To go in the other direction we have

$$\begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_e \\ 0 & 1 & -y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}.$$

This is a translation followed by a rotation; they are the inverses of the rotation and translation we used to build the frame-to-canonical matrix, and when multiplied together they produce the inverse of the frame-to-canonical matrix, which is (not surprisingly) called the canonical-to-frame matrix:

$$\mathbf{p}_{uv} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xy}.$$

The canonical-to-frame matrix takes points expressed in the canonical frame and converts them to the same points expressed in the (u,v) frame. We have written this matrix as the inverse of the frame-to-canonical matrix because it can't immediately be written down using the canonical coordinates of $\mathbf{e}$, $\mathbf{u}$, and $\mathbf{v}$. But remember that all coordinate systems are equivalent; it's only our convention of storing vectors in terms of x- and y-coordinates that creates this seeming asymmetry. The canonical-to-frame matrix *can* be expressed simply in terms of the (u, v) coordinates of $\mathbf{o}$, $\mathbf{x}$, and $\mathbf{y}$:

$$\mathbf{p}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}.$$

All these ideas work strictly analogously in 3D, where we have

$$\begin{aligned} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} \\ \mathbf{p}_{xyz} &= \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uvw}, \end{aligned} \tag{6.8}$$

and

$$\begin{aligned} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} &= \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} \\ \mathbf{p}_{uvw} &= \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xyz}. \end{aligned} \tag{6.9}$$

Frequently Asked Questions

- Can't I just hardcode transforms rather than use the matrix formalisms?

Yes, but in practice it is harder to derive, harder to debug, and not any more efficient. Also, all current graphics APIs use this matrix formalism so it must be understood even to use graphics libraries.

- The bottom row of the matrix is always (0,0,0,1). Do I have to store it?

You do not have to store it unless you include perspective transforms (Chapter 7).

Notes

The derivation of the transformation properties of normals is based on *Properties of Surface Normal Transformations* (Turkowski, 1990). In many treatments through the mid-1990s, vectors were represented as row vectors and premultiplied, e.g., $\mathbf{b} = \mathbf{a}\mathbf{M}$. In our notation this would be $\mathbf{b}^{\mathrm{T}} = \mathbf{a}^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}$. If you want to find a rotation matrix $\mathbf{R}$ that takes one vector $\mathbf{a}$ to a vector $\mathbf{b}$ of the same length: $\mathbf{b} = \mathbf{R}\mathbf{a}$ you could use two rotations constructed from orthonormal bases. A more efficient method is given in *Efficiently Building a Matrix to Rotate One Vector to Another* (Akenine-Möller, Haines, & Hoffman, 2008).

Exercises

1. Write down the 4×4 3D matrix to move by (x_m, y_m, z_m).
2. Write down the 4×4 3D matrix to rotate by an angle θ about the y-axis.
3. Write down the 4×4 3D matrix to scale an object by 50% in all directions.
4. Write the 2D rotation matrix that rotates by 90 degrees clockwise.
5. Write the matrix from Exercise 4 as a product of three shear matrices.
6. Find the inverse of the rigid body transformation:

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\;0\;0 & 1 \end{bmatrix}$$

where $\mathbf{R}$ is a 3×3 rotation matrix and $\mathbf{t}$ is a 3-vector.

7. Show that the inverse of the matrix for an affine transformation (one that has all zeros in the bottom row except for a one in the lower right entry) also has the same form.

8. Describe in words what this 2D transform matrix does:

$$\begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

9. Write down the 3×3 matrix that rotates a 2D point by angle θ about a point $\mathbf{p} = (x_p, y_p)$.

10. Write down the 4×4 rotation matrix that takes the orthonormal 3D vectors $\mathbf{u} = (x_u, y_u, z_u)$, $\mathbf{v} = (x_v, y_v, z_v)$, and $\mathbf{w} = (x_w, y_w, z_w)$, to orthonormal 3D vectors $\mathbf{a} = (x_a, y_a, z_a)$, $\mathbf{b} = (x_b, y_b, z_b)$, and $\mathbf{c} = (x_c, y_c, z_c)$, So $M\mathbf{u} = \mathbf{a}$, $M\mathbf{v} = \mathbf{b}$, and $M\mathbf{w} = \mathbf{c}$.

11. What is the inverse matrix for the answer to the previous problem?

7

Viewing

In the previous chapter, we saw how to use matrix transformations as a tool for arranging geometric objects in 2D or 3D space. A second important use of geometric transformations is in moving objects between their 3D locations and their positions in a 2D view of the 3D world. This 3D to 2D mapping is called a *viewing transformation*, and it plays an important role in object-order rendering, in which we need to rapidly find the image-space location of each object in the scene.

When we studied ray tracing in Chapter 4, we covered the different types of perspective and orthographic views and how to generate viewing rays according to any given view. This chapter is about the inverse of that process. Here we explain how to use matrix transformations to express any parallel or perspective view. The transformations in this chapter project 3D points in the scene (world space) to 2D points in the image (image space), and they will project any point on a given pixel's viewing ray back to that pixel's position in image space.

If you have not looked at it recently, it is advisable to review the discussion of perspective and ray generation in Chapter 4 before reading this chapter.

By itself, the ability to project points from the world to the image is only good for producing *wireframe* renderings—renderings in which only the edges of objects are drawn, and closer surfaces do not occlude more distant surfaces (Figure 7.1). Just as a ray tracer needs to find the closest surface intersection along each viewing ray, an object-order renderer displaying solid-looking objects has to work out which of the (possibly many) surfaces drawn at any given point on the screen is closest and display only that one. In this chapter, we assume we

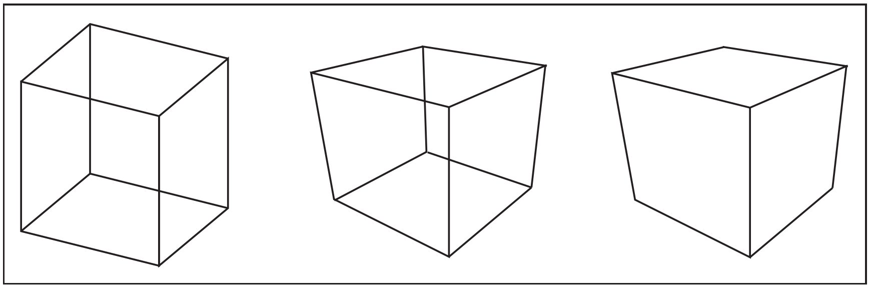


Figure 7.1. Left: wireframe cube in orthographic projection. Middle: wireframe cube in perspective projection. Right: perspective projection with hidden lines removed.

are drawing a model consisting only of 3D line segments that are specified by the (x, y, z) coordinates of their two endpoints. Later chapters will discuss the machinery needed to produce renderings of solid surfaces.

7.1 Viewing Transformations

The viewing transformation has the job of mapping 3D locations, represented as (x, y, z) coordinates in the canonical coordinate system, to coordinates in the image, expressed in units of pixels. It is a complicated beast that depends on many different things, including the camera position and orientation, the type of projection, the field of view, and the resolution of the image. As with all complicated transformations it is best approached by breaking it up into a product of several simpler transformations. Most graphics systems do this by using a sequence of three transformations:

Some APIs use "viewing transformation" for just the piece of our viewing transformation that we call the camera transformation.

- A *camera transformation* or *eye transformation*, which is a rigid body transformation that places the camera at the origin in a convenient orientation. It depends only on the position and orientation, or *pose*, of the camera.

- A *projection transformation*, which projects points from camera space so that all visible points fall in the range -1 to 1 in x and y. It depends only on the type of projection desired.

- A *viewport transformation* or *windowing transformation*, which maps this unit image rectangle to the desired rectangle in pixel coordinates. It depends only on the size and position of the output image.

To make it easy to describe the stages of the process (Figure 7.2), we give names to the coordinate systems that are the inputs and output of these transformations.

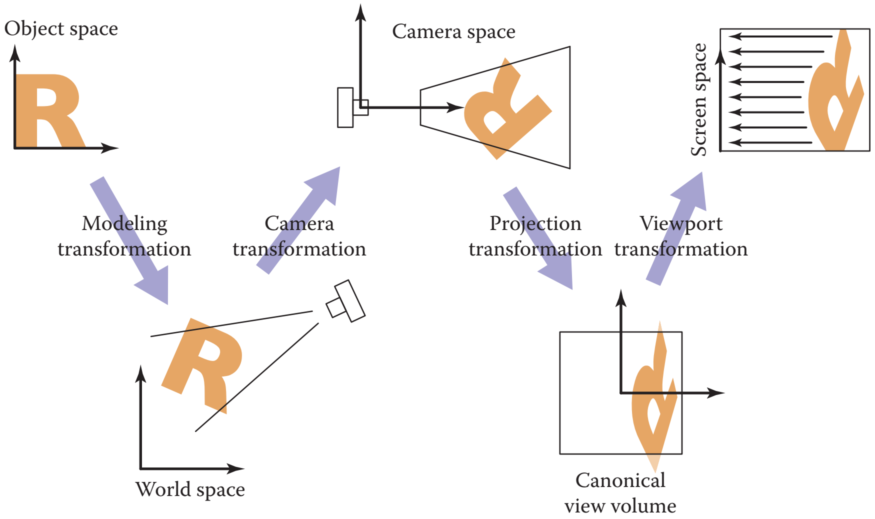


Figure 7.2. The sequence of spaces and transformations that gets objects from their original coordinates into screen space.

The camera transformation converts points in canonical coordinates (or world space) to *camera coordinates* or places them in *camera space*. The projection transformation moves points from camera space to the *canonical view volume*. Finally, the viewport transformation maps the canonical view volume to *screen space*.

Each of these transformations is individually quite simple. We'll discuss them in detail for the orthographic case beginning with the viewport transformation, then cover the changes required to support perspective projection.

Other names: camera space is also "eye space" and the camera transformation is sometimes the "viewing transformation;" the canonical view volume is also "clip space" or "normalized device coordinates;" screen space is also "pixel coordinates."

7.1.1 The Viewport Transformation

We begin with a problem whose solution will be reused for any viewing condition. We assume that the geometry we want to view is in the *canonical view volume*, and we wish to view it with an orthographic camera looking in the $-z$ direction. The canonical view volume is the cube containing all 3D points whose Cartesian coordinates are between -1 and $+1$—that is, $(x, y, z) \in [-1, 1]^3$ (Figure 7.3). We project $x = -1$ to the left side of the screen, $x = +1$ to the right side of the screen, $y = -1$ to the bottom of the screen, and $y = +1$ to the top of the screen.

The word "canonical" crops up again—it means something arbitrarily chosen for convenience. For instance, the unit circle could be called the "canonical circle."

Recall the conventions for pixel coordinates from Chapter 3: each pixel "owns" a unit square centered at integer coordinates; the image boundaries have a halfunit overshoot from the pixel centers; and the smallest pixel center coordinates are $(0, 0)$. If we are drawing into an image (or window on the screen) that has n_x by n_y pixels, we need to map the square $[-1, 1]^2$ to the rectangle $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

Mapping a square to a potentially non-square rectangle is not a problem; *x* and *y* just end up with different scale factors going from canonical to pixel coordinates.

For now, we will assume that all line segments to be drawn are completely inside the canonical view volume. Later we will relax that assumption when we discuss *clipping*.

Since the viewport transformation maps one axis-aligned rectangle to another, it is a case of the windowing transform given by Equation (6.6):

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}. \tag{7.1}$$

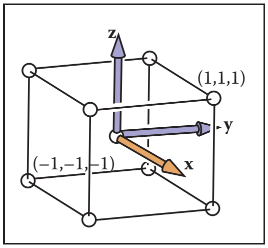


Figure 7.3. The canonical view volume is a cube with side of length two centered at the origin.

Note that this matrix ignores the z-coordinate of the points in the canonical view volume, because a point's distance along the projection direction doesn't affect where that point projects in the image. But before we officially call this the *viewport matrix*, we add a row and column to carry along the z-coordinate without changing it. We don't need it in this chapter, but eventually we will need the z values because they can be used to make closer surfaces hide more distant surfaces (see Section 8.2.3).

$$M_{\text{vp}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.2}$$

7.1.2 The Orthographic Projection Transformation

Of course, we usually want to render geometry in some region of space other than the canonical view volume. Our first step in generalizing the view will keep the view direction and orientation fixed looking along $-z$ with $+y$ up, but will allow arbitrary rectangles to be viewed. Rather than replacing the viewport matrix, we'll augment it by multiplying it with another matrix on the right.

