Three-Dimensional Viewing

When we model a three-dimensional scene, each object in the scene is typically defined with a set of surfaces that form a closed boundary around the object interior. And, for some applications, we may need also to specify information about the interior structure of an object.

In addition to procedures that generate views of the surface features of an object, graphics packages sometimes provide routines for displaying internal components or cross-sectional views of a solid object.

Viewing functions process the object descriptions through a set of procedures that ultimately project a specified view of the objects onto the surface of a display device. Many processes in three-dimensional viewing, such as the clipping routines, are similar to those in the two-dimensional viewing pipeline.

But three-dimensional viewing involves some tasks that are not present in two-dimensional viewing. For example, projection routines are needed to transfer the scene to a view on a planar surface, visible parts of a scene must be identified, and, for a realistic display, lighting effects and surface characteristics must be taken into account.

Viewing a Three-Dimensional Scene:

To obtain a display of a three-dimensional world-coordinate scene, we first set up a cordinate reference for the viewing, or "camera," parameters.

This coordinate reference defines the position and orientation for a view plane (or projection plane) that corresponds to a camera film plane (Figure 1).

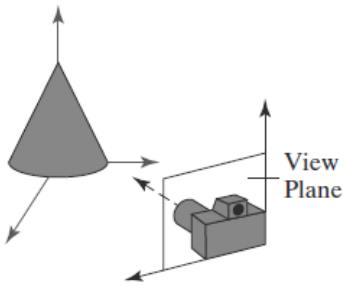

FIGURE 1
Coordinate reference for obtaining a selected view of a three-dimensional scene.

Object descriptions are then transferred to the viewing reference coordinates and projected onto the view plane.

We can generate a view of an object on the output device in wireframe (outline) form, or we can apply lighting and surface-rendering techniques to obtain a realistic shading of the visible surfaces.

Projections:

Unlike a camera picture, we can choose different methods for projecting a scene onto the view plane.

One method for getting the description of a solid object onto a view plane is to project points on the object surface along parallel lines. This technique, called **parallel projection**, is used in engineering and architectural drawings to represent an object with a set of views that show accurate dimensions of the object, as in Figure 2.

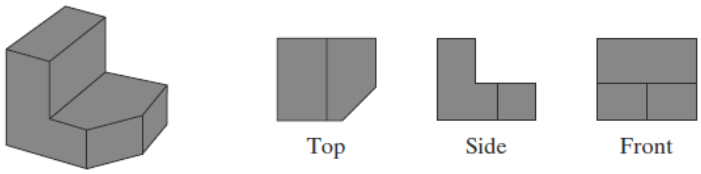

FIGURE 2
Three parallel-projection views of an object, showing relative proportions from different viewing positions.

Another method for generating a view of a three-dimensional scene is to project points to the view plane along converging paths. This process, called a **perspective projection,** causes objects farther from the viewing position to be displayed smaller than objects of the same size that are nearer to the viewing position.

A scene that is generated using a perspective projection appears more realistic, because this is the way that our eyes and a camera lens form images. Parallel lines along the viewing direction appear to converge to a distant point in the background, and objects in the background appear to be smaller than objects in the foreground.

Depth Cueing:

With few exceptions, depth information is important in a three-dimensional scene so that we can easily identify, for a particular viewing direction, which is the front and which is the back of each displayed object.

Figure 3 illustrates the ambiguity that can result when a wire-frame object is displayed without depth information.

There are several ways in which we can include depth information in the two-dimensional representation of solid objects.

A simple method for indicating depth with wire-frame displays is to vary the brightness of line segments according to their distances from the viewing position.

Figure 4 shows a wire-frame object displayed with depth cueing. The lines closest to the viewing position are displayed with the highest intensity, and lines farther away are displayed with decreasing intensities.

Depth cueing is applied by choosing a maximum and a minimum intensity value and a range of distances over which the intensity is to vary.

Another application of depth cuing is modeling the effect of the atmosphere on the perceived intensity of objects. More distant objects appear dimmer to us than nearer objects due to light scattering by dust

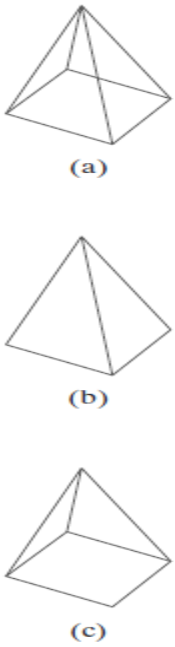

FIGURE 3

particles, haze, and smoke. Some atmospheric effects can even change the perceived color of an object, and we can model these effects with depth cueing.

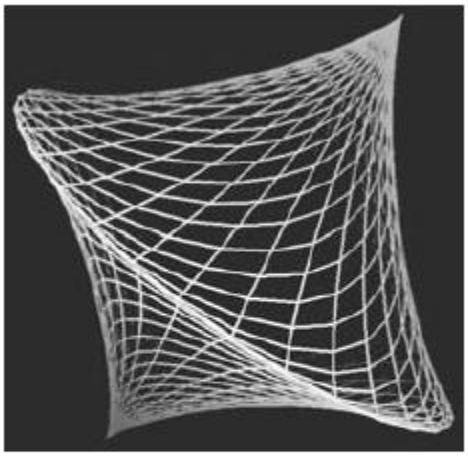

FIGURE 4
A wire-frame object displayed with depth cueing, so that the brightness of lines decreases from the front of the object to the back.

Surface Rendering:

Added realism is attained in displays by rendering object surfaces using the lighting conditions in the scene and the assigned surface characteristics.

We set the lighting conditions by specifying the color and location of the light sources, and we can also set background illumination effects.

Surface properties of objects include whether a surface is transparent or opaque and whether the surface is smooth or rough. We set values for parameters to model surfaces such as glass, plastic, wood-grain patterns, and the bumpy appearance of an orange.

Color Plate 9
A realistic room display, achieved with a perspective projection, illumination effects, and selected surface properties. (*Courtesy of John Snyder, Jed Lengyel, Devendra Kalra, and Al Barr, California Institute of Technology. © 1992 Caltech.*)

Exploded and Cutaway Views:

Many graphics packages allow objects to be defined as hierarchical structures, so that internal details can be stored. Exploded and cutaway views of such objects can then be used to show the internal structure and relationship of the object parts.

An alternative to exploding an object into its component parts is a cutaway view, which removes part of the visible surfaces to show internal structure.

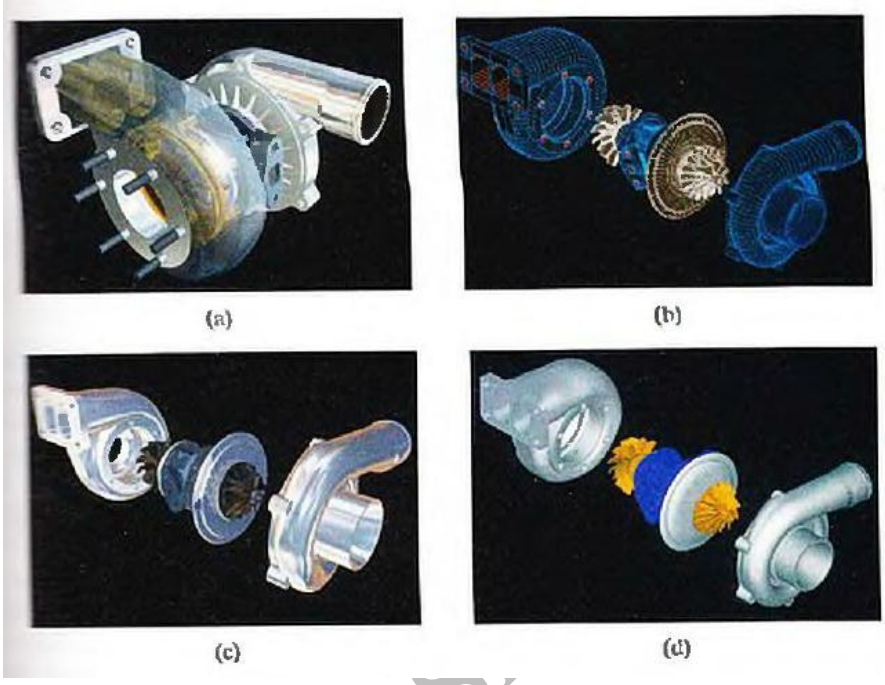

FIGURE 7-8 A fully rendered and assembled turbine (a) can be viewed as an exploded wire-frame display (b), a surface-rendered exploded display (c), or a surface-rendered, color-coded, exploded display (d). (*Courtesy of Autodesk, Inc.*)

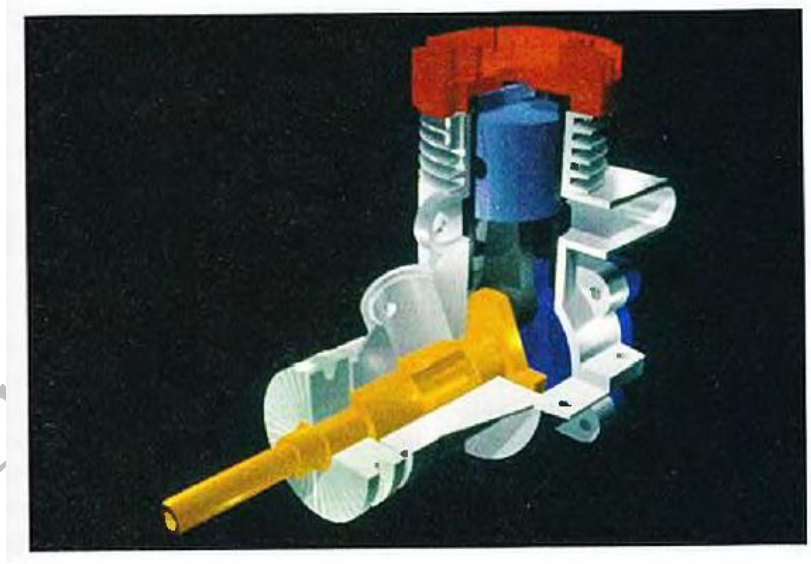

FIGURE 7-9 Color-coded cutaway view of a lawn mower engine, showing the structure and relationship of internal components. (*Courtesy of Autodesk, Inc.*)

Three-Dimensional and Stereoscopic Viewing:

Other methods for adding a sense of realism to a computer-generated scene include three-dimensional displays and stereoscopic views.

Three-dimensional views can be obtained by reflecting a raster image from a vibrating, flexible mirror. The vibrations of the mirror are synchronized with the display of the scene on the cathode ray tube (CRT). As the mirror vibrates, the focal length varies so that each point in the scene is reflected to a spatial position corresponding to its depth.

Stereoscopic devices present two views of a scene: one for the left eye and the other for the right eye.

The viewing positions correspond to the eye positions of the viewer. These two views are typically displayed on alternate refresh cycles of a raster monitor.

When we view the monitor through special glasses that alternately darken first one lens and then the other, in synchronization with the monitor refresh cycles, we see the scene displayed with a three-dimensional effect.

The Three-Dimensional Viewing Pipeline:

Procedures for generating a computer-graphics view of a three-dimensional scene are somewhat analogous to the processes involved in taking a photograph.

First of all, we need to choose a viewing position corresponding to where we would place a camera. We choose the viewing position according to whether we want to display a front, back, side, top, or bottom view of the scene.

We could also pick a position in the middle of a group of objects or even inside a single object, such as a building or a molecule. Then we must decide on the camera orientation. Which way do we want to point the camera from the viewing position, and how should we rotate it around the line of sight to set the "up" direction for the picture? Finally, when we snap the shutter, the scene is cropped to the size of a selected clipping window, which corresponds to the aperture or lens type of a camera, and light from the visible surfaces is projected onto the camera film.

Some of the viewing operations for a three-dimensional scene are the same as, or similar to, those used in the two-dimensional viewing pipeline.

A two-dimensional viewport is used to position a projected view of the three dimensional scene on the output device, and a two-dimensional clipping window is used to select a view that is to be mapped to the viewport.

In addition, we set up a display window in screen coordinates, just as we do in a two-dimensional application.

Clipping windows, viewports, and display windows are usually specified as rectangles with their edges parallel to the coordinate axes. In three-dimensional viewing, however, the clipping window is positioned on a selected view plane, and scenes are clipped against an enclosing volume of space, which is defined by a set of clipping planes.

The viewing position, view plane, clipping window, and clipping planes are all specified within the viewing-coordinate reference frame.

