CHAPTER EIGHT

THREE DIMENSIONS

INTRODUCTION

Some graphics applications are two-dimensional; charts and graphs, certain maps, and some artist's creations may be strictly two-dimensional entities. But we live in a three-dimensional world, and in many design applications we must deal with describing three-dimensional objects. If the architect would like to see how the structure will actually look, then a three-dimensional model can allow him to view the structure from different viewpoints. The aircraft designer may wish to model the behavior of the craft under the three-dimensional forces and stresses. Here again, a three-dimensional description is required. Some simulation applications, such as docking a spaceship or landing an airplane, also require a three-dimensional view of the world.

In this chapter we shall generalize our system to handle models of three-dimensional objects. We shall extend our transformations to allow translation and rotation in three-dimensional space. Since the viewing surface is only two-dimensional, we must consider ways of projecting our object onto this flat surface to form the image. Both parallel and perspective projections will be discussed.

3D GEOMETRY

Let us begin our discussion of three dimensions by reviewing the methods of analytic geometry for specifying objects. The simplest object is, of course, the point. As in the case of two dimensions, we can specify a point by establishing a coordinate system and listing the coordinates of the point. We will need an additional coordinate axis for the third dimension (three axes in all, one for height, one for width, and a third for depth). We can arrange the three axes to be at right angles to each other and label the

Scanned by CamScanner

width and height axes x and y, respectively, to correspond to the two-dimensional case. The third axis, for depth, is named the z axis. We have a choice as to which end of the z axis will be given positive values. (See Figure 8-1.)

The orientation of the coordinate system is determined as follows. If the thumb of the right hand points in the positive z direction as one curls the fingers of the right hand from x into y, then the coordinates are called a *right-handed* system. If, however, the thumb points in the negative z direction then it is *left-handed*. Whereas most of geometry uses a right-handed system, computer graphics often prefers the left-handed coordinates. We shall adopt the right-handed system for our use. (The right-handed system is normally used in mathematics, although in computer graphics we also find the left-handed system in use so that objects behind the display screen will have positive z values.) Any point in space may now be specified by telling how far over, up, and out it is from the origin, where the three axes intersect. These distances are written as the ordered triple (x, y, z). (See Figure 8-2.)

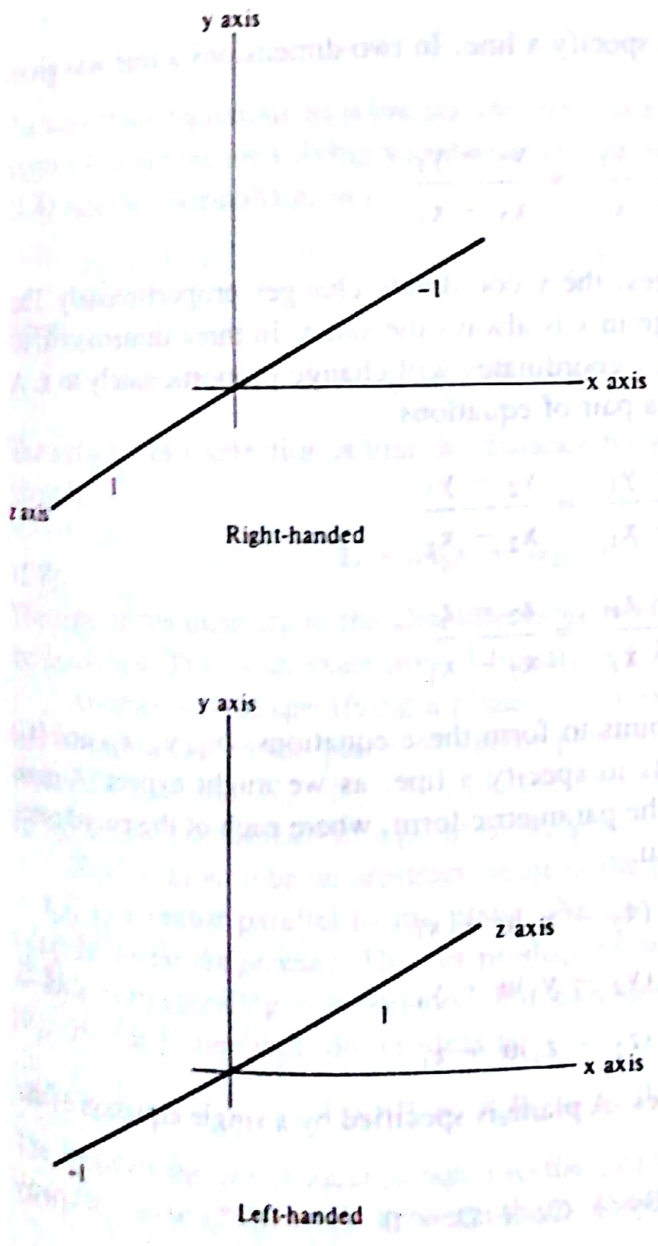


FIGURE 8-1
Right-handed and left-handed coordinate systems.

Scanned by CamScanner

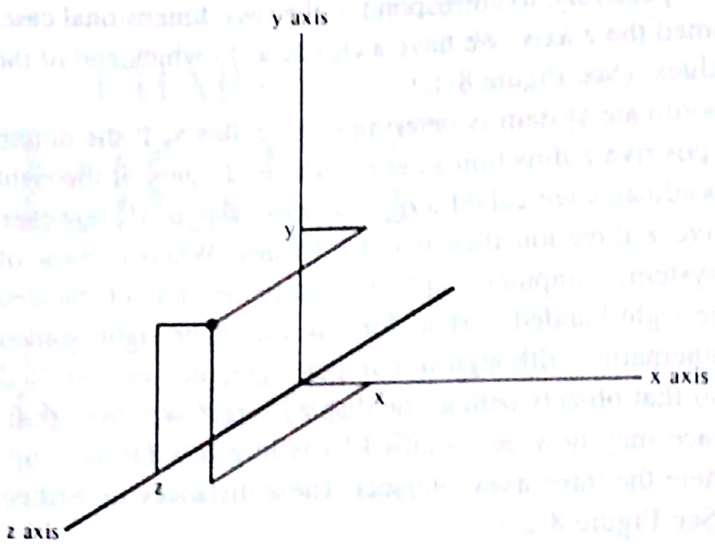


FIGURE 8-2
Position of the point (x, y, z).

We can use the coordinates to specify a line. In two dimensions a line was given by an equation involving x and y.

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \tag{8.1}$$

This equation states that as x changes, the y coordinate changes proportionately. The ratio of the change in y to the change in x is always the same. In three dimensions, as we move along the line, both y and z coordinates will change proportionately to x. A line in three dimensions is given by a pair of equations

$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{y_2 - y_1}{x_2 - x_1} \\ \frac{z - z_1}{x - x_1} &= \frac{z_2 - z_1}{x_2 - x_1} \end{aligned} \tag{8.2}$$

It requires the coordinates of two points to form these equations: (x_1, y_1, z_1) and (x_2, y_2, z_2). Thus it still takes two points to specify a line, as we might expect. A more symmetrical expression for a line is the parametric form, where each of the coordinates is expressed in terms of a parameter u.

$$\begin{aligned} x &= (x_2 - x_1)u + x_1 \\ y &= (y_2 - y_1)u + y_1 \\ z &= (z_2 - z_1)u + z_1 \end{aligned} \tag{8.3}$$

We shall also want to work with planes. A plane is specified by a single equation of the form.

$$Ax + By + Cz + D = 0 \tag{8.4}$$

Scanned by CamScanner

Notice that one of the constants (for example A, if it is not zero) may be divided out of the equation so that

$$x + B_1y + C_1z + D_1 = 0 \tag{8.5}$$

is an equation for the same plane when

$$B_1 = B / A, \quad C_1 = C / A, \quad \text{and} \quad D_1 = D / A \tag{8.6}$$

It therefore requires only three constants B_1, C_1, and D_1 to specify the plane. The equation for a particular plane may be determined if we know the coordinates of three points (not all in a line) which lie within it: (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3). We can determine the equation in the following manner. Since each point is in the plane, it must satisfy the plane's equation

$$\begin{aligned} x_1 + B_1y_1 + C_1z_1 + D_1 &= 0 \\ x_2 + B_1y_2 + C_1z_2 + D_1 &= 0 \\ x_3 + B_1y_3 + C_1z_3 + D_1 &= 0 \end{aligned} \tag{8.7}$$

We have three equations to solve for the three unknowns B_1, C_1, and D_1, and standard algebra techniques for solving simultaneous equations will yield their values.

Another normalization is

$$A_2 = A / d, \quad B_2 = B / d, \quad C_2 = C / d, \quad \text{and} \quad D_2 = D / d \tag{8.8}$$

where

$$d = (A^2 + B^2 + C^2)^{1/2} \tag{8.9}$$

The value of this selection is that the distance between a point (x, y, z) and the plane is given by

$$L = |A_2x + B_2y + C_2z + D_2| \tag{8.10}$$

The sign of the quantity in the absolute value bars indicates on which side of the plane the point lies. This is an extension of Equation 1.30 for lines.

Another way of specifying a plane is by a single point in the plane and the direction perpendicular to the plane. A vector perpendicular to a plane is called a normal vector. We can call $[N_x \quad N_y \quad N_z]$ the displacements for the normal vector, and (x_p, y_p, z_p) are the coordinates of a point in the plane. (See Figures 8-3 and 8-4.)

If (x, y, z) is to be an arbitrary point in the plane, then $[(x - x_p) \quad (y - y_p) \quad (z - z_p)]$ is a vector parallel to the plane. We can derive an equation for the plane by using the *vector dot product*. The dot product of two vectors is the sum of the products of their corresponding components. For example, if $A = [A_x \quad A_y \quad A_z]$ and $B = [B_x \quad B_y \quad B_z]$, then their dot product is

$$A \cdot B = A_xB_x + A_yB_y + A_zB_z \tag{8.11}$$

The result of the dot product is equal to the product of the lengths of the two vectors times the cosine of the angle between them. (See Figure 8-5.)

Scanned by CamScanner

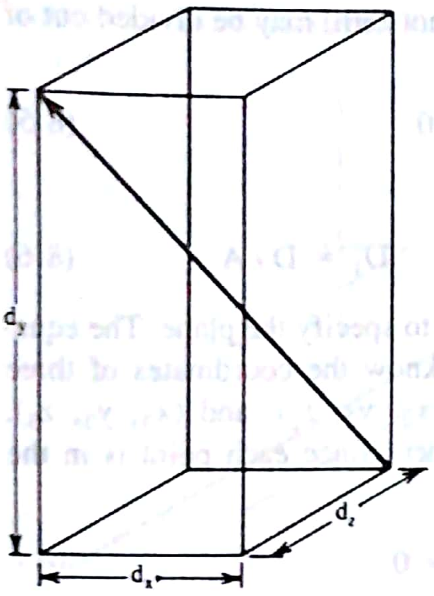


FIGURE 8-3
A vector in three dimensions.

$$A \cdot B = |A||B|\cos\theta \qquad (8.12)$$

Now the angle between any vector parallel to a plane and the normal vector to the plane is $\pi/2$ radians (because that is what we mean by a normal vector). Since the cosine of $\pi/2$ is 0, we know that the dot product of the normal vector with a vector parallel to the plane is 0. We can find a vector parallel to the plane by taking the difference of two points within the plane. Therefore, if

$$N_x(x - x_p) + N_y(y - y_p) + N_z(z - z_p) = 0 \qquad (8.13)$$

is a true equation then the vector formed by the difference between (x, y, z) and (x_p, y_p, z_p) must be parallel to the plane. And since (x_p, y_p, z_p) is in the plane and we can get to (x, y, z) by moving along a vector parallel to the plane, we know that (x, y, z) is also a point in the plane. (See Figure 8-6.) So Equation 8.13 is then an equation for the plane.

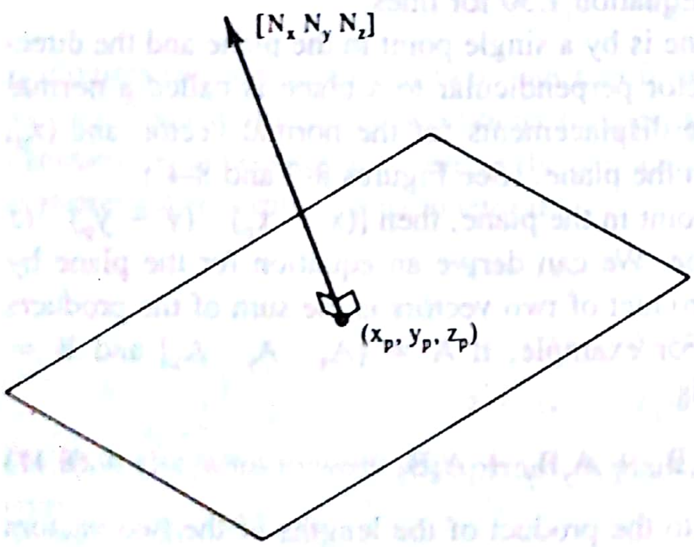


FIGURE 8-4
A vector normal to a plane.

Scanned by CamScanner

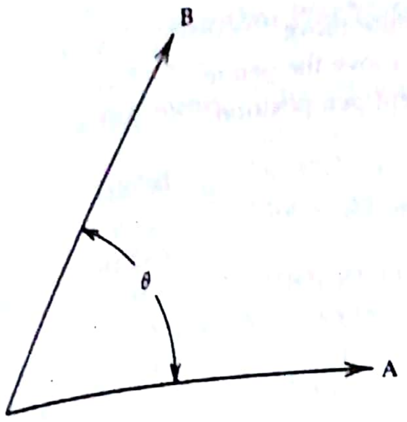


FIGURE 8-5
The angle between two vectors.

3D PRIMITIVES

Now that we know how to express points, lines, and planes in three dimensions, let us consider algorithms which allow the user to model three-dimensional objects. We have provided the user with commands for moving the "pen" and drawing lines and polygons in two dimensions. Extending these operations to three dimensions will mean requesting the user to provide three coordinate specifications instead of two. At some time in the display process we must project these three coordinates onto the two screen coordinates, but we shall place this burden on the DISPLAY-FILE-ENTER routine and postpone its discussion for the moment. As in the case of two dimensions, we picture an imaginary pen or pointer which the user commands to move from point to point. We save this pen's current position in some global variables, only now three variables are required: DF-PEN-X, DF-PEN-Y,and DF-PEN-Z, one for each of the three coordinates. The three-dimensional LINE and MOVE algorithms are as follows:

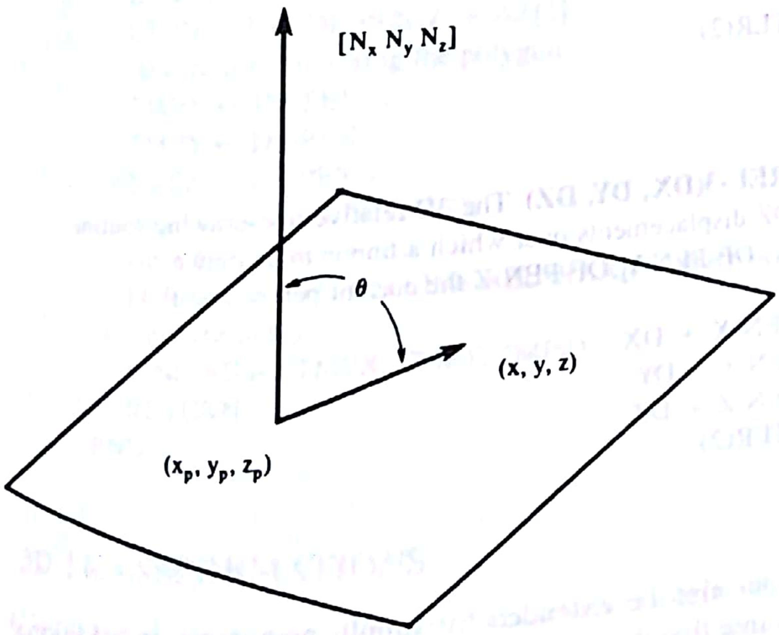


FIGURE 8-6
The angle between a vector normal to a plane and a vector parallel to the plane is 90 degrees.

8.1 Algorithm MOVE-ABS-3(X, Y, Z) The 3D absolute move

```
Arguments   X, Y, Z world coordinates of the point to move the pen to
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z current pen position
BEGIN
    DF-PEN-X ← X;
    DF-PEN-Y ← Y;
    DF-PEN-Z ← Z;
    DISPLAY-FILE-ENTER(1);
    RETURN;
END;
```

8.2 Algorithm MOVE-REL-3(DX, DY, DZ) The 3D relative move

```
Arguments   DX, DY, DZ changes to be made to the pen position
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
BEGIN
    DF-PEN-X ← DF-PEN-X + DX;
    DF-PEN-Y ← DF-PEN-Y + DX;
    DF-PEN-Z ← DF-PEN-Z + DZ;
    DISPLAY-FILE-ENTER(1);
    RETURN;
END;
```

8.3 Algorithm LINE-ABS-3(X, Y, Z) The 3D absolute line-drawing routine

```
Arguments   X, Y, Z coordinates of the point to draw the line to
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
BEGIN
    DF-PEN-X ← X;
    DF-PEN-Y ← Y;
    DF-PEN-Z ← Z;
    DISPLAY-FILE-ENTER(2);
    RETURN;
END;
```

8.4 Algorithm LINE-REL-3(DX, DY, DZ) The 3D relative line-drawing routine

```
Arguments   DX, DY, DZ displacements over which a line is to be drawn
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
BEGIN
    DF-PEN-X ← DF-PEN-X + DX;
    DF-PEN-Y ← DF-PEN-Y + DY;
    DF-PEN-Z ← DF-PEN-Z + DZ;
    DISPLAY-FILE-ENTER(2);
    RETURN;
END;
```

Polygon commands can also be extended by simply processing an additional array of coordinates. We assume that the user will provide coordinates which all lie in a plane, but a check could be placed in the algorithms to guarantee it. The following are the three-dimensional versions of our routines for entering polygons.

Scanned by CamScanner

8.5 Algorithm POLYGON-ABS-3(AX, AY, AZ, N) The 3D absolute polygon-drawing routine

```
Arguments   N the number of polygon sides
            AX, AY, AZ arrays of the coordinates of the vertices
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
Local       I for stepping through the polygon sides
BEGIN
   IF N < 3 THEN RETURN ERROR 'SIZE ERROR';
   DF-PEN-X ← AX[N];
   DF-PEN-Y ← AY[N];
   DF-PEN-Z ← AZ[N];
   DISPLAY-FILE-ENTER(N);
   FOR I = 1 TO N DO LINE-ABS-3(AX[I], AY[I], AZ[I]);
   RETURN;
END;
```

8.6 Algorithm POLYGON-REL-3(AX, AY, AZ, N) The 3D relative polygon-drawing routine

```
Arguments   N the number of polygon sides
            AX, AY, AZ arrays of displacements for the polygon sides
Global      DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
Local       I for stepping through the polygon sides
            TMPX, TMPY, TMPZ storage of point at which the polygon is closed
BEGIN
   IF N < 3 THEN RETURN ERROR 'SIZE ERROR';
   move to starting vertex
   DF-PEN-X ← DF-PEN-X + AX[1];
   DF-PEN-Y ← DF-PEN-Y + AY[1];
   DF-PEN-Z ← DF-PEN-Z + AZ[1];
   save vertex for closing the polygon
   TMPX ← DF-PEN-X;
   TMPY ← DF-PEN-Y;
   TMPZ ← DF-PEN-Z;
   DISPLAY-FILE-ENTER(N);
   enter the polygon sides
   FOR I = 2 TO N DO LINE-REL-3(AX[I], AY[I], AZ[I]);
   close the polygon
   LINE-ABS-3(TMPX, TMPY, TMPZ);
   RETURN;
END;
```

3D TRANSFORMATIONS

Using the above routines the user can construct a three-dimensional "drawing" of his object. But the actual display surface is two-dimensional. The two-dimensional image corresponds to a particular view of the three-dimensional object. The process of finding which points on the flat screen correspond to the lines and surfaces of the object involves a viewing transformation. We have seen how it is useful to translate, scale, and rotate images. We shall begin by considering the generalization of these transformations to three dimensions. We shall then extend our transformations to include parallel and perspective projections.

We saw that a two-dimensional scaling transformation matrix was of the form

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

or in homogeneous coordinates,

$$\begin{vmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Three dimensions gives us a third coordinate which can have its own scaling factor, so we shall require a 3 × 3 matrix rather than a 2 × 2.

