9 Orthographic Projections

We may now address the problem of drawing views of three-dimensional objects on our (necessarily) two-dimensional graphics screen. The simple method we describe here is a direct generalisation ofthe method introduced in chapter 4 for two-dimensional objects. Again it involves the use of (up to) three *positions*. To illustrate these ideas we first give a brief outline, and then expand on this by using pictorial and numerical examples. We start by defining an arbitrary but fixed triad of axes in space which we call the ABSOLUTE system. Then, as in the two-dimensional case, we consider the three positions: (1) the SETUP position, (2) the ACTUAL position and (3) the OBSERVED position.

(1) The SETUP Position

Most scenes will be composed of simple objects (such as cube(s) – see example 9.1) which are set at a particular position and orientation in space. It is very inefficient to calculate by hand the complicated coordinates ofevery vertex of these objects and input them into the program. Instead we look at each object in turn and initially define it in an elementary way relative to the ABSOLUTE triad, usually setting it about the origin. The information required will be that of vertices (*x*-coordinate, *y*-coordinate and *z*-coordinate), and perhaps lines (which join pairs ofvertices) or (later when we consider hidden surface algorithms) facets, which are polygonal planar areas bounded by the above-mentioned lines. This elementary definition of the object is called its SETUP position. We could also have other information such as the colour of the object.

(2) The ACTUAL Position

We may then use the matrix techniques of the last chapter to generate a matrix that will move the object from its SETUP position to its required ACTUAL position relative to the ABSOLUTE axes. We shall call this the SETUP to ACTUAL matrix P.

(3) The OBSERVED Position

Viewing an object in three-dimensional space naturally involves an observer (the eye – and note only one eye!) placed at a position (EX, EY, EZ) relative to the ABSOLUTE axes looking in a fixed direction: this direction of view can be uniquely determined by any other point on the line of sight (DX, DY, DZ), say. The head can also be tilted, but more of this later. What the eye sees when it looks at a three-dimensional object is a projcction of the vertices, lines and facets of the object on to a (two-dimensional) *view plane* which is normal to the line of sight. In order to calculate such projections we must standardise our approach. We use matrix methods to transform all the points in space so that the eye is placed at the origin, and the line of sight is along the positive *z*-axis. This is the OBSERVED position, and the matrix that transforms the ACTUAL to OBSERVED position is called Q throughout this book. The method for calculating Q will be dealt with in detail later, but for the time being we shall assume that the eye is already at the origin and is looking along the *z*-axis: so in this simple case Q is the identity matrix.

When all the points in space have been moved into this OBSERVED position we note that the view plane is now parallel to the *x*/*y* plane through the origin. Having moved the eye into the correct position, we are now ready to project the object on to the view plane. But note, as yet we have neither defined the position of the view plane (we have only its normal), nor have we described the type of projection of three-dimensional space on to the plane. These two requirements are closely related. In this book we shall consider three possible projections – in a later chapter we shall deal with the perspective and *stereoscopic projection*, but first we introduce the simplest projection – the *orthographic*.

The orthographic Projection

Nothing could be simpler. In the orthographic projection we can set the view plane to be *any* plane with normal vector along the line of sight. When transformed into the OBSERVED position, the view plane will be any plane that is parallel to the *x*/*y* plane given by the equation $z = 0$. For simplicity we take the *x*/*y* plane through the origin. The vertices of the object are projected on to the view plane by the simple expedient of seiting their *z*-coordinates to zero. Thus any two different points in the OBSERVED position, (x, y, z) and (x, y, z') say (where $z \neq z'$), are projected on to the same point $(x, y, 0)$ on the view plane. Then we identify the *x*/*y* values on the plane with points in the graphics screen coordinate system (usually centred on the screen) by using the methods of chapter 2. Once the vertices have been projected on to the view plane and then on to the screen, we can construct the projection of lines and facets. These are related to the projected vertices in exactly the same way as the original lines and facets are related to the original vertices.

Before considering in detail the general case where the eye and direction of view are arbitrarily positioned, we shall consider an elementary example to demonstrate the orthographic projection.

Example 9.1

Use the above ideas to draw an orthographic projection of a cube. Figures such as those in figure 9.1 are called *wire diagrams* or *skeletons* (for obvious reasons).

In the SETUP position the cube may be thought to consist of eight vertices (1, 1, 1), (1, 1, –1), (1, –1, –1), (1, –1, 1),(–1, 1, 1), (–1, 1, –1),(–1, –1, –1) and (–1, –1 , 1): vertices are labelled numerically 1 to 8. The twelve lines that form the wire cube join vertices 1 to 2, 2 to 3, 3 to 4, 4 to 1; 5 to 6, 6 to 7, 7 to 8, 8 to 1; 1 to 5, 2 to 6, 3 to 7 and 4 to 8.

Figure 9.1a shows the simplest possible example of an orthographic projection of the cube, where even the SETUP to ACTUAL matrix is the identity matrix, that is the cube stays in its SETUP position. We get a square: pairs of parallel lines from the front and back of the cube project into the same line on the screen. We put a '+' in these diagrams to show the position of the *z*-axis in the OBSERVED position (into the screen).

Figure 9.1b shows the same cube drawn after the following three transformations place it in its ACTUAL position:

(a) Rotate the cube by an angle $\alpha = -0.927295218$ radian about the *z*-axis – matrix A. This example is contrived so that $\cos\alpha = 3/5$ and $\sin\alpha = -4/5$, so ensuring that the rotation matrices consist of uncomplicated elements.
(b) Translate it by the vector (–1, 0, 0) – matrix B.
(c) Rotate it by an angle $-\alpha$ about the *y*-axis – matrix C.

The SETUP to ACTUAL matrix is thus P = C × B × A, where

$$A = \begin{pmatrix} 3/5 & 4/5 & 0 & 0 \\ -4/5 & 3/5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 3/5 & 0 & 4/5 & 0 \\ 0 & 1 & 0 & 0 \\ -4/5 & 0 & 3/5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and P is given by the matrix

$$P = \frac{1}{25} \begin{pmatrix} 9 & 12 & 20 & -15 \\ -20 & 15 & 0 & 0 \\ -12 & -16 & 15 & 20 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

So the above eight vertex coordinate triples in the SETUP position are transformed into the following eight ACTUAL coordinate triples: (26/25, –5/25,

7/25), (–14/25, –5/25, –23/25), (–38/25, –35/25, 9/25), (2/25, –35/25, 39/25), (8/25, 35/25, 31/25), (–32/25, 35/25, 1/25), (–56/25, 5/25, 33/25), (–16/25, 5/25, 63/25).

