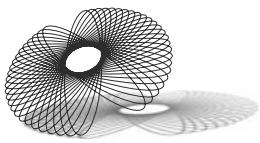


CHAPTER 6



VIEWING AND PROJECTION

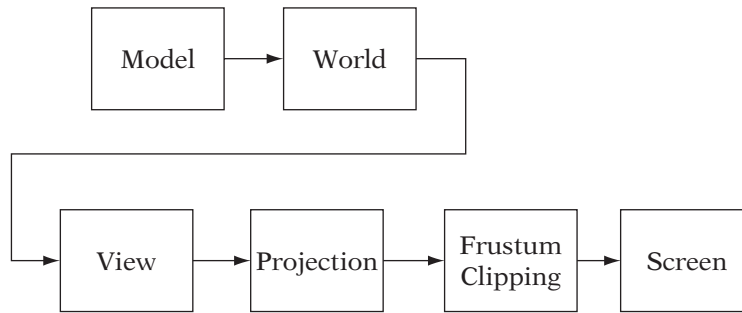
6.1 INTRODUCTION

In previous chapters we've discussed how to represent objects, basic transformations we can apply to these objects, and how we can use these transformations to move and manipulate our objects within our virtual world. With that background in place, we can begin to discuss the mathematics underlying the techniques we use to display our game objects on a monitor or other visual display medium.

It doesn't take much justification to understand why we might want to view the game world — after all, games are primarily a visual media. Other sensory outputs are of course possible, particularly sound and haptic (or touch) feedback. Both have become more sophisticated and in their own way provide another representation of the relative three-dimensional (3D) position and orientation of game objects. But in the current market, when we think of games, we first think of what we can see.

To achieve this, we'll be using a continuation of our transformation process known as the *graphics pipeline*. Figure 6.1 shows the situation. We already have a transformation that takes our model from its local space to world space. At each stage of the graphics pipeline, we continue to concatenate matrices to this matrix. Our goal is to build a single matrix to transform the points in our object from their local configuration to a two-dimensional (2D) representation suitable for displaying.

The first part of the display process involves setting up a virtual viewer or camera, which allows us to control which objects lie in our current view. As we'll see, this camera is just like any other object in the game; we can

**FIGURE 6.1** The graphics pipeline.

set the camera's position and orientation based on an affine transformation. Inverting this transformation is the first stage of our pipeline: It allows us to transform objects in the world frame into the point of view of the camera object.

From there we will want to build and concatenate a matrix that transforms our objects in view into coordinates so they can be represented in an image. This flattening or *projection* takes many forms, and we'll discuss several of the most commonly used projections. In particular, we'll derive perspective projection, which most closely mimics our viewpoint of the real world.

At this point, it is usually convenient to cull out any objects that will not be visible on our screen, and possibly cut, or clip, others that intersect the screen boundaries. This will make our final rendering process much faster.

The final stage is to transform our projected coordinates and stretch and translate them to fit a specific portion of the screen, known as the viewport. This is known as the screen transformation.

In addition, we'll cover how to reverse this process so we can take a mouse click on our 2D screen and use it to select objects in our 3D world. This process, known as *picking*, can be useful when building an interface with 3D elements. For example, selecting units in a 3D real-time strategy game is done via picking.

As with other chapters, we'll be discussing how to implement these transformations in production code. Because our primary platform is OpenGL, for the most part we'll be focusing on its pipeline and how it handles the viewing and projective transformations. However, we will also cover the cases where it may differ from graphics APIs, particularly Direct3D.

One final note before we begin: There is no standard representation for this process. In other books you may find these stages broken up in different ways, depending on the rendering system the authors are trying to present. However, the ultimate goal is the same: Take an object in the world and transform it from a viewer's perspective onto a 2D medium.

6.2 VIEW FRAME AND VIEW TRANSFORMATION

6.2.1 DEFINING A VIRTUAL CAMERA

In order to render objects in the world, we need to represent the notion of a viewer. This could be the main character's viewpoint in a first-person shooter, or an over-the-shoulder view in a third-person adventure game, or a zoomed-out wide shot in a strategy game. We may want to control properties of our viewer to simulate a virtual camera, for example, we may want to create an in-game scripted sequence where we pan across a screen or follow a set path through a space. We encapsulate these properties into a single entity, commonly called the *camera*.

For now, we'll consider only the most basic properties of the camera needed for rendering. We are trying to answer two questions [8]: Where am I? Where am I looking? We can think of this as someone taking an actual camera, placing it on a tripod, and aiming it at an object of interest.

The answer to the first question is the camera's position, E , which is variously called the *eyepoint*, the *view position*, or the *view space origin*. As we mentioned, this could be the main character's eye position, a location over his shoulder, or a spot pulled back from the action. While this position can be placed relative to another object's location, it is usually cleaner and easier to manage if we represent it in the world frame.

A partial answer to the second question is a vector called the *view direction vector*, or \mathbf{v}_{dir} , which points along the facing direction for the camera. This could be a vector from the camera position to an object or point of interest, a vector indicating the direction the main character is facing, or a fixed direction if we're trying to simulate a top-down view for a strategy game. For the purposes of setting up the camera, this is also specified in the world frame.

Having a single view direction vector is not enough to specify our orientation, since there are an infinite number of rotations around that vector. To constrain our possibilities down to one, we specify a second vector orthogonal to the first, called the *view up vector*, or \mathbf{v}_{up} . This indicates the direction out of the top of the camera. From these two we can take the cross product to get the *view side vector*, or \mathbf{v}_{side} , which usually points out toward the camera's right. Normalizing these three vectors and adding the view position gives us an orthonormal basis and an origin, or an affine frame. This is the camera's local frame, also known as the *view frame*, (Figure 6.2).

The three view vectors specify where the view orientation is relative to the world frame. However, we also need to define where these vectors are from the perspective of the camera. The standard order used by most viewing systems is to make the camera's y -axis represent the view up vector in the camera's local space, and the camera's x -axis represent the corresponding view side vector. This aligns our camera's local coordinates so that x values vary left

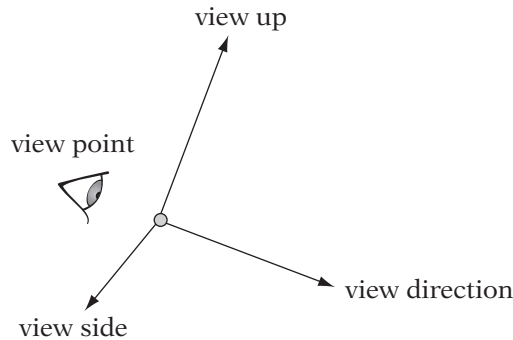


FIGURE 6.2 View frame relative to the world frame.

and right along the plane of the screen and y values vary up and down, which is very intuitive.

The remaining question is what to do with z and the view direction. In most systems, the z -axis is treated as the camera-relative view direction vector (Figure 6.3(a)). This has a nice intuitive feel: As objects in front of the viewer move farther away, their z values relative to the camera will increase. The value of z can act as a measure of the distance between the object and the camera, which we can use for hidden object removal. Note, however, that this is a left-handed system, as $(\hat{\mathbf{v}}_{side} \times \hat{\mathbf{v}}_{up}) \cdot \hat{\mathbf{v}}_{dir} < 0$.

OpenGL does not follow the standard model; instead, it chooses a slightly different approach. It maintains a right-handed system where the camera-relative view direction is aligned with the negative z -axis (Figure 6.3(b)). So in this case, the farther away the object is, its $-z$ coordinate gets larger relative to the camera. This is not as convenient for distance calculations, but it does allow us to remain in a right-handed coordinate system. This avoids having to worry about reflections when transforming from the world frame to the view frame, as we'll see below.

6.2.2 CONSTRUCTING THE VIEW-TO-WORLD TRANSFORMATION

Now that we have a way of representing and setting camera position and orientation, what do we do with it? The first step in the rendering process is to move all of the objects in our world so that they are no longer relative to the world frame, but are relative to the camera's view. Essentially, we want to transform the objects from the world frame to the view frame. This gives us a sense of what we can see from our camera position. In the view frame, those objects along the line of the view direction vector (i.e., the $-z$ -axis in the case of OpenGL) are in front of the camera and so will most

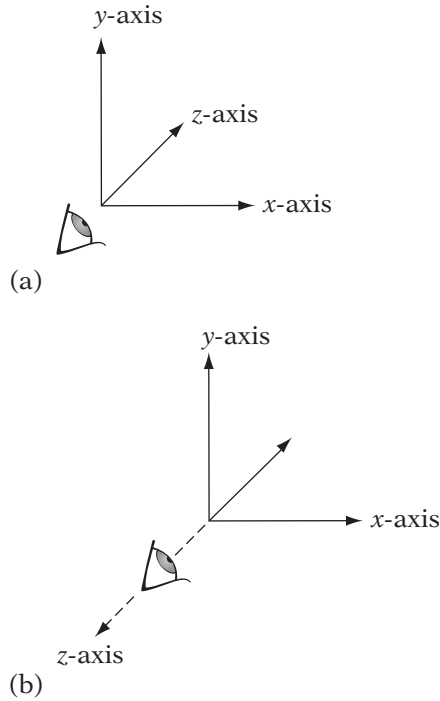


FIGURE 6.3 (a) Standard view frame axes. (b) OpenGL view frame axes.

likely be visible in our scene. Those on the other side of the plane formed by the view position, the view side vector, and the view up vector are behind the camera, and therefore not visible. In order to achieve this situation, we need to create a transformation from world space to view space, known as the world-to-view transformation, or more simply, the view transformation. We can represent this transformation as $\mathbf{M}_{world \rightarrow view}$.

However, rather than building this transformation directly, we usually find it easier to build $\mathbf{M}_{world \rightarrow view}^{-1}$, or $\mathbf{M}_{view \rightarrow world}$, first, and then invert to get our final world-to-view frame transformation. In order to build this, we'll make use of the principles we introduced in Chapter 4. If we look again at Figure 6.2, we note that we have an affine frame—the view frame—represented in terms of the world frame.

We can use this information to define the transformation from the view frame to the world frame as a 4×4 affine matrix. The origin E of the view frame is translated to the view position, so the translation vector \mathbf{y} is equal to $E - O$. We'll abbreviate this as \mathbf{v}_{pos} . Similarly, the view vectors represent how the standard basis vectors in view space are transformed into world space and become columns in the upper left 3×3 matrix \mathbf{A} . To build \mathbf{A} , however, we need

to define which standard basis vector in the view frame maps to a particular view vector in the world frame.

Recall that in the standard case, the camera's local x -axis represents $\hat{\mathbf{v}}_{side}$, the y -axis represents $\hat{\mathbf{v}}_{up}$, and the z -axis represents $\hat{\mathbf{v}}_{dir}$. This mapping indicates which columns the view vectors should be placed in, and the view position translation vector takes its familiar place in the right-most column. The corresponding transformation matrix is

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & \hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \end{bmatrix} \quad (6.1)$$

Note that in this case we are mapping from a left-handed view frame to the right-handed world frame, so the upper 3×3 is not a pure rotation but a rotation concatenated with a reflection.

For OpenGL, the only change is that we want to look down the $-z$ -axis. This is the same as the z -axis mapping to the negative view direction vector. So, the corresponding matrix is

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & -\hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \end{bmatrix} \quad (6.2)$$

In this case, since we are mapping from a right-handed frame to a right-handed frame, no reflection is necessary, and the upper 3×3 matrix is a pure rotation. Not having a reflection can actually be a benefit, particularly with some culling methods.

6.2.3 CONTROLLING THE CAMERA

It's not enough that we have a transformation for our camera that encapsulates position and orientation. More often we'll want to move it around the world. Positioning our camera is a simple enough matter of translating the view position, but controlling view orientation is another problem. One way is to specify the view vectors directly and build the matrix as described. This assumes, of course, that we already have a set of orthogonal vectors we want to use for our viewing system.