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

If using homogeneous coordinates, a 4 × 4 matrix is required.

$$\begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.14}$$

Transformation of a point is done by multiplication by the matrix, just as in Chapter 4.

$$\begin{vmatrix} x_1 & y_1 & z_1 & w_1 \end{vmatrix} = \begin{vmatrix} x & y & z & w \end{vmatrix} \begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} s_x x & s_y y & s_z z & w \end{vmatrix} \tag{8.15}$$

We used the bottom row of the homogeneous coordinate matrix for translation values. This will still be the case for three dimensions, only the matrix is now 4 × 4 instead of 3 × 3. To translate by t_x in the x direction, t_y in the y direction, and t_z in the z direction, we multiply by the matrix

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{vmatrix} \tag{8.16}$$

Scanned by CamScanner

When we considered rotation of an object in two dimensions, we developed a matrix for rotation about the origin.

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

We can generalize this to a three-dimensional rotation about the z axis. If we rotate about the z axis, all z coordinates remain unchanged, while the x and y coordinates behave the same as in the two-dimensional case. (See Figure 8-7.)

A matrix which does this in homogeneous coordinates is

$$R_z = \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.17}$$

In the above rotation, we are thinking of the axes as fixed while some object in the space is moved. We could also think in terms of the object being fixed while the axes are moved. The difference between these two interpretations is the direction of rotation. Fixing the axes and rotating the object counterclockwise is the same as fixing the object and moving the axes clockwise.

Rotation about the x or y axis is done by a matrix of the same general form as in Equation 8.17, as we might expect from the symmetry of the axes. To rotate about the x axis so that y is turned into z, we use the homogeneous coordinate matrix

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.18}$$

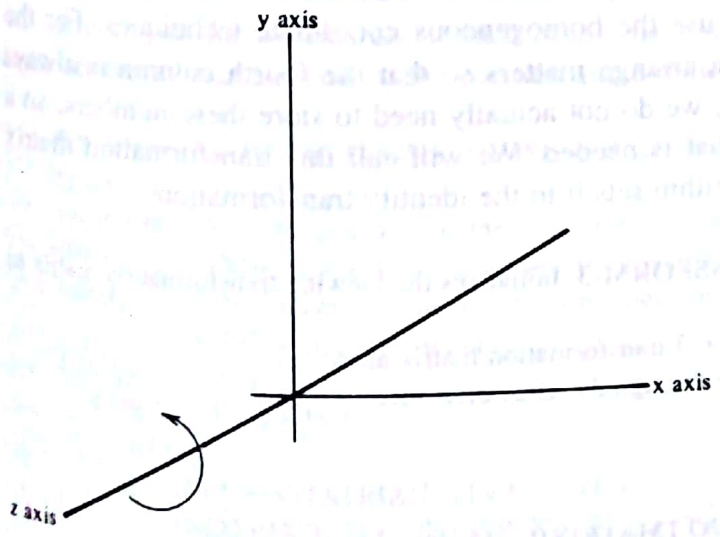


FIGURE 8-7
Rotation about the z axis.

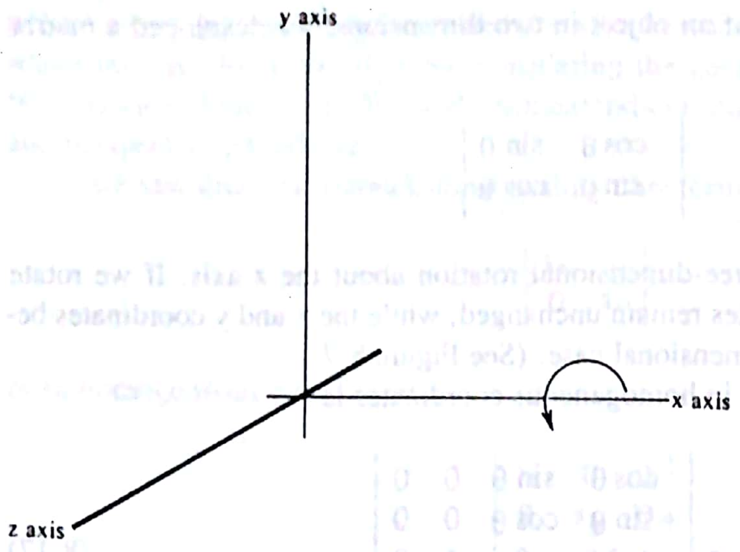


FIGURE 8-8
Rotation about the x axis.

(See Figure 8-8.) To rotate about the y axis so that z is turned into x, we use the homogeneous coordinate matrix

$$R_y = \begin{vmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad (8.19)$$

(See Figure 8-9.)

Three-dimensional transformations are useful in presenting different views of an object. An example of a program which lets the user interactively specify the transformation parameters for viewing a molecule is shown in Plate 3. Since we will find translation and rotation about an axis useful in creating a viewing transformation, we will develop algorithms for creating transformation matrices involving these operations. We shall do this by means of a routine which creates an identity transformation matrix (NEW-TRANSFORMATION-3) and routines which have the effect of multiplying the current transformation matrix by a matrix for translation or rotation about one of the principal axes. We will use the homogeneous coordinate techniques. For the viewing transformation, we can arrange matters so that the fourth column is always three 0s and a 1. Knowing this, we do not actually need to store these numbers, so a 4 × 3 array of storage is all that is needed. We will call this transformation matrix TMATRIX. The following algorithm sets it to the identity transformation.

8.7 Algorithm NEW-TRANSFORM-3 Initializes the viewing transformation matrix to the identity

```
Global    TMATRIX a 4 × 3 transformation matrix array
Local     I, J for stepping through the array elements
BEGIN
    FOR I = 1 TO 4 DO
        BEGIN
            FOR J = 1 TO 3 DO TMATRIX[I, J] ← 0;
```

Scanned by CamScanner

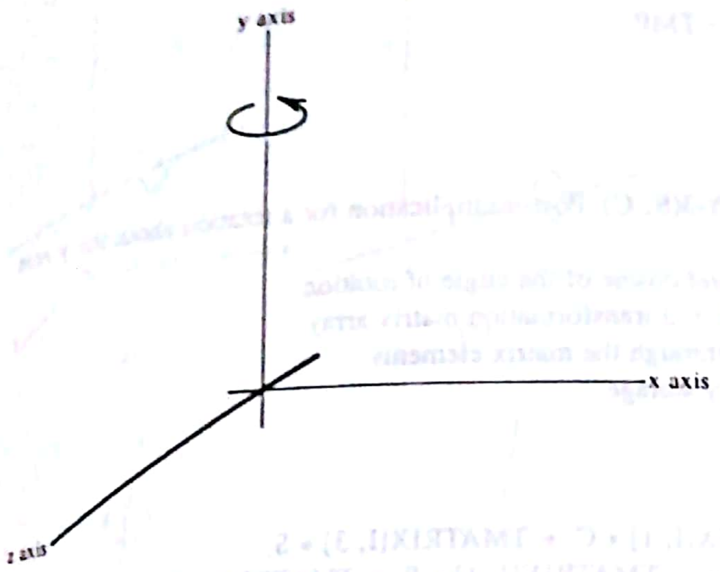


FIGURE 8-9
Rotation about the y axis.

```
            IF I ≠ 4 THEN TMATRIX[I, I] ← 1;
        END;
    RETURN;
END;
```

The next algorithm has the effect of multiplying the TMATRIX array by a translation matrix.

8.8 Algorithm TRANSLATE-3(TX, TY, TZ) Post-multiplies the viewing transformation matrix by a translation

Arguments	TX, TY, TZ the amount of the translation
Global	TMATRIX a 4 × 3 transformation matrix array

```
BEGIN
    TMATRIX[4, 1] ← TMATRIX[4, 1] + TX;
    TMATRIX[4, 2] ← TMATRIX[4, 2] + TY;
    TMATRIX[4, 3] ← TMATRIX[4, 3] + TZ;
    RETURN;
END;
```

Algorithms for rotating about each of the major axes are given below. The arguments are the sine and cosine of the rotation angle rather than the angle itself.

8.9 Algorithm ROTATE-X-3(S, C) Post-multiplication for a rotation about the x axis (y into z)

Arguments	S, C the sine and cosine of the rotation angle
Global	TMATRIX a 4 × 3 transformation matrix array
Local	I for stepping through the matrix elements
	TMP temporary storage

```
BEGIN
    FOR I = 1 TO 4 DO
        BEGIN
            TMP ← TMATRIX[I, 2] * C − TMATRIX[I, 3] * S;
            TMATRIX[I, 3] ← TMATRIX[I, 2] * S + TMATRIX[I, 3] * C;
```

Scanned by CamScanner

```
            TMATRIX[I, 2] ← TMP;
        END;
    RETURN;
END;
```

8.10 Algorithm ROTATE-Y-3(S, C) Post-multiplication for a rotation about the y axis (z into x)

Arguments	S, C the sine and cosine of the angle of rotation
Global	TMATRIX a 4 × 3 transformation matrix array
Local	I for stepping through the matrix elements
	TMP temporary storage

```
BEGIN
    FOR I = 1 TO 4 DO
        BEGIN
            TMP ← TMATRIX[I, 1] * C + TMATRIX[I, 3] * S;
            TMATRIX[I, 3] ← − TMATRIX[I, 1] * S + TMATRIX[I, 3] * C;
            TMATRIX[I, 1] ← TMP;
        END;
    RETURN;
END;
```

8.11 Algorithm ROTATE-Z-3(S, C) Post-multiplication for a rotation about the z axis (x into y)

Arguments	S, C the sine and cosine of the angle of rotation
Global	TMATRIX a 4×3 transformation matrix array
Local	I for stepping through the matrix elements
	TMP temporary storage

```
BEGIN
    FOR I = 1 TO 4 DO
        BEGIN
            TMP ← TMATRIX[I, 1] * C − TMATRIX[I, 2] * S;
            TMATRIX[I, 2] ← TMATRIX[I, 1] * S + TMATRIX[I, 2] * C;
            TMATRIX[I, 1] ← TMP;
        END;
    RETURN;
END;
```

ROTATION ABOUT AN ARBITRARY AXIS

Let us take a moment to become more familiar with three-dimensional transformations by solving a problem. While we have developed transformations for rotation about a coordinate axis, in general any line in space can serve as an axis for rotation. The problem is to derive a transformation matrix for a rotation of angle θ about an arbitrary line. We will build this transformation out of those which we have already encountered. We shall perform a translation to move the origin onto the line. We shall then make rotations about the x and y axes to align the z axis with the line. The rotation about the line then becomes a rotation about the z axis. Finally, we apply inverse transformations for the rotations about the y and x axes and for the translation to restore the line and coordinates to their original orientation. (See Figure 8-10.)

Scanned by CamScanner

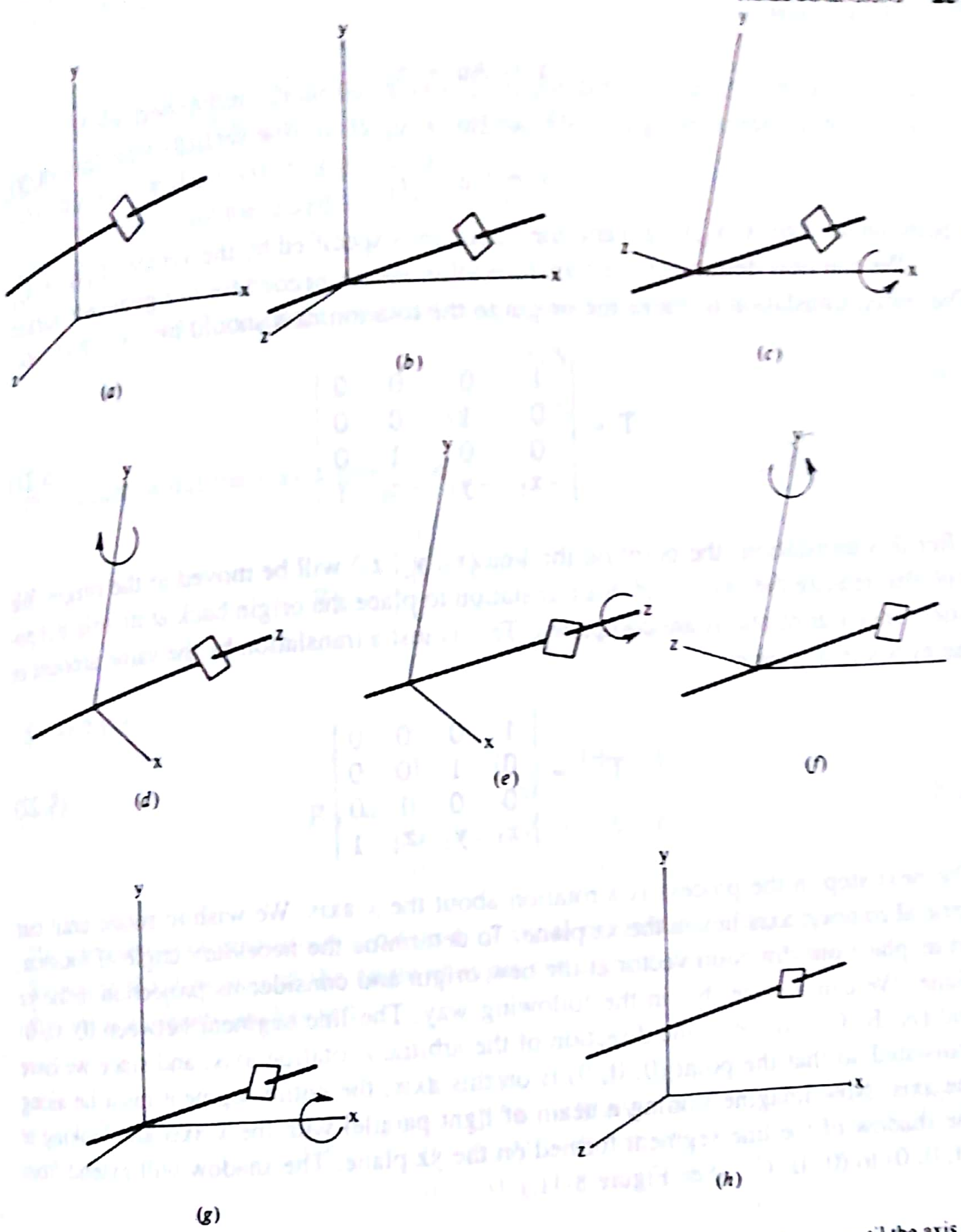


FIGURE 8-10
(*a*) To rotate about an arbitrary axis, (*b*) first translate the axis to the origin, (*c*) rotate about x until the axis of rotation is in the xz plane, (*d*) rotate about y until the z axis corresponds to the axis of rotation, (*e*) rotate about z (the axis of rotation) instead of fixing the object and rotating the axes (we have in this figure fixed the axes and rotated the object), (*f*) reverse the rotation about y, (*g*) reverse the rotation about x, and (*h*) reverse the translation.

We should decide upon a convenient representation for the line which is to be the axis of rotation. A point on the line, together with the line's direction, is sufficient to specify the line. This will prove to be a good form for our purposes. The point provides information for the translation, and the direction will tell us the correct angles of rotation for aligning the z axis. We can find this form from the parametric equations for the line in the following manner. Given the line

Scanned by CamScanner

$$x = Au + x_1$$
$$y = Bu + y_1 \qquad (8.20)$$
$$z = Cu + z_1$$

a point on the line is (x_1, y_1, z_1) and the direction is specified by the vector of [A B C].

We can now determine the transformation matrix needed for our general rotation. The initial translation to move the origin to the rotation axis should be

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 1 \end{vmatrix} \qquad (8.21)$$

After this translation, the point on the line (x_1, y_1, z_1) will be moved to the origin. We will also require the inverse of this translation to place the origin back at its original position once our rotations are completed. This is just a translation by the same amount in the opposite direction

$$T^{-1} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_1 & y_1 & z_1 & 1 \end{vmatrix} \qquad (8.22)$$

The next step in the process is a rotation about the x axis. We wish to rotate until our general rotation axis lies in the xz plane. To determine the necessary angle of rotation, let us place our direction vector at the new origin and consider its projection in the yz plane. We can picture this in the following way: The line segment between (0, 0, 0) and (A, B, C) will be in the direction of the arbitrary rotation axis, and since we have translated so that the point (0, 0, 0) is on this axis, the entire segment must lie along the axis. Now imagine shining a beam of light parallel with the x axis and looking at the shadow of the line segment formed on the yz plane. The shadow will extend from (0, 0, 0) to (0, B, C). (See Figure 8-11.)

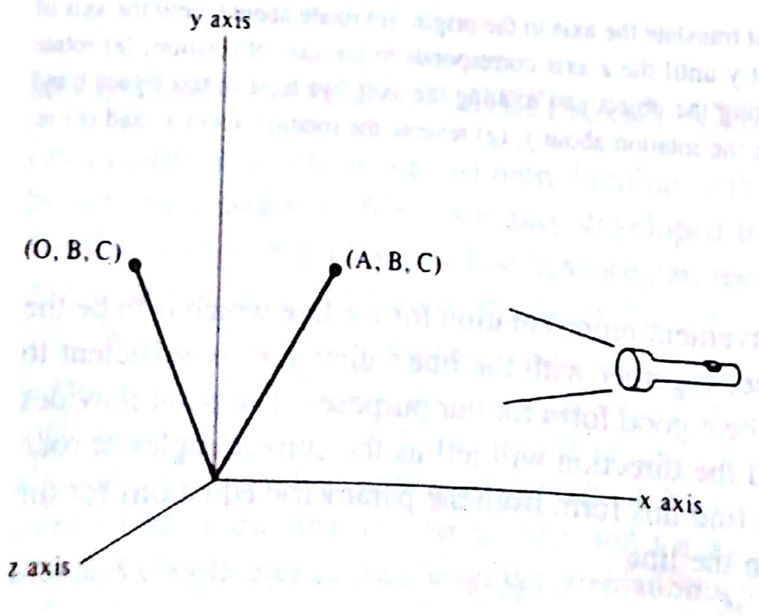


FIGURE 8-11
Projection of a line segment onto the yz plane.

Scanned by CamScanner

If we now rotate about the x axis until the arbitrary axis is in the xz plane, the line segment's shadow will lie along the z axis. How large an angle I is needed to place this shadow on the z axis? (See Figure 8-12.)

We know that the length of the shadow V will be

$$V = (B^2 + C^2)^{1/2} \tag{8.23}$$

and from the definition of the sine and cosine, we see that

$$\sin I = B/V$$
$$\cos I = C/V \tag{8.24}$$

The rotation about the x axis should be

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos I & \sin I & 0 \\ 0 & -\sin I & \cos I & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.25}$$

So we have

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & C/V & B/V & 0 \\ 0 & -B/V & C/V & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.26}$$

The inverse transformation is a rotation of equal magnitude in the opposite direction. Reversing the direction of the angle changes the sign of the sine elements, but leaves the cosine elements unchanged.

$$R_x^{-1} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & C/V & -B/V & 0 \\ 0 & B/V & C/V & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.27}$$

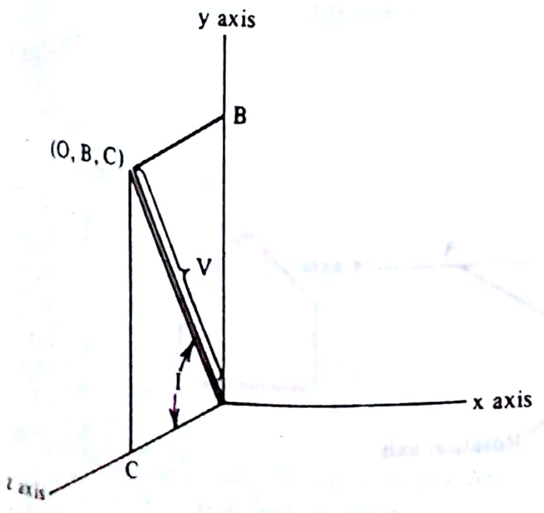


FIGURE 8-12
Parameters of the line segment projection.

Scanned by CamScanner

Now we can picture our rotation axis as lying in the xz plane. (See Figure 8-13.)

