CHAPTER FOUR

TRANSFORMATIONS

INTRODUCTION

One of the major assets of computer graphics is the ease with which certain alterations of the picture may be performed. The manager can alter the scale of the graphs in a report. The architect can view a building from a different angle. The cartographer can change the size of a chart. The animator can change the position of a character. These changes are easy to perform because the graphic image has been coded as numbers and stored within the computer. The numbers may be modified by mathematical operations called *transformations*.

Transformations allow us to uniformly alter the entire picture. It is in fact often easier to change the entire computer-drawn image than it is to alter only part of it. This can provide a useful complement to hand drawing techniques, where it is usually easier to change a small portion of a drawing than it is to create an entirely new picture.

In this chapter we shall consider the geometric transformations of scaling, translation, and rotation. We shall see how they can be simply expressed in terms of matrix multiplications. We shall introduce homogeneous coordinates in order to uniformly treat translations and in anticipation of three-dimensional perspective transformations. The algorithms presented in this chapter will describe two-dimensional scale, translation, and rotation routines.

MATRICES

Our computer graphics images are generated from a series of line segments which are represented by the coordinates of their endpoints. Certain changes in an image can be easily made by performing mathematical operations on these coordinates. Before we consider some of the possible transformations, let us review some of the mathematical tools we shall need, namely, matrix multiplication.

For our purposes we will consider a matrix to be a two-dimensional array of numbers. For example,

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix} \quad \begin{vmatrix} 1 \\ -1 \\ 0 \end{vmatrix} \quad \begin{vmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{vmatrix}$$

are four different matrices.

Suppose we define the matrix A to be

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} \tag{4.1}$$

Then the element in the second row and third column would be A(2, 3) and would have the value 6.

The matrix operation which concerns us most is that of multiplication. Matrix multiplication is more complex than the simple product of two numbers; it involves simple products and sums of the matrix elements. Not every pair of matrices can be multiplied. We can multiply two matrices A and B together if the number of columns of the first matrix (A) is the same as the number of rows of the second matrix (B). For example, if we chose the first matrix to be the matrix A defined in Equation 4.1 and matrix B to be

$$B = \begin{vmatrix} 1 & 0 \\ -1 & 2 \\ 0 & 1 \end{vmatrix} \tag{4.2}$$

then we can multiply A times B because A has 3 columns and B has 3 rows. Unlike multiplication of numbers, the multiplication of matrices is not commutative; that is, while we can multiply A times B, we cannot multiply B times A, because B has only 2 columns (which does not match the 3 rows of A). When we multiply two matrices, we get a matrix as a result. This product matrix will have the same number of rows as the first matrix of the two being multiplied and the same number of columns as the second matrix. Multiplying the 3 × 3 matrix A times the 3 × 2 matrix B gives a 3 × 2 matrix result C.

The elements of the product matrix C are given in terms of the elements of matrices A and B by the following formula:

$$C(i, k) = \sum_j A(i, j)\, B(j, k) \tag{4.3}$$

For our particular example of C = AB

$$C = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} \begin{vmatrix} 1 & 0 \\ -1 & 2 \\ 0 & 1 \end{vmatrix} \tag{4.4}$$

the element C(1, 1) is found by multiplying each element of the first row of A by the corresponding element of the first column of B and adding these products together.

$$\begin{aligned} C(1,1) &= A(1,1)B(1,1) + A(1,2)B(2,1) + A(1,3)B(3,1) \\ &= (1)(1) + (2)(-1) + (3)(0) = -1 \end{aligned} \tag{4.5}$$

The element C(3, 2) would be

$$\begin{aligned} C(3,2) &= A(3,1)B(1,2) + A(3,2)B(2,2) + A(3,3)B(3,2) \\ &= (7)(0) + (8)(2) + (9)(1) = 25 \end{aligned} \tag{4.6}$$

Performing this arithmetic for every element of C shows us that

$$C = \begin{vmatrix} -1 & 7 \\ -1 & 16 \\ -1 & 25 \end{vmatrix} \tag{4.7}$$

Multiplication is associative. This means that if we have several matrices to multiply together, it does not matter which we multiply first. In mathematical notation:

$$A(BC) = (AB)C \tag{4.8}$$

This is a very useful property; it will allow us to combine several graphics transformations into a single transformation, thereby making our calculations more efficient.

There is a set of matrices with the property that when they multiply another matrix, they reproduce that matrix. For this reason, the matrices in this set are called *identity matrices*. They are square matrices (same number of rows and columns) with all the elements 0 except the elements of the main diagonal, which are all 1. For example,

$$\begin{vmatrix} 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

and so on.

We can see that if

$$I = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.9}$$

then

$$A = AI \tag{4.10}$$

SCALING TRANSFORMATIONS

Now how does all this apply to graphics? Well, we can consider a point $P_1 = [x_1 \quad y_1]$ as being a 1 × 2 matrix. If we multiply it by some 2 × 2 matrix T, we will obtain another 1 × 2 matrix which we can interpret as another point

$$[x_2 \quad y_2] = P_2 = P_1T \tag{4.11}$$

Thus, the matrix T gives a mapping between an original point P_1 and a new point P_2. Remember that our image is stored as a list of endpoints. What will happen if we transform every point by means of multiplication by T and display the result? What will this new image look like? The answer, of course, depends upon what elements are in matrix T. If, for example, matrix T were the identity matrix

$$T = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \tag{4.12}$$

then the image would be unchanged.

If, however, we choose T to be T_1

$$T_1 = \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix} \tag{4.13}$$

then,

$$[x_2 \quad y_2] = [x_1 \quad y_1] \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix} = [2x_1 \quad y_1] \tag{4.14}$$

Every new x coordinate would be twice as large as the old value. Horizontal lines would become twice as long on the new image. The new image would have the same height, but would appear to be stretched to twice the width of the original. (See Figure 4-1.)

The transformation matrix

$$T_2 = \begin{vmatrix} 0.5 & 0 \\ 0 & 1 \end{vmatrix} \tag{4.15}$$

would shrink all x coordinates to one-half their original value. The image would have the original height but would be squeezed to one-half the width.

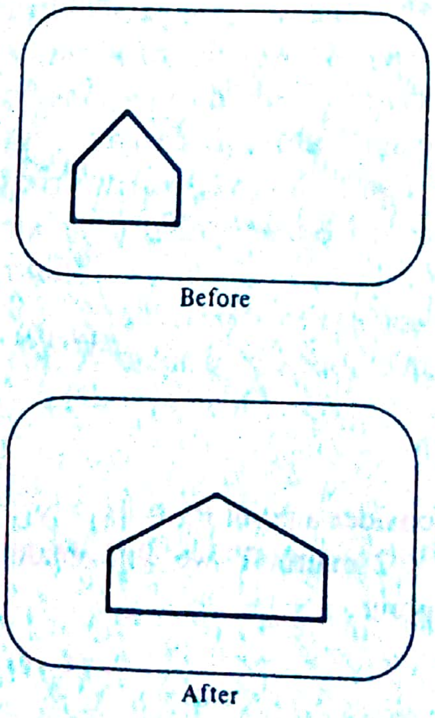

FIGURE 4-1
Scaling x coordinates by 2.

Now, if we were to stretch the image to twice the width and then compress it to one-half the new width,

$$P_2 = (P_1T_1)T_2 \tag{4.16}$$

we would expect to get the original image back again. Let us check that this is so by multiplying the two transformations T_1 and T_2 together first. The associative property for matrix multiplication allows us to write

$$P_2 = P_1(T_1T_2) \tag{4.17}$$

Multiplying the two transformations together combines them into a single transformation.

$$T_1T_2 = \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix} \begin{vmatrix} 0.5 & 0 \\ 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \tag{4.18}$$

But this is just the expected identity matrix, which will not change an image.

We can make an image twice as tall with the same width by finding a transformation which just multiplies the y coordinate by 2. Such a transformation is given by

$$T_3 = \begin{vmatrix} 1 & 0 \\ 0 & 2 \end{vmatrix} \tag{4.19}$$

Multiplying an arbitrary point by this matrix shows that this is true. (See Figure 4-2.)

$$[x_2 \quad y_2] = [x_1 \quad y_1] \begin{vmatrix} 1 & 0 \\ 0 & 2 \end{vmatrix} = [x_1 \quad 2y_1] \tag{4.20}$$

By applying both transformations T_1 and T_3, we will make the image both twice as wide and twice as tall. In other words, we would have a similar image, only twice as big.

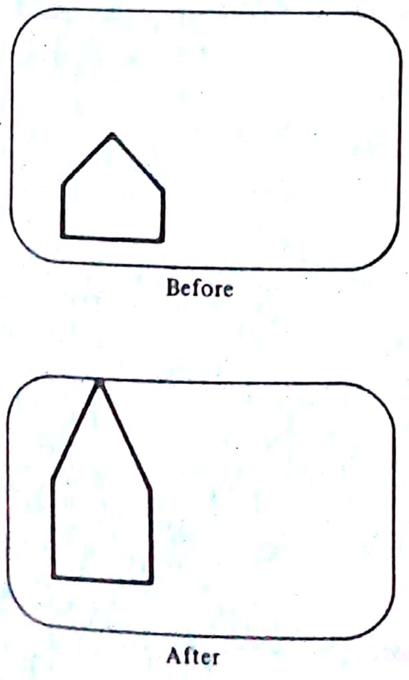

FIGURE 4-2
Scaling y coordinates by 2.

$$P_2 = P_1 T_1 T_3 \tag{4.21}$$

Again, by multiplying the two transformation matrices together, we can obtain a single transformation matrix for making the entire image twice as large. (See Figure 4-3.)

$$T_4 = T_1 T_3 = \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 \\ 0 & 2 \end{vmatrix} = \begin{vmatrix} 2 & 0 \\ 0 & 2 \end{vmatrix} \tag{4.22}$$

In general, transformations of the form

$$S = \begin{vmatrix} s_x & 0 \\ 0 & s_y \end{vmatrix} \tag{4.23}$$

change the size and the proportion of the image. They are called *scaling transformations*
s_x is the *scale factor* for the x coordinate and s_y for the y coordinate.
Note that when we scale the image, every point except the origin changes. This
means that not only the size of an image will change but also its position. A scale in x
by a factor greater than 1 will cause the image to shift to the right, along with making
it wider. A scale in x by a factor less than 1 will shift the image to the left. A scale in y
will shift the image up and down as well as change its height.

