FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Table of Contents

12 Write a program to perform a series of transformations on a 30×30 square whose centroid lies at $(-20, -20, 0)$ and that is contained in the xy plane. Use three-dimensional OpenGL matrix operations to perform the transformations. The square should first be reflected in the x axis, then rotated counterclockwise by 45° about its center, then sheared in the x direction by a value of 2.

13 Modify the program from the previous exercise so that the transformation sequence can be applied to any two-dimensional polygon, with vertices specified as user input.

14 Modify the example program in the previous exercise so that the order of the geometric transformation sequence can be specified as user input.

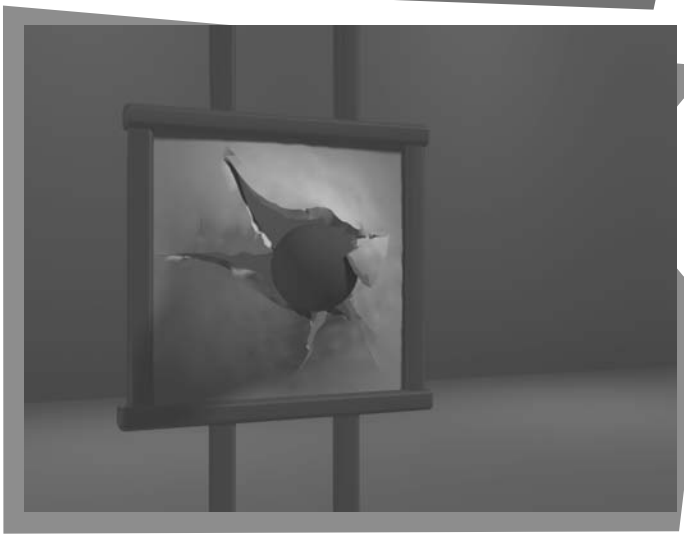
15 Modify the example program from the previous exercise so that the geometric transformation parameters are specified as user input.

IN MORE DEPTH

1 You have not yet been exposed to the material necessary to construct three-dimensional representations of the objects in your application, so you can instead embed the two-dimensional polygonal approximations to those objects in a three dimensional scene and perform three-dimensional transformations on those approximations using the techniques in this chapter. In this exercise, you will set up a set of transformations to produce an animation. Define the three-dimensional transformation matrices to do this using homogeneous coordinate representations. If two or more objects act as a single "unit" in certain behaviors that are easier to model in terms of relative positions, you can use the techniques in Section 6 to convert the local transformations of the objects relative to each other (in their own coordinate frame) into transformations in the world coordinate frame.

2 Use the matrices you designed in the previous exercise to produce an animation. You should employ the OpenGL matrix operations for three-dimensional transformations and have the matrices produce small changes in position for each of the objects in the scene. Since you haven t yet covered the material necessary for generating views of a threedimensional scene, simply display the animation using a two-dimensional orthogonal projection, with all of the polygons in the scene being contained in the xy plane. The transformations themselves, however, are still three-dimensional.

Three-Dimensional Viewing



For two-dimensional graphics applications, viewing operations transfer positions from the world-coordinate plane to pixel positions in the plane of the output device. Using the rectangular boundaries for the clipping window and the viewport, a two-dimensional package clips a scene and maps it to device coordinates. Three-dimensional viewing operations, however, are more involved, because we now have many more choices as to how we can construct a scene and how we can generate views of the scene on an output device.

1 Overview of Three-Dimensional Viewing Concepts

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 coordinate 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). 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.

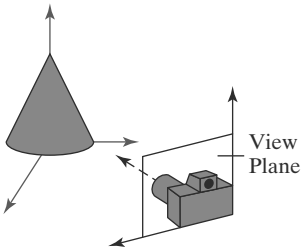


FIGURE 1
Coordinate reference for obtaining a selected view of a three-dimensional scene.

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.

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.

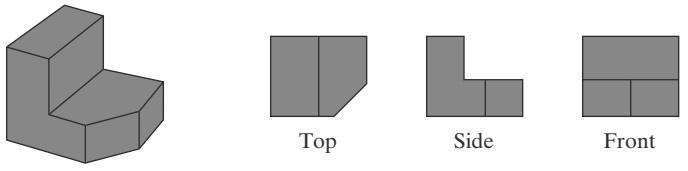


FIGURE 2
Three parallel-projection views of an object, showing relative proportions from different viewing positions.

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 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.

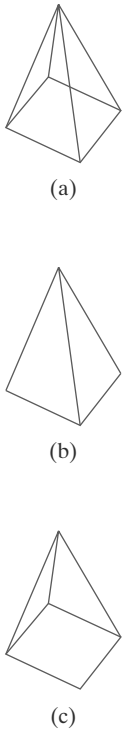


FIGURE 3
The wire-frame representation of the pyramid in (a) contains no depth information to indicate whether the viewing direction is (b) downward from a position above the apex or (c) upward from a position below the base.

Identifying Visible Lines and Surfaces

We can also clarify depth relationships in a wire-frame display using techniques other than depth cueing. One approach is simply to highlight the visible lines or to display them in a different color. Another technique, commonly used for engineering drawings, is to display the nonvisible lines as dashed lines. Or we could remove the nonvisible lines from the display, as in Figures 3(b) and 3(c). But removing the hidden lines also removes information about the shape of the back surfaces of an object, and wire-frame representations are generally used to get an indication of an object's overall appearance, front and back.

When a realistic view of a scene is to be produced, back parts of the objects are completely eliminated so that only the visible surfaces are displayed. In this case, surface-rendering procedures are applied so that screen pixels contain only the color patterns for the front surfaces.

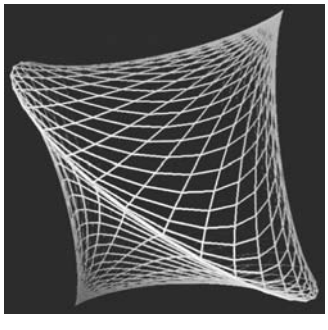


FIGURE 4
A wire-frame object displayed with depth cueing, so that the brightness of lines decreases from the front of the object to the back.

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. In Color Plate 9 surface-rendering methods are combined with perspective and visible-surface identification to generate a degree of realism in a displayed scene.

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.

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.

2 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 (Figure 5). 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.

We need to keep in mind, however, that the camera analogy can be carried only so far, because we have more flexibility and many more options for generating views of a scene with a computer-graphics program than we do with a real camera. We can choose to use either a parallel projection or a perspective projection, we can selectively eliminate parts of a scene along the line of sight, we can move the projection plane away from the "camera" position, and we can even get a picture of objects in back of our synthetic camera.

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 threedimensional scene on the output device, and a two-dimensional clipping window is used to

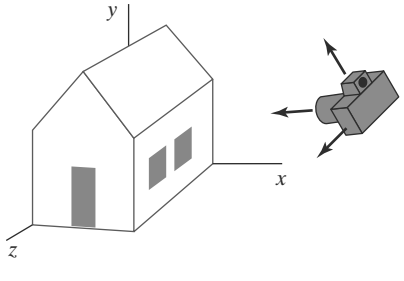


FIGURE 5
Photographing a scene involves selection of the camera position and orientation.

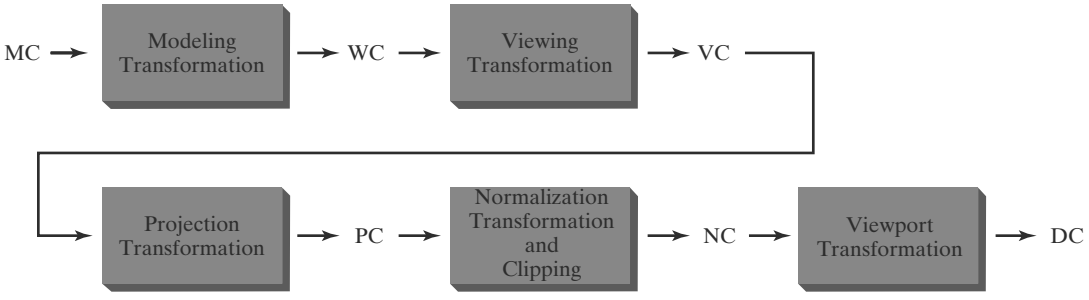


FIGURE 10-6
General three-dimensional transformation pipeline, from modeling coordinates (MC) to world coordinates (WC) to viewing coordinates (VC) to projection coordinates (PC) to normalized coordinates (NC) and, ultimately, to device coordinates (DC).

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 6 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. In this way, the coordinate transformations can be concatenated for maximum efficiency.

As in two-dimensional viewing, the viewport limits could be given in normalized coordinates or in device coordinates. In developing the viewing algorithms, we will assume that the viewport is to be specified in device coordinates and that normalized coordinates are transferred to viewport coordinates, following the clipping operations. 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.

3 Three-Dimensional Viewing-Coordinate Parameters

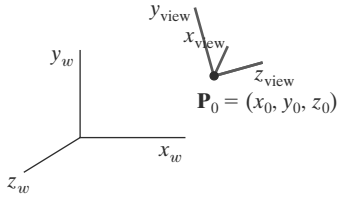


FIGURE 7
A right-handed viewing-coordinate system, with axes x_{view}, y_{view}, and z_{view}, relative to a right-handed world-coordinate frame.

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 $\mathbf{P}_0 = (x_0, y_0, z_0)$ 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 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 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.

The View-Plane Normal Vector

Because the viewing direction is usually along the 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 z_{view} axis, can be defined with a **view-plane normal vector N**, as shown in Figure 8.

An additional scalar parameter is used to set the position of the view plane at some coordinate value z_{vp} along the z_{view} axis, as illustrated in Figure 9. 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 z_{view} direction. Thus, the view plane is always parallel to the $x_{view}y_{view}$ 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. 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.

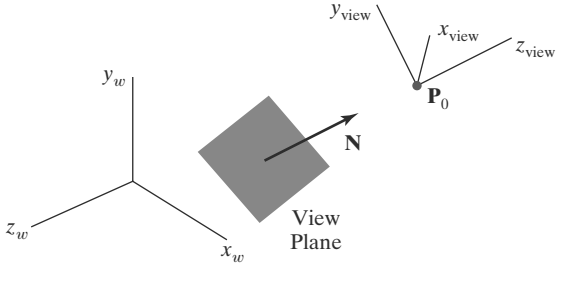


FIGURE 8
Orientation of the view plane and view-plane normal vector **N**.

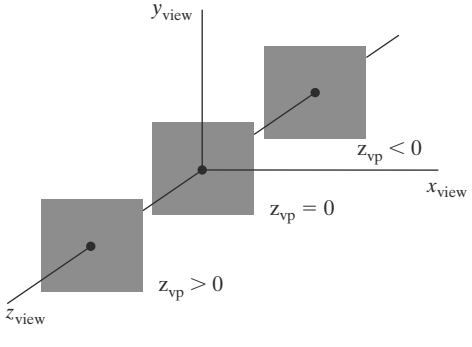


FIGURE 9
Three possible positions for the view plane along the z_{view} axis.

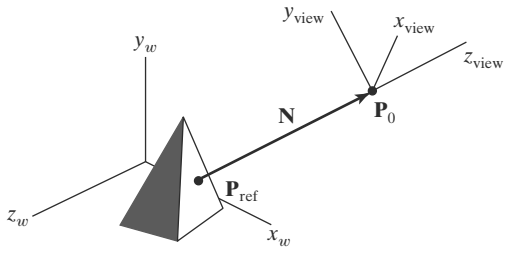


FIGURE 10
Specifying the view-plane normal vector **N** as the direction from a selected reference point $\mathbf{P}_{ref}$ to the viewing-coordinate origin $\mathbf{P}_0$.

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 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 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.

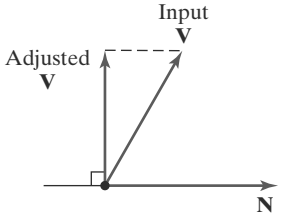


FIGURE 11
Adjusting the input direction of the view-up vector **V** to an orientation perpendicular to the view-plane normal vector **N**.

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 $\mathbf{V} = (0, 1, 0)$.

The uvn Viewing-Coordinate Reference Frame

Left-handed viewing coordinates are sometimes used in graphics packages, with the viewing direction in the positive z_{view} direction. With a left-handed system, increasing z_{view} values are interpreted as being farther from the viewing position along the line of sight. But right-handed viewing systems are more common, because they have the same orientation as the world-reference frame. This allows a graphics package to deal with only one coordinate orientation for both world and viewing references. Although some early graphics packages defined viewing coordinates within a left-handed frame, right-handed viewing coordinates are now used by the graphics standards. However, left-handed coordinate references are often used to represent screen coordinates and for the normalization transformation.

Because the view-plane normal **N** defines the direction for the z_{view} axis and the view-up vector **V** is used to obtain the direction for the y_{view} axis, we need only determine the direction for the 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 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 y_{view} axis. Following these procedures, we obtain the following set of unit axis vectors for a right-handed viewing coordinate system.

$$\begin{aligned} \mathbf{n} &= \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z) \\ \mathbf{u} &= \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z) \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z) \end{aligned} \tag{1}$$

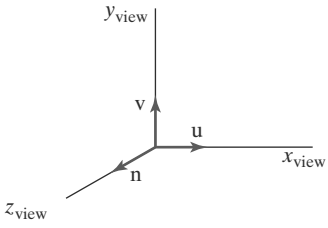


FIGURE 12
A right-handed viewing system defined with unit vectors **u**, **v**, and **n**.

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. 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.

4 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

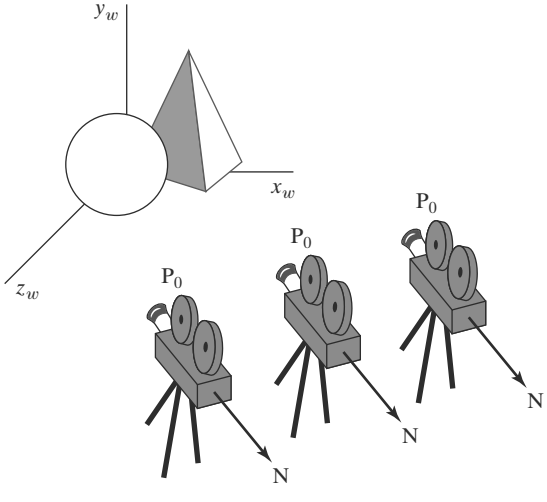


FIGURE 13
Panning across a scene by changing the viewing position, with a fixed direction for **N**.

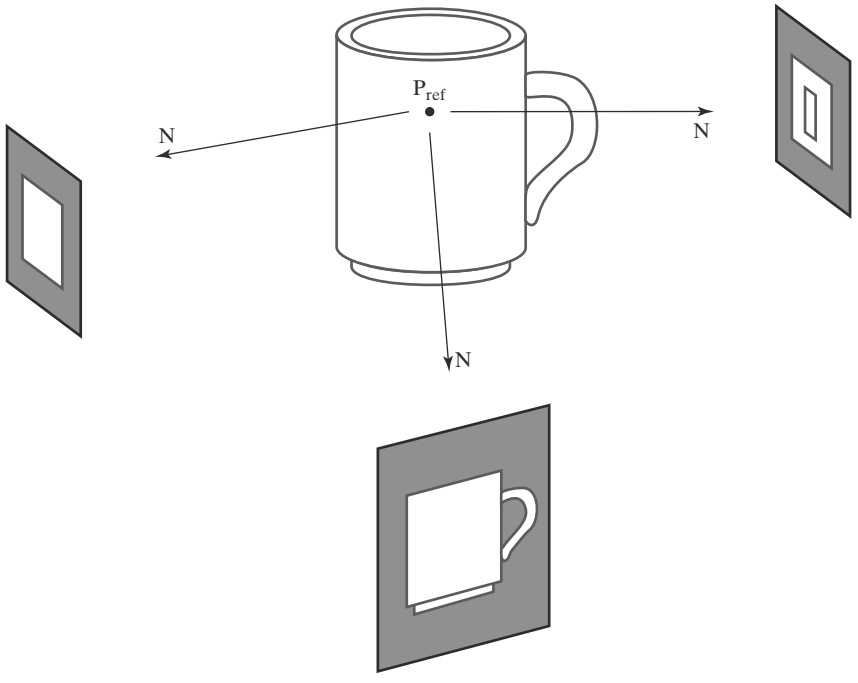


FIGURE 14
Viewing an object from different directions using a fixed reference point.

coordinate systems:

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

The viewing-coordinate origin is at world position $\mathbf{P_0} = (x_0, y_0, z_0)$. Therefore, the matrix for translating the viewing origin to the world origin is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

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

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

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:

$$\begin{aligned} \mathbf{M}_{WC,\,VC} &= \mathbf{R} \cdot \mathbf{T} \\ &= \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u} \cdot \mathbf{P}_0 \\ v_x & v_y & v_z & -\mathbf{v} \cdot \mathbf{P}_0 \\ n_x & n_y & n_z & -\mathbf{n} \cdot \mathbf{P}_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4}$$

Translation factors in this matrix are calculated as the vector dot product of each of the **u**, **v**, and **n** unit vectors with $\mathbf{P}_0$, which represents a vector from the world origin to the viewing origin. In other words, the translation factors are the negative projections of $\mathbf{P}_0$ on each of the viewing-coordinate axes (the negative components of $\mathbf{P}_0$ in viewing coordinates). These matrix elements are evaluated as

$$\begin{aligned} -\mathbf{u} \cdot \mathbf{P}_0 &= -x_0 u_x - y_0 u_y - z_0 u_z \\ -\mathbf{v} \cdot \mathbf{P}_0 &= -x_0 v_x - y_0 v_y - z_0 v_z \\ -\mathbf{n} \cdot \mathbf{P}_0 &= -x_0 n_x - y_0 n_y - z_0 n_z \end{aligned} \tag{5}$$

Matrix 4 transfers world-coordinate object descriptions to the viewing reference frame.

5 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 straight-line segment defined with endpoint coordinates $\mathbf{P}_1$ and $\mathbf{P}_2$. 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 $\mathbf{P}_1$ and $\mathbf{P}_2$, 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.

6 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,

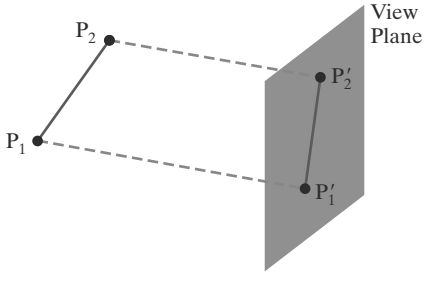


FIGURE 15
Parallel projection of a line segment onto a view plane.

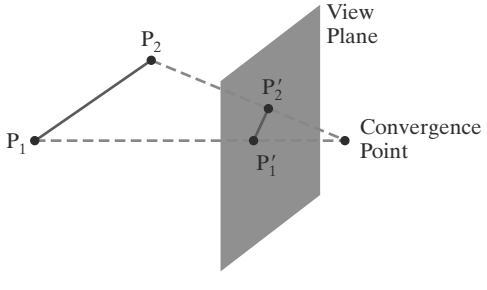


FIGURE 16
Perspective projection of a line segment onto a view plane.

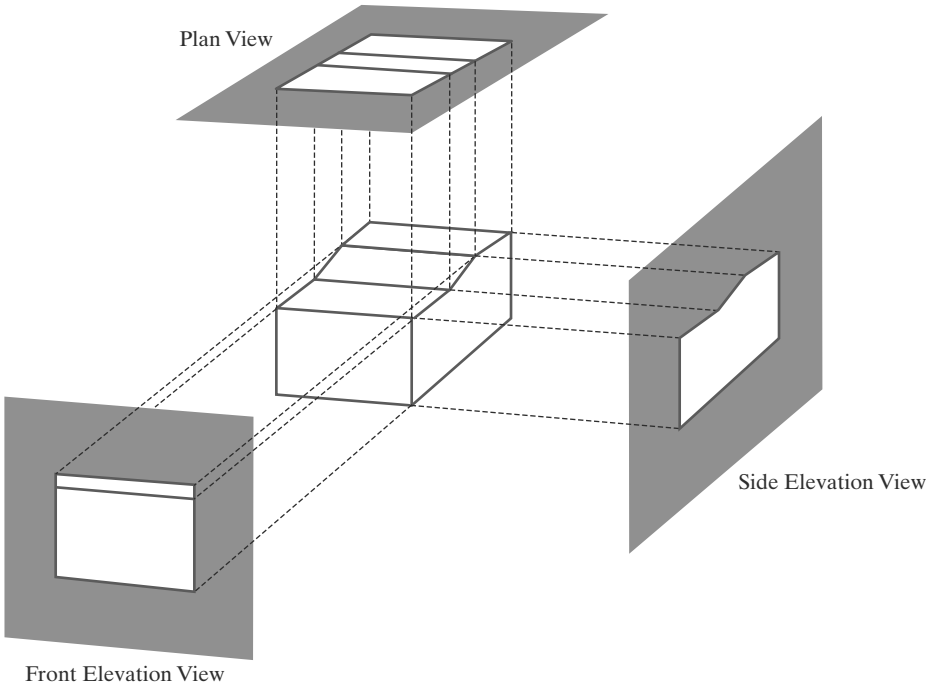


FIGURE 17
Orthogonal projections of an object, displaying plan and elevation views.

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*. 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. We can obtain the isometric projection shown in this figure by aligning the view-plane 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. This is not the case in a general axonometric projection, where scaling factors may be different for the three principal directions.

Orthogonal Projection Coordinates

With the projection direction parallel to the 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

$$x_p = x, \qquad y_p = y \tag{6}$$

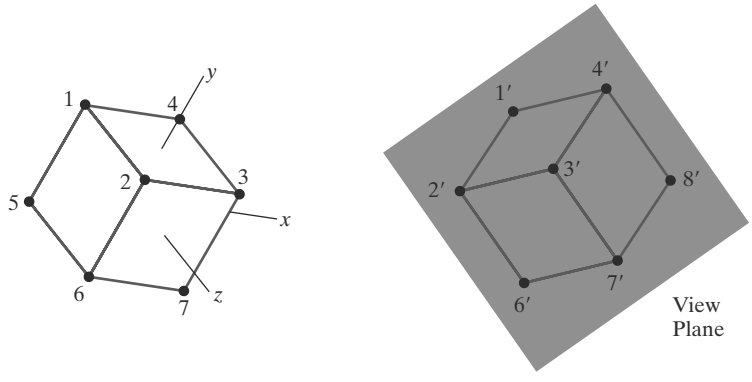


FIGURE 18
An isometric projection of a cube.

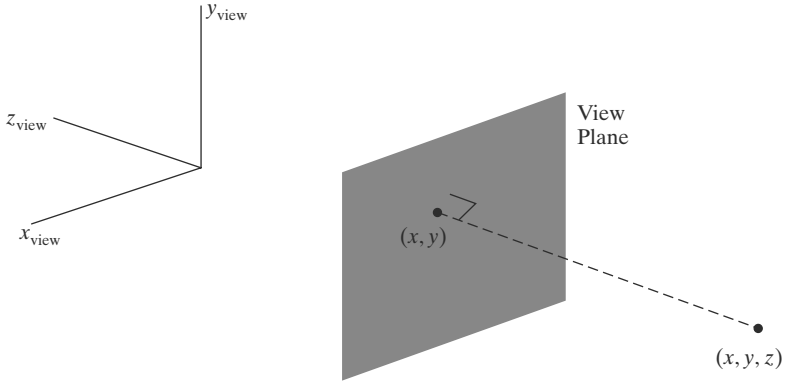


FIGURE 19
An orthogonal projection of a spatial position onto a view plane.