Under these constraints, the view volume is an axis-aligned box, and we'll name the coordinates of its sides so that the view volume is $[l, r] \times [b, t] \times [f, n]$ shown in Figure 7.4. We call this box the *orthographic view volume* and refer to

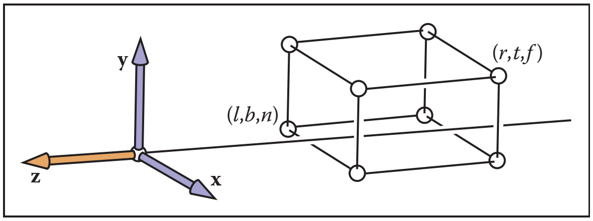


Figure 7.5. The orthographic view volume is along the negative z-axis, so f is a more negative number than n, thus $n > f$.

the bounding planes as follows:

$$
\begin{aligned}
x = l &\equiv \text{left plane},\\
x = r &\equiv \text{right plane},\\
y = b &\equiv \text{bottom plane},\\
y = t &\equiv \text{top plane},\\
z = n &\equiv \text{near plane},\\
z = f &\equiv \text{far plane}.
\end{aligned}
$$

That vocabulary assumes a viewer who is looking along the *minus* z-axis with his head pointing in the y-direction.[1] This implies that $n > f$, which may be unintuitive, but if you assume the entire orthographic view volume has negative z values then the $z = n$ "near" plane is closer to the viewer if and only if $n > f$; here f is a smaller number than n, i.e., a negative number of larger absolute value than n.

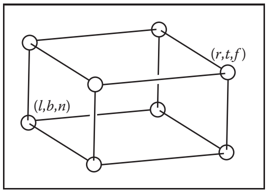


Figure 7.4. The orthographic view volume.

This concept is shown in Figure 7.5. The transform from orthographic view volume to the canonical view volume is another windowing transform, so we can simply substitute the bounds of the orthographic and canonical view volumes into Equation (6.7) to obtain the matrix for this transformation:

$$
\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.3}
$$

This matrix is very close to the one used traditionally in OpenGL, except that n, f, and $z_{\text{canonical}}$ all have the opposite sign.

[1] Most programmers find it intuitive to have the x-axis pointing right and the y-axis pointing up. In a right-handed coordinate system, this implies that we are looking in the $-z$ direction. Some systems use a left-handed coordinate system for viewing so that the gaze direction is along $+z$. Which is best is a matter of taste, and this text assumes a right-handed coordinate system. A reference that argues for the left-handed system instead is given in the notes at the end of the chapter.

To draw 3D line segments in the orthographic view volume, we project them into screen x- and y-coordinates and ignore z-coordinates. We do this by combining Equations (7.2) and (7.3). Note that in a program we multiply the matrices together to form one matrix and then manipulate points as follows:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = (\mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The z-coordinate will now be in $[-1, 1]$. We don't take advantage of this now, but it will be useful when we examine z-buffer algorithms.

This is a first example of how matrix transformation machinery makes graphics programs clean and efficient.

The code to draw many 3D lines with endpoints $\mathbf{a}_i$ and $\mathbf{b}_i$ thus becomes both simple and efficient:

```
construct $\mathbf{M}_{\text{vp}}$
construct $\mathbf{M}_{\text{orth}}$
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
   $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
   $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
   drawline$(x_p, y_p, x_q, y_q)$
```

7.1.3 The Camera Transformation

We'd like to be able to change the viewpoint in 3D and look in any direction. There are a multitude of conventions for specifying viewer position and orientation. We will use the following one (see Figure 7.6):

- the eye position $\mathbf{e}$,
- the gaze direction $\mathbf{g}$,
- the view-up vector $\mathbf{t}$.

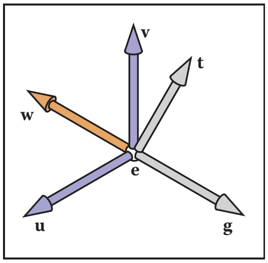


Figure 7.6. The user specifies viewing as an eye position **e**, a gaze direction **g**, and an up vector **t**. We construct a right-handed basis with **w** pointing opposite to the gaze and **v** being in the same plane as **g** and **t**.

The eye position is a location that the eye "sees from." If you think of graphics as a photographic process, it is the center of the lens. The gaze direction is any vector in the direction that the viewer is looking. The view-up vector is any vector in the plane that both bisects the viewer's head into right and left halves and points "to the sky" for a person standing on the ground. These vectors provide us with enough information to set up a coordinate system with origin $\mathbf{e}$ and a $\mathbf{uvw}$ basis,

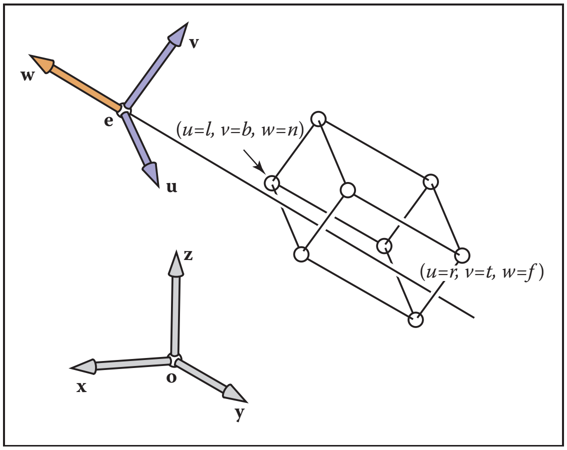


Figure 7.7. For arbitrary viewing, we need to change the points to be stored in the "appropriate" coordinate system. In this case it has origin **e** and offset coordinates in terms of **uvw**.

using the construction of Section 2.4.7:

$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|},$$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|},$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

Our job would be done if all points we wished to transform were stored in coordinates with origin $\mathbf{e}$ and basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$. But as shown in Figure 7.7, the coordinates of the model are stored in terms of the canonical (or world) origin $\mathbf{o}$ and the $\mathbf{x}$-, $\mathbf{y}$-, and $\mathbf{z}$-axes. To use the machinery we have already developed, we just need to convert the coordinates of the line segment endpoints we wish to draw from xyz-coordinates into uvw-coordinates. This kind of transformation was discussed in Section 6.5, and the matrix that enacts this transformation is the canonical-to-basis matrix of the camera's coordinate frame:

$$\mathbf{M}_{\text{cam}} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (7.4)$$

Alternatively, we can think of this same transformation as first moving $\mathbf{e}$ to the origin, then aligning $\mathbf{u}, \mathbf{v}, \mathbf{w}$ to $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

To make our previously z-axis-only viewing algorithm work for cameras with any location and orientation, we just need to add this camera transformation to the product of the viewport and projection transformations, so that it converts the incoming points from world to camera coordinates before they are projected:

```
construct $\mathbf{M}_{\text{vp}}$
construct $\mathbf{M}_{\text{orth}}$
construct $\mathbf{M}_{\text{cam}}$
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{M}_{\text{cam}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
   $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
   $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
   drawline$(x_p, y_p, x_q, y_q)$
```

Again, almost no code is needed once the matrix infrastructure is in place.

7.2 Projective Transformations

We have left perspective for last because it takes a little bit of cleverness to make it fit into the system of vectors and matrix transformations that has served us so well up to now. To see what we need to do, let's look at what the perspective projection transformation needs to do with points in camera space. Recall that the viewpoint is positioned at the origin and the camera is looking along the z-axis.

For the moment we will ignore the sign of z to keep the equations simpler, but it will return on page 150.

The key property of perspective is that the size of an object on the screen is proportional to $1/z$ for an eye at the origin looking up the negative z-axis. This can be expressed more precisely in an equation for the geometry in Figure 7.8:

$$y_s = \frac{d}{z}y, \tag{7.5}$$

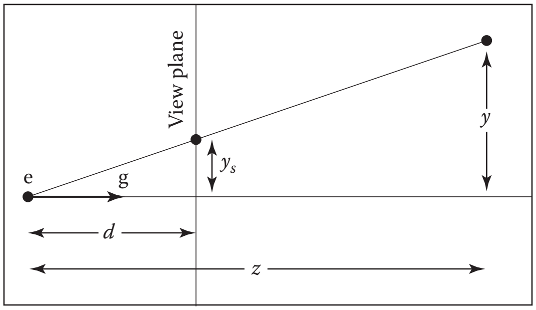


Figure 7.8. The geometry for Equation (7.5). The viewer's eye is at **e** and the gaze direction is **g** (the minus z-axis). The view plane is a distance d from the eye. A point is projected toward **e** and where it intersects the view plane is where it is drawn.

where y is the distance of the point along the y-axis, and y_s is where the point should be drawn on the screen.

We would really like to use the matrix machinery we developed for orthographic projection to draw perspective images; we could then just multiply another matrix into our composite matrix and use the algorithm we already have. However, this type of transformation, in which one of the coordinates of the input vector appears in the denominator, can't be achieved using affine transformations.

We can allow for division with a simple generalization of the mechanism of homogeneous coordinates that we have been using for affine transformations. We have agreed to represent the point (x, y, z) using the homogeneous vector $[x\ y\ z\ 1]^{\mathrm{T}}$; the extra coordinate, w, is always equal to 1, and this is ensured by always using $[0\ 0\ 0\ 1]^{\mathrm{T}}$ as the fourth row of an affine transformation matrix.

Rather than just thinking of the 1 as an extra piece bolted on to coerce matrix multiplication to implement translation, we now define it to be the denominator of the x-, y-, and z-coordinates: the homogeneous vector $[x\ y\ z\ w]^{\mathrm{T}}$ represents the point $(x/w, y/w, z/w)$. This makes no difference when $w = 1$, but it allows a broader range of transformations to be implemented if we allow any values in the bottom row of a transformation matrix, causing w to take on values other than 1.

Concretely, linear transformations allow us to compute expressions like

$$x' = ax + by + cz$$

and affine transformations extend this to

$$x' = ax + by + cz + d.$$

Treating w as the denominator further expands the possibilities, allowing us to compute functions like

$$x' = \frac{ax + by + cz + d}{ex + fy + gz + h};$$

this could be called a "linear rational function" of x, y, and z. But there is an extra constraint—the denominators are the same for all coordinates of the transformed point:

$$\begin{aligned} x' &= \frac{a_1x + b_1y + c_1z + d_1}{ex + fy + gz + h}, \\ y' &= \frac{a_2x + b_2y + c_2z + d_2}{ex + fy + gz + h}, \\ z' &= \frac{a_3x + b_3y + c_3z + d_3}{ex + fy + gz + h}. \end{aligned}$$

Expressed as a matrix transformation,

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and

$$(x', y', z') = (\tilde{x}/\tilde{w}, \tilde{y}/\tilde{w}, \tilde{z}/\tilde{w}).$$

A transformation like this is known as a *projective transformation* or a *homography*.

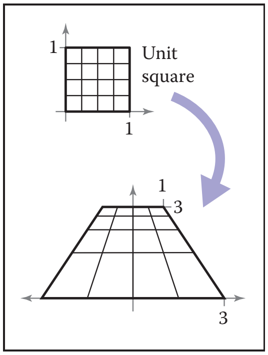


Figure 7.9. A projective transformation maps a square to a quadrilateral, preserving straight lines but not parallel lines.

Example. The matrix

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 3 & 0 \\ 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

represents a 2D projective transformation that transforms the unit square ($[0, 1] \times [0, 1]$) to the quadrilateral shown in Figure 7.9.

For instance, the lower-right corner of the square at $(1, 0)$ is represented by the homogeneous vector $[1\ 0\ 1]^{\mathrm{T}}$ and transforms as follows:

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 3 & 0 \\ 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \frac{1}{3} \end{bmatrix},$$

which represents the point $(1/\frac{1}{3}, 0/\frac{1}{3})$, or $(3, 0)$. Note that if we use the matrix

$$3\mathbf{M} = \begin{bmatrix} 6 & 0 & -3 \\ 0 & 9 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

instead, the result is $[3\ 0\ 1]^{\mathrm{T}}$, which also represents $(3, 0)$. In fact, any scalar multiple $c\mathbf{M}$ is equivalent: the numerator and denominator are both scaled by c, which does not change the result.

There is a more elegant way of expressing the same idea, which avoids treating the w-coordinate specially. In this view a 3D projective transformation is simply a 4D linear transformation, with the extra stipulation that all scalar multiples of a vector refer to the same point:

$$\mathbf{x} \sim \alpha\mathbf{x} \quad \text{for all } \alpha \neq 0.$$

The symbol $\sim$ is read as "is equivalent to" and means that the two homogeneous vectors both describe the same point in space.

