



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 are drawing a model consisting only of 3D line segments that are specified by

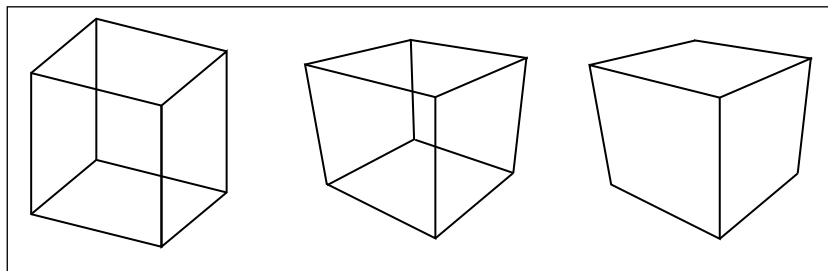


Figure 7.1. Left: wireframe cube in orthographic projection. Middle: wireframe cube in perspective projection. Right: perspective projection with hidden lines removed.

the (x, y, z) coordinates of their two end points. 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:

- 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. The camera transformation converts points in canonical coordinates (or world

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

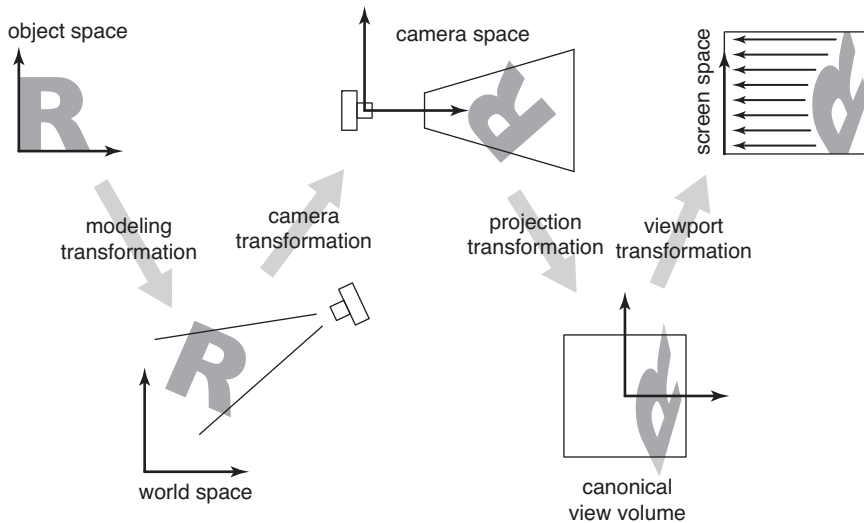


Figure 7.2. The sequence of spaces and transformations that gets objects from their original coordinates into screen space.

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.

Recall the conventions for pixel coordinates from Chapter 3: each pixel “owns” a unit square centered at integer coordinates; the image boundaries have a half-unit overshoot from the pixel centers; and the smallest pixel center coordinates

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

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.

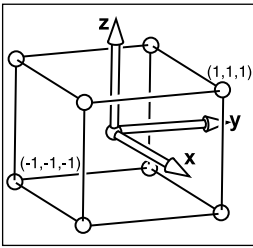


Figure 7.3. The canonical view volume is a cube with side of length two centered at the origin.

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 $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

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):

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

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).

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

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



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 .

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:

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

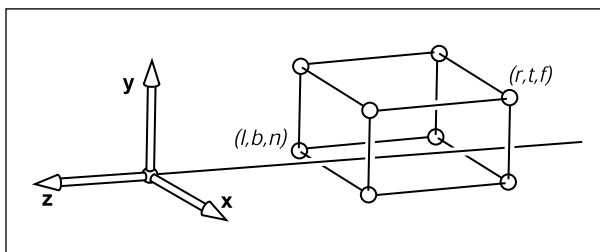


Figure 7.5. The orthographic view volume is along the negative z -axis, so f is a more negative number than n , thus $n > f$.

¹Most 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.

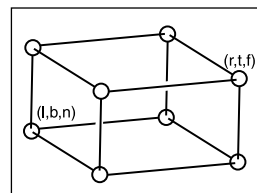


Figure 7.4. The orthographic view volume.

n and f appear in what might seem like reverse order because $n - f$, rather than $f - n$, is a positive number.