For example (1, 1, 1) is transfornned into (26/25, –5/25, 7/25) because

$$\frac{1}{25}\begin{pmatrix} 9 & 12 & 20 & 15 \\ -20 & 15 & 0 & 0 \\ -12 & -16 & 15 & 20 \\ 0 & 0 & 0 & 25 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{25}\begin{pmatrix} 26 \\ -5 \\ 7 \\ 25 \end{pmatrix}$$

Since the ACTUAL to OBSERVED matrix Q is the identity matrix, the projected coordinates on the view plane are thus (26/25, –5/25), (–14/25, –5/25), (–38/25, –35/25), (2/25, –35/25), (8/25, 35/25), (–32/25, 35/25), (–56/25, 5/25), (–16/25, 5/25). We can place these points on the screen and join them with lines in the same order as they were defined in the SETUP cube.

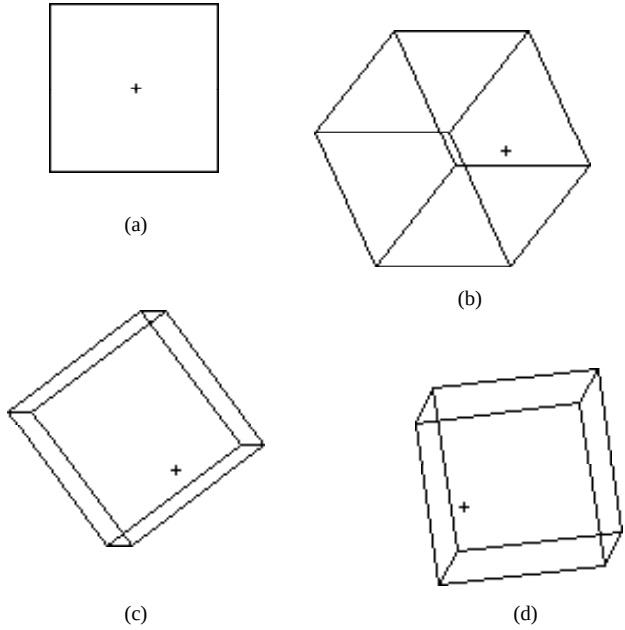

Figure 9.1

Construction of the ACTUAL to OBSERVED Matrix Q

We assume that the eye is at (EX, EY, EZ) relative to the ABSOLUTE axes, looking towards the point (DX, DY, DZ). The OBSERVED position is achieved in the following sequence of steps.

(1) A matrix D translates all the points in space by a vector (−DX, −DY, −DZ) so that now the eye is at (EX − DX, EY − DY, EZ − DZ) = (FX, FY, FZ) say, looking towards the origin:

$$D = \begin{pmatrix} 1 & 0 & 0 & -DX \\ 0 & 1 & 0 & -DY \\ 0 & 0 & 1 & -DZ \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(2) A matrix E changes (FX, FY, FZ) into (r, 0, FZ) by rotating space by an angle $-\alpha$, where $\alpha = \tan^{-1}(FY/FX)$, about the z-axis. Here $r^2 = FX^2 + FY^2$ and $r > 0$:

$$E = \frac{1}{r}\begin{pmatrix} FX & FY & 0 & 0 \\ -FY & FX & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & 0 & 0 & r \end{pmatrix}$$

(3) A matrix F transforms (r, 0, FZ) into (0, 0, $-s$) by rotating space by an angle $\pi - \theta$ about the y-axis – where $\theta = \tan^{-1}(r/FZ)$. Here $s^2 = r^2 + FZ^2 = FX^2 + FY^2$ FZ^2 and $s > 0$:

$$F = \frac{1}{s}\begin{pmatrix} -FZ & 0 & r & 0 \\ 0 & s & 0 & 0 \\ -r & 0 & -FZ & 0 \\ 0 & 0 & 0 & s \end{pmatrix}$$

(4) The transformation thus far places the eye at (0, 0, −s) on the negative x-axis looking towards the origin and at the same distance from it (s) as (EX, EY, EZ) was from (DX, DY, DZ). We now generate a matrix G which moves the eye to the origin:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & s \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(5) If in example 9.1 we now premultiply P = C × B × A by our first approximation to the ACTUAL to OBSERVED matrix Q (= G × F × E × D) to find the SETUP to OBSERVED matrix R = Q × P = G × F × E × D × C × B × A, we draw figure 9.1c by orthographic projection. This view is not really satisfactory because the matrix Q places the cube at an arbitrary orientation within the view plane. It is much better to standardise our view, and one of the most popular ways is to maintain the vertical, that is a line that was vertical (that is, parallel to the *y*-axis) in its ACTUAL position remains vertical after transformation by Q into its OBSERVED position. Take the vertical line from (DX, DY, DZ) to (DX, DY + 1, DZ). Because of this peculiar construction, we note that intermediate matrix K (F × E × D) transforms this line into one that joins (0, 0, 0) to (K(1, 2), K(2, 2), K(3, 2)) = (p, q, r), say. So if we further rotate about the *z*-axis by an angle $\beta = \tan^{-1}(K(1,2)/K(2,2)) = \tan^{-1}(p/q) = \tan^{-1}(-FY \times FZ/(s \times FX))$ using a matrix H, before multiplying by G, then the vertical is maintained:

$$H = \frac{1}{t}\begin{pmatrix} q & -p & 0 & 0 \\ p & q & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{pmatrix}$$

where $t^2 = p^2 + q^2$ and thus

$$H \times \begin{pmatrix} p \\ q \\ r \\ 1 \end{pmatrix} = \frac{1}{t}\begin{pmatrix} q & -p & 0 & 0 \\ p & q & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & t \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ r \\ 1 \end{pmatrix}$$