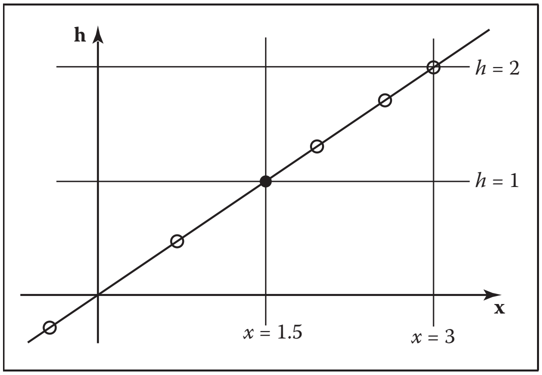


Figure 7.10. The point $x = 1.5$ is represented by any point on the line $x = 1.5h$, such as points at the hollow circles. However, before we interpret x as a conventional Cartesian coordinate, we first divide by h to get $(x,h) = (1.5,1)$ as shown by the black point.

Example. In 1D homogeneous coordinates, in which we use 2-vectors to represent points on the real line, we could represent the point (1.5) using the homogeneous vector $[1.5\ 1]^{\mathrm{T}}$, or any other point on the line $x = 1.5h$ in homogeneous space. (See Figure 7.10.)

In 2D homogeneous coordinates, in which we use 3-vectors to represent points in the plane, we could represent the point $(-1, -0.5)$ using the homogeneous vector $[-2; -1; 2]^{\mathrm{T}}$, or any other point on the line $\mathbf{x} = \alpha[-1\ \ -0.5\ 1]^{\mathrm{T}}$. Any homogeneous vector on the line can be mapped to the line's intersection with the plane $w = 1$ to obtain its Cartesian coordinates. (See Figure 7.11.)

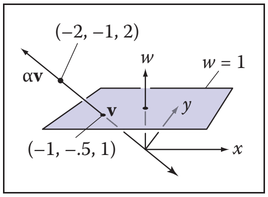


Figure 7.11. A point in homogeneous coordinates is equivalent to any other point on the line through it and the origin, and normalizing the point amounts to intersecting this line with the plane $w = 1$.

It's fine to transform homogeneous vectors as many times as needed, without worrying about the value of the w-coordinate—in fact, it is fine if the w-coordinate is zero at some intermediate phase. It is only when we want the ordinary Cartesian coordinates of a point that we need to normalize to an equivalent point that has $w = 1$, which amounts to dividing all the coordinates by w. Once we've done this we are allowed to read off the (x, y, z)-coordinates from the first three components of the homogeneous vector.

7.3 Perspective Projection

The mechanism of projective transformations makes it simple to implement the division by z required to implement perspective. In the 2D example shown in Figure 7.8, we can implement the perspective projection with a matrix transformation as follows:

$$\begin{bmatrix} y_s \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}.$$

This transforms the 2D homogeneous vector $[y; z; 1]^{\mathrm{T}}$ to the 1D homogeneous vector $[dy\ z]^{\mathrm{T}}$, which represents the 1D point (dy/z) (because it is equivalent to the 1D homogeneous vector $[dy/z\ 1]^{\mathrm{T}}$. This matches Equation (7.5).

For the "official" perspective projection matrix in 3D, we'll adopt our usual convention of a camera at the origin facing in the $-z$ direction, so the distance of the point (x, y, z) is $-z$. As with orthographic projection, we also adopt the notion of near and far planes that limit the range of distances to be seen. In this context, we will use the near plane as the projection plane, so the image plane distance is $-n$.

Remember, $n < 0$.

The desired mapping is then $y_s = (n/z)y$, and similarly for x. This transformation can be implemented by the *perspective matrix*:

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The first, second, and fourth rows simply implement the perspective equation. The third row, as in the orthographic and viewport matrices, is designed to bring the z-coordinate "along for the ride" so that we can use it later for hidden surface removal. In the perspective projection, though, the addition of a non-constant denominator prevents us from actually preserving the value of z—it's actually impossible to keep z from changing while getting x and y to do what we need them to do. Instead we've opted to keep z unchanged for points on the near or far planes.

More on this later.

There are many matrices that could function as perspective matrices, and all of them nonlinearly distort the z-coordinate. This specific matrix has the nice properties shown in Figures 7.12 and 7.13; it leaves points on the $(z = n)$-plane entirely alone, and it leaves points on the $(z = f)$-plane while "squishing" them in x and y by the appropriate amount. The effect of the matrix on a point (x, y, z) is

$$\mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \sim \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}.$$

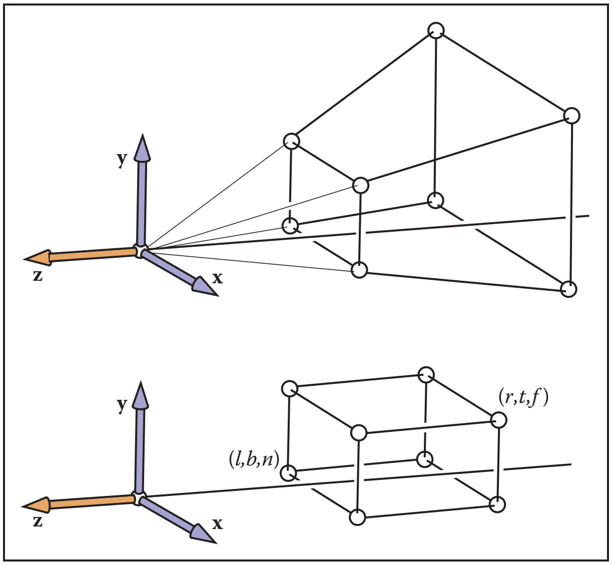


Figure 7.12. The perspective projection leaves points on the $z = n$ plane unchanged and maps the large $z = f$ rectangle at the back of the perspective volume to the small $z = f$ rectangle at the back of the orthographic volume.

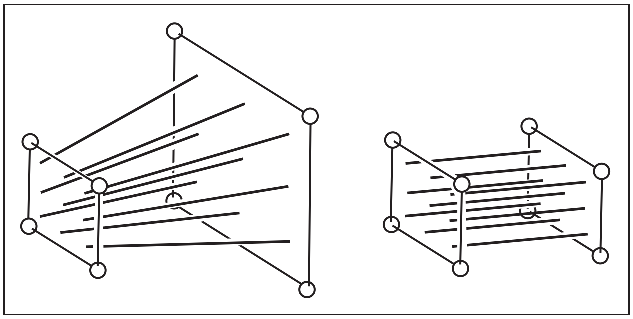


Figure 7.13. The perspective projection maps any line through the origin/eye to a line parallel to the z-axis and without moving the point on the line at $z = n$.

As you can see, x and y are scaled and, more importantly, divided by z. Because both n and z (inside the view volume) are negative, there are no "flips" in x and y. Although it is not obvious (see the exercise at the end of the chapter), the transform also preserves the relative order of z values between $z = n$ and $z = f$, allowing us to do depth ordering after this matrix is applied. This will be important later when we do hidden surface elimination.

Sometimes we will want to take the inverse of $\mathbf{P}$, for example, to bring a screen coordinate plus z back to the original space, as we might want to do for picking. The inverse is

$$\mathbf{P}^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}.$$

Since multiplying a homogeneous vector by a scalar does not change its meaning, the same is true of matrices that operate on homogeneous vectors. So we can write the inverse matrix in a prettier form by multiplying through by nf:

This matrix is not literally the inverse of the matrix **P**, but the transformation it describes *is* the inverse of the transformation described by **P**.

$$\mathbf{P}^{-1} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & fn \\ 0 & 0 & -1 & n+f \end{bmatrix}.$$

Taken in the context of the orthographic projection matrix $\mathbf{M}_{\text{orth}}$ in Equation (7.3), the perspective matrix simply maps the perspective view volume (which is shaped like a slice, or *frustum*, of a pyramid) to the orthographic view volume (which is an axis-aligned box). The beauty of the perspective matrix is that once we apply it, we can use an orthographic transform to get to the canonical view volume. Thus, all of the orthographic machinery applies, and all that we have added is one matrix and the division by w. It is also heartening that we are not "wasting" the bottom row of our four by four matrices!

Concatenating $\mathbf{P}$ with $\mathbf{M}_{\text{orth}}$ results in the *perspective projection matrix*,

$$\mathbf{M}_{\text{per}} = \mathbf{M}_{\text{orth}}\mathbf{P}.$$

One issue, however, is: How are *l,r,b,t* determined for perspective? They identify the "window" through which we look. Since the perspective matrix does not change the values of x and y on the $(z = n)$-plane, we can specify (l, r, b, t) on that plane.

To integrate the perspective matrix into our orthographic infrastructure, we simply replace $\mathbf{M}_{\text{orth}}$ with $\mathbf{M}_{\text{per}}$, which inserts the perspective matrix $\mathbf{P}$ after the camera matrix $\mathbf{M}_{\text{cam}}$ has been applied but before the orthographic projection. So the full set of matrices for perspective viewing is

$$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}.$$

The resulting algorithm is:

```
compute $\mathbf{M}_{\text{vp}}$
compute $\mathbf{M}_{\text{per}}$
compute $\mathbf{M}_{\text{cam}}$
```

```
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{per}}\mathbf{M}_{\text{cam}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
  $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
  $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
  drawline$(x_p/w_p, y_p/w_p, x_q/w_q, y_q/w_q)$
```

Note that the only change other than the additional matrix is the divide by the homogeneous coordinate w.

Multiplied out, the matrix $\mathbf{M}_{\text{per}}$ looks like this:

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

This or similar matrices often appear in documentation, and they are less mysterious when one realizes that they are usually the product of a few simple matrices.

Example. Many APIs such as *OpenGL* (Shreiner, Neider, Woo, & Davis, 2004) use the same canonical view volume as presented here. They also usually have the user specify the absolute values of n and f. The projection matrix for *OpenGL* is

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

Other APIs send n and f to 0 and 1, respectively. Blinn (J. Blinn, 1996) recommends making the canonical view volume $[0,1]^3$ for efficiency. All such decisions will change the the projection matrix slightly.

7.4 Some Properties of the Perspective Transform

An important property of the perspective transform is that it takes lines to lines and planes to planes. In addition, it takes line segments in the view volume to line segments in the canonical volume. To see this, consider the line segment

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}).$$

When transformed by a 4×4 matrix $\mathbf{M}$, it is a point with possibly varying homogeneous coordinate:

$$\mathbf{Mq} + t(\mathbf{MQ} - \mathbf{Mq}) \equiv \mathbf{r} + t(\mathbf{R} - \mathbf{r}).$$

The homogenized 3D line segment is

$$\frac{\mathbf{r} + t(\mathbf{R} - \mathbf{r})}{w_r + t(w_R - w_r)}. \tag{7.6}$$

If Equation (7.6) can be rewritten in a form

$$\frac{\mathbf{r}}{w_r} + f(t)\left(\frac{\mathbf{R}}{w_R} - \frac{\mathbf{r}}{w_r}\right), \tag{7.7}$$

then all the homogenized points lie on a 3D line. Brute force manipulation of Equation (7.6) yields such a form with

$$f(t) = \frac{w_R t}{w_r + t(w_R - w_r)}. \tag{7.8}$$

It also turns out that the line segments do map to line segments preserving the ordering of the points (Exercise 8), i.e., they do not get reordered or "torn."

A byproduct of the transform taking line segments to line segments is that it takes the edges and vertices of a triangle to the edges and vertices of another triangle. Thus, it takes triangles to triangles and planes to planes.

7.5 Field-of-View

While we can specify any window using the (l, r, b, t) and n values, sometimes we would like to have a simpler system where we look through the center of the window. This implies the constraint that

$$\begin{aligned} l &= -r, \\ b &= -t. \end{aligned}$$

If we also add the constraint that the pixels are square, i.e., there is no distortion of shape in the image, then the ratio of r to t must be the same as the ratio of the number of horizontal pixels to the number of vertical pixels:

$$\frac{n_x}{n_y} = \frac{r}{t}.$$

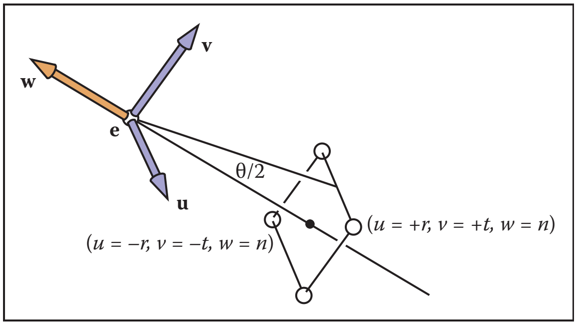


Figure 7.14. The field-of-view θ is the angle from the bottom of the screen to the top of the screen as measured from the eye.