The more usual case is that we only know the view direction. For example, suppose we want to continually focus on a particular object in the world (known as the look-at object). We can construct the view direction by subtracting the view position from the object's position. But whether we have a given view direction or we generate it from the look-at object, we still need two other orthogonal vectors to properly construct an orthogonal basis. We can calculate them by using one additional piece of information: the world up vector. This is a fixed vector representing the "up" direction in the world frame. In our case, we'll use the z -axis basis vector \mathbf{k} (Figure 6.4), although in general, any vector that we care to call "up" will do. For example, suppose we



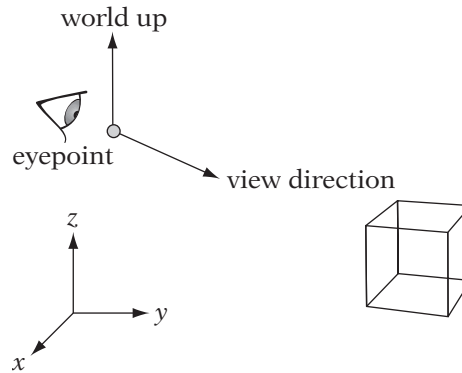


FIGURE 6.4 Look-at representation.

had a mission on a boat at sea and wanted to give the impression that the boat was rolling from side to side, without affecting the simulation. One method is to change the world up vector over time, oscillating between two keeled-over orientations, and use that to calculate your camera orientation.

For now, however, we'll use \mathbf{k} as our world up vector. Our goal is to compute orthonormal vectors in the world frame corresponding to our view vectors, such that one of them is our view direction vector $\hat{\mathbf{v}}_{dir}$, and our view up vector $\hat{\mathbf{v}}_{up}$ matches the world up vector as closely as possible. Recall that we can use Gram-Schmidt orthogonalization to create orthogonal vectors from a set of nonorthogonal vectors, so

$$\mathbf{v}_{up} = \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir}) \hat{\mathbf{v}}_{dir}$$

Normalizing gives us $\hat{\mathbf{v}}_{up}$. We can take the cross product to get the view side vector:

$$\hat{\mathbf{v}}_{side} = \hat{\mathbf{v}}_{dir} \times \hat{\mathbf{v}}_{up}$$

We don't need to normalize in this case because the two vector arguments are orthonormal. The resulting vectors can be placed as columns in the transformation matrix as before.

One problem may arise if we are not careful: What if $\hat{\mathbf{v}}_{dir}$ and \mathbf{k} are parallel? If they are equal, we end up with

$$\begin{aligned} \mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir}) \hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - 1 \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0} \end{aligned}$$

If they point in opposite directions we get

$$\begin{aligned}\mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir}) \hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - (-1) \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0}\end{aligned}$$

Clearly, neither case will lead to an orthonormal basis.

The recovery procedure is to pick an alternative vector that we know is not parallel, such as \mathbf{i} or \mathbf{j} . This will lead to what seems like an instantaneous rotation around the z -axis. To understand this, raise your head upward until you are looking at the ceiling. If you keep going, you'll end up looking at the wall behind you, but upside down. To maintain the view looking right-side up, you'd have to rotate your head 180 degrees around your view direction (don't try this at home). This is not a very pleasing result, so avoid aligning the view direction with the world up vector whenever possible.

There is a third possibility for controlling camera orientation. Suppose we want to treat our camera just like a normal object and specify a rotation matrix and translation vector. To do this we'll need to specify a starting orientation Ω for our camera and then apply our rotation matrix to find our camera's final orientation, after which we can apply our translation. Which orientation is chosen is somewhat arbitrary, but some are more intuitive and convenient than others. In our case, we'll say that in our default orientation the camera has an initial view direction along the world x -axis, an initial view up along the world z -axis, and an initial view side along the $-y$ -axis. This aligns the view up vector with the world up vector, and using the x -axis as the view direction fits the convention we set for objects' local space in Chapter 4.

Substituting these values into the view-to-world matrix for the standard left-handed view frame (equation 6.1) gives

$$\Omega_s = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The equivalent matrix for the right-handed OpenGL view frame (using equation 6.2) is

$$\Omega_{ogl} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Whichever system we are using, after this we apply our rotation to orient our frame in the direction we wish and, finally, the translation for the view

position. If the three column vectors in our rotation matrix are \mathbf{u} , \mathbf{v} , and \mathbf{w} , then for OpenGL the final transformation matrix is

$$\begin{aligned}\mathbf{M}_{view \rightarrow world} &= \mathbf{TR}\mathbf{\Omega}_{ogl} \\ &= \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\mathbf{j} & \mathbf{k} & -\mathbf{i} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -\mathbf{v} & \mathbf{w} & -\mathbf{u} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

6.2.4 CONSTRUCTING THE WORLD-TO-VIEW TRANSFORMATION

Using the techniques in the previous two sections, now we can create a transformation that takes us from view space to world space. To create the reverse operator, we need only to invert the transformation. Since we know that it is an affine transformation, we can invert it as

$$\mathbf{M}_{world \rightarrow view} = \begin{bmatrix} \mathbf{R}^{-1} & -(\mathbf{R}^{-1} \mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where \mathbf{R} is the upper 3×3 block of our view-to-world transformation. And since \mathbf{R} is the product of either a reflection and rotation matrix (in the standard case) or two rotations (in the OpenGL case), it is an orthogonal matrix, so we can compute its inverse by taking the transpose:

$$\mathbf{M}_{world \rightarrow view} = \begin{bmatrix} \mathbf{R}^T & -(\mathbf{R}^T \mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$



In practice, this transformation is usually calculated directly, rather than taking the inverse of an existing transformation. For example, OpenGL has a utility call `gluLookAt()` that computes the view transformation assuming a view position, desired view position, and world up vector. One possible implementation is as follows.

```
void LookAt( const IvVector3& eye,
             const IvVector3& lookAt,
             const IvVector3& up )
{
    // compute view vectors
    IvVector3 viewDir = lookAt - eye;
    IvVector3 viewSide;
```

```

IvVector3 viewUp;
viewDir.Normalize();
viewUp = up - up.Dot(viewDir)*viewDir;
viewUp.Normalize();
viewSide = viewDir.Cross(viewUp);

// now set up matrices
// build transposed rotation matrix
IvMatrix33 rotate;
rotate.SetRows( viewSide, viewUp, -viewDir );

// transform translation

IvVector3 eyeInv = -(rotate*eye);

// build 4x4 matrix
IvMatrix44 matrix;
matrix.Rotation(rotate);
matrix(0,3) = eyeInv.x;
matrix(1,3) = eyeInv.y;
matrix(2,3) = eyeInv.z;

// set view to world transformation
::SetViewTransform( matrix.mV );
}

```

Note that we use the method `IvMatrix33:SetRows()` to set the transformed basis vectors since we're setting up the inverse matrix, namely, the transpose. There is also no recovery code if the view direction and world up vectors are collinear—it is assumed that any external routine will ensure this does not happen. The renderer method `::SetViewTransform()` stores the calculated view transformation and is discussed in more detail in Section 6.7.

6.3 PROJECTIVE TRANSFORMATION

6.3.1 DEFINITION

Now that we have a method for controlling our view position and orientation, and for transforming our objects into the view frame, we can look at the second stage of the graphics pipeline: taking our 3D space and transforming it into a form suitable for display on a 2D medium. This process of transforming from \mathbb{R}^3 to \mathbb{R}^2 is called *projection*.

We've already seen one example of projection: using the dot product to project one vector onto another. In our current case, we want to project the points that make up the vertices of an object onto a plane, called the *projection plane* or the *view plane*. We do this by following a *line of projection* through each point and determining where it hits the plane. These lines could be perpendicular to the plane, but as we'll see, they don't have to be.

To understand how this works, we'll look at a very old form of optical projection known as the *camera obscura* (Latin for “dark room”). Suppose one enters a darkened room on a sunny day, and there is a small hole allowing a fraction of sunlight to enter the room. This light will be projected onto the opposite wall of the room, displaying an image of the world outside, albeit upside down and flipped left to right (Figure 6.5). This is the same principle that allows a pinhole camera to work; the hole is acting like the focal point of a lens. In this case, all the lines of projection pass through a single *center of projection*. We can determine where a point will project to on the plane by constructing a line through both the original point and the center of projection and calculating where it will intersect the plane of projection. The virtual film in this case is a rectangle on the view plane, known as the *view window*. This will eventually get mapped to our display.

This sort of projection is known as *perspective projection*. Note that this relates to our perceived view in the real world. As an object moves farther away, its corresponding projection will shrink on the projection plane. Similarly, lines that are parallel in view space will appear to converge as their extreme points move farther away from the view position. This gives us a result consistent with our expected view in the real world. If we stand on some railroad tracks and look down a straight section, the rails will converge in the distance, and the ties will appear to shrink in size and become closer together. In most cases, since we are rendering real-world scenes — or at least, scenes that we want to be perceived as real world — this will be the projection we will use.

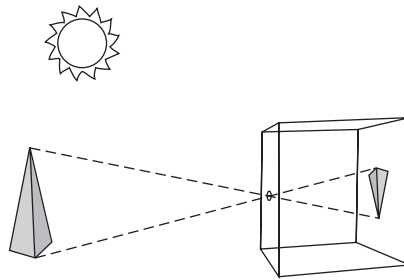


FIGURE 6.5 Camera obscura.

There is, of course, one minor problem: The projected image is upside down and backwards. One possibility is just to flip the image when we display it on our medium. This is what happens with a camera: The image is captured on film upside down, but we can just rotate the negative or print to view it properly. This is not usually done in graphics. Instead, the projection plane is moved to the other side of the center of projection, which is now treated as our view position (Figure 6.6). As we'll see, the mathematics for projection in this case are quite simple, and the objects located in the forward direction of our view will end up being projected right-side up. The objects behind the view will end up projecting upside down, but (a) we don't want to render them anyway, and (b) as we'll see, there are ways of handling this situation.

An alternate type of projection is *parallel projection*, which can be thought of as a perspective projection where the center of projection is infinitely distant. In this case, the lines of projection do not converge; they always remain parallel (Figure 6.7), hence the name. The placement of the view position and view plane is irrelevant in this case, but we place them in the same relative location to maintain continuity with perspective projection.

Parallel projection produces a very odd view if used for a scene: Objects remain the same size no matter how distant they are, and parallel lines remain parallel. Parallel projections are usually used for computer-assisted design (CAD) programs, where maintaining parallel lines is important. They are also useful for rendering 2D elements like interfaces; no matter how far from the eye a model is placed, it always will be the same size, presumably the size we expect.

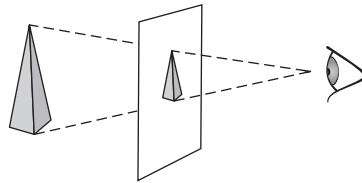


FIGURE 6.6 Perspective projection.

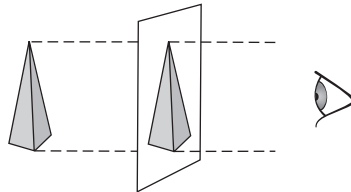


FIGURE 6.7 Orthographic parallel projection.

A parallel projection where the lines of projection are perpendicular to the view plane is called an *orthographic projection*. By contrast, if they are not perpendicular to the view plane, this is known as an *oblique projection* (Figure 6.8). Two common oblique projections are the *cavalier projection*, where the projection angle is 45 degrees, and the *cabinet projection*, where the projection angle is $\cot^{-1}(1/2)$. When using cavalier projections, projected lines have the same length as the original lines, so there is no perceived foreshortening. This is useful when printing blueprints, for example, as any line can be measured to find the exact length of material needed to build the object. With cabinet projections, lines perpendicular to the projection plane foreshorten to half their length (hence the $\cot^{-1}(1/2)$), which gives a more realistic look without sacrificing the need for parallel lines.

We can also have oblique perspective projections where the line from the center of the view window to the center of projection is not perpendicular to the view plane. For example, suppose we need to render a mirror. To do so, we'll render the space using a plane reflection transformation and clip it to the boundary of the mirror. The plane of the mirror is our projection plane, but it may be at an angle to our view direction (Figure 6.9). For now, we'll concentrate on constructing projective transformations perpendicular to the projection plane and examine these special cases later.

As a side note, oblique projections can occur in the real world. The classic pictures we see of tall buildings, shot from the ground but with parallel sides,

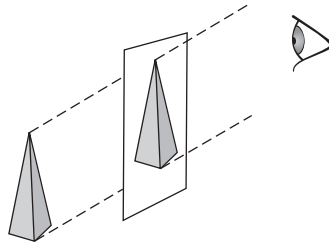


FIGURE 6.8 Oblique parallel projection.