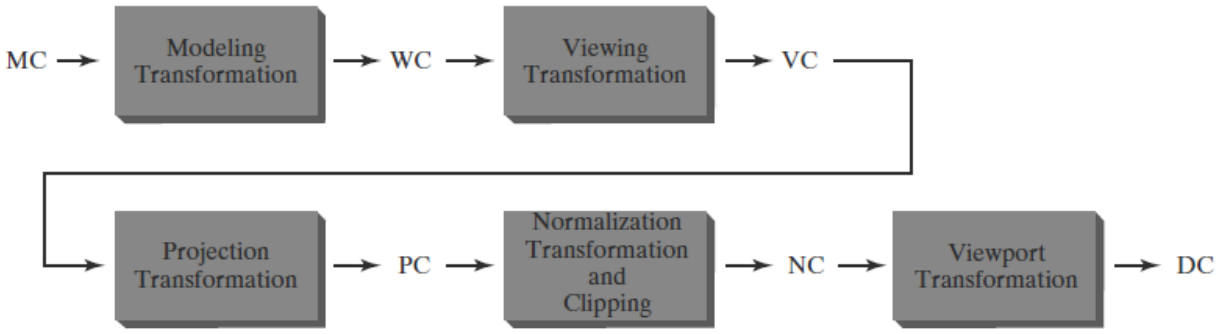

FIGURE 10-6
General three-dimensional transformation pipeline, from modeling coordinates (MC) to world coordinates (WC) to viewing coordinates (VC) to projection coordinates (PC) to normalized coordinates (NC) and, ultimately, to device coordinates (DC).

The above Figure shows the general processing steps for creating and transforming a three-dimensional scene to device coordinates.

Once the scene has been modeled in world coordinates, a viewing-coordinate system is selected and the description of the scene is converted to viewing coordinates.

The viewing coordinate system defines the viewing parameters, including the position and orientation of the projection plane (view plane), which we can think of as the camera film plane.

A two-dimensional clipping window, corresponding to a selected camera lens, is defined on the projection plane, and a three-dimensional clipping region is established. This clipping region is called the view volume, and its shape and size depends on the dimensions of the clipping window, the type of projection we choose, and the selected limiting positions along the viewing direction.

Projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane.

Objects are mapped to normalized coordinates, and all parts of the scene outside the view volume are clipped off.

The clipping operations can be applied after all device-independent coordinate transformations (from world coordinates to normalized coordinates) are completed.

As in two-dimensional viewing, the viewport limits could be given in normalized coordinates or in device coordinates.

There are also a few other tasks that must be performed, such as identifying visible surfaces and applying the surface-rendering procedures.

The final step is to map viewport coordinates to device coordinates within a selected display window. Scene descriptions in device coordinates are sometimes expressed in a left-handed reference frame so that positive distances from the display screen can be used to measure depth values in the scene.

Three-Dimensional Viewing-Coordinate Parameters:

Establishing a three-dimensional viewing reference frame is similar to setting up the two-dimensional viewing reference frame. We first select a world-coordinate position $P_0 =(x_0, y_0, z_0)$ for the viewing origin, which is called the view point or viewing position. (Sometimes the view point is also referred to as the eye position or the camera position.)

And we specify a view-up vector V, which defines the $\mathbf{y_{view}}$ direction. For three-dimensional space, we also need to assign a direction for one of the remaining two coordinate axes. This is typically accomplished with a second vector that defines the $\mathbf{z_{view}}$ axis, with the viewing direction along this axis. Figure 7 illustrates the positioning of a three-dimensional viewing-coordinate frame within a world system.

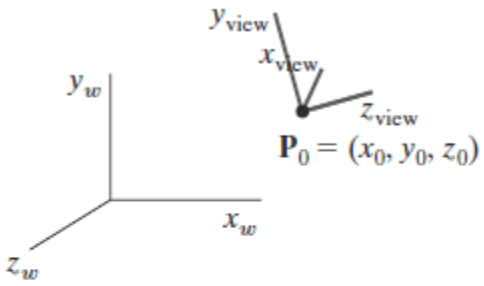

FIGURE 7
A right-handed viewing-coordinate system, with axes x_{view}, y_{view}, and z_{view}, relative to a right-handed world-coordinate frame.

The View-Plane Normal Vector:

Because the viewing direction is usually along the $\mathbf{z_{view}}$ axis, the view plane, also called the projection plane, is normally assumed to be perpendicular to this axis. Thus, the orientation of the view plane, as well as the direction for the positive $\mathbf{z_{view}}$ axis, can be defined with a view-plane normal vector N, as shown in Figure 8.

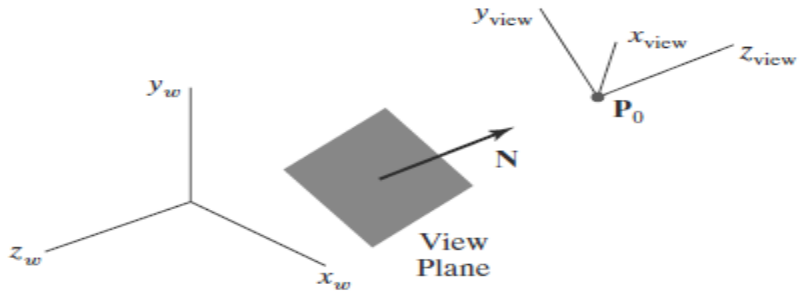

FIGURE 8
Orientation of the view plane and view-plane normal vector **N**.

An additional scalar parameter is used to set the position of the view plane at some coordinate value $\mathbf{z_{vp}}$ along the $\mathbf{z_{view}}$ axis, as illustrated in Figure 9.

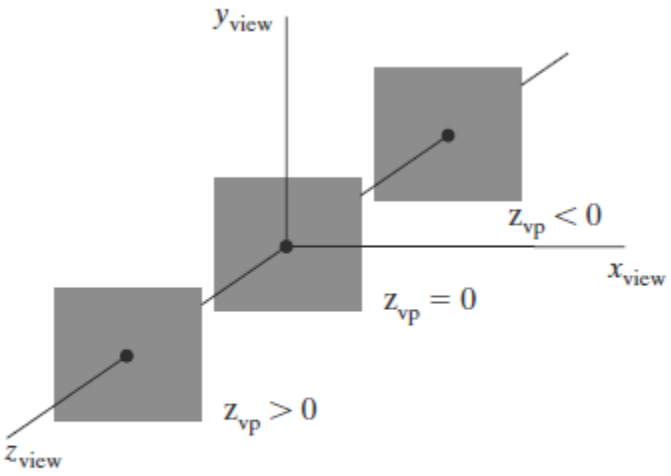

FIGURE 9
Three possible positions for the view plane along the z_{view} axis.

This parameter value is usually specified as a distance from the viewing origin along the direction of viewing, which is often taken to be in the negative $\mathbf{z_{view}}$ direction. Thus, the view plane is always parallel to the $\mathbf{x_{view}y_{view}}$ plane, and the projection of objects to the view plane corresponds to the view of the scene that will be displayed on the output device.

Vector **N** can be specified in various ways. In some graphics systems, the direction for **N** is defined to be along the line from the world-coordinate origin to a selected point position. Other systems take N to be in the direction from a reference point $\mathbf{P_{ref}}$ to the viewing origin $\mathbf{P_0}$, as in Figure 10.

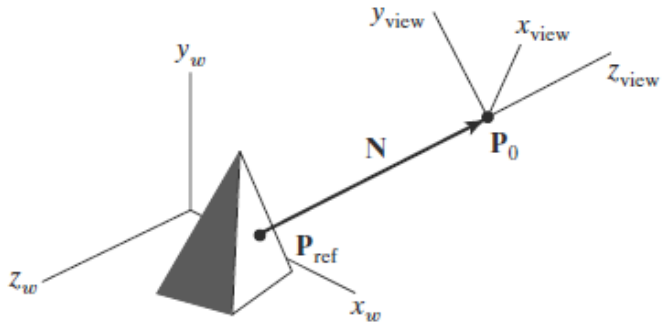

FIGURE 10
Specifying the view-plane normal vector **N** as the direction from a selected reference point $\mathbf{P_{ref}}$ to the viewing-coordinate origin $\mathbf{P_0}$.

In this case, the reference point is often referred to as a look-at point within the scene, with the viewing direction opposite to the direction of **N.**

We could also define the view-plane normal vector, and other vector directions, using direction angles. These are the three angles, α, β, and γ , that a spatial line makes with the x, y, and z axes, respectively.

But it is usually much easier to specify a vector direction with two point positions in a scene than with direction angles.

The View-Up Vector:

Once we have chosen a view-plane normal vector N, we can set the direction for the view-up vector V. This vector is used to establish the positive direction for the $\mathbf{y}_{view}$ axis.

Usually, V is defined by selecting a position relative to the world-coordinate origin, so that the direction for the view-up vector is from the world origin to this selected position. Because the view-plane normal vector N defines the direction for the $\mathbf{z}_{view}$ axis, vector V should be perpendicular to N.

But, in general, it can be difficult to determine a direction for V that is precisely perpendicular to N. Therefore, viewing routines typically adjust the user-defined orientation of vector V, as shown in Figure 11, so that V is projected onto a plane that is perpendicular to the view-plane normal vector.

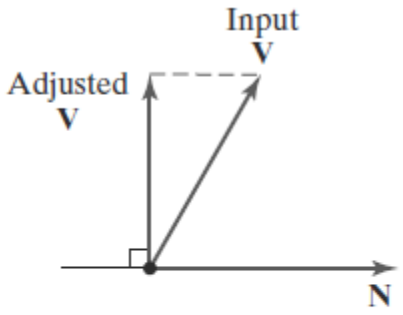

FIGURE 11
Adjusting the input direction of the view-up vector **V** to an orientation perpendicular to the view-plane normal vector **N**.

We can choose any direction for the view-up vector V, so long as it is not parallel to **N**. A convenient choice is often in a direction parallel to the world y_w axis; that is, we could set **V = (0, 1, 0).**

The uvn Viewing-Coordinate Reference Frame:

Since the view-plane normal **N** defines the direction for the $\mathbf{z}_{view}$ axis and the view-up vector V is used to obtain the direction for the $\mathbf{y}_{view}$ axis, we need only determine the direction for the $\mathbf{x}_{view}$ axis.

Using the input values for **N** and **V**, we can compute a third vector, **U**, that is perpendicular to both **N** and **V**.

Vector **U** then defines the direction for the positive $\mathbf{x}_{view}$ axis. We determine the correct direction for **U** by taking the vector cross product of **V** and **N** so as to form a right-handed viewing frame.

The vector cross product of **N** and **U** also produces the adjusted value for **V**, perpendicular to both **N** and **U**, along the positive $\mathbf{y_{view}}$ axis.

Following these procedures, we obtain the following set of unit axis vectors for a right-handed viewing coordinate system.

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z)$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)$$

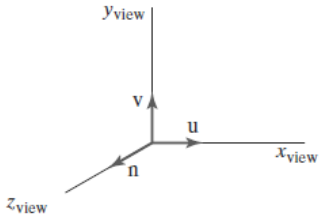

FIGURE 12
A right-handed viewing system defined with unit vectors **u**, **v**, and **n**.

The coordinate system formed with these unit vectors is often described as a **uvn viewing-coordinate reference frame** (Figure 12).

Generating Three-Dimensional Viewing Effects:

By varying the viewing parameters, we can obtain different views of objects in a scene. For instance, from a fixed viewing position, we could change the direction of **N** to display objects at positions around the viewing-coordinate origin.

We could also vary **N** to create a composite display consisting of multiple views from a fixed camera position.

We can simulate a wide viewing angle by producing seven views of the scene from the same viewing position, but with slight shifts in the viewing direction; the views are then combined to form a composite display.

Similarly, we generate stereoscopic views by shifting the viewing direction as well as shifting the view point slightly to simulate the two eye positions.

In interactive applications, the normal vector **N** is the viewing parameter that is most often changed. Of course, when we change the direction for **N**, we also have to change the other axis vectors to maintain a right-handed viewing-coordinate system.

If we want to simulate an animation panning effect, as when a camera moves through a scene or follows an object that is moving through a scene, we can keep the direction for N fixed as we move the view point, as illustrated in Figure 13.

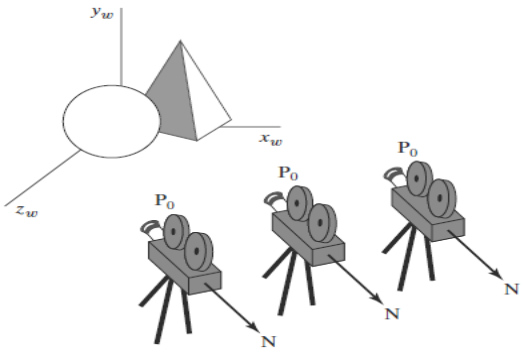

FIGURE 13
Panning across a scene by changing the viewing position, with a fixed direction for **N**.

And to display different views of an object, such as a side view and a front view, we could move the view point around the object, as in Figure 14. Alternatively, different views of an object or group of objects can be generated using geometric transformations without changing the viewing parameters.