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 x_{view} and y_{view} axes, as shown in Figure 20. 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

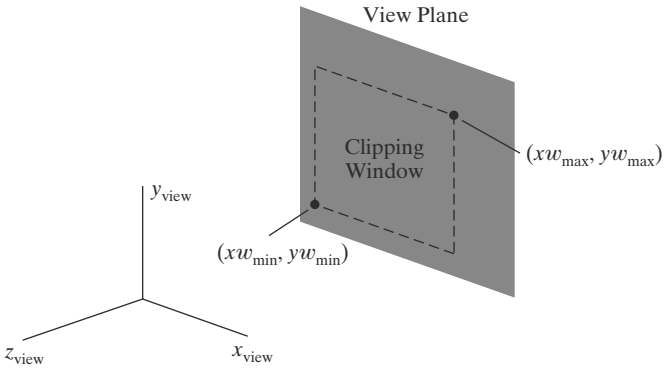


FIGURE 20
A clipping window on the view plane, with minimum and maximum coordinates given in the viewing reference system.

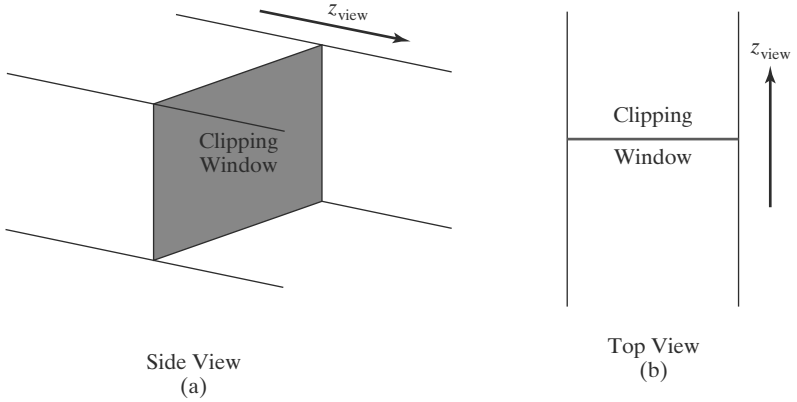


FIGURE 21
Infinite orthogonal-projection view volume.

through the edges of the clipping window to form an infinite clipping region, as in Figure 21.

We can limit the extent of the orthogonal view volume in the 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 z_{view} axis, we usually have $z_{far} < z_{near}$, so that the far plane is father out along the negative z_{view} axis. Some graphics libraries provide these two planes as options, and other libraries require them. 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. 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.

Graphics packages provide varying degrees of flexibility in the positioning of the near and far clipping planes, including options for specifying additional clipping planes at other positions in the scene. In general, the near and far planes can be in any relative position to each other to achieve various viewing effects, including positions that are on opposite sides of the view point. Similarly, the view plane can sometimes be placed in any position relative to the near and far clipping planes, although it is often taken to be coincident with the near clipping plane. However, providing numerous positioning options for the clipping and view planes usually results in less efficient processing of a three-dimensional scene.

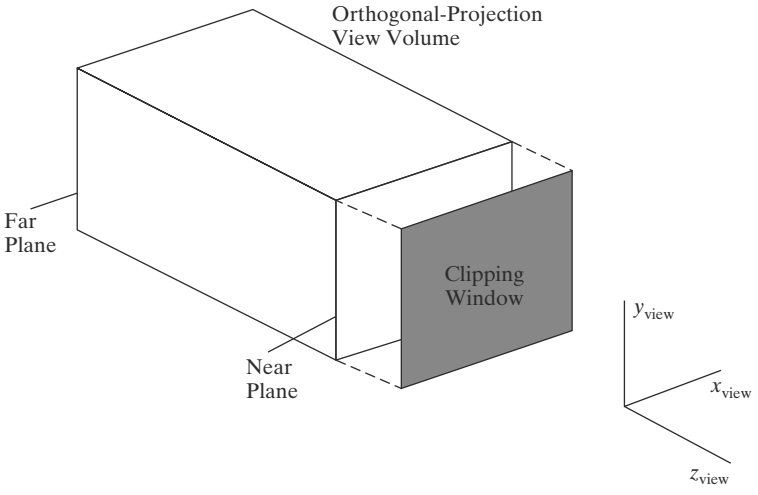


FIGURE 22
A finite orthogonal view volume with the view plane "in front" of the near plane.

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.

Because 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). 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

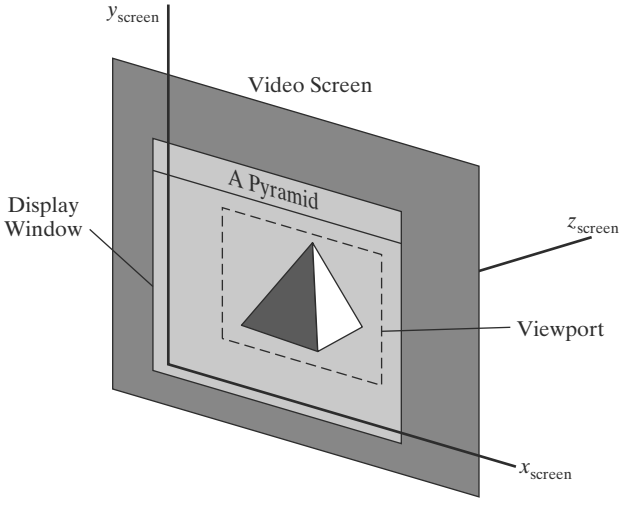


FIGURE 23
A left-handed screen-coordinate reference frame.

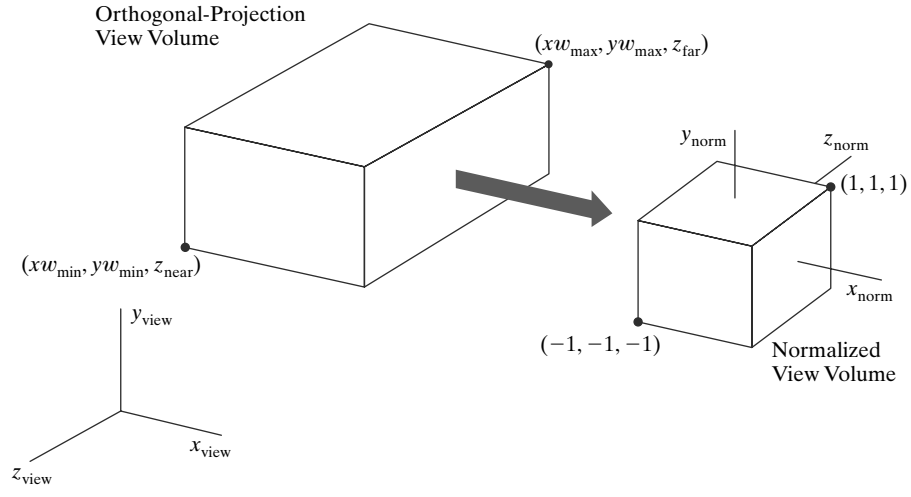


FIGURE 24
Normalization transformation from an orthogonal-projection view volume to the symmetric normalization cube within a left-handed reference frame.

normalization cube within a left-handed reference frame. Also, z-coordinate positions for the near and far planes are denoted as z_{near} and 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. The normalization transformation for the orthogonal view volume is

$$\mathbf{M}_{ortho,norm} = \begin{bmatrix} \frac{2}{xw_{max} - xw_{min}} & 0 & 0 & -\frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \frac{2}{yw_{max} - yw_{min}} & 0 & -\frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & \frac{-2}{z_{near} - z_{far}} & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

This matrix is multiplied on the right by the composite viewing transformation $\mathbf{R} \cdot \mathbf{T}$ (Section 4) 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.

7 Oblique Parallel Projections

In general, a parallel-projection view of a scene is obtained by transferring object descriptions to the view plane along projection paths that can be in any selected

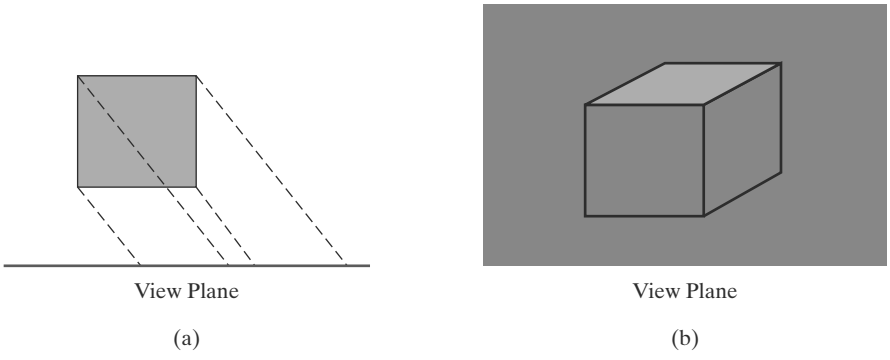


FIGURE 25
An oblique parallel projection of a cube, shown in a top view (a), produces a view (b) containing multiple surfaces of the cube.

direction relative to the view-plane normal vector. When the projection path is not perpendicular to the view plane, this mapping is called an **oblique parallel projection.** Using this projection, we can produce combinations such as a front, side, and top view of an object, as in Figure 25. Oblique parallel projections are defined by a vector direction for the projection lines, and this direction can be specified in various ways.

Oblique Parallel Projections in Drafting and Design

For applications in engineering and architectural design, an oblique parallel projection is often specified with two angles, α and ϕ, as shown in Figure 26. A spatial position (x, y, z), in this illustration, is projected to (x_p, y_p, z_{vp}) on a view plane, which is at location z_{vp} along the viewing z axis. Position (x, y, z_{vp}) is the corresponding orthogonal-projection point. The oblique parallel projection line from (x, y, z) to (x_p, y_p, z_{vp}) has an intersection angle α with the line on the projection plane that joins (x_p, y_p, z_{vp}) and (x, y, z_{vp}). This view-plane line, with length L, is at an angle ϕ with the horizontal direction in the projection plane. Angle α can be assigned a value between 0 and 90°, and angle ϕ can vary from 0 to 360°. We can express the projection coordinates in terms of x, y, L, and ϕ as

$$\begin{aligned} x_p &= x + L\cos\phi \\ y_p &= y + L\sin\phi \end{aligned} \tag{8}$$

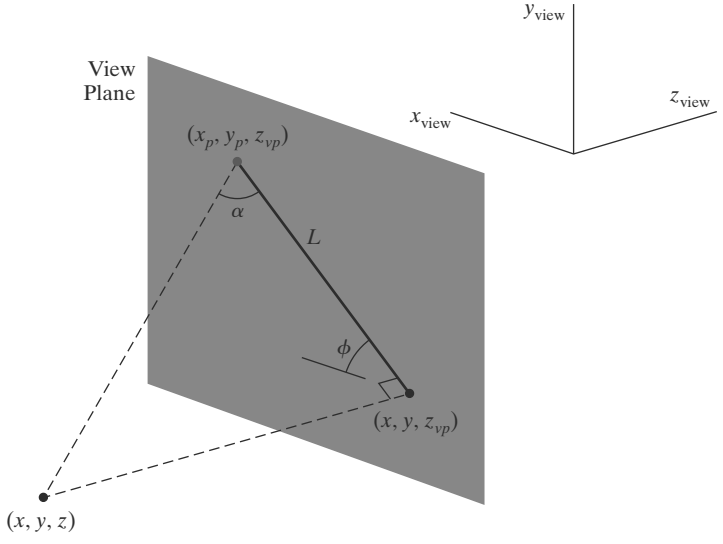


FIGURE 26
An oblique parallel projection of coordinate position (x, y, z) to position (x_p, y_p, z_{vp}) on a projection plane at position z_{vp} along the z_{view} axis.

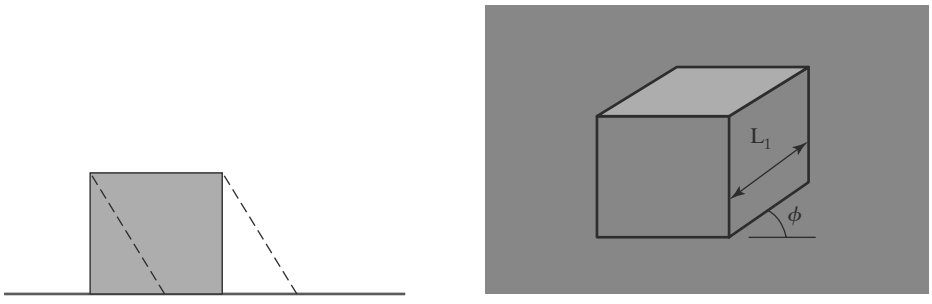


FIGURE 27
An oblique parallel projection (a) of a cube (top view) onto a view plane that is coincident with the front face of the cube produces the combination front, side, and top view shown in (b).

Length L depends on the angle α and the perpendicular distance of the point (x, y, z) from the view plane:

$$\tan\alpha = \frac{z_{vp} - z}{L} \tag{9}$$

Thus

$$\begin{aligned} L &= \frac{z_{vp} - z}{\tan\alpha} \\ &= L_1(z_{vp} - z) \end{aligned} \tag{10}$$

where $L_1 = \cot\alpha$, which is also the value of L when $z_{vp} - z = 1$. We can then write the oblique parallel projection equations 8 as

$$\begin{aligned} x_p &= x + L_1(z_{vp} - z)\cos\phi \\ y_p &= y + L_1(z_{vp} - z)\sin\phi \end{aligned} \tag{11}$$

An orthogonal projection is obtained when $L_1 = 0$ (which occurs at the projection angle $\alpha = 90^\circ$).

Equations 11 represent a z-axis shearing transformation. In fact, the effect of an oblique parallel projection is to shear planes of constant z and project them onto the view plane. The (x, y) positions on each plane of constant z are shifted by an amount proportional to the distance of the plane from the view plane, so that angles, distances, and parallel lines in the plane are projected accurately. This effect is shown in Figure 27, where the view plane is positioned at the front face of a cube. The back plane of the cube is sheared and overlapped with the front plane in the projection to the viewing surface. A side edge of the cube connecting the front and back planes is projected into a line of length L_1 that makes an angle ϕ with a horizontal line in the projection plane.

Cavalier and Cabinet Oblique Parallel Projections

Typical choices for angle ϕ are 30° and 45°, which display a combination view of the front, side, and top (or front, side, and bottom) of an object. Two commonly used values for α are those for which $\tan\alpha = 1$ and $\tan\alpha = 2$. For the first case, $\alpha = 45^\circ$ and the views obtained are called **cavalier** projections. All lines perpendicular to the projection plane are projected with no change in length. Examples of cavalier projections for a cube are given in Figure 28.

When the projection angle α is chosen so that $\tan\alpha = 2$, the resulting view is called a **cabinet** projection. For this angle ($\approx 63.4^\circ$), lines perpendicular to the viewing surface are projected at half their length. Cabinet projections appear more realistic than cavalier projections because of this reduction in the length of perpendiculars. Figure 29 shows examples of cabinet projections for a cube.

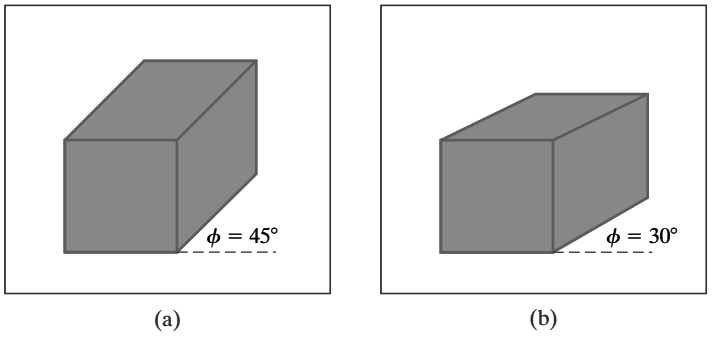


FIGURE 28
Cavalier projections of a cube onto a view plane for two values of angle ϕ. The depth of the cube is projected with a length equal to that of the width and height.

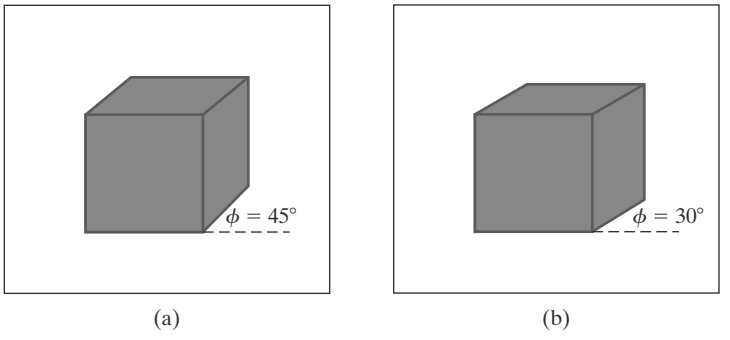


FIGURE 29
Cabinet projections of a cube onto a view plane for two values of angle ϕ. The depth is projected with a length that is one half that of the width and height of the cube.

Oblique Parallel-Projection Vector

In graphics programming libraries that support oblique parallel projections, the direction of projection to the view plane is specified with a **parallel-projection vector, $\mathbf{V}_p$**. This direction vector can be designated with a reference position relative to the view point, as we did with the view-plane normal vector, or with any other two points. Some packages use a reference point relative to the center of the clipping window to define the direction for a parallel projection. If the projection vector is specified in world coordinates, it must first be transformed to viewing coordinates using the rotation matrix discussed in Section 4. (The projection vector is unaffected by the translation, because it is simply a direction with no fixed position.)

Once the projection vector $\mathbf{V}_p$ is established in viewing coordinates, all points in the scene are transferred to the view plane along lines that are parallel to this vector. Figure 30 illustrates an oblique parallel projection of a spatial point to the view plane. We can denote the components of the projection vector relative to the viewing-coordinate frame as $\mathbf{V}_p = (V_{px}, V_{py}, V_{pz})$, where $V_{py}/V_{px} = \tan\phi$. Then, comparing similar triangles in Figure 30, we have

$$\frac{x_p - x}{z_{vp} - z} = \frac{V_{px}}{V_{pz}}$$
$$\frac{y_p - y}{z_{vp} - z} = \frac{V_{py}}{V_{pz}}$$

And we can write the equivalent of the oblique parallel-projection equations 11 in terms of the projection vector as

$$x_p = x + (z_{vp} - z)\frac{V_{px}}{V_{pz}}$$
$$y_p = y + (z_{vp} - z)\frac{V_{py}}{V_{pz}} \qquad (12)$$

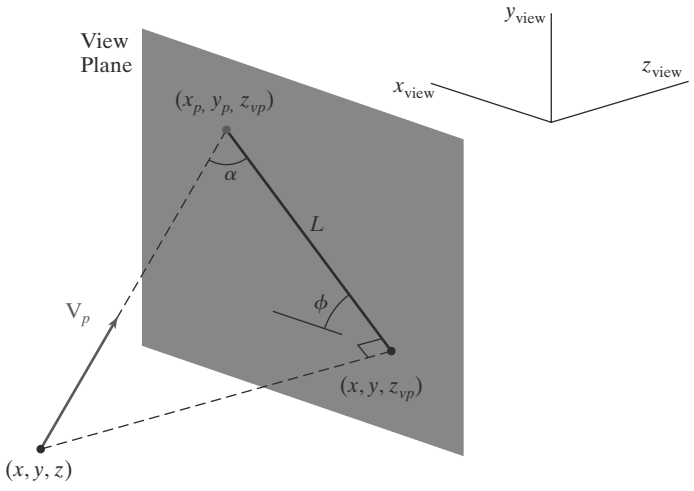


FIGURE 30
Oblique parallel projection of position (x, y, z) to a view plane along a projection line defined with vector $\mathbf{V}_p$.

The oblique parallel-projection coordinates in 12 reduce to the orthogonal-projection coordinates 6 when $V_{px} = V_{py} = 0$.

Clipping Window and Oblique Parallel-Projection View Volume

A view volume for an oblique parallel projection is set up using the same procedures as in an orthogonal projection. We select a clipping window on the view plane with coordinate positions $(xw_{\min}, yw_{\min})$ and $(xw_{\max}, yw_{\max})$, for the lower-left and upper-right corners of the clipping rectangle. The top, bottom, and sides of the view volume are then defined by the direction of projection and the edges of the clipping window. In addition, we can limit the extent of the view volume by adding a near plane and a far plane, as in Figure 31. The finite oblique parallel-projection view volume is an oblique parallelepiped.

Oblique parallel projections may be affected by changes in the position of the view plane, depending on how the projection direction is to be specified. In some systems, the oblique parallel-projection direction is parallel to the line connecting a reference point to the center of the clipping window. Therefore, moving the position of the view plane or clipping window without adjusting the reference point changes the shape of the view volume.

Oblique Parallel-Projection Transformation Matrix

Using the projection-vector parameters from the equations in 12, we can express the elements of the transformation matrix for an oblique parallel

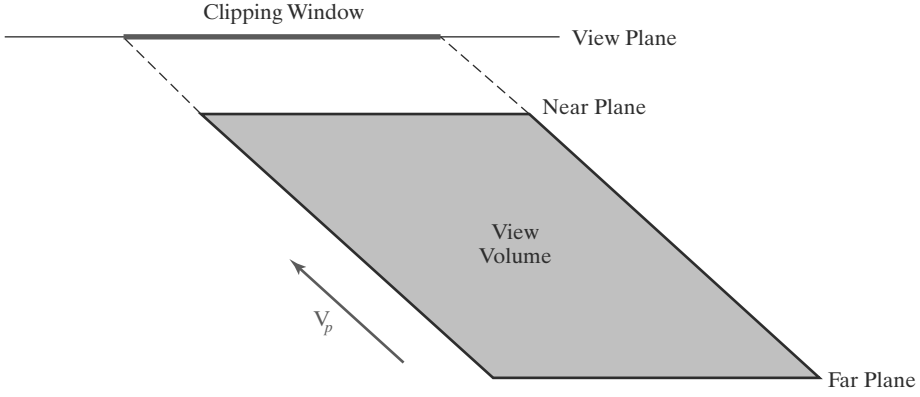


FIGURE 31
Top view of a finite view volume for an oblique parallel projection in the direction of vector $\mathbf{V}_p$.

projection as

$$\mathbf{M}_{\text{oblique}} = \begin{bmatrix} 1 & 0 & -\frac{V_{px}}{V_{pz}} & z_{vp}\frac{V_{px}}{V_{pz}} \\ 0 & 1 & -\frac{V_{py}}{V_{pz}} & z_{vp}\frac{V_{py}}{V_{pz}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

This matrix shifts the values of the x and y coordinates by an amount proportional to the distance from the view plane, which is at position z_{vp} on the z_{view} axis. The z values of spatial positions are unchanged. If $V_{px} = V_{py} = 0$, we have an orthogonal projection and matrix 13 is reduced to the identity matrix.

For a general oblique parallel projection, matrix 13 represents a z-axis shearing transformation. All coordinate positions within the oblique view volume are sheared by an amount proportional to their distance from the view plane. The effect is to shear the oblique view volume into a rectangular parallelepiped, as illustrated in Figure 32. Thus, positions inside the view volume are sheared into orthogonal-projection coordinates by the oblique parallel-projection transformation.

Normalization Transformation for an Oblique Parallel Projection

Because the oblique parallel-projection equations convert object descriptions to orthogonal-coordinate positions, we can apply the normalization procedures following this transformation. The oblique view volume has been converted to a rectangular parallelepiped, so we use the same procedures as in Section 6.