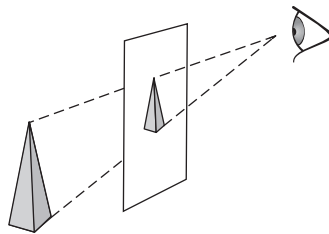


FIGURE 6.9 Oblique perspective projection.

are done with a “view camera.” This device has an accordion-pleated hood that allows the photographer to bend and tilt the lens up while keeping the film parallel to the side of the building. Ansel Adams also used such a camera to capture some of his famous landscape photographs.

6.3.2 NORMALIZED DEVICE COORDINATES

Before we begin projecting, our objects have passed through the view stage of the pipeline and so are in view frame coordinates. We will be projecting from this space in \mathbb{R}^3 to the view plane, which is in \mathbb{R}^2 . In order to accomplish this, it will be helpful to define a frame for the space of the view plane. We’ll use as our origin the center of the view window, and create basis vectors that align with the sides of the view window, with magnitudes of half the width and height of the window, respectively (Figure 6.10(a)). Within this frame, our view window is transformed into a square two units wide and centered at the origin, bounded by the $x = 1$, $x = -1$, $y = 1$, and $y = -1$ lines (Figure 6.10(b)).

Using this as our frame provides a certain amount of flexibility when mapping to devices of varying size. Rather than transform directly to our screen area, which could be of variable width and height, we use this normalized form as an intermediate step to simplify our calculations and then do the screen conversion as our final step. Because of this, coordinates in this frame are known as *normalized device coordinates*.

To take advantage of the normalized device coordinate frame, or *NDC space*, we’ll want to create our projection so that it always gives us the -1 to 1 behavior, regardless of the exact view configuration. This helps us to compartmentalize the process of projection (just as the view matrix did for viewing). When we’re done projecting, we’ll stretch and translate our NDC values to match the width and height of our display.

To simplify this mapping to the NDC frame, we will begin by using a view window in the view frame with a height of two units. This means that for the case of a centered view window, xy coordinates on the view plane will be equal to the projected coordinates in the NDC frame. In this way we can consider the projection as related to the view plane in view coordinates and not worry about a subsequent transformation.

6.3.3 VIEW FRUSTUM

The question remains: How do we determine what will lie within our view window? We could, naively, project all of the objects in the world to the view plane and then, when converting them to pixels, ignore those pixels that lie outside of the view window. However, for a large number of objects this would be very inefficient. It would be better to constrain our space to a convex volume,

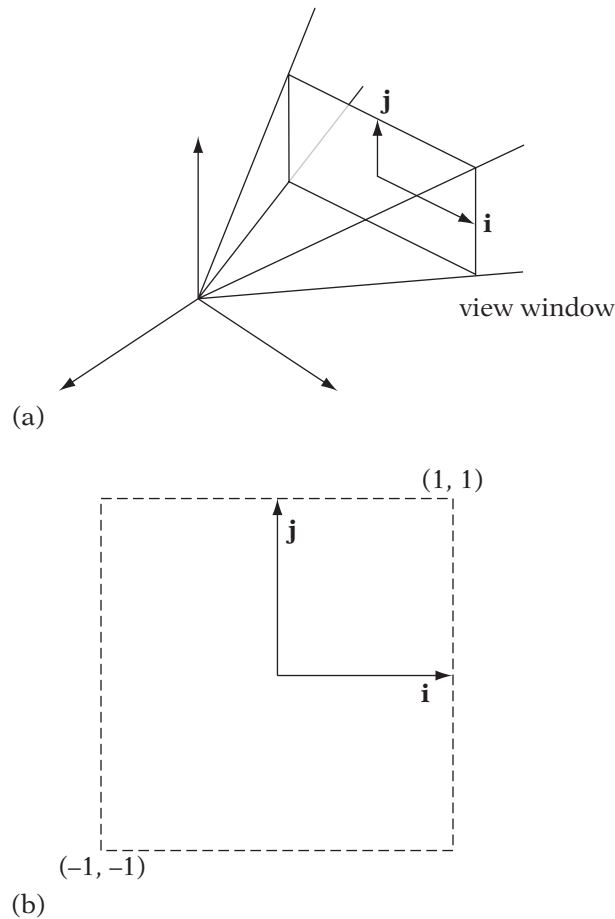


FIGURE 6.10 (a) NDC frame in view window, and (b) view window after NDC transformation.

specified by a set of six planes. Anything inside these planes will be rendered; everything outside them will be ignored. This volume is known as the *view frustum*, or *view volume*.

To constrain what we render in the view frame xy directions, we specify four planes aligned with the edges of the view window. For perspective projection each plane is specified by the view position and two adjacent vertices of the view window (Figure 6.11), producing a semi-infinite pyramid. The angle between the upper plane and the lower plane is called the *vertical field of view*.

There is a relationship between field of view, view window size, and view plane distance: Given two, we can easily find the third. For example, we can fix the view window size, adjust the field of view, and then compute the distance

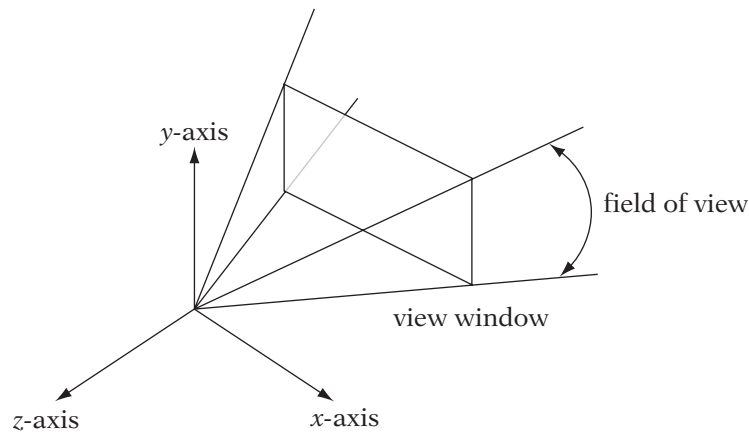


FIGURE 6.11 Perspective view frustum (right-handed system).

to the view plane. As the field of view gets larger, the distance to the view plane needs to get smaller to maintain the view window size. Similarly, a small field of view will lead to a longer view plane distance. Alternatively, we can set the distance to the view plane to a fixed value and use the field of view to determine the size of our view window. The larger the field of view, the larger the window and the more objects are visible in our scene. This gives us a primitive method for creating telephoto (narrow field of view) or wide-angle (wide field of view) lenses. We will discuss the relationship among these three quantities in more detail when we cover perspective projection.

In our case, the view window size is fixed, so when adjusting our field of view, we will move the view plane relative to the center of projection. This continues to match our camera analogy: The film size is fixed and the lens moves in and out to create a telephoto or wide-angle effect.

Usually the field of view chosen needs to match the display medium, as the user perceives it, as much as possible. For a standard monitor placed about three feet away, the monitor only covers about a 25- to 30-degree field of view from the perspective of the user, so we would expect that we would use a field of view of that size in the game. However, this constrains the amount we can see in the game to a narrow area, which feels unnatural because we're used to a 180-degree field of view in the real world. The usual compromise is to set the field of view to the range of 60–90 degrees. The distortion is not that perceptible and it allows the user to see more of the game world. If the monitor were stretched to cover more of your personal field of view, as in a widescreen monitor or some virtual reality systems, a larger field of view would be appropriate. And of course, if the desired effect is of a telephoto or wide-angle lens, a narrower or wider field of view, respectively, is appropriate.

For parallel projection, the xy culling planes are parallel to the direction of projection, so opposite planes are parallel and we end up with a parallelepiped that is open at two ends (Figure 6.12). There is no concept of field of view in this case.

In both cases, to complete a closed view frustum we also define two planes that constrain objects in the view frame z -direction: the near and far planes (Figure 6.13). With perspective projection it may not be obvious why we need a near plane, since the xy planes converge at the center of projection, closing the viewing region at that end. However, as we will see when we start talking about the perspective transformation, rendering objects at the view frame origin (which in our case is the same as the center of projection) can lead to a possible division by zero. This would adversely affect our rendering process. We could also, like some viewing systems, use the view plane as the near plane, but not doing so allows us a little more flexibility.

In some sense, the far plane is optional. Since we don't have an infinite number of objects or an infinite amount of game space, we could forego using the far plane and just render everything within the five other planes. However, the far plane is useful for culling objects and area from our rendering process, so having a far plane is good for efficiency's sake. It is also extremely important in the hidden surface removal method of z -buffering; the distance between the near and far planes is a factor in determining the precision we can expect in our z values. We'll discuss this in more detail in Chapter 9.

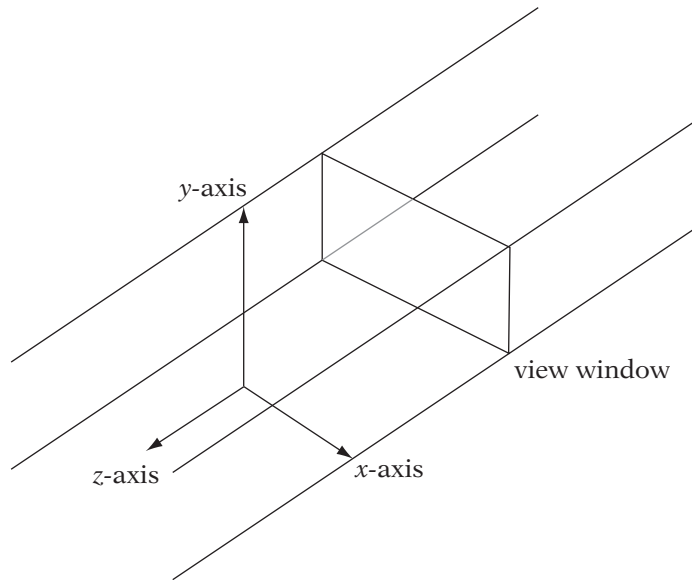


FIGURE 6.12 Parallel view frustum (right-handed system).

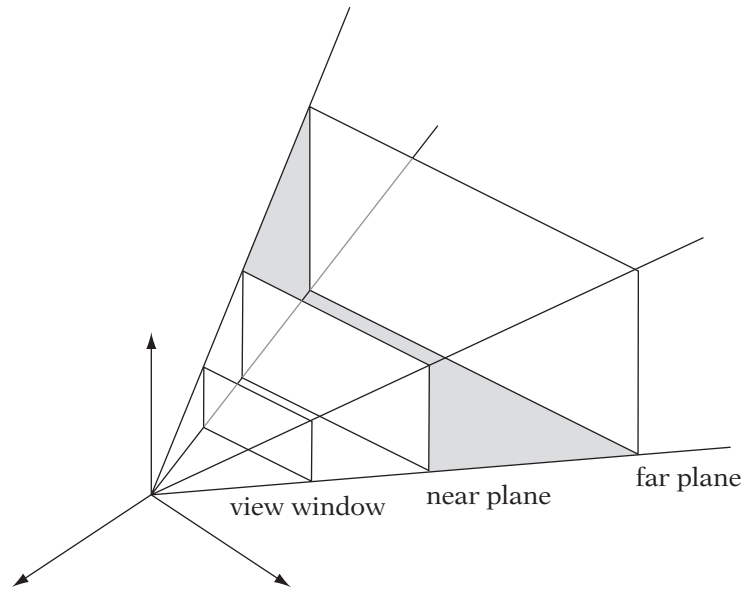


FIGURE 6.13 View frustum with near plane and far plane.

6.3.4 HOMOGENEOUS COORDINATES

There is one more topic we need to cover before we can start discussing projection. Previously we stated that a point in \mathbb{R}^3 can be represented by $(x, y, z, 1)$ without explaining much about what that might mean. This representation is part of a more general representation for points known as homogeneous coordinates, which prove useful to us when handling perspective projections. In general, homogeneous coordinates work as follows: If we have a “standard” representation in n -dimensional space, then we can represent the same point in a $(n + 1)$ -dimensional space by scaling the original coordinates by a single value and then adding the scalar to the end as our final coordinate. Since we can choose from an infinite number of scalars, a single point in \mathbb{R}^n will be represented by an infinite number of points in the $(n + 1)$ -dimensional space. This $(n + 1)$ -dimensional space is called a *real projective space* or $\mathbb{R}P^n$. In computer graphics parlance, the real projective space $\mathbb{R}P^3$ is also often called *homogeneous space*.