Once n_x and n_y are specified, this leaves only one degree of freedom. That is often set using the *field-of-view* shown as θ in Figure 7.14. This is sometimes called the vertical field-of-view to distinguish it from the angle between left and right sides or from the angle between diagonal corners. From the figure we can see that

$$\tan\frac{\theta}{2} = \frac{t}{|n|}.$$

If n and θ are specified, then we can derive t and use code for the more general viewing system. In some systems, the value of n is hard-coded to some reasonable value, and thus we have one fewer degree of freedom.

Frequently Asked Questions

• Is orthographic projection ever useful in practice?

It is useful in applications where relative length judgments are important. It can also yield simplifications where perspective would be too expensive as occurs in some medical visualization applications.

• The tessellated spheres I draw in perspective look like ovals. Is this a bug?

No. It is correct behavior. If you place your eye in the same relative position to the screen as the virtual viewer has with respect to the viewport, then these ovals will look like circles because they themselves are viewed at an angle.

• Does the perspective matrix take negative z values to positive z values with a reversed ordering? Doesn't that cause trouble?

Yes. The equation for transformed z is

$$z' = n + f - \frac{fn}{z}.$$

So $z = +\epsilon$ is transformed to $z' = -\infty$ and $z = -\epsilon$ is transformed to $z = \infty$. So any line segments that span $z = 0$ will be "torn" although all points will be projected to an appropriate screen location. This tearing is not relevant when all objects are contained in the viewing volume. This is usually assured by *clipping* to the view volume. However, clipping itself is made more complicated by the tearing phenomenon as is discussed in Chapter 8.

• The perspective matrix changes the value of the homogeneous coordinate. Doesn't that make the move and scale transformations no longer work properly?

Applying a translation to a homogeneous point we have

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} hx \\ hy \\ hz \\ h \end{bmatrix} = \begin{bmatrix} hx + ht_x \\ hy + ht_y \\ hz + ht_z \\ h \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}.$$

Similar effects are true for other transforms (see Exercise 5).

Notes

Most of the discussion of viewing matrices is based on information in *Real-Time Rendering* (Akenine-Möller et al., 2008), the *OpenGL Programming Guide* (Shreiner et al., 2004), *Computer Graphics* (Hearn & Baker, 1986), and *3D Game Engine Design* (Eberly, 2000).

Exercises

1. Construct the viewport matrix required for a system in which pixel coordinates count down from the top of the image, rather than up from the bottom.

2. Multiply the viewport and orthographic projection matrices, and show that the result can also be obtained by a single application of Equation (6.7).

3. Derive the third row of Equation (7.3) from the constraint that z is preserved for points on the near and far planes.

4. Show algebraically that the perspective matrix preserves order of z values within the view volume.

5. For a 4×4 matrix whose top three rows are arbitrary and whose bottom row is $(0, 0, 0, 1)$, show that the points $(x, y, z, 1)$ and (hx, hy, hz, h) transform to the same point after homogenization.

6. Verify that the form of $\mathbf{M}_p^{-1}$ given in the text is correct.

7. Verify that the full perspective to canonical matrix $\mathbf{M}_{\text{projection}}$ takes (r, t, n) to $(1, 1, 1)$.

8. Write down a perspective matrix for $n = 1$, $f = 2$.

9. For the point $\mathbf{p} = (x, y, z, 1)$, what are the homogenized and unhomogenized results for that point transformed by the perspective matrix in Exercise 6?

10. For the eye position $\mathbf{e} = (0, 1, 0)$, a gaze vector $\mathbf{g} = (0, -1, 0)$, and a view-up vector $\mathbf{t} = (1, 1, 0)$, what is the resulting orthonormal $\mathbf{uvw}$ basis used for coordinate rotations?

11. Show, that for a perspective transform, line segments that start in the view volume do map to line segments in the canonical volume after homogenization. Further, show that the relative ordering of points on the two segments is the same. *Hint*: Show that the $f(t)$ in Equation (7.8) has the properties $f(0) = 0$, $f(1) = 1$, the derivative of f is positive for all $t \in [0, 1]$, and the homogeneous coordinate does not change sign.

8

The Graphics Pipeline

The previous several chapters have established the mathematical scaffolding we need to look at the second major approach to rendering: drawing objects one by one onto the screen, or *object-order rendering*. Unlike in ray tracing, where we consider each pixel in turn and find the objects that influence its color, we'll now instead consider each geometric object in turn and find the pixels that it could have an effect on. The process of finding all the pixels in an image that are occupied by a geometric primitive is called *rasterization*, so object-order rendering can also be called rendering by rasterization. The sequence of operations that is required, starting with objects and ending by updating pixels in the image, is known as the *graphics pipeline*.

Any graphics system has one or more types of "primitive object" that it can handle directly, and more complex objects are converted into these "primitives." Triangles are the most often used primitive.

Rasterization-based systems are also called *scanline renderers*.

Object-order rendering has enjoyed great success because of its efficiency. For large scenes, management of data access patterns is crucial to performance, and making a single pass over the scene visiting each bit of geometry once has significant advantages over repeatedly searching the scene to retrieve the objects required to shade each pixel.

The title of this chapter suggests that there is only one way to do object-order rendering. Of course this isn't true—two quite different examples of graphics pipelines with very different goals are the hardware pipelines used to support interactive rendering via APIs like OpenGL and Direct3D and the software pipelines used in film production, supporting APIs like RenderMan. Hardware pipelines must run fast enough to react in real time for games, visualizations, and user interfaces. Production pipelines must render the highest quality animation and visual effects possible and scale to enormous scenes, but may take much more time to do so. Despite the different design decisions resulting from these divergent goals, a remarkable amount is shared among most, if not all, pipelines, and this chapter attempts to focus on these common fundamentals, erring on the side of following the hardware pipelines more closely.

The work that needs to be done in object-order rendering can be organized into the task of rasterization itself, the operations that are done to geometry before rasterization, and the operations that are done to pixels after rasterization. The most common geometric operation is applying matrix transformations, as discussed in the previous two chapters, to map the points that define the geometry from object space to screen space, so that the input to the rasterizer is expressed in pixel coordinates, or *screen space*. The most common pixelwise operation is *hidden surface removal* which arranges for surfaces closer to the viewer to appear in front of surfaces farther from the viewer. Many other operations also can be included at each stage, thereby achieving a wide range of different rendering effects using the same general process.

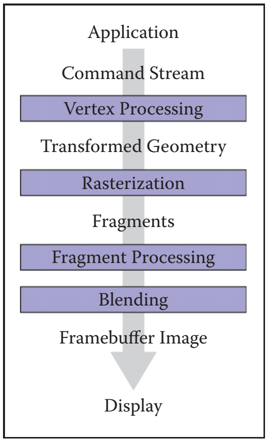


Figure 8.1. The stages of a graphics pipeline.

For the purposes of this chapter, we'll discuss the graphics pipeline in terms of four stages (Figure 8.1). Geometric objects are fed into the pipeline from an interactive application or from a scene description file, and they are always described by sets of vertices. The vertices are operated on in the *vertex-processing stage*, then the primitives using those vertices are sent to the *rasterization stage*. The rasterizer breaks each primitive into a number of *fragments*, one for each pixel covered by the primitive. The fragments are processed in the *fragment processing stage*, and then the various fragments corresponding to each pixel are combined in the *fragment blending stage*.

We'll begin by discussing rasterization, then illustrate the purpose of the geometric and pixel-wise stages by a series of examples.

8.1 Rasterization

Rasterization is the central operation in object-order graphics, and the *rasterizer* is central to any graphics pipeline. For each primitive that comes in, the rasterizer has two jobs: it *enumerates* the pixels that are covered by the primitive and it *interpolates* values, called attributes, across the primitive—the purpose for these attributes will be clear with later examples. The output of the rasterizer is a set of *fragments*, one for each pixel covered by the primitive. Each fragment "lives" at a particular pixel and carries its own set of attribute values.

In this chapter, we will present rasterization with a view toward using it to render three-dimensional scenes. The same rasterization methods are used to draw lines and shapes in 2D as well—although it is becoming more and more common to use the 3D graphics system "under the covers" to do all 2D drawing.

8.1.1 Line Drawing

Most graphics packages contain a line drawing command that takes two endpoints in screen coodinates (see Figure 3.10) and draws a line between them. For example, the call for endpoints (1,1) and (3,2) would turn on pixels (1,1) and (3,2) and fill in one pixel between them. For general screen coordinate endpoints (x_0, y_0) and (x_1, y_1), the routine should draw some "reasonable" set of pixels that approximates a line between them. Drawing such lines is based on line equations, and we have two types of equations to choose from: implicit and parametric. This section describes the approach using implicit lines.

Even though we often use integer-valued endpoints for examples, it's important to properly support arbitrary endpoints.

Line Drawing Using Implicit Line Equations

The most common way to draw lines using implicit equations is the *midpoint* algorithm (Pitteway (1967); van Aken and Novak (1985)). The midpoint algorithm ends up drawing the same lines as the *Bresenham algorithm* (Bresenham, 1965) but it is somewhat more straightforward.

The first thing to do is find the implicit equation for the line as discussed in Section 2.5.2:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0. \qquad (8.1)$$

We assume that $x_0 \leq x_1$. If that is not true, we swap the points so that it is true. The slope m of the line is given by

$$m = \frac{y_1 - y_0}{x_1 - x_0}.$$

The following discussion assumes $m \in (0, 1]$. Analogous discussions can be derived for $m \in (-\infty, -1]$, $m \in (-1, 0]$, and $m \in (1, \infty)$. The four cases cover all possibilities.

For the case $m \in (0, 1]$, there is more "run" than "rise," i.e., the line is moving faster in x than in y. If we have an API where the y-axis points downward, we might have a concern about whether this makes the process harder, but, in fact, we can ignore that detail. We can ignore the geometric notions of "up" and "down," because the algebra is exactly the same for the two cases. Cautious readers can confirm that the resulting algorithm works for the y-axis downward case. The key assumption of the midpoint algorithm is that we draw the thinnest line possible

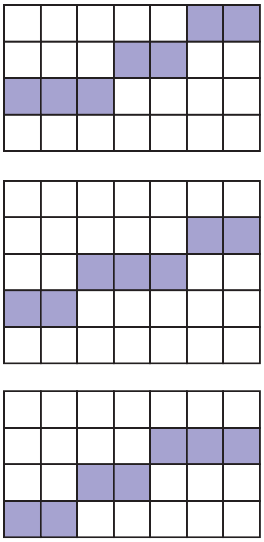


Figure 8.2. Three "reasonable" lines that go seven pixels horizontally and three pixels vertically.

that has no gaps. A diagonal connection between two pixels is not considered a gap.

As the line progresses from the left endpoint to the right, there are only two possibilities: draw a pixel at the same height as the pixel drawn to its left, or draw a pixel one higher. There will always be exactly one pixel in each column of pixels between the endpoints. Zero would imply a gap, and two would be too thick a line. There may be two pixels in the same row for the case we are considering; the line is more horizontal than vertical so sometimes it will go right, and sometimes up. This concept is shown in Figure 8.2, where three "reasonable" lines are shown, each advancing more in the horizontal direction than in the vertical direction.

The midpoint algorithm for $m \in (0, 1]$ first establishes the leftmost pixel and the column number (x-value) of the rightmost pixel and then loops horizontally establishing the row (y-value) of each pixel. The basic form of the algorithm is:

```
$y = y_0$
for $x = x_0$ to $x_1$ do
   draw$(x, y)$
   if (some condition) then
      $y = y + 1$
```

Note that x and y are integers. In words this says, "keep drawing pixels from left to right and sometimes move upward in the y-direction while doing so." The key is to establish efficient ways to make the decision in the *if* statement.

An effective way to make the choice is to look at the *midpoint* of the line between the two potential pixel centers. More specifically, the pixel just drawn is pixel (x, y) whose center in real screen coordinates is at (x, y). The candidate pixels to be drawn to the right are pixels $(x + 1, y)$ and $(x + 1, y + 1)$. The midpoint between the centers of the two candidate pixels is $(x + 1, y + 0.5)$. If the line passes below this midpoint we draw the bottom pixel, and otherwise we draw the top pixel (Figure 8.3).