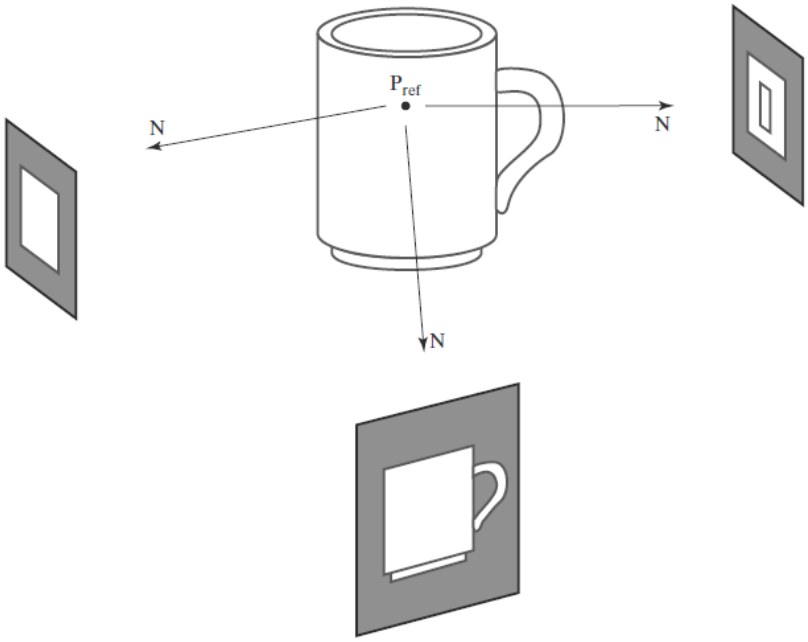

FIGURE 14
Viewing an object from different directions using a fixed reference point.

Transformation from World to Viewing Coordinates:

In the three-dimensional viewing pipeline, the first step after a scene has been constructed is to transfer object descriptions to the viewing-coordinate reference frame. This conversion of object descriptions is equivalent to a sequence of transformations that superimposes the viewing reference frame onto the world frame.

We can accomplish this conversion using the methods for transforming between coordinate system :

1. Translate the viewing-coordinate origin to the origin of the world coordinate system.
2. Apply rotations to align the $\mathbf{x_{view}}$, $\mathbf{y_{view}}$, and $\mathbf{z_{view}}$ axes with the world $\mathbf{x_w}$, $\mathbf{y_w}$, and $\mathbf{z_w}$ axes, respectively.

The viewing-coordinate origin is at world position $\mathbf{P = (x_0, y_0, z_0)}$. Therefore, the matrix for translating the viewing origin to the world origin is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the rotation transformation, we can use the unit vectors **u, v,** and **n** to form the composite rotation matrix that superimposes the viewing axes onto the world frame. This transformation matrix is

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the elements of matrix **R** are the components of the **uvn** axis vectors.

The coordinate transformation matrix is then obtained as the product of the preceding translation and rotation matrices:

$$\mathbf{M}_{WC,\,VC} = \mathbf{R} \cdot \mathbf{T} = \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u} \cdot \mathbf{P}_0 \\ v_x & v_y & v_z & -\mathbf{v} \cdot \mathbf{P}_0 \\ n_x & n_y & n_z & -\mathbf{n} \cdot \mathbf{P}_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation factors in this matrix are calculated as the vector dot product of each of the **u, v**, and **n** unit vectors with P_0, which represents a vector from the world origin to the viewing origin.

In other words, the translation factors are the negative projections of P_0 on each of the viewing-coordinate axes (the negative components of P_0 in viewing coordinates). These matrix elements are evaluated as

$$-\mathbf{u} \cdot \mathbf{P}_0 = -x_0 u_x - y_0 u_y - z_0 u_z$$
$$-\mathbf{v} \cdot \mathbf{P}_0 = -x_0 v_x - y_0 v_y - z_0 v_z$$
$$-\mathbf{n} \cdot \mathbf{P}_0 = -x_0 n_x - y_0 n_y - z_0 n_z$$

Projection Transformations:

In the next phase of the three-dimensional viewing pipeline, after the transformation to viewing coordinates, object descriptions are projected to the view plane.

Graphics packages generally support both parallel and perspective projections. In a parallel projection, coordinate positions are transferred to the view plane along parallel lines. Figure 15 illustrates a parallel projection for a straightline segment defined with endpoint coordinates P1 and P2.

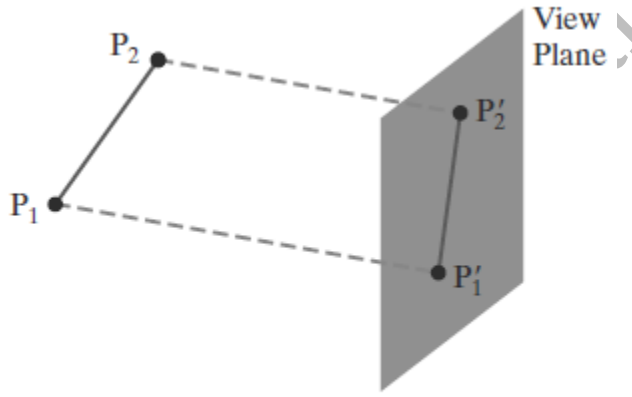

FIGURE 15
Parallel projection of a line segment onto a view plane.

A parallel projection preserves relative proportions of objects, and this is the method used in computer aided drafting and design to produce scale drawings of three-dimensional objects. All parallel lines in a scene are displayed as parallel when viewed with a parallel projection.

There are two general methods for obtaining a parallel-projection view of an object: We can project along lines that are perpendicular to the view plane, or we can project at an oblique angle to the view plane.

For a perspective projection, object positions are transformed to projection coordinates along lines that converge to a point behind the view plane. An example of a perspective projection for a straight-line segment, defined with endpoint coordinates P1 and P2, is given in Figure 16.

Unlike a parallel projection, a perspective projection does not preserve relative proportions of objects. But perspective views of a scene are more realistic because distant objects in the projected display are reduced in size.

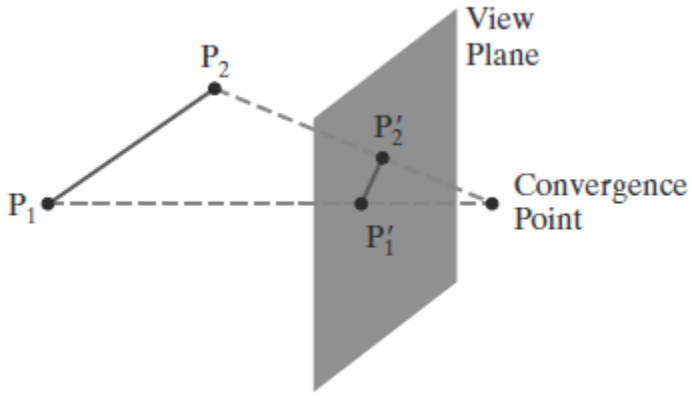

FIGURE 16
Perspective projection of a line segment onto a view plane.

Orthogonal Projections:

A transformation of object descriptions to a view plane along lines that are all parallel to the view-plane normal vector **N** is called an orthogonal projection (or, equivalently, an orthographic projection). This produces a parallel-projection transformation in which the projection lines are perpendicular to the view plane.

Orthogonal projections are most often used to produce the front, side, and top views of an object, as shown in Figure 17. Front, side, and rear orthogonal projections of an object are called **elevations**; and a top orthogonal projection is called a **plan view**.

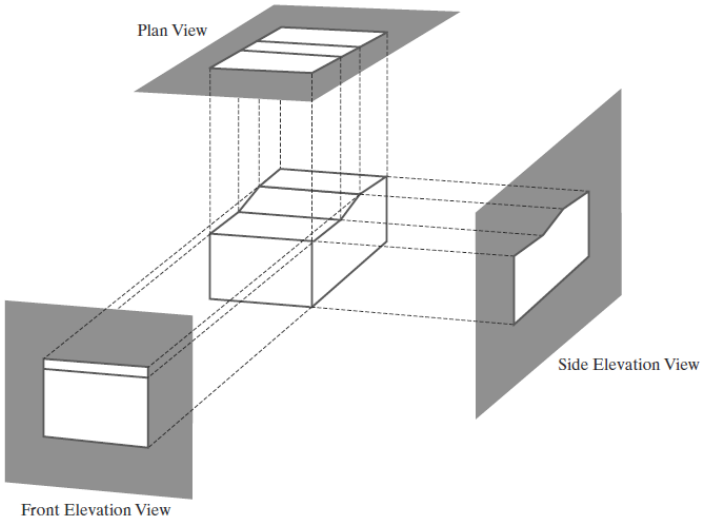

FIGURE 17
Orthogonal projections of an object, displaying plan and elevation views.

Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings.

Axonometric and Isometric Orthogonal Projections:

We can also form orthogonal projections that display more than one face of an object. Such views are called axonometric orthogonal projections.

The most commonly used axonometric projection is the isometric projection, which is generated by aligning the projection plane (or the object) so that the plane intersects each coordinate axis in which the object is defined, called the principal axes, at the same distance from the origin.

Figure 18 shows an isometric projection for a cube

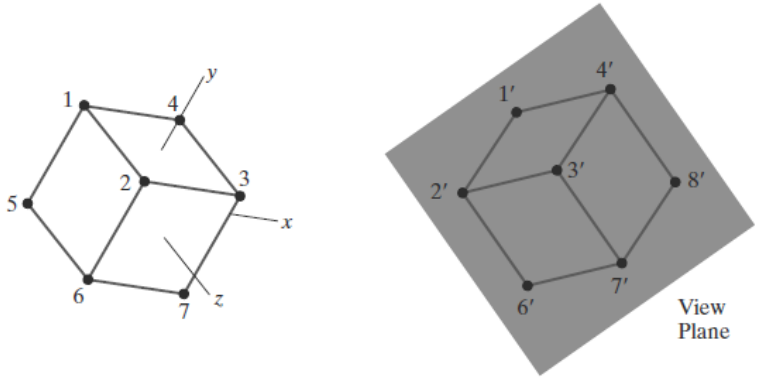

FIGURE 18
An isometric projection of a cube.

We can obtain the isometric projection shown in this figure by aligning the viewplane normal vector along a cube diagonal.

There are eight positions, one in each octant, for obtaining an isometric view. All three principal axes are foreshortened equally in an isometric projection, so that relative proportions are maintained.

Orthogonal Projection Coordinates:

With the projection direction parallel to the $\mathbf{z_{view}}$ axis, the transformation equations for an orthogonal projection are trivial. For any position (x, y, z) in viewing coordinates, as in Figure 19, the projection coordinates are

$x_p = x, y_p = y$

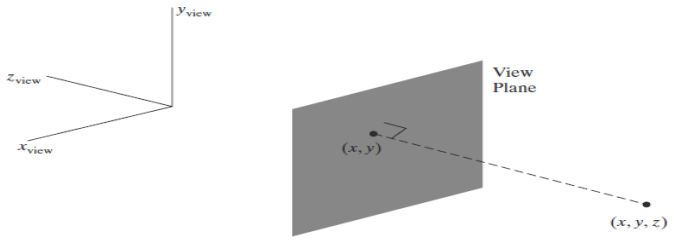

FIGURE 19
An orthogonal projection of a spatial position onto a view plane.

The z-coordinate value for any projection transformation is preserved for use in the visibility determination procedures. And each three-dimensional coordinate point in a scene is converted to a position in normalized space.

Clipping Window and Orthogonal-Projection View Volume:

In the camera analogy, the type of lens is one factor that determines how much of the scene is transferred to the film plane.

A wide-angle lens takes in more of the scene than a regular lens. For computer-graphics applications, we use the rectangular clipping window for this purpose.

As in two-dimensional viewing, graphics packages typically require that clipping rectangles be placed in specific positions.

In OpenGL, we set up a clipping window for three-dimensional viewing just as we did for two-dimensional viewing, by choosing two-dimensional coordinate positions for its lower-left and upper-right corners.

For three-dimensional viewing, the clipping window is positioned on the view plane with its edges parallel to the $\mathbf{x}_{\mathbf{view}}$ and $\mathbf{y}_{\mathbf{view}}$ axes, as shown in Figure 20.

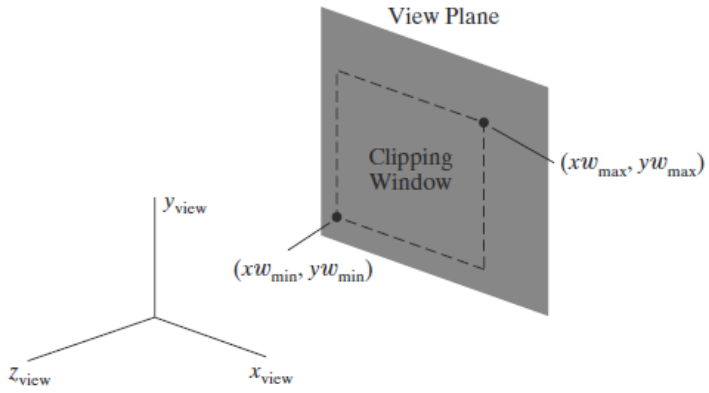

FIGURE 20
A clipping window on the view plane, with minimum and maximum coordinates given in the viewing reference system.

If we want to use some other shape or orientation for the clipping window, we must develop our own viewing procedures.

The edges of the clipping window specify the x and y limits for the part of the scene that we want to display. These limits are used to form the top, bottom, and two sides of a clipping region called the orthogonal-projection view volume. Because projection lines are perpendicular to the view plane, these four boundaries are planes that are also perpendicular to the view plane and that pass through the edges of the clipping window to form an infinite clipping region, as in Figure 21.