Thus the complete transformation (figure 9.id) is achieved by the matrix R = Q × P = G × H × F × E × D × C × B × A, and the projection of the line joining points (DX, DY, DZ) to (DX, DY + 1, DZ) is the line joining (0, 0) to (0, *t*) on the screen; that is, the vertical – matrix G does not affect the *x*/*y* values. Note that this technique works in all cases except where (EX, EY, EZ) is vertically above (DX, DY, DZ) to start with, and naturally in this case maintaining the vertical makes no sense. The procedure ‘look3’ (listing 9.1), given (EX, EY, EZ) and (DX, DY, DZ), generates the ACTUAL to OBSERVED matrix in the steps shown above, and at each step premultiplies the matrix R: so at the end of the process R will hold its original matrix value premultiplied by Q. If we wish to store Q explicitly then we need first to set R to the identity matrix (using ‘idR3’), then call ‘look3’, and finally copy array R into array Q. Procedure ‘1ook3’ can be

radically reduced if we assume that the eye always looks at the origin (that is, DX = DY = DZ = 0). Furthermore with the orthographic projection the OBSERVED position of the eye need not be at the origin, it merely needs to be on the *z*-axis: again the procedure can be cut down. We give the general case, which will be essential for later perspective projections

Listing 9.1

```
8200 REM look3 / maintain vertical
8210 DEF PROClook3
8220 LOCAL FX,FY,FZ,THETA
8230 CLS : INPUT"(EX,EY,EZ)",EX,EY,EZ
8240 INPUT"(DX,DY,DZ)",DX,DY,DZ
8250 PROCtran3(-DX,-DY,-DZ) : PROCmult3
8260 FX=EX-DX : FY=EY-DY : FZ=EZ-DZ
8270 THETA=FNangle(FX,FY)
8280 PROCrot3(-THETA,3) : PROCmult3
8290 DIST=SQR(FX*FX+FY*FY)
8300 THETA=FNangle(FZ,DIST)
8310 PROCrot3(PI-THETA,2) : PROCmult3
8320 DIST=SQR(DIST*DIST+FZ*FZ)
8330 THETA=FNangle(DIST*FX,-FY*FZ)
8340 PROCrot3(THETA,3) : PROCmult3
8350 PROCtran3(0,0,DIST) : PROCmult3
8360 ENDPROC
```

If required, we can extend this program to deal with the situation where the head is tilted through an angle γ from the vertical. This is achieved by further rotating space by $-\gamma$ about the *z*-axis. Thus matrix H should then rotate about the *z*-axis by an angle $\beta - \gamma$.

The construction of the ACTUAL to OBSERVED matrix is obviously independent of everything other than the position of the eye, line of sight and the tilt of the head. So if we wish to view a series of objects from the same position, we can store Q and use it repeatedly for placing each object.

How to Define an Object

It is now time to deal with the problem of representing objects to the computer. There is no definite solution, it really depends on what is being drawn and how it is projected. In this section we described various ways of setting up a data-base to hold the information that is necessary for drawing any given scene, but make no comment on their usefulness. This is considered in the remainder of the book where We give examples to illustrate the value of particular methods in different situations. We shall be using arrays to hold large sets of data, and so naturally the amount of space given to arrays will depend on the amount of information that is required for a scene: be sure that when you declare these arrays there is enough space for all the information – if in doubt, overestimate your store requirements.

Vertices
We will always need to define vertices and other special reference points in a scene, and these we store as *x*-coordinates, y-coordinates and *z*-coordinates in arrays X, Y and Z respectively, assuming that if the total number is not known explicitly then this value is calculated as NOV. So there must be space for not less than NOV values in each of the three arrays. These vertices may be in the SETUP, the ACTUAL or the OBSERVED position, it depends on the context of the problem. There will also be situations (perspective in particular) when we need to store the *x*/*y* coordinates of the projections of these NOV vcrtices – in arrays XD and YD. Naturally this is unnecessary in the case of an orthographic projection of points in the OBSERVED position since we can use the values already stored in the X and Y arrays. The choice of data-base really depends on the scene and type of projection.

Lines
We can store information on NOL (say) line segments in the two-dimensional integer array LIN. The Ith line is defined by the integer indices (between 1 and NOV) of the two points at each end of the line – we store the indices in LIN(1, 1) and LIN(2, 1). The true coordinate values of the two points at each end of the line segment can be found from the X, Y and Z arrays. We normally assume that these lines are coloured implicitly by the program, usually black.

Facets
A facet is a convex polygonal area on the surface of a three-dimensional object, and can be defined in a number of ways. Most facets will be triangular or quadrilateral, rarely greater than six-sided, so we usually assume than no facet has greater than six sides in order to minimise waste of store. The NOF facets can be defined in terms of the indices of the vertices at their corners in array FACET: FACET(I, J) is the index of the I^{th} vertex on the J^{th} facet. Naturally if the facet is not hexagonal then some of the values are *garbage* so we need to store array SIZE, the number of vertices/edges on each facet. We can implicitly colour each facet or store it as an integer array COL, and we may implicitly colour the lines that form the edge of the facet. Another method is to store the facet in terms of the indices of the lines in the object in array FACET, which would thus refer to array LIN: FACET(I, J) would now be the index of the I^{th} line on the edge of the J^{th} facet. There are many other methods for representing these, and other elements of a three-dimensional object: you choose the one most suitable to your particular situation.

Construction Procedures and the 'Building Block' Method

For any required object we define a construction procedure that needs as parameters a matrix R to move vertices into position and and any other

information about the size of the object (if the object is to be stored in the SETUP position then naturally no matrix is needed). The procedure can then define the vertices, lines, facets or any other elements of the object, and use the matrix R to move the vertices of the object into the required position. Depending on the context of the program the procedure can then either draw the object, or extend a data-base that contains this information. We shall give examples of both methods.

We can construct a scene that contains a number of similar objects (so the data will be in either the ACTUAL or the OBSERVED position). There is no need to produce a new construction procedure for each occurrence of the object, all we do each time is calculate a new SETUP to ACTUAL matrix P, and enter it (for the ACTUAL position) or Q × P (for the OBSERVED position) into the same procedure. Naturally we shall require one new procedure for each different type of object.

The complete scene is achieved by the execution of a main program (listing 9.2), which INPUTs the MODE of the picture (usually modes 1 or 4), prepares the graphics screen by using input values of HORIZ and VERT, and finally calls a procedure 'scene3' which organises the objects in space and then draws them. The main program below will be used in all the three-dimensional graphics programs that follow, so do not alter it without very good reason.