SIN AND COS

The next transformation we would like to consider is that for rotation. To prepare for the discussion of rotations we shall review some basic trigonometry. Suppose we have a point $P_1 = (x_1, y_1)$ and we rotate it about the origin by an angle θ to get a new position $P_2 = (x_2, y_2)$. We wish to find a transformation which will change (x_1, y_1) into (x_2, y_2). But, before we can check any transformation to see if it is correct, we must first know what (x_2, y_2) should be in terms of (x_1, y_1) and θ. To determine this we shall need the trigonometric functions *sine* and *cosine* (abbreviated *sin* and *cos*). We can de-

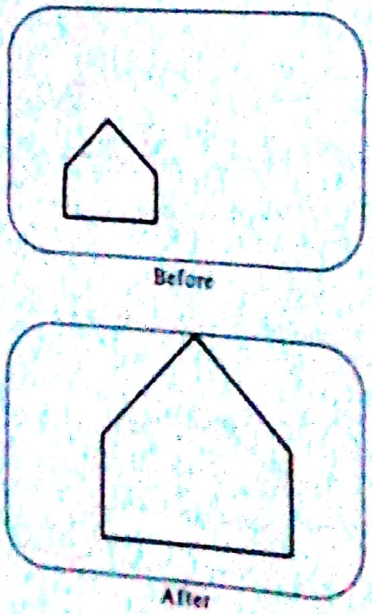

FIGURE 4-3
Scaling both x and y coordinates.

fine sin and cos for an angle θ in the following manner. Let us draw a line segment from the origin at the angle θ counterclockwise from the x axis, and suppose that the line segment we have drawn has length L. (See Figure 4-4.)

The line segment will then have endpoints (0, 0) and (x, y) and length

$$L = (x^2 + y^2)^{1/2}$$

Then, the ratio of the height of the (x, y) endpoint above the x axis (the y-coordinate value) and the length of the segment will be the sine of the angle

$$\sin\theta = \frac{y}{(x^2 + y^2)^{1/2}} \tag{4.24}$$

and the ratio of the distance to the right of the y axis (the x-coordinate value) and the length of the segment will be the cosine of the angle

$$\cos\theta = \frac{x}{(x^2 + y^2)^{1/2}} \tag{4.25}$$

Note that if we draw a segment with length L = 1, then

$$\sin\theta = y \quad \text{and} \quad \cos\theta = x \tag{4.26}$$

ROTATION

To determine the form for the *rotation transformation matrix*, consider the point (1, 0). If we rotate this point counterclockwise by an angle θ, it becomes (cos θ, sin θ) (see Figure 4-5), so

$$[\cos\theta \quad \sin\theta] = \begin{vmatrix} 1 & 0 \end{vmatrix} \begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} a, & b \end{vmatrix} \tag{4.27}$$

If we rotate the point (0, 1) counterclockwise by an angle θ, it becomes (−sin θ, cos θ). (See Figure 4-6.)

$$[-\sin\theta \quad \cos\theta] = \begin{vmatrix} 0 & 1 \end{vmatrix} \begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} c, & d \end{vmatrix} \tag{4.28}$$

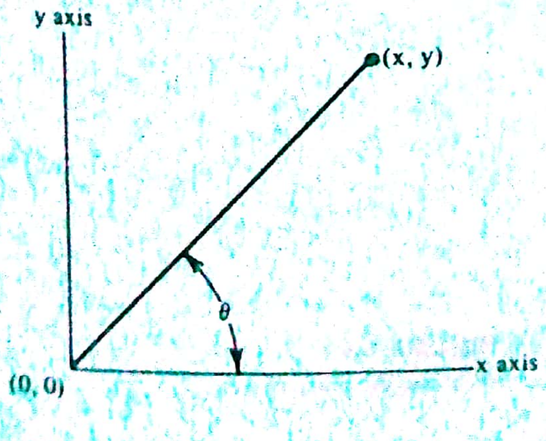

FIGURE 4-4
Definition of angle.

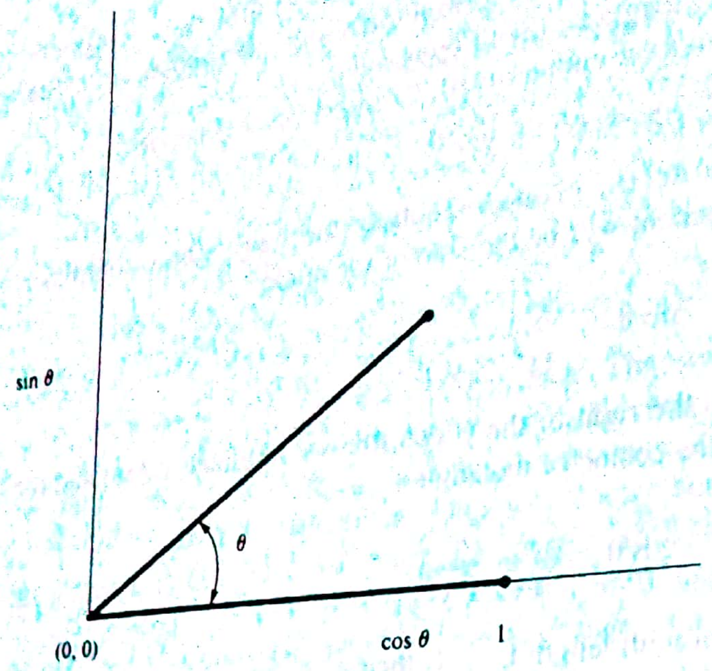

FIGURE 4-5
Rotating the point (1, 0).

From these equations we can see the values of a, b, c, and d needed to form the rotation matrix. The transformation matrix for a counterclockwise rotation of θ about the origin is

$$R = \begin{vmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{vmatrix} \tag{4.29}$$

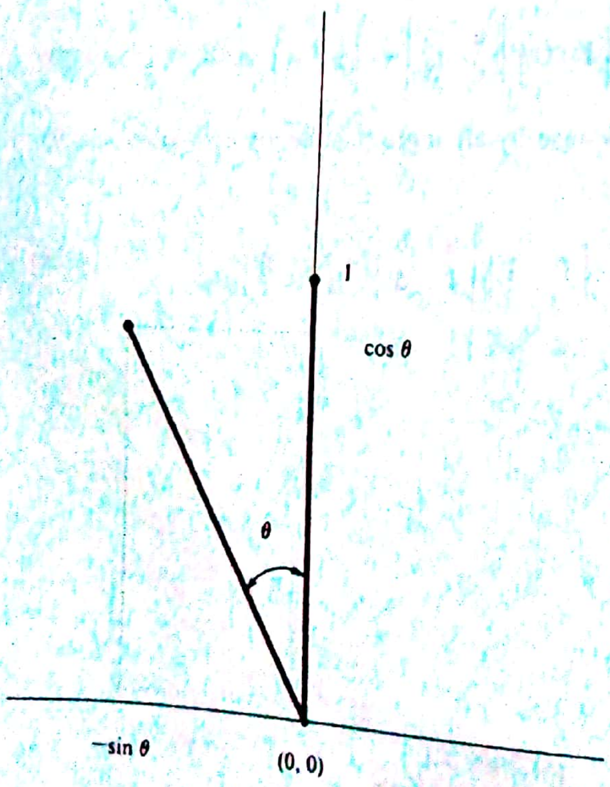

FIGURE 4-6
Rotating the point (0, 1).

As an example, suppose we wished to rotate the point (2, 3) counterclockwise by an angle of $\pi/6$ radians. (See Figure 4-7.) Then the rotation matrix would be

$$\begin{vmatrix} \cos\frac{\pi}{6} & \sin\frac{\pi}{6} \\ -\sin\frac{\pi}{6} & \cos\frac{\pi}{6} \end{vmatrix} = \begin{vmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{vmatrix} \tag{4.30}$$

and the rotated point would be

$$\begin{vmatrix} 2 & 3 \end{vmatrix} \begin{vmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{vmatrix} = \begin{vmatrix} 0.232 & 3.598 \end{vmatrix} \tag{4.31}$$

We can rotate an entire line segment by rotating both the endpoints which specify it.

The sign of an angle determines the direction of rotation. We have defined the rotation matrix so that a positive angle will rotate the image in a counterclockwise direction with respect to the axes. In order to rotate in the clockwise direction we use a negative angle, so the rotation matrix for an angle θ clockwise would be

$$R = \begin{vmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{vmatrix} \tag{4.32}$$

or since

$$\cos(-\theta) = \cos\theta \tag{4.33}$$

and

$$\sin(-\theta) = -\sin\theta \tag{4.34}$$

this may be rewritten as

$$R = \begin{vmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{vmatrix} \tag{4.35}$$

for clockwise rotations about the origin.

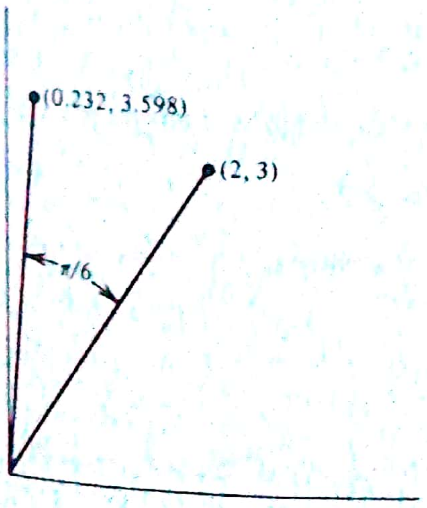

FIGURE 4-7
Rotation of (2,3) by $\pi/6$.

HOMOGENEOUS COORDINATES AND TRANSLATION

Suppose we wished to rotate about some point other than the origin. If we had some way of moving the entire image around on the screen, we could accomplish such a rotation by first moving the image until the center of rotation was at the origin, then performing the rotation as we just discussed, and finally, moving the image back where it belongs.

Moving the image is called *translation*. It is easily accomplished by adding to each point the amount by which we want the picture shifted. If we wish the image shifted 2 units to the right, we would add 2 to the x coordinate of every point. To move it down 1 unit, add −1 to every y coordinate. (See Figure 4-8.)