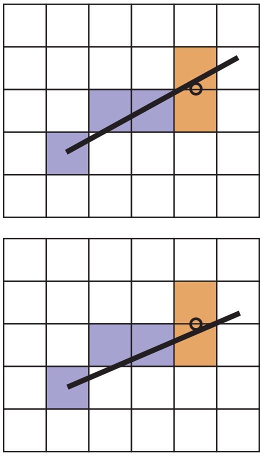


Figure 8.3. Top: the line goes above the midpoint so the top pixel is drawn. Bottom: the line goes below the midpoint so the bottom pixel is drawn.

To decide whether the line passes above or below $(x+1, y+0.5)$, we evaluate $f(x, y + 0.5)$ in Equation (8.1). Recall from Section 2.5.1 that $f(x, y) = 0$ for points (x, y) on the line, $f(x, y) > 0$ for points on one side of the line, and $f(x, y) < 0$ for points on the other side of the line. Because $-f(x, y) = 0$ and $f(x, y) = 0$ are both perfectly good equations for the line, it is not immediately clear whether $f(x, y)$ being positive indicates that (x, y) is above the line, or whether it is below. However, we can figure it out; the key term in Equation (8.1) is the y term $(x_1 - x_0)y$. Note that $(x_1 - x_0)$ is definitely positive because $x_1 > x_0$. This means that as y increases, the term $(x_1 - x_0)y$ gets larger (i.e., more positive or less negative). Thus, the case $f(x, +\infty)$ is definitely positive, and definitely above the line, implying points above the line are all positive. Another way to look at it is that the y component of the gradient vector is positive. So above the line, where y can increase arbitrarily, $f(x, y)$ must be positive. This means we can make our code more specific by filling in the *if* statement:

```
**if** $f(x+1, y+0.5) < 0$ **then**
  $y = y + 1$
```

The above code will work nicely for lines of the appropriate slope (i.e., between zero and one). The reader can work out the other three cases which differ only in small details.

If greater efficiency is desired, using an *incremental* method can help. An incremental method tries to make a loop more efficient by reusing computation from the previous step. In the midpoint algorithm as presented, the main computation is the evaluation of $f(x+1, y+0.5)$. Note that inside the loop, after the first iteration, either we already evaluated $f(x-1, y+0.5)$ or $f(x-1, y-0.5)$ (Figure 8.4). Note also this relationship:

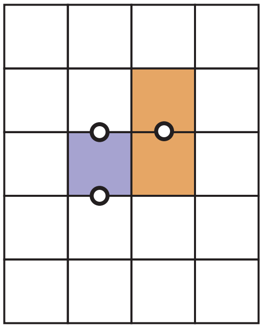


Figure 8.4. When using the decision point shown between the two orange pixels, we just drew the blue pixel, so we evaluated f at one of the two left points shown.

$$
\begin{aligned}
f(x+1, y) &= f(x, y) + (y_0 - y_1) \\
f(x+1, y+1) &= f(x, y) + (y_0 - y_1) + (x_1 - x_0).
\end{aligned}
$$

This allows us to write an incremental version of the code:

```
$y = y_0$
$d = f(x_0 + 1, y_0 + 0.5)$
**for** $x = x_0$ to $x_1$ **do**
  draw$(x, y)$
  **if** $d < 0$ **then**
    $y = y + 1$
    $d = d + (x_1 - x_0) + (y_0 - y_1)$
  **else**
    $d = d + (y_0 - y_1)$
```

This code should run faster since it has little extra setup cost compared to the non-incremental version (that is not always true for incremental algorithms), but it may accumulate more numeric error because the evaluation of $f(x, y + 0.5)$ may be composed of many adds for long lines. However, given that lines are rarely longer than a few thousand pixels, such an error is unlikely to be critical. Slightly longer setup cost, but faster loop execution, can be achieved by storing $(x_1 - x_0) + (y_0 - y_1)$ and $(y_0 - y_1)$ as variables. We might hope a good compiler would do that for us, but if the code is critical, it would be wise to examine the results of compilation to make sure.

8.1.2 Triangle Rasterization

We often want to draw a 2D triangle with 2D points $\mathbf{p}_0 = (x_0, y_0)$, $\mathbf{p}_1 = (x_1, y_1)$, and $\mathbf{p}_2 = (x_2, y_2)$ in screen coordinates. This is similar to the line drawing problem, but it has some of its own subtleties. As with line drawing, we may wish to interpolate color or other properties from values at the vertices. This is straightforward if we have the barycentric coordinates (Section 2.7). For example, if the vertices have colors $\mathbf{c}_0$, $\mathbf{c}_1$, and $\mathbf{c}_2$, the color at a point in the triangle with barycentric coordinates (α, β, γ) is

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2.$$

This type of interpolation of color is known in graphics as *Gouraud* interpolation after its inventor (Gouraud, 1971).

Another subtlety of rasterizing triangles is that we are usually rasterizing triangles that share vertices and edges. This means we would like to rasterize adjacent triangles so there are no holes. We could do this by using the midpoint algorithm to draw the outline of each triangle and then fill in the interior pixels. This would mean adjacent triangles both draw the same pixels along each edge. If the adjacent triangles have different colors, the image will depend on the order in which the two triangles are drawn. The most common way to rasterize triangles that avoids the order problem and eliminates holes is to use the convention that pixels are drawn if and only if their centers are inside the triangle, i.e., the barycentric coordinates of the pixel center are all in the interval $(0, 1)$. This raises the issue of what to do if the center is exactly on the edge of the triangle. There are several ways to handle this as will be discussed later in this section. The key observation is that barycentric coordinates allow us to decide whether to draw a pixel and what color that pixel should be if we are interpolating colors from the vertices. So our problem of rasterizing the triangle boils down to efficiently finding the barycentric coordinates of pixel centers (Pineda, 1988). The brute-force rasterization algorithm is:

```
for all $x$ do
   for all $y$ do
      compute $(\alpha, \beta, \gamma)$ for $(x, y)$
      if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
         $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
         drawpixel $(x, y)$ with color $\mathbf{c}$
```

The rest of the algorithm limits the outer loops to a smaller set of candidate pixels and makes the barycentric computation efficient.

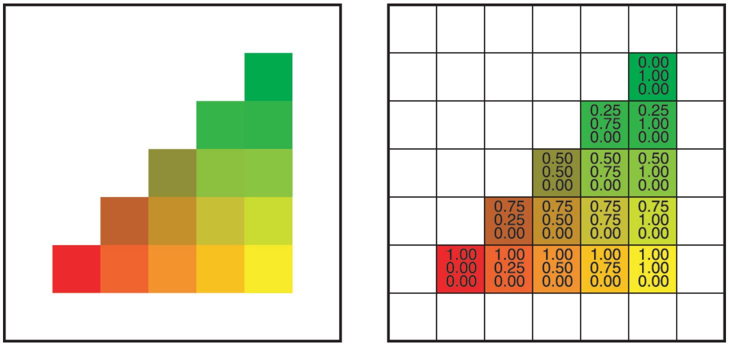
We can add a simple efficiency by finding the bounding rectangle of the three vertices and only looping over this rectangle for candidate pixels to draw. We can compute barycentric coordinates using Equation (2.32). This yields the algorithm:

```
$x_{\text{min}} = \text{floor}(x_i)$
$x_{\text{max}} = \text{ceiling}(x_i)$
$y_{\text{min}} = \text{floor}(y_i)$
$y_{\text{max}} = \text{ceiling}(y_i)$
for $y = y_{\text{min}}$ to $y_{\text{max}}$ do
  for $x = x_{\text{min}}$ to $x_{\text{max}}$ do
    $\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$
    $\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$
    $\gamma = f_{01}(x, y)/f_{01}(x_2, y_2)$
    if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) then
      $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
      drawpixel $(x, y)$ with color $\mathbf{c}$
```

Here f_{ij} is the line given by Equation (8.1) with the appropriate vertices:

$$
\begin{aligned}
f_{01}(x, y) &= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0,\\
f_{12}(x, y) &= (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1,\\
f_{20}(x, y) &= (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2.
\end{aligned}
$$

Note that we have exchanged the test $\alpha \in (0, 1)$ with $\alpha > 0$ etc., because if all of α, β, γ are positive, then we know they are all less than one because $\alpha + \beta + \gamma = 1$. We could also compute only two of the three barycentric variables



					0.00 1.00 0.00	
				0.25 0.75 0.00	0.25 1.00 0.00	
			0.50 0.50 0.00	0.50 0.75 0.00	0.50 1.00 0.00	
		0.75 0.25 0.00	0.75 0.50 0.00	0.75 0.75 0.00	0.75 1.00 0.00	
	1.00 0.00 0.00	1.00 0.25 0.00	1.00 0.50 0.00	1.00 0.75 0.00	1.00 1.00 0.00	

Figure 8.5. A colored triangle with barycentric interpolation. Note that the changes in color components are linear in each row and column as well as along each edge. In fact it is constant along every line, such as the diagonals, as well.

and get the third from that relation, but it is not clear that this saves computation once the algorithm is made incremental, which is possible as in the line drawing algorithms; each of the computations of α, β, and γ does an evaluation of the form $f(x, y) = Ax + By + C$. In the inner loop, only x changes, and it changes by one. Note that $f(x + 1, y) = f(x, y) + A$. This is the basis of the incremental algorithm. In the outer loop, the evaluation changes for $f(x, y)$ to $f(x, y + 1)$, so a similar efficiency can be achieved. Because α, β, and γ change by constant increments in the loop, so does the color **c**. So this can be made incremental as well. For example, the red value for pixel $(x + 1, y)$ differs from the red value for pixel (x, y) by a constant amount that can be precomputed. An example of a triangle with color interpolation is shown in Figure 8.5.

Dealing with Pixels on Triangle Edges

We have still not discussed what to do for pixels whose centers are exactly on the edge of a triangle. If a pixel is exactly on the edge of a triangle, then it is also on the edge of the adjacent triangle if there is one. There is no obvious way to award the pixel to one triangle or the other. The worst decision would be to not draw the pixel because a hole would result between the two triangles. Better, but still not good, would be to have both triangles draw the pixel. If the triangles are transparent, this will result in a double-coloring. We would really like to award the pixel to exactly one of the triangles, and we would like this process to be simple; which triangle is chosen does not matter as long as the choice is well defined.

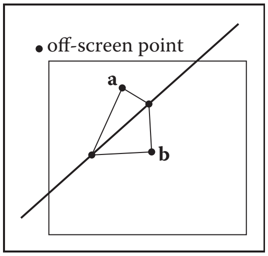


Figure 8.6. The off-screen point will be on one side of the triangle edge or the other. Exactly one of the non-shared vertices **a** and **b** will be on the same side.

One approach is to note that any off-screen point is definitely on exactly one side of the shared edge and that is the edge we will draw. For two non-overlapping triangles, the vertices not on the edge are on opposite sides of the edge from each other. Exactly one of these vertices will be on the same side of the edge as the off-screen point (Figure 8.6). This is the basis of the test. The test if numbers p and q have the same sign can be implemented as the test $pq > 0$, which is very efficient in most environments.

Note that the test is not perfect because the line through the edge may also go through the off-screen point, but we have at least greatly reduced the number of problematic cases. Which off-screen point is used is arbitrary, and $(x, y) = (-1, -1)$ is as good a choice as any. We will need to add a check for the case of a point exactly on an edge. We would like this check not to be reached for common cases, which are the completely inside or outside tests. This suggests:

$$\begin{aligned} x_{\text{min}} &= \text{floor}(x_i) \\ x_{\text{max}} &= \text{ceiling}(x_i) \\ y_{\text{min}} &= \text{floor}(y_i) \end{aligned}$$

```
$y_{\text{max}} = \text{ceiling}(y_i)$
$f_\alpha = f_{12}(x_0, y_0)$
$f_\beta = f_{20}(x_1, y_1)$
$f_\gamma = f_{01}(x_2, y_2)$
for $y = y_{\text{min}}$ to $y_{\text{max}}$ do
    for $x = x_{\text{min}}$ to $x_{\text{max}}$ do
        $\alpha = f_{12}(x, y)/f_\alpha$
        $\beta = f_{20}(x, y)/f_\beta$
        $\gamma = f_{01}(x, y)/f_\gamma$
        if $(\alpha \geq 0$ and $\beta \geq 0$ and $\gamma \geq 0)$ then
            if $(\alpha > 0$ or $f_\alpha f_{12}(-1, -1) > 0)$ and
                $(\beta > 0$ or $f_\beta f_{20}(-1, -1) > 0)$ and
                $(\gamma > 0$ or $f_\gamma f_{01}(-1, -1) > 0)$ then
                $\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$
                drawpixel $(x, y)$ with color $\mathbf{c}$
```

We might expect that the above code would work to eliminate holes and double-draws only if we use exactly the same line equation for both triangles. In fact, the line equation is the same only if the two shared vertices have the same order in the draw call for each triangle. Otherwise the equation might flip in sign. This could be a problem depending on whether the compiler changes the order of operations. So if a robust implementation is needed, the details of the compiler and arithmetic unit may need to be examined. The first four lines in the pseudocode above must be coded carefully to handle cases where the edge exactly hits the pixel center.