Listing 9.2

```
100 REM MAIN PROGRAM
110 INPUT"Which mode "MOWD : MODE MOWD
120 INPUT"HORIZ,VERT",HORIZ,VERT
130 PROCstart(3,0)
140 PROCsetorigin(HORIZ/2,VERT/2)
150 PROCscene3
160 STOP
```

'scene3' declares all the arrays that are required for storing information about a scene, together with the matrices A, B, R and (perhaps) Q for moving objects into position. If required the values of NOV and NOL (or NOF) are initialised, and these will be updated in later construction procedures. For each individual object (a 'block'), 'scene3' must calculate a matrix P that moves this block into the ACTUAL position, and then call the construction procedure by using the correct matrix R (perhaps SETUP to ACTUAL or SETUP to OBSERVED). All the blocks finally construct the finished scene. Sometimes the drawing of the projection is done inside the construction procedure, or it can be elsewhere in other procedures that are specifically designed for special forms of drawing (as in hidden line and hidden surface pictures): it depends on what is being drawn and what is required of the view. As usual, because of the restriction of not passing array parameters into procedures, we do not normally explicitly generate P and

Q: we usually rely on updating matrix R. If we require the ACTUAL to OBSERVED matrix then this procedure calls 'look3' . Should we need to store Q then we must first call 'idR3' which sets matrix R to the identity – remember all matrix operations are done via matrices A and R, using matrix B to hold intermediate values.

Our first example of this method is listing 9.3, which is the 'scene3' procedure that is needed to construct a picture of a single cube as shown in figure 9.1d. The scene can be viewed from any position with the vertical maintained. We also have a construction procedure 'cube' (listing 9.4) which generates the data for a cube with sides of length 2. It places the vertices, eight sets of coordinate triples, in arrays X, Y and Z. There is no need to store the lines of the cube explicitly, we get the information from a DATA statement and draw the lines straight away. The data for figure 9.1d are HORIZ = 8, VERT = 6, (EX, EY, EZ) = (–2, 2, 2) and (DX, DY, DZ) = (–1, 0, 0).

Listing 9.3

```
6000 REM scene 3 / cube (example 9.1)
6010 DEF PROCscene3
6020 DIM X(8),Y(8),Z(8)
6030 DIM A(4,4),B(4,4),R(4,4)
6040 PROCidR3
6050 PROCrot3(-0.92729522,3) : PROCmult3
6060 PROCtran3(-1,0,0) : PROCmult3
6070 PROCrot3(0.92729522,2) : PROCmult3
6080 PROClook3
6090 PROCcube
6100 ENDPROC
```

Listing 9.4

```
6500 REM cube / data not stored, lines drawn
6510 DEF PROCcube
6520 LOCALI%,XX,YY,ZZ,L1,L2
6530 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1, -1,1,1, -1,1,-1, -1,-
1,-1, -1,-1,1
6540 DATA 1,2, 2,3, 3,4, 4,1, 5,6, 6,7, 7,8, 8,5, 1,5, 2,6, 3,7,
4,8
6550 RESTORE
6559 REM READ vertex data, transform with matrix R
6560 FOR I%=1 TO 8
6570 READ XX,YY,ZZ
6580 X(I%)=R(1,1)*XX+R(1,2)*YY+R(1,3)*ZZ+R(1,4)
6590 Y(I%)=R(2,1)*XX+R(2,2)*YY+R(2,3)*ZZ+R(2,4)
6600 Z(I%)=R(3,1)*XX+R(3,2)*YY+R(3,3)*ZZ+R(3,4)
6610 NEXT I%
6619 REM draw lines
6620 FOR I%=1 TO 12
6630 READ L1,L2
6640 PROCmoveto(X(L1),Y(L1))
6650 PROClineto(X(L2),Y(L2))
6660 NEXT I%
6670 ENDPROC
```

We could have more than one cube in the scene. For example, should we rewrite ‘scene3’ as in listing 9.5, keeping all the other procedures the same, we would get figure 9.2. Note that the X, Y and Z values of the previous cube are overwritten in the second call to ‘cube' . Also, because we have the same ACTUAL to OBSERVED matrix for both cubes (they have different SETUP to ACTUAL matrices) we need to store Q so that it can also be used for the second cube. Remember Q must premultiply the array P that moves the second cube into the ACTUAL position. The data for figure 9.2 are HORIZ = 8, VERT = 6, (EX, EY, EZ) = (3, 2, 1) and (DX, DY, DZ) = (0, 0, 0).

Listing 9.5

```
6000 REM scene 3 / two cubes not stored
6010 DEF PROCscene3
6020 LOCAL I%,J%
6030 DIM X(8),Y(8),Z(8)
6040 DIM A(4,4),B(4,4),R(4,4),Q(4,4)
6049 REM calculate and store Q, draw first cube
6050 PROCidR3 : PROClook3 : PROCmult3 : PROCcube
6060 FOR I%=1 TO 4 : FOR J%=1 TO 4
6070 Q(I%,J%)=R(I%,J%)
6080 NEXT J% : NEXT I%
6089 REM put cube 2 in ACTUAL position
6090 PROCidR3
6100 PROCtran3(3,1.5,2) : PROCmult3
6109 REM then in OBSERVED position
6110 FOR I%=1 TO 4 : FOR J%=1 TO 4
6120 A(I%,J%)=Q(I%,J%)
6130 NEXT J% : NEXT I%
6139 REM draw second cube
6140 PROCmult3 : PROCcube
6150 ENDPROC
```

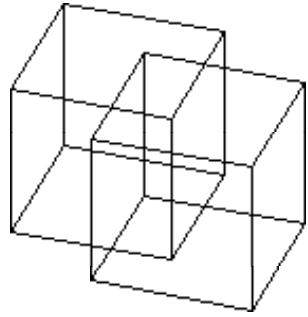

Figure 9.2

Exercise 9.1
Extend procedure 'cube' so that information about the size of a rectangular block is input, so enabling the procedure to construct a block of length LH, breadth BH and height HT: multiply the *x*-values of the SETUP cube by LH/2, the *y*-values by HT/2 and the *z*-values by BH/2.