In general, in order to translate the image to the right and up by (t_x, t_y), every point (x_1, y_1) is replaced by a new point (x_2, y_2) where

$$x_2 = x_1 + t_x, \qquad y_2 = y_1 + t_y \tag{4.36}$$

Unfortunately, this way of describing translation does not use a matrix, so it cannot be combined with other transformations by simple matrix multiplication. Such a combination would be desirable; for example, we have seen that rotating about a point other than the origin can be done by a translation, a rotation, and another translation. We would like to be able to combine these three transformations into a single transformation for the sake of efficiency and elegance. One way of doing this is to use homogeneous coordinates. In *homogeneous coordinates* we use 3 × 3 matrices instead of 2 × 2, introducing an additional dummy coordinate w; points are specified by three numbers instead of two. The first homogeneous coordinate will be the product of x and w, the second will be the product of y and w, and the third will just be w. A coordinate point (x, y) will be represented by the triple (xw, yw, w). The x and y coordinates can easily be recovered by dividing the first and second numbers by the third. We will not really use the extra number w until we consider three-dimensional perspective transforma-

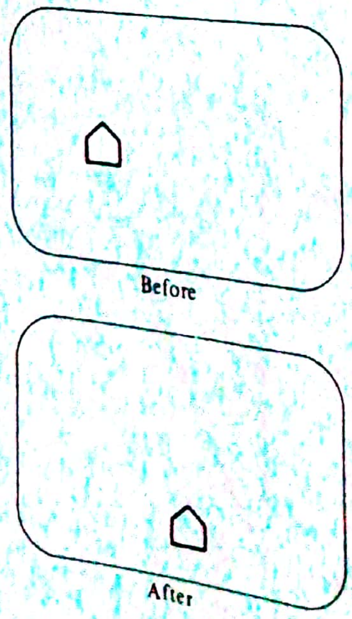

FIGURE 4-8
Translation.

tions. In two dimensions its value is usually kept at 1 for simplicity. Still, we will discuss it in its generality in anticipation of the three-dimensional transformations.

In homogeneous coordinates our scaling matrix

$$\begin{vmatrix} s_x & 0 \\ 0 & s_y \end{vmatrix}$$

becomes

$$S = \begin{vmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.37}$$

If we apply this to the point (xw, yw, w), we obtain

$$\begin{vmatrix} xw & yw & w \end{vmatrix} \begin{vmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} s_x xw & s_y yw & w \end{vmatrix} \tag{4.38}$$

Dividing by the third number w gives

$$(s_x x, s_y y)$$

which is the correctly scaled point.

The counterclockwise rotation matrix

$$\begin{vmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{vmatrix}$$

becomes, using homogeneous coordinates,

$$R = \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.39}$$

Applying it to the point (x, y) with homogeneous coordinate (xw, yw, w) gives

$$[xw \quad yw \quad w] \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} =$$

$$[(xw\cos\theta - yw\sin\theta) \quad (xw\sin\theta + yw\cos\theta) \quad w] \tag{4.40}$$

for the correctly rotated point

$$(x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta)$$

The homogeneous coordinate transformation matrix for a translation of t_x, t_y is

$$T = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{vmatrix} \tag{4.41}$$

To show that this is so, we apply the matrix

$$[xw \quad yw \quad w] \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{vmatrix} = [(xw + t_x w) \quad (yw + t_y w) \quad w] \qquad (4.42)$$

for the translated point $(x + t_x, y + t_y)$

COORDINATE TRANSFORMATIONS

We have shown how applying a transformation to a point yields a new point, but transformations may also be used to change coordinate systems. For example, a distance measured in inches can be converted to the same distance measured in centimeters by means of a scale. The actual operation of the transformation is the same as already described; only the interpretation changes. The coordinates of the point, when multiplied by the transformation, represent the same point, only measured in different coordinates.

Translations are useful coordinate transformations when the origins are not aligned. For example, we think of the lower-left corner of the display as being the origin (0, 0), but in some display systems this point may actually correspond to pixel (1, 1). Another example is a display which places the (0, 0) pixel in the upper-left corner and numbers the scan lines from top to bottom. This is sometimes done on alphanumeric printers because it is the order in which the lines are printed, and on raster displays because it is the order that they are actually scanned. To convert between these coordinates and the ones we have been using, we need a scale of 1 in x but −1 in y to reverse the scan-line order, and also a translation in y by the vertical screen dimension to move the origin to the proper corner.

Rotations may also be used in coordinate transformations, but are usually for angles of $\pi/2$ (90 degrees). For example, a printer using 8.5 by 11 inch paper may have the y axis along the long edge and the x axis along the short edge. This is called *portrait mode*. We might prefer to orient the y axis along the short edge and the x axis along the long edge (perhaps so that the orientation of the paper is a better fit to the shape of a television's display). This is termed *landscape mode*. A rotation of $\pi/2$ and a translation to reposition the origin in the lower-left corner will do this.

We have already seen an example of a coordinate transformation, although we did not use transformation-matrix notation. The example is the transformation from normalized device coordinates to actual device coordinates. The arithmetic we used to do the conversion was

```
XI ← X * WIDTH + WIDTH-START;
YI ← Y * HEIGHT + HEIGHT-START;
```

We can see now that this is a scale by WIDTH for x and HEIGHT for y, followed by a translation by WIDTH-START and HEIGHT-START. The full transformation matrix for this change in coordinates is

$$D = \begin{vmatrix} \text{WIDTH} & 0 & 0 \\ 0 & \text{HEIGHT} & 0 \\ \text{WIDTH - START} & \text{HEIGHT - START} & 1 \end{vmatrix} \qquad (4.43)$$

ROTATION ABOUT AN ARBITRARY POINT

Now let's determine the transformation matrix for a counterclockwise rotation about point (x_C, y_C). (See Figure 4-9.)

We shall do this by three transformation steps. We shall translate the point (x_C, y_C) to the origin, rotate about the origin, and then translate the center of rotation back where it belongs. (See Figure 4-10.)

Matrix multiplication is not commutative. Multiplying A times B will not always yield the same result as multiplying B times A. We must be careful to order the matrices so that they correspond to the order of the transformations on the image. We shall place the coordinates of the point on the left and the transformation matrix on the right. With this ordering, if an additional matrix product is introduced on the right (*post-multiplication*), then the corresponding transformation will be carried out after the original transformation. If an additional matrix product is introduced on the left of the original transformation matrix (between it and the coordinates of the point), then the new transformation takes place prior to the original transformation. Multiplying on the left is called *pre-multiplication*. We use post-multiplication in the construction of our general rotation.

The translation which moves (x_C, y_C) to the origin is

$$T_1 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_C & -y_C & 1 \end{vmatrix} \tag{4.44}$$

the rotation is

$$R = \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.45}$$

and the translation to move the center point back to its correct position is

$$T_2 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_C & y_C & 1 \end{vmatrix} \tag{4.46}$$

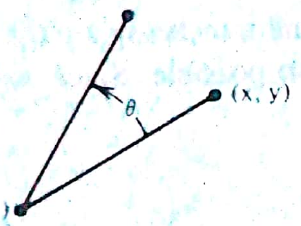

FIGURE 4-9
Rotation about an arbitrary point

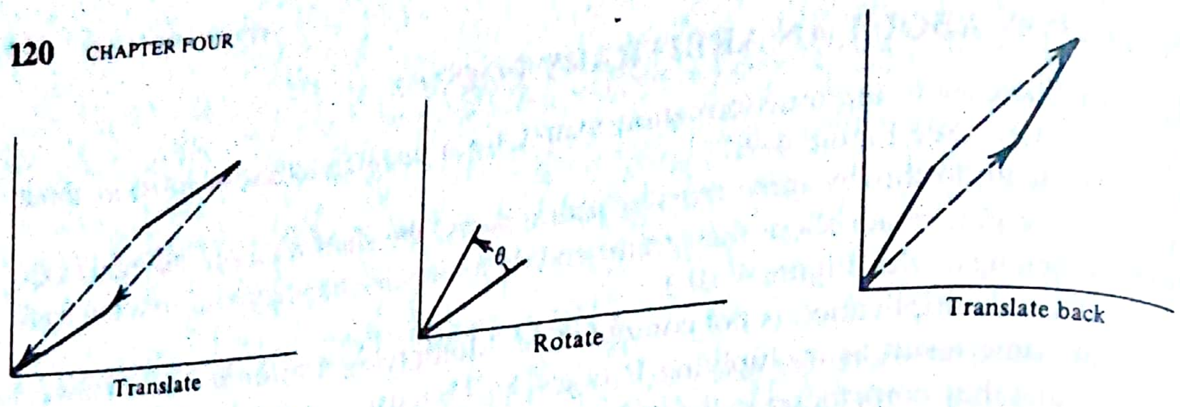

FIGURE 4-10
Three steps in the rotation about an arbitrary point.

To transform a point, we would multiply

$$((([xw \quad yw \quad w]\, T_1)R)\, T_2)$$

but we reassociate and multiply all the transformation matrices together first to form an overall transformation matrix

$$[xw \quad yw \quad w](T_1(R\, T_2))$$

$$T_1 R T_2 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_C & -y_C & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_C & y_C & 1 \end{vmatrix}$$

$$= \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_C & -y_C & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ x_C & y_C & 1 \end{vmatrix}$$

$$= \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ -x_C\cos\theta + y_C\sin\theta + x_C & -x_C\sin\theta - y_C\cos\theta + y_C & 1 \end{vmatrix} \qquad (4.47)$$

This is the overall transformation for a rotation by θ counterclockwise about the point (x_C, y_C). Note that this matrix may also be formed by an initial rotation of θ, followed by a single translation by the values in its third row.

OTHER TRANSFORMATIONS

The three transformations of scaling, rotating, and translating are the most useful and the most common. Other transformations are also possible. Since any 2 × 2 transformation matrix

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

can be converted to a 3 × 3 homogeneous coordinate matrix as

$$\begin{vmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

we will present only the 2 × 2 form for some of these transformations:

$$\begin{vmatrix} -1 & 0 \\ 0 & 1 \end{vmatrix} \quad \text{reflection in the y axis (see Figure 4-11)}$$

$$\begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix} \quad \text{reflection in the x axis (see Figure 4-12)}$$

$$\begin{vmatrix} -1 & 0 \\ 0 & -1 \end{vmatrix} \quad \text{reflection in the origin (see Figure 4-13)}$$

$$\begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix} \quad \text{reflection in the line } y = x \text{ (see Figure 4-14)}$$

$$\begin{vmatrix} 0 & -1 \\ -1 & 0 \end{vmatrix} \quad \text{reflection in the line } y = -x \text{ (see Figure 4-15)}$$

$$\begin{vmatrix} 1 & a \\ 0 & 1 \end{vmatrix} \quad \text{y shear (see Figure 4-16)}$$

$$\begin{vmatrix} 1 & 0 \\ b & 1 \end{vmatrix} \quad \text{x shear (see Figure 4-17)}$$

The first three reflections are just scales with negative scale factors. The reflections in the lines $y = x$ and $y = -x$ can be done by a scale followed by a rotation.