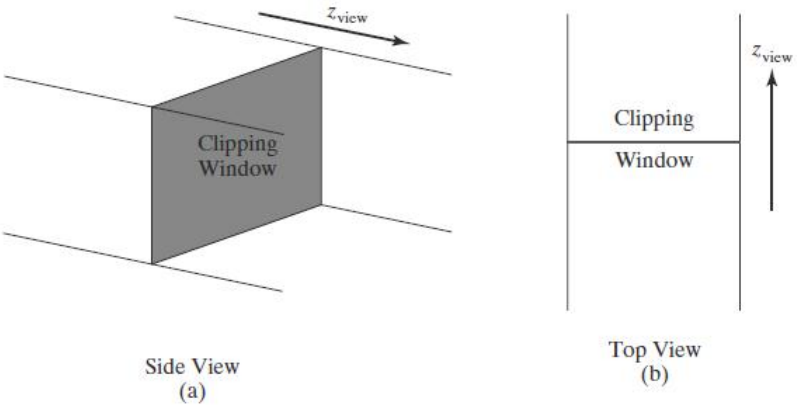

FIGURE 21
Infinite orthogonal-projection view volume.

We can limit the extent of the orthogonal view volume in the $\mathbf{z_{view}}$ direction by selecting positions for one or two additional boundary planes that are parallel to the view plane.

These two planes are called the near-far clipping planes, or the front-back clipping planes. The near and far planes allow us to exclude objects that are in front of or behind the part of the scene that we want to display.

With the viewing direction along the negative $\mathbf{z_{view}}$ axis, we usually have $\mathbf{z_{far} < z_{near}}$, so that the far plane is father out along the negative $\mathbf{z_{view}}$ axis.

When the near and far planes are specified, we obtain a finite orthogonal view volume that is a *rectangular parallelepiped*, as shown in Figure 22 along with one possible placement for the view plane.

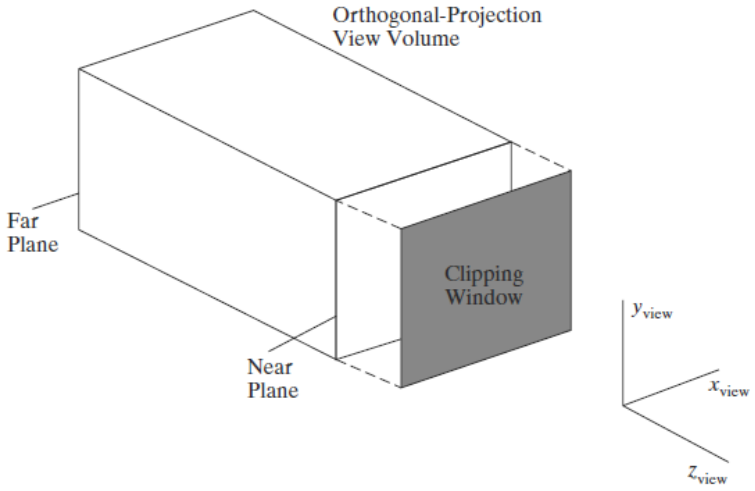

FIGURE 22
A finite orthogonal view volume with the view plane "in front" of the near plane.

Our view of the scene will then contain only those objects within the view volume, with all parts of the scene outside the view volume eliminated by the clipping algorithms.

Normalization Transformation for an Orthogonal Projection:

Using an orthogonal transfer of coordinate positions onto the view plane, we obtain the projected position of any spatial point (x, y, z) as simply (x, y). Thus, once we have established the limits for the view volume, coordinate descriptions inside this rectangular parallelepiped are the projection coordinates, and they can be mapped into a normalized view volume without any further projection processing.

Some graphics packages use a unit cube for this normalized view volume, with each of the x, y, and z coordinates normalized in the range from 0 to 1.

Another normalization-transformation approach is to use a symmetric cube, with coordinates in the range from −1 to 1.

Since screen coordinates are often specified in a left-handed reference frame (Figure 23), normalized coordinates also are often specified in a left-handed system. This allows positive distances in the viewing direction to be directly interpreted as distances from the screen (the viewing plane).

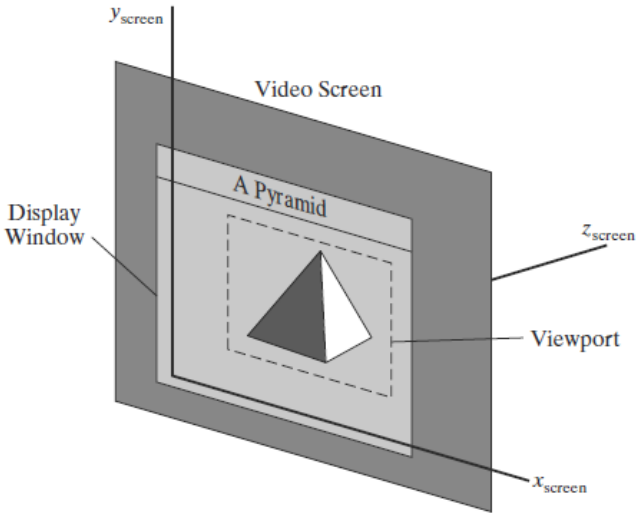

FIGURE 23
A left-handed screen-coordinate reference frame.

Thus, we can convert projection coordinates into positions within a left-handed normalized-coordinate reference frame, and these coordinate positions will then be transferred to left handed screen coordinates by the viewport transformation.

To illustrate the normalization transformation, we assume that the orthogonal-projection view volume is to be mapped into the symmetric normalization cube within a left-handed reference frame.

Also, z-coordinate positions for the near and far planes are denoted as $\mathbf{z_{near}}$ and $\mathbf{z_{far}}$, respectively.

Figure 24 illustrates this normalization transformation. Position (x_{min}, y_{min}, z_{near}) is mapped to the normalized position (−1, −1, −1), and position (x_{max}, y_{max}, z_{far}) is mapped to (1, 1, 1).

Transforming the rectangular-parallelepiped view volume to a normalized cube is similar to the methods for converting the clipping window into the normalized symmetric square.

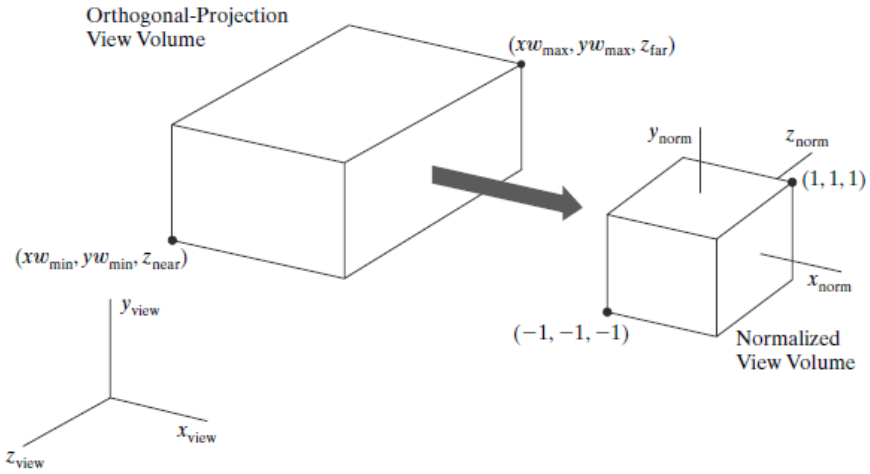

FIGURE 24
Normalization transformation from an orthogonal-projection view volume to the symmetric normalization cube within a left-handed reference frame.

The normalization transformation for the orthogonal view volume is

$$\mathbf{M}_{ortho,norm} = \begin{bmatrix} \frac{2}{xw_{max} - xw_{min}} & 0 & 0 & -\frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \frac{2}{yw_{max} - yw_{min}} & 0 & -\frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & \frac{-2}{z_{near} - z_{far}} & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is multiplied on the right by the composite viewing transformation **R·T** to produce the complete transformation from world coordinates to normalized orthogonal-projection coordinates.

At this stage of the viewing pipeline, all device-independent coordinate transformations are completed and can be concatenated into a single composite matrix.

Thus, the clipping procedures are most efficiently performed following the normalization transformation.

After clipping, procedures for visibility testing, surface rendering, and the viewport transformation can be applied to generate the final screen display of the scene.

Perspective Projections:

Although a parallel-projection view of a scene is easy to generate and preserves relative proportions of objects, it does not provide a realistic representation.

To simulate a camera picture, we need to consider that reflected light rays from the objects in a scene follow converging paths to the camera film plane.

We can approximate this geometric-optics effect by projecting objects to the view plane along converging paths to a position called the projection reference point (or center of projection). Objects are then displayed with foreshortening effects, and projections of distant objects are smaller than the projections of objects of the same size that are closer to the view plane (Figure 33).

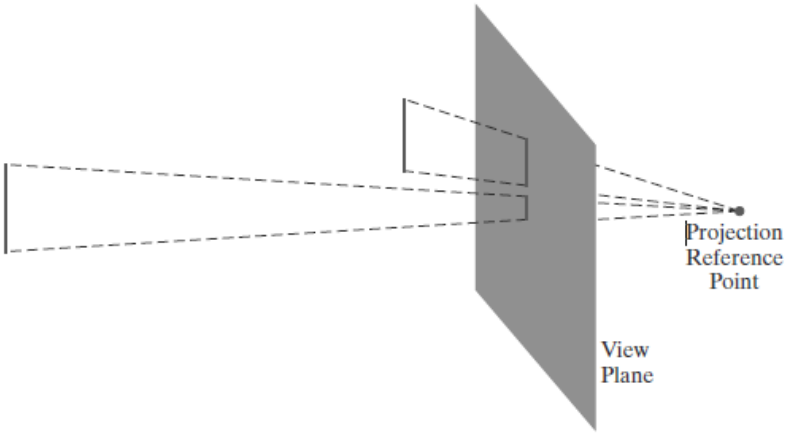

FIGURE 33
A perspective projection of two equal-length line segments at different distances from the view plane.

Perspective-Projection Transformation Coordinates:

We can sometimes select the projection reference point as another viewing parameter in a graphics package, but some systems place this convergence point at a fixed position, such as at the view point. Figure 34 shows the projection path of a spatial position **(x, y, z)** to a general projection reference point at **(x_{prp}, y_{prp}, z_{prp})**.

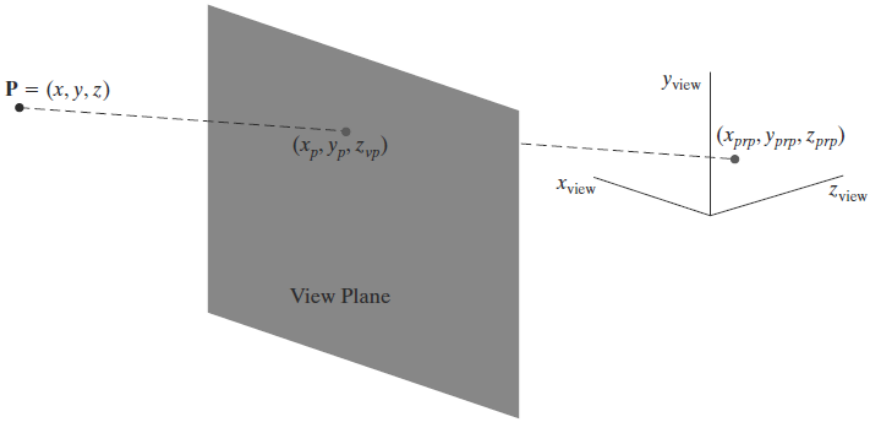

FIGURE 34
A perspective projection of a point **P** with coordinates (x, y, z) to a selected projection reference point. The intersection position on the view plane is (x_p, y_p, z_{vp}).

The projection line intersects the view plane at the coordinate position **(x_p, y_p, z_{vp})**, where **z_{vp}** is some selected position for the view plane on the **z_{view}** axis.

We can write equations describing coordinate positions along this perspective-projection line in parametric form as

$$x' = x - (x - x_{prp})u$$
$$y' = y - (y - y_{prp})u \qquad 0 \le u \le 1$$
$$z' = z - (z - z_{prp})u$$

Coordinate position (x^1, y^1, z^1) represents any point along the projection line.

When u = 0, we are at position P = (x, y, z). At the other end of the line, u = 1 and we have the projection reference-point coordinates **(x_{prp}, y_{prp}, z_{prp})**.

On the view plane, $\mathbf{z^1 = z_{vp}}$ and we can solve the z^1 equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Substituting this value of u into the equations for x^1 and y^1, we obtain the general perspective-transformation equations

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

Perspective-Projection Equations: Special Cases

Various restrictions are often placed on the parameters for a perspective projection. Depending on a particular graphics package, positioning for either the projection reference point or the view plane may not be completely optional.

