# *Matrix Representation of Transformations on Two-Dimensional Space*

In chapter 2 we saw the need to translate pictures of objects about the screen. Rather than perpetually to change the screen coordinate system, it is conceptually much easier to define an object in the simplest terms possible (as vertices in the form of pixel or coordinate values, together with line and area information that is related to the vertices), and then transform the object to various parts of the screen but keeping the screen coordinate system fixed. We shall restrict ourselves to linear transformations (see below). It will often be necessary to transform a large number of vertices, and to do this efficiently we use matrices. Before looking at such matrix representations we should explain exactly what we mean by a matrix, and also by a *column vector*. In fact we restrict ourselves to square matrices: to 3 × 3 (said 3 by 3) for the study of two-dimensional space, and later we use 4 × 4 matrices when considering three-dimensional space. Such a 3 × 3 matrix (A say) is simply a group of real numbers placed in a block of 3 rows by 3 columns: a column vector (D say) is a group of numbers placed in a column of 3 rows:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}$$

A general entry in the matrix is usually written $A_{ij}$ the first subscript denotes the ith row, and the second subscript the $j^{\text{th}}$ column (for example, $A_{23}$ represents the value in the second row of the third column). The entry in the column vector, $D_i$, denotes the value in the $i^{\text{th}}$ row. All these named entries will be explicitly replaced by numerical values and it is important to realise that the *information* stored in a matrix or column vector is not just the individual values but it is also the position of these values within the matrix or vector. Naturally BASIC programs are written along a line (no subscripts or superscripts) and hence matrices and vectors are implemented as arrays and the subscript values appear inside round brackets following the array identifier.

Matrices can be added. Matrix C = A + B, the sum of two matrices A and B, and is defined by the general entry Cij; thus:

$$C_{ij} = A_{ij} + B_{ij} \quad 1 \le i,\ j \le 3$$

Matrix A can be multiplied by a scalar $k$ to form a matrix B:

$$B_{ij} = k \times A_{ij} \quad 1 \le i, j \le 3$$

We can multiply a matrix A by a column vector D to produce another column vector E thus

$$E_i = A_{i1} \times D_1 + A_{i2} \times D_2 + A_{i3} \times D_3 = \sum_k A_{ik} \times D_k \quad \text{where } 1 \le i, k \le 3$$

The $i^{\text{th}}$ row element of the new column vector is the sum of the products of the corresponding elements of the $i^{\text{th}}$ row of the matrix with those in the column vector.

Furthermore, we can calculate the product (matrix) C = A × B of two matrices A and B:

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + A_{i3} \times B_{3j} = \sum_k A_{ik} \times B_{kj} \quad \text{where } 1 \le i, j, k \le 3$$

We take the sum (in order) of the elements in the $i^{\text{th}}$ row of the first matrix multiplied by the elements in the $j^{\text{th}}$ column of the second. It should be noted that the product of matrices is not necessarily commutative, that is A × B need not be the same as B × A. For example

$$\begin{pmatrix} 0&1&0 \\ 0&0&1 \\ 1&0&0 \end{pmatrix} \times \begin{pmatrix} 0&0&1 \\ 0&1&0 \\ 1&0&0 \end{pmatrix} = \begin{pmatrix} 0&1&0 \\ 1&0&0 \\ 0&0&1 \end{pmatrix} \text{ but } \begin{pmatrix} 0&0&1 \\ 0&1&0 \\ 1&0&0 \end{pmatrix} \times \begin{pmatrix} 0&1&0 \\ 0&0&1 \\ 1&0&0 \end{pmatrix} = \begin{pmatrix} 1&0&0 \\ 0&0&1 \\ 0&1&0 \end{pmatrix}$$

Experiment with these ideas until you have enough confidence to use them in the theory that follows. For those who want more details about the theory of matrices we recommend books by Finkbeiner (1978) and by Stroud (1982).

There is a special matrix called the *indentity matrix* I (sometimes called the unit matrix) :

$$I = \begin{pmatrix} 1&0&0 \\ 0&1&0 \\ 0&0&1 \end{pmatrix}$$

Also for every matrix A we can calculate its *determinant* det(A):

$$\begin{aligned} \det(A) = {} & A_{11} \times (A_{22} \times A_{33} - A_{23} \times A_{32}) + A_{12} \times (A_{23} \times A_{31} - A_{21} \times A_{33}) \\ & + A_{13} \times (A_{21} \times A_{22} - A_{22} \times A_{31}) \end{aligned}$$

Any matrix whose determinant is non-zero is called *non-singular*, and those whose determinant is zero are called *singular*. All non-singular matrices A have an *inverse* $A^{-1}$, which has the property that $A \times A^{-1} = I$ and $A^{-1} \times A = I$. For methods of calculating an inverse of a matrix see Finkbeiner (1978); also see listing 7.4 in chapter 7 which uses the Adjoint method.

We shall now consider the transformation of points in space. Suppose a point $(x, y)$ - 'before' - is transformed to $(x', y')$ - 'after'. We shall completely understand the transformation if we can find equations that relate the 'before' and 'after' points. A linear transformation is one that defines the 'after' point in terms of linear combinations of the coordinates of the 'before' point (that is, the equations contain only multiples of $x$, $y$ and additional real values); the transformation includes neither non-unit powers, nor multiples of $x$ and $y$, nor other variables. Such equations may be written as

$$x' = A_{11} \times x + A_{12} \times y + A_{13}$$

$$y' = A_{21} \times x + A_{22} \times y + A_{23}$$

The A values are called the codficients of the equation. As we can see, the result of the transformation is a combination of multiples of $x$-values, $y$-values and unity. We may add another equation:

$$1 = A_{31} \times x + A_{32} \times y + A_{33}$$

For this to be true for all values of $x$ and $y$, we see that $A_{31} = A_{32} = 0$ and $A_{33} = 1$. Although this may seem a pointless exercise, we shall see that it is in fact very useful. For if we set each point vector $(x, y)$ (also called a *row vector* for obvious reasons) in the form of a three-dimensional column vector

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

then the above three equations can be written in the form of a matrix multiplied by a column vector:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

So if we store the transformation as a matrix, we can transform every required point by considering it to be a column vector and premultiplying this by the matrix.