Again it should be noted that the modular approach we have adopted may not be the most efficient method of drawing three-dimensional pictures. We chose this descriptive method in order to break down the complex situation into manageable pieces. Once the reader has mastered these concepts he should *cannibalise* our programs for the sake of effiiciency. However, to show the value of this modular approach we give another example, which illustrates just how quickly programs can be altered to draw new scenes and situations. As the scenes get more complicated you may run out of store in modes 1 or 0. You should either run your programs in mode 4 (if you need only two-colour output) or load the complete program into store after having set PAGE = &1100, and it is also advisable to delete all REMarks and unused procedures (such as 'triangle' or 'scale').

Example 9.2
We wish to view a fixed scene (for example, the one shown in figure 9.2) from a variety of observation points.

In this case it is better to store the vertex coordinates of the scene in the ACTUAL position, rather than the OBSERVED position, and store the line information in array LIN. The 'scene3' procedure (listing 9.6) must first set NOV and NOL to zero and then place the objects in their ACTUAL position by using matrix R = P. The construction procedure 'cube' (listing 9.7) must therefore be altered to update the data-base (but note that the same procedure could be used to store vertices in their OBSERVED position: it needs only a different R = Q × P). Then for each different view point and direction the 'scene3' procedure must clear the screen, set R to the identity matrix and call 'look3', and then call a special new 'drawit' procedure (listing 9.8) which uses the matrix R (which holds the values of Q, the ACTUAL to OBSERVED matrix) to put the points in the OBSERVED position and orthographically project them into arrays XD and YD (we cannot use X and Y because this would corrupt our ACTUAL data-base). Procedure 'drawit' which was labelled in 'scene3' can then use the information in array LIN to draw the picture on the screen.

If the observer is travelling in a straight line and always looking in the same direction we need not even calculate Q each time, but simply initially manipulate space so that the observer is looking along the *z*-axis; then we can use the 'setorigin' procedure to move the observer instead! After you have gained expertise in drawing three-dimensional projections, you should choose your

Listing 9.6

```
6000 REM scene3 / 2 cubes stored.
6010 DEF PROCscene3
6020 DIM X(16),Y(16),Z(16),XD(16),YD(16)
6030 DIM LIN(2,24),A(4,4),B(4,4),R(4,4)
6039 REM" put cubes in ACTUAL position
6040 NOV=0 : NOL=0
6050 PROCidR3 : PROCcube
6060 PROCtran3(3,1.5,2) : PROCmult3
6070 PROCcube
6079 REM" draw in OBSERVED position
6080 PROCidR3 : PROClook3
6090 PROCdrawit
6100 GOTO 6080
6110 ENDPROC
```

Listing 9.7

```
6500 REM cube / add to data base
6510 DEF PROCcube
6520 LOCAL I%,XX,YY,ZZ,L1,L2
6530 DATA 1,2, 2,3, 3,4, 4,1, 5,6, 6,7, 7,8, 8,5,
         1,5, 2,6, 3,7, 4,8
6540 DATA 1,1,1, 1,1,-1, 1,-1,-1, 1,-1,1,
         -1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1
6550 RESTORE
6559 REM store line information
6560 FOR I%=1 TO 12
6570 READ L1,L2 : NOL=NOL+1
6580 LIN(1,NOL)=L1+NOV : LIN(2,NOL)=L2+NOV
6590 NEXT I%
6599 REM store vertex information put in position by matrix R
6600 FOR I%=1 TO 8
6610 READ XX,YY,ZZ : NOV=NOV+1
6620 X(NOV)=R(1,1)*XX+R(1,2)*YY+R(1,3)*ZZ+R(1,4)
6630 Y(NOV)=R(2,1)*XX+R(2,2)*YY+R(2,3)*ZZ+R(2,4)
6640 Z(NOV)=R(3,1)*XX+R(3,2)*YY+R(3,3)*ZZ+R(3,4)
6650 NEXT I%
6660 ENDPROC
```

Listing 9.8

```
7000 REM drawit
7010 DEF PROCdrawit
7020 LOCAL I%,L1,L2 : CLG
7029 REM put in OBSERVED position
7030 FOR I%=1 TO NOV
7040 XD(I%)=R(1,1)*X(I%)+R(1,2)*Y(I%)+R(1,3)*Z(I%)+R(1,4)
7050 YD(I%)=R(2,1)*X(I%)+R(2,2)*Y(I%)+R(2,3)*Z(I%)+R(2,4)
7060 NEXT I%
7069 REM draw lines of object
7070 FOR I%=1 TO NOL
7080 L1=LIN(1,I%) : L2=LIN(2,I%)
7090 PROCmoveto(XD(L1),YD(L1))
7100 PROClineto(XD(L2),YD(L2))
7110 NEXT I%
7120 ENDPROC
```

construction and viewing method with care. You will rarely need to go through the complete method given in this chapter, there will always be short-cuts.

Exercise 9.2
Produce construction procedures for a tetrahedron, pyramid etc. For example

(a) Tetrahedron: vertices(1, 1, 1), (1, –1, –1), (–1, 1, –1) and (–1, –1, 1); lines 1 to 2, 1 to 3, 1 to 4, 2 to 3, 2 to 4 and 3 to 4.
(b) Pyramid with square of side 1 and height HT: vertices (0, HT, 0), (1, 0, 1), (1 , 0, –1), (–1, 0 –1) and (–1, 0, 1); lines 1 to 2, 1 to 3, 1 to 4,1 to 5, 2 to 3, 3 to 4, 4 to 5 and 5 to 1 .

Exercise 9.3
Set up a line drawing of any planar object in the x/y plane (for example, the outline of an alphabetic character or string of characters) and view them in various orientations in three-dimensional space. You can place such planar objects on the side of a cube. All you need do is extend the 'cube' procedure above to include extra vertices and lines to define the symbols.