The shear transformations cause the image to slant. The y shear preserves all the x-coordinate values but shifts the y value. The amount of change in the y value depends upon the x position. This causes horizontal lines to transform into lines which slope up or down. The x shear maintains the y coordinates, but changes the x values which causes vertical lines to tilt right or left. It is possible to form the shear transformations out of sequences of rotations and scales, although it is much easier to just form the matrix directly. It is also possible to build rotation and some scaling transformations out of shear transformations.

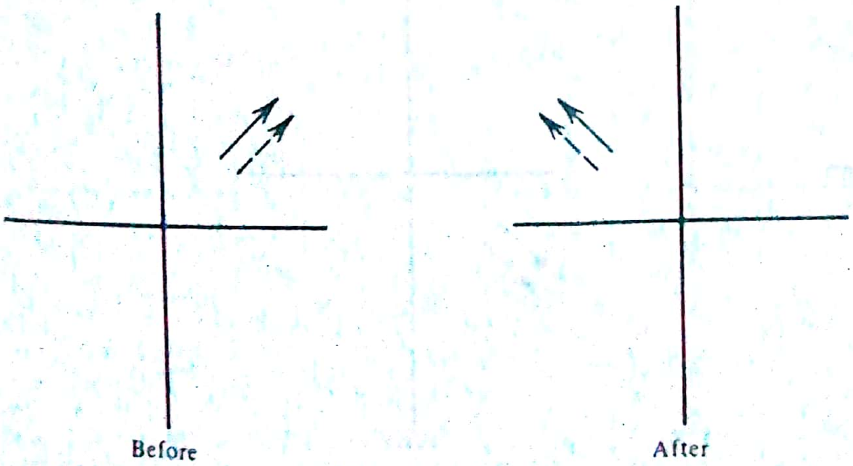

FIGURE 4-11
Reflection in the y axis.

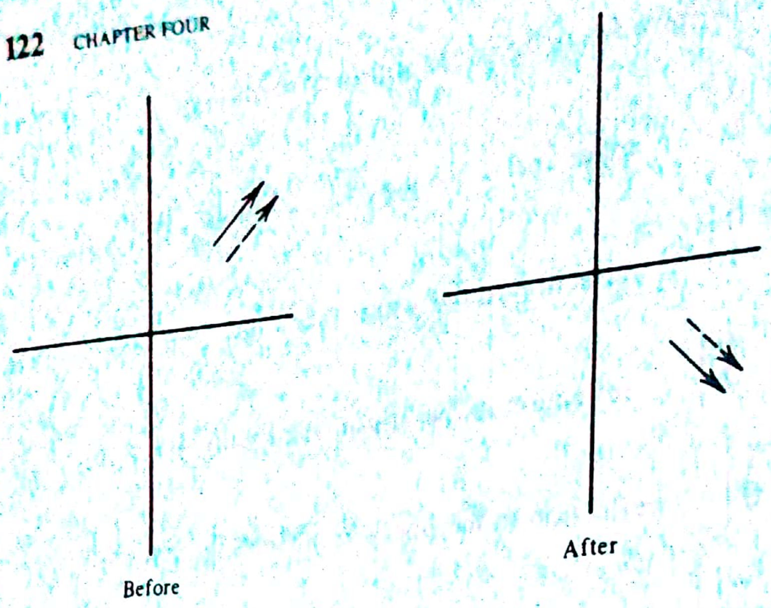

FIGURE 4-12
Reflection in the x axis.

INVERSE TRANSFORMATIONS

We have seen how to use transformations to map each point (x, y) into a new point (x′, y′). Sometimes, however, we are faced with the problem of undoing the effect of a transformation; given the transformed point (x′, y′), we must find the original point (x, y). An example occurs when the user indicates a particular position on a displayed image. The display may show an object which has undergone a transformation. If we want to know the corresponding point on the original object, then we must undo the transformation.

Undoing a transformation appears to be a transformation itself. Given any point (x′, y′), we need a way of calculating a new point (x, y). Is there perhaps a transformation matrix which will do this? Often there is, and it can be determined by *matrix inver-*

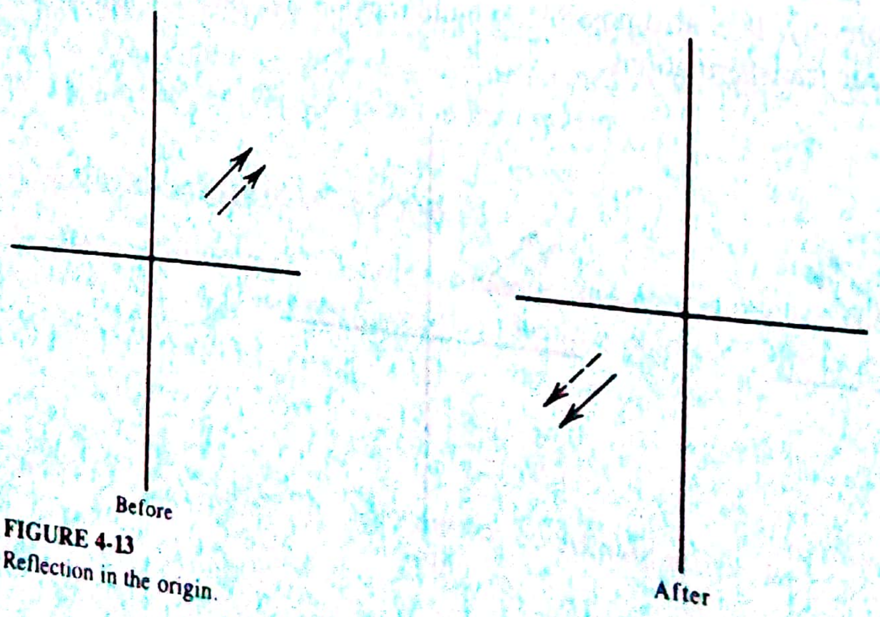

FIGURE 4-13
Reflection in the origin.

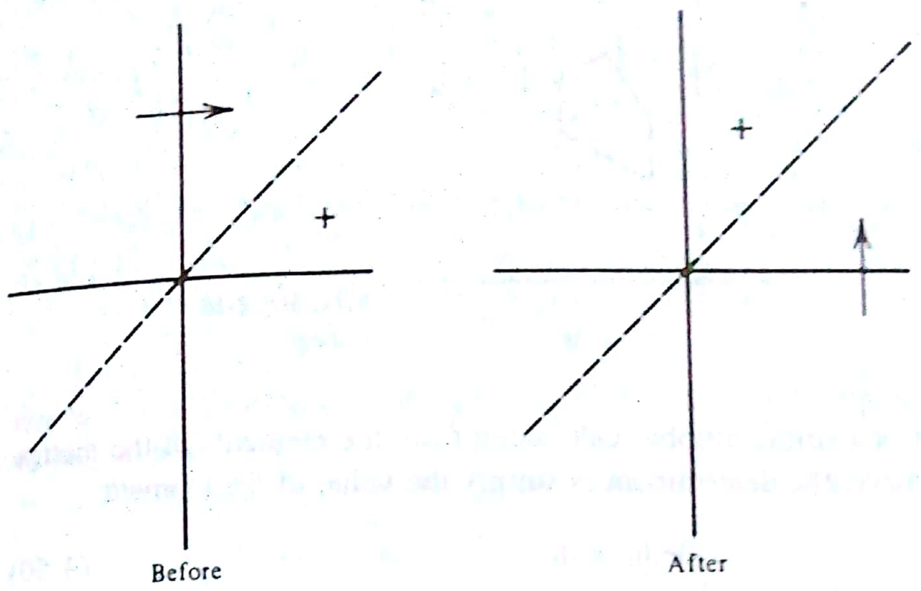

FIGURE 4-14
Reflection in the line $y = x$.

sion. The inverse of a matrix is another matrix such that when the two are multiplied together, the identity matrix results.

If the inverse of matrix T is T^{-1}, then

$$TT^{-1} = T^{-1}T = I \tag{4.48}$$

To see that this is what we need, consider Equation 4.11 which transforms point P_1 to yield point P_2. If we multiply both sides of this equation by the inverse of transformation matrix T, we get

$$P_2T^{-1} = P_1TT^{-1} = P_1I = P_1 \tag{4.49}$$

This shows that the inverse of T transforms P_2 back into P_1.

This inverse of a matrix can be a handy thing to have, so we shall show how to find it. We shall do so in terms of another matrix calculation called the *determinant*. The

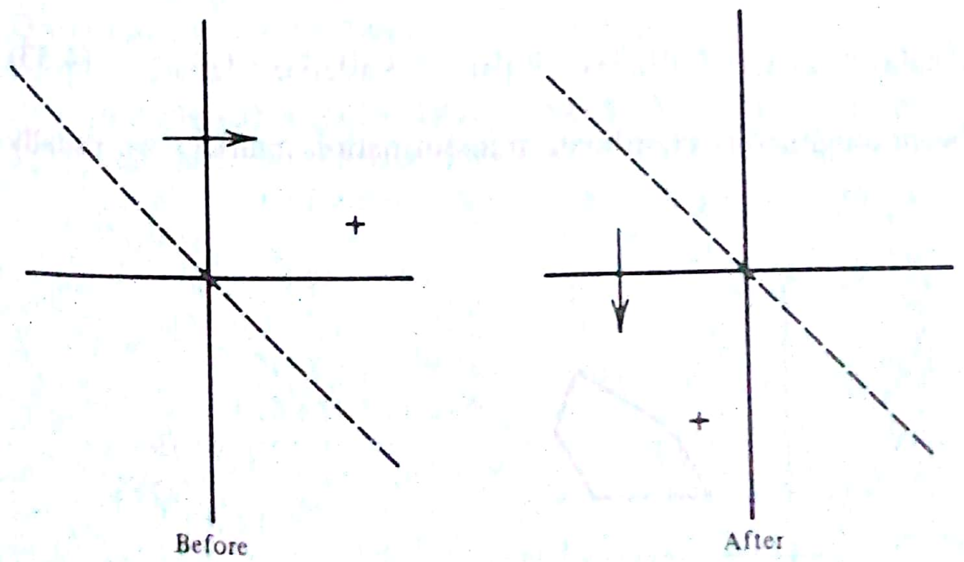

FIGURE 4-15
Reflection in the line $y = -x$.