Many writers ofbooks on computer graphics do not like the use ofcolumn vectors. They prefer to extend the row vector, that is (x, y), to (x, y, 1) and post-multiply the row vector by the matrix so that the above equations in matrix form become

$$(x', y', 1) = (x, y, 1) \quad \times \begin{pmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$$

Note that this matrix is the *transpose* of the matrix of coefficients in the equations. This causes a great deal of confusion among those who are not confident in the use of matrices. It is for this reason that we keep to the column vector notation in this book. As you get more practice in the use of matrices it is a good idea to rewrite some (or all) of the following transformation procedures in the other notation. It is not really important which method you finally use *as long as you are consistent*. (Note that the transpose B of a matrix A is given by $B_{ij} = A_{ij}$ where $1 \le i, j \le 3$.)

## Combinations of Transformations

A very useful property of this matrix representation of transformations is that if we wish to combine two transformations, say transformation (= matrix) A followed by transformation B, then the combined transformation is represented by their product C = B × A. Note the order of multiplication - the matrix that represents the first transformation is *premultiplied* by the second. This is because the final matrix will be used to premultiply a column vector that represents a point, and so the first transformation matrix must appear on the right of the product and the last on the left. (If we had used the row vector method then the product would appear in the *natural order* from left to right - this is the price we pay for identifying the transfonnation matrix with the coefficients ofthe equation.)

So we need to introduce a procedure ‘mult2’ (see listing 4.1) which forms the product of two matrices. The BASIC computer language does not allow the transmission of array parameters into procedures, so we must invent an efficient means of coping with this limitation. We assume that all matrix multiplication operates on matrices A and R to give the produht matrix B, and when the product is obtained B is copied back into R. The reason for the choice of identifiers and the final copy will become evident as we progress. We also need a procedure ‘idR2’ (see listing 4.1) which sets R to the identity matrix. Should we need to form the product ofa sequence of matrices we first set R = I and then for each of the matrices, from right to left, we name each A and call the procedure ‘mult2’ in turn. At the end of the process R contains the matrix product of the sequence.

All natural transformations may be reduced to a combination of three basic forms of linear transformation: translation, scaling and rotation about the coordinate origin. It should also be noted that all valid applications of these transformations return non-singular matrices. The procedures that follow

*Listing 4.1*

```
9100 REM mult2
9110 DEF PROCmult2
9120 LOCAL I%,J%,K%
9130 FOR I%=1 TO 2
9140 FOR J%=1 TO 3
9150 B(I%,J%)=A(I%,1)*R(1,J%)+A(I%,2)*R(2,J%)
9160 NEXT J%
9170 B(I%,3)=B(I%,3)+A(I%,3)
9210 NEXT I%
9220 FOR I%=1 TO 2
9230 FOR J%=1 TO 3
9240 R(I%,J%)=B(I%,J%)
9250 NEXT J%
9260 NEXT I%
9270 ENDPROC

9300 REM idR2
9310 DEF PROCidR2
9320 R(1,1)=1 : R(1,2)=0 : R(1,3)=0
9330 R(2,1)=0 : R(2,2)=1 : R(2,3)=0
9340 ENDPROC
```

generate a matrix called A for each of the three types of transformation, so that each transformation procedure can be used in conjunction with ‘mult2’ to produce combinations of transformations.

## Translation

A ‘before’ point (x, y) is moved by a vector (TX, TY) to (x' , y' ) say. This produces the equations

$$x' = 1 \times x + 0 \times y + \mathrm{TX}$$

$$y' = 0 \times x + 1 \times y + \mathrm{TY}$$

so the matrix that describes this transformation is

$$\begin{pmatrix} 1 & 0 & \mathrm{TX} \\ 0 & 1 & \mathrm{TY} \\ 0 & 0 & 1 \end{pmatrix}$$

A procedure, ‘tran2’, for generating such a matrix A given the values TX and TY is given in listing 4.2.

*Listing 4.2*

```
9000 REM tran2
9010 DEF PROCtran2(TX,TY)
9020 A(1,3)=TX : A(2,3)=TY
9030 A(1,1)= 1 : A(1,2)= 0
9040 A(2,1)= 0 : A(2,2)= 1
9050 ENDPROC
```

### Scaling

The *x*-coordinate of a point in space is scaled by a factor SX, and the *y*-coordinate by SY, thus

$$x' = \mathrm{SX} \times x + 0 \times y + 0$$

$$y' = 0 \times x + \mathrm{SY} \times y + 0$$

giving the matrix

$$\begin{pmatrix} \mathrm{SX} & 0 & 0 \\ 0 & \mathrm{SY} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Usually SX and SY are both positive, but if one or both are negative this creates a reflection as well as a scaling. In particular, if SX = –1 and SY = 1 then the point is reflected about the *y*-axis. A program segment, ‘scale2’, to produce such a scaling matrix A given SX and SY is given in listing 4.3.

*Listing 4.3*

```
8900 REM scale2
8910 DEF PROCscale2(SX,SY)
8920 LOCAL I%,J%
8930 FOR I%=1 TO 3
8940 FOR J%=1 TO 3
8950 A(I%,J%)=0
8960 NEXT J%
8970 NEXT I%
8980 A(1,1)=SX : A(2,2)=SY : A(3,3)=1
8990 ENDPROC
```

## Rotation about the Origin

If we rotate a point in an anticlockwise direction (the normal mathematical orientation) about the origin by an angle θ then the equations are

$$x' = \sin\theta \times x - \sin\theta \times y + 0$$

$$y' = \sin\theta \times x - \cos\theta \times y + 0$$

and the matrix is

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

The procedure, ‘rot2’, to produce a rotation matrix, A, for an angle θ is given in listing 4.4.