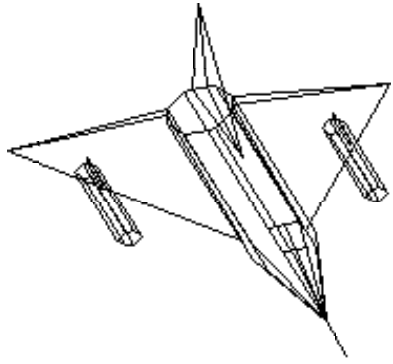

Figure 9.3

Thus far we have restricted our pictures to those of the simple cube. This is so that the methods we give are not obscured by the complexity of defining objects. Our programs will work for any object provided that it fits within the limitations of store (and time) that are available on the BBC micro. For complex objects we merely extend the size of our arrays, although some objects will have properties that enable us to minimise store requirements. Consider the *jet* shown in figure 9.3 – it possesses two-fold symmetry, which can be used to our advantage. We

assume that the plane of symmetry is the *y*/*z* plane, and so for every point (*x*, *y*, *z*) on the jet there is also a corresponding point (–*x*, *y*, *z*). To draw figure 9.3 we use all the graphics and 4 × 4 matrix routines, listing 9.1 and 9.2, together with listing 9.9, 'scene3' and construction procedure 'jet' which generates all the vertices of the aeroplane that have positive *x*-coordinates, and thus stores information only about one-half of the jet. To construct the complete aeroplane we also need a 'drawit' procedure (also in listing 9.9) which draws one side of the jet and then, by reversing the signs of all the *x*-values, draws the other.

It is simple to construct these figures, just plan your object in various sections on a piece of graph paper, number the important vertices and note which pairs of vertices are joined by lines. The coordinate values can be read directly from the grid on the paper. The data for figure 9.3 are HORIZ = 160, VERT = 120, (EX, EY, EZ) = (1, 2, 3) and (DX, DY, DZ) = (0, 0, 0).

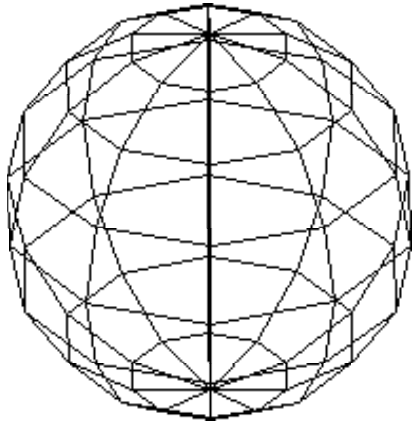

Figure 9.4

Bodies of Revolution

This far in our construction of objects we have relied on DATA to input all the information about lines and vertices. We now consider a type of object where only a small amount of information is required for a quite complex object – this is a body of revolution, an example of which is shown in figure 9.4.

The method is simply to create a defining sequence of NUMV lines in the *x*/*y* plane through the origin; this is called the definition set. We then revolve

Listing 9.9

```
6000 REM scene3 / jet
6010 DEF PROCscene3
6020 DIM X(37),Y(37),Z(37),XD(37),YD(37)
6030 DIM LIN(2,46),A(4,4),B(4,4),R(4,4)
6040 PROCidR3 : PROClook3
6050 PROCjet : PROCdrawit
6060 ENDPROC

6500 REM jet
6510 DEF PROCjet
6520 LOCAL I%
6530 DATA 0,0,80,  0,0,64,  0,8,32,  4,8,32,  8,4,32,  8,0,32,
         4,-4,32,  0,8,-32, 4,8,-32, 8,4,-32, 8,0,-32,  4,-4,-32,
         0,-4,-32,  8,0,24,  48,0,-32,  8,2,-32,  0,8,0,  2,8,-32,
         0,32,-32,  28,-4,-24,  30,-2,-24,  32,-2,-24,  34,-4,-24
6540 DATA 32,-6,-24,  30,-6,-24,  28,-4,8,  30,-2,8,  32,-2,8,
         34,-4,8,  32,-6,8,  30,-6,8,  31,0,-24,  31,-2,-24,
         31,-2,-12,  31,0,-12,  0,6,40,  3,6,40
6550 DATA 1,2,  2,3,  2,4,  2,5,  2,6,  2,7,  3,4,  4,9,  5,10,
         6,11,  7,12,  8,9,  9,10,  10,11,  11,12,  12,13,  14,15,
         15,10,  15,16,  14,16,  17,18,  17,19,  18,19
6560 DATA 20,21,  21,22,  22,23,  23,24,  24,25,  25,20,  26,27,  27,28,
         28,29,  29,30,  30,31,  31,26,  20,26,  21,27,  22,28,  23,29,
         24,30,  25,31,  32,33,  33,34,  34,35,  35,32,  36,37
6570 RESTORE : NOV=37 : NOL=46
6579 REM SETUP vertices and lines
6580 FOR I%=1 TO NOV : READ X(I%),Y(I%),Z(I%) : NEXT I%
6590 FOR I%=1 TO NOL : READ LIN(1,I%),LIN(2,I%) : NEXT I%
6600 ENDPROC

7000 REMdrawit / two halves of jet
7010 DEF PROCdrawit
7020 LOCAL I%,J%,S%,XX,YY,ZZ,L1,L2
7030 S%=1
7039 REM loop through two halves
7040 FOR J%=1 TO 2
7049 REM vertices in OBSERVED position
7050 FOR I%=1 TO NOV
7060 XX=S%*X(I%) : YY=Y(I%) : ZZ=Z(I%)
7070 XD(I%)=R(1,1)*XX+R(1,2)*YY+R(1,3)*ZZ+R(1,4)
7080 YD(I%)=R(2,1)*XX+R(2,2)*YY+R(2,3)*ZZ+R(2,4)
7090 NEXT I%
7099 REM draw lines
7100 FOR I%=1 TO NOL
7110 L1=LIN(1,I%) : L2=LIN(2,I%)
7120 PROCmoveto(XD(L1),YD(L1))
7130 PROClineto(XD(L2),YD(L2))
7140 NEXT I%
7150 S%=-1
7160 NEXT J%
7170 ENDPROC
```

this set about the vertical (*y*-axis) NUMH – further times to create new vertical sets. The NUMV lines in the definition set are formed by joining the NUMV + 1 vertices (XD(1), YD(1), 0) (where $1 \leq I \leq NUMV + 1$) in order. From this we generate NUMH different vertical sets: the J^{th} vertical set is the definition set

rotated through an angle PHI + 2π(J – 1)/NUMH about the vertical *y*-axis, for some input value PHI(f). As well as the set of NUMH xx NUMV vertical lines we also introduce horizontal lines. We consider a single point (XD(I), YD(I), 0) at the end of a line segment in the definition set: as we rotate about the vertical axis it moves into NUMH positions (provided that the point is not on the axis of revolution):