Following the normalization example in Section 6, we again map to the symmetric normalized cube within a left-handed coordinate frame. Thus, the complete transformation, from viewing coordinates to normalized coordinates, for an oblique parallel projection is

$$\mathbf{M}_{\text{oblique,norm}} = \mathbf{M}_{\text{ortho,norm}} \cdot \mathbf{M}_{\text{oblique}} \tag{14}$$

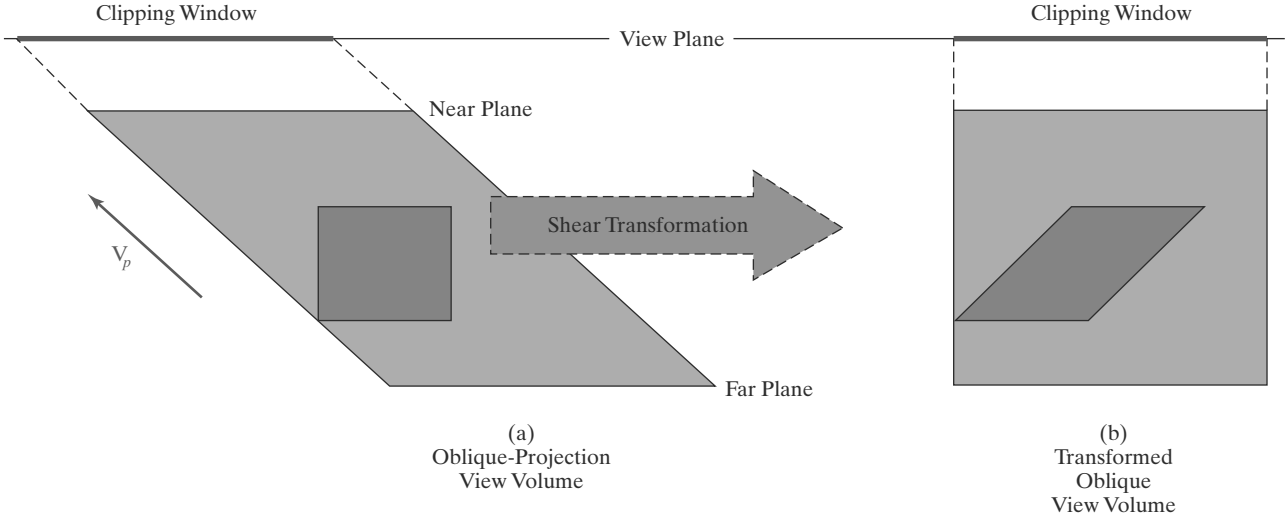


FIGURE 32
Top view of an oblique parallel-projection transformation. The oblique view volume is converted into a rectangular parallelepiped, and objects in the view volume, such as the green block, are mapped to orthogonal-projection coordinates.

Transformation $\mathbf{M}_{\text{oblique}}$ is matrix 13, which converts the scene description to orthogonal-projection coordinates; and transformation $\mathbf{M}_{\text{ortho,norm}}$ is matrix 7, which maps the contents of the orthogonal view volume to the symmetric normalization cube.

To complete the viewing transformations (with the exception of the mapping to viewport screen coordinates), we concatenate matrix 14 to the left of the transformation $\mathbf{M}_{WC,VC}$ from Section 4. Clipping routines can then be applied to the normalized view volume, followed by the determination of visible objects, the surface-rendering procedures, and the viewport transformation.

8 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).

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 $(x_{prp}, y_{prp}, z_{prp})$. 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

$$\begin{aligned} x' &= x - (x - x_{prp})u \\ y' &= y - (y - y_{prp})u \qquad 0 \le u \le 1 \\ z' &= z - (z - z_{prp})u \end{aligned} \tag{15}$$

Coordinate position (x', y', z') represents any point along the projection line. When $u = 0$, we are at position $\mathbf{P} = (x, y, z)$. At the other end of the line, $u = 1$ and

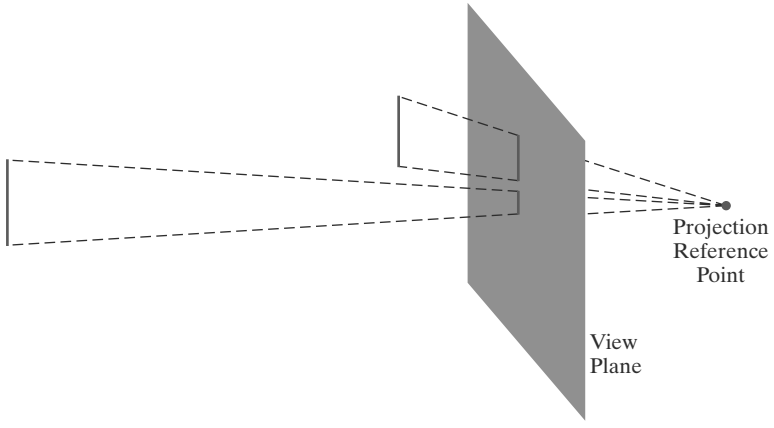


FIGURE 33
A perspective projection of two equal-length line segments at different distances from the view plane.

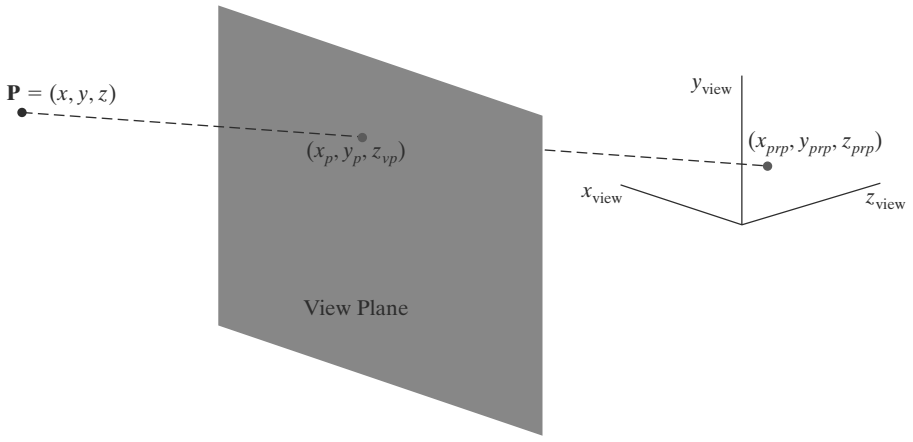


FIGURE 34
A perspective projection of a point **P** with coordinates (x, y, z) to a selected projection reference point. The intersection position on the view plane is (x_p, y_p, z_{vp}).

we have the projection reference-point coordinates $(x_{prp}, y_{prp}, z_{prp})$. On the view plane, $z' = z_{vp}$ and we can solve the z' equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z} \tag{16}$$

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

$$\begin{aligned} x_p &= x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right) \\ y_p &= y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right) \end{aligned} \tag{17}$$

Calculations for a perspective mapping are more complex than the parallel-projection equations, because the denominators in the perspective calculations 17 are functions of the z coordinate of the spatial position. Therefore, we now need to formulate the perspective-transformation procedures a little differently so that this mapping can be concatenated with the other viewing transformations. But first we take a look at some of the properties of Equations 17.

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 z_{view} axis, then

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

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) \tag{18}$$

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

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

$$x_p = x\left(\frac{z_{vp}}{z}\right), \qquad y_p = y\left(\frac{z_{vp}}{z}\right) \tag{19}$$

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. $z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) - x_{prp}\left(\frac{z}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) - y_{prp}\left(\frac{z}{z_{prp} - z}\right) \tag{20}$$

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

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

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) \tag{21}$$

Of course, we cannot have the projection reference point on the view plane. In that case, the entire scene would project to a single point. The view plane is usually placed between the projection reference point and the scene, but, in general, the view plane could be placed anywhere except at the projection point. If the projection reference point is between the view plane and the scene, objects are inverted on the view plane (Figure 35). 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, as illustrated in Figure 36. If the projection

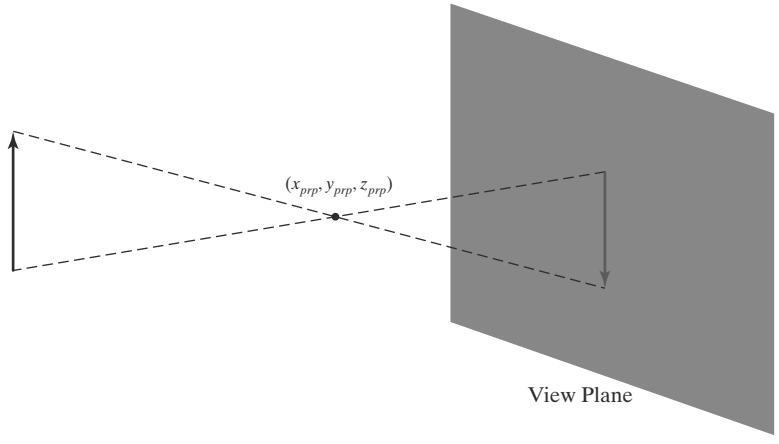


FIGURE 35
A perspective-projection view of an object is upside down when the projection reference point is between the object and the view plane.

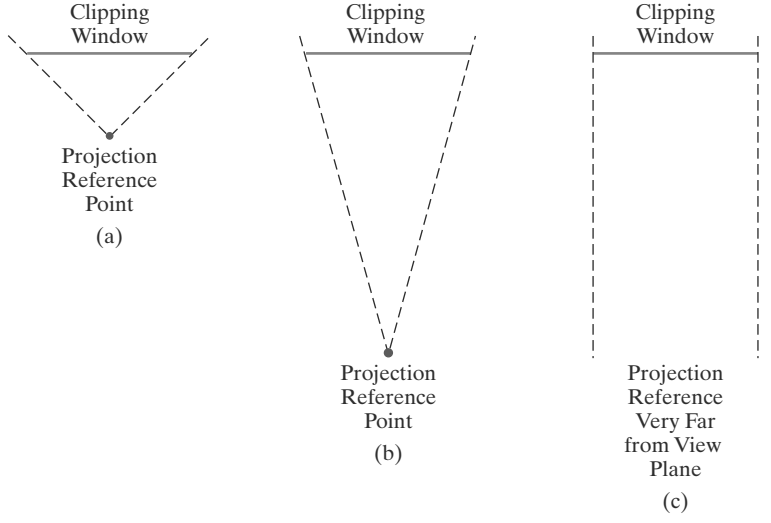


FIGURE 36
Changing perspective effects by moving the projection reference point away from the view plane.

reference point is close to the view plane, perspective effects are emphasized; that is, closer objects will appear much larger than more distant objects of the same size. Similarly, as the projection reference point moves farther from the view plane, the difference in the size of near and far objects decreases. When the projection reference point is very far from the view plane, a perspective projection approaches a parallel projection.

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. In the projected view (b), the view plane is aligned parallel to the xy object plane so that only the object z axis is intersected. This orientation produces a one-point perspective projection with a z-axis

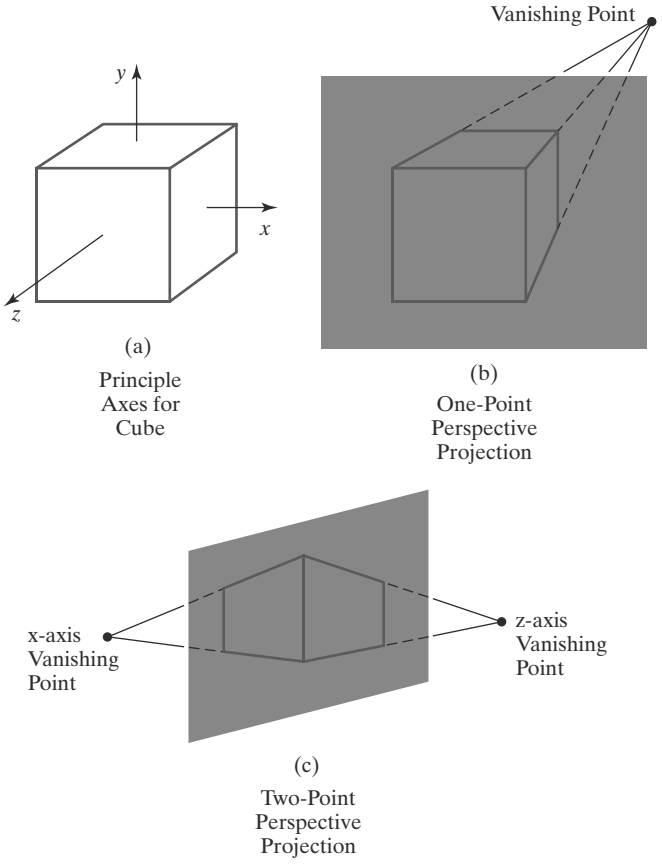


FIGURE 37
Principal vanishing points for perspective-projection views of a cube. When the cube in (a) is projected to a view plane that intersects only the z axis, a single vanishing point in the z direction (b) is generated. When the cube is projected to a view plane that intersects both the z and x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

vanishing point. For the view shown in (c), the projection plane intersects both the x and z axes but not the y axis. The resulting two-point perspective projection contains both x-axis and z-axis vanishing points. There is not much increase in the realism of a three-point perspective projection compared to a two-point projection, so three-point projections are not used as often in architectural and engineering drawings.

Perspective-Projection View Volume

We again 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). 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 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. Sometimes the near and far planes are required in a graphics package, and sometimes they are optional.

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. Some systems restrict the placement of the view plane relative to the near and far planes, and other systems allow it to be placed anywhere except

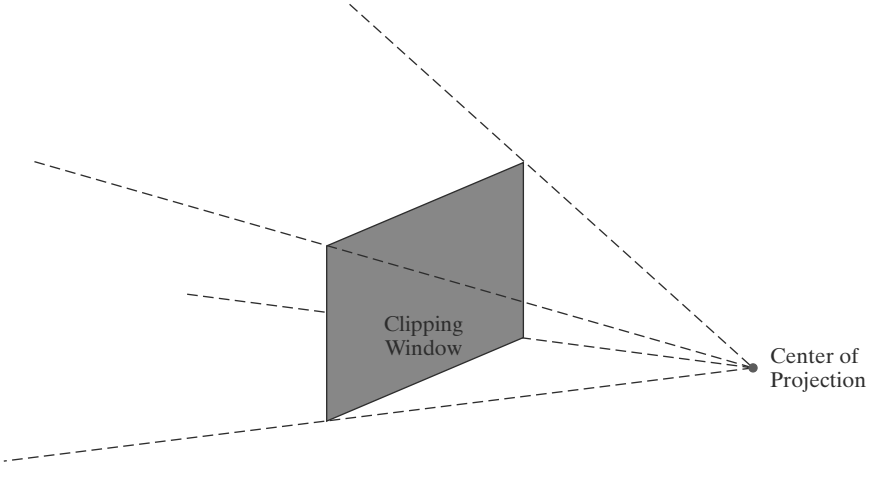


FIGURE 38
An infinite, pyramid view volume for a perspective projection.

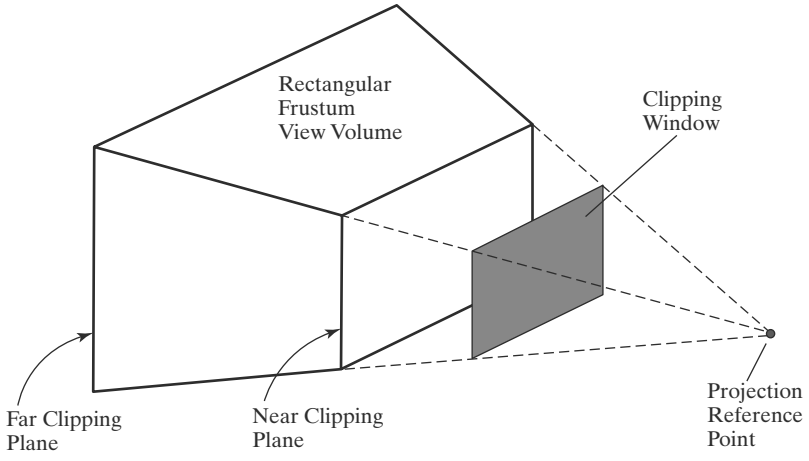


FIGURE 39
A perspective-projection frustum view volume with the view plane "in front" of the near clipping plane.

at the position of the projection reference point. If the view plane is "behind" the projection reference point, objects are inverted, as shown in Figure 35.

Perspective-Projection Transformation Matrix

Unlike a parallel projection, we cannot directly use the coefficients of the x and y coordinates in equations 17 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

$$x_p = \frac{x_h}{h}, \qquad y_p = \frac{y_h}{h} \tag{22}$$

where the homogeneous parameter has the value

$$h = z_{prp} - z \tag{23}$$

The numerators in 22 are the same as in equations 17:

$$\begin{aligned} x_h &= x(z_{prp} - z_{vp}) + x_{prp}(z_{vp} - z) \\ y_h &= y(z_{prp} - z_{vp}) + y_{prp}(z_{vp} - z) \end{aligned} \tag{24}$$

Thus, we can set up a transformation matrix to convert a spatial position to homogeneous coordinates so that the matrix contains only the perspective parameters and not coordinate values. 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:

$$\mathbf{P}_h = \mathbf{M}_{\text{pers}} \cdot \mathbf{P} \tag{25}$$

where $\mathbf{P}_h$ is the column-matrix representation of the homogeneous point (x_h, y_h, z_h, h) and $\mathbf{P}$ is the column-matrix representation of the coordinate position $(x, y, z, 1)$. (Actually, the perspective matrix would be concatenated with the other viewing-transformation matrices, and then the composite matrix would be applied to the world-coordinate description of a scene to produce homogeneous coordinates.) 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.

Setting up matrix elements for obtaining the homogeneous-coordinate x_h and y_h values in 24 is straightforward, but we must also structure the matrix to preserve depth (z-value) information. Otherwise, the z coordinates are distorted by the homogeneous-division parameter h. We can do this by setting up the matrix elements for the z transformation so as to normalize the perspective-projection z_p coordinates. There are various ways that we could choose the matrix elements to produce the homogeneous coordinates 24 and the normalized z_p value for a spatial position (x, y, z). The following matrix gives one possible way to formulate a perspective-projection matrix.

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} z_{prp} - z_{vp} & 0 & -x_{prp} & x_{prp}z_{prp} \\ 0 & z_{prp} - z_{vp} & -y_{prp} & y_{prp}z_{prp} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{prp} \end{bmatrix} \tag{26}$$

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.

Matrix 26 converts the description of a scene into homogeneous parallel-projection coordinates. However, the frustum view volume can have any orientation, so that these transformed coordinates could correspond to an oblique parallel projection. This occurs if the frustum view volume is not symmetric. If the frustum view volume for the perspective projection is symmetric, the resulting parallel-projection coordinates correspond to an orthogonal projection. We next consider these two possibilities.

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.

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

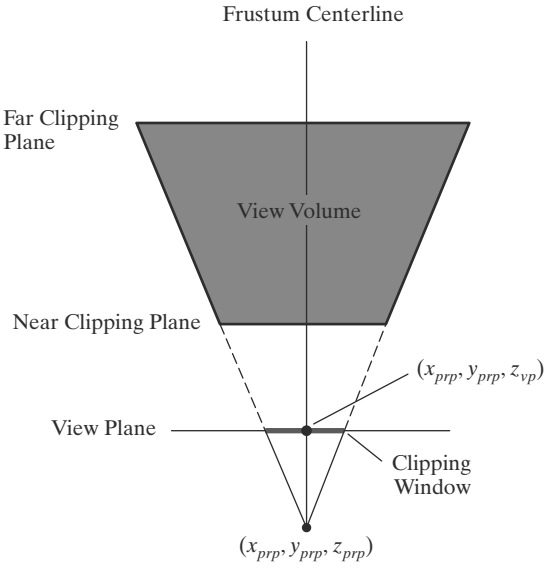


FIGURE 40
A symmetric perspective-projection frustum view volume, with the view plane between the projection reference point and the near clipping plane. This frustum is symmetric about its centerline when viewed from above, below, or either side.

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. Typically, 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.

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

$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{height}/2}{z_{prp} - z_{vp}} \tag{27}$$

so that the clipping-window height can be calculated as

$$\text{height} = 2(z_{prp} - z_{vp}) \tan\left(\frac{\theta}{2}\right) \tag{28}$$

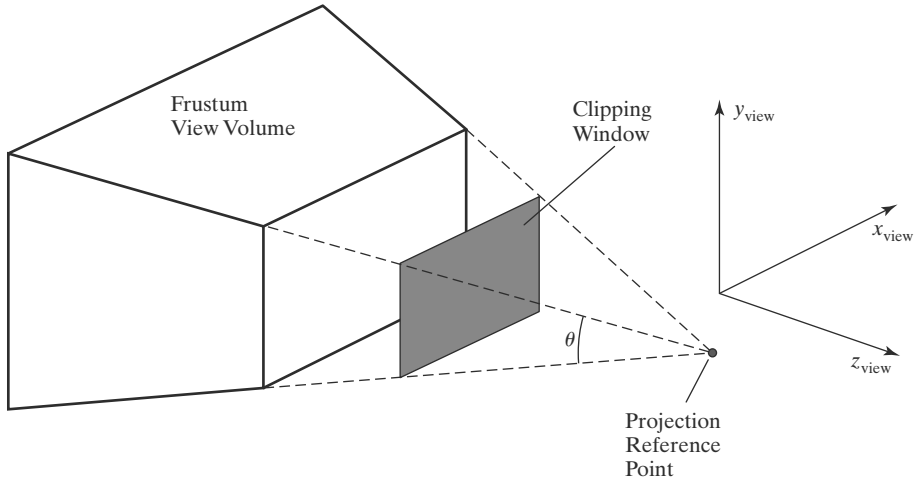


FIGURE 41
Field-of-view angle θ for a symmetric perspective-projection view volume, with the clipping window between the near clipping plane and the projection reference point.

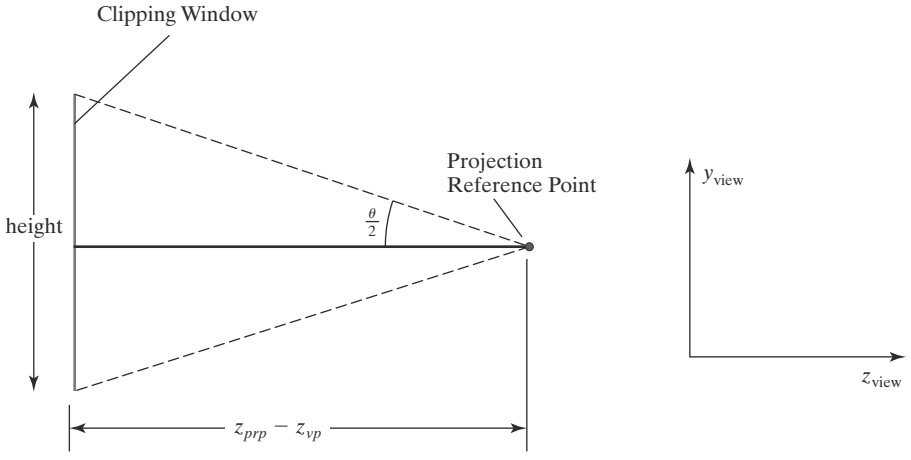


FIGURE 42
Relationship between the field-of-view angle θ, the height of the clipping window, and the distance between the projection reference point and the view plane.

Therefore, the diagonal elements with the value $z_{prp} - z_{vp}$ in matrix 26 could be replaced by either of the following two expressions.

$$z_{prp} - z_{vp} = \frac{\text{height}}{2} \cot\left(\frac{\theta}{2}\right)$$
$$= \frac{\text{width} \cdot \cot(\theta/2)}{2 \cdot \text{aspect}} \tag{29}$$

In some graphics libraries, fixed positions are used for the view plane and the projection reference point, so that a symmetric perspective projection is completely specified by the field-of-view angle, the aspect ratio of the clipping window, and the distances from the viewing position to the near and far clipping planes. The same aspect ratio is usually applied to the specification of the viewport.

If the field-of-view angle is decreased in a particular application, the foreshortening effects of a perspective projection are also decreased. This is comparable to moving the projection reference point farther from the view plane. Also, decreasing the field-of-view angle decreases the height of the clipping window, and this provides a method for zooming in on small regions of a scene. Similarly, a large field-of-view angle results in a large clipping-window height (a zoom out), and it increases perspective effects, which is what we achieve when we set the projection reference point close to the view plane. Figure 43 illustrates the effects of various field-of-view angles for a fixed-width clipping window.