*Listing 4.4*

```
8600 REM rot2
8610 DEF PROCrot2(THETA)
8620 LOCAL I%,J%
8630 FOR I%=1 TO 3
8640 FOR J%=1 TO 3
8650 A(I%,J%)=0
8660 NEXT
8670 NEXT
8680 A(3,3)=1
8690 CT=COS(THETA) : ST=SIN(THETA)
8700 A(1,1)=CT : A(2,2)=CT
8710 A(1,2)=-ST : A(2,1)=ST
8720 ENDPROC
```

## Inverse Transformations

For every transformation there is an inverse transformation that will restore the points in space to their original position. If a transfomiation is represented by a matrix A, then the inverse transformation is represented by the inverse matrix $A^{-1}$. There is no need to calculate this inverse by using listing 7.4, we can find it directly by using listings 4.2, 4.3 and 4.4, with parameters derived from the parameters of the original transformation:

(1) A translation by (TX, TY) is inverted by a translation by ( TX, –TY).
(2) A scaling by SX and SY is inverted by a scaling by 1/SX and 1/SY (naturally both SX and SY are non-zero, for otherwise the two-dimensional space would contract into a line or a point).
(3) A rotation by an angle θ is inverted by a rotation by an angle –θ.

(4) If the transformation matrix is a product of a number of translation, scaling and rotation matrices $A \times B \times C \times \ldots \times L \times M \times N$ (say), then the inverse transformation matrix is

$$N^{-1} \times M^{-1} \times L^{-1} \times \ldots C^{-1} \times B^{-1} \times A^{-1}$$

Note the order of multiplication!

### The Placing of an Object

We are often required to draw a given object at various points on the screen, and arbitrary orientations. It would be very inefficient to calculate by hand the coordinates of vertices for each position of the object and input them to the program. Instead we first define an arbitrary but fixed coordinate system for two-dimensional space, which we shall call the ABSOLUTE system. Then we give the coordinates of the vertices of the object in some simple way, usually about the origin, which we call the SETUP position. Lines and areas within the object are defined in terms of the vertices. We can then use matrices to move the vertices of the object from the SETUP to the ACTUAL position in the ABSOLUTE system. The lines and areas maintain their relationship with the now transformed vertices. The matrix that relates the SETUP to the ACTUAL position will be called P throughout this book (we sometimes give it a letter subscript to identify it uniquely from other such matrices). Because of the restriction of not passing arrays as parameters into subprograms, we shall not normally explicitly generate array P, instead it will be implicitly used to update the array R.

### Looking at the Object

Thus objects in a scene can be moved relative to the ABSOLUTE coordinate axes. When observing such a scene, the eye is assumed to be looking directly at at (DX, DY) of the ABSOLUTE system and the head tilted through an angle a (ALPHA). It would be convenient to assume that it is looking at the origin and there is no tilt of the head (we call this the OBSERVED position). Therefore we generate another matrix that will transform space so that the eye is moved from its ACTUAL position to this OBSERVED position. The ACTUAL to OBSERVED matrix is named Q throughout this book, and is achieved by first translating all points in space by a vector (–DX, –DY), matrix A, and then rotating them by an angle $-\alpha$, matrix B (note the minus signs!). Thus $Q = B \times A$, which is generated in procedure ‘look2’, given as listing 4.5. Normally we do not calculate Q explicitly since, as usual, it is used to update R; however, if it is necessary to use the values of the matrix repeatedly then obviously it is sensible to store Q.

*Listing 4.5*

```
8200 REM look2
8210 DEF PROClook2
8220 CLS : INPUT"(DX,DY) ",DX,DY
8230 INPUT"ALPHA ",ALPHA
8240 PROCtran2(-DX,-DY) : PROCmult2
8250 PROCrot2(-ALPHA) : PROCmult2
8260 ENDPROC
```

### Drawing an Object

Combining the SETUP to ACTUAL matrix P, with the ACTUAL to OBSERVED matrix Q, we get the SETUP to OBSERVED matrix R = Q × P (we shall always use R to denote this matrix _ and remember R is always the result of our ‘mult2’ procedure). transforming all the SETUP vertices by R, with the corresponding movement ofline and area information, means that the coordinates of the object are given relative to the observer who is looking at the origin of the ABSOLUTE coordinate system with head upright, and who is in fact really looking at a graphics screen. So we identify the ABSOLUTE coordinate system with the system of the screen to find the position of the vertices on the screen, and then draw the vertices, lines and areas that compose the object. In practice this is achieved by a construction procedure which uses matrix R. It will set up the vertex, line and area information, transform the vertices by using R, and perhaps finally draw the object; see exatnple 4.1 below. Later we shall see that there are certain situations where it is more elcient to store the vertex, line and area information. For exanple, the vertex coordinates can be stored in arrays X and Y, line information in a two-dimensional array LIN or area information in a two-dimensional array FACET. Vertices may be stored in their SETUP, ACTUAL or OBSERVED position - it really depends on the context of the program. This SETUP to ACTUAL to OBSERVED method will enable us to draw a dynamic series of scenes - objects can move relative to the ABSOLUTE axes, and to themselves, while simultaneously the observer can move independently around the scene. To start with, however, we shall consider the simplest case of a fixed scene.

### Complicated Pictures - the ‘Building Block’ Method

We can draw pictures that contain a number of similar objects. There is no need to produce a new procedure for each occurrence of the object, all we do each time is to calculate a new SETUP to OBSERVED matrix and enter this into the same procedure. Naturally we shall require one procedure for each new type of object in the picture. The final picture is achieved by the execution of a procedure that is named ‘scene2’ which will be called from the standard main program (listing 4.6). This main program defines the MODE of the picture, centres the graphics area after having input HORIZ and VERT, and then calls ‘scene2’.

*Listing 4.6*

```
100 REM MAIN PROGRAM
110 MODE 1
120 INPUT"HORIZ,VERT",HORIZ,VERT
130 PROCstart(3,0)
140 PROCsetorigin(HORIZ/2,VERT/2)
150 PROCscene2
160 STOP
```

‘scene2’ declares all the necessary arrays and then, if required, calls ‘look2’ to generate Q; if more than one object is to be drawn then we store Q. For each individual object (or block) we calculate a matrix P and call the required construction procedure using R = Q × P. All the blocks finally build into the finished picture. To distinguish between different occurrences of these matrices in what follows, we sometimes add a subscript to the names P and R.