(XD(I) × cos(θ + ϕ), YD(I), XD(I) × sin (θ + ϕ)) where
θ = 2π(J – 1)/NUMH with I ≤ J ≤ NUMH

These NUMH points are joined in order, and the NUMHth position is joined back to the first, to give the Ith horizontal set. So there are (NUMH – *n*) × NUMV horizontal lines, where n is the number of vertices on the axis of rotation. Listing 9.10 is a construction procedure 'revbod', which draws the body of revolution when given NUMV, NUMH, PHI, the original set of vertices in XD and YD and the positional matrix R. Listing 9.11 is the 'scene3' procedure which creates the scene of a spheroid in figure 9.4 by placing eight points from a semicircle into the definition set: HORIZ = 3.2, VERT = 2.4, PHI = π/25, NUMH = 10, NUMV = 8, viewed from (1, 2, 3) looking at (0, 0, 0).

Listing 9.10

```
6500 REM revbod / body of revolution
6510 DEF PROCrevbod
6520 LOCAL I%,J%,THETA,TD,N1,C,S,XX,YY,ZZ
6530 THETA=PHI : TD=PI*2/NUMH
6540 N1=NUMV+1 : C=COS(PHI) : S=SIN(PHI)
6550 FOR I%=1 TO N1
6560 XX=XD(I%)*C : YY=YD(I%) : ZZ=XD(I%)*S
6570 X(I%)=R(1,1)*XX+R(1,2)*YY+R(1,3)*ZZ+R(1,4)
6580 Y(I%)=R(2,1)*XX+R(2,2)*YY+R(2,3)*ZZ+R(2,4)
6590 NEXT I%
6599 REM loop thru second vertical set
6600 FOR J%=1 TO NUMH
6610 THETA=THETA+TD : C=COS(THETA) : S=SIN(THETA)
6620 FOR I%=1 TO N1
6630 XX=XD(I%)*C : YY=YD(I%) : ZZ=XD(I%)*S
6640 X(I%+N1)=R(1,1)*XX+R(1,2)*YY+R(1,3)*ZZ+R(1,4)
6650 Y(I%+N1)=R(2,1)*XX+R(2,2)*YY+R(2,3)*ZZ+R(2,4)
6660 NEXT I%
6669 REM join vertical lines
6670 PROCmoveto(X(1),Y(1))
6680 FOR I%=2 TO N1
6690 PROClineto(X(I%),Y(I%))
6700 NEXT I%
6709 REM join horizontal lines
6710 FOR I%=1 TO N1
6720 PROCmoveto(X(I%),Y(I%))
6730 PROClineto(X(I%+N1),Y(I%+N1))
6739 REM second set becomes first set
6740 X(I%)=X(I%+N1) : Y(I%)=Y(I%+N1)
6750 NEXT I% : NEXT J%
6760 ENDPROC
```

Listing 9.11

```
6000 REM scene3 / spheroid
6010 DEF PROCscene3
6020 LOCAL I%,THETA,TD
6030 DIM X(32),Y(32),XD(16),YD(16)
6040 DIM A(4,4),B(4,4),R(4,4)
6050 INPUT"NUMBER OF HORIZONTAL LINES",NUMH
6060 INPUT"NUMBER OF VERTICAL LINES",NUMV
6070 INPUT"INITIAL ROTATION",PHI
6080 THETA=PI/2 : TD=PI/NUMV
6089 REM definition set is semicircle
6090 FOR I%=1 TO NUMV+1
6100 XD(I%)=COS(THETA) : YD(I%)=SIN(THETA)
6110 THETA=THETA+TD
6120 NEXT I%
6130 PROCidR3 : PROClook3
6140 PROCrevbod
6150 ENDPROC
```

Exercise 9.4

Experiment with this technique – any line sequence will do. Try an ellipsoid: this is essentially the same as the spheroid except that the definition set is produced from a semi-ellipse rather than a semicircle. There is no need to produce only convex bodies: lines can cut one another or cross to and fro over the *y*-axis, and *x*-values can move up and down.

This idea can be extended into a *body of rotation*. Now as the set of lines moves around the central axis, the*y*-values of the points do not stay fixed. They can move in a regular manner, that is they can drop by the same amount with each rotation through 2π/NUMH. Now, of course, the lines may make more than one complete rotation about the axis – see figure 9.5. Write a program to implement a body of rotation.

Figure 9.5

Animation of Line Drawings

We can animate simple line drawings like those created in this chapter by using the method of redefining the logical-actual colour relationships. The technique is to produce *n* (for some even integer *n*) separate pictures of an object in different positions. We have a white background (logical colour binary 11) and a black foreground (logical 00). The object will be drawn in logical colours 0l or 10; at any time one logical colour will be set to actual white and the other to actual black. By ANDing a picture on to the screen in white an invisible picture will be drawn, which can be made visible later by redefining the logical colour to be actual black. ANDing a white line over a black pixel will leave the black pixel on the screen as required. ORing the same picture on to the screen in the opposite logical colour to which it was originally drawn will delete the picture from the screen memory (whether it be visible or invisible). This deletion will not leave holes in the visible lines from the other views still on the screen. This gives us a simple method:

(1) With the $(i-1)^{th}$ picture visible draw the i^{th} view so that it is invisible.
(2) Make the i^{th} picture visible and the $(i-1)^{th}$ invisible by redefining the logical-actual relationship.
(3) Delete the $(i-1)^{th}$ view when it is invisible.

Here *i* varies from 1 to *n* (the number of views). If the views are such that the $(n+1)^{th}$ view (if there was one) is the same as the first then we have an infinite movie.