In addition to being amenable to an incremental implementation, there are several potential early exit points. For example, if α is negative, there is no need to compute β or γ. While this may well result in a speed improvement, profiling is always a good idea; the extra branches could reduce pipelining or concurrency and might slow down the code. So as always, test any attractive-looking optimizations if the code is a critical section.

Another detail of the above code is that the divisions could be divisions by zero for degenerate triangles, i.e., if $f_\gamma = 0$. Either the floating point error conditions should be accounted for properly, or another test will be needed.

8.1.3 Clipping

Simply transforming primitives into screen space and rasterizing them does not quite work by itself. This is because primitives that are outside the view volume—particularly, primitives that are behind the eye—can end up being rasterized, leading to incorrect results. For instance, consider the triangle shown in Figure 8.7.

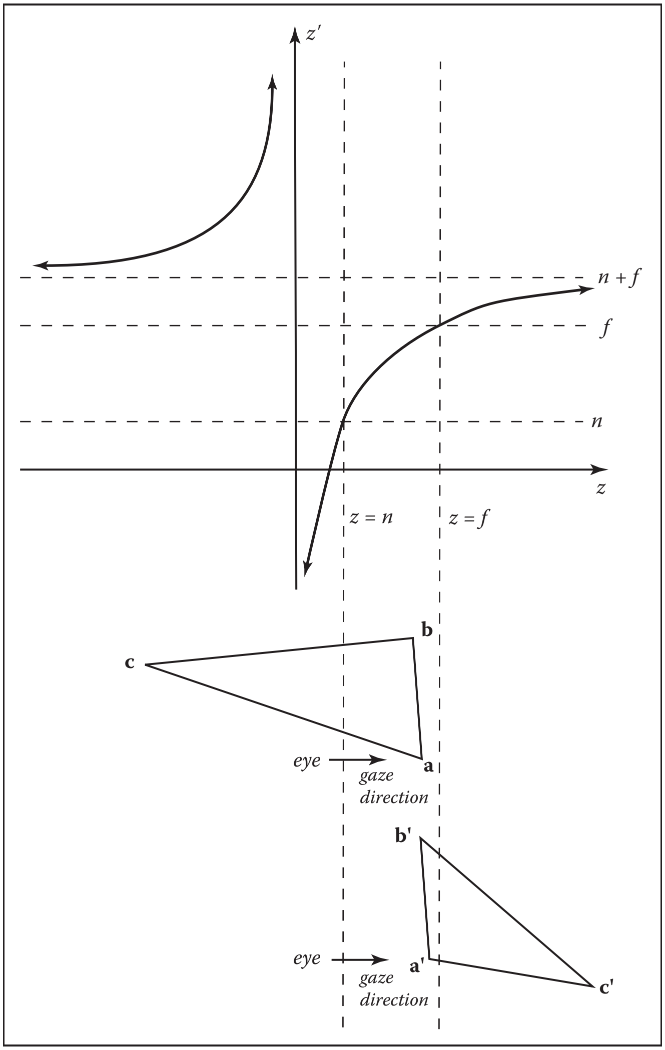


Figure 8.7. The depth z is transformed to the depth z' by the perspective transform. Note that when z moves from positive to negative, z' switches from negative to positive. Thus vertices behind the eye are moved in front of the eye beyond $z' = n + f$. This will lead to wrong results, which is why the triangle is first clipped to ensure all vertices are in front of the eye.

Two vertices are in the view volume, but the third is behind the eye. The projection transformation maps this vertex to a nonsensical location behind the far plane, and if this is allowed to happen the triangle will be rasterized incorrectly. For this reason, rasterization has to be preceded by a *clipping* operation that removes parts of primitives that could extend behind the eye.

Clipping is a common operation in graphics, needed whenever one geometric entity "cuts" another. For example, if you clip a triangle against the plane $x = 0$, the plane cuts the triangle into two parts if the signs of the x-coordinates of the vertices are not all the same. In most applications of clipping, the portion of the triangle on the "wrong" side of the plane is discarded. This operation for a single plane is shown in Figure 8.8.

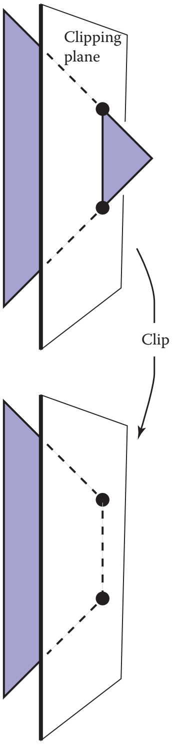


Figure 8.8. A polygon is clipped against a clipping plane. The portion "inside" the plane is retained.

In clipping to prepare for rasterization, the "wrong" side is the side outside the view volume. It is always safe to clip away all geometry outside the view volume—that is, clipping against all six faces of the volume—but many systems manage to get away with only clipping against the near plane.

This section discusses the basic implementation of a clipping module. Those interested in implementing an industrial-speed clipper should see the book by Blinn mentioned in the notes at the end of this chapter.

The two most common approaches for implementing clipping are

1. in world coordinates using the six planes that bound the truncated viewing pyramid,

2. in the 4D transformed space before the homogeneous divide.

Either possibility can be effectively implemented (J. Blinn, 1996) using the following approach for each triangle:

```
for each of six planes do
  if (triangle entirely outside of plane) then
    break (triangle is not visible)
  else if triangle spans plane then
    clip triangle
    if (quadrilateral is left) then
      break into two triangles
```

8.1.4 Clipping Before the Transform (Option 1)

Option 1 has a straightforward implementation. The only question is, "What are the six plane equations?" Because these equations are the same for all triangles rendered in the single image, we do not need to compute them very efficiently. For this reason, we can just invert the transform shown in Figure 5.11 and apply it to the eight vertices of the transformed view volume:

$$\begin{aligned}(x, y, z) = &(l, b, n)\\ &(r, b, n)\\ &(l, t, n)\\ &(r, t, n)\\ &(l, b, f)\\ &(r, b, f)\\ &(l, t, f)\\ &(r, t, f).\end{aligned}$$

The plane equations can be inferred from here. Alternatively, we can use vector geometry to get the planes directly from the viewing parameters.

8.1.5 Clipping in Homogeneous Coordinates (Option 2)

Surprisingly, the option usually implemented is that of clipping in homogeneous coordinates before the divide. Here the view volume is 4D, and it is bounded by 3D volumes (hyperplanes). These are

$$\begin{aligned}-x + lw &= 0,\\ x - rw &= 0,\\ -y + bw &= 0,\\ y - tw &= 0,\\ -z + nw &= 0,\\ z - fw &= 0.\end{aligned}$$

These planes are quite simple, so the efficiency is better than for Option 1. They still can be improved by transforming the view volume $[l, r] \times [b, t] \times [f, n]$ to $[0, 1]^3$. It turns out that the clipping of the triangles is not much more complicated than in 3D.

8.1.6 Clipping against a Plane

No matter which option we choose, we must clip against a plane. Recall from Section 2.5.5 that the implicit equation for a plane through point $\mathbf{q}$ with normal $\mathbf{n}$ is

$$f(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{q}) = 0.$$

This is often written

$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + D = 0. \tag{8.2}$$

Interestingly, this equation not only describes a 3D plane, but it also describes a line in 2D and the volume analog of a plane in 4D. All of these entities are usually called planes in their appropriate dimension.

If we have a line segment between points $\mathbf{a}$ and $\mathbf{b}$, we can "clip" it against a plane using the techniques for cutting the edges of 3D triangles in BSP tree programs described in Section 12.4.3. Here, the points $\mathbf{a}$ and $\mathbf{b}$ are tested to determine whether they are on opposite sides of the plane $f(\mathbf{p}) = 0$ by checking whether $f(\mathbf{a})$ and $f(\mathbf{b})$ have different signs. Typically $f(\mathbf{p}) < 0$ is defined to be "inside" the plane, and $f(\mathbf{p}) > 0$ is "outside" the plane. If the plane does split the line, then we can solve for the intersection point by substituting the equation for the parametric line,

$$\mathbf{p} = \mathbf{a} + t(\mathbf{b} - \mathbf{a}),$$

into the $f(\mathbf{p}) = 0$ plane of Equation (8.2). This yields

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{b} - \mathbf{a})) + D = 0.$$

Solving for t gives

$$t = \frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{a} - \mathbf{b})}.$$

We can then find the intersection point and "shorten" the line.

To clip a triangle, we again can follow Section 12.4.3 to produce one or two triangles.

8.2 Operations Before and After Rasterization

Before a primitive can be rasterized, the vertices that define it must be in screen coordinates, and the colors or other attributes that are supposed to be interpolated across the primitive must be known. Preparing this data is the job of the *vertex-processing* stage of the pipeline. In this stage, incoming vertices are transformed by the modeling, viewing, and projection transformations, mapping them from their original coordinates into screen space (where, recall, position is measured in terms of pixels). At the same time, other information, such as colors, surface normals, or texture coordinates, is transformed as needed; we'll discuss these additional attributes in the examples below.

After rasterization, further processing is done to compute a color and depth for each fragment. This processing can be as simple as just passing through an interpolated color and using the depth computed by the rasterizer; or it can involve complex shading operations. Finally, the blending phase combines the fragments generated by the (possibly several) primitives that overlapped each pixel to compute the final color. The most common blending approach is to choose the color of the fragment with the smallest depth (closest to the eye).

The purposes of the different stages are best illustrated by examples.

8.2.1 Simple 2D Drawing

The simplest possible pipeline does nothing in the vertex or fragment stages, and in the blending stage the color of each fragment simply overwrites the value of the previous one. The application supplies primitives directly in pixel coordinates, and the rasterizer does all the work. This basic arrangement is the essence of many simple, older APIs for drawing user interfaces, plots, graphs, and other 2D content. Solid color shapes can be drawn by specifying the same color for all vertices of each primitive, and our model pipeline also supports smoothly varying color using interpolation.

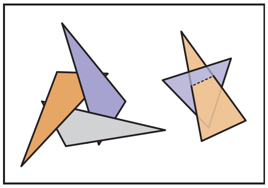


Figure 8.9. Two occlusion cycles, which cannot be drawn in back-to-front order.

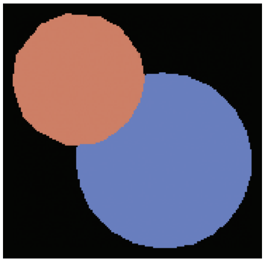


Figure 8.10. The result of drawing two spheres of identical size using the minimal pipeline. The sphere that appears smaller is farther away but is drawn last, so it incorrectly overwrites the nearer one.

8.2.2 A Minimal 3D Pipeline

To draw objects in 3D, the only change needed to the 2D drawing pipeline is a single matrix transformation: the vertex-processing stage multiplies the incoming vertex positions by the product of the modeling, camera, projection, and viewport matrices, resulting in screen-space triangles that are then drawn in the same way as if they'd been specified directly in 2D.

One problem with the minimal 3D pipeline is that in order to get occlusion relationships correct—to get nearer objects in front of farther away objects—primitives must be drawn in back-to-front order. This is known as the *painter's algorithm* for hidden surface removal, by analogy to painting the background of a painting first, then painting the foreground over it. The painter's algorithm is a perfectly valid way to remove hidden surfaces, but it has several drawbacks. It cannot handle triangles that intersect one another, because there is no correct order in which to draw them. Similarly, several triangles, even if they don't intersect, can still be arranged in an *occlusion cycle*, as shown in Figure 8.9, another case in which the back-to-front order does not exist. And most importantly, sorting the primitives by depth is slow, especially for large scenes, and disturbs the efficient flow of data that makes object-order rendering so fast. Figure 8.10 shows the result of this process when the objects are not sorted by depth.