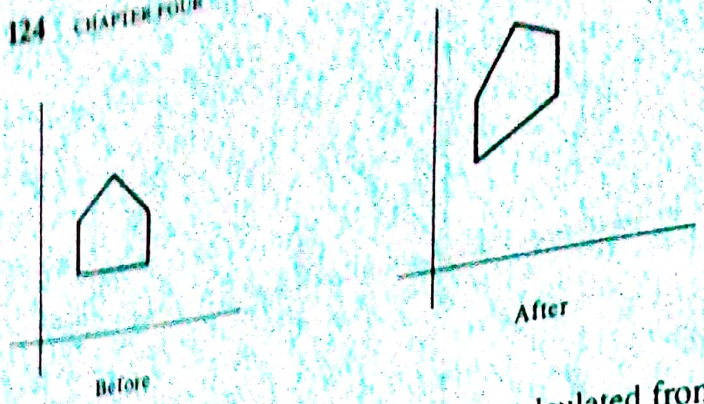

FIGURE 4-16
y shear.

determinant of a matrix is a single number calculated from the elements of the matrix. For a single element matrix, the determinant is simply the value of the element.

$$\det|t| = t \tag{4.50}$$

For larger matrices, we express the determinant as a combination of the determinants of smaller matrices. The smaller matrices are called the *minors* and are made by removing a row and column from the larger matrix. Call M_{ij} the matrix formed by removing row i and column j from matrix T. Then the determinant of T is given by

$$\det T_j = \Sigma\, t_{ij}(-1)^{i+j} \det M_{ij} \tag{4.51}$$

This says that we pick some row of the original matrix T and multiply each element in the row by the determinant of the minor for that position, alternating signs. The sum of these numbers is the full determinant.

The determinant of a 2 × 2 matrix is

$$\det \begin{vmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{vmatrix} = t_{11}t_{22} - t_{12}t_{21} \tag{4.52}$$

The determinant of a 3 × 3 matrix is

$$\det T = t_{11}(t_{22}t_{33} - t_{23}t_{32}) - t_{12}(t_{21}t_{33} - t_{23}t_{31}) + t_{13}(t_{21}t_{32} - t_{22}t_{31}) \tag{4.53}$$

The determinant of the homogeneous coordinate transformation matrices we usually deal with is

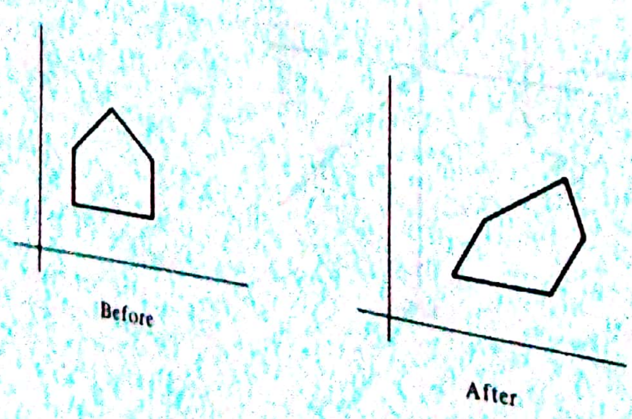

FIGURE 4-17
x shear.

$$\det \begin{vmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{vmatrix} = ae - bd \qquad (4.54)$$

Now we can express the inverse of a matrix in terms of determinants.

$$t'_{ij} = \frac{(-1)^{i+j} \det M_{ji}}{\det T} \qquad (4.55)$$

where t'_{ij} is an element of the inverse of matrix T. Note that the order of i and j is reversed on the minor.

The inverse of the homogeneous coordinate transformation matrix is

$$\text{inv} \begin{vmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{vmatrix} = \frac{1}{ae - bd} \begin{vmatrix} e & -d & 0 \\ -b & a & 0 \\ bf - ce & cd - af & ae - bd \end{vmatrix} \qquad (4.56)$$

TRANSFORMATION ROUTINES

Now that we have seen the mathematics behind transformations, let's look at some algorithms to actually perform them. We shall construct routines for translating, rotating, and scaling. The routines that create the transformations will modify a homogeneous coordinate transformation matrix. This matrix can then be applied to any point to obtain the corresponding transformed point. While we use the notion of homogeneous coordinates, we shall not actually store the third coordinate w; instead, we shall arrange our transformations so that w will always be 1. We can therefore just use 1 whenever we need the coordinate w. For the same reason, we shall not store the last column of the transformation matrix. This column would always contain 0, 0, 1; and since we know this, we can avoid actually storing these numbers. We shall save the 3 × 3 homogeneous transformation matrix in the 3 × 2 array named H.

We begin with a routine to set the transformation matrix to the identity matrix. This clears the transformation so that we can start fresh.

4.1 Algorithm IDENTITY-MATRIX (H) Routine to create the identity transformation

```
Argument    H is a transformation array of 3 × 2 elements.
Local       I, J variables for stepping through the H array
BEGIN
    FOR I = 1 TO 3 DO
        FOR J = 1 TO 2 DO
            IF I = J THEN H[I, J] ← 1
            ELSE H[I, J] ← 0;
    RETURN;
END;
```

The next algorithm causes a scaling transformation. It has the effect of multiplying the matrix H on the right by a scaling transformation matrix of

$$\begin{vmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

We notice that there are a lot of zeros in this matrix, so if we follow the multiplication steps of Equation 4.3, we will be multiplying and adding many noncontributing terms. The algorithm given here avoids this wasted effort by only dealing with the nonzero terms.

4.2 Algorithm MULTIPLY-IN-SCALE (SX, SY, H) Routine to post-multiply the transformation matrix by a scale transformation

```
Arguments   SX is the x scale factor
            SY is the y scale factor
            H is a 3 × 2 transformation matrix
Local       I for stepping through the array
BEGIN
   FOR I = 1 TO 3 DO
      BEGIN
         H[I, 1] ← H[I, 1] * SX;
         H[I, 2] ← H[I, 2] * SY;
      END;
   RETURN;
END;
```

For translation TX, TY, we post-multiply H by the translation matrix

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ TX & TY & 1 \end{vmatrix}$$

Again we simplify the multiplication by neglecting zero terms. Since the third column of the transformation matrix is always 0, 0, 1, the algorithm for translation is as follows.

4.3 Algorithm MULTIPLY-IN-TRANSLATION (TX, TY, H) Routine to post-multiply the transformation matrix by a translation

```
Arguments   TX translation in the x direction
            TY translation in the y direction
            H a 3 × 2 transformation matrix
BEGIN
   H[3, 1] ← H[3, 1] + TX;
   H[3, 2] ← H[3, 2] + TY;
   RETURN;
END;
```

For a rotation of A radians counterclockwise, we post-multiply by

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

The algorithm takes the angle as an argument, calculates the sine and cosine, and then performs the matrix multiplication for nonzero terms.

4.4 Algorithm MULTIPLY-IN-ROTATION(A, H) Routine to post-multiply the transformation matrix by a rotation

```
Arguments   A angle of counterclockwise rotation
            H a 3 × 2 transformation matrix
Local       S, C the sine and cosine values
            I for stepping through the array
            TEMP temporary storage of the first column
BEGIN
   C ← COS(A);
   S ← SIN(A);
   FOR I = 1 TO 3 DO
      BEGIN
         TEMP ← H[I, 1] * C − H[I, 2] * S;
         H[I, 2] ← H[I, 1] * S + H[I, 2] * C;
         H[I, 1] ← TEMP;
      END;
   RETURN;
END;
```

The above routines serve to create a transformation matrix, but we still need a routine to apply the resulting transformation. The following routine transforms a single point. The point coordinates are passed to the subroutine as arguments, and the transformed point is returned in the same variables.

4.5 Algorithm DO-TRANSFORMATION(X, Y, H) Routine to transform a point

```
Arguments   X, Y the coordinates of the point to be transformed
            H a 3 × 2 transformation matrix
Local       TEMP temporary storage for the new X value.
BEGIN
   TEMP ← X * H[1, 1] + Y * H[2, 1] + H[3, 1];
   Y ← X * H[1, 2] + Y * H[2, 2] + H[3, 2];
   X ← TEMP;
   RETURN;
END;
```

To perform a transformation, we must form the appropriate transformation matrix and then apply it to the points in our display. There are several ways this might be done. In our approach we shall deny the user access to the MULTIPLY-IN-SCALE, MULTIPLY-IN-TRANSLATION, and MULTIPLY-IN-ROTATION routines. The user will therefore not be able to build up complex transformations by multiplying several scales, translations, or rotations (if he wishes to do this, he will have to write his own routines). Instead, the user will be allowed only one scale, one rotation, and one translation, to be applied in that order. The transformations will be applied as the display file is interpreted. The user can change the values of these transformations at any time, but the image will be formed using the values in effect at the time the display file is in-

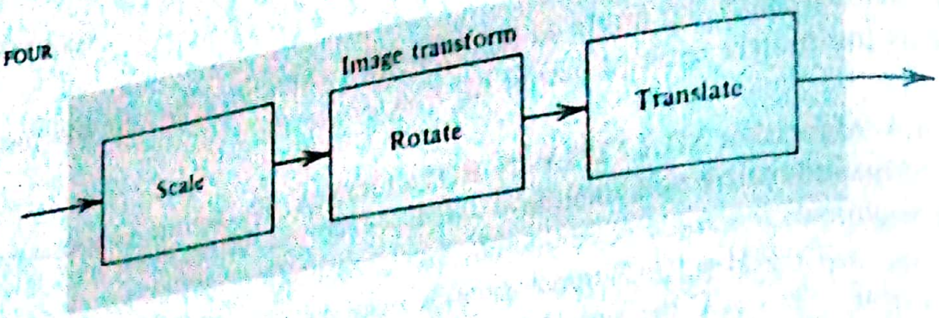

FIGURE 4-18
The image transformation.

terpreted. The following three algorithms describe user routines for saving the transformation parameters until it is time to interpret the display file.

4.6 Algorithm TRANSLATE(TX, TY) User routine to set the translation parameters

```
Arguments   TX, TY the translation amount
Global      TRNX, TRNY storage for the translation parameters
BEGIN
   TRNX ← TX;
   TRNY ← TY;
   CALL NEWFRAME;
   RETURN;
END;
```

4.7 Algorithm SCALE(SX, SY) User routine to set the scaling parameters

```
Arguments   SX, SY the scaling factors
Global      SCLX, SCLY storage for the scale parameters
BEGIN
   SCLX ← SX;
   SCLY ← SY;
   CALL NEWFRAME;
   RETURN;
END;
```

4.8 Algorithm ROTATE(A) User routine to set the rotation angle

```
Argument    A the rotation angle
Global      ANGL a place to save the rotation angle
BEGIN
   ANGL ← A;
   CALL NEWFRAME;
   RETURN;
END;
```

We now need a routine which will take the transformation parameters specified by the user and use them to form a transformation matrix. This is just a matter of passing the values to the transformation matrix multiplication routines which we defined above. (See Figure 4-18.)