This modular approach for solving the problem of defining and drawing a picture may not be the most efficient, but from our experience it does greatly clarify the situation for beginners, enabling them to ask the right questions about constructing a required scene. Also when dealing with animation we shall see that this approach minimises problems in scenes where not only are the objects moving relative to one another, but also the obsever himself is moving. Naturally if the head is upright then matrix Q can be replaced by a call to ‘setorigin’ which changes the screen coordinate system. Or if the eye is looking at the origin, head upright, then Q is the identity matrix I, so it plays no part in transforming the picture and the ‘look2’ procedure may be ignored. We shall make no such assumptions and work with the most general situation: it is a useful exercise throughout this book for the reader to cannibalise our programs in order to make them more efficient for specific cases. It is our aim to explain the concepts in the most general and straightforward terms, even if it is at the expense of efficiency and speed. The reader can return to these programs when he is ready and fully understands the ideas of transforming space. Later we shall give some hints on how to make these changes, but at the moment this would only confuse the issue.

However, the most important reason for this modular approach will be seen when we come to drawing pictures of three-dimensional objects. We shall define these three-dimensional constructions as an extension of the ideas above and full understanding of two-dimensional transformations is essential before we can go on to higher dimensions.

***Example 4.1***

Consider a simple flag SETUP that consists of three coloured areas and two lines that are defined by vertices (labelled 1 to 12) taken from the set (5, 5), (−5, 5),

(−5, −5), (5, −5), (4, 5), (−4, 5), (−5, 4), (−5, −4), (−4, −5), (4, −5), (5, −4) and (5, 4). The three areas (or facets) are given by vertices 1, 2, 3, 4 (facet 1), 1, 5, 8, 3, 9, 12 (facet 2) and 2, 7, 10, 4, 11, 6 (facet 3). The two lines are given by vertices 1, 3 (line 1) and 2, 4 (line 2). This information is stored in a DATA statement and recalled when required. See figure 4.1 , which shows a flag that was drawn on a screen 16 units by 12 units; the SETUP to ACTUAL matrix is the identity and the ACTUAL to OBSERVED matrix is such that the observer is looking at the origin with head upright. Listing 4.7 gives the necessary procedure ‘scene2’ which moves the object into position and takes a general view, and listing 4.8 is the required construction procedure ‘flag’. Note that ‘flag’, which uses matrix R to transform the vertices (and hence the object) into their OBSERVED position, does not store the vertex values for this position in a permanent data-base. Instead the values are kept in arrays X and Y for the duration of the procedure and if the procedure is re-entered to draw another flag then these array locations are used again.

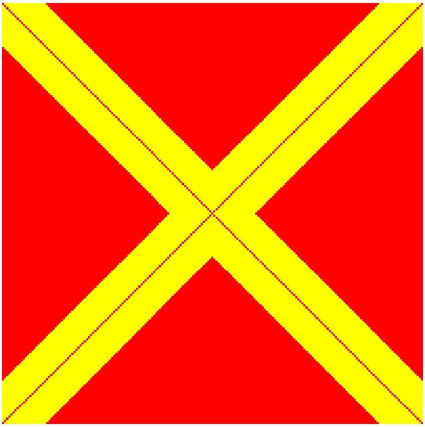


*Figure 4.1*

Listing 4.7

```
6000REM"scene 2 / flag not stored, single view
6010DEF PROCscene2
6020DIM X(12),Y(12),A(3,3),B(3,3),R(3,3)
6030PROCidR2 : PROClook2
6040PROCflag
6050ENDPROC
```

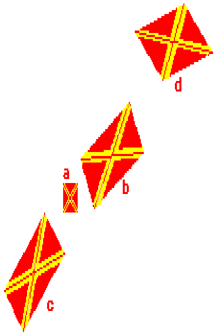


*Figure 4.2*

*Listing 4.8*

```
6500 REM flag / data not stored
6510 DEF PROCflag
6520 LOCAL I%,XX,YY
6530 RESTORE 6540
6540 DATA 5,5, -5,5, -5,-5, 5,-5, 4,5, -4,5, -5,4, -5,-4, -4,-5,
          4,-5, 5,-4, 5,4
6550 FOR I%=1 TO 12 : READ XX,YY
6560 X(I%)=R(1,1)*XX+R(1,2)*YY+R(1,3)
6569 REM"READ facet information
6570 Y(I%)=R(2,1)*XX+R(2,2)*YY+R(2,3)
6580 NEXT I%
6589 REM draw red base of flag
6590 GCOL0,1
6599 REM READ line information
6600 MOVE FNX(X(2)),FNY(Y(2)) : MOVE FNX(X(1)),FNY(Y(1))
6610 PLOT85,FNX(X(3)),FNY(Y(3)) : PLOT85,FNX(X(4)),FNY(Y(4))
6619 REM draw two yellow diagonal stripes
6620 GCOL0,2
6629 REM READ vertex information and move it into position
6630 MOVE FNX(X(1)),FNY(Y(1)) : MOVE FNX(X(5)),FNY(Y(5))
6640 PLOT85,FNX(X(12)),FNY(Y(12)) : PLOT 85,FNX(X(8)),FNY(Y(8))
6650 PLOT85,FNX(X(9)),FNY(Y(9)) : PLOT85,FNX(X(3)),FNY(Y(3))
6660 MOVE FNX(X(2)),FNY(Y(2)) : MOVE FNX(X(6)),FNY(Y(6))
6670 PLOT85,FNX(X(7)),FNY(Y(7)) : PLOT85,FNX(X(11)),FNY(Y(11))
6680 PLOT85,FNX(X(10)),FNY(Y(10)) : PLOT85,FNX(X(4)),FNY(Y(4))
6689 REM draw two red diagonal lines
6690 GCOL0,1
6700 MOVE FNX(X(1)),FNY(Y(1)) : DRAW FNX(X(3)),FNY(Y(3))
6710 MOVE FNX(X(2)),FNY(Y(2)) : DRAW FNX(X(4)),FNY(Y(4))
6720 ENDPROC
```

***Example 4.2***

Suppose we wish to draw figure 4.2, which includes four flags labelled (a), (b), (c) and (d) on a screen that is 240 units by 180 units. For simplicity in this picture we shall assume that Q is the identity matrix, so the head is upright and the eye looks at the SETUP origin. Flag (a) is placed identically at its SETUP position (that is, $R_a = 1$) whereas flag (b) is moved from its SETUP to ACTUAL position by the following transformations:

(1) Scale the figure with SX = 4 and SY = 2, so producing matrix A.
(2) Rotate the figure through $\pi/6$ radians, so giving matrix B.
(3) Translate the figure by TX = 30 and TY = 15, so producing matrix C.