The rotation about the x axis has left the x coordinate unchanged. We also know that the overall length of the segment

$$L = (A^2 + B^2 + C^2)^{1/2} \tag{8.28}$$

is unchanged. The z coordinate will be

$$(L^2 - A^2)^{1/2} = (B^2 + C^2)^{1/2} = V \tag{8.29}$$

We wish to rotate by an angle J about the y axis so that the line segment aligns with the z axis.

$$\begin{aligned} \sin J &= A/L \\ \cos J &= V/L \end{aligned} \tag{8.30}$$

The rotation matrix will be

$$\begin{aligned} R_y &= \begin{vmatrix} \cos J & 0 & \sin J & 0 \\ 0 & 1 & 0 & 0 \\ -\sin J & 0 & \cos J & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \\ &= \begin{vmatrix} V/L & 0 & A/L & 0 \\ 0 & 1 & 0 & 0 \\ -A/L & 0 & V/L & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \end{aligned} \tag{8.31}$$

The signs are different in Equation 8.31 from those in Equation 8.19 because we are rotating from x into z instead of from z into x. The inverse for this transformation is

$$R_y^{-1} = \begin{vmatrix} V/L & 0 & -A/L & 0 \\ 0 & 1 & 0 & 0 \\ A/L & 0 & V/L & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.32}$$

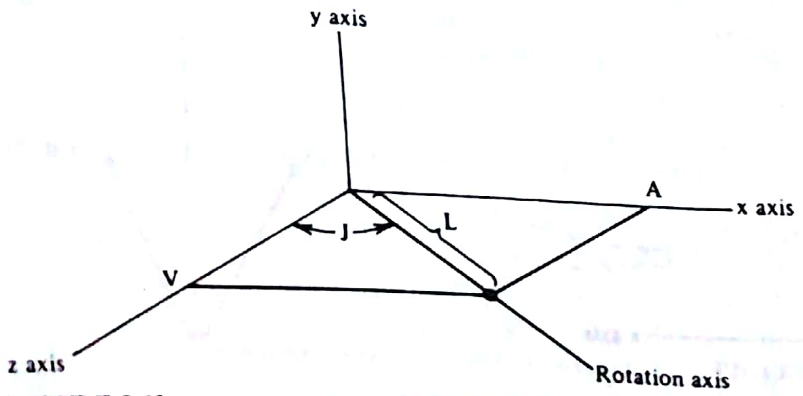


FIGURE 8-13
The rotation axis lying within the xz plane.

Scanned by CamScanner

Finally we are in a position to rotate by an angle θ about the arbitrary axis. Since we have aligned the arbitrary axis with the z axis, a rotation by θ about the z axis is needed.

$$R_z = \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.33}$$

The actual transformation for a rotation θ about an arbitrary axis is given by the product of the above transformations.

$$R_\theta = TR_xR_yR_zR_y^{-1}R_x^{-1}T^{-1} \tag{8.34}$$

PARALLEL PROJECTION

We have talked about creating and transforming three-dimensional objects, but since our viewing surface is only two-dimensional, we must have some way of *projecting* our three-dimensional object onto the two-dimensional screen. (See Figure 8-14.)

Perhaps the simplest way of doing this is just to discard the z coordinate. This is a special case of a method known as *parallel projection*. A parallel projection is formed by extending parallel lines from each vertex on the object until they intersect the plane

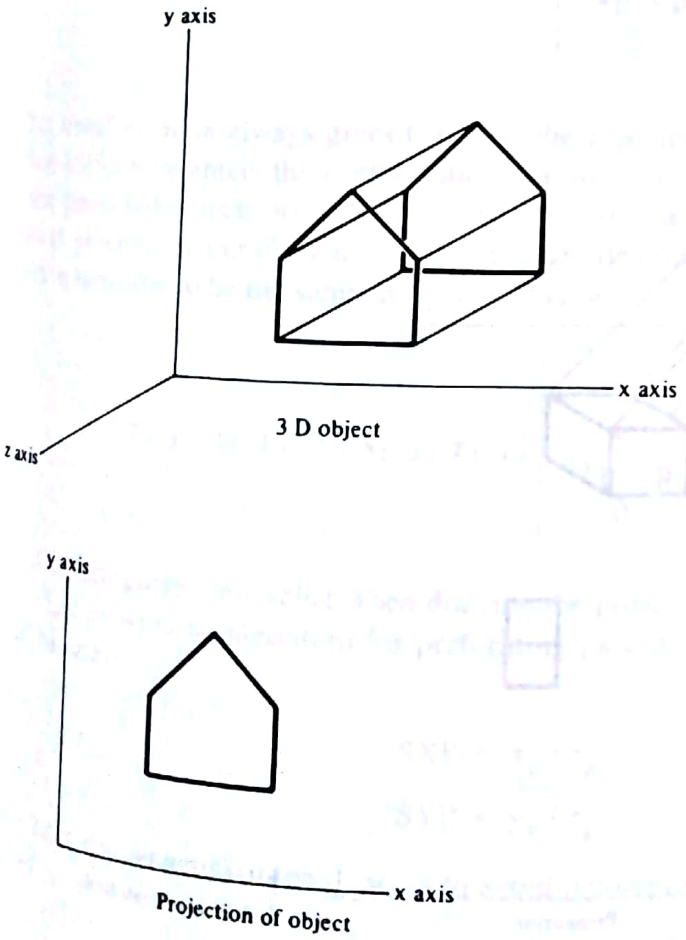


FIGURE 8-14
A three-dimensional object and its projection.

of the screen. The point of intersection is the projection of the vertex. We connect the projected vertices by line segments which correspond to connections on the original object. (See Figure 8-15.)

Our special case of discarding the z coordinate is the case where the screen, or viewing surface, is parallel to the xy plane, and the lines of projection are parallel to the z axis. As we move along these lines of projection, only the z coordinate changes; x and y values remain constant. So the point of intersection with the viewing surface has the same x and y coordinates as does the vertex on the object. The projected image is formed from the x and y coordinates, and the z value is discarded.

In a general parallel projection, we may select any direction for the lines of projection (so long as they do not run parallel to the plane of the viewing surface). Suppose that the direction of projection is given by the vector $[x_p \quad y_p \quad z_p]$ and that the image is to be projected onto the xy plane. If we have a point on the object at (x_1, y_1, z_1), we wish to determine where the projected point (x_2, y_2) will lie. Let us begin by writing the equations for a line passing through the point (x, y, z) and in the direction of projection. This is easy to do using the parametric form

$$\begin{aligned} x &= x_1 + x_p u \\ y &= y_1 + y_p u \\ z &= z_1 + z_p u \end{aligned} \tag{8.35}$$

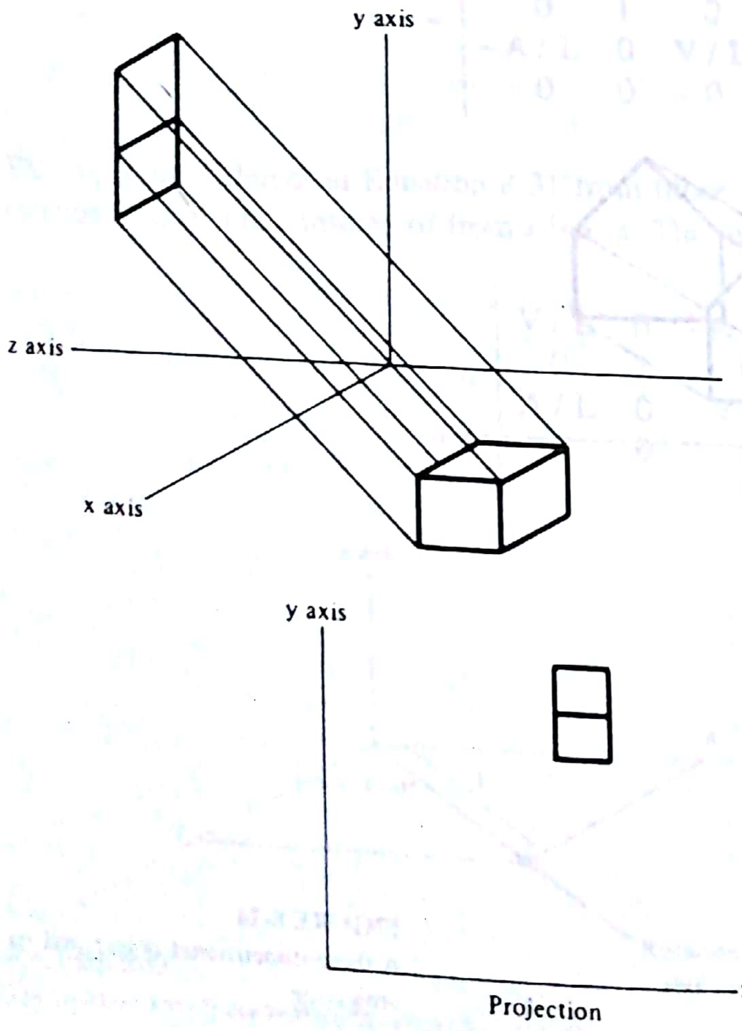


FIGURE 8-15
A parallel projection.

Scanned by CamScanner

Now we ask, where does this line intersect the xy plane? That is, what are the x and y values when z is 0? If z is 0, the third equation tells us that the parameter u is

$$u = -\frac{z_1}{z_p} \tag{8.36}$$

Substituting this into the first two equations gives

$$x_2 = x_1 - z_1 (x_p / z_p)$$
$$y_2 = y_1 - z_1 (y_p / z_p) \tag{8.37}$$

This projection formula is in fact a transformation which may be written in matrix form

$$[\,x_2\ y_2\,] = [\,x_1\ y_1\ z_1\,] \begin{vmatrix} 1 & 0 \\ 0 & 1 \\ -xp/zp & -yp/zp \end{vmatrix} \tag{8.38}$$

or in full homogeneous coordinates

$$[\,x_2\ y_2\ z_2\ 1\,] = [\,x_1\ y_1\ z_1\ 1\,] \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -xp/zp & -yp/zp & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.39}$$

This transformation always gives 0 for z_2, the z position of the view plane, but it is often useful to maintain the z information. We will find it useful in Chapter 9 when we order projected objects according to their z position as part of the hidden-surface removal process. A transformation that includes determining a z-coordinate value z_2 (which turns out to be the same as z_1) is as follows:

$$[\,x_2\ y_2\ z_2\ 1\,] = [\,x_1\ y_1\ z_1\ 1\,] \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -xp/zp & -yp/zp & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.40}$$

We will just ignore the z value when drawing the projected image.

Let us write an algorithm for performing parallel projections. We shall assume that the ratios

$$SXP = x_p / z_p \tag{8.41}$$
$$SYP = y_p / z_p$$

have been calculated and stored, so as to avoid recomputing them for every point projected.

8.12 Algorithm PARALLEL-TRANSFORM(X, Y, Z) Parallel projection of a point

```
Arguments   X, Y, Z the point to be projected, also for return of result
Global      SXP, SYP the parallel projection vector ratios
BEGIN
    X ← X − Z * SXP;
    Y ← Y − Z * SYP;
    RETURN;
END;
```

PERSPECTIVE PROJECTION

An alternative projection procedure is a *perspective projection*. In a perspective projection, the further away an object is from the viewer, the smaller it appears. This provides the viewer with a depth cue, an indication of which portions of the image correspond to parts of the object which are close or far away. In a perspective projection, the lines of projection are not parallel. Instead, they all converge at a single point called the *center of projection*. It is the intersections of these converging lines with the plane of the screen that determine the projected image. The projection gives the image which would be seen if the viewer's eye were located at the center of projection. The lines of projection would correspond to the paths of the light rays coming from the object to the eye. (See Figure 8-16.)

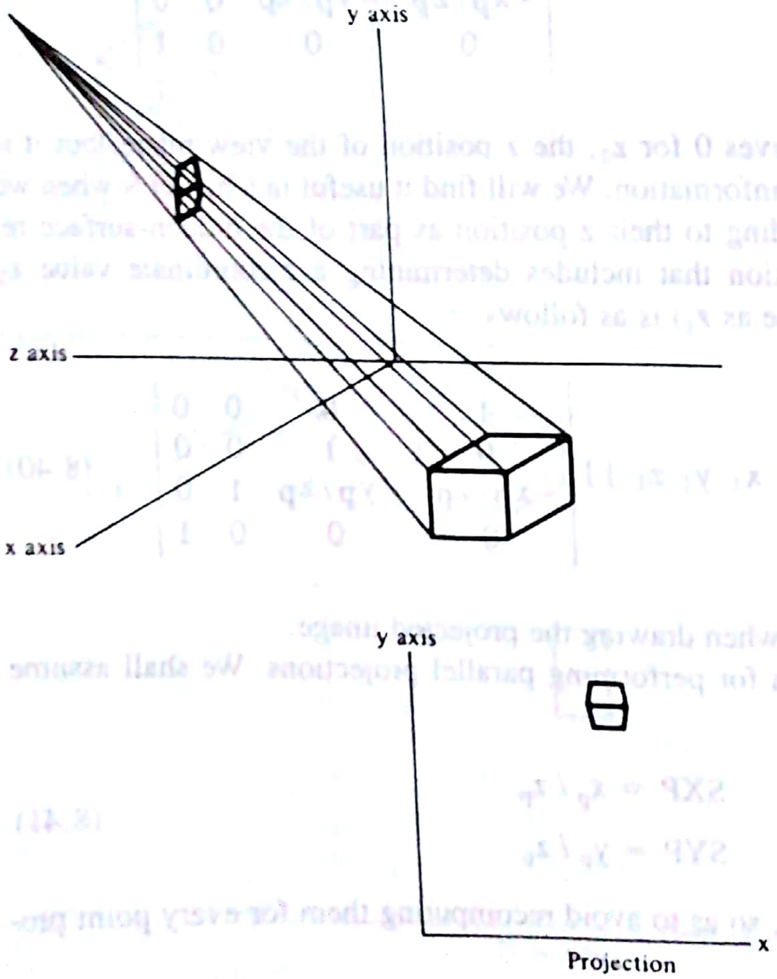


FIGURE 8-16
A perspective projection.

If the center of projection is at (x_c, y_c, z_c) and the point on the object is (x_1, y_1, z_1), then the projection ray will be the line containing these points and will be given by

$$x = x_c + (x_1 - x_c)u$$
$$y = y_c + (y_1 - y_c)u$$
$$z = z_c + (z_1 - z_c)u \tag{8.42}$$

The projected point (x_2, y_2) will be the point where this line intersects the xy plane. The third equation tells us that u, for this intersection point $(z = 0)$, is

$$u = -\frac{z_c}{z_1 - z_c} \tag{8.43}$$

Substituting into the first two equations gives

$$x_2 = x_c - z_c \frac{x_1 - x_c}{z_1 - z_c}$$

$$y_2 = y_c - z_c \frac{y_1 - y_c}{z_1 - z_c} \tag{8.44}$$

With a little algebra, we can rewrite this as

$$x_2 = \frac{x_c z_1 - x_1 z_c}{z_1 - z_c}$$

$$y_2 = \frac{y_c z_1 - y_1 z_c}{z_1 - z_c} \tag{8.45}$$

This projection can be put into the form of a transformation matrix if we take full advantage of the properties of homogeneous coordinates. The form of the matrix is

$$P = \begin{vmatrix} -z_c & 0 & 0 & 0 \\ 0 & -z_c & 0 & 0 \\ x_c & y_c & 0 & 1 \\ 0 & 0 & 0 & -z_c \end{vmatrix} \tag{8.46}$$

To show that this transformation works, consider the point (x_1, y_1, z_1). In homogeneous coordinates we would have

$$[x_1w_1 \quad y_1w_1 \quad z_1w_1 \quad w_1]$$

Multiplying by the transformation matrix gives

$$[\,x_2w_2 \ \ y_2w_2 \ \ z_2w_2 \ \ w_2\,] = [\,x_1w_1 \ \ y_1w_1 \ \ z_1w_1 \ \ w_1\,] \begin{vmatrix} -z_c & 0 & 0 & 0 \\ 0 & -z_c & 0 & 0 \\ x_c & y_c & 0 & 1 \\ 0 & 0 & 0 & -z_c \end{vmatrix}$$

$$= [-x_1w_1z_c + z_1w_1x_c \quad -y_1w_1z_c + z_1w_1y_c \quad 0 \quad z_1w_1 - z_cw_1\,] \tag{8.47}$$

Scanned by CamScanner

so

$$w_2 = z_1 w_1 - z_c w_1 \tag{8.48}$$

and

$$z_2 w_2 = 0 \tag{8.49}$$

which gives

$$z_2 = 0 \tag{8.50}$$

and

$$x_2 w_2 = -x_1 w_1 z_c + z_1 w_1 x_c \tag{8.51}$$

which gives

$$x_2 = \frac{x_c z_1 - x_1 z_c}{z_1 - z_c} \tag{8.52}$$

and

$$y_2 w_2 = -y_1 w_1 z_c + z_1 w_1 y_c \tag{8.53}$$

which gives

$$y_2 = \frac{y_c z_1 - y_1 z_c}{z_1 - z_c} \tag{8.54}$$

The resulting point (x_2, y_2) is then indeed the correctly projected point.

An equivalent form of the projection transformation is

$$P_1 = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -x_c/z_c & -y_c/z_c & 0 & -1/z_c \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.55}$$

The change is the factor of $-1/z_c$. Because we divide the first three coordinates (x_w, y_w, z_w) by w to obtain the actual position, changing all four coordinates by some common factor has no net effect. In the literature, the perspective transformation is often defined such that the center of projection is located at the origin and the view plane is positioned at $z = d$. For this situation, the transformation is given by

$$P_2 = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.56}$$

Scanned by CamScanner

We can show that the descriptions are equivalent as follows. To get the perspective transformation with the center of projection at (x_c, y_c, z_c), we can first translate this point to the origin, and then use the perspective transformation matrix P_2 given in Equation 8.55, where d is $-z_c$. We finally translate back to the original coordinates. The details of the proof are left as an exercise.

The above transformation maps points on an object to points on the view plane. We shall, however, find it useful to use a modified version of this transformation. We want a form which yields the same x and y values so that we can still display the image, but we would like to compute a z value different from zero. We would like to find z values such that we can preserve the depth relationship between objects, even after they are transformed. If object A lies in front of object B, we want the perspective transformation of object A to lie in front of the perspective transformation of object B. Furthermore, we want to compute z values such that after transforming the points on a straight line, we still have a straight line. We want the transformation to preserve planarity so that polygons are not warped into nonpolygons. The reason we want to do this is to allow us to establish depth order for hidden-surface removal (in Chapter 9) after having performed the perspective transformation. A form of the perspective transformation which meets these requirements is as follows:

$$P = \begin{vmatrix} -z_c & 0 & 0 & 0 \\ 0 & -z_c & 0 & 0 \\ x_c & y_c & -1 & 1 \\ 0 & 0 & 0 & -z_c \end{vmatrix} \tag{8.57}$$

This yields a transformed z value given by

$$z_2 = \frac{z_1}{z_1 - z_c} \tag{8.58}$$

We shall now present an algorithm for performing a perspective transformation of a point.

8.13 Algorithm PERSPECTIVE-TRANSFORM(X, Y, Z) For perspective projection of a point

Arguments	X, Y, Z the view plane coordinates of the point
Global	XC, YC, ZC the center of projection
Local	D the denominator in the calculations
Constant	ROUNDOFF some small number greater than any round-off error
	VERY-LARGE a very large number approximating infinity

```
BEGIN
    D ← ZC − Z
    IF |D| < ROUNDOFF THEN
        BEGIN
            X ← (X − XC) * VERY-LARGE
            Y ← (Y − YC) * VERY-LARGE
            Z ← VERY-LARGE
        END
```

Scanned by CamScanner

```
        ELSE
            BEGIN
                X ← (X * ZC − XC * Z) / D;
                Y ← (Y * ZC − YC * Z) / D;
                Z ← Z / D;
            END;
        RETURN;
    END;
```

VIEWING PARAMETERS

We have seen how parallel and perspective projections can be used to form a two-dimensional image from a three-dimensional object as seen from the front. But suppose we wish to view the object from the side, or the top, or even from behind. How can this be done? All that we need to do is to apply some rotation transformations before projecting. There are two equivalent ways of thinking about this. We can think of the *view plane* (that is, the plane of our display surface) as fixed and the object as rotated, or we can picture the object as fixed and the view plane as repositioned. (See Figure 8-17.)

In our system we shall use this second description. It is as if the view plane were the film in a camera. Every display-file segment represents a photograph taken by this camera. (See Figure 8-18.)

We can move the camera anywhere, so we can view the object from any angle. The picture taken by this synthetic camera is what is shown on the display surface, as if the film is developed and stuck upon the screen. The user is given routines by which he may change a number of viewing parameters. By setting the parameters, he can position the synthetic camera.(See Figure 8-19.)

The first parameters we shall consider are the coordinates of the *view reference point* (XR, YR, ZR). The view reference point is the center of attention. All other viewing parameters are expressed relative to this point. If we rotate the view, it will be a rotation about the view reference point (not about the origin). We can think of the view reference point as an anchor to which we have tied a string. The synthetic camera is attached to the other end of the string. By changing other viewing parameters, we can swing the camera through an arc or change the length of the string. One end of the string is always attached to the view reference point.

The direction of this imaginary string is given by the *view plane normal* vector [DXN DYN DZN]. This normal vector is the direction perpendicular to the view plane (the view plane is the film in the camera). This means that the camera always looks along the string toward the view reference point. The camera is pointed in the direction of the view plane normal.

The length of the string is given by the VIEW-DISTANCE parameter. This tells how far the camera is positioned from the view reference point. The view plane is positioned VIEW-DISTANCE away from the view reference point in the direction of the view plane normal.