To simplify the perspective calculations, the projection reference point could be limited to positions along the $\mathbf{z_{view}}$ axis, then

1. $x_{prp} = y_{prp} = 0$:

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right)$$

Sometimes the projection reference point is fixed at the coordinate origin, and

2. $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$:

$$x_p = x\left(\frac{z_{vp}}{z}\right), \qquad y_p = y\left(\frac{z_{vp}}{z}\right)$$

If the view plane is the **uv** plane and there are no restrictions on the placement of the projection reference point, then we have

3. $z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) - x_{prp}\left(\frac{z}{z_{prp} - z}\right)$$

$$y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) - y_{prp}\left(\frac{z}{z_{prp} - z}\right)$$

With the uv plane as the view plane and the projection reference point on the z_{view} axis, the perspective equations are

4. $x_{prp} = y_{prp} = z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right)$$

With the scene between the view plane and the projection point, objects are simply enlarged as they are projected away from the viewing position onto the view plane.
Perspective effects also depend on the distance between the projection reference point and the view plane.

Vanishing Points for Perspective Projections:

When a scene is projected onto a view plane using a perspective mapping, lines that are parallel to the view plane are projected as parallel lines. But any parallel lines in the scene that are not parallel to the view plane are projected into converging lines.

The point at which a set of projected parallel lines appears to converge is called a vanishing point.

Each set of projected parallel lines has a separate vanishing point. For a set of lines that are parallel to one of the principal axes of an object, the vanishing point is referred to as a principal vanishing point.

We control the number of principal vanishing points (one, two, or three) with the orientation of the projection plane, and perspective projections are accordingly classified as one-point, two-point, or three-point projections.

The number of principal vanishing points in a projection is equal to the number of principal axes that intersect the view plane.

Figure 37 illustrates the appearance of one-point and two point perspective projections for a cube.

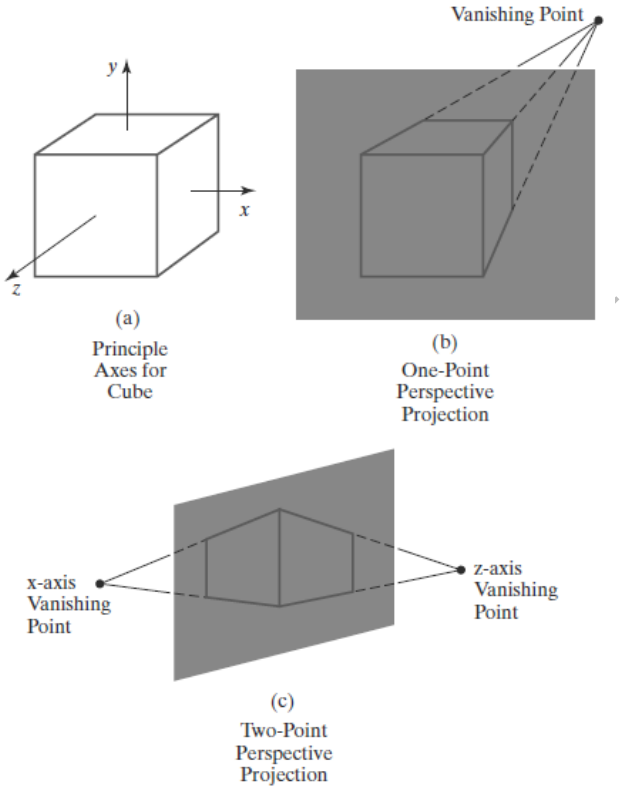

FIGURE 37
Principal vanishing points for perspective-projection views of a cube. When the cube in (a) is projected to a view plane that intersects only the *z* axis, a single vanishing point in the *z* direction (b) is generated. When the cube is projected to a view plane that intersects both the *z* and *x* axes, two vanishing points (c) are produced.

Perspective-Projection View Volume:

We create a view volume by specifying the position of a rectangular clipping window on the view plane. But now the bounding planes for the view volume are not parallel, because the projection lines are not parallel.

The bottom, top, and sides of the view volume are planes through the window edges that all intersect at the projection reference point. This forms a view volume that is an infinite rectangular pyramid with its apex at the center of projection (Figure 38).

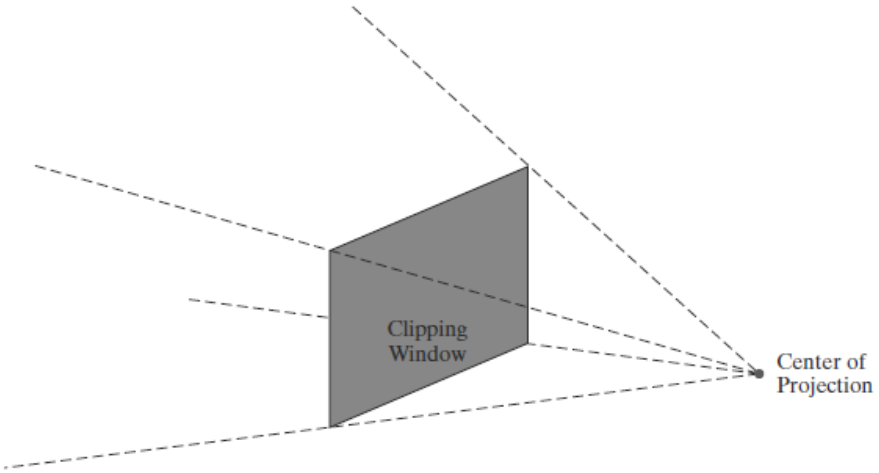

FIGURE 38
An infinite, pyramid view volume for a perspective projection.

All objects outside this pyramid are eliminated by the clipping routines. A perspective-projection view volume is often referred to as a pyramid of vision because it approximates the cone of vision of our eyes or a camera.

The displayed view of a scene includes only those objects within the pyramid, just as we cannot see objects beyond our peripheral vision, which are outside the cone of vision.

By adding near and far clipping planes that are perpendicular to the $\mathbf{z_{view}}$ axis (and parallel to the view plane), we chop off parts of the infinite, perspective projection view volume to form a **truncated pyramid, or frustum**, view volume.

Figure 39 illustrates the shape of a finite, perspective-projection view volume with a view plane that is placed between the near clipping plane and the projection reference point.

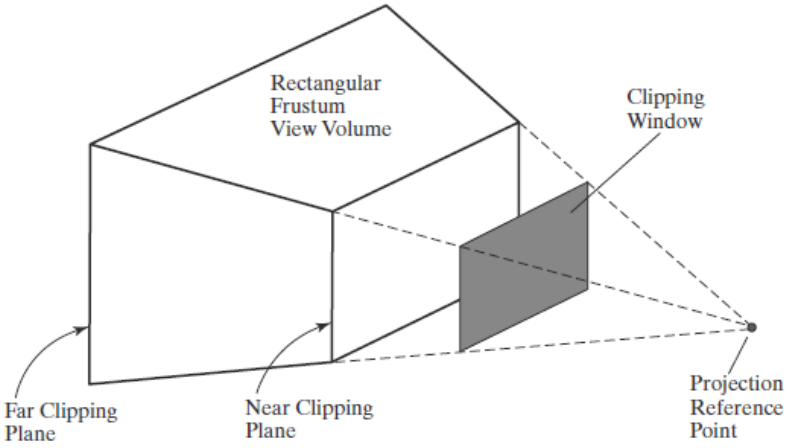

FIGURE 39
A perspective-projection frustum view volume with the view plane "in front" of the near clipping plane.

Usually, both the near and far clipping planes are on the same side of the projection reference point, with the far plane farther from the projection point than the near plane along the viewing direction. And, as in a parallel projection, we can use the near and far planes simply to enclose the scene to be viewed.

But with a perspective projection, we could also use the near clipping plane to take out large objects close to the view plane that could project into unrecognizable shapes within the clipping window.

Similarly, the far clipping plane could be used to cut out objects far from the projection reference point that might project to small blots on the view plane.

Perspective-Projection Transformation Matrix:

Unlike a parallel projection, we cannot directly use the coefficients of the x and y coordinates to form the perspective-projection matrix elements, because the denominators of the coefficients are functions of the z coordinate.

But we can use a three-dimensional, homogeneous-coordinate representation to express the perspective-projection equations in the form

$$x_p = \frac{x_h}{h}, \qquad y_p = \frac{y_h}{h}$$

where the homogeneous parameter has the value

$$h = z_{prp} - z$$

$$x_h = x(z_{prp} - z_{vp}) + x_{prp}(z_{vp} - z)$$
$$y_h = y(z_{prp} - z_{vp}) + y_{prp}(z_{vp} - z)$$

The perspective-projection transformation of a viewing-coordinate position is then accomplished in two steps.
First, we calculate the homogeneous coordinates using the perspective-transformation matrix:

$$\mathbf{P}_h = \mathbf{M}_{\text{pers}} \cdot \mathbf{P}$$

where $\mathbf{P_h}$ is the column-matrix representation of the homogeneous point (x_h, y_h, z_h, h) and **P** is the column-matrix representation of the coordinate position (x, y, z, 1).

Second, after other processes have been applied, such as the normalization transformation and clipping routines, homogeneous coordinates are divided by parameter **h** to obtain the true transformation-coordinate positions.
The following matrix gives one possible way to formulate a perspective-projection matrix.

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} z_{prp} - z_{vp} & 0 & -x_{prp} & x_{prp}z_{prp} \\ 0 & z_{prp} - z_{vp} & -y_{prp} & y_{prp}z_{prp} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{prp} \end{bmatrix}$$

Parameters s_z and t_z are the scaling and translation factors for normalizing the projected values of z-coordinates. Specific values for s_z and t_z depend on the normalization range we select.

Symmetric Perspective-Projection Frustum:

The line from the projection reference point through the center of the clipping window and on through the view volume is the centerline for a perspective projection frustum.
If this centerline is perpendicular to the view plane, we have a symmetric frustum (with respect to its centerline) as in Figure 40.

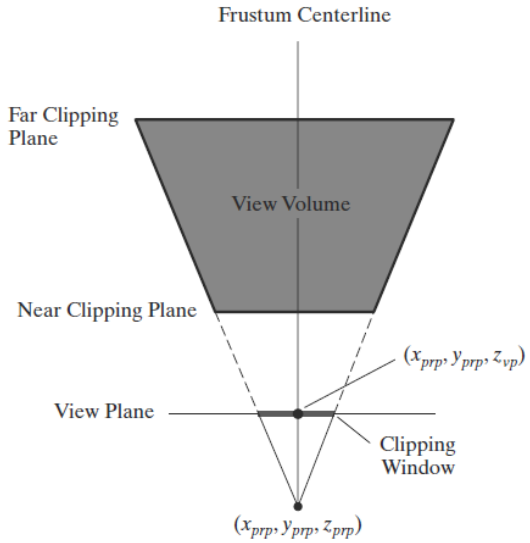

FIGURE 40
A symmetric perspective-projection frustum view volume, with the view plane between the projection reference point and the near clipping plane. This frustum is symmetric about its centerline when viewed from above, below, or either side.

Because the frustum centerline intersects the view plane at the coordinate location (x_{prp}, y_{prp}, z_{vp}), we can express the corner positions for the clipping window in terms of the window dimensions:

$$xw_{\min} = x_{prp} - \frac{\text{width}}{2}, \qquad xw_{\max} = x_{prp} + \frac{\text{width}}{2}$$

$$yw_{\min} = y_{prp} - \frac{\text{height}}{2}, \qquad yw_{\max} = y_{prp} + \frac{\text{height}}{2}$$

Therefore, we could specify a symmetric perspective-projection view of a scene using the width and height of the clipping window instead of the window coordinates. This uniquely establishes the position of the clipping window, because it is symmetric about the x and y coordinates of the projection reference point.

Another way to specify a symmetric perspective projection is to use parameters that approximate the properties of a camera lens.

A photograph is produced with a symmetric perspective projection of a scene onto the film plane. Reflected light rays from the objects in a scene are collected on the film plane from within the "cone of vision" of the camera. This cone of vision can be referenced with a field-of-view angle, which is a measure of the size of the camera lens.

A large field-of-view angle, for example, corresponds to a wide-angle lens. In computer graphics, the cone of vision is approximated with a symmetric frustum, and we can use a field-of-view angle to specify an angular size for the frustum.

The field-of-view angle is the angle between the top clipping plane and the bottom clipping plane of the frustum, as shown in Figure 41.

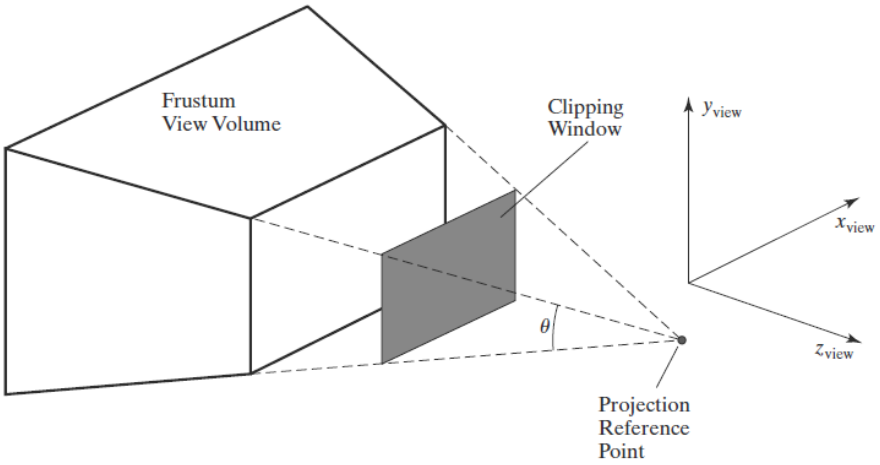

FIGURE 41
Field-of-view angle θ for a symmetric perspective-projection view volume, with the clipping window between the near clipping plane and the projection reference point.