4.9 Algorithm BUILD-TRANSFORMATION Routine to build the image transformation matrix

```
Global      ANGL, SCLX, SCLY, TRNX, TRNY the transformation parameters
            IMAGE-XFORM a 3 × 2 array containing the image transformation
```

```
BEGIN
   IDENTITY-MATRIX(IMAGE-XFORM);
   MULTIPLY-IN-SCALE(SCLX, SCLY, IMAGE-XFORM);
   MULTIPLY-IN-ROTATION(ANGL, IMAGE-XFORM);
   MULTIPLY-IN-TRANSLATION(TRNX, TRNY, IMAGE-XFORM);
   RETURN;
END;
```

To apply these transformations to a picture, we shall modify three of the algorithms from previous chapters. We shall extend the MAKE-PICTURE-CURRENT algorithm to include a call on BUILD-TRANSFORMATION. Thus the parameter settings at the time the MAKE-PICTURE-CURRENT routine is executed will be the values used for the transformation.

4.10 Algorithm MAKE-PICTURE-CURRENT (Algorithm 2.12 revisited) User routine to show the current display file

```
Global    FREE the index of the next free display-file cell
          ERASE-FLAG indicates if frames should be cleared
BEGIN
   IF ERASE-FLAG THEN
      BEGIN
         ERASE;
         ERASE-FLAG ← FALSE;
      END;
   BUILD-TRANSFORMATION;
   IF FREE > 1 THEN INTERPRET(1, FREE − 1);
   DISPLAY;
   FREE ← 1;
   RETURN;
END;
```

Each point retrieved from the display should be multiplied by the transformation matrix, and the resulting product should be used. This can be accomplished by modifying the INTERPRET and LOAD-POLYGON routines to call GET-TRANSFORMED-POINT instead of GET-POINT. (See Figure 4-19.)

4.11 Algorithm INTERPRET(START, COUNT) (Algorithm 3.6 revisited) Scan the display file performing the instructions

```
Arguments  START the starting index of the display-file scan
           COUNT the number of instructions to be interpreted
Local      NTH the display-file index
           OP, X, Y the display file instruction
BEGIN
   a loop to do all desired instructions
   FOR NTH = START TO START + COUNT − 1 DO
      BEGIN
         GET-TRANSFORMED-POINT(NTH, OP, X, Y);
         IF OP < −31 THEN DOCHAR(OP, X, Y)
```

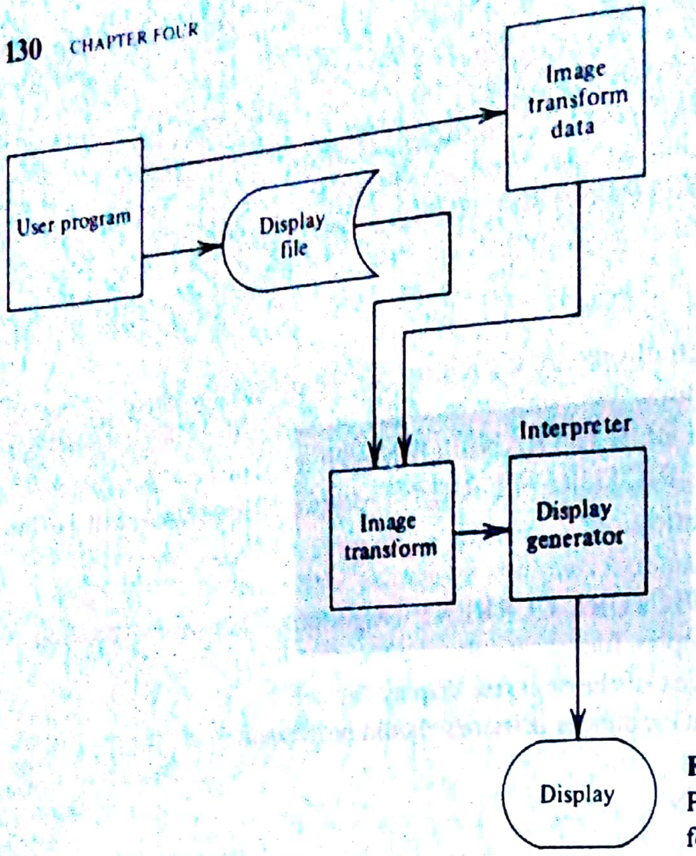

Figure 4-19
Picture generation with image transformation.

```
            ELSE IF OP < 1 THEN DOSTYLE(OP)
                ELSE IF OP = 1 THEN DOMOVE(X, Y)
                    ELSE IF OP = 2 THEN DOLINE(X, Y)
                        ELSE DOPOLYGON(OP, X, Y, NTH);
        END;
    RETURN;
END;
```

4.12 Algorithm LOAD-POLYGON(I, EDGES) (A modification of algorithm 3.9) A routine to retrieve polygon side information from the display file
Positions are converted to actual screen coordinates

Arguments	I the display-file index of the instruction
	EDGES for return of the number of sides stored
Global	WIDTH-START, HEIGHT-START starting index of the screen
	WIDTH width in pixels of the screen
	HEIGHT height in pixels of the actual screen
Local	X1, Y1, X2, Y2 edge endpoints in device coordinates
	I1 for stepping through the display file
	K for stepping through the polygon sides
	DUMMY for a dummy argument

```
BEGIN
    set starting point for a side
    GET-TRANSFORMED-POINT(I, SIDES, X1, Y1);
    X1 ← X1 * WIDTH + WIDTH-START + 0.5;
    adjust y coordinate to nearest scan line
    Y1 ← INT(Y1 * HEIGHT + HEIGHT-START + 0.5);
    get index of first side command
    I1 ← I + 1;
    initialize an index for storing side data
    EDGES ← 1;
    a loop to get information about each side
    FOR K = 1 TO SIDES DO
        BEGIN
            get next vertex
            GET-TRANSFORMED-POINT(I1, DUMMY, X2, Y2);
            X2 ← X2 * WIDTH + WIDTH-START + 0.5;
            Y2 ← INT( Y2 * HEIGHT + HEIGHT-START + 0.5);
            see if horizontal line
            IF Y1 = Y2 THEN X1 ← X2
            ELSE
                BEGIN
                    save data about side in order of largest y
                    POLY-INSERT(EDGES, X1, Y1, X2, Y2);
                    increment index for side data storage
                    EDGES ← EDGES + 1;
                    old point is reset
                    Y1 ← Y2:
                    X1 ← X2;
                END;
            I1 ← I1 + 1;
        END;
    set EDGES to be a count of the edges stored.
    EDGES ← EDGES − 1;
    RETURN;
END;
```

The GET-TRANSFORMED-POINT routine retrieves an instruction from the display file and applies the transformation to it.

4.13 Algorithm GET-TRANSFORMED-POINT(NTH, OP, X, Y) Retrieve and transform the Nth instruction from the display file

```
Argument    NTH the index of the desired instruction
            OP, X, Y the instruction to be returned
Global      IMAGE-XFORM a 3 × 2 array containing the image transformation
BEGIN
    GET-POINT(NTH, OP, X, Y);
    IF OP > 0 OR OP < −31 THEN DO-TRANSFORMATION(X, Y, IMAGE-XFORM);
    RETURN;
END;
```

TRANSFORMATIONS AND PATTERNS

In Chapter 3 we described how patterns might be used to fill polygons. We showed how to implement patterns which are registered to the imaging surface. This is what is needed for patterns which give gray levels or simple textures such as stripes or weaves.

But there is an alternative use of patterns where it would be better if the pattern could be moved with respect to the imaging surface. This occurs when patterns are used to display pictures. It is sometimes better to represent a picture directly as a pixel pattern, instead of using lines and polygons. This may be because the shapes in the picture are curved (for example, characters). Or it may be that there are no lines at all, as often occurs with pictures extracted from photographs. For this type of pattern we may want to be able to move it around on the display. We might also like to be able to change the scale and orientation. We might like to rotate the picture or make it bigger. In other words, we would like to apply a transformation to the pattern. (See Figure 4-20.)

We shall show one way that this can be done using the image transformation which we have just developed. The resulting program will behave in the following way. Suppose we create a polygon and a pattern for filling it. Now suppose that we apply an image transformation to scale the polygon; we shall also scale the pattern so that the picture looks the same except for size. If we use the image transformation to rotate the polygon, the pattern will rotate with it, and if we use the image transformation to translate the polygon, the pattern it is filled with will be translated, too. The way we shall implement this is by extending the FILLIN algorithm. FILLIN determines the intensity value for each point in the polygon. In Chapter 3 we showed how the intensity for a point [x, y] could be found by looking in a pattern table; but now,

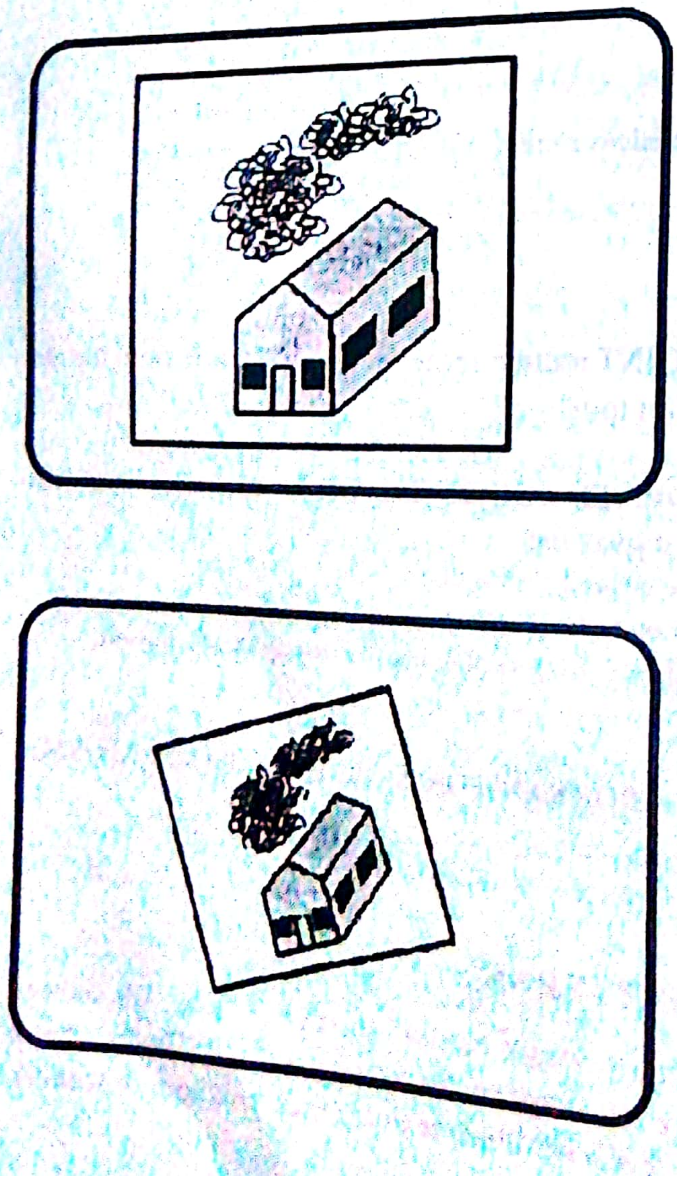

FIGURE 4-20
When the filling pattern is a picture, it should be transformed with the polygon.