When the perspective-projection view volume is a symmetric frustum, the perspective transformation maps locations inside the frustum to orthogonal-projection coordinates within a rectangular parallelepiped. The centerline of the parallelepiped is the frustum centerline, because this line is already perpendicular to the view plane (Figure 44). This is a consequence of the fact that all positions along a projection line within the frustum map to the same point (x_p, y_p) on the view plane. Thus, each projection line is converted by the perspective transformation to a line that is perpendicular to the view plane and, thus, parallel to the frustum centerline. With the symmetric frustum converted to an orthogonal-projection view volume, we can next apply the normalization transformation.

Oblique Perspective-Projection Frustum

If the centerline of a perspective-projection view volume is not perpendicular to the view plane, we have an **oblique frustum.** Figure 45 illustrates the general

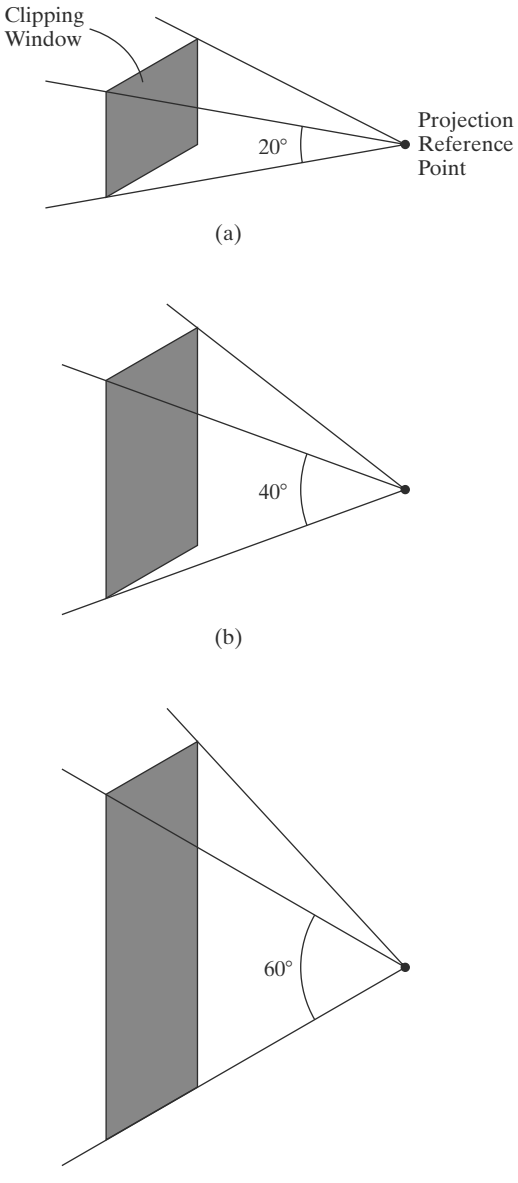


FIGURE 43
Increasing the size of the field-of-view angle increases the height of the clipping window and increases the perspective-projection foreshortening.

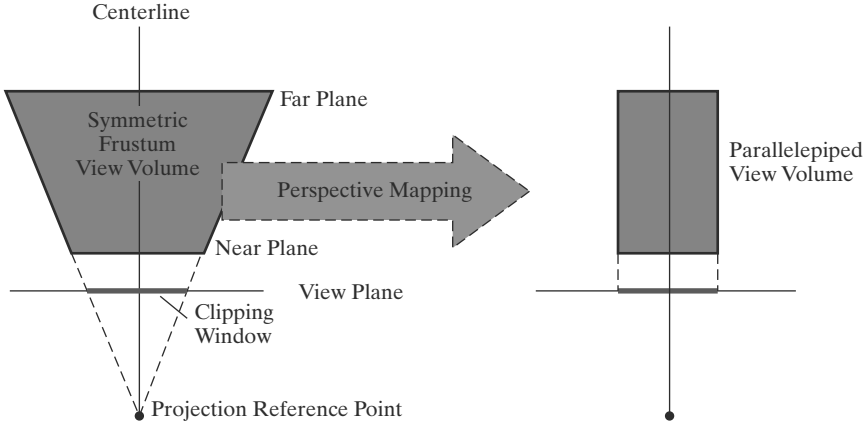


FIGURE 44
A symmetric frustum view volume is mapped to an orthogonal parallelepiped by a perspective-projection transformation.

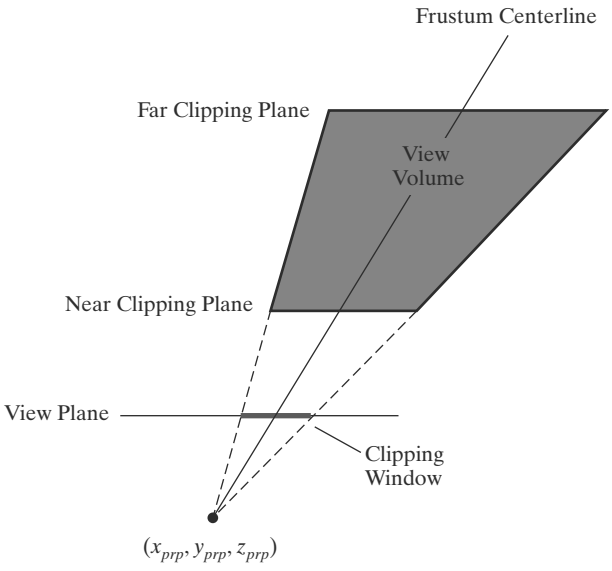


FIGURE 45
An oblique frustum, as viewed from at least one side or a top view, with the view plane positioned between the projection reference point and the near clipping plane.

appearance of an oblique perspective-projection view volume. In this case, we can first transform the view volume to a symmetric frustum and then to a normalized view volume.

An oblique perspective-projection view volume can be converted to a symmetric frustum by applying a z-axis shearing-transformation matrix. This transformation shifts all positions on any plane that is perpendicular to the z axis by an amount that is proportional to the distance of the plane from a specified z-axis reference position. In this case, the reference position is z_{prp}, which is the z coordinate of the projection reference point. And we need to shift by an amount that will move the center of the clipping window to position (x_{prp}, y_{prp}) on the view plane. Because the frustum centerline passes through the center of the clipping window, this shift adjusts the centerline so that it is perpendicular to the view plane, as in Figure 40.

The computations for the shearing transformation, as well as for the perspective and normalization transformations, are greatly reduced if we take the projection reference point to be the viewing-coordinate origin. We could do this with no loss in generality by translating all coordinate positions in a scene so that our selected projection reference point is shifted to the coordinate origin. Or we could have initially set up the viewing-coordinate reference frame so that its origin is at the projection point that we want for a scene. And, in fact, some graphics libraries do fix the projection reference point at the coordinate origin.

Taking the projection reference point as $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$, we obtain the elements of the required shearing matrix as

$$\mathbf{M}_{z\,\text{shear}} = \begin{bmatrix} 1 & 0 & \text{sh}_{zx} & 0 \\ 0 & 1 & \text{sh}_{zy} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{30}$$

We can also simplify the elements of the perspective-projection matrix a bit more if we place the view plane at the position of the near clipping plane. And, because we now want to move the center of the clipping window to coordinates (0, 0)

on the view plane, we need to choose values for the shearing parameters such that

$$\begin{bmatrix} 0 \\ 0 \\ z_{near} \\ 1 \end{bmatrix} = \mathbf{M}_{z\,shear} \cdot \begin{bmatrix} \frac{xw_{min} + xw_{max}}{2} \\ \frac{yw_{min} + yw_{max}}{2} \\ z_{near} \\ 1 \end{bmatrix} \tag{31}$$

Therefore, the parameters for this shearing transformation are

$$\begin{aligned} sh_{zx} &= -\frac{xw_{min} + xw_{max}}{2\, z_{near}} \\ sh_{zy} &= -\frac{yw_{min} + yw_{max}}{2\, z_{near}} \end{aligned} \tag{32}$$

Similarly, with the projection reference point at the viewing-coordinate origin and with the near clipping plane as the view plane, the perspective-projection matrix 26 is simplified to

$$\mathbf{M}_{pers} = \begin{bmatrix} -z_{near} & 0 & 0 & 0 \\ 0 & -z_{near} & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{33}$$

Expressions for the z-coordinate scaling and translation parameters will be determined by the normalization requirements.

Concatenating the simplified perspective-projection matrix 33 with the shear matrix 30, we obtain the following oblique perspective-projection matrix for converting coordinate positions in a scene to homogeneous orthogonal-projection coordinates. The projection reference point for this transformation is the viewing-coordinate origin, and the near clipping plane is the view plane.

$$\begin{aligned} \mathbf{M}_{obliquepers} &= \mathbf{M}_{pers} \cdot \mathbf{M}_{z\,shear} \\ &= \begin{bmatrix} -z_{near} & 0 & \frac{xw_{min} + xw_{max}}{2} & 0 \\ 0 & -z_{near} & \frac{yw_{min} + yw_{max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \end{aligned} \tag{34}$$

Although we no longer have options for the placement of the projection reference point and the view plane, this matrix provides an efficient method for generating a perspective-projection view of a scene without sacrificing a great deal of flexibility.

If we choose the clipping-window coordinates so that $xw_{max} = -xw_{min}$ and $yw_{max} = -yw_{min}$, the frustum view volume is symmetric and matrix 34 reduces to matrix 33. This is because the projection reference point is now at the origin of the viewing-coordinate frame. We could also use Equations 29, with $z_{prp} = 0$ and $z_{vp} = z_{near}$, to express the first two diagonal elements of this matrix in terms of the field-of-view angle and the clipping-window dimensions.

Normalized Perspective-Projection Transformation Coordinates

Matrix 34 transforms object positions in viewing coordinates to perspective-projection homogeneous 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

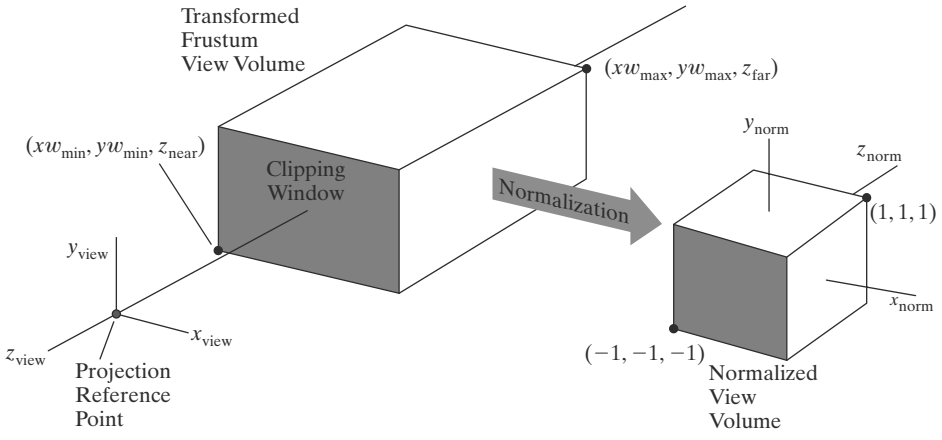


FIGURE 46
Normalization transformation from a transformed perspective-projection view volume (rectangular parallelepiped) to the symmetric normalization cube within a left-handed reference frame, with the near clipping plane as the view plane and the projection reference point at the viewing-coordinate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

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). We have already included the normalization parameters for z coordinates in the perspective-projection matrix 34, but we still 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 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

$$\mathbf{M}_{xy\,\text{scale}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{35}$$

Concatenating the xy-scaling matrix with matrix 34 produces the following normalization matrix for a perspective-projection transformation.

$$\begin{aligned} \mathbf{M}_{\text{normpers}} &= \mathbf{M}_{xy\,\text{scale}} \cdot \mathbf{M}_{\text{obliquepers}} \\ &= \begin{bmatrix} -z_{\text{near}} s_x & 0 & s_x \dfrac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} s_y & s_y \dfrac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \end{aligned} \tag{36}$$

From this transformation, we obtain the homogeneous coordinates:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M}_{\text{normpers}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{37}$$

And the projection coordinates are

$$\begin{aligned} x_p &= \frac{x_h}{h} = \frac{-z_{near} s_x x + s_x(xw_{min} + xw_{max})/2}{-z} \\ y_p &= \frac{y_h}{h} = \frac{-z_{near} s_y y + s_y(yw_{min} + yw_{max})/2}{-z} \\ z_p &= \frac{z_h}{h} = \frac{s_z z + t_z}{-z} \end{aligned} \tag{38}$$

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 $(x, y, z) = (xw_{min}, yw_{min}, z_{near})$, and we want the projection coordinates to be $(x_p, y_p, z_p) = (1, 1, 1)$ when the input coordinates are $(x, y, z) = (xw_{max}, yw_{max}, z_{far})$. Therefore, when we solve equations 38 for the normalization parameters using these conditions, we obtain

$$\begin{aligned} s_x &= \frac{2}{xw_{max} - xw_{min}}, & s_y &= \frac{2}{yw_{max} - yw_{min}} \\ s_z &= \frac{z_{near} + z_{far}}{z_{near} - z_{far}}, & t_z &= \frac{2\, z_{near}\, z_{far}}{z_{near} - z_{far}} \end{aligned} \tag{39}$$

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

$$\mathbf{M}_{normpers} = \begin{bmatrix} \frac{-2z_{near}}{xw_{max} - xw_{min}} & 0 & \frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} & 0 \\ 0 & \frac{-2z_{near}}{yw_{max} - yw_{min}} & \frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} & 0 \\ 0 & 0 & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} & -\frac{2z_{near} z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{40}$$

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, using Equations 29, with the projection reference point at the origin and the view plane at the position of the near clipping plane, we have

$$\mathbf{M}_{normsymmpers} = \begin{bmatrix} \frac{\cot\left(\frac{\theta}{2}\right)}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} & -\frac{2z_{near}\, z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{41}$$

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 $\mathbf{R} \cdot \mathbf{T}$. 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.

9 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.**

The x and y transformation equations from the normalized clipping window to positions within a rectangular viewport are given in matrix 8-10. 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

$$\mathbf{M}_{\text{normviewvol,3D screen}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & 0 & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{42}$$

In normalized coordinates, the $z_{\text{norm}} = -1$ 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 $z_{\text{screen}} = 0$. Thus, the lower-left corner of the viewport screen area is at position $(xv_{\min}, yv_{\min}, 0)$ and the upper-right corner is at position $(xv_{\max}, yv_{\max}, 0)$.

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*. In later chapters, we explore the algorithms for determining the visible surface positions and their colors.

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.

10 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 $\mathbf{P}_0 = (\texttt{x0}, \texttt{y0}, \texttt{z0})$, the reference position as $\mathbf{P}_{\text{ref}} = (\texttt{xref}, \texttt{yref}, \texttt{zref})$, and the view-up vector as $\mathbf{V} = (\texttt{Vx}, \texttt{Vy}, \texttt{Vz})$. The positive z_{view} axis for the viewing frame is in the direction $\mathbf{N} = \mathbf{P}_0 - \mathbf{P}_{\text{ref}}$, and the unit axis vectors for the viewing reference frame are calculated with Equations 1.

Because the viewing direction is along the $-z_{\text{view}}$ axis, the reference position $\mathbf{P}_{\text{ref}}$ 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. And 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 $\mathbf{V}$, which is adjusted to a direction perpendicular to $\mathbf{N}$.

Viewing parameters specified with the `gluLookAt` function are used to form the viewing-transformation matrix 4 that we derived in Section 4. This matrix is formed as a combination of a translation, which shifts the viewing origin to the world origin, and a rotation, which aligns the viewing axes with the world axes.

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

$$\mathbf{P}_0 = (0, 0, 0)$$
$$\mathbf{P}_{\text{ref}} = (0, 0, -1)$$
$$\mathbf{V} = (0, 1, 0)$$

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 `dnear` and `dfar` denote distances in the negative z_{view} direction from the viewing-coordinate origin. For example, if `dfar` $= 55.0$, 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 `dnear` $<$ `dfar`.

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 using matrix 7, with $z_{near} = -$ `dnear` and $z_{far} = -$ `dfar`.

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 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 clipping window, as long as parameters `dnear` and `dfar` 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 `dnear` $= -1.0$ and `dfar` $= 1.0$.

There is no OpenGL function for generating an oblique projection. To produce an oblique-projection view of a scene, we could set up our own projection matrix as in Equation 14. Then we need to make this the current OpenGL projection matrix. Another way to generate an oblique-projection view is to rotate the scene into an appropriate position so that an orthogonal projection in the z_{view} direction yields the desired view.

OpenGL Symmetric Perspective-Projection Function

There are two functions available for producing a perspective-projection view of a scene. One of these functions generates a symmetric frustum view volume about the viewing direction (the negative z_{view} axis). The other function can be used for either a symmetric-perspective projection or an oblique-perspective projection. For both functions, the projection reference point is the viewing-coordinate origin and the near clipping plane is the view plane.

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 `dnear` and `dfar` must be assigned positive numerical values, and the positions of the near and far planes are calculated as $z_{near} = -$`dnear` and $z_{far} = -$`dfar`.

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. And the description of a scene is converted to normalized, homogeneous projection coordinates with matrix 41.

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.

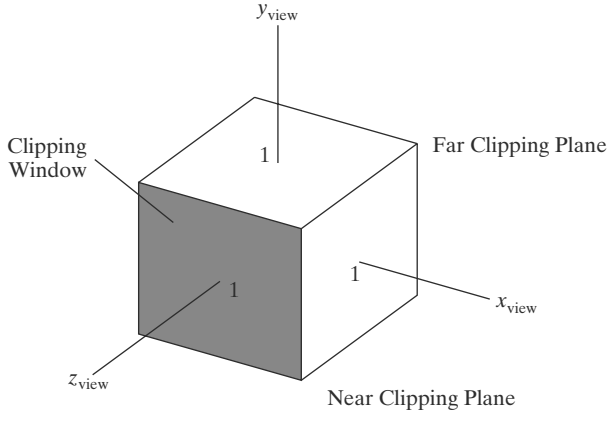


FIGURE 47
Default orthogonal-projection view volume. Coordinate extents for this symmetric cube are from -1 to $+1$ in each direction. The near clipping plane is at $z_{near} = 1$, and the far clipping plane is at $z_{far} = -1$.

```
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 $z_{near} = -\texttt{dnear}$ and $z_{far} = -\texttt{dfar}$.

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 the depth information for each xy position is stored in the depth buffer.

A rectangular viewport is defined with the following OpenGL function.

```
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.

Display windows are created and managed with GLUT routines. The default viewport in OpenGL is the size and position of the current display window.

OpenGL Three-Dimensional Viewing Program Example

A perspective-projection view of a square, as shown in Figure 48, is displayed using the following program example. The square is defined in the xy plane, and a viewing-coordinate origin is selected to view the front face at an angle. Choosing the center of the square as the look-at point, we obtain a perspective view using the `glFrustum` function. If we move the viewing origin around to the other side of the polygon, the back face would be displayed as a wire-frame object.

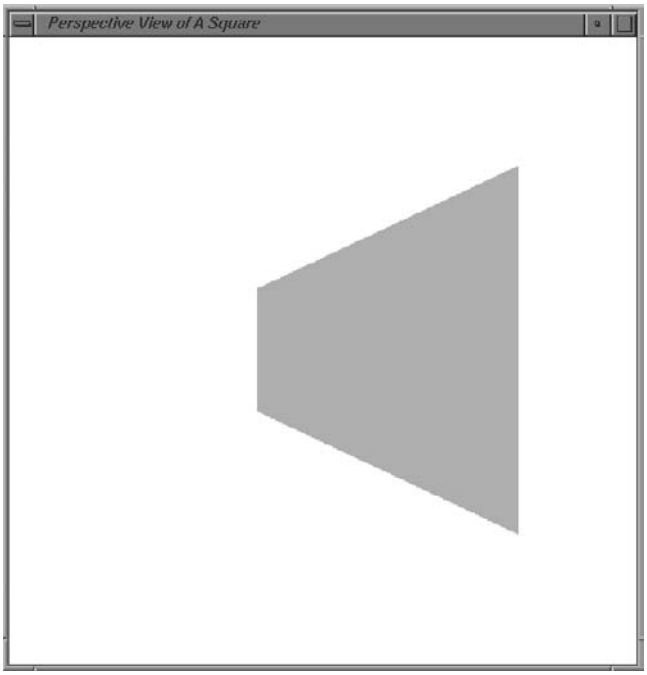


FIGURE 48
Output display generated by the three-dimensional viewing example program.

```
#include <GL/glut.h>

GLint winWidth = 600, winHeight = 600;   //  Initial display-window size.

GLfloat x0 = 100.0, y0 = 50.0, z0 = 50.0;  //  Viewing-coordinate origin.
GLfloat xref = 50.0, yref = 50.0, zref = 0.0;  //  Look-at point.
GLfloat Vx = 0.0, Vy = 1.0, Vz = 0.0;          //  View-up vector.

/*  Set coordinate limits for the clipping window:  */
GLfloat xwMin = -40.0, ywMin = -60.0, xwMax = 40.0, ywMax = 60.0;

/*  Set positions for near and far clipping planes:  */
GLfloat dnear = 25.0, dfar = 125.0;

void init (void)
{
   glClearColor (1.0, 1.0, 1.0, 0.0);

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

   glMatrixMode (GL_PROJECTION);
   glFrustum (xwMin, xwMax, ywMin, ywMax, dnear, dfar);
}

void displayFcn (void)
{
   glClear (GL_COLOR_BUFFER_BIT);
```

```
   /*  Set parameters for a square fill area.  */
   glColor3f (0.0, 1.0, 0.0);         //  Set fill color to green.
   glPolygonMode (GL_FRONT, GL_FILL);
   glPolygonMode (GL_BACK, GL_LINE);  //  Wire-frame back face.
   glBegin (GL_QUADS);
      glVertex3f (0.0, 0.0, 0.0);
      glVertex3f (100.0, 0.0, 0.0);
      glVertex3f (100.0, 100.0, 0.0);
      glVertex3f (0.0, 100.0, 0.0);
   glEnd ( );

   glFlush ( );
}

void reshapeFcn (GLint newWidth, GLint newHeight)
{
   glViewport (0, 0, newWidth, newHeight);

   winWidth = newWidth;
   winHeight = newHeight;
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (50, 50);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Perspective View of A Square");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (reshapeFcn);
   glutMainLoop ( );
}
```

11 Three-Dimensional Clipping Algorithms

Previously, we discussed the advantages of using the normalized boundaries of the clipping window in two-dimensional clipping algorithms. Similarly, we can apply three-dimensional clipping algorithms to the normalized boundaries of the view volume. This allows the viewing pipeline and the clipping procedures to be implemented in a highly efficient way. All device-independent transformations (geometric and viewing) are concatenated and applied before executing the clipping routines. And each of the clipping boundaries for the normalized view volume is a plane that is parallel to one of the Cartesian planes, regardless of the projection type and original shape of the view volume. Depending on whether the view volume has been normalized to a unit cube or to a symmetric cube with edge length 2, the clipping planes have coordinate positions either at 0 and 1 or at -1 and 1. For the symmetric cube, the equations for the three-dimensional

clipping planes are

$$\begin{aligned} xw_{\min} &= -1, \qquad xw_{\max} = 1 \\ yw_{\min} &= -1, \qquad yw_{\max} = 1 \\ zw_{\min} &= -1, \qquad zw_{\max} = 1 \end{aligned} \tag{43}$$

The x and y clipping boundaries are the normalized limits for the clipping window, and the z clipping boundaries are the normalized positions for the near and far clipping planes.

Clipping algorithms for three-dimensional viewing identify and save all object sections within the normalized view volume for display on the output device. All parts of objects that are outside the view-volume clipping planes are eliminated. And the algorithms are now extensions of two-dimensional methods, using the normalized boundary planes of the view volume instead of the straight-line boundaries of the normalized clipping window.