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 & 30 \\ 0 & 1 & 15 \\ 0 & 0 & 1 \end{pmatrix}$$

The complete transformation is given by $R_b = Q \times P_b = I \times P_b = P_b = C \times B \times A$ (note the order of matrix multiplication, and that the subscript distinguishes the placing of flag (b) from the others).

If instead we used the order A × B × C (giving matrix $P_d$), then

$$P_b = \begin{pmatrix} 2\sqrt{3} & -1 & 30 \\ 2 & \sqrt{3} & 15 \\ 0 & 0 & 1 \end{pmatrix} \qquad P_d = \begin{pmatrix} 2\sqrt{3} & -2 & 60\sqrt{3} - 30 \\ 1 & \sqrt{3} & 15\sqrt{3} + 30 \\ 0 & 0 & 1 \end{pmatrix}$$

which are obviously two different transformations. Matrix $R_d = Q \times P_d = I \times P_d$ produces flag (d). Note how this flag is not symmetrical about two mutually perpendicular axes as are the other three flags; be very careful with the use of the scaling transformation - remember scaling is defined about the origin and this will cause distortions in the shape of an object that is moved away from the origin!

To illustrate this example further we shall show how to calculate the ACTUAL position of the four corners of flag (b) on the screen by setting the coordinates in the form of a column vector and premultiplying it by matrix $R_b = I \times P_b$. For example

$$\begin{pmatrix} 2\sqrt{3} & -1 & 30 \\ 2 & \sqrt{3} & 15 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 5 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} 10\sqrt{3} + 25 \\ 5\sqrt{3} + 25 \\ 1 \end{pmatrix} \quad \text{etc}$$

When returned to normal vector form we see that the four vertices (5, 5), (−5, 5), (−5, −5) and (5, −5) have been transformed to $(10\sqrt{3} + 25, 5\sqrt{3} + 25)$, $(-10\sqrt{3} + 35, -5\sqrt{3} + 5)$, and $(10\sqrt{3} + 35, -5\sqrt{3} + 25)$ respectively.

Flag (c) is flag (b) reflected in the line $3y = -4x - 9$. This line cuts the $y$-axis at $(0, -3)$ and makes an angle $\alpha = \cos^{-1}(-3/5) = \sin^{-1}(4/5) = \tan^{-1}(-3/4)$ with the positive $x$-axis. If we move space by a vector $(0, 3)$, matrix D say, this line will go through the origin. Furthermore, if we rotate space by $-\alpha$, matrix E say, the line is now identical with the $x$-axis. Matrix F can reflect the flag in the $x$-xis, $E^{-1}$ puts the line back at an angle $\alpha$ with the $x$-axis, and finally $D^{-1}$ returns the line to its original position. Matrix $G = D^{-1} \times E^{-1} \times F \times E \times D$ will therefore reflect all the ACTUAL vertices of flag (b) about the line $3y = -4x - 9$ and $R_c = 1 \times P_c = G \times P_b$ can therefore be used to draw flag (c). That is we use matrix $P_b$ to move the flag to position (b) and then G to place it in position (c):

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix} \quad E = \begin{pmatrix} -3/5 & 4/5 & 0 \\ -4/5 & -3/5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$P_c = \frac{1}{25} \begin{pmatrix} -48 - 14\sqrt{3} & 7 - 24\sqrt{3} & -642 \\ 14 - 48\sqrt{3} & 24 + 7\sqrt{3} & -669 \\ 0 & 0 & 25 \end{pmatrix}$$

Figure 4.2 is drawn by using the new ‘scene2’ procedure of listing 4.9 : note that this ‘scene2’ does not call ‘look2’, since it is assumed that the eye is looking the origin with the head erect. The main program and the ‘flag’ procedure, as well as all the other graphics package procedures, stay unchanged.

*Listing 4.9*

```
6000 REM scene 2 / 4 flags not stored fixed view
6010 DEF PROCscene2
6020 DIM X(12),Y(12),A(3,3),B(3,3),R(3,3)
6029 REM flag a)
6030 PROCidR2 : PROCflag
6039 REM flag b)
6040 PROCscale2(4,2) : PROCmult2
6050 PROCrot2(PI/6) : PROCmult2
6060 PROCtran2(30,15) : PROCmult2
6070 PROCflag
6080 PROCtran2(0,3) : PROCmult2
6089 REM flag c)
6090 THETA=FNangle(-3,4)
6100 PROCrot2(-THETA) : PROCmult2
6110 PROCscale2(1,-1) : PROCmult2
6120 PROCrot2(THETA) : PROCmult2
6130 PROCtran2(0,-3) : PROCmult2
6140 PROCflag
6149 REM flag d)
6150 PROCidR2
6160 PROCtran2(30,15) : PROCmult2
6170 PROCrot2(PI/6) :PROCmult2
6180 PROCscale2(4,2) : PROCmult2
6190 PROCflag
6200 ENDPROC
```

***Exercise 4.1***

In order to convince yourself that this program may be used to deal with the general situation, you should run this program using non-zero values of DX, DY or a so that the ACTUAL to OBSERVED matrix Q is not the identity matrix. Your ‘scene2’ procedure should call ‘look2’ to calculate Q, which must be stored. Then for each object in the scene, in turn, calculate the SETUP to ACTUAL matrix P (which ‘mult2’ places in R), premutliply it by Q (which has to be copied into matrix A for use with ‘mult2’) and finally enter the construction procedure with the product matrix R = Q × P.

***Exercise 4.2***

Use the above procedures to draw diagrams that are similar to figure 4.2, but where the number, position and direction of the flags are read in from the keyboard. You can produce procedures to draw more complicated objects, we have chosen a very simple example so that the algorithms would not be obscured by the complexity of objects. The above method can deal with as many vertices, lines and coloured areas as the Model B can handle within time and storage limitations.

***Exercise 4.3***

By using loops in the program we can draw ordered sequences of the objects; for example, they may all have the same orientation but their points of reference (the origin in the SETUP position) may be equally spaced along any line $\boldsymbol{p} + \mu\boldsymbol{q}$. We can set up a loop with index parameter $\mu$ and draw one flag for each pass through the loop. For each value of $\mu$ we can alter the parameters of translation in a regular way within the loop (using $\mu$, $\boldsymbol{p}$ and $\boldsymbol{q}$). The new values of these parameters are used to calculate a different SETUP to ACTUAL matrix for each occurrence, and this moves the object into a new ACTUAL position. R = Q × P = I × P is used to observe and draw each object on the screen. With these ideas, construct a line of flags on the screen.