We now have two coordinate systems. We have the object coordinates which we used to model our object, and we have *view plane coordinates*, which are attached to the

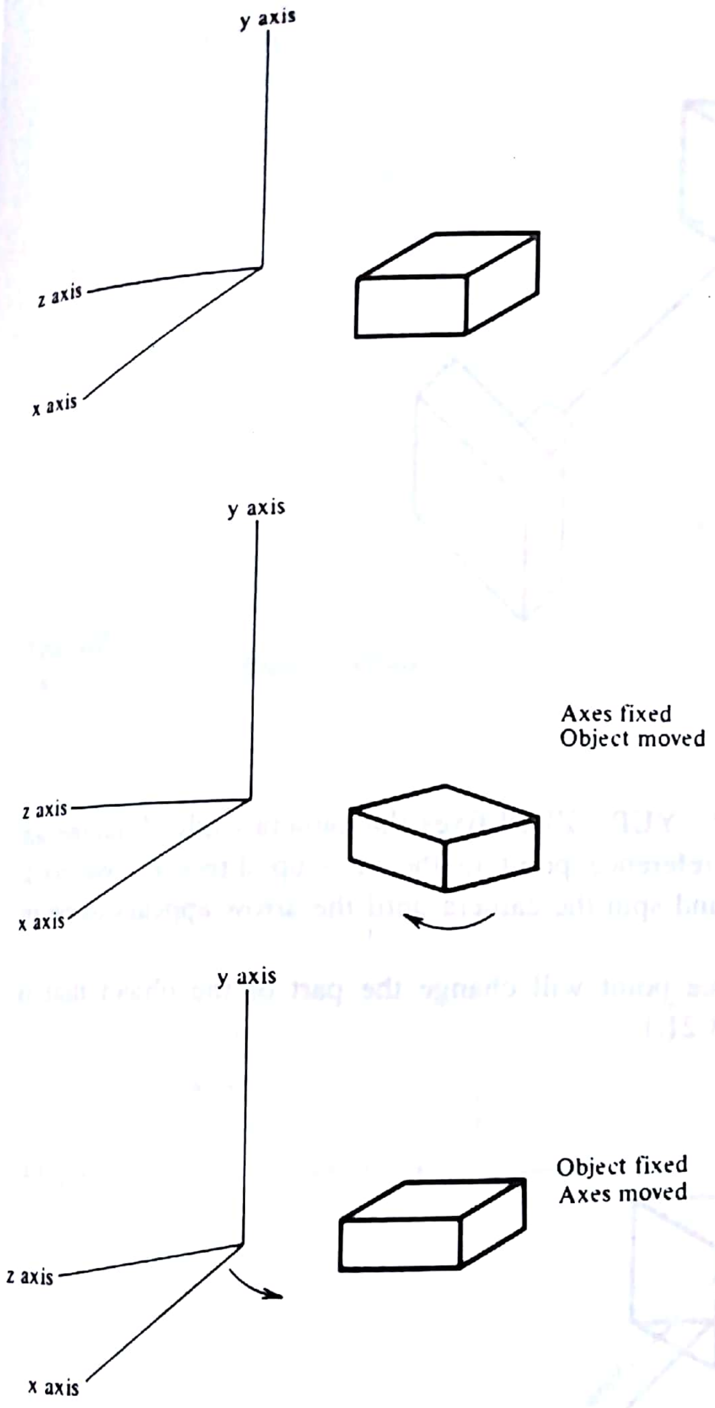


FIGURE 8-17
Rotating an object is equivalent to rotating the axes in the opposite direction.

view plane. Think of the *object coordinates* as being painted on the "floor" which supports the object, while the view plane coordinates have been printed on the film in the synthetic camera. We can place the origin of the view plane coordinates at the point where the string attaches to the film, that is, where a line parallel to the view plane normal, passing through the view reference point, intersects the view plane.

There is one more viewing parameter we need to discuss. Imagine holding the string fixed and spinning the camera on it so that the string serves as the axis of rotation. At each angle, the photograph will show the same scene, but rotated so that a different part of the object is up. (See Figure 8-20.)

Scanned by CamScanner

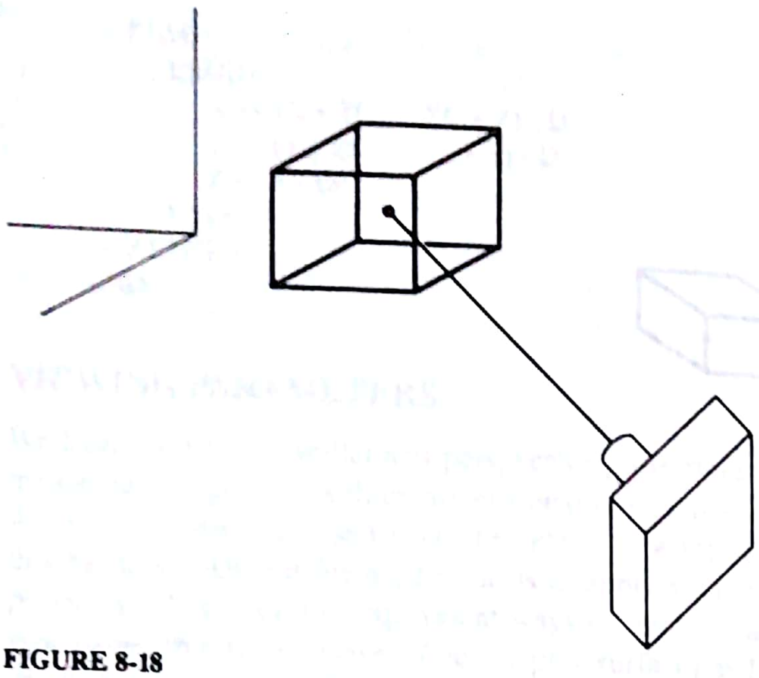


FIGURE 8-18
The synthetic camera analogy.

The *view-up* direction [XUP YUP ZUP] fixes the camera angle. Imagine an arrow extending from the view reference point in the view-up direction. We look through the camera's viewfinder and spin the camera until the arrow appears to be in the camera's ''up'' direction.

Changing the view reference point will change the part of the object that is shown at the origin. (See Figure 8-21.)

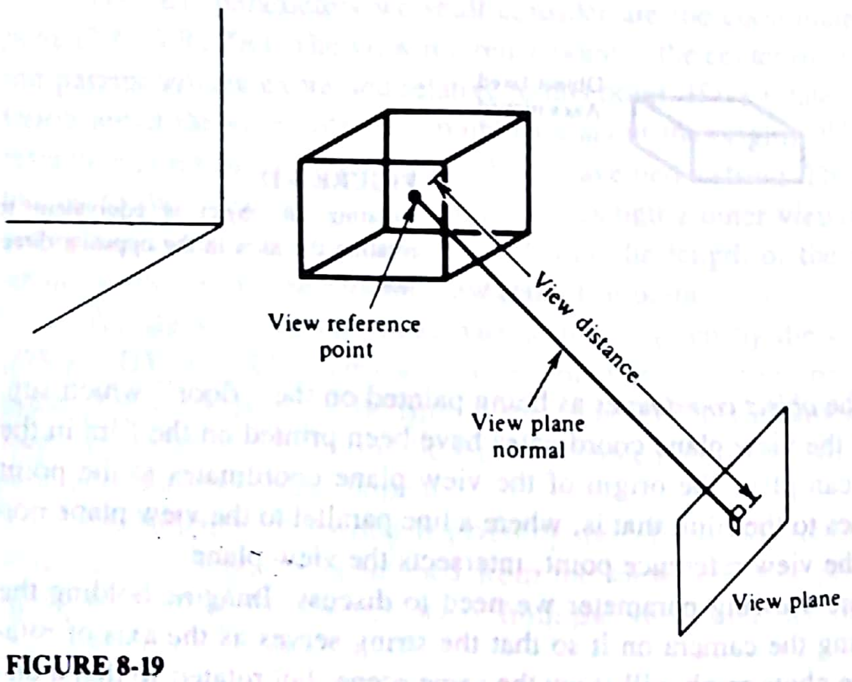


FIGURE 8-19
Three-dimensional viewing parameters.

Scanned by CamScanner

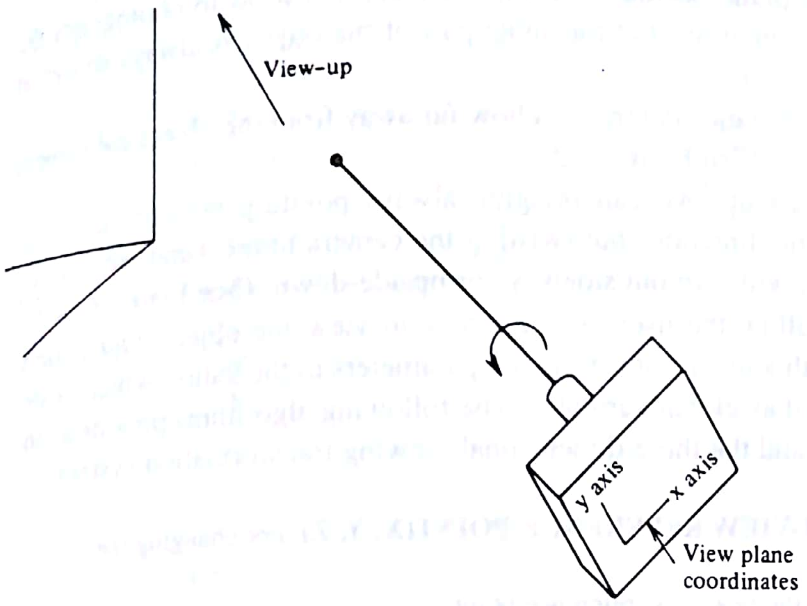


FIGURE 8-20
View-up and the view plane coordinates.

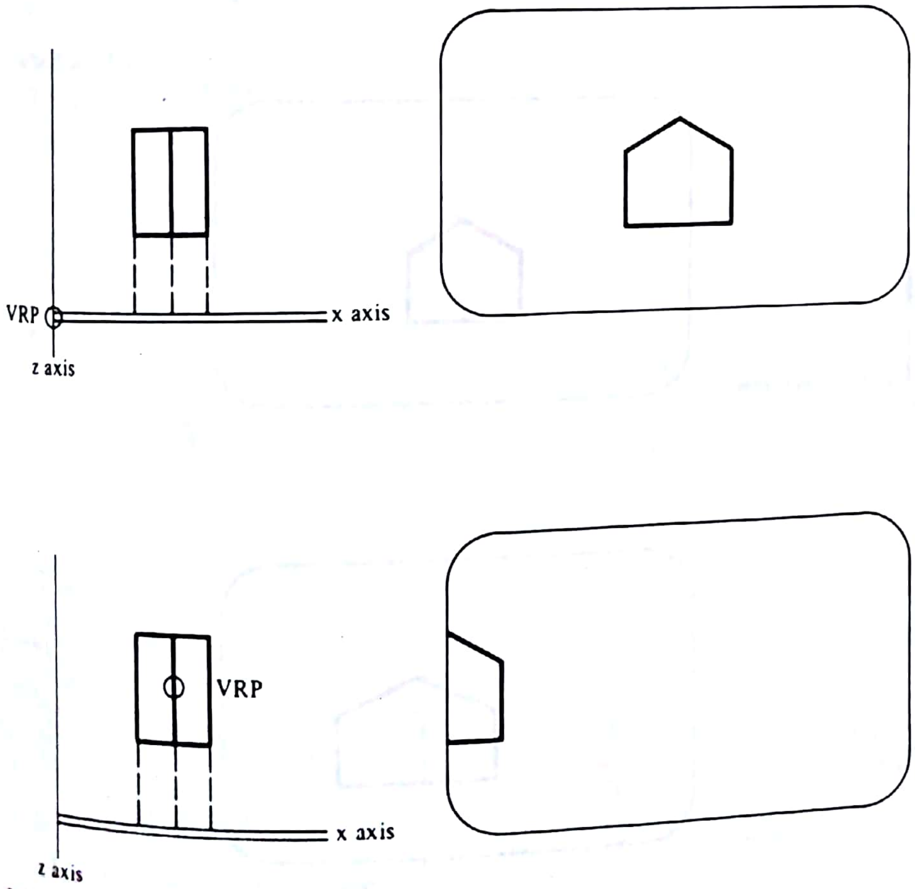


FIGURE 8-21
Changing the view reference point.

Scanned by CamScanner

Changing the view plane normal will move the camera so as to photograph the object from a different orientation, but the same part of the object is always shown at the origin. (See Figure 8-22.)

Changing the view distance determines how far away from the object the camera is when it takes the picture. (See Figure 8-23.)

When changing view-up, we can imagine always pointing the camera at the same object from the same direction, but twisting the camera in our hands so that the picture, when developed, will turn out sideways or upside-down. (See Figure 8-24.)

These parameters allow the user to select how to view the object. Our system must provide the user with a means of setting the parameters to the values which he desires. The values are saved as global variables. The following algorithms provide an interface between the user and the three-dimensional viewing transformation system.

8.14 Algorithm SET-VIEW-REFERENCE-POINT(X, Y, Z) For changing the view reference point

```
Arguments   X, Y, Z the new view reference point
Global      XR, YR, ZR permanent storage for the reference point
BEGIN
    XR ← X;
    YR ← Y;
    ZR ← Z;
    RETURN;
END;
```

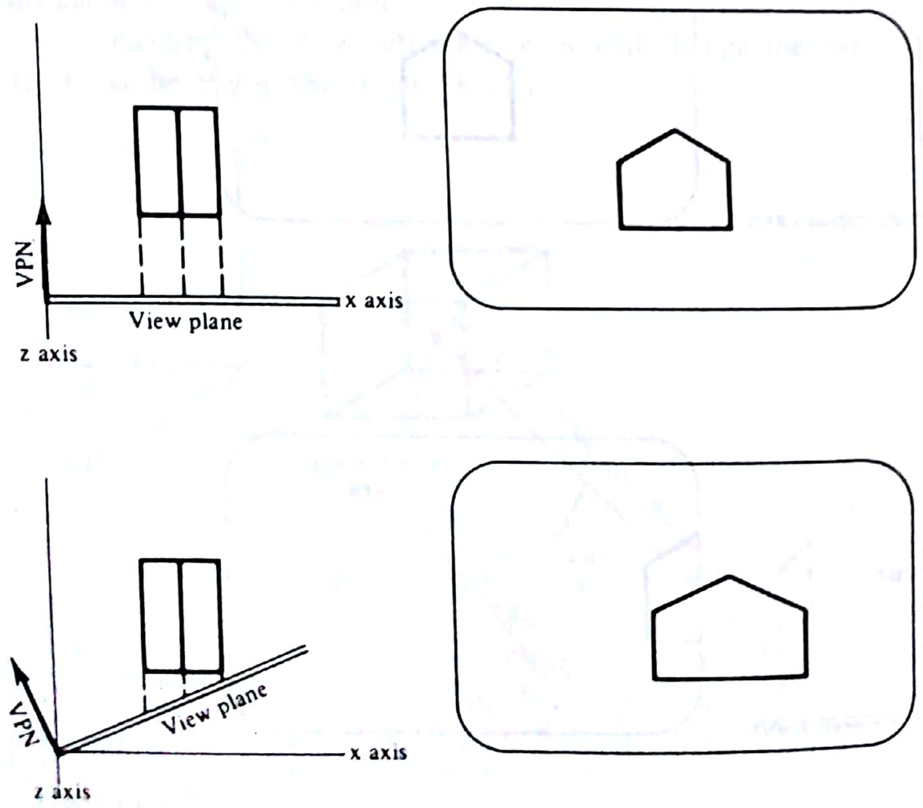
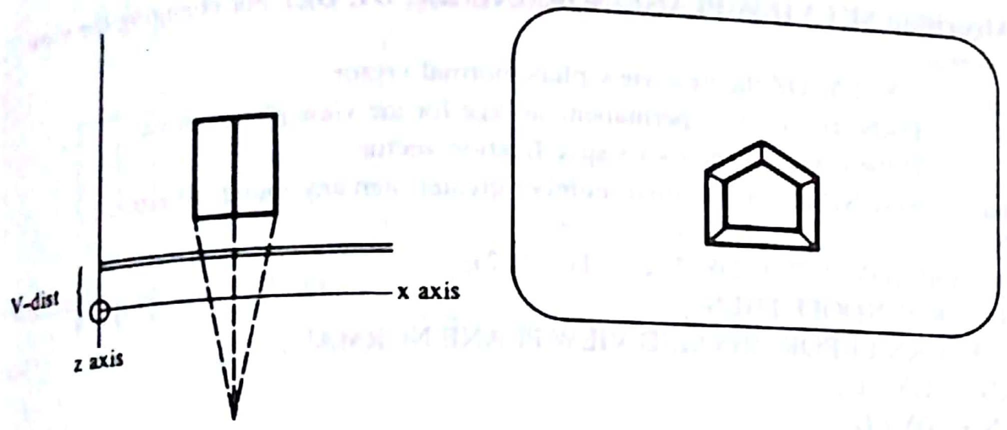


FIGURE 8-22
Changing the view plane normal.

Scanned by CamScanner



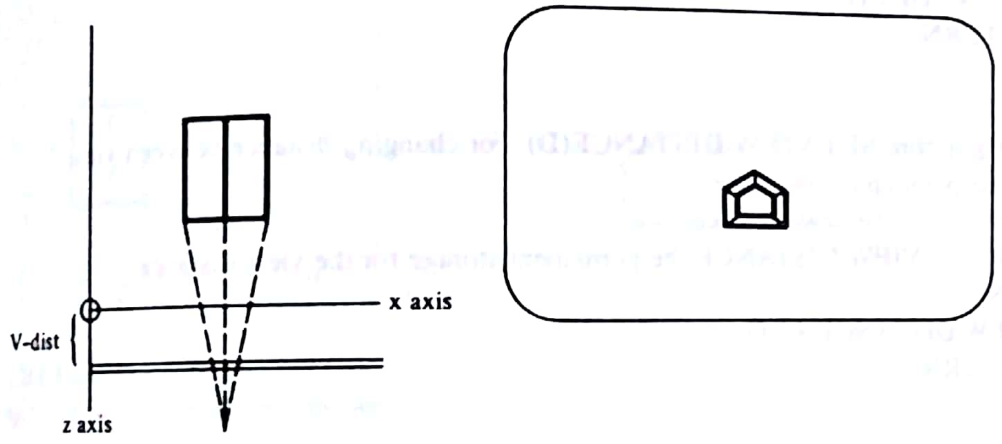
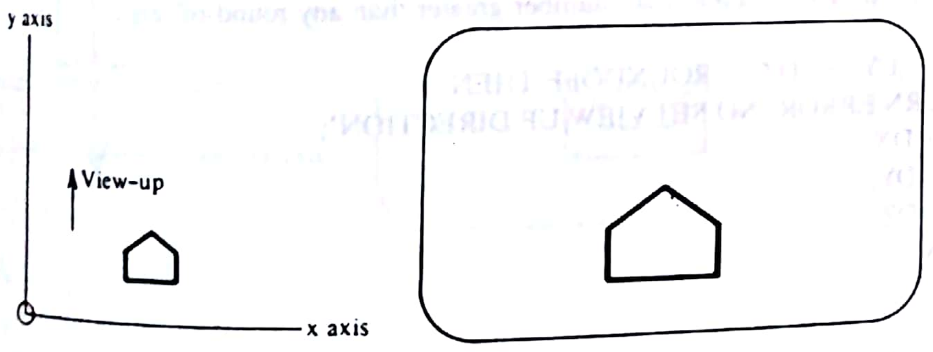


FIGURE 8-23
Changing the view distance.



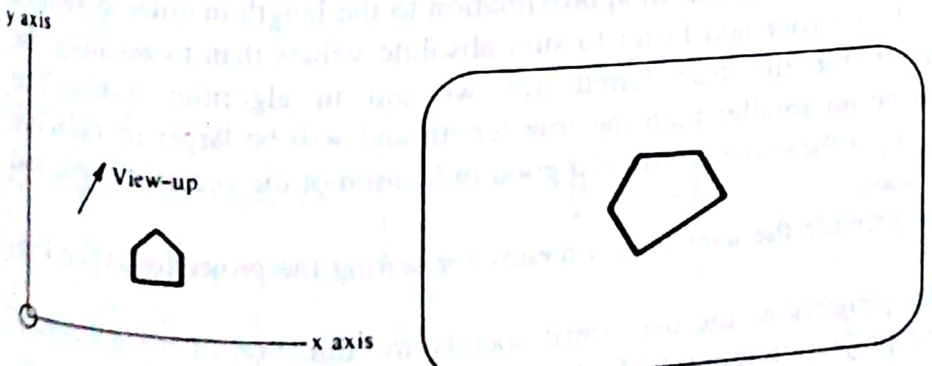


FIGURE 8-24
Changing the view-up direction.