FILLIN must find the intensity value for the transformed point [x', y'] = [x, y]H, where H is the image transformation and [x, y] is in normalized device coordinates. Given the point [x', y'], we need to find the point [x, y] and use it to look up the intensity in the pattern table. In effect, we are asking for the intensity of the original pattern at the point which gets transformed into the point being imaged. To find [x, y] from [x', y'], we need the inverse transformation. We can build the inverse transformation from the inverses of the scale, rotation, and translation components.

The inverse of the scaling transformation of Equation 4.37 is

$$S^{-1} = \begin{vmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.57}$$

For example, scaling by one-half undoes the effect of scaling by two.

The inverse of the rotation matrix of Equation 4.39 is

$$R^{-1} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} \cos(-\theta) & \sin(-\theta) & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{4.58}$$

This says that we can undo the effect of a counterclockwise rotation by a rotation of the same amount in the clockwise direction.

Finally, the inverse of the translation matrix of Equation 4.41 is

$$T^{-1} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -t_x & -t_y & 1 \end{vmatrix} \tag{4.59}$$

We undo the effect of a translation by translating the same amount in the opposite direction.

Our image transformation is the product of a scale, a rotation, and a translation. Its inverse can be built from the inverses of its components. However, when we take something apart, we do it in the reverse sequence of when we put it together, and the inverse of the image transformation H is the product of its components in reverse order.

$$H^{-1} = (S\,R\,T)^{-1} = T^{-1}\,R^{-1}\,S^{-1} \tag{4.60}$$

This is almost what we need for transforming the pattern. The image transformation is applied to normalized device coordinates, but the points in FILLIN have been transformed to the actual device coordinates; so in order to apply H^{-1}, we should first convert the point back to the normalized coordinates by applying the inverse of this coordinate transformation. Once we have applied the inverse of the image transformation, we must convert the result back again to the actual device coordinates. This may seem like a lot of work, but actually it is not, because the coordinate transformation matrix (and its inverse) can be multiplied in with the inverse image transformation matrix so that there is only a single net transformation which we must apply to each point.

We saw the transformation from normalized device coordinates to actual device coordinates in Equation 4.43. This is the product of a scale (S_n) by the WIDTH and HEIGHT, followed by a translation (T_n) by WIDTH-START and HEIGHT-START. So the full transformation which should be applied to each point of the polygon being filled is

$$T_n^{-1}\, S_n^{-1}\, H^{-1}\, S_n\, T_n$$

Let's extend the BUILD-TRANSFORMATION algorithm so that it builds both the image transformation and the inverse transformation needed to transform patterns.

4.14 Algorithm BUILD-TRANSFORMATION (Revision of algorithm 4.9) Routine to build the image transformation matrix and its inverse

```
Global   ANGL, SCLX, SCLY, TRNX, TRNY the transformation parameters
         IMAGE-XFORM a 3 × 2 array containing the image transformation
         INVERSE-IMAGE-XFORM a 3 × 2 array for the inverse of the image
         transformation

BEGIN
   IDENTITY-MATRIX(IMAGE-XFORM);
   MULTIPLY-IN-SCALE(SCLX, SCLY, IMAGE-XFORM);
   MULTIPLY-IN-ROTATION(ANGL, IMAGE-XFORM);
   MULTIPLY-IN-TRANSLATION(TRNX, TRNY, IMAGE-XFORM);
   IDENTITY-MATRIX(INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-TRANSLATION(– WIDTH-START, – HEIGHT-START,
      INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-SCALE(1 / WIDTH, 1 / HEIGHT, INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-TRANSLATION(– TRNX, – TRNY, INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-ROTATION(– ANGL, INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-SCALE(1 / SCLX, 1 / SCLY, INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-SCALE(WIDTH, HEIGHT, INVERSE-IMAGE-XFORM);
   MULTIPLY-IN-TRANSLATION(WIDTH-START, HEIGHT-START,
      INVERSE-IMAGE-XFORM);
   RETURN;
END;
```

Before presenting the modified FILLIN algorithm, let's define a flag which the user can set to indicate whether or not he wants patterns to be transformed.

4.15 Algorithm SET-TRANSFORM-PATTERN(ON-OFF) User routine to indicate whether polygon fill patterns should undergo the image transformation

```
Argument   ON-OFF the user's choice
Global     XFORM-PATTERN a flag to indicate transformation of patterns
BEGIN
   XFORM-PATTERN ← ON-OFF;
   RETURN;
END;
```

Now we shall revise the FILLIN algorithm to allow image transformations on the patterns used to fill polygons.

4.16 Algorithm FILLIN(X1, X2, Y) (Revision of algorithm 3.17) Fills in scan line Y from X1 to X2

```
Arguments   X1, X2 end positions of the scan line to be filled
            Y the scan line to be filled
Global      FILL-PATTERN pattern table index of the pattern to use
            PATTERN-X, PATTERN-Y, PATTERNS the pattern table
            FRAME the two-dimensional frame buffer array
            XFORM-PATTERN a flag to indicate transformation of patterns
            INVERSE-IMAGE-XFORM a 3 × 2 array for the inverse of the image
            transformation
Local       X for stepping across the scan line
            PX, PY for accessing the pattern
            PATTERN-TO-USE the pattern to be used in filling
            PAT-X, PAT-Y the x and y dimensions of the pattern
BEGIN
   IF X1 = X2 THEN RETURN;
   PAT-X ← PATTERN-X[FILL-PATTERN];
   PAT-Y ← PATTERN-Y[FILL-PATTERN];
   if a local array is used as shown in the following statement,
   then the implementation should avoid copying of individual elements
   PATTERN-TO-USE ← PATTERNS[FILL-PATTERN];
   IF XFORM-PATTERN THEN
      BEGIN
         FOR X = X1 TO X2 DO
            BEGIN
               PX ← X;
               PY ← Y;
               DO-TRANSFORMATION(PX, PY, INVERSE-IMAGE-XFORM);
               PX ← MOD(INT(PX + 0.5), PAT-X) + 1;
               PY ← MOD(INT(PY + 0.5), PAT-Y) + 1;
               FRAME[X, Y] ← PATTERN-TO-USE[PX, PY];
            END;
      END
   ELSE
      BEGIN
         PX ← MOD(X1, PAT-X) + 1;
         PY ← MOD(Y, PAT-Y) + 1;
         FOR X = X1 TO X2 DO
            BEGIN
               FRAME[X, Y] ← PATTERN-TO-USE[PX, PY];
               IF PX = PAT-X THEN PX ← 1
               ELSE PX ← PX + 1;
            END;
      END;
   RETURN;
END;
```

Note that for pattern sizes restricted to powers of 2, the MOD operation can be replaced by a logical AND instruction, which is usually much faster.

The algorithm has been extended to include a check for transformation of the pattern. If the transformation should be done, then the inverse image transformation is applied to each point that is to be filled. The result is rounded to the nearest pixel, and MOD operations are used to determine the appropriate pattern element. Although they may look different, this new portion of the algorithm and the portion written in Chapter 3 are really quite similar. In fact, if the transformation is the identity, then the top and bottom halves of the algorithm have exactly the same effect. The difference is that in the untransformed case we can keep track of our position in the pattern, and can thereby avoid doing the MOD operations for every point. In the transformed case we cannot do this, and the MOD operators are needed.

Transformations on patterns should be used for pictures, but should be avoided for simple shading patterns and repetitive designs. There are two reasons for this. One is that the transformation of every pixel in the filled area can be quite time-consuming. The second is that transforming a pattern introduces aliasing effects. When transformed, the pixel positions do not correspond directly to the pattern positions. We must guess the proper pixel intensity from the intensity value of a nearby pattern element. This guess introduces errors. Antialiasing techniques can be used at the expense of more time. For example, we can set the pixel intensity to be a weighted average of the nearby pattern element values.

INITIALIZATION

We need a routine which will initialize the parameters so that the user will not have to set them unless he wishes to. The default parameter values should give the identity transformation.

4.17 Algorithm INITIALIZE-4 Initialization routine

```
BEGIN
    INITIALIZE-3;
    CALL SCALE(1, 1);
    CALL ROTATE(0);
    CALL TRANSLATE(0, 0);
    SET-TRANSFORM-PATTERN(FALSE);
    RETURN;
END;
```

DISPLAY PROCEDURES

There may be times, when the image transformation presented here is inadequate, when the user may wish to perform more than just a single scale, rotation, and translation operation. For example, a rotation about an arbitrary center point is most easily handled by a translation, followed by a rotation, followed by another translation. In these situations, the user should add one or more transformation levels to the system. He should write his own LINE and MOVE routines which multiply their arguments by his own transformation matrix before calling the system LINE-ABS-2 and MOVE-

ABS-2 routines to actually do the drawing. A transformation matrix designed and created by the user may be as complex as necessary, involving many component transformations. Routines written according to this prescription will transform point values before they are entered into the display file. The system extension detailed in the above algorithms, on the other hand, operates on points as they are read from the display file. Transformations may be carried out at both stages of the processing.

One situation in which multiple transformations are useful occurs when a picture is made of a few basic components combined according to some hierarchical structure. For example, we may have a routine which draws a flower petal. By combining petals with different positions and orientations, flowers may be drawn. By combining flowers with different sizes and positions, a flower bush may be created. Transformations of several bushes can form a garden, and so on. (See Figure 4-21.)