Clipping in Three-Dimensional Homogeneous Coordinates

Computer-graphics libraries process spatial positions as four-dimensional homogeneous coordinates so that all transformations can be represented as 4 by 4 matrices. As each coordinate position enters the viewing pipeline, it is converted to a four-dimensional representation:

$$(x, y, z) \rightarrow (x, y, z, 1)$$

After a position has passed through the geometric, viewing, and projection transformations, it is now in the homogeneous form

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{44}$$

where matrix $\mathbf{M}$ represents the concatenation of all the various transformations from world coordinates to normalized, homogeneous projection coordinates, and the homogeneous parameter h may no longer have the value 1. In fact, h can have any real value, depending on how we represented objects in the scene and the type of projection we used.

If the homogeneous parameter h does have the value 1, the homogeneous coordinates are the same as the Cartesian projection coordinates. This is often the case for a parallel-projection transformation. But a perspective projection produces a homogeneous parameter that is a function of the z coordinate for any spatial position. The perspective-projection homogeneous parameter can even be negative. This occurs when coordinate positions are behind the projection reference point. Also, rational spline representations for object surfaces are often formulated in homogeneous coordinates, where the homogeneous parameter can be positive or negative. Therefore, if clipping is performed in projection coordinates after division by the homogeneous parameter h, some coordinate information can be lost and objects may not be clipped correctly.

An effective method for dealing with all possible projection transformations and object representations is to apply the clipping routines to the homogeneous-coordinate representations of spatial positions. And, because all view volumes can be converted to a normalized cube, a single clipping procedure can be implemented in hardware to clip objects in homogeneous coordinates against the normalized clipping planes.

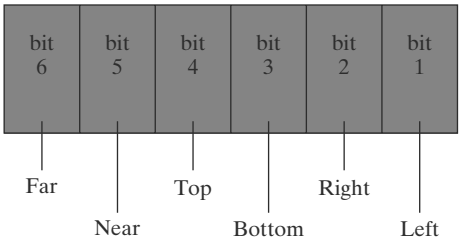


FIGURE 49
A possible ordering for the view-volume clipping boundaries corresponding to the region-code bit positions.

Three-Dimensional Region Codes

We extend the concept of a region code to three dimensions by simply adding a couple of additional bit positions to accommodate the near and far clipping planes. Thus, we now use a six-bit region code, as illustrated in Figure 49. Bit positions in this region-code example are numbered from right to left, referencing the left, right, bottom, top, near, and far clipping planes, in that order.

For a three-dimensional scene, we need to apply the clipping routines to the projection coordinates, which have been transformed to a normalized space. After the projection transformation, each point in a scene has the four-component representation $\mathbf{P} = (xh, yh, zh, h)$. Assuming that we are clipping against the boundaries of the normalized symmetric cube (Eqs. 43), then a point is inside this normalized view volume if the projection coordinates of the point satisfy the following six inequalities:

$$-1 \le \frac{x_h}{h} \le 1, \qquad -1 \le \frac{y_h}{h} \le 1, \qquad -1 \le \frac{z_h}{h} \le 1 \tag{45}$$

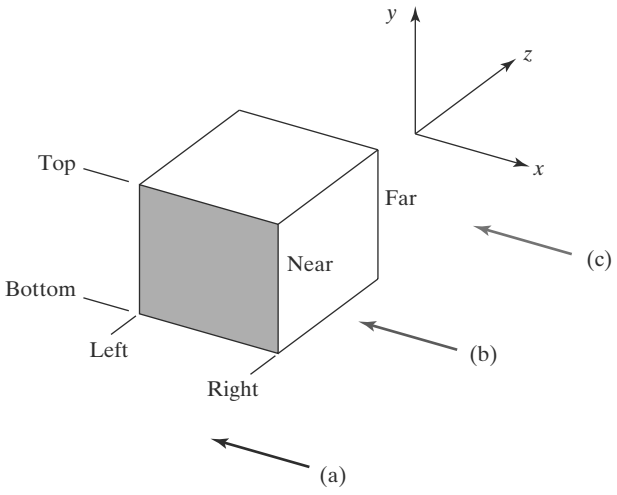
Unless we have encountered an error, the value of the homogeneous parameter h is nonzero. Before implementing region-code procedures, we can first check for the possibility of a homogeneous parameter with either a zero value or an extremely small magnitude. Also, the homogeneous parameter can be either positive or negative. Therefore, assuming $h \neq 0$, we can write the preceding inequalities in the form

$$\begin{aligned} -h \le x_h \le h, \quad -h \le y_h \le h, \quad -h \le z_h \le h \quad &\text{if } h > 0 \\ h \le x_h \le -h, \quad h \le y_h \le -h, \quad h \le z_h \le -h \quad &\text{if } h < 0 \end{aligned} \tag{46}$$

In most cases $h > 0$, and we can then assign the bit values in the region code for a coordinate position according to the tests:

$$\begin{aligned} &\text{bit } 1 = 1 \quad \text{if } h + x_h < 0 \quad \text{(left)} \\ &\text{bit } 2 = 1 \quad \text{if } h - x_h < 0 \quad \text{(right)} \\ &\text{bit } 3 = 1 \quad \text{if } h + y_h < 0 \quad \text{(bottom)} \\ &\text{bit } 4 = 1 \quad \text{if } h - y_h < 0 \quad \text{(top)} \\ &\text{bit } 5 = 1 \quad \text{if } h + z_h < 0 \quad \text{(near)} \\ &\text{bit } 6 = 1 \quad \text{if } h - z_h < 0 \quad \text{(far)} \end{aligned} \tag{47}$$

These bit values can be set using the same approach as in two-dimensional clipping. That is, we simply use the sign bit of one of the calculations $h \pm x_h$, $h \pm y_h$, or $h \pm z_h$ to set the corresponding region-code bit value. Figure 50 lists the 27 region codes for a view volume. In those cases where $h < 0$ for some point, we could apply clipping using the second set of inequalities in 46 or we could negate the coordinates and clip using the tests for $h > 0$.



011001	011000	011010
010001	010000	010010
010101	010100	010110

Region Codes
In Front of Near Plane
(a)

001001	001000	001010
000001	000000	000010
000101	000100	000110

Region Codes
Between Near and Far Planes
(b)

101001	101000	101010
100001	100000	100010
100101	100100	100110

Region Codes
Behind Far Plane
(c)

FIGURE 50
Values for the three-dimensional, six-bit region code that identifies spatial positions relative to the boundaries of a view volume.

Three-Dimensional Point and Line Clipping

For standard point positions and straight-line segments that are defined in a scene that is not behind the projection reference point, all homogeneous parameters are positive and the region codes can be established using the conditions in 47. Then, once we have set up the region code for each position in a scene, we can easily identify a point position as outside the view volume or inside the view volume. For instance, a region code of 101000 tells us that the point is above and directly behind the view volume, while the region code 000000 indicates a point within the volume (Figure 50). Thus, for point clipping, we simply eliminate any individual point whose region code is not 000000. In other words, if any one of the tests in 47 is negative, the point is outside the view volume.

Methods for three-dimensional line clipping are essentially the same as for two-dimensional lines. We can first test the line endpoint region codes for trivial acceptance or rejection of the line. If the region code for both endpoints of a line is 000000, the line is completely inside the view volume. Equivalently, we can trivially accept the line if the logical *or* operation on the two endpoint region codes produces a value of 0. And we can trivially reject the line if the logical *and* operation on the two endpoint region codes produces a value that is not 0. This nonzero value indicates that both endpoint region codes have a 1 value in the same bit position, and hence the line is completely outside one of the clipping planes. As an example of this, the line from $\mathbf{P}_3$ to $\mathbf{P}_4$ in Figure 51 has the endpoint region-code values of 010101 and 100110. So this line is completely below the bottom clipping plane. If a line fails these two tests, we next analyze the line equation to determine whether any part of the line should be saved.

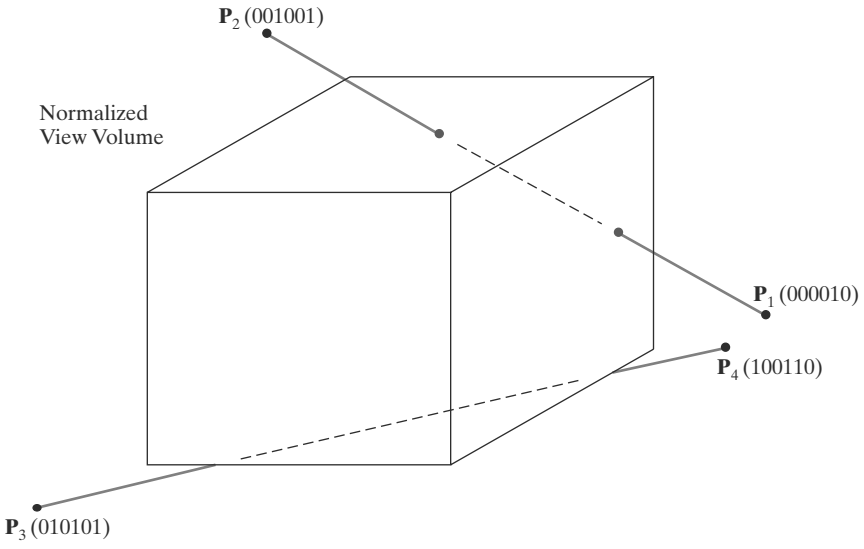


FIGURE 51
Three-dimensional region codes for two line segments. Line $\overline{P_1P_2}$ intersects the right and top clipping boundaries of the view volume, while line $\overline{P_3P_4}$ is completely below the bottom clipping plane.

Equations for three-dimensional line segments are conveniently expressed in parametric form, and the clipping methods of Cyrus-Beck or Liang-Barsky can be extended to three-dimensional scenes. For a line segment with endpoints $\mathbf{P}_1 = (x_{h1}, y_{h1}, z_{h1}, h_1)$ and $\mathbf{P}_2 = (x_{h2}, yh_2, zh_2, h_2)$, we can write the parametric equation describing any point position along the line as

$$\mathbf{P} = \mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u \qquad 0 \le u \le 1 \tag{48}$$

When the line parameter has the value $u = 0$, we are at position $\mathbf{P}_1$. And $u = 1$ brings us to the other end of the line, $\mathbf{P}_2$. Writing the parametric line equation explicitly, in terms of the homogeneous coordinates, we have

$$\begin{aligned} x_h &= x_{h1} + (x_{h2} - x_{h1})u \\ y_h &= y_{h1} + (y_{h2} - y_{h1})u \\ z_h &= z_{h1} + (z_{h2} - z_{h1})u \\ h &= h_1 + (h_2 - h_1)u \end{aligned} \qquad 0 \le u \le 1 \tag{49}$$

Using the endpoint region codes for a line segment, we can first determine which clipping planes are intersected. If one of the endpoint region codes has a 0 value in a certain bit position while the other code has a 1 value in the same bit position, then the line crosses that clipping boundary. In other words, one of the tests in 47 generates a negative value, while the same test for the other endpoint of the line produces a nonnegative value. To find the intersection position with this clipping plane, we first use the appropriate equations in 49 to determine the corresponding value of parameter u. Then we calculate the intersection coordinates.

As an example of the intersection-calculation procedure, we consider the line segment $\overline{P_1P_2}$ in Figure 51. This line intersects the right clipping plane, which can be described with the equation $x_{\max} = 1$. Therefore, we determine the intersection value for parameter u by setting the x-projection coordinate equal to 1:

$$x_p = \frac{x_h}{h} = \frac{x_{h1} + (x_{h2} - x_{h1})u}{h_1 + (h_2 - h_1)u} = 1 \tag{50}$$

Solving for parameter u, we obtain

$$u = \frac{x_{h1} - h_1}{(x_{h1} - h_1) - (x_{h2} - h_2)} \tag{51}$$

Next, we determine the values y_p and z_p on this clipping plane, using the calculated value for u. In this case, the y_p and z_p intersection values are within the ± 1 boundaries of the view volume and the line does cross into the view-volume interior. So we next proceed to locate the intersection position with the top clipping plane. That completes the processing for this line segment, because the intersection points with the top and right clipping planes identify the part of the line that is inside the view volume and all the line sections that are outside the view volume.

When a line intersects a clipping boundary but does not enter the view-volume interior, we continue the line processing as in two-dimensional clipping. The section of the line outside that clipping boundary is eliminated, and we update the region-code information and the values for parameter u for the part of the line inside that boundary. Then we test the remaining section of the line against the other clipping planes for possible rejection or for further intersection calculations.

Line segments in three-dimensional scenes are usually not isolated. They are most often components in the description for the solid objects in the scene, and we need to process the lines as part of the surface-clipping routines.

Three-Dimensional Polygon Clipping

Graphics packages typically deal only with scenes that contain "graphics objects." These are objects whose boundaries are described with linear equations, so that each object is composed of a set of surface polygons. Therefore, to clip objects in a three-dimensional scene, we apply the clipping routines to the polygon surfaces. Figure 52, for example, highlights the surface sections of a pyramid that are to be clipped, and the dashed lines show sections of the polygon surfaces that are inside the view volume.

We can first test a polyhedron for trivial acceptance or rejection using its coordinate extents, a bounding sphere, or some other measure of its coordinate limits. If the coordinate limits of the object are inside all clipping boundaries, we save the entire object. If the coordinate limits are all outside any one of the clipping boundaries, we eliminate the entire object.

When we cannot save or eliminate the entire object, we can next process the vertex lists for the set of polygons that define the object surfaces. Applying

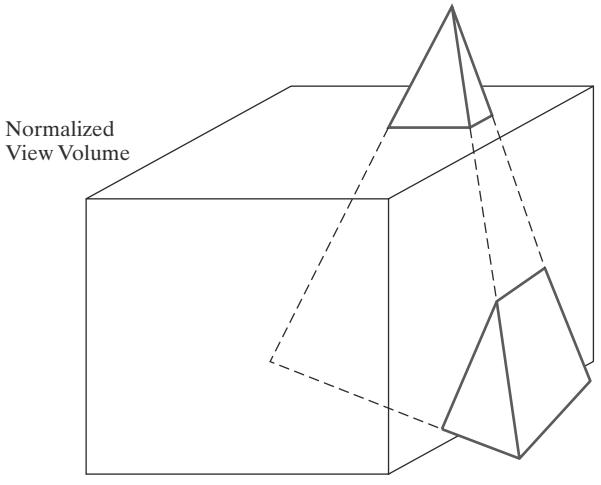


FIGURE 52
Three-dimensional object clipping. Surface sections that are outside the view-volume clipping planes are eliminated from the object description, and new surface facets may need to be constructed.

methods similar to those in two-dimensional polygon clipping, we can clip edges to obtain new vertex lists for the object surfaces. We may also need to create some new vertex lists for additional surfaces that result from the clipping operations. And the polygon tables are updated to add any new polygon surfaces and to revise the connectivity and shared-edge information about the surfaces.

To simplify the clipping of general polyhedra, polygon surfaces are often divided into triangular sections and described with triangle strips. We can then clip the triangle strips using the Sutherland-Hodgman approach. Each triangle strip is processed in turn against the six clipping planes to obtain the final vertex list for the strip.

For concave polygons, we can apply splitting methods to obtain a set of triangles, for example, and then clip the triangles. Alternatively, we could clip three-dimensional concave polygons using the Weiler-Atherton algorithm.

Three-Dimensional Curve Clipping

As in polyhedra clipping, we first check to determine whether the coordinate extents of a curved object, such as a sphere or a spline surface, are completely inside the view volume. Then we can check to determine whether the object is completely outside any one of the six clipping planes.

If the trivial rejection-acceptance tests fail, we locate the intersections with the clipping planes. To do this, we solve the simultaneous set of surface equations and the clipping-plane equation. For this reason, most graphics packages do not include clipping routines for curved objects. Instead, curved surfaces are approximated as a set of polygon patches, and the objects are then clipped using polygon-clipping routines. When surface-rendering procedures are applied to polygon patches, they can provide a highly realistic display of a curved surface.

Arbitrary Clipping Planes

It is also possible, in some graphics packages, to clip a three-dimensional scene using additional planes that can be specified in any spatial orientation. This option is useful in a variety of applications. For example, we might want to isolate or clip off an irregularly shaped object, eliminate part of a scene at an oblique angle for a special effect, or slice off a section of an object along a selected axis to show a cross-sectional view of its interior.

Optional clipping planes can be specified along with the description of a scene, so that the clipping operations can be performed prior to the projection transformation. However, this also means that the clipping routines are implemented in software.

A clipping plane can be specified with the plane parameters A, B, C, and D. The plane then divides three-dimensional space into two parts, so that all parts of a scene that lie on one side of the plane are clipped off. Assuming that objects behind the plane are to be clipped, then any spatial position (x, y, z) that satisfies the following inequality is eliminated from the scene.

$$Ax + By + Cz + D < 0 \tag{52}$$

As an example, if the plane-parameter array has the values $(A, B, C, D) = (1.0, 0.0, 0.0, 8.0)$, then any coordinate position satisfying $x + 8.0 < 0.0$ (or, $x < -8.0$) is clipped from the scene.

To clip a line segment, we can first test its two endpoints to see if the line is completely behind the clipping plane or completely in front of the plane. We can represent inequality 52 in a vector form using the plane normal vector

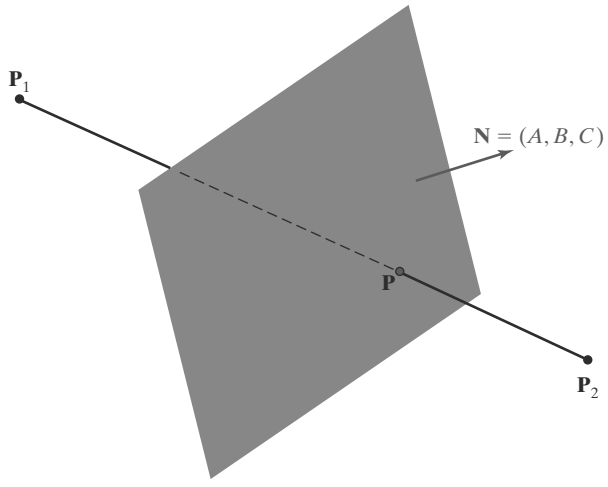


FIGURE 53
Clipping a line segment against a plane with normal vector **N**.

$\mathbf{N} = (A, B, C)$. Then, for a line segment with endpoint positions $\mathbf{P}_1$ and $\mathbf{P}_2$, we clip the entire line if both endpoints satisfy

$$\mathbf{N} \cdot \mathbf{P}_k + D < 0, \qquad k = 1, 2 \tag{53}$$

We save the entire line if both endpoints satisfy

$$\mathbf{N} \cdot \mathbf{P}_k + D \geq 0, \qquad k = 1, 2 \tag{54}$$

Otherwise, the endpoints are on opposite sides of the clipping plane, as in Figure 53, and we calculate the line intersection point.

To calculate the line-intersection point with the clipping plane, we can use the following parametric representation for the line segment:

$$\mathbf{P} = \mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u, \qquad 0 \leq u \leq 1 \tag{55}$$

Point $\mathbf{P}$ is on the clipping plane if it satisfies the plane equation

$$\mathbf{N} \cdot \mathbf{P} + D = 0 \tag{56}$$

Substituting the expression for $\mathbf{P}$ from Equation 55, we have

$$\mathbf{N} \cdot [\mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u] + D = 0 \tag{57}$$

Solving this equation for parameter u, we obtain

$$u = \frac{-D - \mathbf{N} \cdot \mathbf{P}_1}{\mathbf{N} \cdot (\mathbf{P}_2 - \mathbf{P}_1)} \tag{58}$$

We then substitute this value of u into the vector parametric line representation 55 to obtain values for the x, y, and z intersection coordinates. For the example in Figure 53, the line segment from $\mathbf{P}_1$ to $\mathbf{P}$ is clipped and we save the section of the line from $\mathbf{P}$ to $\mathbf{P}_2$.

For polyhedra, such as the pyramid in Figure 54, we apply similar clipping procedures. We first test to see if the object is completely behind or completely in front of the clipping plane. If not, we process the vertex list for each polygon surface. Line-clipping methods are applied to each polygon edge in succession, just as in view-volume clipping, to produce the surface vertex lists. But in this case, we have to deal with only one clipping plane.

Clipping a curved object against a single clipping plane is easier than clipping the object against the six planes of a view volume. However, we still need to solve a set of nonlinear equations to locate intersections, unless we approximate the curve boundaries with straight-line sections.

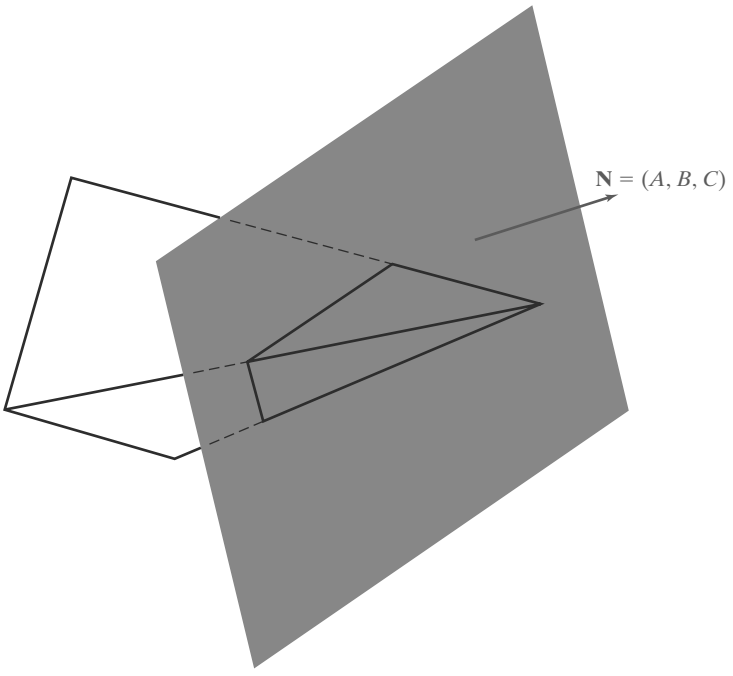


FIGURE 54
Clipping the surfaces of a pyramid against a plane with normal vector **N**. The surfaces in front of the plane are saved, and the surfaces of the pyramid behind the plane are eliminated.

12 OpenGL Optional Clipping Planes

In addition to the six clipping planes enclosing the view volume, OpenGL provides for the specification of additional clipping planes in a scene. Unlike the view-volume clipping planes, which are each perpendicular to one of the coordinate axes, these additional planes can have any orientation.

We designate an optional clipping plane and activate clipping against that plane with the statements

```
glClipPlane (id, planeParameters);
glEnable (id);
```

Parameter `id` is used as an identifier for a clipping plane. This parameter is assigned one of the values `GL_CLIP_PLANE0`, `GL_CLIP_PLANE1`, and so forth, up to a facility-defined maximum. The plane is then defined using the four-element array `planeParameters`, whose elements are the double-precision, floating-point values for the four plane-equation parameters A, B, C, and D. An activated clipping plane that has been assigned the identifier `id` is turned off with

```
glDisable (id);
```

The plane parameters A, B, C, and D are transformed to viewing coordinates and used to test viewing-coordinate positions in a scene. Subsequent changes in viewing or geometric-transformation parameters do not affect the stored plane parameters. Therefore, if we set up optional clipping planes before specifying any geometric or viewing transformations, the stored plane parameters are the same as the input parameters. Also, because the clipping routines for these planes are applied in viewing coordinates, and not in the normalized coordinate space, the performance of a program can be degraded when optional clipping planes are activated.

Any points that are "behind" an activated OpenGL clipping plane are eliminated. Thus, a viewing-coordinate position (x, y, z) is clipped if it satisfies condition 52.

Six optional clipping planes are available in any OpenGL implementation, but more might be provided. We can find out how many optional clipping planes are possible for a particular OpenGL implementation with the inquiry

```
glGetIntegerv (GL_MAX_CLIP_PLANES, numPlanes);
```

Parameter `numPlanes` is the name of an integer array that is to be assigned an integer value equal to the number of optional clipping planes that we can use.