8.15 Algorithm SET-VIEW-PLANE-NORMAL(DX, DY, DZ) For changing the view plane normal

```
Arguments   DX, DY, DZ the new view plane normal vector
Global      DXN, DYN, DZN permanent storage for the view plane normal
Local       D the length of the user's specification vector
Constant    ROUNDOFF some small number greater than any round-off error
BEGIN
   D ← SQRT(DX ↑ 2 + DY ↑ 2 + DZ ↑ 2);
   IF D < ROUNDOFF THEN
      RETURN ERROR 'INVALID VIEW PLANE NORMAL';
   DXN ← DX / D;
   DYN ← DY / D;
   DZN ← DZ / D;
   RETURN;
END;
```

8.16 Algorithm SET-VIEW-DISTANCE(D) For changing distance between view reference point and view plane

```
Argument    D the new distance
Global      VIEW-DISTANCE the permanent storage for the view distance
BEGIN
   VIEW-DISTANCE ← D;
   RETURN;
END;
```

8.17 Algorithm SET-VIEW-UP(DX, DY, DZ) For changing the direction which will be vertical on the image

```
Global      DXUP, DYUP, DZUP permanent storage for the view-up direction
Constant    ROUNDOFF some small number greater than any round-off error
BEGIN
   IF |DX| + |DY| + |DZ| < ROUNDOFF THEN
      RETURN ERROR 'NO SET-VIEW-UP DIRECTION';
   DXUP ← DX;
   DYUP ← DY;
   DZUP ← DZ;
   RETURN;
END;
```

In the above algorithm we use an approximation to the length in order to avoid a zero-length vector. It is easier and faster to sum absolute values than to calculate the square root required for the true length (as we did in algorithm 8.15). The approximation will be no smaller than the true length and will be larger by no more than a factor of $\sqrt{3}$. It therefore provides a good indication of the vector's length and will catch the zero cases.

We must also provide the user with a means for setting the projection. (See Figures 8-25 and 8-26.)

For a parallel projection, the user must specify the direction of the projection lines. A perspective projection requires the location of the center of a projection point. The values are saved in global variables. A flag is used to indicate whether a perspec-

Scanned by CamScanner

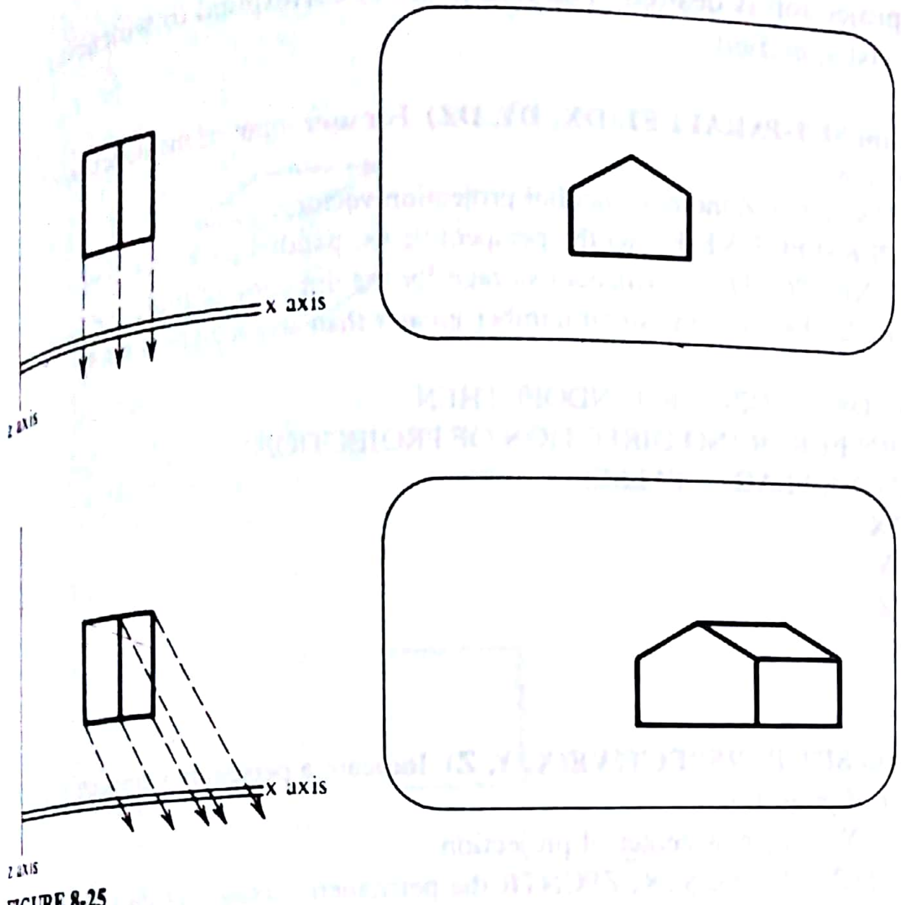


FIGURE 8-25
Changing the parallel projection direction.

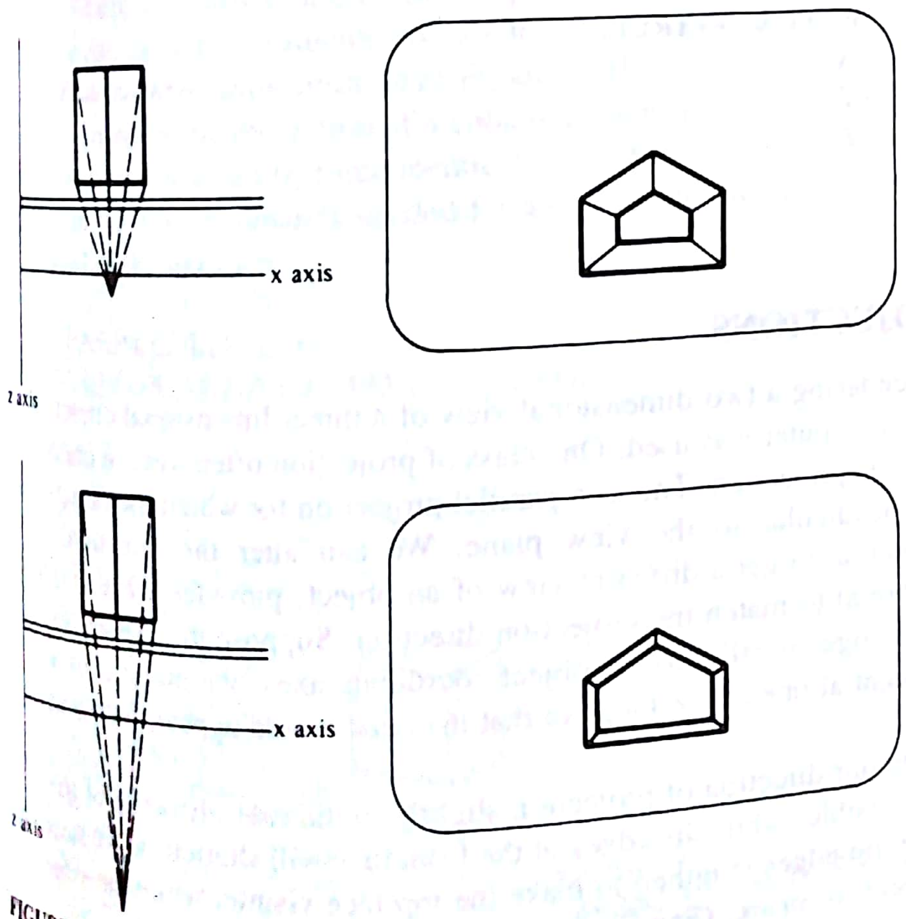


FIGURE 8-26
Changing the center of perspective projection.

Scanned by CamScanner

tive or a parallel projection is desired. The flag is set to correspond to whichever type of projection was last specified.

8.18 Algorithm SET-PARALLEL(DX, DY, DZ) For user input of the direction of parallel projection

```
Arguments   DX, DY, DZ the new parallel projection vector
Global      PERSPECTIVE-FLAG the perspective vs. parallel projection flag
            DXP, DYP, DZP permanent storage for the direction of projection
Constant    ROUNDOFF some small number greater than any round-off error
BEGIN
   IF |DX| + |DY| + |DZ| < ROUNDOFF THEN
      RETURN ERROR 'NO DIRECTION OF PROJECTION';
   PERSPECTIVE-FLAG ← FALSE;
   DXP ← DX;
   DYP ← DY;
   DZP ← DZ;
   RETURN;
END;
```

8.19 Algorithm SET-PERSPECTIVE(X, Y, Z) Indicate a perspective projection and save the center of projection

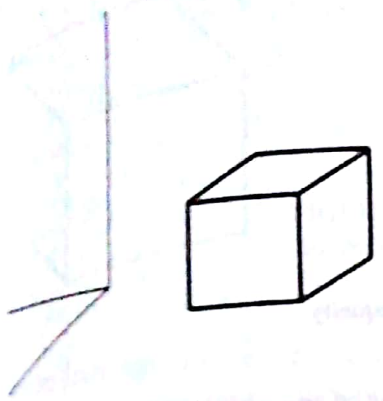
```
Arguments   X, Y, Z the new center of projection
Global      XPCNTR, YPCNTR, ZPCNTR the permanent storage for the center
            of projection
            PERSPECTIVE-FLAG perspective vs. parallel projection flag
BEGIN
   PERSPECTIVE-FLAG ← TRUE;
   XPCNTR ← X;
   YPCNTR ← Y;
   ZPCNTR ← Z;
   RETURN;
END;
```

SPECIAL PROJECTIONS

The problem of rendering a two-dimensional view of a three-dimensional object existed long before the computer was used. One class of projection often used in drafting is called an *axonometric* projection. This is a parallel projection for which the direction of projection is perpendicular to the view plane. We can alter the direction of an axonometric projection to get a different view of an object, provided we also change the view plane normal to match the projection direction. Suppose that we are looking at a cube that has edges parallel to the object coordinate axes. We might start with a view looking straight at one of the faces so that the cube would appear to be a square. (See Figure 8-27.)

If we change our direction of projection slightly to the side, then one of the side faces will become visible, while the edges of the front face will shorten. If we raise our angle of view, the top edges lengthen to make the top face visible, while the edges on the side faces appear to shrink. (See Figure 8-28.)

Scanned by CamScanner



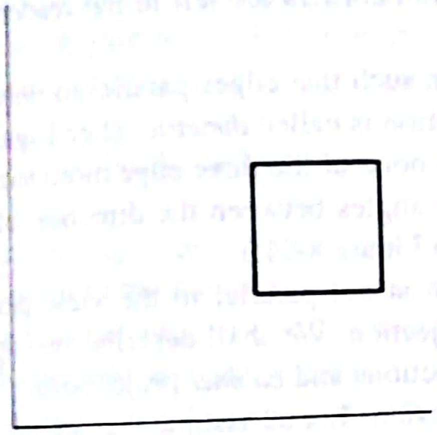


FIGURE 8-27
Looking straight at one of the faces of the cube.

There is a particular direction of projection for which all edges will appear shortened from their three-dimensional length by the same factor. This special direction is called an *isometric* projection. (See Figure 8-29.)

An isometric projection of a cube will show a corner of the cube in the middle of the image surrounded by three identical faces. From the symmetry of the situation, we can see that the commands needed for an isometric projection of an object with sides parallel to the axes are:

```
SET-PARALLEL(1, 1, 1);
SET-VIEW-PLANE-NORMAL(-1, -1, -1);
```

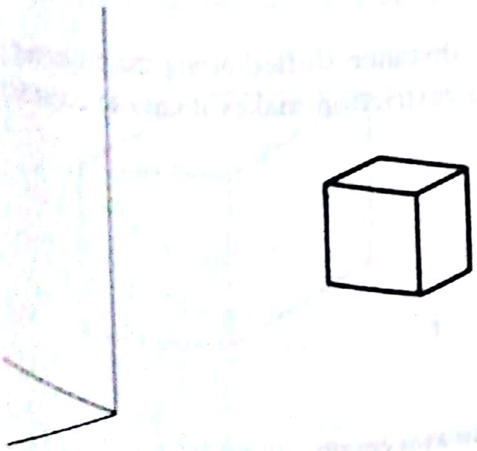


FIGURE 8-28
Changing the point of view shortens some edges and lengthens others.

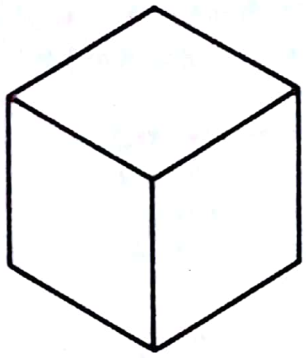


FIGURE 8-29
Isometric projection shortens all axes equally.

This projection will focus on the upper right-front corner of the object. The appropriate commands for projections of the other seven corners are left to the reader as an exercise.

If a viewing transformation is chosen such that edges parallel to only two of the axes are equally shortened, then the projection is called *dimetric*. (See Figure 8-30.)

A *trimetric* projection is one in which none of the three edge directions is equally shortened. There are no symmetries in the angles between the direction of projection and the directions of the object edges. (See Figure 8-31.)

If the direction of parallel projection is not parallel to the view plane normal, then we have what is called an *oblique* projection. We shall describe two special types of oblique projections, called *cavalier* projections and *cabinet* projections.

Let us first discuss the cavalier projection. Let us assume that we are viewing an object with edges parallel to the coordinate axes. The view plane will be parallel to the front face (parallel to the xy plane). For a cavalier projection, the direction of projection is slanted so that points with positive z coordinates will be projected down and to the left on the view plane. Points with negative z coordinates will be projected up and to the right. The angle of the projected z axis (the ratio of up to right) can be whatever we desire, but the distance the point shifts in the projected z direction must equal the actual three-dimensional z distance from the view plane. The projection command which will create a cavalier projection at angle A is

SET-PARALLEL(COS(A), SIN(A), 1).

Actually, any SET-PARALLEL command with arguments in the same ratio as those above will work as well. (See Figure 8-32.)

We stated that for a cavalier projection, the distance shifted along the projected z axis was equal to the actual z-axis distance. This restriction makes it easy to construct

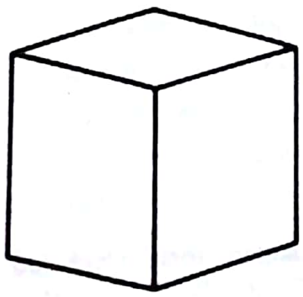


FIGURE 8-30
Dimetric projection shortens two axes equally.

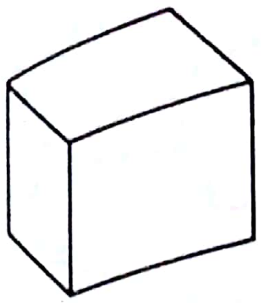


FIGURE 8-31
Trimetric projection shortens all axes differently.

these drawings. However, the result is an object which appears elongated along the z direction. An alternative which is still easy to construct with a scale and triangle is to shift only half the actual z distance along the projected z axis. This is called a cabinet projection. (See Figure 8-33.) It can be created by a SET-PARALLEL call with arguments in the ratio of

SET-PARALLEL(COS(A), SIN(A), 2).

Perspective projections can be classified as one-point, two-point, or three-point. These names refer to the number of vanishing points required for a construction of the drawing. Our routines are sufficiently general to generate all three types of perspective projection. A *one-point perspective* projection occurs when one of the faces of a rectangular object is parallel to the view plane. (See Figure 8-34.)

A *two-point perspective* projection refers to the situation where one set of edges runs parallel to the view plane, but none of the faces is parallel to it. (See Figure 8-35.)

A *three-point perspective* projection is the case where none of the edges is parallel to the view plane. (See Figure 8-36.)

CONVERSION TO VIEW PLANE COORDINATES

The user enters the description of the object in terms of the object coordinates. But our particular point of view corresponds to expressing the object in the view plane coordinates. The process of generating a particular view of the object is one of transforming from one coordinate system to another. While this problem may appear complex, it is actually one which we have already solved. The steps to be taken are the same as those

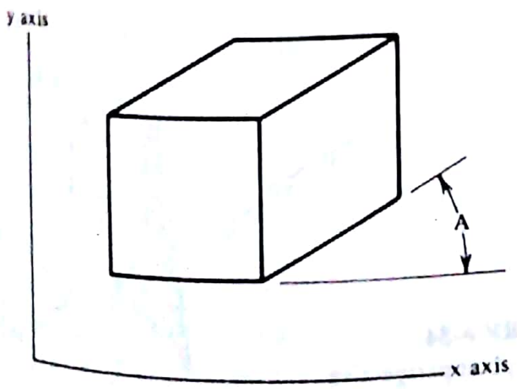


FIGURE 8-32
A cavalier projection.

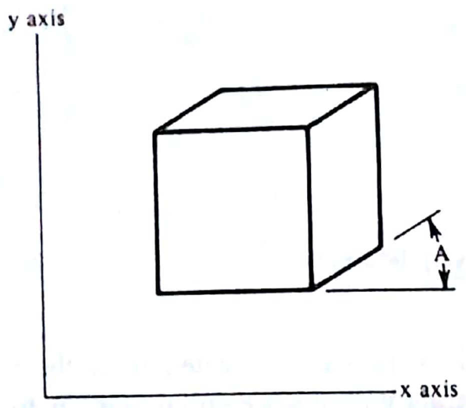
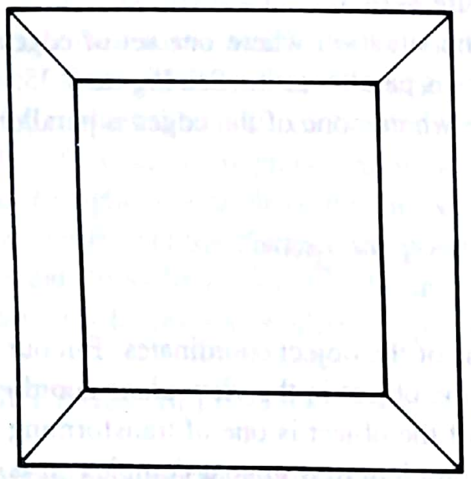


FIGURE 8-33
A cabinet projection.

which were used for a rotation about an arbitrary axis. We wish to perform a series of transformations which will change the object coordinates into the view plane coordinates. The first step is a translation to move the origin to the correct position for the view plane coordinate system. This is a shift first to the view reference point, and then along the view plane normal by the VIEW-DISTANCE. After the origin is in place, we align the z axis. In the rotation problem, we saw how to rotate a line onto the z axis.



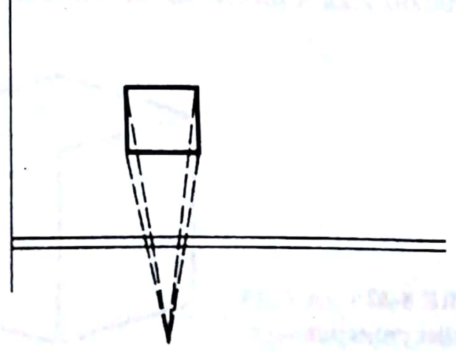


FIGURE 8-34
A one-point perspective.

Scanned by CamScanner

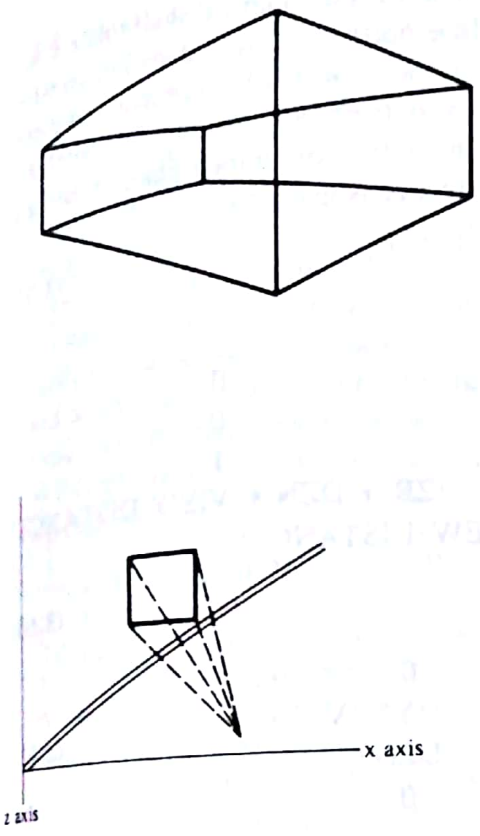


FIGURE 8-35
A two-point perspective.