Suppose we start with a point (x, y, z) in \mathbb{R}^3 , and we want to map it to a point (x', y', z', w) in homogeneous space. We pick a scalar for our fourth element w , and scale the other elements by it, to get (xw, yw, zw, w) . As we might expect, our standard value for w will be 1, so (x, y, z) maps to $(x, y, z, 1)$. To map back to 3D space, divide the first three coordinates by w , so (x', y', z', w) goes to $(x'/w, y'/w, z'/w)$. Since our standard value for w is just 1, we could

just drop the w : $(x', y', z', 1) \rightarrow (x', y', z')$. However, in the cases that we'll be concerned with next, we need to perform the division by w .

What happens when $w = 0$? In this case, a point in $\mathbb{R}P^3$ doesn't represent a point in \mathbb{R}^3 , but a vector. We can think of this as a "point at infinity." While we will try to avoid cases where $w = 0$, they do creep in, so checking for this before performing the homogeneous division is often wise.

6.3.5 PERSPECTIVE PROJECTION



Since this is the most common projective transform we'll encounter, we'll begin by constructing the mathematics necessary for the perspective projection. To simplify things, let's take a 2D view of the situation on the yz plane and ignore the near and far planes for now (Figure 6.14). We have the y -axis pointing up, as in the view frame, and the projection direction along the negative z -axis as it would be in OpenGL. The point on the left represents our center of projection, and the vertical line our view plane. The diagonal lines represent our y culling planes.

Suppose we have a point P_v in view coordinates that lies on one of the view frustum planes, and we want to find the corresponding point P_s that lies on the view plane. Finding the y coordinate of P_s is simple: We follow the line of projection along the plane until we hit the top of the view window. Since the height of the view window is 2 and is centered on 0, the y coordinate of P_s is half the height of the view window, or 1. The z coordinate will be negative since we're looking along the negative z -axis and will have a magnitude equal to the distance d from the view position to the projection plane. So, the z coordinate will be $-d$.

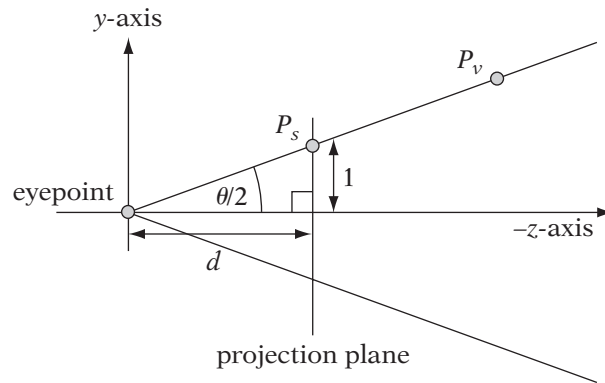


FIGURE 6.14 Perspective projection construction.

But how do we compute d ? As we see, the cross section of the y view frustum planes are represented as lines from the center of projection through the extents of the view window $(1, d)$ and $(-1, d)$. The angle between these lines is our field of view θ_{fov} . We'll simplify things by considering only the area that lies above the negative z -axis; this bisects our field of view to an angle of $\theta_{fov}/2$. If we look at the triangle bounded by the negative z -axis, the cross section of the upper view frustum plane, and the cross section of the projection plane, we can use trigonometry to compute d . Since we know the distance between the negative z -axis and the extreme point P_s is 1, we can say that

$$\frac{1}{d} = \tan(\theta_{fov}/2)$$

Rewriting this in terms of d , we get

$$\begin{aligned} d &= \frac{1}{\tan\left(\frac{\theta_{fov}}{2}\right)} \\ &= \cot\left(\frac{\theta_{fov}}{2}\right) \end{aligned}$$

So for this fixed-view window size, as long as we know the angle of field of view, we can compute the distance d , and vice versa.

This gives the coordinates for any point that lies on the upper y view frustum plane; in this 2D cross section they all project down to a single point $(1, -d)$. Similarly, points that lie on the lower y frustum plane will project to $(-1, -d)$. But suppose we have a general point (y_v, z_v) in view space. We know that its projection will lie on the view plane as well, so its z_{ndc} coordinate will be $-d$. But how do we find y_{ndc} ?

We can compute this by using similar triangles (Figure 6.15). If we have a point (y_v, z_v) , the length of the sides of the corresponding right triangle in our diagram are y_v and $-z_v$ (since we're looking down the $-z$ -axis, any visible z_v is negative, so we need to negate it to get a positive value). The length of sides of the right triangle for the projected point are y_{ndc} and d . By similar triangles (both have the same angles), we get

$$\frac{y_{ndc}}{d} = \frac{y_v}{-z_v}$$

Solving for y_{ndc} , we get

$$y_{ndc} = \frac{dy_v}{-z_v}$$

This gives us the coordinate in the y direction. If our view region was square, then we could use the same formula for the x direction. Most, however,

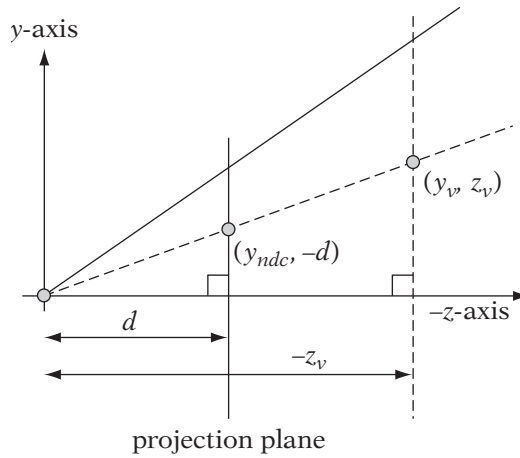


FIGURE 6.15 Perspective projection similar triangles.

are rectangular to match the relative dimensions of a computer monitor or other viewing device. We must correct for this by the aspect ratio of the view region. The aspect ratio a is defined as

$$a = \frac{w_v}{h_v}$$

where w_v and h_v are the width and height of the view rectangle, respectively. We're going to assume that the NDC view window height remains at 2 and correct the NDC view width by the aspect ratio. This gives us a formula for similar triangles of

$$\frac{ax_{ndc}}{d} = \frac{x_v}{-z_v}$$

Solving for x_{ndc} :

$$x_{ndc} = \frac{dx_v}{-az_v}$$

So, our final projection transformation equations are

$$\begin{aligned} x_{ndc} &= \frac{dx_v}{-az_v} \\ y_{ndc} &= \frac{dy_v}{-z_v} \end{aligned}$$

The first thing to notice is that we are dividing by a z coordinate, so we will not be able to represent the entire transformation by a matrix operation, since it is neither linear nor affine. However, it does have some affine elements—scaling by d and d/a , for example—which can be performed by a transformation matrix. This is where the conversion from homogeneous space comes in. Recall that to transform from $\mathbb{R}P^3$ to \mathbb{R}^3 we need to divide the other coordinates by the w value. If we can set up our matrix to map $-z_v$ to our w value, we can take advantage of the homogeneous divide to handle the nonlinear part of our transformation. We can write the situation before the homogeneous divide as a series of linear equations:

$$\begin{aligned}x' &= \frac{d}{a}x \\y' &= dy \\z' &= dz \\w' &= -z\end{aligned}$$

and treat this as a four-dimensional (4D) linear transformation. Looking at our basis vectors, \mathbf{e}_0 will map to $(d/a, 0, 0, 0)$, \mathbf{e}_1 to $(0, d, 0, 0)$, \mathbf{e}_2 to $(0, 0, d, -1)$, and \mathbf{e}_3 to $(0, 0, 0, 0)$, since w is not used in any of the equations.

Based on this, our homogeneous perspective matrix is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

As expected, our transformed w value no longer will be 1. Also note that the right-most column of this matrix is all zeros, which means that this matrix has no inverse. This is to be expected, since we are losing one dimension of information. Individual points in view space that lie along the same line of projection will project to a single point in NDC space. Given only the points in NDC space, it would be impossible to reconstruct their original positions in view space.

Let's see how this matrix works in practice. If we multiply it by a generic point in view space, we get

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} dx_v/a \\ dy_v \\ dz_v \\ -z_v \end{bmatrix}$$

Dividing out the w (also called the reciprocal divide), we get

$$\begin{aligned}x_{ndc} &= \frac{dx_v}{-az_v} \\y_{ndc} &= \frac{dy_v}{-z_v} \\z_{ndc} &= -d\end{aligned}$$

which is what we expect.

So far, we have dealt with projecting x and y and completely ignored z . In the preceding derivation all z values map to $-d$, the negative of the distance to the projection plane. While losing a dimension makes sense conceptually (we are projecting from a 3D space down to a 2D plane, after all), for practical reasons it is better to keep some measure of our z values around for z -buffering and other depth comparisons (discussed in more detail in Chapter 9). Just as we're mapping our x and y values within the view window to an interval of $[-1, 1]$, we'll do the same for our z values within the near plane and far plane positions. We'll specify the near and far values n and f relative to the view position, so points lying on the near plane have a z_v value of $-n$, which maps to a z_{ndc} value of -1 . Those points lying on the far plane have a z_v value of $-f$ and will map to 1 (Figure 6.16).

We'll derive our equation for z_{ndc} in a slightly different way than our xy coordinates. There are two parts to mapping the interval $[-n, -f]$ to $[-1, 1]$. The first is scaling the interval to a width of 2, and the second is translating it to $[-1, 1]$. Ordinarily, this would be a straightforward linear process, however, we also have to contend with the final w divide. Instead, we'll create

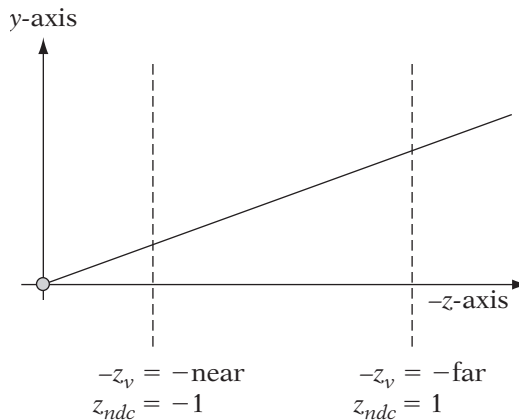


FIGURE 6.16 Perspective projection: z values.

a perspective matrix with unknowns for the scaling and translation factors and use the fact that we know the final values for $-n$ and $-f$ to solve for the unknowns. Our starting perspective matrix, then, is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where A and B are our unknown scale and translation factors, respectively. If we multiply this by a point $(0, 0, -n)$ on our near plane, we get

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -An + B \\ n \end{bmatrix}$$

Dividing out the w gives

$$z_{ndc} = -A + \frac{B}{n}$$

We know that any point on the near plane maps to a normalized device coordinate of -1 , so we can substitute -1 for z_{ndc} and solve for B , which gives us

$$B = (A - 1)n \quad (6.3)$$

Now we'll substitute equation 6.3 into our original matrix and multiply by a point $(0, 0, -f)$ on the far plane:

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & (A - 1)n \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Af + (A - 1)n \\ f \end{bmatrix}$$

This gives us a z_{ndc} of

$$\begin{aligned} z_{ndc} &= -A + (A - 1)\frac{n}{f} \\ &= -A + A\left(\frac{n}{f}\right) - \frac{n}{f} \\ &= A\left(\frac{n}{f} - 1\right) - \frac{n}{f} \end{aligned}$$

Setting z_{ndc} to 1 and solving for A , we get

$$\begin{aligned} A \left(\frac{n}{f} - 1 \right) - \frac{n}{f} &= 1 \\ A \left(\frac{n}{f} - 1 \right) &= \frac{n}{f} + 1 \\ A &= \frac{\frac{n}{f} + 1}{\frac{n}{f} - 1} \\ &= \frac{n + f}{n - f} \end{aligned}$$

If we substitute this into equation 6.3, we get

$$B = \frac{2nf}{n - f}$$

So, our final perspective matrix is

$$\mathbf{M}_{persp} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The matrix that we have generated is the same one produced by an OpenGL call: `gluPerspective()`. This function takes the field of view,¹ aspect ratio, and near and far plane settings, builds the perspective matrix, and multiplies it by the current matrix.

It is important to be aware that this matrix will not work for all viewing systems. For one thing, for most other viewing systems (i.e., other than OpenGL), our view frame looks down the positive z -axis, so this affects both our xy and z transformations. For example, in this case we have mapped $[-n, -f]$ to $[-1, 1]$. With the standard system we would want to begin by mapping $[n, f]$ to the NDC z range. In addition, this range is not always set to $[-1, 1]$. Direct3D, for one, has a default mapping of to $[0, 1]$ in the z direction.