8.2.3 Using a z-Buffer for Hidden Surfaces

In practice, the painter's algorithm is rarely used; instead a simple and effective hidden surface removal algorithm known as the *z-buffer* algorithm is used. The method is very simple: at each pixel we keep track of the distance to the closest surface that has been drawn so far, and we throw away fragments that are farther away than that distance. The closest distance is stored by allocating an extra value for each pixel, in addition to the red, green, and blue color values, which is known as the depth, or z-value. The *depth buffer*, or z-buffer, is the name for the grid of depth values.

The z-buffer algorithm is implemented in the fragment blending phase, by comparing the depth of each fragment with the current value stored in the z-buffer. If the fragment's depth is closer, both its color and its depth value overwrite the values currently in the color and depth buffers. If the fragment's depth is farther away, it is discarded. To ensure that the first fragment will pass the depth test, the z buffer is initialized to the maximum depth (the depth of the far plane). Irrespective of the order in which surfaces are drawn, the same fragment will win the depth test, and the image will be the same.

Of course there can be ties in the depth test, in which case the order may well matter.

The z-buffer algorithm requires each fragment to carry a depth. This is done simply by interpolating the z-coordinate as a vertex attribute, in the same way that color or other attributes are interpolated.

The z-buffer is such a simple and practical way to deal with hidden surfaces in object-order rendering that it is by far the dominant approach. It is much simpler than geometric methods that cut surfaces into pieces that can be sorted by depth, because it avoids solving any problems that don't need to be solved. The depth order only needs to be determined at the locations of the pixels, and that is all that the z-buffer does. It is universally supported by hardware graphics pipelines and is also the most commonly used method for software pipelines. Figure 8.11 shows an example result.

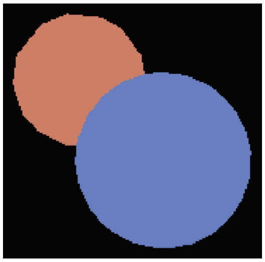


Figure 8.11. The result of drawing the same two spheres using the z-buffer.

Precision Issues

In practice, the z-values stored in the buffer are nonnegative integers. This is preferable to true floats because the fast memory needed for the z-buffer is somewhat expensive and is worth keeping to a minimum.

The use of integers can cause some precision problems. If we use an integer range having B values $\{0, 1, \ldots, B-1\}$, we can map 0 to the near clipping plane $z = n$ and $B-1$ to the far clipping plane $z = f$. Note, that for this discussion, we assume z, n, and f are positive. This will result in the same results as the negative case, but the details of the argument are easier to follow. We send each z-value to

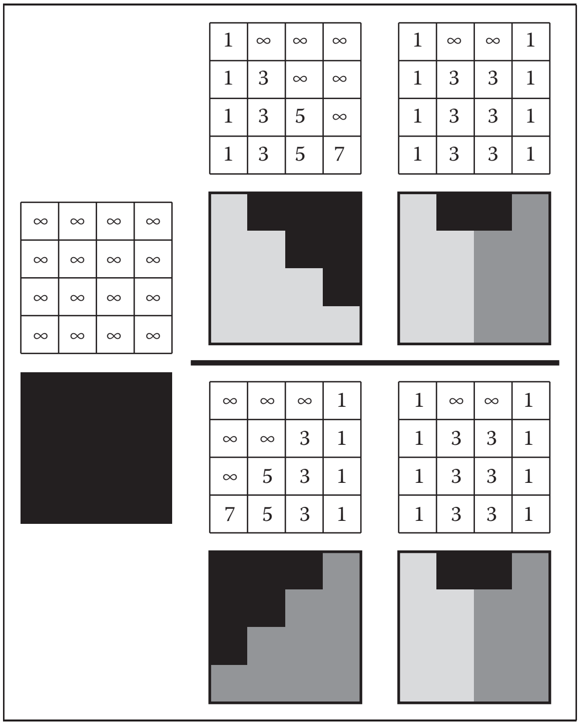


Figure 8.12. A z-buffer rasterizing two triangles in each of two possible orders. The first triangle is fully rasterized. The second triangle has every pixel computed, but for three of the pixels the depth-contest is lost, and those pixels are not drawn. The final image is the same regardless.

a "bucket" with depth $\Delta z = (f - n)/B$. We would not use the integer z-buffer if memory were not a premium, so it is useful to make B as small as possible.

If we allocate b bits to store the z-value, then $B = 2^b$. We need enough bits to make sure any triangle in front of another triangle will have its depth mapped to distinct depth bins.

For example, if you are rendering a scene where triangles have a separation of at least one meter, then $\Delta z < 1$ should yield images without artifacts. There are two ways to make Δz smaller: move n and f closer together or increase b. If b is fixed, as it may be in APIs or on particular hardware platforms, adjusting n and f is the only option.

The precision of z-buffers must be handled with great care when perspective images are created. The value Δz above is used *after* the perspective divide. Recall from Section 7.3 that the result of the perspective divide is

$$z = n + f - \frac{fn}{z_w}.$$

The actual bin depth is related to z_w, the world depth, rather than z, the post-perspective divide depth. We can approximate the bin size by differentiating both sides:

$$\Delta z \approx \frac{fn\Delta z_w}{z_w^2}.$$

Bin sizes vary in depth. The bin size in world space is

$$\Delta z_w \approx \frac{z_w^2 \Delta z}{fn}.$$

Note that the quantity Δz is as previously discussed. The biggest bin will be for $z' = f$, where

$$\Delta z_w^{\max} \approx \frac{f\Delta z}{n}.$$

Note that choosing $n = 0$, a natural choice if we don't want to lose objects right in front of the eye, will result in an infinitely large bin—a very bad condition. To make $\Delta z_w^{\max}$ as small as possible, we want to minimize f and maximize n. Thus, it is always important to choose n and f carefully.

8.2.4 Per-vertex Shading

So far the application sending triangles into the pipeline is responsible for setting the color; the rasterizer just interpolates the colors and they are written directly into the output image. For some applications this is sufficient, but in many cases we want 3D objects to be drawn with shading, using the same illumination equations that we used for image-order rendering in Chapter 4. Recall that these equations require a light direction, an eye direction, and a surface normal to compute the color of a surface.

One way to handle shading computations is to perform them in the vertex stage. The application provides normal vectors at the vertices, and the positions and colors of the lights are provided separately (they don't vary across the surface, so they don't need to be specified for each vertex). For each vertex, the direction to the viewer and the direction to each light are computed based on the positions of the camera, the lights, and the vertex. The desired shading equation is evaluated to compute a color, which is then passed to the rasterizer as the vertex color. Per-vertex shading is sometimes called *Gouraud shading*.

One decision to be made is the coordinate system in which shading computations are done. World space or eye space are good choices. It is important to choose a coordinate system that is orthonormal when viewed in world space, because shading equations depend on angles between vectors, which are not preserved by operations like nonuniform scale that are often used in the modeling transformation, or perspective projection, often used in the projection to the canonical view volume. Shading in eye space has the advantage that we don't need to keep track of the camera position, because the camera is always at the origin in eye space, in perspective projection, or the view direction is always $+z$ in orthographic projection.

Per-vertex shading has the disadvantage that it cannot produce any details in the shading that are smaller than the primitives used to draw the surface, because it only computes shading once for each vertex and never in between vertices. For instance, in a room with a floor that is drawn using two large triangles and illuminated by a light source in the middle of the room, shading will be evaluated only at the corners of the room, and the interpolated value will likely be much too dark in the center. Also, curved surfaces that are shaded with specular highlights must be drawn using primitives small enough that the highlights can be resolved.

Figure 8.13 shows our two spheres drawn with per-vertex shading.

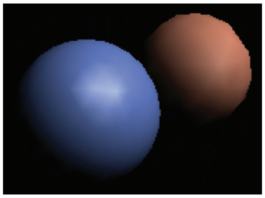


Figure 8.13. Two spheres drawn using per-vertex (Gouraud) shading. Because the triangles are large, interpolation artifacts are visible.

8.2.5 Per-fragment Shading

To avoid the interpolation artifacts associated with per-vertex shading, we can avoid interpolating colors by performing the shading computations *after* the interpolation, in the fragment stage. In per-fragment shading, the same shading equations are evaluated, but they are evaluated for each fragment using interpolated vectors, rather than for each vertex using the vectors from the application.

Per-fragment shading is sometimes called Phong shading, which is confusing because the same name is attached to the Phong illumination model.

In per-fragment shading, the geometric information needed for shading is passed through the rasterizer as attributes, so the vertex stage must coordinate with the fragment stage to prepare the data appropriately. One approach is to interpolate the eye-space surface normal and the eye-space vertex position, which then can be used just as they would in per-vertex shading.

Figure 8.14 shows our two spheres drawn with per-fragment shading.

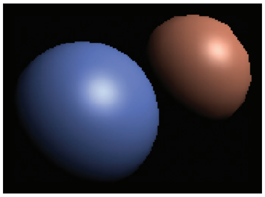


Figure 8.14. Two spheres drawn using per-fragment shading. Because the triangles are large, interpolation artifacts are visible.

8.2.6 Texture Mapping

Textures (discussed in Chapter 11) are images that are used to add extra detail to the shading of surfaces that would otherwise look too homogeneous and artificial. The idea is simple: each time shading is computed, we read one of the values used in the shading computation—the diffuse color, for instance—from a texture instead of using the attribute values that are attached to the geometry being rendered. This operation is known as a *texture lookup*: the shading code specifies a *texture coordinate*, a point in the domain of the texture, and the texture-mapping system finds the value at that point in the texture image and returns it. The texture value is then used in the shading computation.

The most common way to define texture coordinates is simply to make the texture coordinate another vertex attribute. Each primitive then knows where it lives in the texture.

8.2.7 Shading Frequency

The decision about where to place shading computations depends on how fast the color changes—the *scale* of the details being computed. Shading with large-scale features, such as diffuse shading on curved surfaces, can be evaluated fairly infrequently and then interpolated: it can be computed with a low *shading frequency*. Shading that produces small-scale features, such as sharp highlights or detailed textures, needs to be evaluated at a high shading frequency. For details that need to look sharp and crisp in the image, the shading frequency needs to be at least one shading sample per pixel.

So large-scale effects can safely be computed in the vertex stage, even when the vertices defining the primitives are many pixels apart. Effects that require a high shading frequency can also be computed at the vertex stage, as long as the vertices are close together in the image; alternatively, they can be computed at the fragment stage when primitives are larger than a pixel.

For example, a hardware pipeline as used in a computer game, generally using primitives that cover several pixels to ensure high efficiency, normally does most shading computations per fragment. On the other hand, the PhotoRealistic RenderMan system does all shading computations per vertex, after first subdividing, or *dicing*, all surfaces into small quadrilaterals called *micropolygons* that are about the size of pixels. Since the primitives are small, per-vertex shading in this system achieves a high shading frequency that is suitable for detailed shading.

8.3 Simple Antialiasing

Just as with ray tracing, rasterization will produce jagged lines and triangle edges if we make an all-or-nothing determination of whether each pixel is inside the primitive or not. In fact, the set of fragments generated by the simple triangle rasterization algorithms described in this chapter, sometimes called standard or *aliased* rasterization, is exactly the same as the set of pixels that would be mapped to that triangle by a ray tracer that sends one ray through the center of each pixel.

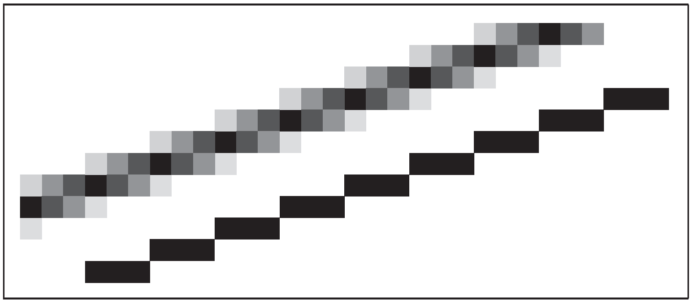


Figure 8.15. An antialiased and a jaggy line viewed at close range so individual pixels are visible.

Also as in ray tracing, the solution is to allow pixels to be partly covered by a primitive (Crow, 1978). In practice, this form of blurring helps visual quality, especially in animations. This is shown as the top line of Figure 8.15.

There are a number of different approaches to antialiasing in rasterization applications. Just as with a ray tracer, we can produce an antialiased image by setting each pixel value to the average color of the image over the square area belonging to the pixel, an approach known as *box filtering*. This means we have to think of all drawable entities as having well-defined areas. For example, the line in Figure 8.15 can be thought of as approximating a one-pixel-wide rectangle.

There are better filters than the box, but a box filter will suffice for all but the most demanding applications.