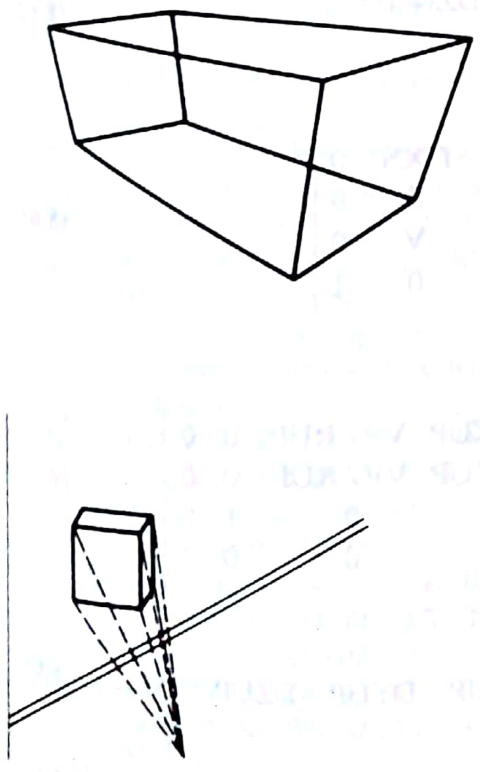


FIGURE 8-36
A three-point perspective.

Scanned by CamScanner

Here, we wish that line to be the object coordinate's z axis, and we shall rotate it into the view plane coordinate's z axis (the view plane normal). This is done in two steps. First, a rotation about the x axis places the line in the view plane coordinate's xz plane. Then a rotation about the y axis moves the z axis to its proper position. Now all that is needed is to rotate about the z axis until the x and y axes are in their place in the view plane coordinates. The entire transformation sequence is given by

$$\mathrm{TMATRIX} = \mathrm{T}R_xR_yR_z \tag{8.59}$$

where

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -(\mathrm{XR} + \mathrm{DXN} * \text{VIEW-DISTANCE}) & -(\mathrm{YR} + \mathrm{DYN} * \text{VIEW-DISTANCE}) & -(\mathrm{ZR} + \mathrm{DZN} * \text{VIEW-DISTANCE}) & 1 \end{vmatrix} \tag{8.60}$$

$$R_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & -\mathrm{DZN}/\mathrm{V} & -\mathrm{DYN}/\mathrm{V} & 0 \\ 0 & \mathrm{DYN}/\mathrm{V} & -\mathrm{DZN}/\mathrm{V} & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.61}$$

and

$$\mathrm{V} = (\mathrm{DYN}^2 + \mathrm{DZN}^2)^{1/2} \tag{8.62}$$

also

$$R_y = \begin{vmatrix} \mathrm{V} & 0 & -\mathrm{DXN} & 0 \\ 0 & 1 & 0 & 0 \\ \mathrm{DXN} & 0 & \mathrm{V} & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.63}$$

and

$$R_z = \begin{vmatrix} \text{YUP-VP}/\mathrm{RUP} & \text{XUP-VP}/\mathrm{RUP} & 0 & 0 \\ -\text{XUP-VP}/\mathrm{RUP} & \text{YUP-VP}/\mathrm{RUP} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{8.64}$$

where

$$[\text{XUP-VP} \quad \text{YUP-VP} \quad \mathrm{Z} \quad 1] = [\mathrm{DXUP} \quad \mathrm{DYUP} \quad \mathrm{DZUP} \quad 1]\, R_xR_y \tag{8.65}$$

and

$$\mathrm{RUP} = (\text{XUP-VP}^2 + \text{YUP-VP}^2)^{1/2} \tag{8.66}$$

Scanned by CamScanner

What are these XUP-VP and YUP-VP variables? We are trying to finish up the transformation process by rotating the x and y axes into position. The correct position is obtained when the y axis is aligned with the projection on the view plane of the view-up vector. But the view-up vector was specified in the object coordinates. It is much easier to work in the (as yet incomplete) view plane coordinate system. In the view plane system, we can project the view-up vector onto the view plane by just ignoring the z coordinate. The x and y values will be in the ratio of the sine and cosine of the angle needed to correctly align the up direction. So what we are doing is performing our partial transformation (R_xR_y) on the view-up direction. The translation part is not needed because we are working on a vector (which has no position, only magnitude and direction). The z coordinate is ignored to project onto the view plane, and the distance from the origin to the projected point is calculated. Dividing the x and y values by this distance yields the sine and cosine of the needed rotation angle. The following algorithm will create the view plane coordinate transformation.

8.20 Algorithm MAKE-VIEW-PLANE-TRANSFORMATION For making the viewing transformation

```
Global    XR, YR, ZR the view reference point
          DXN, DYN, DZN the view plane normal
          DXUP, DYUP, DZUP the view-up direction
          TMATRIX a 4 × 3 transformation matrix array
          PERSPECTIVE-FLAG the perspective projection flag
          VIEW-DISTANCE distance between view reference point and view plane
Local     V, XUP-VP, YUP-VP, RUP for storage of partial results
Constant  ROUNDOFF some small number greater than any round-off error
BEGIN
   start with the identity matrix
   NEW-TRANSFORM-3;
   translate so that view plane center is new origin
   TRANSLATE-3(-(XR + DXN * VIEW-DISTANCE),
      -(YR + DYN * VIEW-DISTANCE), -(ZR + DZN * VIEW-DISTANCE));
   rotate so that view plane normal is z axis
   V ← SQRT(DYN ↑ 2 + DZN ↑ 2);
   IF V > ROUNDOFF THEN ROTATE-X-3(- DYN / V, - DZN / V);
   ROTATE-Y-3( DXN, V);
   determine the view-up direction in these new coordinates
   XUP-VP ← DXUP * TMATRIX[1, 1] + DYUP * TMATRIX[2, 1] +
      DZUP * TMATRIX[3, 1];
   YUP-VP ← DXUP * TMATRIX[1, 2] + DYUP * TMATRIX[2, 2] +
      DZUP * TMATRIX[3, 2];
   determine rotation needed to make view-up vertical
   RUP ← SQRT(XUP-VP ↑ 2 + YUP-VP ↑ 2);
   IF RUP < ROUNDOFF THEN
      RETURN ERROR 'SET-VIEW-UP ALONG VIEW PLANE NORMAL';
   ROTATE-Z-3(XUP-VP / RUP, YUP-VP / RUP);
   IF PERSPECTIVE-FLAG THEN MAKE-PERSPECTIVE-TRANSFORMATION
   ELSE MAKE-PARALLEL-TRANSFORMATION;
   RETURN;
END;
```

Scanned by CamScanner

The above algorithm also converts the parallel or perspective projection parameters from object coordinates to view plane coordinates. This is done by calling either the MAKE-PARALLEL-TRANSFORMATION or the MAKE-PERSPECTIVE-TRANSFORMATION routine. The MAKE-PERSPECTIVE-TRANSFORMATION routine just applies the transformation matrix to the center of the projection point. An error occurs if the center of projection is on the wrong side of the view plane, that is, if the observer is on the same side of the screen as the object. The MAKE-PARALLEL-TRANSFORMATION routine converts the direction of parallel projection to the view plane coordinates. Since the direction is specified by a vector (no position), the translation portion of the transformation is omitted. An error occurs if the direction of projection turns out to be parallel to the view plane.

8.21 Algorithm MAKE-PERSPECTIVE-TRANSFORMATION Convert center of projection to view plane coordinates

```
Global      XPCNTR, YPCNTR, ZPCNTR the center of projection
            XC, YC, ZC the center of projection in view plane coordinates
BEGIN
    XC ← XPCNTR;
    YC ← YPCNTR;
    ZC ← ZPCNTR;
    VIEW-PLANE-TRANSFORM(XC, YC, ZC);
    IF ZC < 0 THEN
        RETURN ERROR 'CENTER OF PROJECTION BEHIND VIEW PLANE';
    RETURN;
END;
```

8.22 Algorithm MAKE-PARALLEL-TRANSFORMATION Calculation of direction of projection in view plane coordinates

```
Global      TMATRIX a 4 × 3 coordinate transformation matrix array
            DXP, DYP, DZP the parallel projection vector
            VXP, VYP, VZP direction of projection in view plane coordinates
            SXP, SYP the slopes of the projection relative to z direction
Constant    ROUNDOFF some small number greater than any round-off error
BEGIN
    VXP ← DXP * TMATRIX[1, 1] + DYP * TMATRIX[2, 1] +
        DZP * TMATRIX[3, 1];
    VYP ← DXP * TMATRIX[1, 2] + DYP * TMATRIX[2, 2] +
        DZP * TMATRIX[3, 2];
    VZP ← DXP * TMATRIX[1, 3] + DYP * TMATRIX[2, 3] +
        DZP * TMATRIX[3, 3];
    IF |VZP| < ROUNDOFF THEN
        RETURN ERROR 'PROJECTION PARALLEL VIEW PLANE';
    SXP ← VXP / VZP;
    SYP ← VYP / VZP;
    RETURN;
END;
```

CLIPPING IN THREE DIMENSIONS

In Chapter 6 we introduced the idea of a window, which served as a clipping boundary in two-dimensional space. In three-dimensional space the concept can be extended to a clipping volume or view volume. This is a three-dimensional region or box. Objects within the view volume may be seen, while those outside are not displayed. Objects crossing the boundary are cut, and only the portion within the view volume is shown. A view volume may clip the front or back of an object as well as its sides. For example, imagine the image of a building. The picture centers on the door of the building. The size of the entrance increases as you seem to approach it. Passing through the doorway, the outside walls disappear. The display now shows the entry hall. Clipping may be used to remove the front wall of the building, which was hiding its interior. (See Figure 8-37.)

The extension from two-dimensional to three-dimensional clipping is not a difficult one. The methods remain basically the same. The difference lies in the test to see whether or not a point is inside the visible region. Instead of comparing the point against a line, we now must compare the point against a plane. In general, any plane

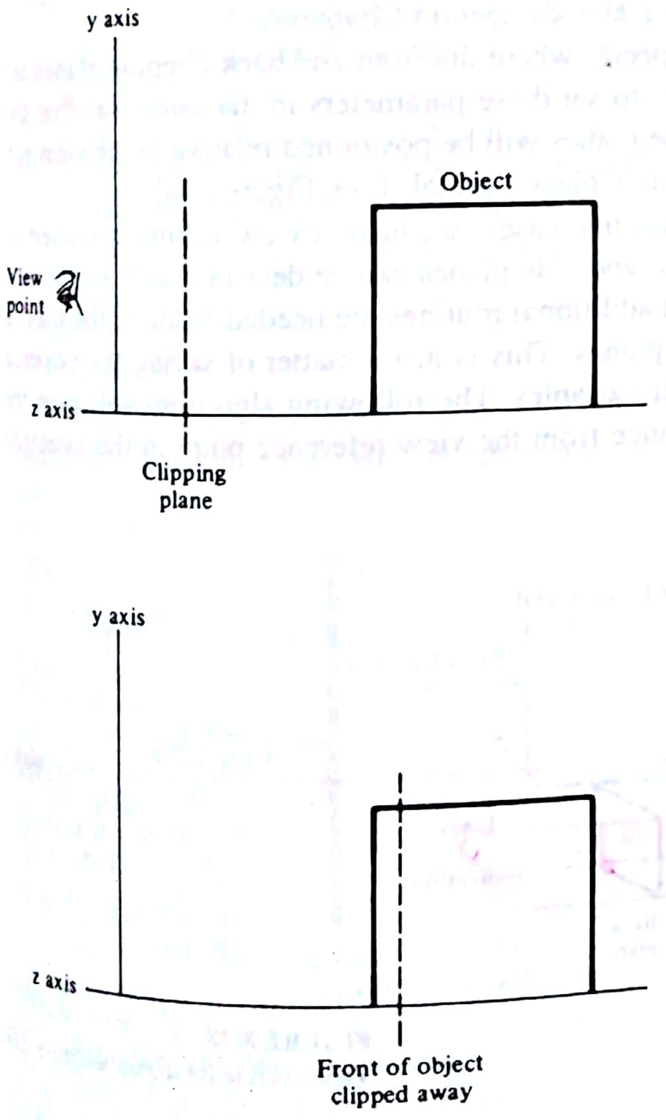


FIGURE 8-37
Moving a clipping plane through an object.

may be used as a boundary, and clipping regions can be arbitrary polyhedra; but there are two *view volume* shapes which are usually used because they are easy to work with. The type of view volume used depends on whether a parallel or a perspective projection is to be employed. For a parallel projection, imagine planes that are in the direction of projection extending from the edges of the window. These planes form a rectangular tube in space. Front and back clipping planes can be added to section this tube into a box. Objects within the box are visible, whereas those outside are clipped away. (See Figure 8-38.) An application of front and back clipping is shown in Plate 2, where the clipping planes are used to show a slice of a molecule. For a perspective projection, we picture rays from the center of projection passing through the window to form a viewing pyramid. This pyramid can be truncated by the front and back clipping planes to form the volume in which objects may be seen. (See Figure 8-39.)

The reason for choosing these regions becomes apparent when we consider what happens when the objects are projected. After projection, objects within the view volume will lie within the window on the view plane. After projection, the clipping planes run parallel to the z axis through the window boundaries, and their equations look the same as the window boundary line equations. This means that if we project the objects before clipping, then the left, right, top, and bottom clipping cases are essentially the same as the window clipping we have already seen in Chapter 6.

We must provide a means to specify where the front and back clipping planes are located. We shall provide algorithms to set these parameters in the same way that the window boundary was specified. The planes will be positioned relative to the view reference point in the direction of the view plane normal. (See Figure 8-40.)

For both the parallel and perspective cases, we have a view volume bounded by six clipping planes. The top, bottom, and side planes can be determined from the window and projection information, but additional routines are needed to allow the user to specify the front and back clipping planes. This is just a matter of saving the position specified by the user in some global variables. The following algorithm will save the positions given in terms of the distance from the view reference point in the direction of the view plane normal.

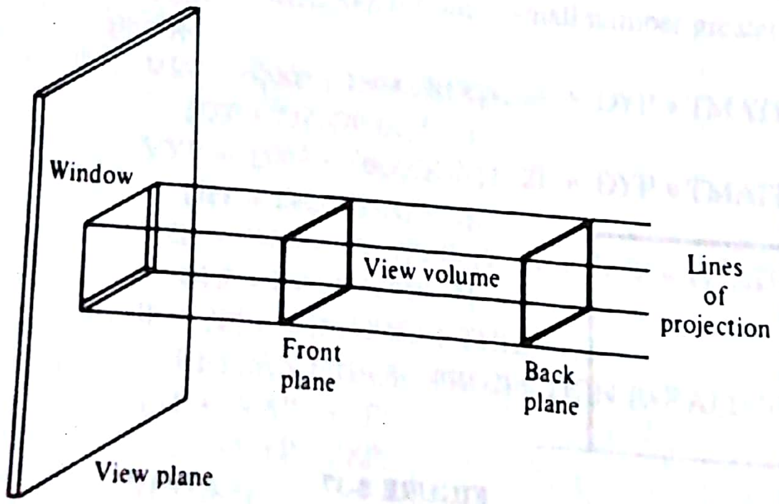


FIGURE 8-38
View volume for a parallel projection.

Scanned by CamScanner

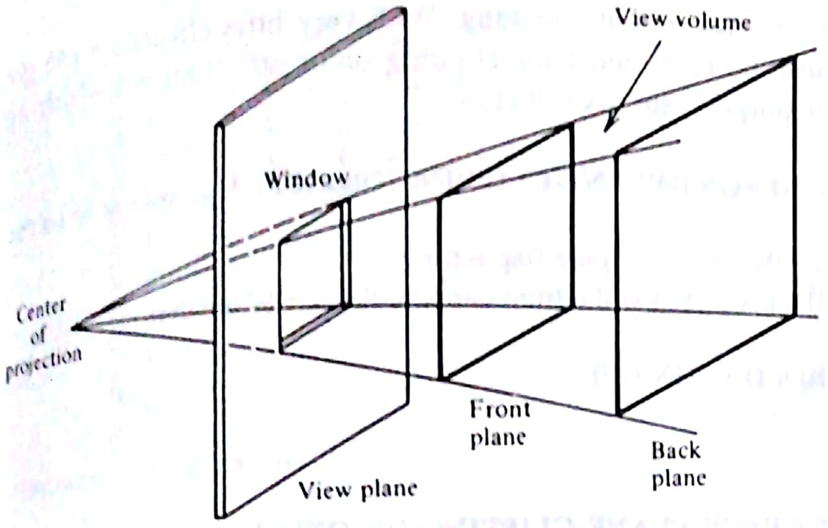


FIGURE 8-39
View volume for a perspective projection.

8.23 Algorithm SET-VIEW-DEPTH(FRONT-DISTANCE, BACK-DISTANCE) User routine to specify the position of front and back clipping planes

```
Arguments   FRONT-DISTANCE, BACK-DISTANCE plane distance from the view
            reference point along the view plane normal
Global      FRONT-HOLD, BACK-HOLD storage for plane positions
BEGIN
   IF FRONT-DISTANCE > BACK-DISTANCE THEN
      RETURN ERROR 'FRONT PLANE BEHIND BACK PLANE';
   FRONT-HOLD ← FRONT-DISTANCE;
   BACK-HOLD ← BACK-DISTANCE;
   RETURN;
END;
```

We could, if we wished, omit the front and/or back clipping tests. All that is needed is a flag which indicates whether to compare a point against the clipping plane

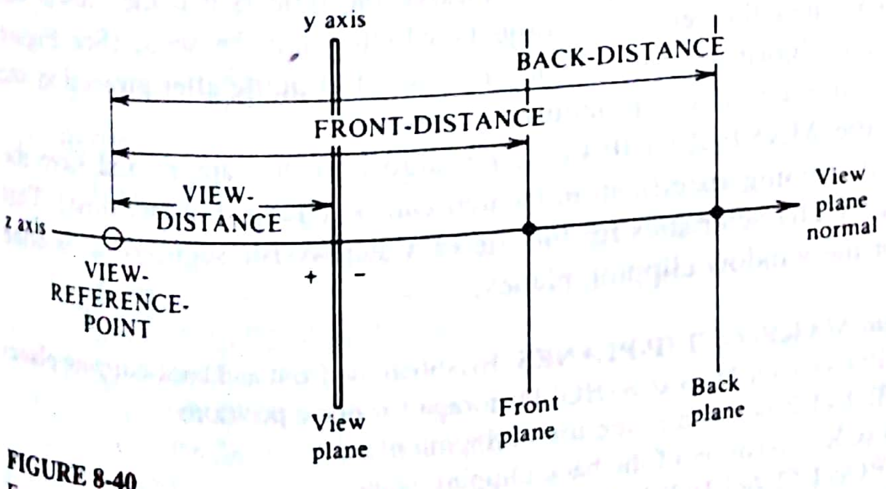


FIGURE 8-40
Front and back clipping plane specification.

or to pass it to the next routine without checking. With very little effort, we can give the user the ability to turn the front and back clipping on or off. Routines which will set clipping flags for this purpose are given below.

8.24 Algorithm SET-FRONT-PLANE-CLIPPING(ON-OFF) User routine to set the front clipping flag

```
Argument   ON-OFF the user's clipping flag setting
Global     FRONT-FLAG-HOLD the front clipping flag set by the user
BEGIN
    FRONT-FLAG-HOLD ← ON-OFF;
    RETURN;
END;
```

8.25 Algorithm SET-BACK-PLANE-CLIPPING(ON-OFF) User routine to set the back clipping flag

```
Argument   ON-OFF the user's clipping flag setting
Global     BACK-FLAG-HOLD the back clipping flag set by the user
BEGIN
    BACK-FLAG-HOLD ← ON-OFF;
    RETURN;
END;
```

We need to decide whether to perform the z-plane clipping before or after projection. Clipping can be done at either point, provided that the $Z/(Z_C - Z)$ transformation is used on the clipping plane position if we are clipping after a perspective projection. The argument for clipping in z after the projection is that CLIP-FRONT and CLIP-BACK algorithms can simply be incorporated into the clipping sequence. The reason why clipping before projection is desirable is that perspective projection requires that objects lie behind the center of projection. Now if we wish to construct programs where the center of projection (the eye of the viewer) moves among objects and perhaps even through them, then we would like to remove from consideration all objects which lie in front of the center of projection before that projection is carried out. That is, we want to employ front clipping to remove the objects that the viewer has passed and then to project the remaining objects which might be seen. (See Figure 8-41.) We shall present algorithms for the simpler case of clipping after projection and leave clipping before projection as a programming problem.

We shall use the MAKE-Z-CLIP-PLANES algorithm to capture and save the user's front and back clipping specification (which can be changed at any time). This will give fixed clipping characteristics for the life of a display-file segment, as is done in NEW-VIEW-2 for the window clipping planes.