Pictures with this structure lend themselves to a subprogram organization. A program to draw a garden could do so by several calls on a subprogram which draws a flower bush. The subprogram which draws the flower bush could do so by several calls on a flower-drawing subprogram. The flower-drawing subprogram could use several calls on the petal subprogram. Notice, however, that these subprogram calls are a little more complicated than those of normal programming languages; they involve the establishment of a transformation matrix. An ordinary subprogram call (CALL PETAL) would always produce the image with the same size and orientation. What is needed is a call which sets up a transformation which is applied to all the points generated in the subprogram before they are entered into the display file, for example,

CALL PETAL WITH SIZE(SX, SY), ANGLE(A), TRANSLATION(TX, TY).

These calls, which involve establishment of a transformation, are named *display procedure calls*, and the subprograms which draw subpictures are known as *display procedures*. Because display procedure calls can be nested, there can be multiple transfor-

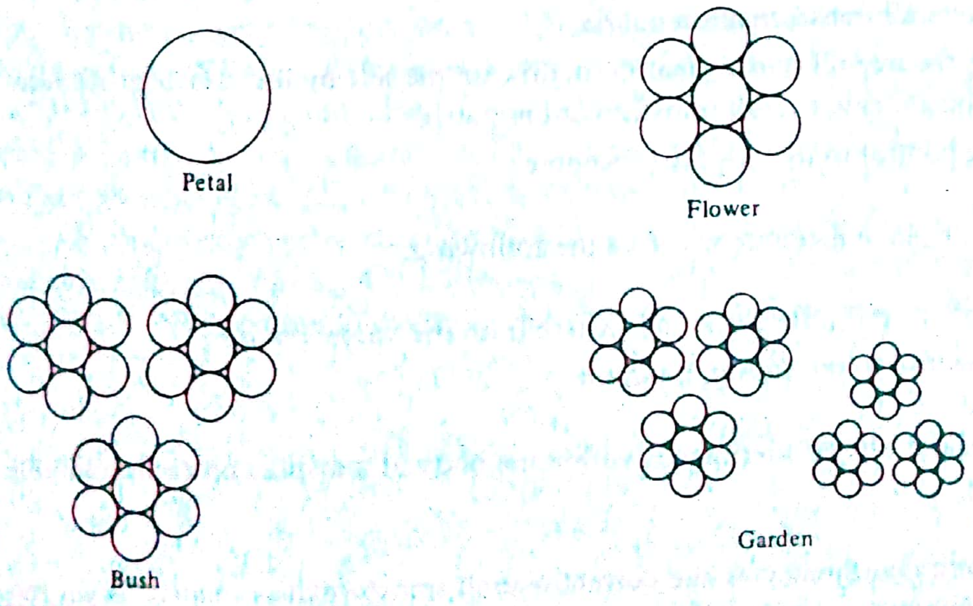

FIGURE 4-21
Hierarchical picture structure.

mations. Suppose, for example, that the GARDEN program calls the FLOWER-BUSH display procedure with transformation T_1. FLOWER-BUSH calls the FLOWER display procedure with transformation T_2. Finally, FLOWER calls the PETAL display procedure with transformation T_3. Then, the overall transformation applied to points generated in PETAL would be the product $T_3T_2T_1$. We can see that each time a new display procedure call is executed, the overall transformation matrix is multiplied on the left by the transformation for that display procedure. Here is a case where pre-multiplication is appropriate. In the GARDEN procedure, we have the identity matrix I. When FLOWER-BUSH is called, this is multiplied by T_1 to give T_1. When this, in turn, calls FLOWER, the T_2 transformation is multiplied in to give T_2T_1. Finally, when PETAL is called, the third transformation is also present: $T_3T_2T_1$. What happens when PETAL finishes and control is returned to FLOWER? We would expect to have the overall transformation which is appropriate for FLOWER, namely $T_2\,T_1$. Likewise, when control is returned to FLOWER-BUSH from FLOWER, the transformation should be T_1, and when control returns to GARDEN, the overall transformation should once again be the identity I. There must therefore be some way of saving the overall transformation before multiplying by the additional transformation for a new display procedure call, so that it may be restored when control returns from that display procedure. When FLOWER calls PETAL, it saves the current overall transformation T_2T_1, multiplies it by T_3 to get the new overall transformation $T_3T_2T_1$, and transfers control to the PETAL. When PETAL is finished and ready to return, it first restores the overall transformation to the value that was saved, T_2T_1, and then returns control to FLOWER. Each display procedure call must save the current overall transformation matrix. Because there can be nested calls, several transformation matrices may have to be stored simultaneously. One possible data structure for storing these matrices is a stack. The last item stored in a stack is the first item to be removed, so it matches the last-entered–first-returned nature of subroutines.

In summary, then, a display procedure call involves the following:

1. Saving the overall transformation matrix
2. Multiplying the overall transformation matrix on the left by the transformation in the call to form a new overall transformation matrix
3. Transferring control to the display procedure

A return from a display procedure involves the following:

1. Restoring the overall transformation matrix from the value saved
2. Returning control to the calling program

The user's LINE and MOVE commands within the body of a display procedure should do the following:

1. Multiply point coordinates by the current overall transformation matrix to get the transformed point.

2. Enter the transformed values into the display file via the system LINE-ABS-2 or MOVE-ABS-2 commands.

AN APPLICATION

Let's consider how our graphics system might be useful in producing animated films. Animation is done by photographing a sequence of drawings, each slightly different from the previous. For example, to show a person moving his arm, a series of drawings is photographed, each drawing showing the arm at a different position. When the images are displayed one after another by a movie projector, we perceive the arm as moving through the sequence. (See Figure 4-22.)

From the artist's point of view, it is desirable to have only a small amount of motion occurring at any one time. The picture can then be constructed of a background which does not change and a foreground which alters from frame to frame. The foreground may be drawn upon a piece of clear plastic and then overlaid upon the background. If the portion of the image which changes is small, then the foreground image which must be redrawn for each frame is small and will require less work. What may be difficult for the human artist is changing the entire scene. For example, if one wished to give the impression of moving into the scene, one might scale the background, making each frame slightly larger. While this sort of change is difficult for the human artist, it is easy for the computer. It may therefore be beneficial to have the computer generate the background scene, while the human artist superimposes the foreground action.

Let's see how our graphics system could be used to generate the background. First, we must construct the full background drawing by using the LINE and POLYGON commands of the previous chapters. Let us assume that this has been done and that we have a routine which will enter appropriate instructions into the display file. By repeatedly executing these commands, the background images for each photograph may be generated. Suppose that the scene is a city street. To show a character moving down the street, we may really want to keep the character centered and move the scenery past him. This can be done by shifting our background image with the TRANSLATE(TX, TY) command. Before each display, we add a little more to TX. Each frame will show the background shifted a little more to the right. This will give the impression that our character has moved to the left. (See Figure 4-23.)

Or suppose that we wish to show our character approaching a building. We could give this effect by having the building grow larger. It could be done by using the SCALE(SX, SY) routine to make the building grow, and the TRANSLATE(TX, TY)

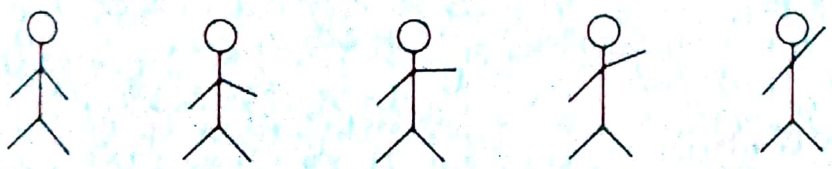

FIGURE 4-22
Animation of arm movement by a series of pictures, each slightly different from the previous.

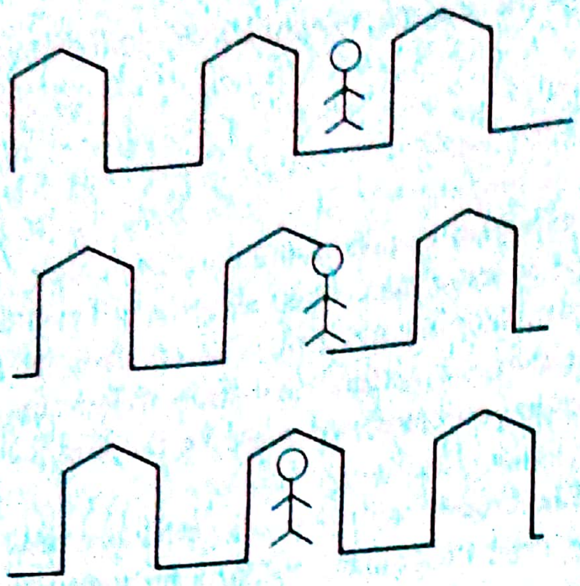

FIGURE 4-23
Apparent movement by translation of the background

routine to keep it centered. With each frame, SX and SY are increased slightly to make the building appear a little larger. (See Figure 4-24.)

FURTHER READING

Other techniques for transforming patterns are found in [BRA80]. Transforming images is also considered in [CAT80]. Aliasing can be a major problem for transformed patterns. One approach to the problem is to find all pattern elements which transform to part of the displayed pixel and then to average their intensity values. A clever way to find the sum of the pattern element intensities is to store the progressive sums of intensity values in the pattern table so that the total intensity for a rectangular area is given by the difference in the values at the corners [CRO84]. A formal description of trans-

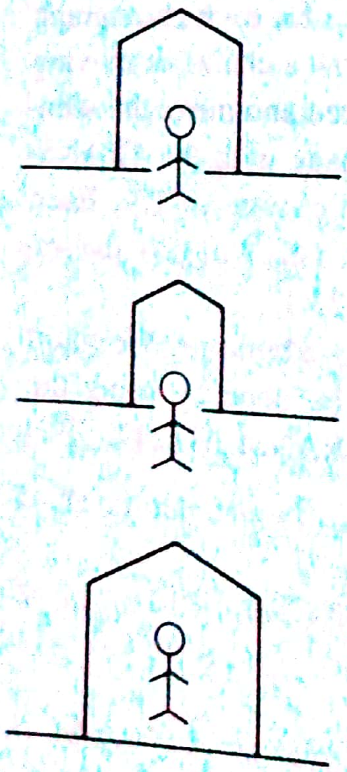

FIGURE 4-24
Apparent motion by scaling of the background.