The default for the `glClipPlane` function is that the clipping-plane parameters A, B, C, and D are each assigned a value of 0 for all optional planes. And, initially, all optional clipping planes are disabled.

13 Summary

Viewing procedures for three-dimensional scenes follow the general approach used in two-dimensional viewing. We first create a world-coordinate scene, either from the definitions of objects in modeling coordinates or directly in world coordinates. Then we set up a viewing-coordinate reference frame and transfer object descriptions from world coordinates to viewing coordinates. Object descriptions are then processed through various routines to device coordinates.

Unlike two-dimensional viewing, however, three-dimensional viewing requires projection routines to transform object descriptions to a viewing plane before the transformation to device coordinates. Also, three-dimensional viewing operations involve more spatial parameters. We can use the camera analogy to describe three-dimensional viewing parameters. A viewing-coordinate reference frame is established with a view reference point (the camera position), a view-plane normal vector $\mathbf{N}$ (the camera lens direction), and a view-up vector $\mathbf{V}$ (the camera up direction). The view-plane position is then established along the viewing z axis, and object descriptions are projected to this plane. Either parallel-projection or perspective-projection methods can be used to transfer object descriptions to the view plane.

Parallel projections are either orthographic or oblique, and they can be specified with a projection vector. Orthographic parallel projections that display more than one face of an object are called axonometric projections. An isometric view of an object is obtained with an axonometric projection that foreshortens each principal axis by the same amount. Commonly used oblique projections are the cavalier projection and the cabinet projection. Perspective projections of objects are obtained with projection lines that meet at the projection reference point. Parallel projections maintain object proportions, but perspective projections decrease the size of distant objects. Perspective projections cause parallel lines to appear to converge to a vanishing point, provided the lines are not parallel to the view plane. Engineering and architectural displays can be generated with one-point, two-point, or three-point perspective projections, depending on the number of principal axes that intersect the view plane. An oblique perspective projection is obtained when the line from the projection reference point to the center of the clipping window is not perpendicular to the view plane.

Objects in a three-dimensional scene can be clipped against a view volume to eliminate unwanted sections of the scene. The top, bottom, and sides of the view volume are formed with planes that are parallel to the projection lines and that pass through the clipping-window edges. Near and far planes (also called front and back planes) are used to create a closed view volume. For a parallel

TABLE 1

Summary of OpenGL Three-Dimensional Viewing Functions

Function	Description
`gluLookAt`	Specifies three-dimensional viewing parameters.
`glOrtho`	Specifies parameters for a clipping window and the near and far clipping planes for an orthogonal projection.
`gluPerspective`	Specifies field-of-view angle and other parameters for a symmetric perspective projection.
`glFrustum`	Specifies parameters for a clipping window and near and far clipping planes for a perspective projection (symmetric or oblique).
`glClipPlane`	Specifies parameters for an optional clipping plane.

projection, the view volume is a parallelepiped. For a perspective projection, the view volume is a frustum. In either case, we can convert the view volume to a normalized cube with boundaries either at 0 and 1 for each coordinate or at -1 and 1 for each coordinate. Efficient clipping algorithms process objects in a scene against the bounding planes of the normalized view volume. Clipping is generally carried out in graphics packages in four-dimensional homogeneous coordinates following the projection and view-volume normalization transformations. Then, homogeneous coordinates are converted to three-dimensional, Cartesian projection coordinates. Additional clipping planes, with arbitrary orientations, can also be used to eliminate selected parts of a scene or to produce special effects.

A function is available in the OpenGL Utility library for specifying three-dimensional viewing parameters (see Table 1). This library also includes a function for setting up a symmetric perspective-projection transformation. Three other viewing functions are available in the OpenGL basic library for specifying an orthographic projection, a general perspective projection, and optional clipping planes. Table 1 summarizes the OpenGL viewing functions discussed in this chapter. In addition, the table lists some viewing-related functions.

REFERENCES

Discussions of three-dimensional viewing and clipping algorithms can be found in Weiler and Atherton (1977), Weiler (1980), Cyrus and Beck (1978), and Liang and Barsky (1984). Homogeneous-coordinate clipping algorithms are described in Blinn and Newell (1978), Riesenfeld (1981), and Blinn (1993, 1996, and 1998). Various programming techniques for three-dimensional viewing are discussed in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995).

A complete listing of three-dimensional OpenGL viewing functions is given in Shreiner (2000). For OpenGL programming examples using three-dimensional viewing, see Woo, et al. (1999). Additional programming examples can be found at Nate Robins's tutorial website: www.xmission.com/~nate/opengl.html.

EXERCISES

1 Write a procedure to set up the matrix that transforms world-coordinate positions to three-dimensional viewing coordinates, given $\mathbf{P}_0$, $\mathbf{N}$, and $\mathbf{V}$. The view-up vector can be in any direction that is not parallel to $\mathbf{N}$.

2 Write a procedure to transform the vertices of a polyhedron to projection coordinates using a parallel projection with any specified projection vector.

3 Write a program to obtain different parallel-projection views of a polyhedron by allowing the user to rotate the polyhedron via the keyboard.

4 Write a procedure to perform a one-point perspective projection of an object.

5 Write a procedure to perform a two-point perspective projection of an object.

6 Develop a routine to perform a three-point perspective projection of an object.

7 Write a program that uses the routines in the previous three exercises to display a three-dimensional cube using a one-, two-, or three-point perspective projection according to input taken from the keyboard, which should be used to switch between projections. The program should also allow the user to rotate the cube in the xz plane around its center. Examine the visual differences of the three different types of projections.

8 Write a routine to convert a perspective projection frustum to a regular parallelepiped.

9 Modify the two-dimensional Cohen-Sutherland line-clipping algorithm to clip three-dimensional lines against the normalized symmetric view volume square.

10 Write a program to generate a set of 10 random lines, each of which has one endpoint within a normalized symmetric view volume and one without. Implement the three-dimensional Cohen-Sutherland line-clipping algorithm designed in the previous exercise to clip the set of lines against the viewing volume.

11 Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip three-dimensional lines against a specified regular parallelepiped.

12 Write a program similar to that in Exercise 10 that generates a set of 10 random lines, each partially outside of a specified regular parallelepiped viewing volume. Use the three-dimensional Liang-Barsky line-clipping algorithm developed in the previous exercise to clip the lines against the viewing volume.

13 Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip a given polyhedron against a specified regular parallelepiped.

14 Write a program to display a cube in a regular parallelepiped viewing volume and allow the user to translate the cube along each axis using keyboard input. Implement the algorithm in the previous exercise to clip the cube when it extends over any of the edges of the viewing volume.

15 Write a routine to perform line clipping in homogeneous coordinates.

16 Devise an algorithm to clip a polyhedron against a defined frustum. Compare the operations needed in this algorithm to those needed in an algorithm that clips against a regular parallelepiped.

17 Extend the Sutherland-Hodgman polygon-clipping algorithm to clip a convex polyhedron against a normalized symmetric view volume.

18 Write a routine to implement the preceding exercise.

19 Write a program similar to the one in Exercise 14 to display a cube in a normalized symmetric view volume that can be translated around the viewing volume via keyboard input. Use the implementation of the polygon-clipping algorithm developed in the previous exercise to clip the cube when it extends over the edge of the viewing volume.

20 Write a routine to perform polyhedron clipping in homogeneous coordinates.

21 Modify the program example in Section 10 to allow a user to specify a view for either the front or the back of the square.

22 Modify the program example in Section 10 to allow the perspective viewing parameters to be specified as user input.

23 Modify the program example in Section 10 to produce a view of any input polyhedron.

24 Modify the program in the preceding exercise to generate a view of the polyhedron using an orthographic projection.

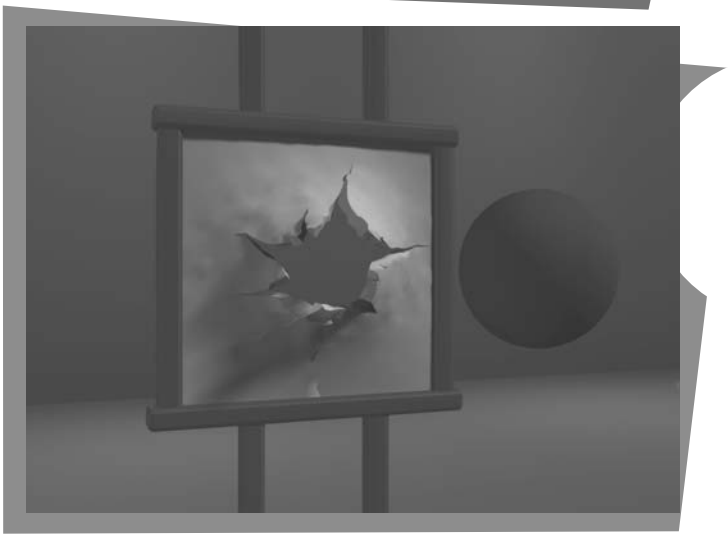
25 Modify the program in the preceding exercise to generate a view of the polyhedron using an oblique parallel projection.

IN MORE DEPTH

1 In this exercise, you will give "depth" to the polygons that represent the objects in your scene and clip them against a normalized view volume. First, choose new z-coordinates and three-dimensional orientations for the polygons in your scene that are appropriate to the snapshot of your application. That is, they should be taken out of the xy plane in which they have been constrained so far and given appropriate depth. Once you have done this, implement an extension of the Sutherland-Hodgman polygon-clipping algorithm that allows clipping of convex polygons against a normalized symmetric view volume. You will use this algorithm in the next exercise to produce a view of some portion of your three-dimensional scene.

2 Choose a view of the scene from the previous exercise that produces a view volume in which all objects are not fully contained. Apply the algorithm for polygon clipping that you developed in the previous exercise against the view volume. Write routines to display the scene using a parallel projection and a perspective projection. Use the OpenGL three-dimensional viewing functions to do this, choosing appropriate parameters to specify the viewing volume in each case. Allow the user to switch between the two projections via keyboard input and note the differences in the visual appearance of the scene in the two cases.

Visible-Surface Detection Methods



A major consideration in the generation of realistic graphics displays is determining what is visible within a scene from a chosen viewing position. There are a number of approaches we can take to accomplish this, and numerous algorithms have been devised for efficient identification and display of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Which method we select for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. The various algorithms are referred to as **visible-surface detection** methods. Sometimes these methods are also referred to as **hidden-surface elimination** methods, although there can be subtle differences between identifying visible surfaces and eliminating hidden surfaces. With a wire-frame display, for example, we may not want to eliminate the hidden surfaces, but rather to display them with dashed boundaries or in some other way to retain information about their shape.

1 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. Line-display algorithms, for instance, generally use object-space methods to identify visible lines in wire-frame displays, but many image-space visible-surface algorithms can be adapted easily to visible-line detection.

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.

2 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

$$Ax + By + Cz + D < 0 \tag{1}$$

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

$$\mathbf{V}_{\text{view}} \cdot \mathbf{N} > 0 \tag{2}$$

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

In a right-handed viewing system with the viewing direction along the negative z_v axis (Figure 2), a polygon is a back face if the z component, C, of its normal vector $\mathbf{N}$ 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

$$C \leq 0 \tag{3}$$

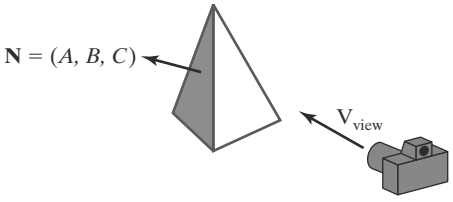


FIGURE 1
A surface normal vector **N** and the viewing-direction vector $\mathbf{V}_{view}$.

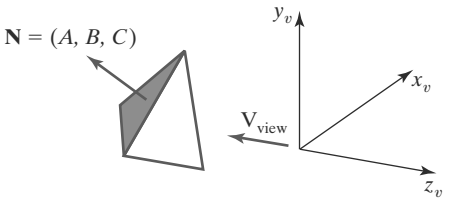


FIGURE 2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

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 1 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 $C \geq 0$ 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. Also, if a scene contains only nonoverlapping convex polyhedra, then again all hidden surfaces are identified with the back-face method.

For other objects, such as the concave polyhedron in Figure 3, more tests must be carried out to determine whether there are additional faces that are totally or partially obscured by other faces. A general scene can be expected to contain overlapping objects along the line of sight, and we then need to determine where the obscured objects are partly or completely hidden by other objects. In general, back-face removal can be expected to eliminate about half of the polygon surfaces in a scene from further visibility tests.

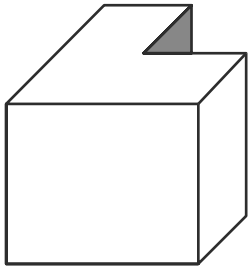


FIGURE 3
View of a concave polyhedron with one face partially hidden by other faces of the object.

3 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 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

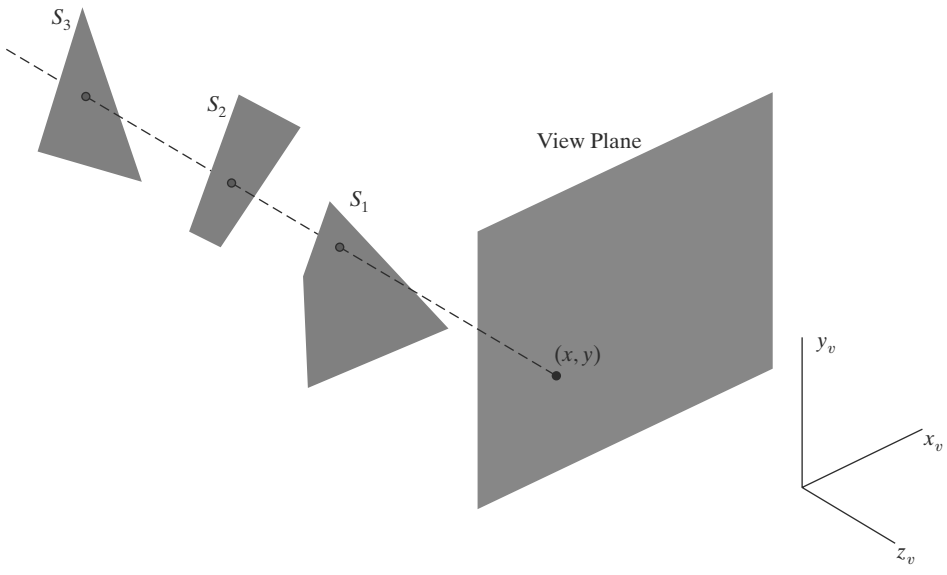


FIGURE 4
Three surfaces overlapping pixel position (x, y) on the view plane. The visible surface, S_1, has the smallest depth value.

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

$$z = \frac{-Ax - By - D}{C} \tag{4}$$

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' of the next position $(x + 1, y)$ along the scan line is obtained from Eq. 4 as

$$z' = \frac{-A(x+1) - By - D}{C} \tag{5}$$

or

$$z' = z - \frac{A}{C} \tag{6}$$

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.

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 Eq. 6.

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

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

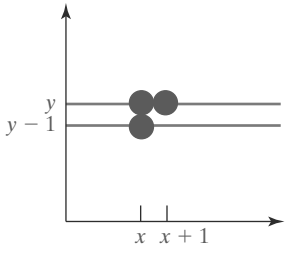


FIGURE 5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

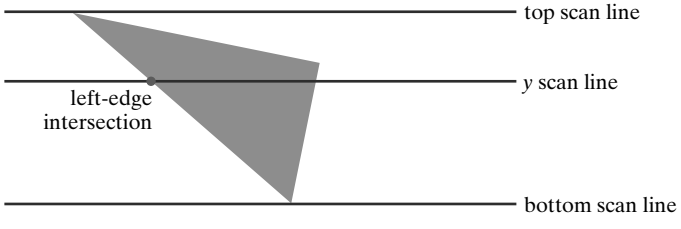


FIGURE 6
Scan lines intersecting a polygon surface.

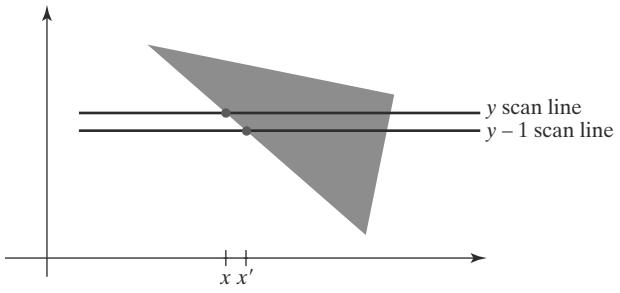


FIGURE 7
Intersection positions on successive scan lines along a left polygon edge.

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

$$z' = z + \frac{A/m + B}{C} \tag{7}$$

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.

An alternative approach is to use a midpoint method or Bresenham-type algorithm for determining the starting x values along edges for each scan line. The method can be applied to curved surfaces by determining depth and color values at each surface projection point.

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene. But it does require the availability of a second buffer in addition to the refresh buffer. A system with a resolution of 1280×1024, for example, would require over 1.3 million positions in the depth buffer, with each position containing enough bits to represent the number of depth increments needed. One way to reduce storage requirements is to process one section of the scene at a time, using a smaller depth buffer. After each view section is processed, the buffer is reused for the next section.

In addition, the basic depth-buffer algorithm often performs needless calculations. Objects are processed in an arbitrary order, so that a color can be computed for a surface point that is later replaced by a closer surface. To alleviate this problem, some graphics packages provide options that allow a user to adjust the depth range for surface testing. This allows distant objects, for example, to be excluded from the depth tests. Using this option, we could even exclude objects that are very close to the projection plane. Hardware implementations of the depth-buffer algorithm are typically an integral component of sophisticated computer-graphics systems.

4 A-Buffer Method

An extension of the depth-buffer ideas is the **A-buffer** procedure (at the other end of the alphabet from "z-buffer," where z represents depth). This depth-buffer extension is an antialiasing, area-averaging, visibility-detection method developed at Lucasfilm Studios for inclusion in the surface-rendering system called

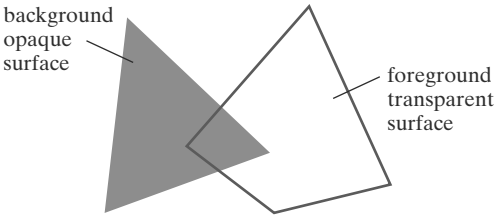


FIGURE 8
Viewing an opaque surface through a transparent surface requires multiple color inputs and the application of color-blending operations.

REYES (an acronym for "Renders Everything You Ever Saw"). The buffer region for this procedure is referred to as the *accumulation buffer,* because it is used to store a variety of surface data, in addition to depth values.

A drawback of the depth-buffer method is that it identifies only one visible surface at each pixel position. In other words, it deals only with opaque surfaces and cannot accumulate color values for more than one surface, as is necessary if transparent surfaces are to be displayed (Figure 8). The A-buffer method expands the depth-buffer algorithm so that each position in the buffer can reference a linked list of surfaces. This allows a pixel color to be computed as a combination of different surface colors for transparency or antialiasing effects.

Each position in the A-buffer has two fields:

- Depth field: Stores a real-number value (positive, negative, or zero).
- Surface data field: Stores surface data or a pointer.

If the depth field is nonnegative, the number stored at that position is the depth of a surface that overlaps the corresponding pixel area. The surface data field then stores various surface information, such as the surface color for that position and the percent of pixel coverage, as illustrated in Figure 9(a). If the depth field for a position in the A-buffer is negative, this indicates multiple-surface contributions to the pixel color. The color field then stores a pointer to a linked list of surface data, as in Figure 9(b). Surface information in the A-buffer includes

- RGB intensity components
- Opacity parameter (percent of transparency)
- Depth
- Percent of area coverage
- Surface identifier
- Other surface-rendering parameters

The A-buffer visibility-detection scheme can be implemented using methods similar to those in the depth-buffer algorithm. Scan lines are processed to

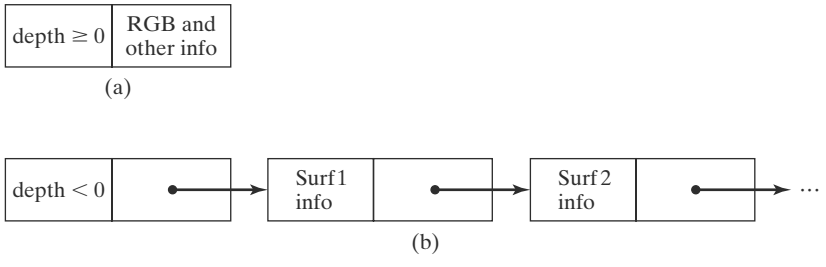


FIGURE 9
Two possible organizations for surface information in an A-buffer representation for a pixel position. When a single surface overlaps the pixel, the surface depth, color, and other information are stored as in (a). When more than one surface overlaps the pixel, a linked list of surface data is stored as in (b).

determine how much of each surface covers each pixel position across the individual scan lines. Surfaces are subdivided into a polygon mesh and clipped against the pixel boundaries. Using the opacity factors and percent of surface coverage, the rendering algorithms calculate the color for each pixel as an average of the contributions from the overlapping surfaces.

5 Scan-Line Method

This image-space method for identifying visible surfaces computes and compares depth values along the various scan lines for a scene. As each scan line is processed, all polygon surface projections intersecting that line are examined to determine which are visible. Across each scan line, depth calculations are performed to determine which surface is nearest to the view plane at each pixel position. When the visible surface has been determined for a pixel, the surface color for that position is entered into the frame buffer.

Surfaces are processed using the information stored in the polygon tables. The edge table contains coordinate endpoints for each line in the scene, the inverse slope of each line, and pointers into the surface-facet table to identify the surfaces bounded by each line. The surface-facet table contains the plane coefficients, surface material properties, other surface data, and possibly pointers into the edge table. To facilitate the search for surfaces crossing a given scan line, an active list of edges is formed for each scan line as it is processed. The active edge list contains only those edges that cross the current scan line, sorted in order of increasing x. In addition, we define a flag for each surface that is set to "on" or "off" to indicate whether a position along a scan line is inside or outside the surface. Pixel positions across each scan line are processed from left to right. At the left intersection with the surface projection of a convex polygon, the surface flag is turned on; at the right intersection point along the scan line, it is turned off. For a concave polygon, scan-line intersections can be sorted from left to right, with the surface flag set to "on" between each intersection pair.

Figure 10 illustrates the scan-line method for locating visible portions of surfaces for pixel positions along a scan line. The active list for scan line 1 contains information from the edge table for edges AB, BC, EH, and FG. For positions along this scan line between edges AB and BC, only the flag for surface S_1 is on. Therefore, no depth calculations are necessary, and color values are calculated from the surface properties and lighting conditions for surface S_1. Similarly, between edges

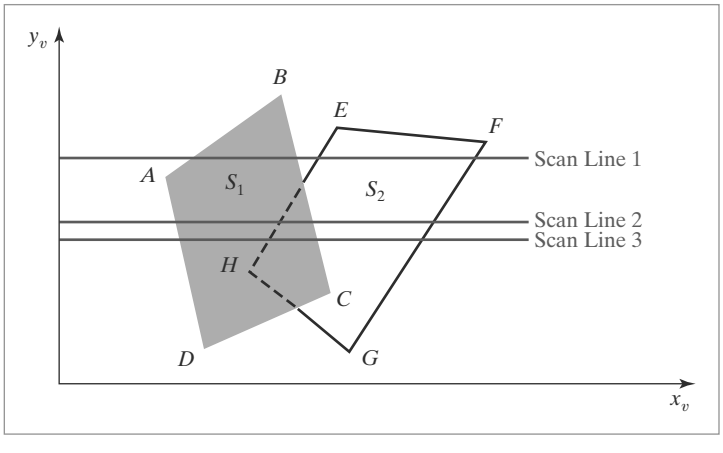


FIGURE 10
Scan lines crossing the view-plane projection of two surfaces, S_1 and S_2. Dashed lines indicate the boundaries of hidden surface sections.