8.26 Algorithm MAKE-Z-CLIP-PLANES Establish the front and back clipping planes

```
Global     FRONT-HOLD, BACK-HOLD storage for plane positions
           FRONT-Z position of the front clipping plane
           BACK-Z position of the back clipping plane
           FRONT-FLAG-HOLD the front clipping flag set by the user
           BACK-FLAG-HOLD the back clipping flag set by the user
```

Scanned by CamScanner

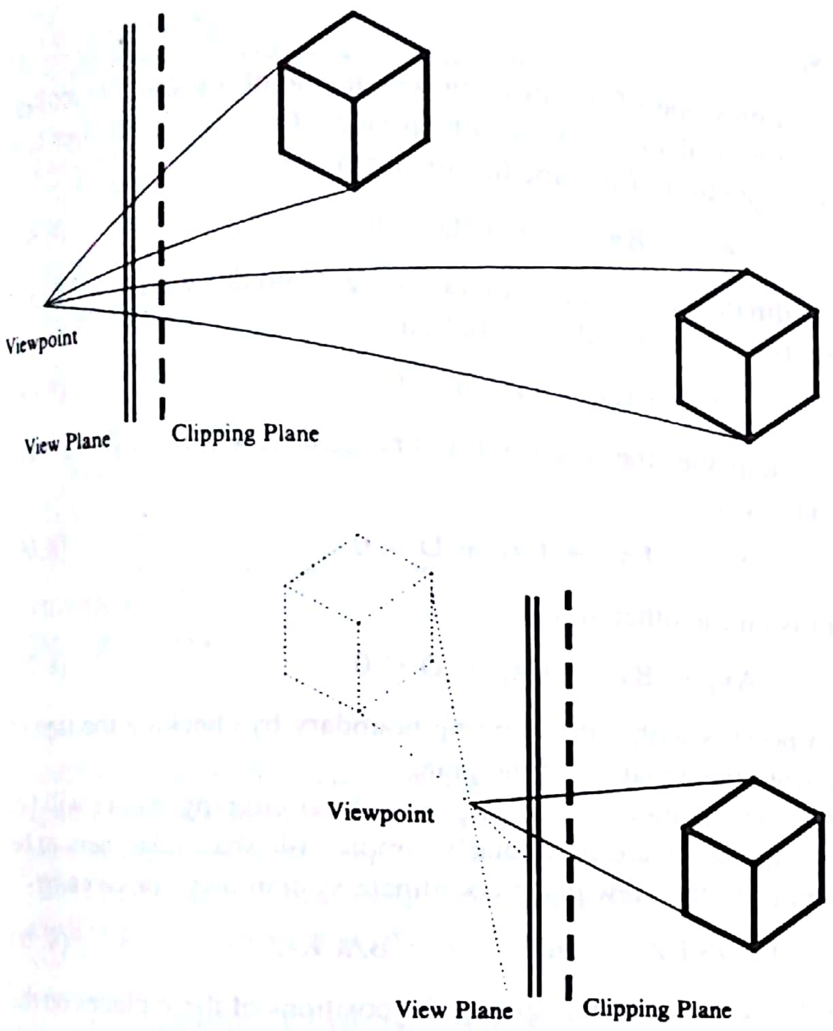


FIGURE 8-41
Front clipping before projection can remove objects that should not be projected.

```
            FRONT-FLAG the front clipping flag
            BACK-FLAG the back clipping flag
            VIEW-DISTANCE the permanent storage for the view distance
            PERSPECTIVE-FLAG the perspective projection flag
            XC, YC, ZC the center of projection in view plane coordinates
BEGIN
    FRONT-FLAG ← FRONT-FLAG-HOLD;
    BACK-FLAG ← BACK-FLAG-HOLD;
    FRONT-Z ← VIEW-DISTANCE − FRONT-HOLD;
    BACK-Z ← VIEW-DISTANCE − BACK-HOLD;
    IF PERSPECTIVE-FLAG THEN
        BEGIN
            FRONT-Z ← FRONT-Z / (ZC − FRONT-Z);
            BACK-Z ← BACK-Z / (ZC − BACK-Z);
        END;
    RETURN;
END;
```

Scanned by CamScanner

CLIPPING PLANES

The clipping problem becomes one of deciding on which side of a plane a point lies. This is similar to the problem of deciding on which side of a line a point lies in a plane. Remember that the equation of a plane has the form

$$Ax + By + Cz + D = 0 \tag{8.67}$$

Suppose we plug the coordinate values of a point (x_1, y_1, z_1) into this equation. Then if the point is on the plane, the equation will be satisfied.

$$Ax_1 + By_1 + Cz_1 + D = 0 \tag{8.68}$$

But if the point is not on the plane, the result will not be zero. It will be positive if the point is on one side of the plane

$$Ax_1 + By_1 + Cz_1 + D > 0 \tag{8.69}$$

and negative if the point is on the other side.

$$Ax_1 + By_1 + Cz_1 + D < 0 \tag{8.70}$$

We can therefore tell if a point is within the clipping boundary by checking the sign of the expression obtained from the equation of the plane.

Let's consider what the equations describing our actual clipping planes will be. The front and back clipping planes are particularly simple. We shall take them to be parallel to the view plane, so in the view plane coordinate system they are given by

$$z = \text{FRONT-Z} \quad \text{and} \quad z = \text{BACK-Z} \tag{8.71}$$

where FRONT-Z and BACK-Z are constants giving the positions of these planes on the z axis. (See Figure 8-42.)

For the point (x_1, y_1, z_1) to be visible, it must be behind or on the front plane and in front of or on the back plane. The tests should then be

$$z_1 \leq \text{FRONT-Z} \quad \text{and} \quad z_1 \geq \text{BACK-Z} \tag{8.72}$$

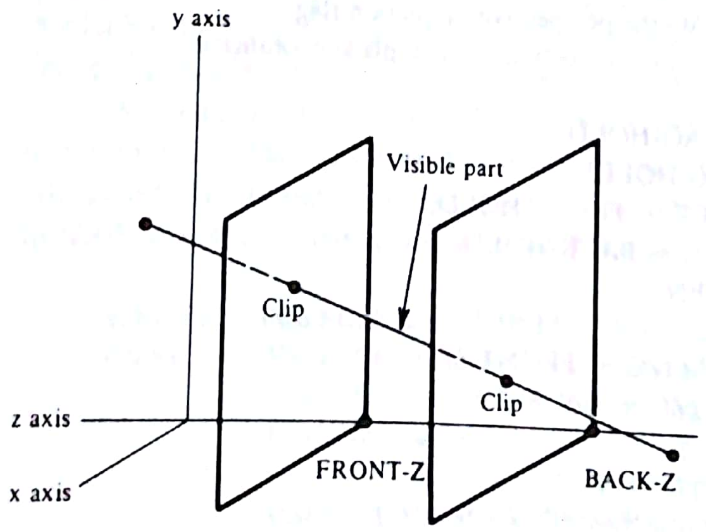


FIGURE 8-42
Front and back clipping.

Scanned by CamScanner

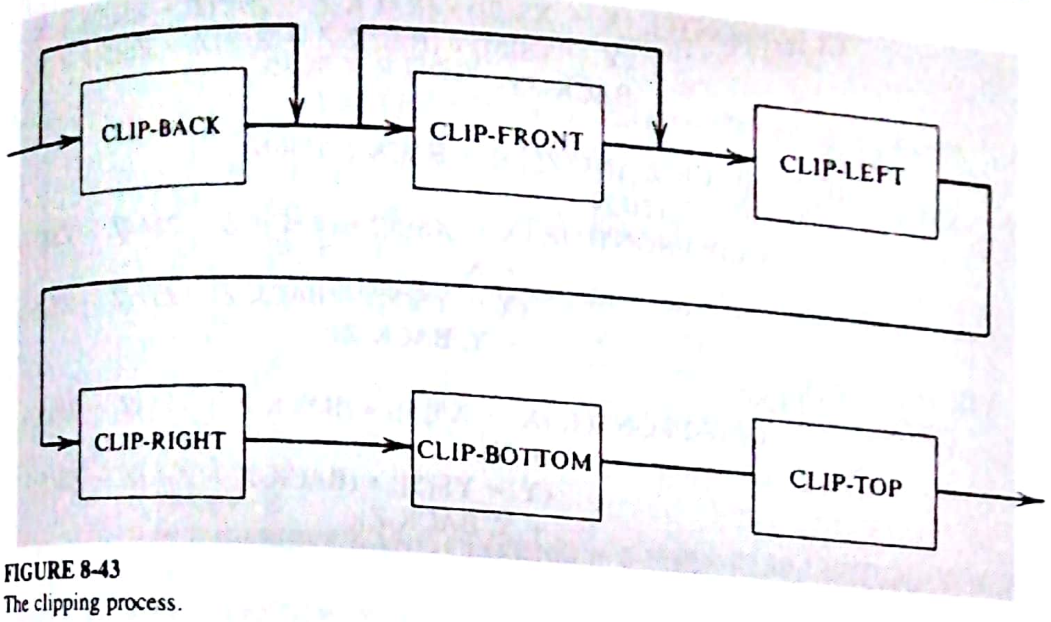


FIGURE 8-43
The clipping process.

Using these tests, we may write algorithms similar to those of Chapter 6 for clipping against back and front planes. These algorithms may be included with upgraded versions of the two-dimensional algorithms to give us three-dimensional clipping. Each visible point passes from one algorithm to the next until it has been checked against all clipping planes. (See Figure 8-43.)

8.27 Algorithm CLIP-BACK(OP, X, Y, Z) Routine for clipping against the back boundary

Arguments	OP, X, Y, Z a display-file instruction
Global	BACK-Z position of the back clipping plane
	BACK-FLAG the back clipping flag
	XS, YS, ZS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
  IF BACK-FLAG THEN
    BEGIN
      IF PFLAG AND NEEDFIRST[5] THEN
        BEGIN
          FIRSTOP[5] ← OP;
          FIRSTX[5] ← X;
          FIRSTY[5] ← Y;
          FIRSTZ[5] ← Z
          NEEDFIRST[5] ← FALSE;
        END
      ELSE
        IF Z ≤ BACK-Z AND ZS[5] < BACK-Z THEN
```

```
                CLIP-FRONT(1, (X - XS[5]) * (BACK-Z - Z) / (Z - ZS[5]) + X,
                          (Y - YS[5]) * (BACK-Z - Z) / (Z - ZS[5]) + Y,
                          BACK-Z)
            ELSE
                IF Z ≤ BACK-Z AND ZS[5] > BACK-Z THEN
                    IF OP > 0 THEN
                        CLIP-FRONT(OP, (X - XS[5]) * (BACK-Z - Z) / (Z - ZS[5])
                                  + X,
                                  (Y - YS[5]) * (BACK-Z - Z) / (Z - ZS[5])
                                  + Y, BACK-Z)
                    ELSE
                        CLIP-FRONT(1, (X - XS[5]) * (BACK-Z - Z) / (Z - ZS[5])
                                  + X,
                                  (Y - YS[5]) * (BACK-Z - Z) / (Z - ZS[5])
                                  + Y, BACK-Z);
        XS[5] ← X;
        YS[5] ← Y;
        ZS[5] ← Z;
        IF Z ≥ BACK-Z AND CLOSING ≠ 5 THEN CLIP-FRONT(OP, X, Y, Z);
      END
    ELSE CLIP-FRONT(OP, X, Y, Z);
    RETURN;
END;
```

8.28 Algorithm CLIP-FRONT(OP, X, Y, Z) Routine for clipping against the front boundary

Arguments	OP, X, Y, Z a display-file instruction
Global	FRONT-Z position of the front clipping plane
	FRONT-FLAG the front clipping flag
	XS, YS, ZS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
  IF FRONT-FLAG THEN
    BEGIN
      IF PFLAG AND NEEDFIRST[6] THEN
        BEGIN
          FIRSTOP[6] ← OP;
          FIRSTX[6] ← X;
          FIRSTY[6] ← Y;
          FIRSTZ[6] ← Z;
          NEEDFIRST[6] ← FALSE;
        END
      ELSE
        IF Z ≤ FRONT-Z AND ZS[6] > FRONT-Z THEN
          CLIP-LEFT(1, (X - XS[6]) * (FRONT-Z - Z) / (Z - ZS[6]) + X,
                   (Y - YS[6]) * (FRONT-Z - Z) / (Z - ZS[6]) + Y,
                   FRONT-Z)
```

Scanned by CamScanner

```
            ELSE
               IF Z ≥ FRONT-Z and ZS[6] < FRONT-Z THEN
                  IF OP > 0 THEN
                     CLIP-LEFT(OP, (X − XS[6]) * (FRONT-Z − Z) / (Z − ZS[6])
                                 + X
                                 (Y − YS[6]) * (FRONT-Z − Z) / (Z − ZS[6])
                                 + Y, FRONT-Z)
                  ELSE
                     CLIP-LEFT(1, WXH, (X − XS[6]) * (FRONT-Z − Z) /
                                       (Z − ZS[6]) + X,
                                       (Y − YS[6]) * (FRONT-Z − Z) /
                                       (Z − ZS[6]) + Y, FRONT-Z);
         XS[6] ← X;
         YS[6] ← Y;
         ZS[6] ← Z;
         IF Z ≤ FRONT-Z AND CLOSING ≠ 6 THEN CLIP-LEFT(OP, X, Y, Z);
      END;
   ELSE CLIP-LEFT(OP, X, Y, Z);
   RETURN;
END;
```

The following routines are extensions of those presented in Chapter 6. In many cases all that is needed is to include a z coordinate along with x and y. In the four routines which actually do the clipping, the z coordinate of any intersection with the clipping plane must be calculated, as well as the x or the y coordinate. We are maintaining the z-coordinate information in anticipation of its use in the hidden-surface routines of the next chapter.

8.29 Algorithm CLIP-LEFT(OP, X, Y, Z). (An extension of algorithm 6.6 to three dimensions) Routine for clipping against the left boundary

Arguments	OP, X, Y, Z a display-file instruction
Global	WXL window left boundary
	XS, YS, ZS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
   IF PFLAG AND NEEDFIRST[1] THEN
      BEGIN
         FIRSTOP[1] ← OP;
         FIRSTX[1] ← X;
         FIRSTY[1] ← Y;
         FIRSTZ[1] ← Z;
         NEEDFIRST[1] ← FALSE;
      END
   Case of drawing from outside in
   ELSE
      IF X ≥ WXL AND XS[1] < WXL THEN
```

Scanned by CamScanner

```
            CLIP-RIGHT(1, WXL, (Y − YS[1]) * (WXL − X) / (X − XS[1]) + Y,
                               (Z − ZS[1]) * (WXL − X) / (X − XS[1]) + Z)
            Case of drawing from inside out
         ELSE
            IF X ≤ WXL AND XS[1] > WXL THEN
               IF OP > 0 THEN
                  CLIP-RIGHT(OP, WXL, (Y − YS[1]) * (WXL − X) / (X − XS[1])
                                        + Y,
                                        (Z − ZS[1]) * (WXL − X) / (X − XS[1])
                                        + Z)
               ELSE
                  CLIP-RIGHT(1, WXL, (Y − YS[1]) * (WXL − X) / (X − XS[1])
                                       + Y,
                                       (Z − ZS[1]) * (WXL − X) / (X − XS[1])
                                       + Z);
         Remember point to serve as one of the endpoints of next line segment
         XS[1] ← X;
         YS[1] ← Y;
         ZS[1] ← Z;
         Case of point inside
         IF X ≥ WXL AND CLOSING ≠ 1 THEN CLIP-RIGHT(OP, X, Y, Z);
         RETURN;
      END;
```

8.30 Algorithm CLIP-RIGHT(OP, X, Y, Z) (An extension of algorithm 6.7 to three dimensions) Routine for clipping against the right boundary

Arguments	OP, X, Y, Z a display-file instruction
Global	WXH window right boundary
	XS, YS, ZS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
   IF PFLAG AND NEEDFIRST[2] THEN
      BEGIN
         FIRSTOP[2] ← OP;
         FIRSTX[2] ← X;
         FIRSTY[2] ← Y;
         FIRSTZ[2] ← Z;
         NEEDFIRST[2] ← FALSE;
      END
   ELSE
      IF X ≤ WXH AND XS[2] > WXH THEN
         CLIP-BOTTOM(1, WXH, (Y − YS[2]) * (WXH − X) / (X − XS[2]) + Y,
                             (Z − ZS[2]) * (WXH − X) / (X − XS[2]) + Z)
      ELSE
         IF X ≥ WXH and XS[2] < WXH THEN
            IF OP > 0 THEN
               CLIP-BOTTOM(OP, WXH, (Y − YS[2]) * (WXH − X) / (X − XS[2])
                                     + Y,
```

Scanned by CamScanner

```
                                    (Z − ZS[2]) * (WXH − X) / (X − XS[2])
                                    + Z)
        ELSE
            CLIP-BOTTOM(1, WXH, (Y − YS[2]) * (WXH − X) / (X − XS[2])
                                   + Y,
                                   (Z − ZS[2]) * (WXH − X) / (X − XS[2])
                                   + Z);
    XS[2] ← X;
    YS[2] ← Y;
    ZS[2] ← Z;
    IF X ≤ WXH AND CLOSING ≠ 2 THEN CLIP-BOTTOM(OP, X, Y, Z);
    RETURN;
END;
```

8.31 Algorithm CLIP-BOTTOM(OP, X, Y, Z) (An extension of algorithm 6.8 to three dimensions) Routine for clipping against the lower boundary

Arguments	OP, X, Y, Z a display-file instruction
Global	WYL window lower boundary
	XS, YS, ZS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
    IF PFLAG AND NEEDFIRST[3] THEN
        BEGIN
            FIRSTOP[3] ← OP;
            FIRSTX[3] ← X;
            FIRSTY[3] ← Y;
            FIRSTZ[3] ← Z;
            NEEDFIRST[3] ← FALSE;
        END
    ELSE
        IF Y ≥ WYL AND YS[3] < WYL THEN
            CLIP-TOP(1, (X − XS[3]) * (WYL − Y) / (Y − YS[3]) + X, WYL,
                        (Z − ZS[3]) * (WYL − Y) / (Y − YS[3]) + Z)
        ELSE
            IF Y ≤ WYL AND YS[3] > WYL THEN
                IF OP > 0 THEN
                    CLIP-TOP(OP, (X − XS[3]) * (WYL − Y) / (Y − YS[3]) + X, WYL,
                                 (Z − ZS[3]) * (WYL − Y) / (Y − YS[3]) + Z)
                ELSE
                    CLIP-TOP(1, (X − XS[3]) * (WYL − Y) / (Y − YS[3]) + X, WYL,
                                (Z − ZS[3]) * (WYL − Y) / (Y − YS[3]) + Z);
    XS[3] ← X;
    YS[3] ← Y;
    ZS[3] ← Z;
    IF Y ≥ WYL AND CLOSING ≠ 3 THEN CLIP-TOP(OP, X, Y, Z);
    RETURN;
END;
```

Scanned by CamScanner

8.32 Algorithm CLIP-TOP(OP, X, Y, Z) (An extension of algorithm 6.9 to three dimensions) Routine for clipping against the upper boundary

```
Arguments   OP, X, Y, Z a display-file instruction
Global      WYH window upper boundary
            XS, YS, ZS arrays containing the last point drawn
            NEEDFIRST array of indicators for saving the first command
            FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
            CLOSING indicates the stage in polygon
BEGIN
    IF PFLAG AND NEEDFIRST[4] THEN
        BEGIN
            FIRSTOP[4] ← OP;
            FIRSTX[4] ← X;
            FIRSTY[4] ← Y;
            FIRSTZ[4] ← Z;
            NEEDFIRST[4] ← FALSE;
        END
    ELSE
        IF Y ≤ WYH AND YS[4] > WYH THEN
            SAVE-CLIPPED-POINT(1, (X − XS[4]) * (WYH − Y) /
                                  (Y − YS[4]) + X, WYH,
                                  (Z − ZS[4]) * (WYH − Y) / (Y − YS[4]) + Z);
        ELSE
            IF Y ≥ WYH AND YS[4] < WYH THEN
                IF OP > 0 THEN
                    SAVE-CLIPPED-POINT(OP, (X − XS[4]) * (WYH − Y) /
                                          (Y − YS[4]) + X, WYH,
                                          (Z − ZS[4]) * (WYH − Y) /
                                          (Y − YS[4]) + Z)
                ELSE
                    SAVE-CLIPPED-POINT(1, (X − XS[4]) * (WYH − Y) /
                                          (Y − YS[4]) + X, WYH,
                                          (Z − ZS[4]) * (WYH − Y) /
                                          (Y − YS[4]) + Z);
    XS[4] ← X;
    YS[4] ← Y;
    ZS[4] ← Z;
    IF Y ≤ WYH AND CLOSING ≠ 4 THEN SAVE-CLIPPED-POINT(OP, X, Y, Z);
    RETURN;
END;
```

8.33 Algorithm SAVE-CLIPPED-POINT(OP, X, Y, Z) (An extension of algorithm 6.10 to three dimensions) Saves clipped polygons in T-buffer and sends lines and characters to the display file