## Efficient Use of Matrices

It is obvious that whatever combination of transfomations we use, the third row of every matrix will always be (0 0 1), If we work with only the top two rows of the matrix this will make our procedures much more efficient. We still keep 3 × 3 rather than 2 × 3 matrices (which is really all we need), because we may have previously written other procedures that assume 3 × 3 matrices. ReDIMensioning the arrays could lead to array bound errors in the earlier procedures - the cost of a few extra real numbers per matrix is a small price to pay to avoid errors. It is also more efficient to use explicit statements rather than loops. Listings 4.1, 4.2, 4.3 and 4.4 are rewritten as listings 4.1a, 4.2a, 4.3a and 4.4a, respectively, to make use of these facts.

*Listing 4.1a*

```
9100 REM mult2
9110 DEF PROCmult2
9120 LOCAL I%,J%,K%
9130 FOR I%=1 TO 2
9140 FOR J%=1 TO 3
9150 B(I%,J%)=A(I%,1)*R(1,J%)+A(I%,2)*R(2,J%)
9160 NEXT J%
9170 B(I%,3)=B(I%,3)+A(I%,3)
9210 NEXT I%
9220 FOR I%=1 TO 2
9230 FOR J%=1 TO 3
9240 R(I%,J%)=B(I%,J%)
9250 NEXT J%
9260 NEXT I%
9270 ENDPROC

9300 REM idR2
9310 DEF PROCidR2
9320 R(1,1)=1 : R(1,2)=0 : R(1,3)=0
9330 R(2,1)=0 : R(2,2)=1 : R(2,3)=0
9340 ENDPROC
```

*Listing 4.2a*

```
9000 REM tran2
9010 DEF PROCtran2(TX,TY)
9020 A(1,3)=TX : A(2,3)=TY
9030 A(1,1)= 1 : A(1,2)= 0
9040 A(2,1)= 0 : A(2,2)= 1
9050 ENDPROC
```

*Listing 4.3a*

```
8900 REM scale2
8910 DEF PROCscale2(SX,SY)
8920 A(1,1)=SX : A(2,2)=SY
8930 A(1,2)= 0 : A(1,3)= 0
8940 A(2,1)= 0 : A(2,3)= 0
8950 ENDPROC
```

*Listing 4.4a*

```
8600 REM rot2
8610 DEF PROCrot2(THETA)
8620 CT=COS(THETA) : ST=SIN(THETA)
8630 A(1,1)= CT : A(2,2)=CT
8640 A(1,2)=-ST : A(2,1)=ST
8650 A(1,3)=  0 : A(2,3)= 0
8660 ENDPROC
```

The construction offigure 4.2 may seem rather contrived since the position of the objects was chosen in an arbitrary way. However, in most diagrams the positioning of objects will be well defined, the values being implicit in the diagram required. Example 4.3 illustrates this.

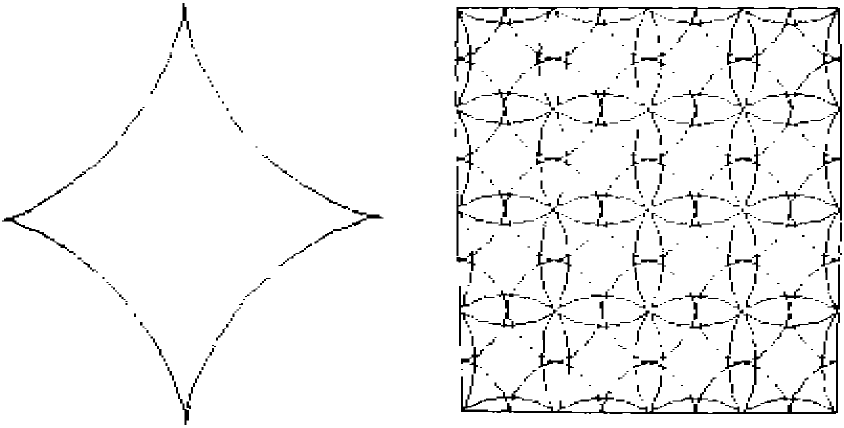
Example 4.3
Write a program to draw an ellipse that has major axis A and minor axis B, an that is centred at the point (CX, CY). The major axis makes an angle θ (THETA) with the positive *x*-direction. Note that the order of transformations is important: first rotate and then translate. If we wish to draw ellipses with major axis horizontal then we need not use matrices, we can stay with the procedure set in exercise 2.5 and use ideas that are similar to those in listing 2.9. Dsting 4.10 gives a ‘scene2’ procedure that reads in data about the ellipse calculates the SETUP to OBSERVED matrix and then calls the construction procedure ‘ellipse’ to draw the ellipse.

*Listing 4.10*

```
6000 REM scene2 / ellipse not stored : fixed view
6010 DEF PROCscene2
6020 DIM A(3,3),B(3,3),R(3,3)
6029 REM major axis A, minor axis B centre (CX,CY),
         angle THETA
6030 INPUT"A,B,CX,CY,THETA",A,B,CX,CY,THETA
6040 PROCidR2 : PROCrot2(THETA) : PROCmult2
6050 PROCtran2(CX,CY) : PROCmult2
6060 PROCellipse(A,B)
6070 ENDPROC
6500 REM"ellipse / points not stored
6510 DEF PROCellipse(A,B)
6520 LOCAL I%,ALPHA,ADIF,XX,YY,XPT,YPT
6529 REM find points (XX,YY) on the ellipse with
         major axis A, minor axis B and placed in
         position using matrix R
6530 XPT=R(1,1)*A+R(1,3) : YPT=R(2,1)*A+R(2,3)
6540 PROCmoveto(XPT,YPT)
6550 ALPHA=0 : ADIF=PI/50
6560 FOR I%=1 TO 100
6570 ALPHA=ALPHA+ADIF
6580 XX=A*COS(ALPHA) : YY=B*SIN(ALPHA)
6590 XPT=R(1,1)*XX+R(1,2)*YY+R(1,3)
6600 YPT=R(2,1)*XX+R(2,2)*YY+R(2,3)
6610 PROClineto(XPT,YPT)
6620 NEXT I%
6630 ENDPROC
```