Listing 9.3
Listing 9.12 gives an implementation of this method for drawing a rotating cube. We change the 'look3' routine so that each time it is called the observer moves to a different position relative to the object. The 'scene3' routine sets up a SPOOL file called ROTCUB on backing store to hold all the graphics commands (the program is too slow to draw the figures in real-time animation). It also uses the 'drawit' routine which draws a 'cube' setup by the procedure from listing 9.7, and the 'lib1' and 'lib3' routines (excluding 'look3' naturally). By typing in the instructions below you load ROTCUB file back into store and then execute the commands in sequence over and over again to get a non-stop movie:

```
*OPT 1, 2: PAGE = &1900

*LOAD ROTCUB 1900
```

The size of the file will be displayed on the screen: in this case &C6C

```
MODE 1: GCOL 0, 131: CLG

REPEAT: FOR I% = 0 to &C6B: VDU I%?&1900: NEXT I%: UNT1L FALSE
```

Listing 9.12

```
6000 REM scene3
6010 DEF PROCscene3
6020 DIM X(8),Y(8),Z(8),XD(8),YD(8),LIN(2,12)
6030 DIM A(4,4),B(4,4),R(4,4)
6040 NOV=0 : NOL=0 : COLOR=2 : OTHER=1
6050 PROCidR3 : PROCcube
6060 *SPOOL"ROTCUB"
6070 FOR A=0.05 TO PI/2 STEP PI/20
6080 GCOL2,COLOR : AL=A
6090 VDU19,COLOR,7,0,0,0,19,OTHER,0,0,0,0
6100 PROCidR3 : PROClook3 : PROCdrawit
6110 VDU19,COLOR,0,0,0,0,19,OTHER,7,0,0,0
6120 AL=A-PI/20
6130 PROCidR3 : PROClook3
6140 GCOL1,COLOR : PROCdrawit
6150 OTHER=COLOR : COLOR=3-COLOR
6160 NEXT A
6170 *SPOOL
6180 ENDPROC

7000 REM drawit
7010 DEF PROCdrawit
7020 LOCAL I%
7030 FOR I%=1 TO NOV
7040 XD(I%)=R(1,1)*X(I%)+R(1,2)*Y(I%)+R(1,3)*Z(I%)+R(1,4)
7050 YD(I%)=R(2,1)*X(I%)+R(2,2)*Y(I%)+R(2,3)*Z(I%)+R(2,4)
7060 NEXT I%
7070 FOR I%=1 TO NOL
7080 L1=LIN(1,I%) : L2=LIN(2,I%)
7090 PROCmoveto(XD(L1),YD(L1))
7100 PROClineto(XD(L2),YD(L2))
7110 NEXT I%
7120 ENDPROC

8200 REM look3
8210 DEF PROClook3
8219 REM adjusted look 3 routine. Observer moves in a circle
        around the origin making an angle AL with the +ve
        x-axis looking at the origin.
8220 EY=2 : EX=SQR(10)*COS(AL) : EZ=SQR(10)*SIN(AL)
8230 DX=0 : DY=0 : DZ=0
8240 PROCtran3(DX,DY,DZ) : PROCmult3
8250 FX=EX-DX : FY=EY-DY : FZ=EZ-DZ
8260 THETA=FNangle(FX,FY)
8270 PROCrot3(-THETA,3) : PROCmult3
8280 DIST=SQR(FX*FX+FY*FY)
8290 THETA=FNangle(FZ,DIST)
8300 PROCrot3(PI-THETA,2) : PROCmult3
8310 THETA=FNangle(R(2,2),R(1,2))
8320 PROCrot3(THETA,3) : PROCmult3
8330 DIST=SQR(DIST*DIST+FZ*FZ)
8340 PROCtran3(0,0,DIST) : PROCmult3
8350 ENDPROC
```

Complete Programs

From now on we shall refer to listings 3.3 (‘angle’), 8.1 (‘mult3’ and ‘idR3’), 8.2 (‘tran3’), 8.3 (’scale3’), 8.4 (‘rot3’), 9.1 (‘1ook3’) and 9.2 (‘main program’) as ‘lib3' . Also from now on it is best to load programs with PAGE = &1100.

I ‘lib1’, ‘lib3’ and listings 9.3 (‘scene3’) and 9.4 (‘cube’). Data required: mode, HORIZ, VERT, (EX, EY, EZ) and (DX, DY, DZ). Try 4, 6, 4, (1, 2, 3), (−1, 0, 1). Also use modes 0 and 1.

II ‘lib1’, ‘lib3’ and listings 9.5 (‘scene3’) and 9.4 (‘cube’). Data required: mode, HORIZ, VERT, (EX, EY, EZ) and (DX, DY, DZ). Try 4, 8, 6, (1, 2, 3), (−1, 0, 1). Make systematic changes to one of these input values and keep all the other parameters fixed.

III ‘lib1’, ‘lib3’ and listings 9.6 (‘scene3’), 9.7 (‘cube’) and 9.8 (‘drawit’). Data required: mode, HORIZ, VERT, and then repeated input of(EX, EY, EZ) and (DX, DY, DZ). Try 4, 8, 6, then (1, 2, 3), (−1, 0, 1); (3, 2, 1), (0, 0, 1). Again make systematic changes to one of the input parameters.

IV ‘lib1’, ‘1ib3’ and listing 9.9 (‘scene3’, ‘jet’ and ‘drawit’). Data required: mode, HORIZ, VERT, and then (EX, EY, EZ) and (DX, DY, DZ). Try 4, 200, 150, then (1, 2, 31), (−1, 0, 30) or (3, 2, 20), (0, 0, 21).Again make systematic changes to one of the input parameters.

V ‘lib1’, ‘lib3’ and listings 9.10 (‘scene3’) and 9.11 (‘revbod’). Data required: mode, HORIZ, VERT, NUMH, NUMV, PHI, (EX, EY, EZ) and (DX, DY, DZ). Try 1, 3.2, 2.4, 10, 10, 1, (1, 2, 3), (0, 0, 0); (3, 2, 1), (0, 0, 0).

VI ‘lib1’, ‘lib3’ (minus ‘1ook3’), listings 9.7 (‘cube’) and 9.12 (‘scene3’, ‘drawit’ and ‘look3’). Data required: mode, HORIZ, VERT. Try 1, 6, 4. This will create a file ROTCUB on backing store. Then type

```
*OPT 1, 2: PAGE = &1900

*LOAD ROTCUB 1900

MODE 1: GCOL 0, 131: CLG

REPEAT: FOR I% = 0 TO &C6B: VDU I%?&1900: NEXT I%: UNT1L FALSE
```