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.

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( $x_p, y_p, x_q, y_q$ )
```

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

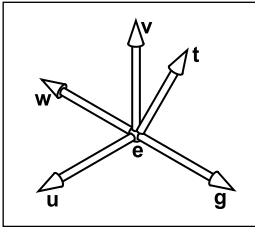


Figure 7.6. The user specifies viewing as an eye position \mathbf{e} , a gaze direction \mathbf{g} , and an up vector \mathbf{t} . We construct a right-handed basis with \mathbf{w} pointing opposite to the gaze and \mathbf{v} being in the same plane as \mathbf{g} and \mathbf{t} .

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} .

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,

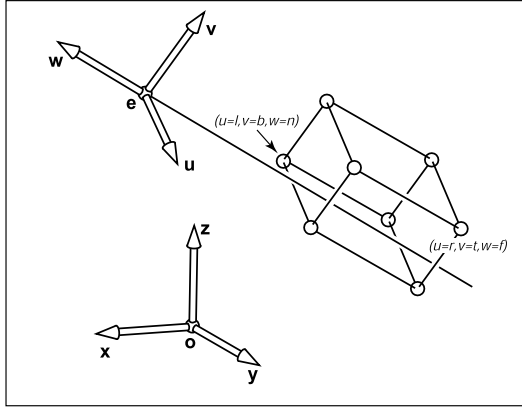


Figure 7.7. For arbitrary viewing, we need to change the points to be stored in the “appropriate” coordinate system. In this case it has origin \mathbf{e} and offset coordinates in terms of \mathbf{uvw} .

using the construction of Section 2.4.7:

$$\begin{aligned}\mathbf{w} &= -\frac{\mathbf{g}}{\|\mathbf{g}\|}, \\ \mathbf{u} &= \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}, \\ \mathbf{v} &= \mathbf{w} \times \mathbf{u}.\end{aligned}$$

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 x -, y -, and 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  $\mathbf{M}_{vp}$ 
construct  $\mathbf{M}_{orth}$ 
construct  $\mathbf{M}_{cam}$ 
 $\mathbf{M} = \mathbf{M}_{vp}\mathbf{M}_{orth}\mathbf{M}_{cam}$ 
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.

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:

$$y_s = \frac{d}{z}y, \quad (7.5)$$

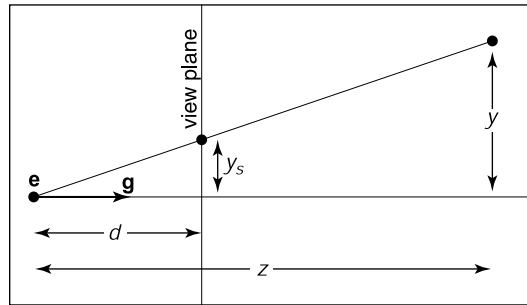


Figure 7.8. The geometry for Equation (7.5). The viewer's eye is at \mathbf{e} and the gaze direction is \mathbf{g} (the minus z -axis). The view plane is a distance d from the eye. A point is projected toward \mathbf{e} and where it intersects the view plane is where it is drawn.

For the moment we will ignore the sign of z to keep the equations simpler, but it will return on page 152.



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 $[x \ y \ z \ 1]^T$; the extra coordinate, w , is always equal to 1, and this is ensured by always using $[0 \ 0 \ 0 \ 1]^T$ 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 $[x \ y \ z \ w]^T$ 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*.

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 $[1 \ 0 \ 1]^T$ 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 $[3 \ 0 \ 1]^T$, 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.

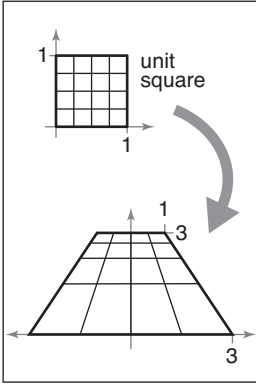


Figure 7.9. A projective transformation maps a square to a quadrilateral, preserving straight lines but not parallel lines.

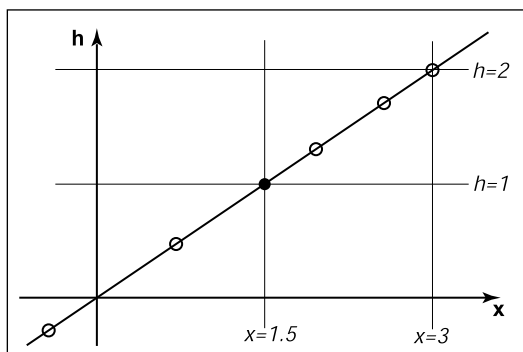