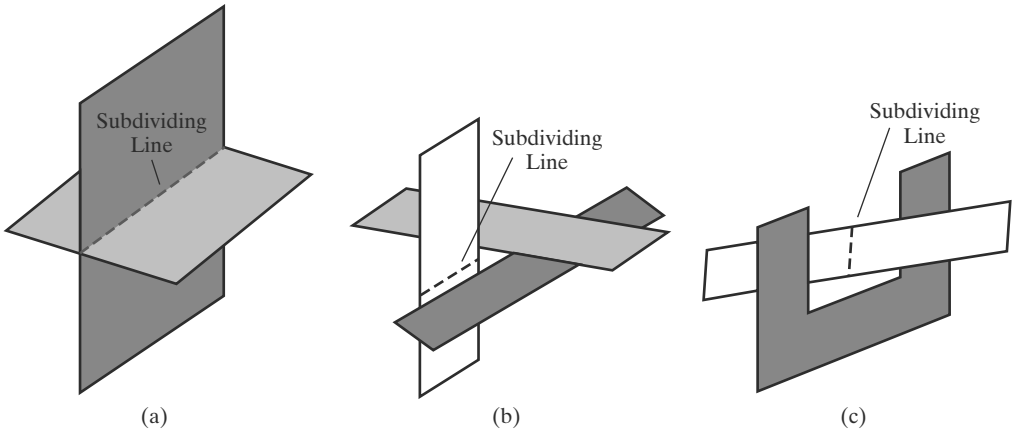


FIGURE 11
Intersecting and cyclically overlapping surfaces that alternately obscure one another.

EH and FG, only the flag for surface S_2 is on. No other positions along scan line 1 intersect surfaces, so the color for those pixels is the background color, which could be loaded into the frame buffer as part of the initialization routine.

For scan lines 2 and 3 in Figure 10, the active edge list contains edges AD, EH, BC, and FG. Along scan line 2 from edge AD to edge EH, only the flag for surface S_1 is on. But between edges EH and BC, the flags for both surfaces are on. Therefore, a depth calculation is necessary, using the plane coefficients for the two surfaces, when we encounter edge EH. For this example, the depth of surface S_1 is assumed to be less than that of S_2, so the color values for surface S_1 are assigned to the pixels across the scan line until boundary BC is encountered. Then the surface flag for S_1 goes off, and the colors for surface S_2 are stored up to edge FG. No other depth calculations are necessary, because we assume that surface S_2 remains behind S_1 once we have determined the depth relationship at edge EH.

We can take advantage of coherence along the scan lines as we pass from one scan line to the next. In Figure 10, scan line 3 has the same active list of edges as scan line 2. No changes have occurred in line intersections, so it is again unnecessary to make depth calculations between edges EH and BC. The two surfaces must be in the same orientation as determined on scan line 2, so the colors for surface S_1 can be entered without further depth calculations.

Any number of overlapping polygon surfaces can be processed with this scan-line method. Flags for the surfaces are set to indicate whether a position is inside or outside, and depth calculations are performed only at the edges of overlapping surfaces. This procedure works correctly only if surfaces do not cut through or otherwise cyclically overlap each other (Figure 11). If any kind of cyclic overlap is present in a scene, we can divide the surfaces to eliminate the overlaps. The dashed lines in this figure indicate where planes could be subdivided to form two distinct surfaces, so that the cyclic overlaps are eliminated.

6 Depth-Sorting Method

Using both image-space and object-space operations, the **depth-sorting** method performs the following basic functions:

1. Surfaces are sorted in order of decreasing depth.
2. Surfaces are scan-converted in order, starting with the surface of greatest depth.

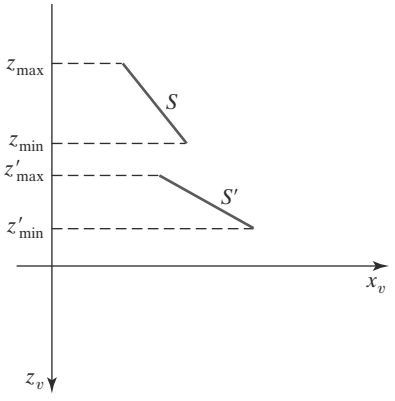


FIGURE 12
Two surfaces with no depth overlap.

Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space.

This visibility-detection method is often referred to as the **painter's algorithm.** In creating an oil painting, an artist first paints the background colors. Next, the most distant objects are added, then the nearer objects, and so forth. At the final step, the foreground is painted on the canvas over the background and the more distant objects. Each color layer covers up the previous layer. Using a similar technique, we first sort surfaces according to their distance from the view plane. The color values for the farthest surface can then be entered into the refresh buffer. Taking each succeeding surface in turn (in decreasing depth order), we "paint" the surface onto the frame buffer over the colors of the previously processed surfaces.

Painting polygon surfaces into the frame buffer according to depth is carried out in several steps. Assuming we are viewing along the z direction, surfaces are ordered on the first pass according to the smallest z value on each surface. The surface S at the end of the list (with the greatest depth) is then compared to the other surfaces in the list to determine whether there are any depth overlaps. If no depth overlaps occur, S is the most distant surface and it is scan-converted. Figure 12 shows two surfaces that overlap in the xy plane but have no depth overlap. This process is then repeated for the next surface in the list. So long as no overlaps occur, each surface is processed in depth order until all have been scan-converted. If a depth overlap is detected at any point in the list, we need to make some additional comparisons to determine whether any of the surfaces should be reordered.

We make the following tests for each surface that has a depth overlap with S. If any one of these tests is true, no reordering is necessary for S and the surface being tested. The tests are listed in order of increasing difficulty:

1. The bounding rectangles (coordinate extents) in the xy directions for the two surfaces do not overlap.
2. Surface S is completely behind the overlapping surface relative to the viewing position.
3. The overlapping surface is completely in front of S relative to the viewing position.
4. The boundary-edge projections of the two surfaces onto the view plane do not overlap.

We perform these tests in the order listed and proceed to the next overlapping surface as soon as we find that one of the tests is true. If all the overlapping surfaces

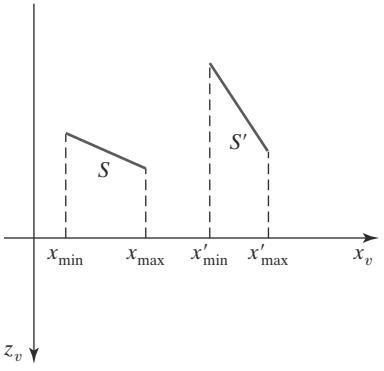


FIGURE 13
Two surfaces with depth overlap but no overlap in the x direction.

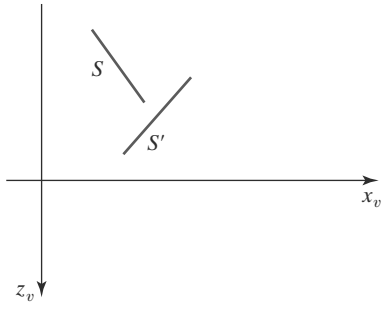


FIGURE 14
Surface S is completely behind the overlapping surface S'.

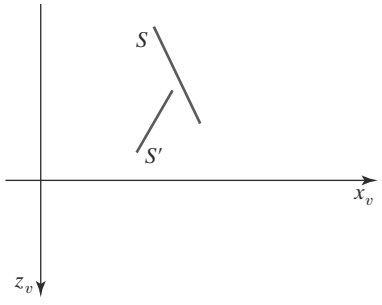


FIGURE 15
Overlapping surface S' is completely in front of surface S, but S is not completely behind S'.

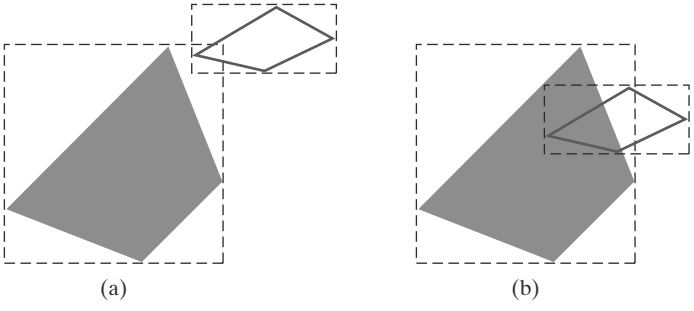


FIGURE 16
Two polygon surfaces with overlapping bounding rectangles in the xy plane.

pass at least one of these tests, then S is the most distant surface. No reordering is then necessary, therefore, and S is scan-converted.

Test 1 is performed in two parts. We check for overlap first in the x direction, then in the y direction. If there is no surface overlap in either of these directions, the two planes cannot obscure one other. An example of two surfaces that overlap in the z direction but not in the x direction is shown in Figure 13.

We can perform tests 2 and 3 using back-front polygon tests. That is, we substitute the coordinates for all vertices of S into the plane equation for the overlapping surface and check the sign of the result. If the plane equations are set up so that the front of the surface is toward the viewing position, then S is behind S' if all vertices of S are in back of S' (Figure 14). Similarly, S' is completely ahead of S if all vertices of S are in front of S'. Figure 15 shows an overlapping surface S' that is completely in front of S, but surface S is not completely behind S' (test 2 is not true).

If tests 1 through 3 have all failed, we perform test 4 to determine whether the two surface projections overlap. As demonstrated in Figure 16, two surfaces may or may not intersect even though their coordinate extents overlap.

Should all four tests fail for an overlapping surface S', we interchange surfaces S and S' in the sorted list. An example of two surfaces that would be reordered with this procedure is given in Figure 17. At this point, we still do not know for certain that we have found the farthest surface from the view plane. Figure 18 illustrates a situation in which we would first interchange S and S''. However, S'' obscures part of S', so we need to interchange S'' and S' to get the three surfaces

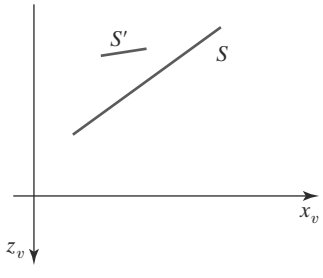


FIGURE 17
Surface S extends to a greater depth, but it obscures surface S'.

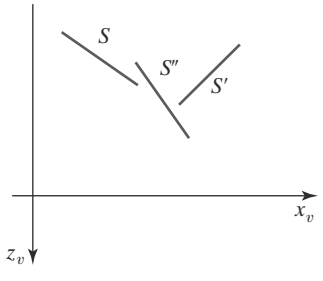


FIGURE 18
Three surfaces that have been entered into the sorted surface list in the order S, S', S'' should be reordered as S', S'', S.

into the correct depth order. Therefore, we need to repeat the testing process for each surface that is reordered in the list.

It is possible for the algorithm just outlined to get into an infinite loop if two or more surfaces alternately obscure each other, as in Figure 11. In such situations, the algorithm would continually rearrange the ordering of the overlapping surfaces. To avoid such loops, we can flag any surface that has been reordered to a farther depth position so that it cannot be moved again. If an attempt is made to switch the surface a second time, we divide it into two parts to eliminate the cyclic overlap. The original surface is then replaced by the two new surfaces, and we continue processing as before.

7 BSP-Tree Method

A **binary space-partitioning** (BSP) tree is an efficient method for determining object visibility by painting surfaces into the frame buffer from back to front, as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are behind or in front of the partitioning plane at each step of the space subdivision, relative to the viewing direction. Figure 19 illustrates the basic concept in this algorithm. With plane P_1, we first partition the space into two sets of objects. One set of objects is in back of plane P_1 relative to the viewing direction, and the other set is in front of P_1. Because one object is intersected by plane P_1, we divide that object into two separate objects, labeled A and B. Objects A and C are in front of P_1, and objects B and D are behind P_1. Because each object list contains more than one object, we partition the space again with plane P_2, recursively

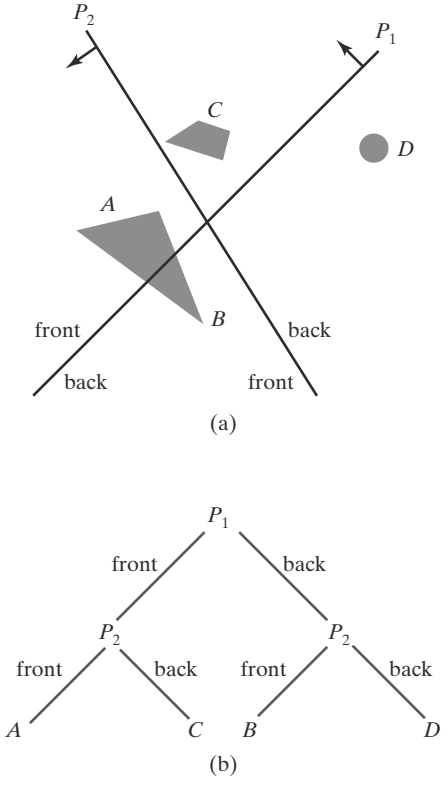


FIGURE 19
A region of space (a) is partitioned with two planes P_1 and P_2 to form the BSP tree representation shown in (b).

processing the front and back object lists. This process continues until all object lists contain no more than one object. This partitioning can be easily represented using a binary tree such as the one shown in Figure 19(b). In this tree, the objects are represented as terminal nodes, with front objects occupying the left branches and back objects occupying the right branches. The location of an object in the tree exactly represents its position relative to each of the partitioning planes.

For objects described with polygon facets, we often choose the partitioning planes to coincide with polygon-surface planes. The polygon equations are then used to identify back and front polygons, and the tree is constructed with one partitioning plane for each polygon face. Any polygon intersected by a partitioning plane is split into two parts.

When the BSP tree is complete, we interpret the tree relative to the position of our viewpoint, beginning at the root node. If the viewpoint is in front of that partitioning plane, we recursively process the back subtree, then recursively process the front subtree. If the viewpoint is behind the partitioning plane, we reverse this, and process the front subtree followed by the back subtree. Thus, the surfaces are generated for display in the order back to front, so that foreground objects are painted over the background objects. Fast hardware implementations for constructing and processing BSP trees are used in some systems.

8 Area-Subdivision Method

This technique for hidden-surface removal is essentially an image-space method, but object-space operations can be used to accomplish depth ordering of surfaces. The **area-subdivision method** takes advantage of area coherence in a scene by locating those projection areas that represent part of a single surface. We apply this method by successively dividing the total view-plane area into smaller and smaller rectangles until each rectangular area contains the projection of part of a single visible surface, contains no surface projections, or the area has been reduced to the size of a pixel.

To implement this method, we need to establish tests that can quickly identify the area as part of a single surface or tell us that the area is too complex to analyze easily. Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles. If the tests indicate that the view is sufficiently complex, we subdivide it. Next, we apply the tests to each of the smaller areas, subdividing these if the tests indicate that visibility of a single surface is still uncertain. We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until we have reached the resolution limit. An easy way to do this is to successively divide the area into four equal parts at each step, as shown in Figure 20. This approach is similar to that used in constructing a quadtree. A viewing area with a pixel resolution of 1024×1024 could be subdivided ten times in this way before a subarea is reduced to the size of a single pixel.

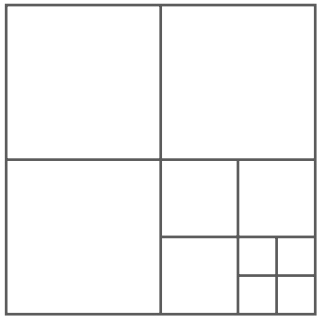


FIGURE 20
Dividing a square area into equal-sized quadrants at each step.

There are four possible relationships that a surface can have with an area of the subdivided view plane. We can describe these relative surface positions using the following classifications (Figure 21).

Surrounding Surface: A surface that completely encloses the area.

Overlapping Surface: A surface that is partly inside and partly outside the area.

Inside Surface: A surface that is completely inside the area.

Outside Surface: A surface that is completely outside the area.

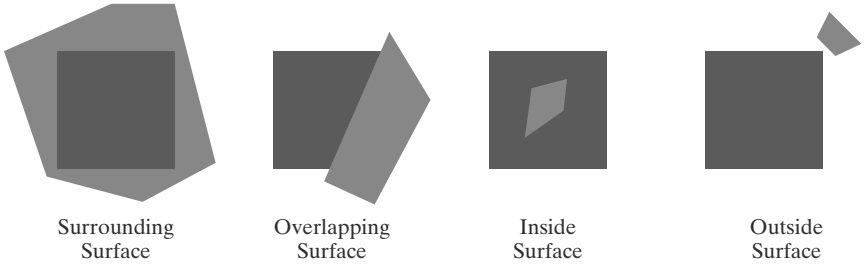


FIGURE 21
Possible relationships between polygon surfaces and a rectangular section of the viewing plane.

The tests for determining surface visibility within a rectangular area can be stated in terms of the four surface classifications illustrated in Figure 21. No further subdivisions of a specified area are needed if one of the following conditions is true.

Condition 1: An area has no inside, overlapping, or surrounding surfaces (all surfaces are outside the area).

Condition 2: An area has only one inside, overlapping, or surrounding surface.

Condition 3: An area has one surrounding surface that obscures all other surfaces within the area boundaries.

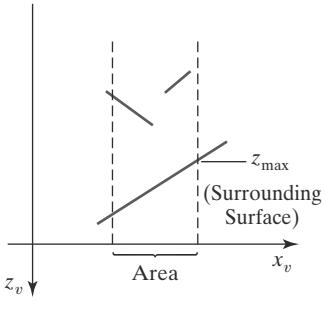


FIGURE 22
Within a specified area, a surrounding surface with a maximum depth of z_{max} obscures all surfaces that have a minimum depth beyond z_{max}.

Initially, we can compare the coordinate extents of each surface with the area boundaries. This will identify the inside and surrounding surfaces, but overlapping and outside surfaces usually require intersection tests. If a single bounding rectangle intersects the area in some way, additional checks are used to determine whether the surface is surrounding, overlapping, or outside. Once a single inside, overlapping, or surrounding surface has been identified, the surface color values are stored in the frame buffer.

One method for testing condition 3 is to sort the surfaces according to minimum depth from the view plane. For each surrounding surface, we then compute the maximum depth within the area under consideration. If the maximum depth of one of these surrounding surfaces is closer to the view plane than the minimum depth of all other surfaces within the area, condition 3 is satisfied. Figure 22 illustrates this situation.

Another method for testing condition 3 that does not require depth sorting is to use plane equations to calculate depth values at the four vertices of the area for all surrounding, overlapping, and inside surfaces. If all four depths for one of the surrounding surfaces are less than the calculated depths for all other surfaces, condition 3 is satisfied. Then the area can be displayed with the colors for that surrounding surface.

For some situations, the previous two testing methods may fail to identify correctly a surrounding surface that obscures all the other surfaces. Further testing could be carried out to identify the single surface that covers the area, but it is faster to subdivide the area than to continue with more complex testing. Once a surface has been identified as an outside or surrounding surface for an area, it will remain in that category for all subdivisions of the area. Furthermore, we can expect to eliminate some inside and overlapping surfaces as the subdivision process continues, so that the areas become easier to analyze. In the limiting case, when a subdivision the size of a pixel is produced, we simply calculate the depth of each relevant surface at that point and assign the color of the nearest surface to that pixel.

As a variation on the basic subdivision process, we could subdivide areas along surface boundaries instead of dividing them in half. If the surfaces have

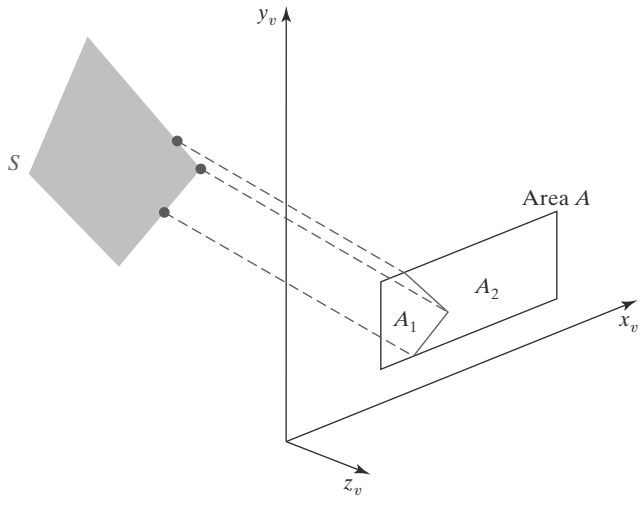


FIGURE 23
Area A is subdivided into A_1 and A_2 using the boundary of surface S on the view plane.

been sorted according to minimum depth, we can use the surface of smallest depth value to subdivide a given area. Figure 23 illustrates this method for subdividing areas. The projection of the boundary of surface S is used to partition the original area into the subdivisions A_1 and A_2. Surface S is then a surrounding surface for A_1, and visibility conditions 2 and 3 can be tested to determine whether further subdividing is necessary. In general, fewer subdivisions are required using this approach, but more processing is needed to subdivide areas and to analyze the relation of surfaces to the subdivision boundaries.

9 Octree Methods

When an octree representation is used for the viewing volume, visible-surface identification is accomplished by searching octree nodes in a front-to-back order. In Figure 24, the foreground of a scene is contained in octants 0, 1, 2, and 3. Surfaces in the front of these octants are visible to the viewer. Any surfaces toward the rear of the front octants or in the back octants (4, 5, 6, and 7) may be hidden by the front surfaces.

We can process the octree nodes of Figure 24 in the order 0, 1, 2, 3, 4, 5, 6, 7. This results in a depth-first traversal of the octree, where the nodes for the four front suboctants of octant 0 are visited before the nodes for the four back

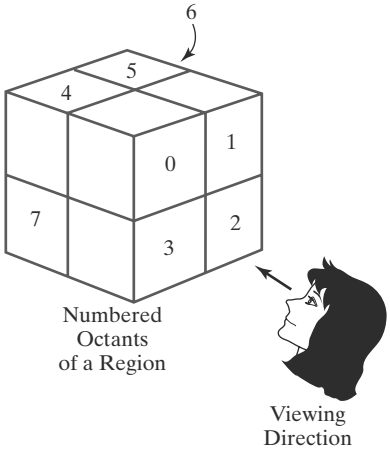
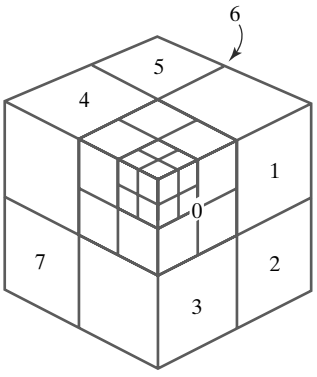
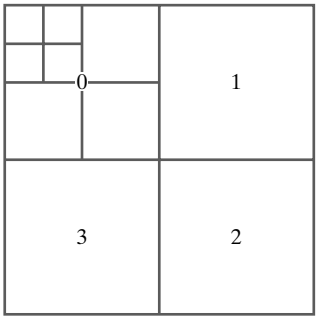


FIGURE 24
Objects in octants 0, 1, 2, and 3 obscure objects in the back octants (4, 5, 6, 7) when the viewing direction is as shown.



Octants in Space



Quadrants for the View Plane

FIGURE 25
Octant divisions for a region of space and the corresponding quadrant plane.

suboctants. The traversal of the octree continues in this order for each octant subdivision.

When a color value is encountered in an octree node, that color is saved in the quadtree only if no values have previously been saved for the same area. In this way, only the front colors are saved. Nodes that have the value "void" are ignored. Any node that is completely obscured is eliminated from further processing, so that its subtrees are not accessed. Figure 25 depicts the octants in a region of space and the corresponding quadrants on the view plane. Contributions to quadrant 0 come from octants 0 and 4. Color values in quadrant 1 are obtained from surfaces in octants 1 and 5, and values in each of the other two quadrants are generated from the pairs of octants aligned with each of these quadrants.