***Exercise 4.4***
Write a procedure fur drawing an individual matrix-transformable object (in this case the astroid shown in figure 4.3a) and then use the matrix techniques to draw combinations of these objects (as in figure 4.3b). An astroid is a closed curve

with the parametric form (R × $\cos^3\theta$, R × $\sin^3\theta$) where $0 \le \theta \le 2\pi$ and R is the radius (the maximum distance from the centre of the object). The parameters needed by this procedure are the radius of the astroid and the transforming matrix. Figure 4.3b is the combination of a large number of two diffirent forms of the astroid. One has radius 1 and is not rotated, the other has radius $\sqrt{2}$ and is rotated through $\pi/4$ radians



(a) (b)

*Figure 4.3*

***Exercise 4.5***

Experiment with these matrix techniques. Write a procedure to generate the matrix that is needed to rotate points in space by an angle θ about an arbitrary point (X, Y) in space (not necessarily the origin). Also produce another procedure to generate the matrix that will reflect points about the general line $ay = bx + c$ (use the ideas given in example 4.2 for the production of flag (c)).

## Storing information about Scenes

It mentioned earlier that certain situations arise when we need to store all the information about a scene in a large data-base rather than lose the information on leaving the construction procedure. Our data-base will consist of vertices, lines and facets, together with information on colour which can be explicitly or implicitly stored. Vertices are stored as arrays X and Y, of size greater than or equal to NOV, the final number of vertices to be stored (these vertices can be stored in the SETUP, ACTUAL or OBSERVED position: it depends on the context of the problem).

Line information is stored in a two-dimensional array LIN whose first index is 1 or 2, and whose second index is a number between 1 and a value greater than or equal to NOL, the final number of lines in the scene. The Ith line joins the two vertices with indices LIN(1, I) to LIN(2, I); hence this information is independent of position, it simply says which two vertices are joined by the Ith line. We shall assume that the colours of lines will be implicitly defined in the program listings.

Information about polygonal areas or facets (≤ NOF in number) may be stored in a two-dimensional array FACET and two one-dimensional arrays SIZE and COL. SIZE(J) holds the number of edges in facet J, COL(J) explicitly defines its colour, and FACET(J, K), where I ≤ J ≤ NOF and I ≤ K ≤ SIZE(J), holds the indices of the vertices that make up the facet. NOV, NOL and NOF values are initialised in the ‘scene2’ procedure and incremented in the construction procedures. Note that if we wish to explicitly colour the lines then another array must be added.

We now no longer require construction procedures to draw lines and facets, we use them only to create the data-base of lines, vertices, facets etc. (transformed by the matrix R). After ‘scene2’ has constructed the final scene in memory it calls another procedure ‘drawit’ to draw the final picture. The ‘scene2’ procedure will be very similar to those mentioned earlier; for example the procedure for drawing figure 4.2 in this new way will be that given in listing 4.9 with three minor changes listed below:

```
6020 DIM X(48), Y(48), LIN(2, 8) FACET(6, 12), S1ZE(12), COL(12),
        A(3, 3), B(3, 3), R(3, 3)

6030 NOV = 0 : NOL = 0 : NOF = 0 : PROCidR2 : PROCflag

6200 PROCdrawit : ENDPROC
```

This is used in conjunction with listing 4.11 which gives the ‘flag’ construction procedure (which now merely sets up the data) and the ‘drawit’ procedure.

Suppose we wish to produce different views of the same scene (again we shall use figure 4.2 as an example), that is, with the same SETUP to ACTUAL matrices P, but different ACTUAL to OBSERVED matrices Q. The obvious solution is to create a data-base for the scene with the vertices in the ACTUAL position (we can use the ‘flag’ procedure of listing 4.11). Now for each new OBSERVED position we calculate Q and enter it into another ‘drawit’ procedure (see listing 4.12 - which is different from listing 4.11) which transfers each vertex from its ACTUAL to its OBSERVED position using Q, stores them in arrays XD and YD so as not to corrupt the X, Y data-base, and recalls them when

*Listing 4.11*

```
6500 REM flag / placed in position by matrix R and stored
6510 DEF PROCflag
6520 LOCAL I%,J%,XX,YY,L1,L2,FVAL
6530 RESTORE 6540
6540 DATA 4,1,1,2,3,4,  6,2,1,5,8,3,9,12,  6,2,2,7,10,4,11,6
6550 DATA 1,3, 2,4
6560 DATA 5,5, -5,5, -5,-5, 5,-5, 4,5, -4,5, -5,4, -5,-4, -4,-5,
          4,-5, 5,-4, 5,4
6569 REM READ facet information
6570 FOR I%=1 TO 3 : NOF=NOF+1 : READ SIZE(NOF),COL(NOF)
6580 FOR J%=1 TO SIZE(NOF) : READ FVAL : FACET(J%,NOF)=FVAL+NOV
6590 NEXT J% : NEXT I%
6599 REM READ line information
6600 FOR I%=1 TO 2 : NOL=NOL+1 : READ L1,L2
6610 LIN(1,NOL)=L1+NOV : LIN(2,NOL)=L2+NOV
6620 NEXT I%
6629 REM READ vertex information and move it into position
6630 FOR I%=1 TO 12 : READ XX,YY : NOV=NOV+1
6640 X(NOV)=R(1,1)*XX+R(1,2)*YY+R(1,3)
6650 Y(NOV)=R(2,1)*XX+R(2,2)*YY+R(2,3)
6660 NEXT I%
6670 ENDPROC

7000 REM drawit
7010 DEF PROCdrawit
7020 LOCAL I%,J%,K% : CLG
7029 REM"draw the NOF facets : explicit colours in array COL
7030 FOR I%=1 TO NOF
7040 GCOL 0,COL(I%)
7050 K%=FACET(2,I%) : MOVE FNX(X(K%)),FNY(Y(K%))
7060 FOR J%=3 TO SIZE(I%)
7069 REM"draw facets
7070 K%=FACET(1,I%) : MOVE FNX(X(K%)),FNY(Y(K%))
7080 K%=FACET(J%,I%) : PLOT 85,FNX(X(K%)),FNY(Y(K%))
7090 NEXT J% : NEXT I%
7099 REM"draw the NOL lines implicit colour 1 (red)
7100 GCOL0,1
7110 FOR I%=1 TO NOL
7120 K%=LIN(1,I%) : PROCmoveto(X(K%),Y(K%))
7130 K%=LIN(2,I%) : PROClineto(X(K%),Y(K%))
7140 NEXT I%
7150 ENDPROC
```

they are required for drawing. When using this method to construct different views of figure 4.2 only the ‘scene2’ and ‘drawit’ procedures differ from their earlier manifestations, and then only slightly. We give them in listing 4.12.