For a given projection reference point and view-plane position, the field-of view angle determines the height of the clipping window (Figure 42), but not the width.

We need an additional parameter to define completely the clipping window dimensions, and this second parameter could be either the window width or the aspect ratio (width/height) of the clipping window.

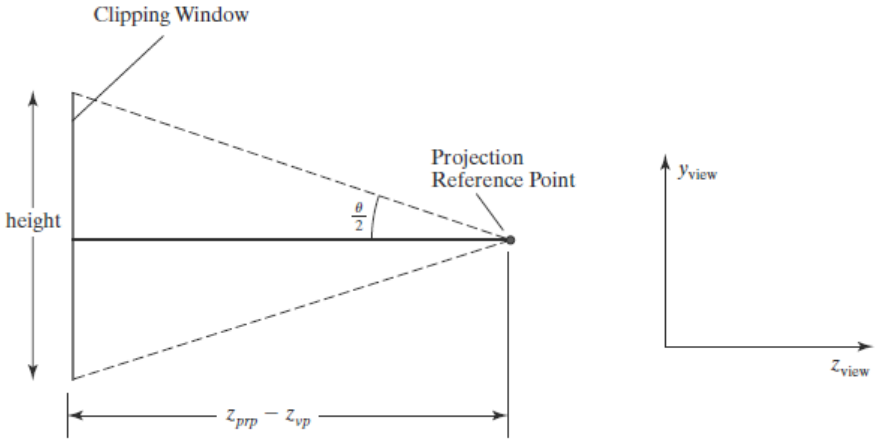

FIGURE 42
Relationship between the field-of-view angle θ, the height of the clipping window, and the distance between the projection reference point and the view plane.

From the right triangles in the diagram of Figure 42, we see that

$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{height}/2}{z_{prp} - z_{vp}}$$

\so that the clipping-window height can be calculated as

$$\text{height} = 2(z_{prp} - z_{vp}) \tan\left(\frac{\theta}{2}\right)$$

Therefore, the diagonal elements with the value $\mathbf{z_{prp} - z_{vp}}$ in matrix could be replaced by either of the following two expressions.

$$z_{prp} - z_{vp} = \frac{\text{height}}{2} \cot\left(\frac{\theta}{2}\right)$$

$$= \frac{\text{width} \cdot \cot(\theta/2)}{2 \cdot \text{aspect}}$$

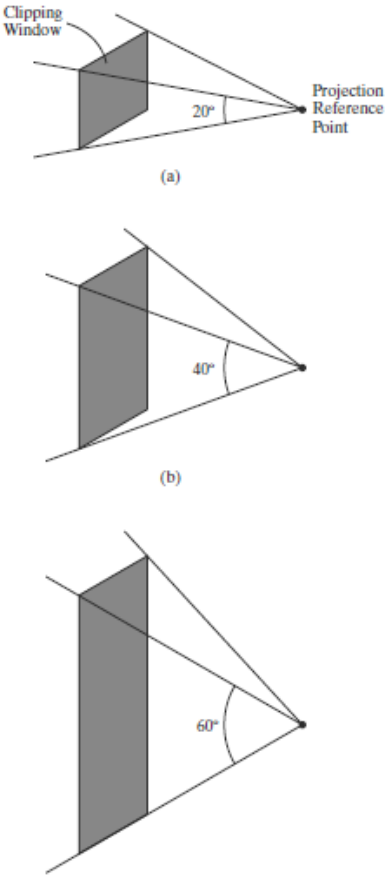

FIGURE 43
Increasing the size of the field-of-view angle increases the height of the clipping window and increases the perspective-projection foreshortening.

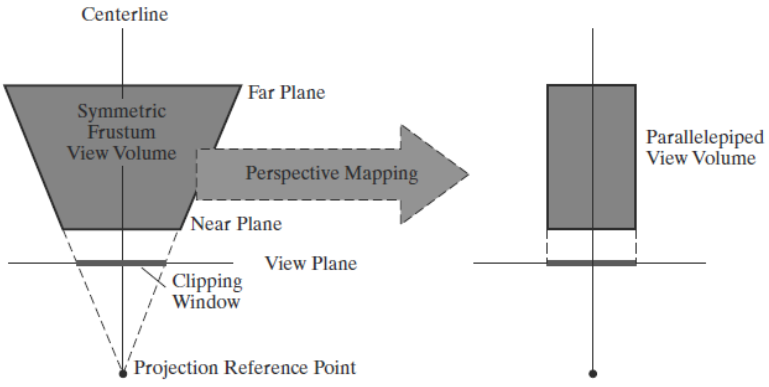

FIGURE 44
A symmetric frustum view volume is mapped to an orthogonal parallelepiped by a perspective-projection transformation.

Normalized Perspective-Projection Transformation Coordinates:

When we divide the homogeneous coordinates by the homogeneous parameter **h**, we obtain the actual projection coordinates, which are orthogonal-projection coordinates. Thus, this perspective projection transforms all points within the frustum view volume to positions within a rectangular parallelepiped view volume. The final step in the perspective transformation process is to map this parallelepiped to a normalized view volume.

We follow the same normalization procedure that we used for a parallel projection. The transformed frustum view volume, which is a rectangular parallelepiped, is mapped to a symmetric normalized cube within a left-handed reference frame (Figure 46).

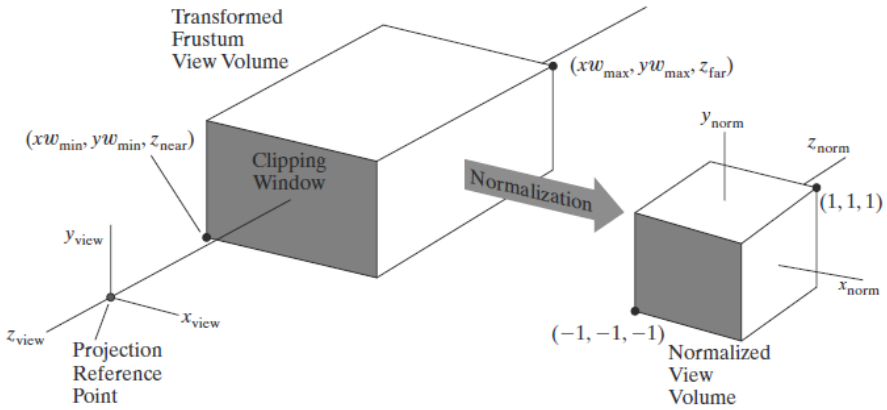

FIGURE 46
Normalization transformation from a transformed perspective-projection view volume (rectangular parallelepiped) to the symmetric normalization cube within a left-handed reference frame, with the near clipping plane as the view plane and the projection reference point at the viewing-coordinate origin.

We need to determine the values for these parameters when we transform to the symmetric normalization cube. Also, we need to determine the normalization transformation parameters for x and y coordinates.

Because the centerline of the rectangular parallelepiped view volume is now the $\mathbf{z_{view}}$ axis, no translation is needed in the x and y normalization transformations: We require only the x and y scaling parameters relative to the coordinate origin.

The scaling matrix for accomplishing the xy normalization is

$$\mathbf{M}_{xy\,\text{scale}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The normalization matrix for a perspective-projection transformation is given as

$$\mathbf{M}_{\text{normpers}} = \mathbf{M}_{xy\,\text{scale}} \cdot \mathbf{M}_{\text{obliquepers}}$$

$$= \begin{bmatrix} -z_{\text{near}} s_x & 0 & s_x \dfrac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} s_y & s_y \dfrac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

From this transformation, we obtain the homogeneous coordinates:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M}_{\text{normpers}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

And the projection coordinates are

$$x_p = \frac{x_h}{h} = \frac{-z_{\text{near}} s_x x + s_x (x w_{\min} + x w_{\max})/2}{-z}$$
$$y_p = \frac{y_h}{h} = \frac{-z_{\text{near}} s_y y + s_y (y w_{\min} + y w_{\max})/2}{-z}$$
$$z_p = \frac{z_h}{h} = \frac{s_z z + t_z}{-z}$$

To normalize this perspective transformation, we want the projection coordinates to be (x_p, y_p, z_p) = (−1, −1, −1) when the input coordinates are (x, y, z) = (x_{wmin}, y_{wmin}, z_{near}), and we want the projection coordinates to be (x_p, y_p, z_p) = (1, 1, 1) when the input coordinates are (x, y, z) = (x_{wmax}, y_{wmax}, z_{far}).

Therefore, when we solve the above equations for the normalization parameters using these conditions, we obtain

$$s_x = \frac{2}{x w_{\max} - x w_{\min}}, \qquad s_y = \frac{2}{y w_{\max} - y w_{\min}}$$
$$s_z = \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}}, \qquad t_z = \frac{2\, z_{\text{near}}\, z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}}$$

And the elements of the normalized transformation matrix for a general perspective-projection are

$$\mathbf{M}_{\text{normpers}} = \begin{bmatrix} \frac{-2z_{\text{near}}}{x w_{\max} - x w_{\min}} & 0 & \frac{x w_{\max} + x w_{\min}}{x w_{\max} - x w_{\min}} & 0 \\ 0 & \frac{-2z_{\text{near}}}{y w_{\max} - y w_{\min}} & \frac{y w_{\max} + y w_{\min}}{y w_{\max} - y w_{\min}} & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & -\frac{2 z_{\text{near}} z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

If the perspective-projection view volume was originally specified as a symmetric frustum, we can express the elements of the normalized perspective transformation in terms of the field-of view angle and the dimensions of the clipping window.

Thus, with the projection reference point at the origin and the view plane at the position of the near clipping plane, we have

$$\mathbf{M}_{\text{normsymmpers}} = \begin{bmatrix} \frac{\cot\left(\frac{\theta}{2}\right)}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & -\frac{2z_{\text{near}}\, z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The complete transformation from world coordinates to normalized perspective-projection coordinates is the composite matrix formed by concatenating this perspective matrix on the left of the viewing-transformation product **R·T**.

Next, the clipping routines can be applied to the normalized view volume. The remaining tasks are visibility determination, surface rendering, and the transformation to the viewport.

The Viewport Transformation and Three-Dimensional Screen Coordinates

Once we have completed the transformation to normalized projection coordinates, clipping can be applied efficiently to the symmetric cube (or the unit cube).

Following the clipping procedures, the contents of the normalized view volume can be transferred to screen coordinates.

For the x and y positions in the normalized clipping window, this procedure is the same as the two-dimensional viewport transformation. But positions throughout the three-dimensional view volume also have a depth (z coordinate), and we need to retain this depth information for the visibility testing and surface-rendering algorithms. So we can now think of the viewport transformation as a mapping to three-dimensional screen coordinates.

We can adapt that matrix to three-dimensional applications by including parameters for the transformation of z values to screen coordinates.

Often the normalized z values within the symmetric cube are renormalized on the range from 0 to 1.0. This allows the video screen to be referenced as z = 0, and depth processing can be conveniently carried out over the unit interval from 0 to 1.

If we include this z renormalization, the transformation from the normalized view volume to three dimensional screen coordinates is

$$\mathbf{M}_{\text{normviewvol,3D screen}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & 0 & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In normalized coordinates, the $\mathbf{z_{norm}} = -1$ face of the symmetric cube corresponds to the clipping-window area. And this face of the normalized cube is mapped to the rectangular viewport, which is now referenced at $\mathbf{z_{screen}} = 0$. Thus, the lower-left corner of the viewport screen area is at position $\mathbf{(xv_{min}, yv_{min}, 0)}$ and the upper-right corner is at position $\mathbf{(xv_{max}, yv_{max}, 0)}$.

Each xy position on the viewport corresponds to a position in the refresh buffer, which contains the color information for that point on the screen. And the depth value for each screen point is stored in another buffer area, called the depth buffer.

We position the rectangular viewport on the screen just as we did for two dimensional applications. The lower-left corner of the viewport is usually placed at a coordinate position specified relative to the lower-left corner of the display window. And object proportions are maintained if we set the aspect ratio of this viewport area to be the same as the clipping window.

OpenGL Three-Dimensional Viewing Functions

The OpenGL Utility library (GLU) includes a function for specifying the three dimensional viewing parameters and another function for setting up a symmetric perspective-projection transformation.

Other functions, such as those for an orthogonal projection, an oblique perspective projection, and the viewport transformation, are contained in the basic OpenGL library.

In addition, GLUT functions are available for defining and manipulating display windows.

OpenGL Viewing-Transformation Function

When we designate the viewing parameters in OpenGL, a matrix is formed and concatenated with the current modelview matrix. Consequently, this viewing matrix is combined with any geometric transformations we may have also specified.