1. Recall that our value d is generated from the field of view by $d = \cot(\theta_{fov}/2)$.

Using the standard view frame and this mapping gives us a perspective transformation matrix of

$$\mathbf{M}_{pD3D} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix can be derived using the same principles described above.

When setting up a perspective matrix, it is good to be aware of the issues involved in rasterizing z values. In particular, to maintain z precision keep the near and far planes as close together as possible. More details on managing perspective z precision can be found in Chapter 9.

6.3.6 OBLIQUE PERSPECTIVE



The matrix we constructed in the previous section is an example of a standard perspective matrix, where the direction of projection through the center of the view window is perpendicular to the view plane. A more general example of perspective is generated by the OpenGL `glFrustum()` call. This call takes six parameters: the near and far z distances, as before, and four values that define our view window on the near z plane: the x interval $[l, r]$ (left, right) and the y interval $[b, t]$ (bottom, top). Figure 6.17(a) shows how this looks in \mathbb{R}^3 , and Figure 6.17(b) shows the cross section on the yz plane. As we can see, these values need not be centered around the z -axis, so we can use them to generate an oblique projection.

To derive this matrix, once again we begin by considering similar triangles in the y direction. Remember that given a point $(y_v, -z_v)$, we project to a point on the view plane $(dy_v/-z_v, -d)$, where d is the distance to the projection. However, since we're using our near plane as our projection plane, this is just $(ny_v/-z_v, -n)$. The projection remains the same, we're just moving the window of projected points that lie within our view frustum.

With our previous derivation, we could stop at this point because our view window on the projection plane was already in the interval $[-1, 1]$. However, our new view window lies in the interval $[b, t]$. We'll have to adjust our values to properly end up in NDC space. The first step is to translate the center of the window, located at $(t + b)/2$, to the origin. Applying this translation to the current projected y coordinate gives us

$$y' = y - \frac{(t + b)}{2}$$

We now need to scale to change our interval from a magnitude of $(t - b)$ to a magnitude of 2 by using a scale factor $2/(t - b)$:

$$y_{ndc} = \frac{2y}{t - b} - \frac{2(t + b)}{2(t - b)} \quad (6.4)$$

If we substitute $ny_v/-z_v$ for y and simplify, we get

$$y_{ndc} = \frac{2n \frac{y_v}{-z_v}}{t - b} - \frac{2(t + b)}{2(t - b)}$$

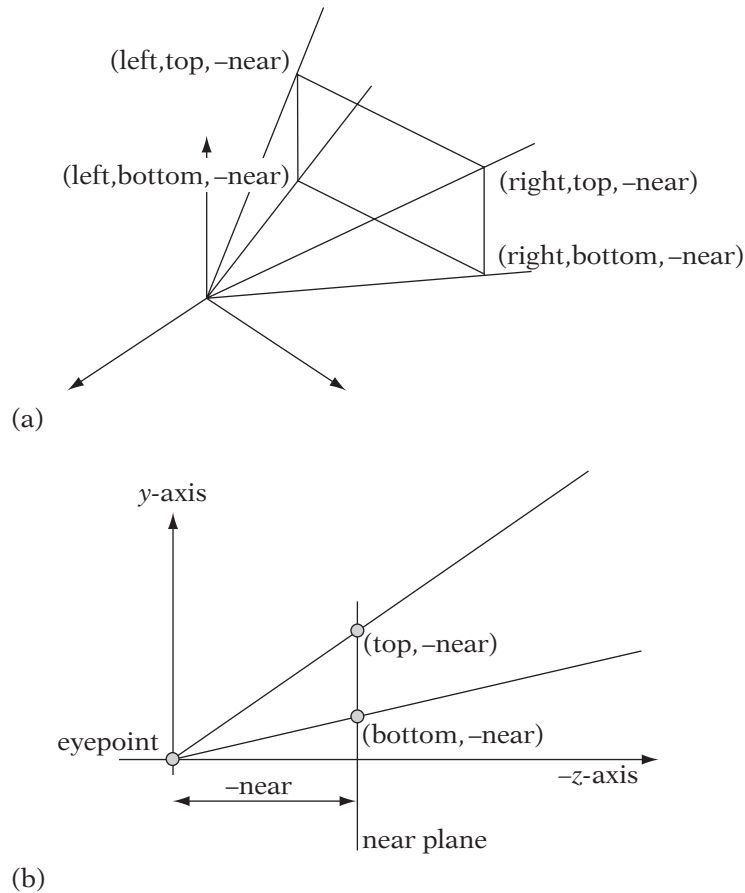


FIGURE 6.17 (a) View window for `glFrustum`, 3D view. (b) View window for `glFrustum`, cross section.

$$\begin{aligned}
&= \frac{2n \frac{y_v}{-z_v}}{t-b} - \frac{(t+b) \frac{-z_v}{-z_v}}{t-b} \\
&= \frac{1}{-z_v} \left(\frac{2n}{t-b} y_v + \frac{t+b}{t-b} z_v \right)
\end{aligned}$$

A similar process gives us the following for the x direction:

$$x_{ndc} = \frac{1}{-z_v} \left(\frac{2n}{r-l} x_v + \frac{r+l}{r-l} z_v \right)$$

We can use the same A and B from our original perspective matrix, so our final projection matrix is

$$\mathbf{M}_{obl\text{persp}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

A casual inspection of this matrix gives some sense of what's going on here. We have a scale in the x , y , and z directions, which provides the mapping to the interval $[-1, 1]$. In addition, we have a translation in the z direction to align our interval properly. However, in the x and y directions, we are performing a z -shear to align the interval, which provides us with the oblique projection.

The equivalent Direct3D matrix is

$$\mathbf{M}_{opD3D} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

As unusual as it might appear, there are a number of applications of oblique perspective projection in real-time graphics. First of all, it can be used in mirrors: We treat the mirror as our view window, the mirror plane as our view plane, and the viewer's location as our view position. If we apply a plane reflection to all of our objects, flipping them around the mirror plane, and then render with the appropriate visual effects, we will end up with a result in the view window that emulates a mirror.

Another application is stereo. By using a single view plane and view window, but separate view positions for each eye that are offset from the standard center of projection, we get slightly different projections of the world. By using either a red-blue system to color each view differently, or some sort of

goggle system that displays the left and right views in each eye appropriately, we can provide a good approximation of stereo vision. We have included an example of this on the CD-ROM.

Finally, this can be used for a system called *fishtank VR*. Normally we think of VR as a helmet attached to someone's head with a display for each eye. However, by attaching a tracking device to a viewer's head we can use a single display and create an illusion that we are looking through a window into a world on the other side. This is much the same principle as the mirror: The display is our view window and the tracked location of the eye is our view position. Add stereo and this gives a very pleasing effect.

6.3.7 ORTHOGRAPHIC PARALLEL PROJECTION

SOURCE CODE
DEMO
Orthographic

After considering perspective projection in two forms, orthographic projection is much easier. Examine Figure 6.18, which shows a side view of our projection space as before, with the lines of projection passing through the view plane and the near and far planes shown as vertical lines. This time the lines of projection are parallel to each other (hence this is a parallel projection) and parallel to the z -axis (hence an orthographic projection).

We can use this to help us generate the matrix for the OpenGL `glOrtho()` call. Like `glFrustum()`, this call takes six parameters: the near and far z distances, and four values l , r , b , and t that define our view window on the near z plane. As before, the near plane is our projection plane, so a point (y_v, z_v) projects to a point $(y_v, -n)$. Note that since this is a parallel projection, there is no division by z or scale by d ; we just use the y value directly. Like `glFrustum()` we now need to consider only values between t and b and scale and translate

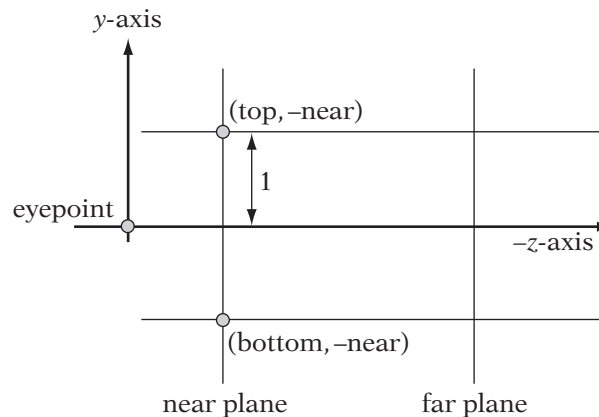


FIGURE 6.18 Orthographic projection construction.

them to the interval $[-1, 1]$. Substituting y_v into our range transformation equation 6.4, we get

$$y_{ndc} = \frac{2y_v}{t-b} - \frac{t+b}{t-b}$$

A similar process gives us the equation for x_{ndc} . We can do the same for z_{ndc} , but since our viewable z values are negative and our values for n and f are positive, we need to negate our z value and then perform the range transformation. The result of all three equations is

$$\mathbf{M}_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There are a few things we can notice about this matrix. First of all, multiplying by this matrix gives us a w value of 1, so we don't need to perform the homogeneous division. This means that our z values will remain linear; that is, they will not compress as they approach the far plane. This gives us better z resolution at far distances than the perspective matrices. It also means that this is a linear transformation matrix and possibly invertible.

Secondly, in the x and y directions, what was previously a z -shear in the oblique perspective matrix has become a translation. Before, we had to use shear, because for a given point the displacement was dependent on the distance from the view position. Because the lines of projection are now parallel, all points displace equally, so only a translation is necessary.

The Direct3D equivalent matrix is

$$\mathbf{M}_{orthoD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.3.8 OBLIQUE PARALLEL PROJECTION



While most of the time we'll want to use orthographic projection, we may from time to time need an oblique parallel projection. For example, suppose for part of our interface we wish to render our world as a set of schematics or display particular objects with a 2D CAD/CAM feel. This set of projections will achieve our goal.

Neither OpenGL nor Direct3D has a particular routine that handles oblique parallel projections, so we'll derive one ourselves. We will give our projection a slight oblique angle ($\cot^{-1}(1/2)$, which is about 63.4 degrees), which gives a 3D look without perspective. More extreme angles in x and y tend to look strangely flat.

Figure 6.19 is another example of our familiar cross section, this time showing the lines of projection for our oblique projection. As we can see, we move one unit in the y direction for every two units we move in the z direction. Using the formula of $\tan(\theta) = \text{opposite}/\text{adjacent}$, we get

$$\tan(\theta) = \frac{2}{1}$$

$$\cot(\theta) = \frac{1}{2}$$

$$\theta = \cot^{-1} \frac{1}{2}$$

which confirms the expected value for our oblique angle.

As before, we'll consider the yz case first and extrapolate to x . Moving one unit in y and two units in $-z$ gives us the vector $(1, -2)$, so the formula for the line of projection for a given point P is

$$L(t) = P + t(1, -2)$$

We're only interested in where this line crosses the near plane, or where

$$P_z - 2t = -n$$

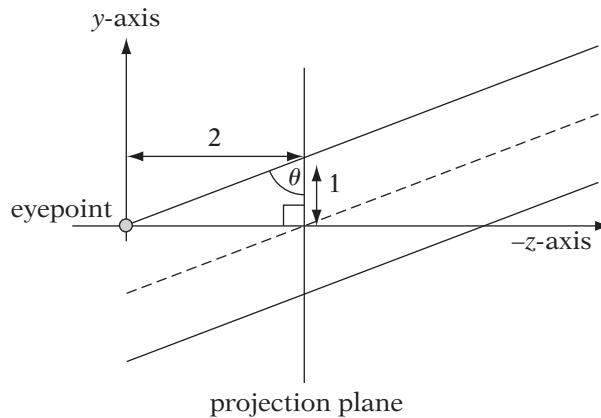


FIGURE 6.19 Example of oblique parallel projection.

Solving for t , we get

$$t = \frac{1}{2}(n + P_z)$$

Plugging this into the formula for the y coordinate of $L(t)$, we get

$$y' = P_y + \frac{1}{2}(n + P_z)$$

Finally, we can plug this into our range transformation equation 6.4 as before to get

$$\begin{aligned} y_{ndc} &= 2 \frac{\left[y_v + \frac{1}{2}(n + z_v) \right]}{t - b} - \frac{t + b}{t - b} \\ &= \frac{2y_v}{t - b} - \frac{t + b}{t - b} + \frac{z_v + n}{t - b} \end{aligned}$$

Once again, we examine our transformation equation more carefully. This is the same as the orthographic transformation we had before, with an additional z -shear, as we'd expect for an oblique projection. In this case, the shear plane is the near plane rather than the xy plane, so we add an additional factor of $\frac{n}{t-b}$ to take this into account.

