three-dimensional shapes - a house, a tetrahedron, and so on. You will need to extend the dimensions of the arrays used if there are more than eight vertices or it there are surfaces with more than four vertices. If the number of vertices in a surface varies within the same object, a separate array will be needed to record the number of vertices in each surface.

3 Organise a 'fly round' each object used in the last exercise, with hidden line removal.