This composite matrix is then applied to transform object descriptions in world coordinates to viewing coordinates. We set the modelview mode with the statement:

glMatrixMode (GL_MODELVIEW);

Viewing parameters are specified with the following GLU function, which is in the OpenGL Utility library because it invokes the translation and rotation routines in the basic OpenGL library.

gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

Values for all parameters in this function are to be assigned double-precision,floating-point values. This function designates the origin of the viewing reference frame as the world-coordinate position $\mathbf{P_0} = \mathbf{(x_0, y_0, z_0)}$, the reference position as $\mathbf{P_{ref}} = \mathbf{(xref, yref, zref)}$, and the view-up vector as $\mathbf{V} = \mathbf{(Vx, Vy, Vz).}$

The positive z_{view} axis for the viewing frame is in the direction $\mathbf{N} = \mathbf{P_0} - \mathbf{P_{ref}}$. Because the viewing direction is along the $-\mathbf{z_{view}}$ axis, the reference position $\mathbf{P_{ref}}$ is also referred to as the "look-at point." This is usually taken to be some position in the center of the scene that we can use as a reference for specifying the projection parameters.

We can think of the reference position as the point at which we want to aim a camera that is located at the viewing origin.

The up orientation for the camera is designated with vector V, which is adjusted to a direction perpendicular to N.

If we do not invoke the gluLookAt function, the default OpenGL viewing parameters are

$$\mathbf{P_0 = (0, 0, 0)}$$
$$\mathbf{P_{ref} = (0, 0, -1)}$$
$$\mathbf{V = (0, 1, 0)}$$

For these default values, the viewing reference frame is the same as the world frame, with the viewing direction along the negative z_{world} axis.

In many applications, we can conveniently use the default values for the viewing parameters.

OpenGL Orthogonal-Projection Function

Projection matrices are stored in the OpenGL projection mode. So, to set up a projection-transformation matrix, we must first invoke that mode with the statement

glMatrixMode (GL_PROJECTION);

Then, when we issue any transformation command, the resulting matrix will be concatenated with the current projection matrix.

Orthogonal-projection parameters are chosen with the function

glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

All parameter values in this function are to be assigned double-precision, floating point numbers.

We use glOrtho to select the clipping-window coordinates and the distances to the near and far clipping planes from the viewing origin.

There is no option in OpenGL for the placement of the view plane. The near clipping plane is always also the view plane, and therefore the clipping window is always on the near plane of the view volume.

Function glOrtho generates a parallel projection that is perpendicular to the view plane (the near clipping plane). Thus, this function creates a finite orthogonal-projection view volume for the specified clipping planes and clipping window.

In OpenGL, the near and far clipping planes are not optional; they must always be specified for any projection transformation.

Parameters $\mathbf{d_{near}}$ and $\mathbf{d_{far}}$ denote distances in the negative $\mathbf{z_{view}}$ direction from the viewing-coordinate origin.

For example, if $\mathbf{d_{far}} = 55.0$, then the far clipping plane is at the coordinate position $z_{far} = -55.0$.

A negative value for either parameter denotes a distance "behind" the viewing origin, along the positive z_{view} axis. We can assign any values (positive, negative, or zero) to these parameters, so long as $d_{near} < d_{far}$.

The resulting view volume for this projection transformation is a rectangular parallelepiped.

Coordinate positions within this view volume are transformed to locations within the symmetric normalized cube in a left-handed reference frame, with $z_{near} = -d_{near}$ and $z_{far} = -d_{far}$.

Default parameter values for the OpenGL orthogonal-projection function are ±1, which produce a view volume that is a symmetric normalized cube in the right-handed viewing-coordinate system. This default is equivalent to issuing the statement

glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

The default clipping window is thus a symmetric normalized square, and the default view volume is a symmetric normalized cube with $z_{near} = 1.0$ (behind the viewing position) and $z_{far} = -1.0$.

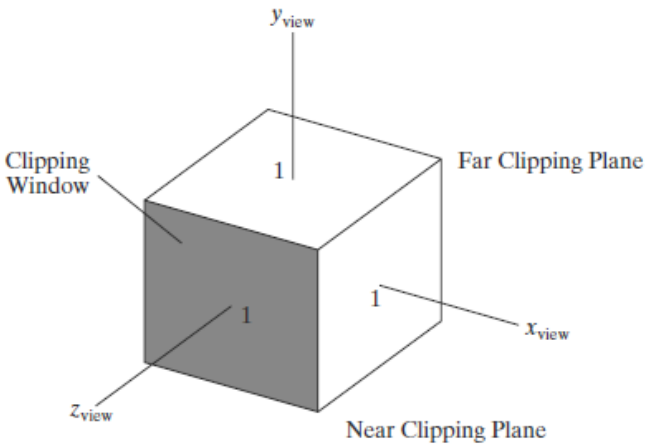

FIGURE 47
Default orthogonal-projection view volume. Coordinate extents for this symmetric cube are from -1 to $+1$ in each direction. The near clipping plane is at $z_{near} = 1$, and the far clipping plane is at $z_{far} = -1$.

Figure 47 shows the appearance and position of the default orthogonal-projection view volume.

For two-dimensional applications, we used the gluOrtho2D function to set up the clipping window.
We could also have used the glOrtho function to specify the clippingwindow, as long as parameters d_{near} and d_{far} were assigned values that were on opposite sides of the coordinate origin.
In fact, a call to gluOrtho2D is equivalent to a call to glOrtho with $d_{near} = -1.0$ and $d_{far} = 1.0$.

OpenGL Symmetric Perspective-Projection Function:
A symmetric, perspective-projection, frustum view volume is set up with the GLU function

gluPerspective (theta, aspect, dnear, dfar);

with each of the four parameters assigned a double-precision, floating-point number.

The first two parameters define the size and position of the clipping window on the near plane, and the second two parameters specify the distances from the view point (coordinate origin) to the near and far clipping planes.
Parameter theta represents the field-of-view angle, which is the angle between the top and bottom clipping planes (Figure 41). This angle can be assigned any value from 0◦ to 180◦. Parameter aspect is assigned a value for the aspect ratio (width/height) of the clipping window.

For a perspective projection in OpenGL, the near and far clipping planes must always be somewhere along the negative z_{view} axis; neither can be "behind" the viewing position. This restriction does not apply to an orthogonal projection, but it precludes the inverted perspective projection of an object when the view plane is behind the view point. Therefore, both d_{near} and d_{far} must be assigned positive numerical values, and the positions of the near and far planes are calculated as $z_{near} = -d_{near}$ and $z_{far} = -d_{far}$.

If we do not specify a projection function, our scene is displayed using the default orthogonal projection. In this case, the view volume is the symmetric normalized cube shown in Figure 47

The frustum view volume set up by the gluPerspective function is symmetric about the negative z_{view} axis.

OpenGL General Perspective-Projection Function

We can use the following function to specify a perspective projection that has either a symmetric frustum view volume or an oblique frustum view volume.

glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

All parameters in this function are assigned double-precision, floating-point numbers.

As in the other viewing-projection functions, the near plane is the view plane and the projection reference point is at the viewing position (coordinate origin). This function has the same parameters as the orthogonal, parallel-projection function, but now the near and far clipping-plane distances must be positive.

The first four parameters set the coordinates for the clipping window on the near plane, and the last two parameters specify the distances from the coordinate origin to the near and far clipping planes along the negative z_{view} axis.

Locations for the near and far planes are calculated as $z_{near} = -d_{near}$ and $z_{far} = -d_{far}$.

The clipping window can be specified anywhere on the near plane. If we select the clipping window coordinates so that $xw_{min} = -xw_{max}$ and $yw_{min} = -yw_{max}$, we obtain a symmetric frustum (about the negative z_{view} axis as its centerline).

Again, if we do not explicitly invoke a projection command, OpenGL applies the default orthogonal projection to the scene. The view volume in this case is the symmetric cube (Figure 47).

OpenGL Viewports and Display Windows

After the clipping routines have been applied in normalized coordinates, the contents of the normalized clipping window, along with the depth information, are transferred to three-dimensional screen coordinates.

The color value for each xy position on the viewport is stored in the refresh buffer (color buffer), and thedepth information for each xy position is stored in the depth buffer.

A rectangular viewport is defined with the follow

glViewport (xvmin, yvmin, vpWidth, vpHeight);

The first two parameters in this function specify the integer screen position of the lower-left corner of the viewport relative to the lower-left corner of the display window. And the last two parameters give the integer width and height of the viewport.

To maintain the proportions of objects in a scene, we set the aspect ratio of the viewport equal to the aspect ratio of the clipping window.

Classification of Visible-Surface Detection Algorithms:

We can broadly classify visible-surface detection algorithms according to whether they deal with the object definitions or with their projected images.

These two approaches are called **object-space** methods and **image-space** methods, respectively.

An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible.

In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane.

Most visible-surface algorithms use image-space methods, although object-space methods can be used effectively to locate visible surfaces in some cases.

Although there are major differences in the basic approaches taken by the various visible-surface detection algorithms, most use sorting and coherence methods to improve performance.

Sorting is used to facilitate depth comparisons by ordering the individual surfaces in a scene according to their distance from the view plane.

Coherence methods are used to take advantage of regularities in a scene. An individual scan line can be expected to contain intervals (runs) of constant pixel intensities, and scan-line patterns often change little from one line to the next. Animation frames contain changes only in the vicinity of moving objects. And constant relationships can often be established between the objects in a scene.

Back-Face Detection:

A fast and simple object-space method for locating the back faces of a polyhedron is based on front-back tests. A point (x, y, z) is behind a polygon surface if

$$Ax + By + Cz + D < 0$$

where A, B,C, and D are the plane parameters for the polygon. When this position is along the line of sight to the surface, we must be looking at the back of the polygon. Therefore, we could use the viewing position to test for back faces.

We can simplify the back-face test by considering the direction of the normal vector N for a polygon surface.

If $\mathbf{V}_{view}$ is a vector in the viewing direction from our camera position, as shown in Figure 1, then a polygon is a back face if

$$\mathbf{V}_{view} \cdot \mathbf{N} > 0$$

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing axis, then we need to consider only the z component of the normal vector N.

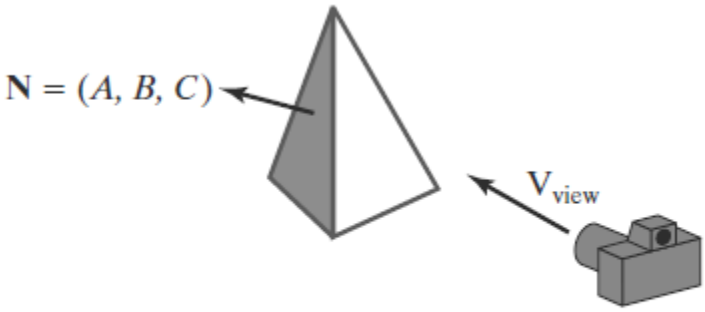

FIGURE 1
A surface normal vector **N** and the viewing-direction vector $\mathbf{V}_{view}$.

In a right-handed viewing system with the viewing direction along the negative $\mathbf{z}_v$ axis (Figure 2), a polygon is a back face if the z component, C, of its normal vector N satisfies C < 0. Also, we cannot see any face whose normal has z component C = 0, because our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a z component value that satisfies the inequality

$$C \leq 0$$

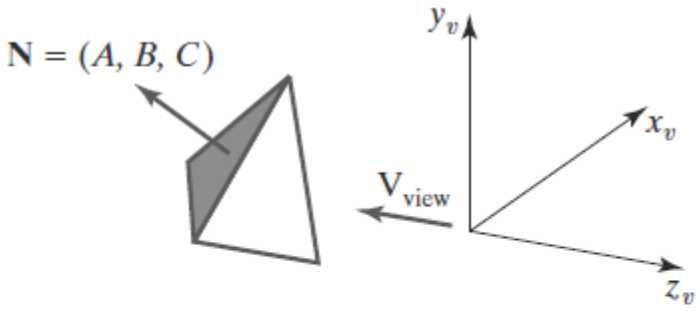

FIGURE 2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

Similar methods can be used in packages that employ a left-handed viewing system. In these packages, plane parameters A, B, C, and D can be calculated from polygon vertex coordinates specified in a clockwise direction (instead of the counterclockwise direction used in

a right-handed system). Inequality then remains a valid test for points behind the polygon. Also, back faces have normal vectors that point away from the viewing position and are identified by C ≥ 0 when the viewing direction is along the positive z_v axis.

By examining parameter C for the different plane surfaces describing an object, we can immediately identify all the back faces. For a single convex polyhedron, such as the pyramid in Figure 2, this test identifies all the hidden surfaces in the scene, because each surface is either completely visible or completely hidden.

In general, back-face removal can be expected to eliminate about half of the polygon surfaces in a scene from further visibility tests.

Depth-Buffer Method:

A commonly used image-space approach for detecting visible surfaces is the depth-buffer method, which compares surface depth values throughout a scene for each pixel position on the projection plane.

Each surface of a scene is processed separately, one pixel position at a time, across the surface. The algorithm is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But we could also apply the same procedures to nonplanar surfaces.

This visibility-detection approach is also frequently alluded to as **the z-buffer** method, because object depth is usually measured along the z axis of a viewing system. However, rather than using actual z coordinates within the scene, depth-buffer algorithms often compute a distance from the view plane along the z axis.