A similar process can be used for x . Since the oblique projection has a z -shear, z is not affected and so,

$$\mathbf{M}_{obl} = \begin{bmatrix} \frac{2}{r-l} & 0 & \frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & \frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Direct3D equivalent matrix is

$$\mathbf{M}_{oblD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.4 CULLING AND CLIPPING

6.4.1 WHY CULL OR CLIP?

We will now take a detour from discussing the transformation aspect of our pipeline to discuss a process that often happens at this point in many renderers. In order to improve rendering, both for speed and appearance's sake, it is necessary to cull and clip objects. Culling is the process of removing objects from consideration for some process, whether it be rendering, simulation, or collision detection. In this case, that means we want to ignore any models or whole pieces of geometry that lie outside of the view frustum, since they will never end up being projected to the view window. In Figure 6.20, the lighter objects lie outside of the view frustum and so will be culled for rendering.

Clipping is the process of cutting geometry to match a boundary, whether it be a polygon or, in our case, a plane. Vertices that lie outside the boundary will be removed and new ones generated for each edge that crosses the boundary. For example, in Figure 6.21 we see a cube being clipped by a plane, showing the extra vertices created where each edge intersects the plane. We'll use this for any models that cross the view frustum, cutting the geometry so that it fits within the frustum. We can think of this as slicing a piece of geometry off for every frustum plane.

Why should we want to use either of these for rendering? For one thing, it is more efficient to remove any data that will not ultimately end up on the screen. While copying the transformed object to the frame buffer (a process called *rasterization*) is almost always done in hardware and thus is fast, it is not free. Anywhere we can avoid unnecessary work is good.

But even if we had infinite rasterization power, we would still want to cull and clip when performing perspective projection. Figure 6.22 shows one

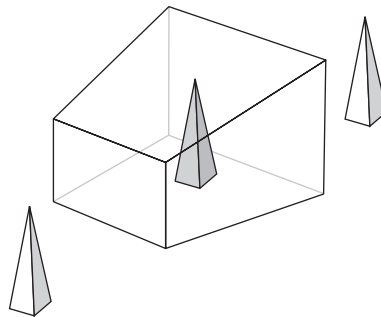
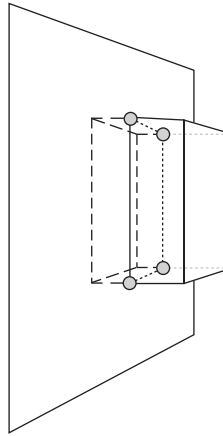
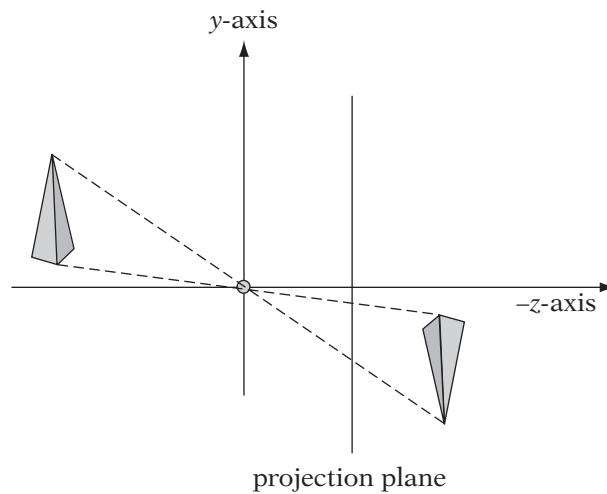


FIGURE 6.20 View frustum culling.

**FIGURE 6.21** View frustum clipping.**FIGURE 6.22** Projection of objects behind the eye.

example why. Recall that we finessed the problem of the camera obscura inverting images by moving the view plane in front of the center of projection. However, we still have the same problem if an object is behind the view position; it will end up projected upside down. The solution is to cull objects that lie behind the view position.

Figure 6.23(a) shows another example. Suppose we have a polygon edge \overline{PQ} that crosses the $z = 0$ plane. Endpoint P projects to a point P' on the view plane, and Q to Q' . With the correct projection, the intermediate points of the

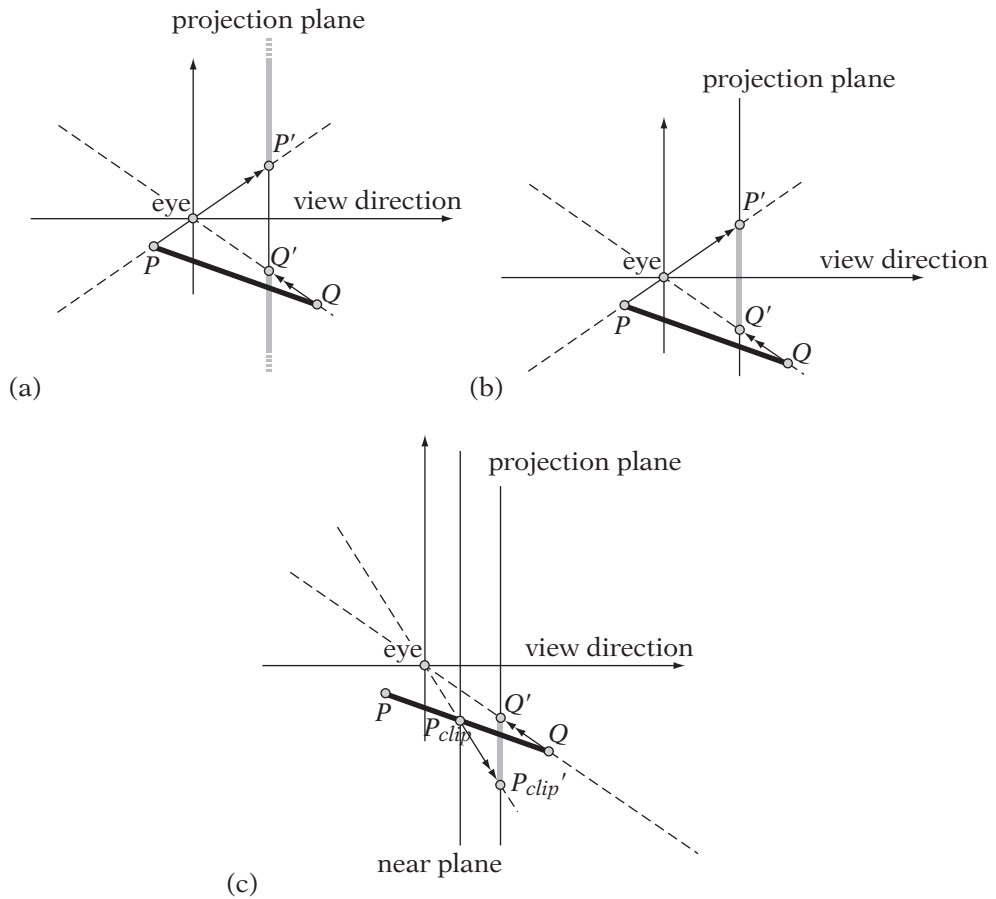


FIGURE 6.23 (a) Projection of line segment crossing behind view point. (b) Incorrect line segment rendering based on projected endpoints. (c) Line segment rendering when clipped to near plane.

line segment should start at the middle of the view, move up, and wrap around to reemerge at the bottom of the view. In practice, however, the rasterizing hardware has only the two projected vertices as input. It will take the vertices and render the shortest line segment between them (Figure 6.23(b)). If we clip the line segment to only the section that is viewable and then project the endpoints (Figure 6.23(c)), we end with only a portion of the line segment, but at least it is from the correct projection.

There is also the problem of vertices that lie on the $z = 0$ plane. When transformed to homogeneous space by the perspective matrix, a point

$(x, y, 0, 1)$ will become $(x', y', z', 0)$. The resulting transformation into NDC space will be a division by 0, which is not valid.

To avoid all of these issues, at the very least we need to set a near plane that lies in front of the eye so that the view position itself does not lie within the view frustum. We first cull any objects that lie on the same side of the near plane as the view position. We then clip any objects that cross the near plane. This avoids both the potential of dividing by 0 (although it is sometimes prudent to check for it anyway, at least in a debug build) and trying to render any line segments passing through infinity.

While clipping to a near plane is a bare minimum, clipping to the top, bottom, left, and right planes is useful as well. While the windowing hardware will usually ignore any pixels that lie outside of a window's visible region (this is commonly known as *scissoring*), it is faster if we can avoid unnecessary rasterization. Also, if we want to set a viewport that covers a subrectangle of a window, not clipping to the border of the viewport may lead to spurious geometry being drawn (although most hardware allows for adjustable scissoring regions; in particular, OpenGL and D3D provide interfaces to set this).

Finally, some hardware has a limited range for screen space positions, for example, 0 to 4095. The viewable area might lie in the center of this range, say from a minimum point of (1728, 1808) to a maximum point of (2688, 2288). The area outside of the viewable area is known as the *guard band*—anything rendered to this will be ignored, since it won't be displayed. In some cases we can avoid clipping in x and y , since we can just render objects whose screen space projection lies within the guard band and know that they will be handled automatically by the hardware. This can improve performance considerably, since clipping can be quite expensive. However, it's not entirely free. Values that lie outside the maximum range for the guard band will wrap around. So, a vertex that would normally project to coordinates that should lie off the screen, say (6096, 6096), will wrap to (2000, 2000)—right in the middle of the viewable area. Unfortunately, the only way to solve this problem is what we were trying to avoid in the first place: clipping in the x and y directions. However, now our clip window encompasses the much larger guard band area, so using the guard band can still reduce the amount of clipping that we have to do overall.

6.4.2 CULLING

A naive method of culling a model against the view frustum is to test each of its vertices against each of the frustum planes in turn. We designate the plane normal for each plane as pointing toward the inside half-space. If for one plane $ax + by + cz + d < 0$ for every vertex $P = (x, y, z)$, then the model lies outside of the frustum and we can ignore it. Conversely, if for all the

frustum planes and all the vertices $ax + by + cz + d > 0$, then we know the model lies entirely inside the frustum and we don't need to worry about clipping it.

While this will work, for models with large numbers of vertices this becomes expensive, probably outweighing any savings we might gain by not rendering the objects. Instead, culling is usually done by approximating the object with a convex bounding volume, such as a sphere, that contains all of the vertices for the object. Rather than test each vertex against the planes, we test only the bounding object. Since it is a convex object and all the vertices are contained within it, we know that if the bounding object lies outside of the view frustum, all of the model's vertices must lie outside as well. More information on computing bounding objects and testing them against planes can be found in Chapter 12.

Bounding objects are usually placed in the world frame to aid with collision detection, so culling is often done in the world frame as well. This requires storing a representation of each frustum plane in world coordinates, but the additional 24 values required is worth the speedup gained. We can find each x or y clipping plane in the view frame by using the view position and two corners of the view window to generate the plane. The two z planes (in OpenGL) are $z = -near$ and $z = -far$, respectively. Transforming them to the world frame is a simple case of using the technique for transforming plane normals, as described in Chapter 4.

While view frustum culling can remove a large number of objects from consideration, it's not the only culling method. In Chapter 7 we'll discuss backface culling, which allows us to determine which polygons are pointing away from the camera so we can ignore them. There also are a large number of culling methods that break up the scene in order to cull objects that aren't visible. This can help with interior levels, so you don't render rooms that may be within the view frustum but not visible because they're blocked by a wall. Such methods are out of the purview of this book but are described in detail in many of the references cited in the following sections.

6.4.3 GENERAL PLANE CLIPPING



To clip polygons, we first need to know how to clip a polygon edge (i.e., a line segment) to a plane. As we'll see, the problem of clipping a polygon to a plane degenerates to handling this case. Suppose we have a line segment \overline{PQ} , with endpoints P and Q , that crosses a plane. We'll say that P is inside our clip space and Q is outside. Our clipped line segment will be \overline{PR} , where R is the intersection of the line segment and the plane (Figure 6.24).