***Exercise 4.6***
Construct a ‘drawit’ procedure for a flag which uses the ‘triangle’ procedure.

***Exercise 4.7***
Construct a dynamic scene. With each new view the flags will move relative to one another in some well-defined manner. The observer should also move in

*Listing 4.12*

```
6000 REM scene2 / 4 flags stored variable view
6010 DEF PROCscene2
6020 DIM X(48),Y(48),XD(48),YD(48),LIN(2,8),FACET(6,12),
        SIZE(12),COL(12),A(3,3),B(3,3),R(3,3)
6028 REM create a data base of flags in ACTUAL position
6029 REM flag a)
6030 NOV=0 : NOL=0 : NOF=0 : PROCidR2 : PROCflag
6039 REM"flag b)
6040 PROCscale2(4,2) : PROCmult2
6050 PROCrot2(PI/6) : PROCmult2
6060 PROCtran2(30,15) : PROCmult2
6070 PROCflag
6079 REM flag c)
6080 PROCtran2(0,3) : PROCmult2
6090 THETA=FNangle(-3,4)
6100 PROCrot2(-THETA) : PROCmult2
6110 PROCscale2(1,-1) : PROCmult2
6120 PROCrot2(THETA) : PROCmult2
6130 PROCtran2(0,-3) : PROCmult2
6140 PROCflag
6149 REM flag d)
6150 PROCidR2
6160 PROCtran2(30,15) : PROCmult2
6170 PROCrot2(PI/6) : PROCmult2
6180 PROCscale2(4,2) : PROCmult2
6190 PROCflag
6199 REM loop through different views
6200 PROCidR2 : PROClook2
6210 PROCdrawit
6220 GOTO 6200
6230 ENDPROC

7000 REM drawit
7010 DEF PROCdrawit
7020 LOCAL I%,J%,K% : CLG
7029 REM move vertices to OBSERVED position using matrix R
7030 FORI%=1 TO NOV
7040 XD(I%)=R(1,1)*X(I%)+R(1,2)*Y(I%)+R(1,3)
7050 YD(I%)=R(2,1)*X(I%)+R(2,2)*Y(I%)+R(2,3)
7060 NEXT I%
7069 REM draw facets
7070 FOR I%=1 TO NOF
7080 GCOL 0,COL(I%)
7090 K%=FACET(2,I%) : MOVE FNX(XD(K%)),FNY(YD(K%))
7100 FOR J%=3 TO SIZE(I%)
7110 K%=FACET(1,I%) : MOVE FNX(XD(K%)),FNY(YD(K%))
7120 K%=FACET(J%,I%) : PLOT 85,FNX(XD(K%)),FNY(YD(K%))
7130 NEXT J% : NEXT I%
7140 GCOL0,1
7149 REM draw lines
7150 FOR I%=1 TO NOL
7160 K%=LIN(1,I%) : PROCmoveto(XD(K%),YD(K%))
7170 K%=LIN(2,I%) : PROClineto(XD(K%),YD(K%))
7180 NEXT I%
7190 ENDPROC
```

some simple way, for example the eye could start looking at the origin, twenty views later it could be looking at the point (100, 100), and with each view the head could tilt a further 0.1 radian. You no longer need to INPUT the values cf (DX, DY) and ALPHA into 'look2', instead they should be calculated by the program.

***Exercise 4.8***

Construct a scene that is a diagrammatic view of a room in your house - with schematic two-dimensional drawings of tables, chairs etc. placed in the room. Each different type of object has its own construction procedure, and the 'scene2' procedure should read in data to place these objects around the room. Once the scene is set produce a variety of views, looking from various points and orientations. Use the menu technique of chapters 5 and 6 to input information .

Or you can set up a line-drawing picture of a map, and again view it from various orientations. The number of possible choices of scene is enormous!

We can choose small values for HORIZ and VERT, which has the effect of the observer zooming up close to parts of a scene, and all external lines will be conveniently clipped off.

---

## Complete Programs

We group the listings 3.3 ('angle'), 4.1a ('mult2' arid 'idR2'), 4.2a ('tran2'), 4.3a ('scale2'), 4.4a ('rot2'), 4.5 ('look2') and 4.6 ('main program') under the heading 'lib2'.

I 'lib1', 'lib2', listings 4.7 ('scene2') and 4.8 ('flag'). Data required: mode, HORIZ, VERT, DX, DY and ALPHA. Try 1, 24, 18, 1, 1, 0.5. Keep any five of these values fixed and systematically make small changes in the other data value.

II 'lib1', 'lib2', listings 4.9 ('scene2') and 4.8 ('flag'). Data required: mode, HORIZ, VERT. Try 1 , 240, 180; 1, 160, 120; 1, 80, 60.

III 'lib1', 'lib2' and listings 4.10 ('scene2' and 'ellipse'). Data required: mode, HORIZ, VERT, A, B, CX, CY, THETA. Try 1, 30, 20, 12, 9, 1, 1, 0.5. Again fix all but one of the values and change the remaining value systematically.

IV 'lib1', 'lib2', listings 4.9 ('scene2' adjusted as described in the text) and 4.11 ('flag' and 'drawit'). Data required: as II above. LOAD with PAGE = &1100.

V 'lib1', 'lib2', listings 4.11 ('flag' but not 'drawit') and 4.12 ('scene2' and 'drawit'). Data required: mode, HORIZ, VERT, DX, DY, ALPHA. Try 1, 240, 180, 5, 5, 1. Systematically change each of the data values in turn. LOAD with PAGE = &1100.