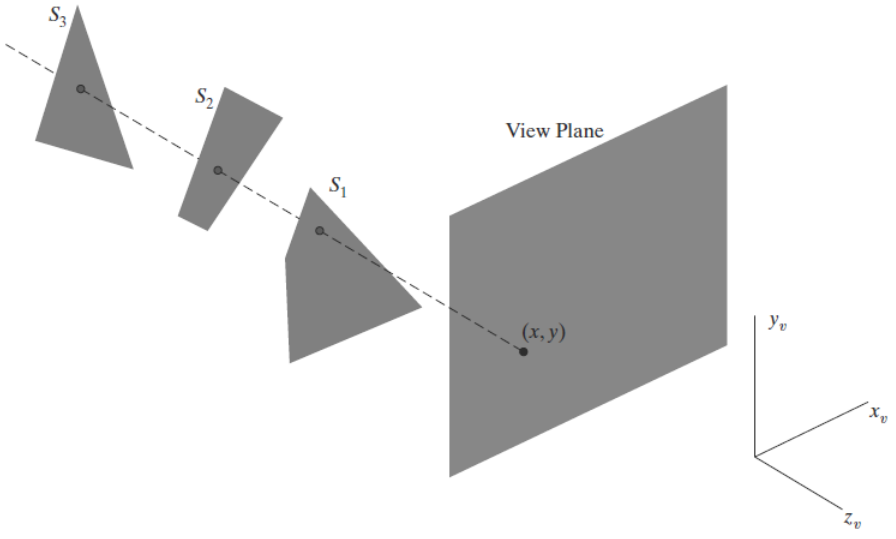

FIGURE 4
Three surfaces overlapping pixel position (x, y) on the view plane. The visible surface, S_1, has the smallest depth value.

Figure 4 shows three surfaces at varying distances along the orthographic projection line from position (x, y) on a view plane. These surfaces can be processed in any order.

As each surface is processed, its depth from the view plane is compared to previously processed surfaces. If a surface is closer than any previously processed surfaces, its surface color is calculated and saved, along with its depth.

The visible surfaces in a scene are represented by the set of surface colors that have been saved after all surface processing is completed.

Implementation of the depth-buffer algorithm is typically carried out in normalized coordinates, so that depth values range from 0 at the near clipping plane (the view plane) to 1.0 at the far clipping plane.

As implied by the name of this method, two buffer areas are required. A depth buffer is used to store depth values for each (x, y) position as surfaces are processed, and the frame buffer stores the surface-color values for each pixel position.

Initially, all positions in the depth buffer are set to 1.0 (maximum depth),and the frame buffer (refresh buffer) is initialized to the background color. Each surface listed in the polygon tables is then processed, one scan line at a time, by calculating the depth value at each (x, y) pixel position. This calculated depth is compared to the value previously stored in the depth buffer for that pixel position.

If the calculated depth is less than the value stored in the depth buffer, the new depth value is stored. Then the surface color at that position is computed and placed in the corresponding pixel location in the frame buffer.

The depth-buffer processing steps are summarized in the following algorithm. This algorithm assumes that depth values are normalized on the range from 0.0 to 1.0 with the view plane at depth= 0 and the farthest depth= 1.We can also apply this algorithm for any other depth range, and some graphics packages allow the user to specify the depth range over which the depth-buffer algorithm is to be applied.

Within the algorithm, the variable z represents the depth of the polygon (that is, its distance from the view plane along the negative z axis).

Depth-Buffer Algorithm

1. Initialize the depth buffer and frame buffer so that for all buffer positions(x, y),

depthBuff (x, y) = 1.0, frameBuff (x, y) = backgndColor

2. Process each polygon in a scene, one at a time, as follows:

- *For each projected (x, y) pixel position of a polygon, calculate the depth z (if not already known).*
- *If z <* ***depthBuff (x, y)****, compute the surface color at that position and set*

depthBuff (x, y) = z, frameBuff (x, y) = surfColor (x, y)

After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the frame buffer contains the corresponding color values for those surfaces.

Given the depth values for the vertex positions of any polygon in a scene, we can calculate the depth at any other point on the plane containing the polygon.

At surface position (x, y), the depth is calculated from the plane equation as

$$z = \frac{-Ax - By - D}{C}$$

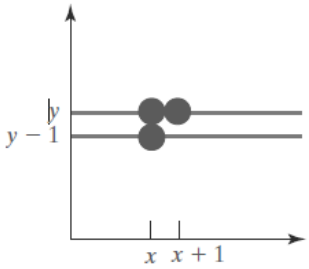

FIGURE 5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

For any scan line (Figure 5), adjacent horizontal x positions across the line differ by ±1, and vertical y values on adjacent scan lines differ by ±1. If the depth of position (x, y) has been determined to be z, then the depth z^1 of the next position (x + 1, y) along the scan line is obtained from Eq. 4 as

$$z' = \frac{-A(x+1) - By - D}{C}$$

or

$$z' = z - \frac{A}{C}$$

The ratio −A/C is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.

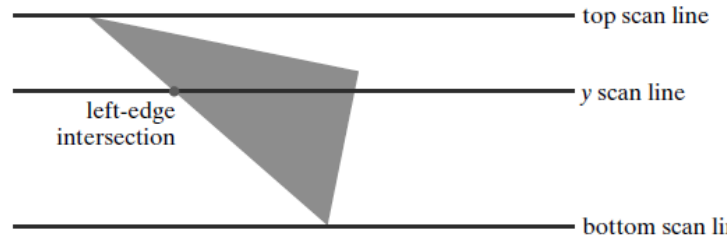

FIGURE 6
Scan lines intersecting a polygon surface.

Processing pixel positions from left to right across each scan line, we start by calculating the depth on a left polygon edge that intersects that scan line(Figure 6). For each successive position across the scan line, we then calculate the depth value using the above given equation.

We can implement the depth-buffer algorithm by starting at a top vertex of the polygon. Then, we could recursively calculate the x-coordinate values down a left edge of the polygon. The x value for the beginning position on each scan line can be calculated from the beginning (edge) x value of the previous scan line as

$$x' = x - \frac{1}{m}$$

where m is the slope of the edge (Figure 7).
Depth values down this edge are obtained recursively as

$$z' = z + \frac{A/m + B}{C}$$

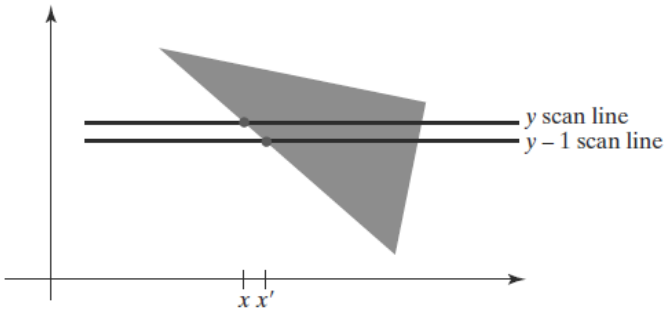

FIGURE 7
Intersection positions on successive scan lines along a left polygon edge.

If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

One slight complication with this approach is that while pixel positions are at integer (x, y) coordinates, the actual point of intersection of a scan line with the edge of a polygon may not be. As a result, it may be necessary to adjust the intersection point by rounding its fractional part up or down, as is done in scan-line polygon fill algorithms.

OpenGL Visibility-Detection Functions:

We can apply both back-face removal and the depth-buffer visibility-testing method to our scenes using functions that are provided in the basic library of OpenGL. In addition, we can use OpenGL functions to construct a wire-frame display of a scene with the hidden lines removed, and we can display scenes with depth cueing.

OpenGL Polygon-Culling Functions:

Back-face removal is accomplished with the functions

```
glEnable (GL_CULL_FACE);
glCullFace (mode);
```

where parameter ***mode*** is assigned the value GL_BACK.

We could use this function to remove the front faces instead, or we could even remove both frnt and back faces.

By default, parameter mode in the ***glCullFace*** function has the value GL_BACK. Therefore, if we activate culling with the **glEnable** function without explicitly invoking function glCullFace, the back faces in a scene will be removed.

The culling routine is turned off with

```
glDisable (GL_CULL_FACE);
```

OpenGL Depth-Buffer Functions

To use the OpenGL depth-buffer visibility-detection routines, we first need to modify the GL Utility Toolkit (GLUT) initialization function for the display mode to include a request for the depth buffer, as well as for the refresh buffer.

We do this, for example, with the statement

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

Depth buffer values can then be initialized with

```
glClear (GL_DEPTH_BUFFER_BIT);
```

Normally, the depth buffer is initialized with the same statement that initializes the refresh buffer to the background color. But we do need to clear the depth buffer each time we want to display a new frame.

In OpenGL, depth values are normalized in the range from 0.0 to 1.0, so that the preceding initialization sets all depth-buffer values to the maximum value 1.0 by default.

The OpenGL depth-buffer visibility-detection routines are activated with the following function:

```
glEnable (GL_DEPTH_TEST);
```

And we deactivate the depth-buffer routines with

```
glDisable (GL_DEPTH_TEST);
```

We can also apply depth-buffer visibility testing using some other initial value for the maximum depth, and this initial value is chosen with the OpenGL function:

```
glClearDepth (maxDepth);
```

Parameter maxDepth can be set to any value between 0.0 and 1.0.

To load this initialization value into the depth buffer, we next must invoke the **glClear (GL DEPTH BUFFER BIT)** function.

Otherwise, the depth buffer is initialized with the default value (1.0). Because surface-color calculations and other processing are not performed for objects that are beyond the specified maximum depth, this function can be used to speed up the depth-buffer routines when a scene contains many distant objects that are behind the foreground objects.

Projection coordinates in OpenGL are normalized to the range from −1.0 to 1.0, and the depth values between the near and far clipping planes are further normalized to the range from 0.0 to 1.0. The value 0.0 corresponds to the near clipping plane (the projection plane), and the value 1.0 corresponds to the far clipping plane.

As an option, we can adjust these normalization values with

glDepthRange (nearNormDepth, farNormDepth);

By default, ***nearNormDepth*** = 0.0 and ***farNormDepth*** = 1.0. But with the glDepthRange function, we can set these two parameters to any values within the range from 0.0 to 1.0, including nearNormDepth > farNormDepth. Using the glDepthRange function, we can restrict the depth-buffer testing to any region of the view volume, and we can even reverse the positions of the near and far planes.

Another option available in OpenGL is the test condition that is to be used for the depth-buffer routines.We specify a test condition with the following function:

glDepthFunc (testCondition);

Parameter ***testCondition*** can be assigned any one of the following eight symbolic constants: GL LESS, GL GREATER, GL EQUAL, GL NOTEQUAL, GL LEQUAL, GL GEQUAL, GL NEVER (no points are processed), and GL ALWAYS (all points are processed). These different tests can be useful in various applications to reduce calculations in depth-buffer processing.

The default value for parameter testCondition is GL LESS, so that a depth value is processed if it has a value that is less than the current value in the depth buffer for that pixel position.

We can also set the status of the depth buffer so that it is in a read-only state or in a read-write state. This is accomplished with

glDepthMask (writeStatus);

When writeStatus = GL TRUE (the default value), we can both read from and write to the depth buffer.With writeStatus = GL FALSE, the write mode for the depth buffer is disabled and we can retrieve values only for comparison in depth testing. This feature is useful when we want to use the same complicated background with displays of different foreground objects.

After storing the background in the depth buffer, we disable the write mode and process the foreground. This allows us to generate a series of frames with different foreground objects or with one object in different positions for an animation sequence. Thus, only the depth values for the background are saved.

Another application of the glDepthMask function is in displaying transparency effects. In this case, we want to save only the depths of opaque objects for visibility testing, not the depths

of the transparent-surface positions. So the write mode for the depth buffer is turned off when a transparent surface is processed.

Similar commands are available for setting the write status for the other buffers (color, index, and stencil).

OpenGL Wire-Frame Surface-Visibility Methods

A wire-frame display of a standard graphics object can be obtained in OpenGL by requesting that only its edges are to be generated. We do this by setting the polygon-mode function as, for example:

glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

But this displays both visible and hidden edges.

To eliminate the hidden lines in a wire-frame display, we can employ the

```
glEnable (GL_DEPTH_TEST);
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
glColor3f (1.0, 1.0, 1.0);
/* Invoke the object-description routine. */
glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
glColor3f (0.0, 0.0, 0.0);
/* Invoke the object-description routine again. */
glDisable (GL_POLYGON_OFFSET_FILL);
```

OpenGL Depth-Cueing Function

We can vary the brightness of an object as a function of its distance from the viewing position with

```
glEnable (GL_FOG);
glFogi (GL_FOG_MODE, GL_ LINEAR);
```

This applies the linear depth function to object colors using dmin = 0.0 and dmax = 1.0. But we can set different values for dmin and dmax with the following function calls:

```
glFogf (GL_FOG_START, minDepth);
glFogf (GL_FOG_END, maxDepth);
```

In these two functions, parameters ***minDepth*** and ***maxDepth*** are assigned floating-point values, although integer values can be used if we change the function suffix to i.

In addition, we can use the glFog function to set an atmosphere color that is to be combined with the color of an object after applying the linear depth cueing function.