To find R , we take the line equation $P + t(Q - P)$, plug it into our plane equation $ax + by + cz + d = 0$, and solve for t . To simplify the equations, we'll define $\mathbf{v} = Q - P$. Substituting the parameterized line coordinates for

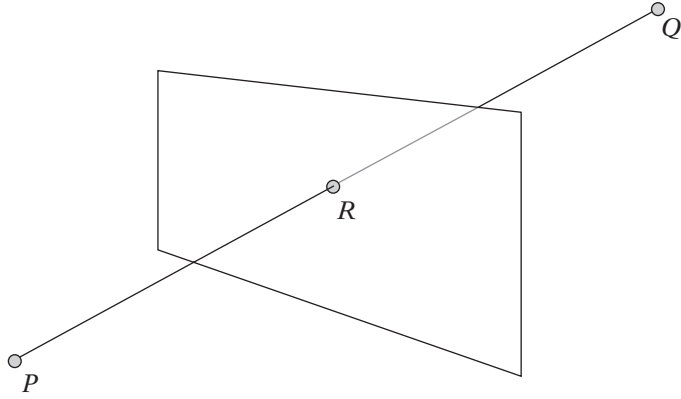


FIGURE 6.24 Clipping edge to plane.

x , y , and z , we get

$$\begin{aligned}
 0 &= a(P_x + tv_x) + b(P_y + tv_y) + c(P_z + tv_z) + d \\
 &= aP_x + tav_x + bP_y + tbv_y + cP_z + tcv_z + d \\
 &= aP_x + bP_y + cP_z + d + t(av_x + bv_y + cv_z) \\
 t &= \frac{-aP_x - bP_y - cP_z - d}{av_x + bv_y + cv_z}
 \end{aligned}$$

And now, substituting in $Q - P$ for \mathbf{v} :

$$t = \frac{(aP_x + bP_y + cP_z + d)}{(aP_x + bP_y + cP_z + d) - (aQ_x + bQ_y + cQ_z + d)}$$

We can use Blinn's notation [7], slightly modified, to simplify this to

$$t = \frac{BCP}{BCP - BCQ}$$

where BCP is the result from the plane equation (the boundary coordinate) when we test P against the plane, and BCQ is the result when we test Q against the plane. The resulting clip point R is

$$R = P + \frac{BCP}{BCP - BCQ}(Q - P)$$

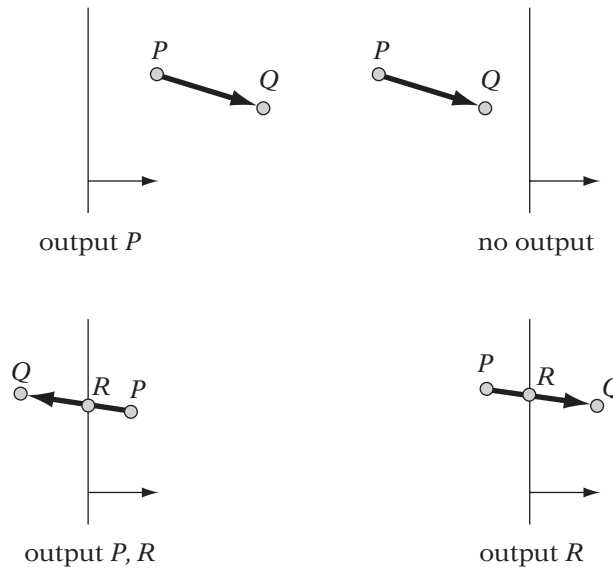


FIGURE 6.25 Four possible cases of clipping an edge against a plane.

To clip a polygon to a plane, we need to clip each edge in turn. A standard method for doing this is to use the Sutherland-Hodgeman algorithm [109]. We first test each edge against the plane. Depending on what the result is, we output particular vertices for the clipped polygon. There are four possible cases for an edge from P to Q (Figure 6.25). If both are inside, then we output P . The vertex Q will be output when we consider it as the start of the next edge. If both are outside, we output nothing. If P is inside and Q is outside, then we compute R , the clip point, and output P and R . If P is outside and Q is inside, then we compute R and output just R —as before, Q will be output as the start of the next edge. The sequence of vertices generated as output will be the vertices of our clipped polygon.

We now have enough information to build a class for clipping vertices, which we'll call `IvClipper`. We can define this as

```
class IvClipper
{
public:
    IvClipper()
    {
        mFirstVertex = true;
    }
    ~IvClipper();
```



```

        Output( end - t*(end - mStart) );
    }
    else
    {
        float t = mBCStart/(mBCStart - BCend);
        Output( mStart + t*(end - mStart) );
    }
}
}

mStart = end;
mBCStart = BCend;
mStartInside = endInside;
mFirstVertex = false;
}

```

Note that we generate t in the same direction for both clipping cases — from inside to outside. Polygons will often share edges. If we were to clip the same edge for two neighboring polygons in different directions, we may end up with two slightly different points due to floating-point error. This will lead to visible cracks in our geometry, which is not desirable. Interpolating from inside to outside for both cases avoids this situation.

To clip against the view frustum, or any other convex volume, we need to clip against each frustum plane. The output from clipping against one plane becomes the input for clipping against the next, creating a clipping pipeline. In practice, we don't store the entire clipped polygon, but pass each output vertex down as we generate it. The current output vertex and the previous one are treated as the edge to be clipped by the next plane. The `Output()` call above becomes a `ClipVertex()` for the next stage.

Note that we have only generated new positions at the clip boundary. There are other parameters that we can associate with an edge vertex, such as colors, normals, and texture coordinates (we'll discuss exactly what these are in Chapters 7–9). These will have to be clipped against the boundary as well. We use the same t value when clipping these parameters, so the clip part of our previous algorithm might become as follows.

```

// if one of them is outside, output clip vertex
if ( !(mStartInside && endInside) )
{
    ...
    clipPosition = startPosition + t*(endPosition - startPosition);
    clipColor = startColor + t*(endColor - startColor);
}

```

```

        clipTexture = startTexture + t*(endTexture - startTexture);
        // Output new clip vertex
    }

```

This is only one example of a clipping algorithm. In most cases, it won't be necessary to write any code to do clipping. The hardware will handle any clipping that needs to be done for rendering. However, for those who have the need or interest, other examples of clipping algorithms are the Liang-Barsky [68], Cohen-Sutherland (found in Foley et al. [38] as well as other graphics texts), and Cyrus-Beck [22] methods. Blinn [8] describes an algorithm for lines that combines many of the features from the previously mentioned techniques; with minor modifications it can be made to work with polygons.

6.4.4 HOMOGENEOUS CLIPPING

In the presentation above, we clip against a general plane. When projecting, however, Blinn and Newell [7] noted that we can simplify our clipping by taking advantage of some properties of our projected points prior to the division by w . Recall that after the division by w , the visible points will have normalized device coordinates lying in the interval $[-1, 1]$, or

$$-1 \leq x/w \leq 1$$

$$-1 \leq y/w \leq 1$$

$$-1 \leq z/w \leq 1$$

Multiplying these equations by w provides the intervals prior to the w division:

$$-w \leq x \leq w$$

$$-w \leq y \leq w$$

$$-w \leq z \leq w$$

In other words, the visible points are bounded by the six planes:

$$w = x$$

$$w = -x$$

$$w = y$$

$$w = -y$$

$$w = z$$

$$w = -z$$

Instead of clipping our points against general planes in the world frame or view frame, we can clip our points against these simplified planes in $\mathbb{R}P^3$ space. For example, the plane test for $w = x$ is $w - x$. The full set of plane tests for a point P are

$$BCP_{-x} = w + x$$

$$BCP_x = w - x$$

$$BCP_{-y} = w + y$$

$$BCP_y = w - y$$

$$BCP_{-z} = w + z$$

$$BCP_z = w - z$$

The previous clipping algorithm can be used, with these plane tests replacing the `IvPlane::Test()` call. While these tests are cheaper to compute in software, their great advantage is that since they don't vary with the projection, they can be built directly into hardware, making the clipping process very fast. Because of this, OpenGL clips at two separate stages in the viewing pipeline. After a point is transformed into the view frame, it is clipped against any user-defined clipping planes set by the `glClippingPlane()` call. Then the point is multiplied by the projection matrix, clipped in homogeneous space, and finally the coordinates are divided by w to place the clipped point in the NDC frame.

There is one wrinkle to homogeneous clipping, however. Figure 6.26 shows the visible region for the x coordinate in homogeneous space. However,

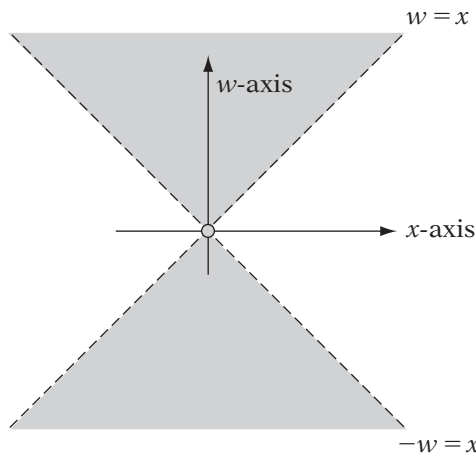


FIGURE 6.26 Homogeneous clip regions for NDC interval $[-1, 1]$.

our plane tests will clip to the upper triangle region of that hourglass shape—any points that lie in the lower region will be inadvertently removed. With the projections that we have defined, this will happen only if we use a negative value for the w value of our points. And since we’ve chosen 1 as the standard w value for points, this shouldn’t happen. However, if you do have points that for some reason have negative w values, Blinn [8] recommends the following procedure: transform, clip, and render your points normally; then multiply your projection matrix by -1 ; and then transform, clip, and render again.

6.5 SCREEN TRANSFORMATION

Now that we’ve covered viewing, projection, and clipping, our final step in transforming our object in preparation for rendering is to map its geometric data from the NDC frame to the screen or device frame. This could represent a mapping to the full display, a window within the display, or an offscreen pixel buffer.

Remember that our coordinates in the NDC frame range from a lower left corner of $(-1, -1)$ to an upper right corner of $(1, 1)$. Real device space coordinates usually range from an upper left corner $(0, 0)$ to a lower right corner (w_s, h_s) , where w_s (screen width) and h_s (screen height) are usually not the same. In addition, in screen space the y -axis is commonly flipped so that y values increase as we move down the screen. Some windowing systems allow you to use the standard y direction, but we’ll assume the default (Figure 6.27).

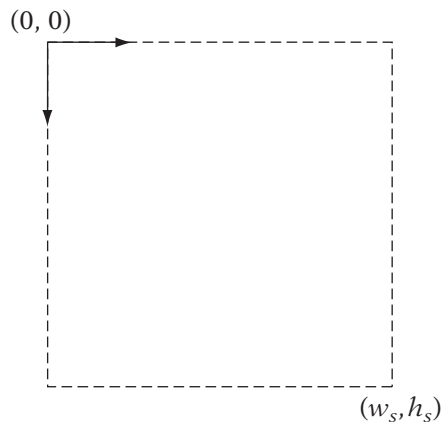


FIGURE 6.27 View window in standard screen space frame.

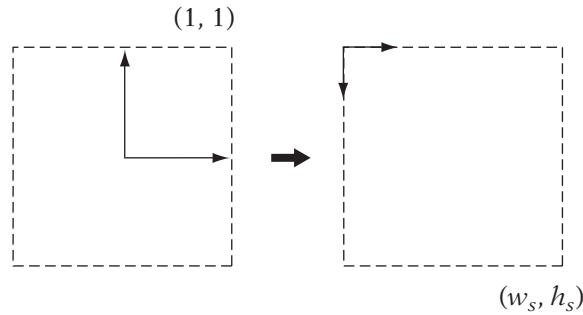


FIGURE 6.28 Mapping NDC space to screen space.

What we'll need to do is map our NDC area to our screen area (Figure 6.28). This consists of scaling it to the same size as the screen, flipping our y direction, and then translating it so that the upper left corner becomes the origin.

Let's begin by considering only the y direction, because it has the special case of the axis flip. The first step is scaling it. The NDC window is two units high, whereas the screen space window is h_s high, so we divide by 2 to scale the NDC window to unit height, and then multiply by h_s to scale to screen height:

$$y' = \frac{h_s}{2} y_{ndc}$$

Since we're still centered around the origin, we can do the axis flip by just negating:

$$y'' = -\frac{h_s}{2} y_{ndc}$$

Finally, we need to translate downwards (which is now the positive y direction) to map the top of the screen to the origin. Since we're already centered on the origin, we need to translate only half the screen height, so

$$y_s = -\frac{h_s}{2} y_{ndc} + \frac{h_s}{2}$$

Another way of thinking of the translation is that we want to map the extreme point $-h_s/2$ to 0, so we need to add $h_s/2$.