The easiest way to implement box-filter antialiasing is by *supersampling*: create images at very high resolutions and then downsample. For example, if our goal is a 256 × 256 pixel image of a line with width 1.2 pixels, we could rasterize a rectangle version of the line with width 4.8 pixels on a 1024 × 1024 screen, and then average 4 × 4 groups of pixels to get the colors for each of the 256 × 256 pixels in the "shrunken" image. This is an approximation of the actual box-filtered image, but works well when objects are not extremely small relative to the distance between pixels.

Supersampling is quite expensive, however. Because the very sharp edges that cause aliasing are normally caused by the edges of primitives, rather than sudden variations in shading within a primitive, a widely used optimization is to sample visibility at a higher rate than shading. If information about coverage and depth is stored for several points within each pixel, very good antialiasing can be achieved even if only one color is computed. In systems like RenderMan that use per-vertex shading, this is achieved by rasterizing at high resolution: it is inexpensive to do so because shading is simply interpolated to produce colors for the many fragments, or visibility samples. In systems with per-fragment shading, such as hardware pipelines, *multisample antialiasing* is achieved by storing for each fragment a single color plus a coverage mask and a set of depth values.

8.4 Culling Primitives for Efficiency

The strength of object-order rendering, that it requires a single pass over all the geometry in the scene, is also a weakness for complex scenes. For instance, in a model of an entire city, only a few buildings are likely to be visible at any given time. A correct image can be obtained by drawing all the primitives in the scene, but a great deal of effort will be wasted processing geometry that is behind the visible buildings, or behind the viewer, and therefore doesn't contribute to the final image.

Identifying and throwing away invisible geometry to save the time that would be spent processing it is known as *culling*. Three commonly implemented culling strategies (often used in tandem) are

- view volume culling—the removal of geometry that is outside the view volume;
- occlusion culling—the removal of geometry that may be within the view volume but is obscured, or occluded, by other geometry closer to the camera;
- backface culling—the removal of primitives facing away from the camera.

We will briefly discuss view volume culling and backface culling, but culling in high performance systems is a complex topic; see (Akenine-Möller et al., 2008) for a complete discussion and for information about occlusion culling.

8.4.1 View Volume Culling

When an entire primitive lies outside the view volume, it can be culled, since it will produce no fragments when rasterized. If we can cull many primitives with a quick test, we may be able to speed up drawing significantly. On the other hand, testing primitives individually to decide exactly which ones need to be drawn may cost more than just letting the rasterizer eliminate them.

View volume culling, also known as *view frustum culling*, is especially helpful when many triangles are grouped into an object with an associated bounding volume. If the bounding volume lies outside the view volume, then so do all the triangles that make up the object. For example, if we have 1000 triangles bounded by a single sphere with center $\mathbf{c}$ and radius r, we can check whether the sphere lies outside the clipping plane,

$$(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0,$$

where $\mathbf{a}$ is a point on the plane, and $\mathbf{p}$ is a variable. This is equivalent to checking whether the signed distance from the center of the sphere $\mathbf{c}$ to the plane is greater than $+r$. This amounts to the check that

$$\frac{(\mathbf{c}-\mathbf{a})\cdot\mathbf{n}}{\|\mathbf{n}\|} > r.$$

Note that the sphere may overlap the plane even in a case where all the triangles do lie outside the plane. Thus, this is a conservative test. How conservative the test is depends on how well the sphere bounds the object.

The same idea can be applied hierarchically if the scene is organized in one of the spatial data structures described in Chapter 12.

8.4.2 Backface Culling

When polygonal models are closed, i.e., they bound a closed space with no holes, then they are often assumed to have outward facing normal vectors as discussed in Chapter 10. For such models, the polygons that face away from the eye are certain to be overdrawn by polygons that face the eye. Thus, those polygons can be culled before the pipeline even starts. The test for this condition is the same one used for silhouette drawing given in Section 10.3.1.

Frequently Asked Questions

- I've often seen clipping discussed at length, and it is a much more involved process than that described in this chapter. What is going on here?

The clipping described in this chapter works, but lacks optimizations that an industrial-strength clipper would have. These optimizations are discussed in detail in Blinn's definitive work listed in the chapter notes.

- How are polygons that are not triangles rasterized?

These can either be done directly scan-line by scan-line, or they can be broken down into triangles. The latter appears to be the more popular technique.

- Is it always better to antialias?

No. Some images look crisper without antialiasing. Many programs use unantialiased "screen fonts" because they are easier to read.

• The documentation for my API talks about "scene graphs" and "matrix stacks." Are these part of the graphics pipeline?

The graphics pipeline is certainly designed with these in mind, and whether we define them as part of the pipeline is a matter of taste. This book delays their discussion until Chapter 12.

• Is a uniform distance z-buffer better than the standard one that includes perspective matrix nonlinearities?

It depends. One "feature" of the nonlinearities is that the z-buffer has more resolution near the eye and less in the distance. If a level-of-detail system is used, then geometry in the distance is coarser and the "unfairness" of the z-buffer can be a good thing.

• Is a software z-buffer ever useful?

Yes. Most of the movies that use 3D computer graphics have used a variant of the software z-buffer developed by Pixar (Cook, Carpenter, & Catmull, 1987).

Notes

A wonderful book about designing a graphics pipeline is *Jim Blinn's Corner: A Trip Down the Graphics Pipeline* (J. Blinn, 1996). Many nice details of the pipeline and culling are in *3D Game Engine Design* (Eberly, 2000) and *Real-Time Rendering* (Akenine-Möller et al., 2008).

Exercises

1. Suppose that in the perspective transform we have $n = 1$ and $f = 2$. Under what circumstances will we have a "reversal" where a vertex before and after the perspective transform flips from in front of to behind the eye or vice versa?

2. Is there any reason not to clip in x and y after the perspective divide (see Figure 11.2, stage 3)?

3. Derive the incremental form of the midpoint line-drawing algorithm with colors at endpoints for $0 < m \leq 1$.

4. Modify the triangle-drawing algorithm so that it will draw exactly one pixel for points on a triangle edge which goes through $(x, y) = (-1, -1)$.

5. Suppose you are designing an integer z-buffer for flight simulation where all of the objects are at least one meter thick, are never closer to the viewer than 4 meters, and may be as far away as 100 km. How many bits are needed in the z-buffer to ensure there are no visibility errors? Suppose that visibility errors only matter near the viewer, i.e., for distances less than 100 meters. How many bits are needed in that case?

9

Signal Processing

In graphics, we often deal with functions of a continuous variable: an image is the first example you have seen, but you will encounter many more as you continue your exploration of graphics. By their nature, continuous functions can't be directly represented in a computer; we have to somehow represent them using a finite number of bits. One of the most useful approaches to representing continuous functions is to use *samples* of the function: just store the values of the function at many different points and *reconstruct* the values in between when and if they are needed.

You are by now familiar with the idea of representing an image using a two-dimensional grid of pixels—so you have already seen a sampled representation! Think of an image captured by a digital camera: the actual image of the scene that was formed by the camera's lens is a continuous function of the position on the image plane, and the camera converted that function into a two-dimensional grid of samples. Mathematically, the camera converted a function of type $\mathbb{R}^2 \rightarrow \mathbf{C}$ (where $\mathbf{C}$ is the set of colors) to a two-dimensional array of color samples, or a function of type $\mathbb{Z}^2 \rightarrow \mathbf{C}$.

Another example of a sampled representation is a 2D digitizing tablet, such as the screen of a tablet computer or a separate pen tablet used by an artist. In this case, the original function is the motion of the stylus, which is a time-varying 2D position, or a function of type $\mathbb{R} \rightarrow \mathbb{R}^2$. The digitizer measures the position of the stylus at many points in time, resulting in a sequence of 2D coordinates, or a function of type $\mathbb{Z} \rightarrow \mathbb{R}^2$. A *motion capture* system does exactly the same thing for a special marker attached to an actor's body: it takes the 3D position of the marker over time ($\mathbb{R} \to \mathbb{R}^3$) and makes it into a series of instantaneous position measurements ($\mathbb{Z} \to \mathbb{R}^3$).

Going up in dimension, a medical CT scanner, used to non-invasively examine the interior of a person's body, measures density as a function of position inside the body. The output of the scanner is a 3D grid of density values: it converts the density of the body ($\mathbb{R}^3 \to \mathbb{R}$) to a 3D array of real numbers ($\mathbb{Z}^3 \to \mathbb{R}$).

These examples seem different, but in fact they can all be handled using exactly the same mathematics. In all cases a function is being sampled at the points of a *lattice* in one or more dimensions, and in all cases we need to be able to reconstruct that original continuous function from the array of samples.

From the example of a 2D image, it may seem that the pixels are enough, and we never need to think about continuous functions again once the camera has discretized the image. But what if we want to make the image larger or smaller on the screen, particularly by non-integer scale factors? It turns out that the simplest algorithms to do this perform badly, introducing obvious visual artifacts known as *aliasing*. Explaining why aliasing happens and understanding how to prevent it require the mathematics of sampling theory. The resulting algorithms are rather simple, but the reasoning behind them, and the details of making them perform well, can be subtle.

Representing continuous functions in a computer is, of course, not unique to graphics; nor is the idea of sampling and reconstruction. Sampled representations are used in applications from digital audio to computational physics, and graphics is just one (and by no means the first) user of the related algorithms and mathematics. The fundamental facts about how to do sampling and reconstruction have been known in the field of communications since the 1920s and were stated in exactly the form we use them by the 1940s (Shannon & Weaver, 1964).

This chapter starts by summarizing sampling and reconstruction using the concrete one-dimensional example of digital audio. Then, we go on to present the basic mathematics and algorithms that underlie sampling and reconstruction in one and two dimensions. Finally, we go into the details of the frequency-domain viewpoint, which provides many insights into the behavior of these algorithms.

9.1 Digital Audio: Sampling in 1D

Although sampled representations had already been in use for years in telecommunications, the introduction of the compact disc in 1982, following the increased use of digital recording for audio in the previous decade, was the first highly visible consumer application of sampling.

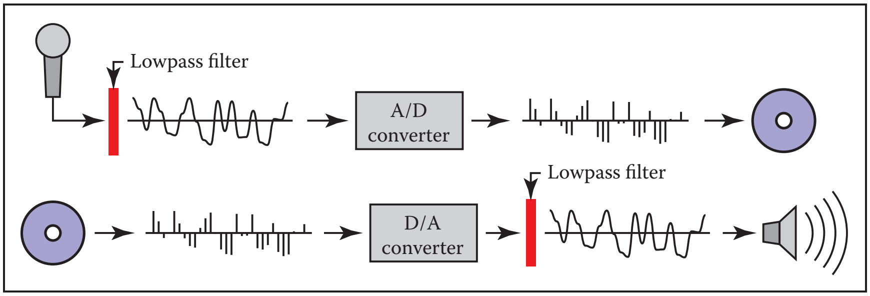


Figure 9.1. Sampling and reconstruction in digital audio.

In audio recording, a microphone converts sound, which exists as pressure waves in the air, into a time-varying voltage that amounts to a measurement of the changing air pressure at the point where the microphone is located. This electrical signal needs to be stored somehow so that it may be played back at a later time and sent to a loudspeaker that converts the voltage back into pressure waves by moving a diaphragm in synchronization with the voltage.

The digital approach to recording the audio signal (Figure 9.1) uses sampling: an *analog-to-digital converter* (*A/D converter*, or *ADC*) measures the voltage many thousand times per second, generating a stream of integers that can easily be stored on any number of media, say a disk on a computer in the recording studio, or transmitted to another location, say the memory in a portable audio player. At playback time, the data is read out at the appropriate rate and sent to a *digital-to-analog converter* (*D/A converter*, or *DAC*). The DAC produces a voltage according to the numbers it receives, and, provided we take enough samples to fairly represent the variation in voltage, the resulting electrical signal is, for all practical purposes, identical to the input.

It turns out that the number of samples per second required to end up with a good reproduction depends on how high-pitched the sounds are that we are trying to record. A sample rate that works fine for reproducing a string bass or a kick drum produces bizarre-sounding results if we try to record a piccolo or a cymbal; but those sounds are reproduced just fine with a higher sample rate. To avoid these *undersampling artifacts* the digital audio recorder *filters* the input to the ADC to remove high frequencies that can cause problems.

Another kind of problem arises on the output side. The DAC produces a voltage that changes whenever a new sample comes in, but stays constant until the next sample, producing a stair-step shaped graph. These stair-steps act like noise, adding a high-frequency, signal-dependent buzzing sound. To remove this *reconstruction artifact*, the digital audio player filters the output from the DAC to smooth out the waveform.