```
Arguments   OP, X, Y, Z a display-file instruction
Global      COUNT-OUT a counter of number of sides on clipped polygon
            PFLAG indicates if a polygon is being clipped
BEGIN
    IF PFLAG THEN
        BEGIN
```

```
            COUNT-OUT ← COUNT-OUT + 1;
            PUT-IN-T(OP, X, Y, Z, COUNT-OUT);
        END
    ELSE VIEWING-TRANSFORM(OP, X, Y);
    RETURN;
END;
```

8.34 Algorithm PUT-IN-T(OP, X, Y, Z, INDEX) (An extension of algorithm 6.11 to three dimensions)

```
Arguments   OP, X, Y, Z the instruction to be stored
            INDEX the position at which to store it
Global      IT, XT, YT, ZT arrays for temporary storage of polygon sides
BEGIN
    IT[INDEX] ← OP;
    XT[INDEX] ← X;
    YT[INDEX] ← Y;
    ZT[INDEX] ← Z;
    RETURN;
END;
```

8.35 Algorithm CLIP-POLYGON-EDGE(OP, X, Y, Z) (An extension of algorithm 6.12 to three dimensions) Close and enter a clipped polygon into the display file

```
Arguments   OP, X, Y, Z a display-file instruction
Global      PFLAG indicates that a polygon is being drawn
            COUNT-IN the number of sides remaining to be processed
            COUNT-OUT the number of sides to be entered in the display file
            IT, XT, YT, temporary storage arrays for a polygon
            NEEDFIRST array of indicators for saving the first command
            FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
            CLOSING indicates the stage in polygon
            FRONT-FLAG, BACK-FLAG indicate whether front and back clipping is
            done
Local       I for stepping through the polygon sides
BEGIN
    COUNT-IN ← COUNT-IN − 1;
    CLIP-BACK(OP, X, Y, Z);
    IF COUNT-IN ≠ 0 THEN RETURN;
    close the clipped polygon
    CLOSING ← 5;
    IF BACK-FLAG AND NOT NEEDFIRST[5] THEN
    CLIP-BACK(FIRSTOP[5], FIRSTX[5], FIRSTY[5], FIRSTZ[5]);
    CLOSING ← 6;
    IF FRONT-FLAG AND NOT NEEDFIRST[6] THEN
    CLIP-FRONT(FIRSTOP[6], FIRSTX[6], FIRSTY[6], FIRSTZ[6]);
    CLOSING ← 1;
    IF NOT NEEDFIRST[1] THEN
    CLIP-LEFT(FIRSTOP[1], FIRSTX[1], FIRSTY[1], FIRSTZ[1]);
    CLOSING ← 2;
    IF NOT NEEDFIRST[2] THEN
    CLIP-RIGHT(FIRSTOP[2], FIRSTX[2], FIRSTY[2], FIRSTZ[2]);
```

Scanned by CamScanner

```
    CLOSING ← 3;
    IF NOT NEEDFIRST[3] THEN
    CLIP-BOTTOM(FIRSTOP[3], FIRSTX[3], FIRSTY[3], FIRSTZ[3]);
    CLOSING ← 4;
    IF NOT NEEDFIRST[4] THEN
    CLIP-TOP(FIRSTOP[4], FIRSTX[4], FIRSTY[4], FIRSTZ[4]);
    CLOSING ← 0;

    PFLAG ← FALSE;
    IF COUNT-OUT < 3 THEN RETURN;
    enter the polygon into the display file
    VIEWING-TRANSFORM(COUNT-OUT, XT[COUNT-OUT], YT[COUNT-OUT]);
    FOR I = 1 TO COUNT-OUT DO VIEWING-TRANSFORM(IT[I], XT[I], YT[I]);
    RETURN;
END;
```

8.36 Algorithm CLIP(OP, X, Y, Z) (An extension of algorithm 6.13 to three dimensions) Top-level clipping routine

```
Arguments  OP, X, Y, Z the instruction being clipped
Global     PFLAG indicates that a polygon is being processed
           COUNT-IN number of polygon sides still to be input
           COUNT-OUT number of clipped polygon sides stored
           XS, YS, ZS arrays for saving the last point drawn
Local      I for initializing the four clipping routines
BEGIN
  IF PFLAG THEN CLIP-POLYGON-EDGE(OP, X, Y, Z)
  ELSE IF OP > 2 THEN
        BEGIN
          PFLAG ← TRUE;
          COUNT-IN ← OP;
          COUNT-OUT ← 0;
          FOR I = 1 TO 6 DO
            BEGIN
              XS[I] ← X;
              YS[I] ← Y;
              ZS[I] ← Z;
            END;
        END
      ELSE CLIP-BACK(OP, X, Y, Z);
  RETURN;
END;
```

THE 3D VIEWING TRANSFORMATION

We finally have all of the tools which are needed to process the user's three-dimensional drawing. We assume that the user has specified the desired viewing parameters, and that the MAKE-VIEW-PLANE-TRANSFORMATION routine has been used to create a view plane coordinate transformation. The user now calls a three-dimensional

Scanned by CamScanner

drawing command such as LINE-ABS-3(X, Y, Z). This command updates the pen position and calls the DISPLAY-FILE-ENTER routine which transforms and projects the position onto the view plane before entering it into the display file. The steps are to first multiply by the view plane transformation matrix to convert the point to view plane coordinates. This in effect changes the viewpoint to that of our synthetic camera. Second, we perform a projection to place the two-dimensional image on the "film." Finally, we save the image in the display file by means of our clipping routine. Doing the clipping after the view plane transformation makes it seem as if our clipping window is attached to the view plane. It governs the size of the "film" in the synthetic camera. (See Figure 8-44.)

An algorithm to multiply a point by the view plane transformation matrix and thereby convert it from object to view plane coordinates is given next.

8.37 Algorithm VIEW-PLANE-TRANSFORM(X, Y, Z) Transforms a point into the view plane coordinate system

Arguments	X, Y, Z point to be transformed, also for return of result
Global	TMATRIX a 4 × 3 transformation matrix array
Local	T three-element array to hold results until calculation finished
	I index for stepping through the TMATRIX columns

```
BEGIN
  FOR I = 1 TO 3 DO
    T[I] ← X * TMATRIX[1, I] + Y * TMATRIX[2, I] +
    Z * TMATRIX[3, I] + TMATRIX[4, I];
  X ← T[1];
  Y ← T[2];
  Z ← T[3];
  RETURN;
END;
```

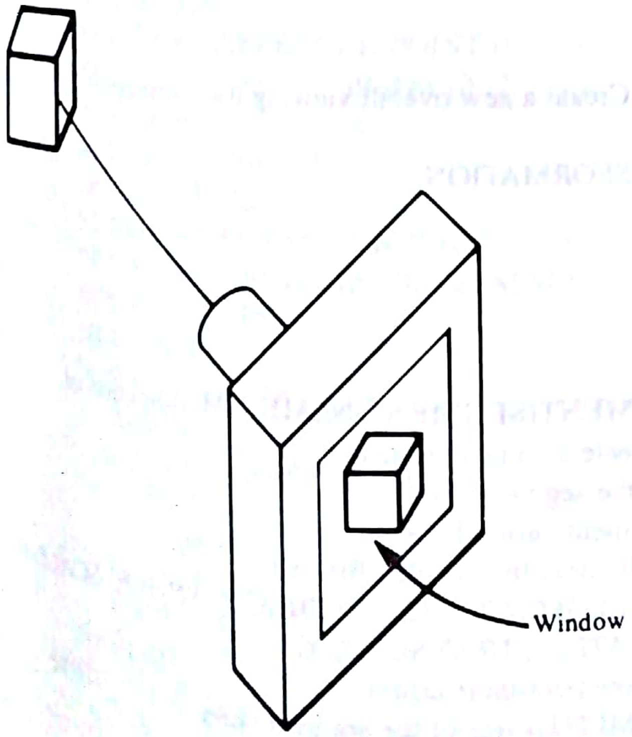


FIGURE 8-44
The window is attached to the view plane.

Scanned by CamScanner

The modified DISPLAY-FILE-ENTER routine is as follows:

8.38 Algorithm DISPLAY-FILE-ENTER(OP) (Revision of algorithm 6.14) Routine to enter an instruction into the display file

```
Argument   OP opcode of instruction to be entered
Global     DF-PEN-X, DF-PEN-Y, DF-PEN-Z the current pen position
           PERSPECTIVE-FLAG perspective vs. parallel projection flag
Local      X, Y, Z hold the point that is transformed
BEGIN
  IF OP < 1 AND OP > -32 THEN PUT-POINT(OP, 0, 0)
  ELSE
    BEGIN
      X ← DF-PEN-X;
      Y ← DF-PEN-Y;
      Z ← DF-PEN-Z;
      VIEW-PLANE-TRANSFORM(X, Y, Z);
      IF PERSPECTIVE-FLAG THEN PERSPECTIVE-TRANSFORM(X, Y, Z)
      ELSE PARALLEL-TRANSFORM(X, Y, Z);
      CLIP(OP, X, Y, Z);
    END;
  RETURN;
END;
```

To finish up, we will extend the CREATE-SEGMENT routine to call an algorithm named NEW-VIEW-3 that forms a new viewing transformation and establishes clipping parameters. This means that a new viewing transformation can be established for each display-file segment. The three-dimensional viewing parameters are therefore established in the same manner as the window specification. The user may set new values at any time, but the new values will not go into effect until a display-file segment is created. Furthermore, a particular viewing specification will remain in effect throughout the segment.

8.39 Algorithm NEW-VIEW-3 Create a new overall viewing transformation

```
BEGIN
  MAKE-VIEW-PLANE-TRANSFORMATION;
  NEW-VIEW-2;
  MAKE-Z-CLIP-PLANES;
  RETURN;
END;
```

8.40 Algorithm CREATE-SEGMENT(SEGMENT-NAME) (Modification of algorithm 7.17) User routine to create a named segment

```
Argument   SEGMENT-NAME the segment name
Global     NOW-OPEN the segment currently open
           FREE the index of the next free display-file cell
           SEGMENT-START, SEGMENT-SIZE, VISIBILITY, ANGLE, SCALE-X,
           SCALE-Y, TRANSLATE-X, TRANSLATE-Y,
           DETECTABLE the segment-table arrays
Constant   NUMBER-OF-SEGMENTS size of the segment table
```

Scanned by CamScanner

```
BEGIN
    IF NOW-OPEN > 0 THEN RETURN ERROR 'SEGMENT STILL OPEN';
    IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
    THEN
        RETURN ERROR 'INVALID SEGMENT NAME';
    IF SEGMENT-SIZE[SEGMENT-NAME] > 0 THEN
        RETURN ERROR 'SEGMENT ALREADY EXISTS';
    NEW-VIEW-3
    SEGMENT-START[SEGMENT-NAME] ← FREE;
    SEGMENT-SIZE[SEGMENT-NAME] ← 0;
    VISIBILITY[SEGMENT-NAME] ← VISIBILITY[0];
    ANGLE[SEGMENT-NAME] ← ANGLE[0];
    SCALE-X[SEGMENT-NAME] ← SCALE-X[0];
    SCALE-Y[SEGMENT-NAME] ← SCALE-Y[0];
    TRANSLATE-X[SEGMENT-NAME] ← TRANSLATE-X[0];
    TRANSLATE-Y[SEGMENT-NAME] ← TRANSLATE-Y[0];
    DETECTABLE[SEGMENT-NAME] ← DETECTABLE[0];
    NOW-OPEN ← SEGMENT-NAME;
    RETURN;
END;
```

We also need an initialization routine to establish default viewing parameters. The default parameters we shall choose are a view reference point at the origin, the view plane in the object coordinate xy plane, the view-up along the object coordinate's y direction, and a parallel projection in the z direction. Front and back clipping will initially be off.

8.41 Algorithm INITIALIZE-8 Initialization of global data

```
Local       I for initialization of the clipping routines
BEGIN
    initialize
    INITIALIZE-7;
    SET-VIEW-REFERENCE-POINT(0, 0, 0);
    SET-VIEW-PLANE-NORMAL(0, 0, -1);
    SET-VIEW-DISTANCE(0);
    SET-VIEW-UP(0, 1, 0);
    SET-PARALLEL(0, 0, 1);
    SET-FRONT-PLANE-CLIPPING(FALSE);
    SET-BACK-PLANE-CLIPPING(FALSE);
    SET-VIEW-DEPTH(0, 0);
    NEW-VIEW-3;
    FOR I = 1 to 6 DO
        BEGIN
            NEEDFIRST[I] ← FALSE;
            XS[I] ← 0;
            YS[I] ← 0;
            ZS[I] ← 0;
        END;
    RETURN;
END;
```

AN APPLICATION

One application relying heavily upon three-dimensional graphics is an airplane flight simulator. A flight simulator can be used as part of a pilot's training. The simulator may look like the real cockpit, only the windshield is replaced by a computer-generated image of the world. This image alters under the pilot's actions in the same manner as his view of the world would change if he were actually flying. In the simulator, the pilot may practice, and even err, without endangering anything but his pride. (See Figure 8-45.)

Let us consider how our three-dimensional graphics system might be used in a flight simulator program. The first thing to be done is to construct a model of the world over which the pilot is to fly. Buildings, runways, fields, lakes, and other landscape features may be constructed using our three-dimensional LINE and POLYGON primitives. Windowing allows us to use real-world dimensions, such as meters. The construction may be somewhat tedious, but it is straightforward. Let us assume that it has been done, that we have a procedure named BUILD-WORLD which contains the commands for drawing the model landscape. We must still project this landscape onto the display screen. We wish to do it in such a manner as to produce the view which the pilot should see from his airplane. We shall consider the view plane to be the cockpit windshield. The view reference point will be attached to the airplane, so as it moves, the windshield (view plane) moves with it. A VIEW-DISTANCE value of zero will do nicely.

It may be convenient to think of the view reference point as centered on the windshield. We can accomplish this with our SET-WINDOW command.

```
SET-WINDOW(-0.5, 0.5, -0.5, 0.5);
```

The orientation of the plane corresponds to the orientation of its windshield, which is set by the SET-VIEW-PLANE-NORMAL command. The pilot may bank or roll the plane. This would change its view-up direction. And finally, we must indicate the projection, which should be a perspective view centered on the pilot's eye. This is behind the windshield, say about 0.5 meters away. (See Figure 8-46.)

Let us outline two routines, one which is called whenever the plane changes its position, and another to be called when the pilot alters the orientation of the plane. To update the plane's position, we change the view reference point and the center of projection.

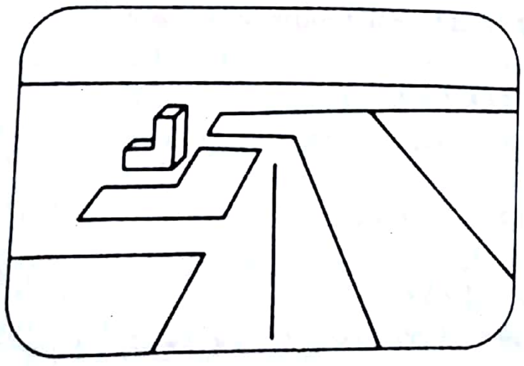


FIGURE 8-45
Airplane flight simulation.

Scanned by CamScanner

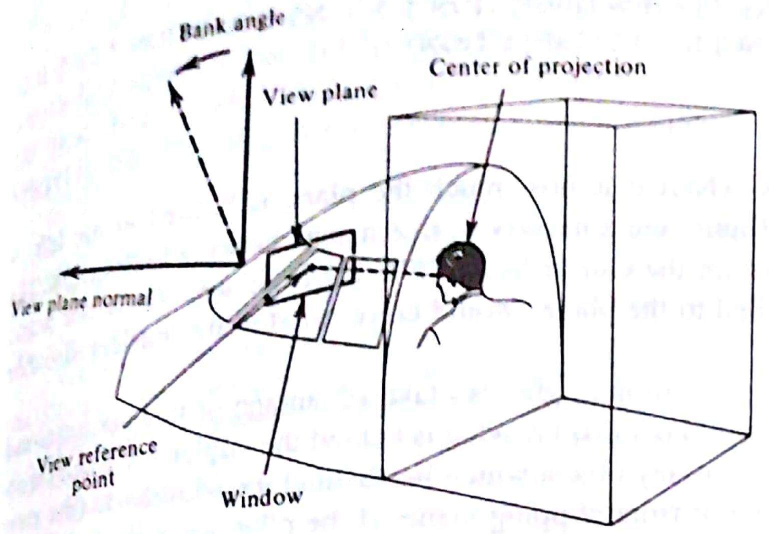


FIGURE 8-46
Parameters for flight simulation.

```
BEGIN
  SET-VIEW-REFERENCE-POINT(EAST-WEST, ALTITUDE, NORTH-SOUTH);
  SET-PERSPECTIVE(EAST-WEST − 0.5 * EW-DIRECTION,
    ALTITUDE − 0.5 * AL-DIRECTION, NORTH-SOUTH
    − 0.5 * NS-DIRECTION);
  CREATE-SEGMENT(2);
  SET-VISIBILITY(2, FALSE);
  BUILD-WORLD;
  CLOSE-SEGMENT;
  SET-VISIBILITY(1, FALSE);
  SET-VISIBILITY(2,TRUE);
  DELETE-SEGMENT(1);
  RENAME-SEGMENT(2, 1);
  RETURN;
END;
```

Here we have used (EAST-WEST, ALTITUDE, NORTH-SOUTH) as the coordinates of the plane's position. The vector [EW-DIRECTION AL-DIRECTION NS-DIRECTION] gives the plane's orientation and is assumed to be normalized. In the SET-PERSPECTIVE command, 0.5 is used as the distance of the pilot from the windshield. It is multiplied by the plane's direction and subtracted from the plane's position to give the pilot's position. The visibility property and segment renaming are used to ensure that the old image of the world is maintained until the new one is ready for display.

A routine to update the plane's orientation might look something like the following:

```
BEGIN
  SET-VIEW-PLANE-NORMAL(EW-DIRECTION, AL-DIRECTION, NS-DIREC-
  TION);
  SET-VIEW-UP(SIN(BANK-ANGLE) * NS-DIRECTION,
```

Scanned by CamScanner

```
        COS(BANK-ANGLE) * (EW-DIRECTION ↑ 2 + NS-DIRECTION ↑ 2) ↑ 0.5,
        - SIN(BANK-ANGLE) * EW-DIRECTION);
    RETURN;
END;
```

In the above, BANK-ANGLE is how much the plane is banked to the left. A value of zero means level flight, and a negative value means banking to the right. The arithmetic which occurs within the call to SET-VIEW-UP finds a vector in the world coordinates which, if attached to the plane, would correspond to the vertical direction in level flight.

A program for flight simulation might also take advantage of three-dimensional clipping. Such a program should not display what is behind the airplane. We should extend the program so that it clips any object which lies behind the windshield (the view plane). We can do this with our front clipping plane. If the pilot has unlimited visibility, then the back clipping plane should not be used. If, on the other hand, we wish to simulate the limiting effects of bad weather on visibility, an approach might be to simply clip away all objects that exceed the range of vision.

Instructions to set up the front and back clipping might look as follows:

```
SET-VIEW-DEPTH(0, VISIBILITY-DISTANCE);
SET-FRONT-PLANE-CLIPPING(TRUE);
SET-BACK-PLANE-CLIPPING(BAD-WEATHER);
```

In the above, VISIBILITY-DISTANCE is the distance which the pilot is able to see. BAD-WEATHER is TRUE if the pilot's visibility is limited, and FALSE otherwise.

FURTHER READING

The mathematics of transformations and projections is further discussed in [ROG76]. An excellent discussion of homogeneous coordinates and transformations is available in [AHU68]. A mathematical discussion of rotations and reflection matrix properties is given in [FIL84]. There is a fine discussion of the homogeneous form of points, lines, and planes in [BLI77]. Homogeneous coordinates and the relationship between projected and Euclidean space are considered in [RIE81]. Homogeneous coordinates and clipping are considered in [BLI78]. A formal description of transformations and clipping in a hierarchical picture structure is described in [MAL78]. A discussion of projections and how they are specified is given in [CAR78]. A discussion of how to use our viewing system is presented in [BER78]. The viewing process is described in [ROG83]. In [MIC80] there is a critical evaluation of the CORE viewing parameters which we have used, and an alternative scheme for describing the view is presented. Perspective projections help to give some realism to three-dimensional objects but still only give a two-dimensional image. Work has been done on devising display hardware that will create three-dimensional images by presenting a different view to each eye. A survey of these techniques is found in [LAN82]. We have shown how to describe three-dimensional objects by their two-dimensional polygon surfaces, but they may also be described by three-dimensional structures. This is called solid modeling and is often used for computer-aided design. Three-dimensional image representation is also useful

Scanned by CamScanner