Effective octree visibility testing is carried out with recursive processing of octree nodes and the creation of a quadtree representation for the visible surfaces. In most cases, both a front and a back octant must be considered in determining the correct color values for a quadrant. But if the front octant is homogeneously filled with some color, we do not process the back octant. For heterogeneous regions, a recursive procedure is called, passing as new arguments the child of the heterogeneous octant and a newly created quadtree node. If the front is empty, it is necessary only to process the child of the rear octant. Otherwise, two recursive calls are made: one for the rear octant and one for the front octant.

Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according to the view selected. Octants can then be renumbered so that the octree representation is always organized with octants 0, 1, 2, and 3 as the front face.

10 Ray-Casting Method

If we consider the line of sight from a pixel position on the view plane through a scene, as in Figure 26, we can determine which objects in the scene (if any) intersect this line. After calculating all ray-surface intersections, we identify the visible surface as the one whose intersection point is closest to the pixel. This visibility-detection scheme uses *ray casting* procedures. Ray casting, as a visibility-detection tool, is based on geometricoptics methods, which trace the paths of light rays. Because there are an infinite number of light rays in a scene and we are interested only in those rays that pass through pixel positions, we can trace

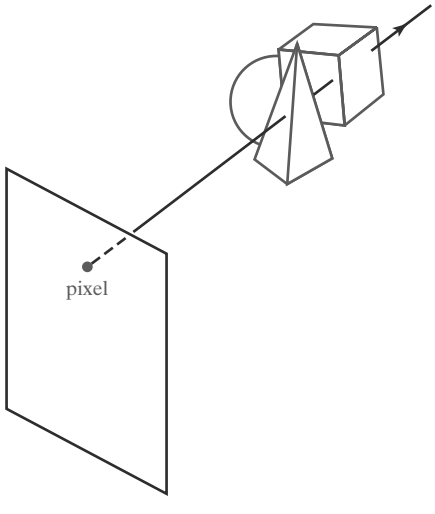


FIGURE 26
A ray along the line of sight from a pixel position through a scene.

the light-ray paths backward from the pixels through the scene. The ray-casting approach is an effective visibility-detection method for scenes with curved surfaces, particularly spheres.

We can think of ray casting as a variation on the depth-buffer method (Section 3). In the depth-buffer algorithm, we process surfaces one at a time and calculate depth values for all projection points over the surface. The calculated surface depths are then compared to previously stored depths to determine visible surfaces at each pixel. In ray casting, we process pixels one at a time and calculate depths for all surfaces along the projection path to that pixel.

Ray casting is a special case of *ray-tracing* algorithms that trace multiple ray paths to pick up global reflection and refraction contributions from multiple objects in a scene. With ray casting, we only follow a ray out from each pixel to the nearest object. Efficient ray-surface intersection calculations have been developed for common objects, particularly spheres.

11 Comparison of Visibility-Detection Methods

The effectiveness of a visible-surface detection method depends on the characteristics of a particular application. If the surfaces in a scene are widely distributed along the viewing direction so that there is very little depth overlap, a depth-sorting or BSP-tree method is often most efficient. When there are few overlaps of the surface projections on the view plane, a scan-line or area-subdivision approach is a fast way to locate visible surfaces.

As a general rule, either the depth-sorting algorithm or the BSP-tree method is a highly effective approach for scenes with only a few surfaces. This is because these scenes usually have few surfaces that overlap in depth. The scan-line method also performs well when a scene contains a small number of surfaces. We can use the scan-line, depth-sorting, or BSP-tree method to identify visible surfaces effectively for scenes with up to several thousand polygon surfaces. With scenes that contain more than a few thousand surfaces, the depth-buffer method or octree approach performs best. The depth-buffer method has a nearly constant processing time, independent of the number of surfaces in a scene. This is because the size of the surface areas decreases as the number of surfaces in the scene increases. Therefore, the depth-buffer method exhibits relatively low performance with simple scenes and relatively high performance with complex scenes. BSP trees are useful when multiple views are to be generated using different view reference points. If a scene contains curved-surface representations, we can use octree or ray-casting methods to identify visible parts of the scene.

When octree representations are used in a system, the visibility-detection process is fast and simple. Only integer additions and subtractions are used in the process, and there is no need to perform sorting or intersection calculations. Another advantage of octrees is that they store more than just the surface geometry. The entire solid region of an object is available for display, which makes the octree representation useful for obtaining cross-sectional slices of three-dimensional objects.

It is possible to combine and implement the different visible-surface detection methods in various ways. In addition, visibility-detection algorithms are often implemented in hardware, and special systems utilizing parallel processing are employed to increase the efficiency of these methods. Special hardware systems are used when processing speed is an especially important consideration, as in the generation of animated views for flight simulators.

12 Curved Surfaces

Effective methods for determining the visibility of objects with curved surfaces include ray casting and octree methods. With ray casting, we calculate ray-surface intersections and locate the smallest intersection distance along the pixel ray. With octrees, we simply search the nodes from front to back to locate the surface color values. Once an octree representation has been established from the input definition of the objects, all visible surfaces are identified with the same processing procedures. No special considerations need be given to different kinds of surfaces, curved or otherwise.

A curved surface can also be approximated as a polygon mesh, and we can then use one of the visible-surface identification methods previously discussed. But for some objects, such as spheres, it could be more efficient as well as more accurate to use ray casting and the equations describing the curved surface.

Curved-Surface Representations

We can represent a surface with an implicit equation of the form $f(x, y, z) = 0$ or with a parametric representation. Spline surfaces, for example, are normally described with parametric equations. In some cases, it is useful to obtain an explicit surface equation, such as with a height function over an xy ground plane:

$$z = f(x, y)$$

Many objects of interest, such as spheres, ellipsoids, cylinders, and cones, have quadratic representations. These surfaces are commonly used to model molecular structures, roller bearings, rings, and shafts.

Scan-line and ray-casting algorithms often involve numerical approximation techniques to solve the surface equation at the intersection point with a scan line or with a pixel ray. Various techniques, including parallel calculations and fast hardware implementations, have been developed for solving the curved-surface intersection equations for commonly used objects.

Surface Contour Plots

For many applications in mathematics, physical sciences, engineering, and other fields, it is useful to display a surface function with a set of contour lines that show the surface shape. The surface may be described with an equation or with data tables, such as topographic data on elevations or population density. With an explicit functional representation, we can plot the visible surface contour lines and eliminate those contour sections that are hidden by the visible parts of the surface.

To obtain an xy plot of a functional surface, we can write the surface representation in the form

$$y = f(x, z) \tag{8}$$

A curve in the xy plane can then be plotted for values of z within some selected range, using a specified interval Δz. Starting with the largest value of z, we plot the curves from "front" to "back" and eliminate hidden sections. We draw the curve sections on the screen by mapping an xy range for the function into an xy pixel screen range. Then, unit steps are taken in x and the corresponding y value for each x value is determined from Eq. 8 for a given value of z.

One way to identify the visible curve sections on the surface is to maintain a list of y_{min} and y_{max} values previously calculated for the pixel x coordinates on the screen. As we step from one pixel x position to the next, we check the calculated

y value against the stored range, y_{min} and y_{max}, for the next pixel. If $y_{min} \leq y \leq y_{max}$, that point on the surface is not visible and we do not plot it. But if the calculated y value is outside the stored y bounds for that pixel, the point is visible. We then plot the point and reset the bounds for that pixel. Similar procedures can be used to project the contour plot onto the xz or yz plane.

We can apply the same methods to a discrete set of data points by determining isosurface lines. For example, if we have a discrete set of z values for an $n_x \times n_y$ grid of xy values, we can determine the path for a line of constant z over the surface using contour plotting methods. Each selected contour line can then be projected onto a view plane and displayed with straight-line segments. Again, lines can be drawn on the display device in a front-to-back depth order, and we eliminate contour sections that pass behind previously drawn (visible) contour lines.

13 Wire-Frame Visibility Methods

Scenes usually do not contain isolated line sections, unless we are displaying a graph, diagram, or network layout. But often we want to view a three-dimensional scene in an outline form to obtain a quick display of the object features. The fastest way to generate a wire-frame view of a scene is to display all object edges. However, it may be difficult to determine the front and back features of the objects in such a display. One solution to this problem is to apply depth cueing, so that the displayed intensity of a line is a function of its distance from the viewer. Alternatively, we can apply visibility tests, so that hidden line sections can be either eliminated or displayed differently from the visible edges. Procedures for determining visibility of object edges are referred to as **wire-frame visibility methods.** They are also called **visible-line detection methods** or **hidden-line detection methods.** In addition, some of the visible-surface methods discussed in preceding sections can be used to test for edge visibility.

Wire-Frame Surface-Visibility Algorithms

A direct approach to identifying visible line sections is to compare edge positions with the positions of the surfaces in a scene. This process involves the same methods used in line-clipping algorithms. That is, we test the position of line endpoints with respect to the boundaries of a specified area, but, for visibility testing, we also need to compare edge and surface depth values. When the projected edge endpoints of a line segment are both within the projected area of a surface, we compare the depth of the endpoints to the surface depth at those (x, y) positions. If both endpoints are behind the surface, we have a hidden edge. If both endpoints are in front of the surface, the edge is visible with respect to that surface. Otherwise, we must calculate intersection positions and determine the depth values at those intersection points. If the edge has greater depth than the surface at the perimeter intersections, part of the edge is hidden by the surface, as in Figure 27(a). Another possibility is that an edge has greater depth at one boundary intersection and less depth than the surface at the other boundary intersection (assuming surfaces are convex). In that case, we need to determine where the edge penetrates the surface interior, as in Figure 27(b). Once we have identified a hidden section of an edge, we could eliminate it, display it as a dashed line, or display it in some other way to distinguish it from the visible sections.

Some of the visible-surface detection methods are readily adapted to wire-frame visibility testing of object edges. Using a back-face method, we could

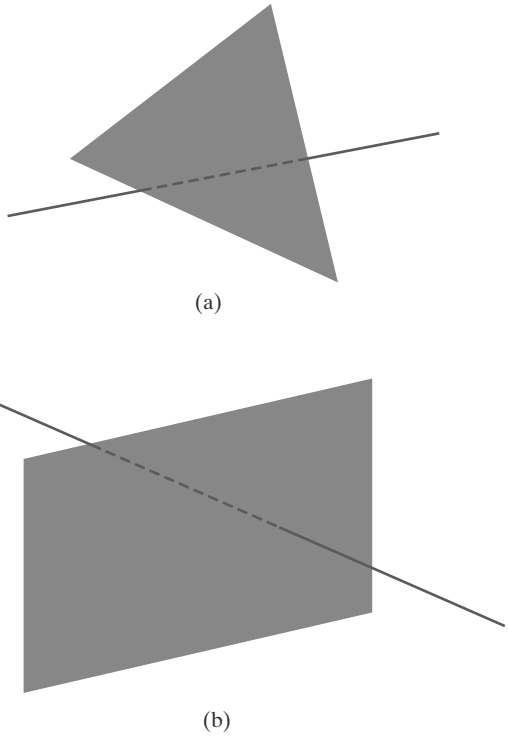


FIGURE 27
Hidden-line sections (dashed) for a line (a) that has greater depth than a surface and a line (b) that is partially behind a surface and partially in front of the surface.

identify all the back surfaces of an object and display only the boundaries for the visible surfaces. With depth sorting, surfaces can be painted into the refresh buffer so that surface interiors are in the background color while boundaries are in the foreground color. By processing the surfaces from back to front, hidden lines are erased by the nearer surfaces. An area-subdivision method can be adapted to hidden-line removal by displaying only the boundaries of visible surfaces. And scan-line methods can be used to display the scan-line intersection positions at the boundaries of visible surfaces.

Wire-Frame Depth-Cueing Algorithm

Another method for displaying visibility information is to vary the brightness of objects in a scene as a function of distance from the viewing position. This **depth-cueing method** is typically applied using the linear function

$$f_{\text{depth}}(d) = \frac{d_{\max} - d}{d_{\max} - d_{\min}} \tag{9}$$

where d is the distance of a point from the viewing position. Values for minimum and maximum depth, $d_{\min}$ and $d_{\max}$, can be set to convenient values for a particular application, or the minimum and maximum depths can be set to the normalization depth range: $d_{\min} = 0.0$ and $d_{\max} = 1.0$. As each pixel position is processed, its color is multiplied by $f_{\text{depth}}(d)$. Thus, nearer points are displayed with higher intensities, and the points at the maximum depth have an intensity equal to 0.

The depth-cueing function can be implemented with various options. In some graphics libraries, a general atmosphere function is available, which can combine depth cueing with atmospheric effects to simulate smoke or haze, for example. Thus, an object s color could be modified by the depth-cueing function and then combined with the atmosphere color.

14 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`. In fact, we could use this function to remove the front faces instead, or we could even remove both front and back faces. If our viewing position is inside a building, for example, then we want to see only the back faces (the inside of the rooms). In this case, we could either set parameter `mode` to `GL_FRONT`, or we could change the definition of front-facing polygons using the `glFrontFace` function. Then, if the viewing position moves outside the building, we can cull the back faces from the display; and in some applications, we might want to view only other primitives in a scene, such as point sets and individual straight-line segments. So, to eliminate all surfaces in a scene, we set parameter `mode` to the OpenGL symbolic constant `GL_FRONT_AND_BACK`.

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 depth-offset method. That is, we first specify the wireframe version of the object using the foreground color, then we specify an interior fill version using a depth offset and the background color for the interior fill. The depth offset ensures that the background-color fill will not interfere with the display of the visible edges. As an example, the following code segment generates a wire-frame display of an object using a white foreground color and a black background color:

```
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 in Eq. 9 to object colors using $d_{\text{min}} = 0.0$ and $d_{\text{max}} = 1.0$. But we can set different values for d_{min} and d_{max} 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.

15 Summary

The simplest visibility test is the back-face detection algorithm, which is fast and effective as an initial screening to eliminate many polygons from further visibility tests. For a single convex polyhedron, back-face detection eliminates all hidden surfaces but, in general, back-face detection cannot completely identify all hidden surfaces.

A commonly used method for identifying all visible surfaces in a scene is the depth-buffer algorithm. When applied to standard graphics objects, this procedure is highly efficient, but it does have extra storage requirements. Two buffers are needed: one to store pixel colors and one to store the depth values for the pixel positions. Fast, incremental, scan-line methods are used to process each polygon in a scene to calculate surface depths. As each surface is processed, the two buffers are updated. An extension of the depth-buffer approach is the A-buffer, which provides additional information for displaying antialiased and transparent surfaces.

Several other visibility-detection methods have been devised. The scan-line method processes all surfaces at once for each scan line. With the depth-sorting method (painter's algorithm), objects are "painted" into the refresh buffer according to their distances from the viewing position. Subdivision schemes for identifying visible parts of a scene include the BSP-tree method, area subdivision, and octree representations. Visible surfaces can also be detected using ray-casting methods, which project lines from the pixel plane into a scene to determine object intersection positions along these projected lines. Ray-casting methods are an integral part of ray-tracing algorithms, which allow scenes to be displayed with global-illumination effects.

Visibility-detection methods are also used in displaying three-dimensional line drawings. With curved surfaces, we can display contour plots. For wire-frame displays of polyhedrons, we search for the various edge sections of the surfaces in a scene that are visible from the viewing position.

We can implement any visibility-detection scheme in an application program by creating our own routines, but graphics libraries commonly provide functions only for back-face removal and the depth-buffer method. In high-end computer-graphics systems, the depth-buffer routines are hardware-implemented.

Functions for polygon culling and for depth-buffer visibility determinations are available in the OpenGL core library. With the polygon-culling routines, we can remove the back faces of standard graphics objects, their front faces, or both. With the depth-buffer routines, we can set the range for the depth tests and the type of depth testing that is to be performed. Wire-frame displays are obtained using the OpenGL polygon-mode and polygon-offset operations. And we can also generate OpenGL scenes using depth-cueing effects. In Table 1, we summarize the OpenGL functions for visibility testing.

TABLE 1

Summary of OpenGL Visibility-Detection Functions

Function	Description
`glCullFace`	Specifies front or back planes of polygons for culling operations when activated with `glEnable (GL_CULL_FACE)`.
`glutInitDisplayMode`	Specifies depth-buffer operations using argument `GLUT_DEPTH`.
`glClear (GL_DEPTH_BUFFER_BIT)`	Initializes depth-buffer values to the default (1.0) or a value specified by the `glClearDepth` function.
`glClearDepth`	Specifies an initial depth-buffer value.
`glEnable (GL_DEPTH_TEST)`	Activates depth-testing operations.
`glDepthRange`	Specifies a range for normalizing depth values.
`glDepthFunc`	Specifies a depth-testing condition.
`glDepthMask`	Sets write status for the depth buffer.
`glPolygonOffset`	Specifies an offset to eliminate hidden lines in a wire-frame display when a background fill color is applied.
`glFog`	Specifies linear depth-cueing operations and values for minimum and maximum depth in the depth-cueing calculations.

REFERENCES

Additional sources of information on visibility algorithms include Elber and Cohen (1990), Franklin and Kankanhalli (1990), Segal (1990), and Naylor, Amanatides, and Thibault (1990). A-buffer methods are presented in Cook, Carpenter, and Catmull (1987), Haeberli and Akeley (1990), and Shilling and Strasser (1993). A summary of contouring methods is given in Earnshaw (1985).

Various programming techniques for visibility testing can be found in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). Woo, et al. (1999) provide additional discussions of the OpenGL visibility-detection functions. A complete listing of the OpenGL functions in the core library and in GLU is presented in Shreiner (2000).

EXERCISES

1 Set up a back-face detection procedure that will identify all the visible faces of any input convex polyhedron that has different-colored surfaces. The polyhedron is to be defined in a right-handed viewing system, and the viewing direction is specified as user input.

2 Implement the procedure in the preceding exercise using an orthographic parallel projection to view visible faces of the input convex polyhedron. Assume that all parts of the object are in front of the view plane.

3 Implement the procedure in Exercise 1 using a perspective projection to view visible faces of the input convex polyhedron. Assume that all parts of the object are in front of the view plane.

4 Write a program to produce an animation of a convex polyhedron. The object is to be rotated incrementally about an axis that passes through the object and is parallel to the view plane. Assume that the object lies completely in front of the view plane. Use an orthographic parallel projection to map the views successively onto the view plane.

5 Modify the program in the preceding exercise to allow the user to switch between an orthographic parallel projection and a perspective projection using keyboard input.

6 Write a routine to implement the depth-buffer method for the display of the visible surfaces of any input polyhedron. The array for the depth-buffer can be set to any convenient size on your system, such as 500×500. How can the storage requirements for the depth buffer be determined from the definition of the objects to be displayed?

7 Modify the procedure in the preceding exercise to display the visible surfaces in a scene containing any number of polyhedrons. Set up efficient methods for storing and processing the various objects in the scene.

8 Write a program using the procedure in the previous exercise that takes as input a set of polyhedrons contained within a (conceptual) sphere of a given radius centered at the origin. Each time a certain key is pressed, the program should generate a new random camera position outside of the sphere and a random look-at point somewhere inside the sphere. The view up vector should always be the positive y unit vector. The program should then display the visible surfaces of the objects in the scene from that viewpoint.

9 Modify the procedure of the preceding exercise to implement the A-buffer algorithm for the display of a scene containing both opaque and transparent surfaces.

10 Extend the procedure developed in the preceding exercise to include antialiasing.

11 Write a program using the procedure in the previous exercise that takes as input a set of polyhedrons contained within a (conceptual) sphere of a given radius centered at the origin, each of which have randomized transparency values. Each time a key is pressed, the program should generate a new random camera position outside of the sphere and a random look-at point somewhere inside the sphere. The view up vector should always be the positive y unit vector. The program should then display the visible surfaces of the objects in the scene from that viewpoint.

12 Develop a program to implement the scan-line algorithm for displaying the visible surfaces of a given polyhedron. Use polygon tables to store the definition of the object, and use coherence techniques to evaluate points along and between scan lines.

13 Write a program to implement the scan-line algorithm for a scene containing several polyhedrons. Use polygon tables to store the definition of the object, and use coherence techniques to evaluate points along and between scan lines.

14 Set up a program to display the visible surfaces of a convex polyhedron using the painter's algorithm. That is, surfaces are to be sorted on depth and painted on the screen from back to front.

15 Write a program that uses the depth-sorting method to display the visible surfaces of any given object with plane faces.

16 Develop a depth-sorting program to display the visible surfaces in a scene containing several polyhedrons.

17 Write a program to display the visible surfaces of a convex polyhedron using the BSP-tree method.

18 Give examples of situations where the two methods discussed for condition 3 in the area-subdivision algorithm will fail to identify correctly a surrounding surface that obscures all other surfaces.

19 Develop an algorithm that would test a given plane surface against a rectangular area to decide whether it is a surrounding, overlapping, inside, or outside surface.

20 Develop an algorithm for generating a quadtree representation for the visible surfaces of an object by applying the area-subdivision tests to determine the values of the quadtree elements.

21 Set up an algorithm to store a quadtree representation of an object in a frame buffer.

22 Set up a procedure to display the visible surfaces of an object that is described with an octree representation.

23 Use the procedure developed in the previous exercise to write a program that displays the visible surfaces of a set of objects represented as octree structures. The viewing parameters should be taken in as input.

24 Devise an algorithm for viewing a single sphere using the ray-casting method.

25 Discuss how antialiasing methods can be incorporated into the various hidden-surface elimination algorithms.

26 Write a routine to produce a surface contour plot for a given surface function $f(x, y)$.

27 Develop an algorithm for detecting visible line sections in a scene by comparing each line in the scene to each polygon surface facet.

28 Discuss how wire-frame displays might be generated with the various visible-surface detection methods discussed in this chapter.

29 Set up a procedure for generating a wire-frame display of a polyhedron with the hidden edges of the object shown as dashed lines.

30 Write a program using the procedure developed in the previous exercise that takes a set of polyhedrons contained within a (conceptual) sphere of a given radius centered at the origin as input and displays them as wireframe objects with the hidden edges of each object shown as dashed lines. Each time a key is pressed, the program should generate a new random camera position outside of the sphere and a random look-at point somewhere inside the sphere. The view up vector should always be the positive y unit vector.

31 Write a program to display a polyhedron with selected faces removed, using the OpenGL polygon-culling functions. Each face of the polygon is to be given a different color, and a face is to be selected for removal with user input. Also, a viewing position and other viewing parameters are to be specified as input values.

32 Modify the program in the preceding exercise to view the polyhedron from any position, using the depth-buffer routines instead of the polygon-culling routines.

33 Modify the program in the preceding exercise so that the depth range and the depth test condition can also be specified as user input.

34 Generate a wire-frame display of a polyhedron using the `glPolygonMode` and `glPolygonOffset` functions as discussed in Section 14.

35 Modify the program of the preceding exercise to display the polyhedron using the depth-cueing function `glFogi`.

36 Modify the program of the preceding exercise to display several polyhedrons that are distributed in depth. The depth-cueing range is to be set with user input.

37 Modify the program in the previous exercise to allow the change the camera position by moving it around the surface of a sphere whose radius is defined as the distance from the camera position to the look-at point, which is assumed to be a point within the coordinate extents of the set of objects in the scene. The distance from the camera to the look-at point is assumed to be large enough to make all objects lie in front of the view plane for any camera position on the sphere.

IN MORE DEPTH

1 Choose a visible surface algorithm in this chapter based on the properties of your application and the strengths and weaknesses of each of the algorithms in terms of their computational complexity. Implement the algorithm and use it to render the visible surfaces of the objects in your scene.

2 Compare the rendering times for your scene with and without visible surface detection using the algorithm that you developed in the previous exercise. Then, do the same using the built-in back-face culling routines provided in OpenGL. Is there any improvement over the built-in routines that you obtain by tailoring the detection algorithm to your specific application? Discuss any further improvements you could implement in the algorithm or modifications that you could make to the object representations to increase rendering performance.

This page intentionally left blank