A similar process, without the axis flip, gives us our x transformation:

$$x_s = \frac{w_s}{2} x_{ndc} + \frac{w_s}{2}$$

This assumes that we want to cover the entire screen with our view window. In some cases, for example in a split-screen console game, we want to cover only a portion of the screen. Again, we'll have a width and height of our screen space area, w_s and h_s , but now we'll have a different upper left corner position for our area: (s_x, s_y) . The first part of the process is the same; we scale the NDC window to our screen space window and flip the y-axis. Now, however, we want to map $(-w_s/2, -h_s/2)$ to (s_x, s_y) , instead of $(0, 0)$. The final translation will be $(w_s/2 + s_x, h_s/2 + s_y)$. This gives us our generalized screen transformation in xy as

$$x_s = \frac{w_s}{2}x_{ndc} + \frac{w_s}{2} + s_x \quad (6.5)$$

$$y_s = -\frac{h_s}{2}y_{ndc} + \frac{h_s}{2} + s_y \quad (6.6)$$

Our z coordinate is a special case. As mentioned, we'll want to use z for depth testing, which means that we'd really prefer it to range from 0 to d_s , where d_s is usually 1. This mapping from $[-1, 1]$ to $[0, d_s]$ is

$$z_s = \frac{d_s}{2}z_{ndc} + \frac{d_s}{2} \quad (6.7)$$

We can, of course, express this as a matrix:

$$\mathbf{M}_{ndc \rightarrow screen} = \begin{bmatrix} \frac{w_s}{2} & 0 & 0 & \frac{w_s}{2} + s_x \\ 0 & -\frac{h_s}{2} & 0 & \frac{h_s}{2} + s_y \\ 0 & 0 & \frac{d_s}{2} & \frac{d_s}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.5.1 PIXEL ASPECT RATIO

Recall that in our projection matrices, we represented the shape of our view window by setting an aspect ratio a . Most of the time it is expected that the value of a chosen in the projection will match the aspect ratio w_s/h_s of the final screen transformation. Otherwise, the resulting image will be distorted. For example, if we use a square aspect ratio ($a = 1.0$) for the projection and a standard aspect ratio of 4:3 for the screen transformation, the image will appear compressed in the y direction. If your image does not quite look right, it is good practice to ensure that these two values are the same.

An exception to this practice arises when your final display has a different aspect ratio than the offscreen buffers that you're using for rendering. For

example, NTSC televisions have 448 scan lines, with 640 analog pixels per scan line, so it is common practice to render to a 640×448 area and then send that to the NTSC converter to be displayed. Using the offscreen buffer size would give an aspect ratio of 10:7. But the actual television screen has a 4:3 aspect ratio, so the resulting image will be distorted, producing stretching in the y direction. The solution is to set $a = 4/3$ despite the aspect ratio of the offscreen buffer. The image in the offscreen buffer will be compressed in the y direction, but then will be proportionally stretched in the y direction when the image is displayed on the television, thereby producing the correct result.

6.6 PICKING



Now that we understand the mathematics necessary for transforming an object from world coordinates to screen coordinates, we can consider the opposite case. In our game we may have enemy objects that we'll want to target. The interface we have chosen involves tracking them with our mouse and then clicking on the screen. The problem is: How do we take our click location and use that to detect which object we've selected (if any)? We need a method that takes our 2D screen coordinates and turns them into a form that we can use to detect object intersection in 3D game space. Effectively we are running our pipeline backwards, from the screen transformation to the projection to the viewing transformation (clipping is ignored as we're already within the boundary of our view window).

For the purposes of discussion, we'll assume that we are using the basic OpenGL perspective matrix. Similar derivations can be created using other projections. Figure 6.29 is yet another cross section showing our problem. Once again, we have our view frustum, with our top and bottom clipping planes, our projection plane, and our near and far planes. Point P_s indicates our click location on the projection plane. If we draw a ray (known as a pick ray) from the view position through P_s , we pass through every point that lies underneath our click location. So to determine which object we have clicked on, we need only generate this point on the projection plane, create the specific ray, and then test each object for intersection with the ray. The closest object to the eye will be the object we're seeking.

To generate our point on the projection plane, we'll have to find a method for going backwards from screen space into view space. To do this we'll have to find a means to "invert" our projection. Matrix inversion seems like the solution, but it is not the way to go. The standard projection matrix has zeros in the right-most column, so it's not invertible. But even using the z -depth projection matrix doesn't help us, because (a) the reciprocal divide makes the process nonlinear, and (b) in any case, our click point doesn't have a z value to plug into the inversion.

Instead, we begin by transforming our screen space point (x_s, y_s) to an NDC space point (x_{ndc}, y_{ndc}) . Since our NDC to screen space transform is affine, this is easy enough: We need only invert our previous equations 6.5 and 6.6. That gives us

$$x_{ndc} = \frac{2(x_s - s_x)}{w_s} - 1$$

$$y_{ndc} = -\frac{2(y_s - s_y)}{h_s} + 1$$

Now the tricky part. We need to transform our point in the NDC frame to the view frame. We'll begin by computing our z_v value. Looking at Figure 6.29 again, this is straightforward enough. We'll assume that our point lies on the projection plane so the z value is just the z location of the plane or $-d$. This leaves our x and y coordinates to be transformed. Again, since our view region covers a rectangle defined by the range $[-a, a]$ (recall that a is our aspect ratio) in the x direction and the range $[-1, 1]$ in the y direction, we only need to scale to get the final point. The view window in the NDC frame ranges from $[-1, 1]$ in y , so no scale is needed in the y direction and we scale by a in the x direction. Our final screen space to view space equations are

$$x_v = \frac{2a}{w_s}(x_s - s_x) - 1$$

$$y_v = -\frac{2}{h_s}(y_s - s_y) + 1$$

$$z_v = -d$$

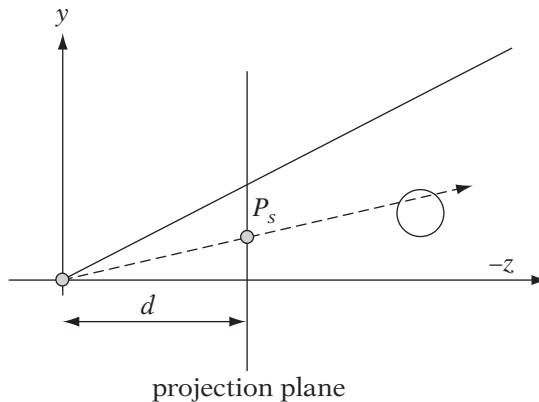


FIGURE 6.29 Pick ray.

Since this is a system of linear equations, we can express this as a 3×3 matrix:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \begin{bmatrix} \frac{2a}{w_s} & 0 & -\frac{2a}{w_s}s_x - 1 \\ 0 & -\frac{2}{h_s} & \frac{2}{h_s}s_y + 1 \\ 0 & 0 & -d \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

From here we have a choice. We can try to detect intersection with an object in the view frame, we can detect in the world frame, or we can detect in the object's local frame. The first involves transforming every object into the view frame and then testing against our pick ray. The second involves transforming our pick ray into the world frame and testing against the world coordinates of each object. For simulation and culling purposes, often we're already pregenerating our world location and bounding information. So, if we're only concerned with testing for intersection against bounding information, it can be more efficient to go with testing in world space. However, usually we test in local space so we can check for intersection within the frame of the stored model vertices. Transforming these vertices into the world frame or the view frame every time we did picking could be prohibitively expensive.

In order to test in the model's local space, we'll have to transform our view space point by the inverse of the viewing transformation. Unlike the perspective transformation, however, this inverse is much easier to compute. Recall that since the view transformation is an affine matrix, we can invert it to get the view-to-world matrix $\mathbf{M}_{view \rightarrow world}$. So, multiplying $\mathbf{M}_{view \rightarrow world}$ by our click point in the view frame gives us our point in world coordinates:

$$P_w = \mathbf{M}_{view \rightarrow world} \cdot P_v$$

We can transform this and our view position E from world coordinates into model coordinates by multiplying by the inverse of the model-to-world matrix:

$$P_l = \mathbf{M}_{world \rightarrow model} \cdot P_w$$

$$E_l = \mathbf{M}_{world \rightarrow model} \cdot E$$

Then, the formula for our pick ray in model space is

$$R(t) = E_l + t(P_l - E_l)$$

We can now use this ray in combination with our objects to find the particular one the user has clicked on. Chapter 12 discusses how to determine intersection between a ray and an object and other intersection problems.

6.7 MANAGEMENT OF VIEWING TRANSFORMATIONS



Up to this point we have presented a set of transformations and corresponding matrices without giving some sense of how they would fit into a game engine. While the thrust of this book is not about writing renderers, we can still provide a general sense of how some renderers and application programming interfaces (APIs) manage these matrices, and how to set transformations for a standard API.

The view, projection, and screen transformations change only if the camera is moved. As this happens rarely, these matrices are usually computed once, stored, and then concatenated with the new world transformation every time a new object instance is rendered. How this is handled depends on the API used. The most direct approach is to concatenate the newly set world transform matrix with the others, creating a single transformation all the way from model space to prehomogeneous divide screen space:

$$M_{model \rightarrow screen} = M_{ndc \rightarrow screen} \cdot M_{projection} \cdot M_{world \rightarrow view} \cdot M_{model \rightarrow world}$$

Multiplying by this single matrix and then performing three homogeneous divisions per vertex generates the screen coordinates for the object. This is extremely efficient, but ignores any clipping we might need to do. In this case, we can concatenate up to homogeneous space, also known as clip space:

$$M_{model \rightarrow clip} = M_{projection} \cdot M_{world \rightarrow view} \cdot M_{model \rightarrow world}$$

Then we transform our vertices by this matrix, clip against the view frustum, perform the homogeneous divide, and either calculate the screen coordinates using equations 6.5–6.7 or multiply by the NDC to screen matrix, as before.

With more complex renderers, we end up separating the transformations further. For example, OpenGL handles lighting and some clipping prior to projection, so it has separate `GL_MODELVIEW` and `GL_PROJECTION` matrix stacks, to which the appropriate matrices have to be concatenated. The vertices are transformed by the top matrix in the `GL_MODELVIEW` stack, lighting and user-defined clipping are computed, and then the vertices are transformed by the top matrix in the `GL_PROJECTION` matrix. The resulting vertices are clipped in homogeneous space, the reciprocal divide is performed as before, and finally they are transformed to screen space.

In our program, we can set the view and projection matrices in OpenGL by the following code.

```
IvMatrix44 projection, viewTransform;

// compute projection and view transformation
...
```

```
// set in OpenGL
glMatrixMode(GL_PROJECTION);
glLoadMatrix( projection );

glMatrixMode(GL_MODELVIEW);
glLoadMatrix( viewTransform );
```

And when we render an object, concatenating the world matrix can be done by the following code.

```
glMatrixMode(GL_MODELVIEW);

// push copy of view matrix to top of stack
glPushMatrix();

// multiply by world matrix
glMultMatrix( worldTransform );

// render
...

// pop to view matrix
glPopMatrix();
```

The push/pop calls provide a means for storing the view transformation without reloading it into the stack. The call `glPushMatrix()` copies the current matrix—in this case, the view matrix—to a new entry on the top of the stack. The subsequent `glMultMatrix()` will postmultiply the world matrix by the copy of the view matrix at the top of the stack. The resulting local-to-view matrix will be used to transform the vertices of our object. Finally, `glPopMatrix()` removes the current matrix from the top of the stack, restoring the view transformation as the top matrix. The effect is to save the view transformation, multiply by the world transformation and use the result to transform the vertices, and then restore the original view transformation.

Direct3D takes this one step further and manages storage of the view transformation by having three separate matrices: one each for the projective, view, and world transformations. These can be set by using the `IDirect3DDevice*::SetTransform()` method, and any concatenation is handled internally to the API.

This leaves the NDC to screen space transformation. Usually the graphics API will not require a matrix but will perform this operation directly. In the xy directions the user is only expected to provide the dimensions and position of the screen window area, also known as the viewport. In OpenGL this is set by