Figure 7.10. The point $x = 1.5$ is represented by any point on the line $x = 1.5h$, such as points at the hollow circles. However, before we interpret x as a conventional Cartesian coordinate, we first divide by h to get $(x, h) = (1.5, 1)$ as shown by the black point.

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 $[1.5 \ 1]^T$, 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 $[-2; -1; 2]^T$, or any other point on the line $\mathbf{x} = \alpha[-1 \ -0.5 \ 1]^T$. 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.)

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.

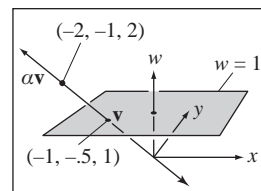


Figure 7.11. A point in homogeneous coordinates is equivalent to any other point on the line through it and the origin, and normalizing the point amounts to intersecting this line with the plane $w = 1$.

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 $[y; z; 1]^T$ to the 1D homogeneous vector $[dy \ z]^T$, which represents the 1D point (dy/z) (because it is equivalent to the 1D homogeneous vector $[dy/z \ 1]^T$). 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 $-n$.

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.

There are many matrices that could function as perspective matrices, and all of them non-linearly 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

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

Remember, $n < 0$.

More on this later.

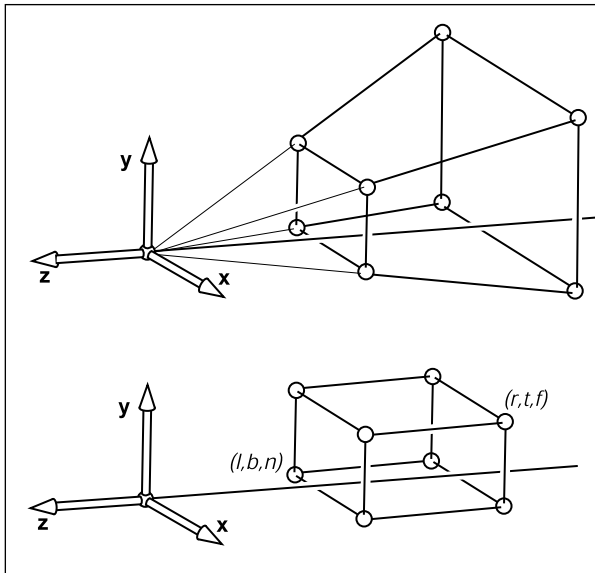


Figure 7.12. The perspective projection leaves points on the $z = n$ plane unchanged and maps the large $z = f$ rectangle at the back of the perspective volume to the small $z = f$ rectangle at the back of the orthographic volume.

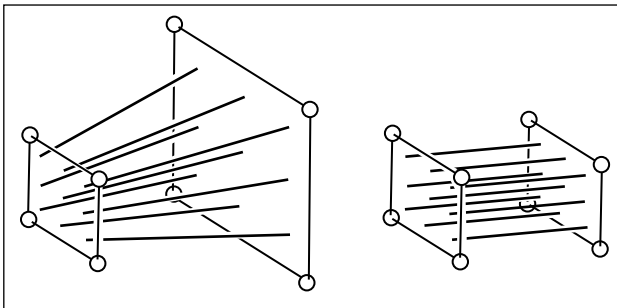


Figure 7.13. The perspective projection maps any line through the origin/eye to a line parallel to the z -axis and without moving the point on the line at $z = n$.

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 :

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

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

Taken in the context of the orthographic projection matrix \mathbf{M}_{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}_{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}_{orth} with \mathbf{M}_{per} , which inserts the perspective matrix \mathbf{P} after the camera matrix \mathbf{M}_{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 .


Multipled out, the matrix \mathbf{M}_{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 et al., 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 set 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:

$$\mathbf{M}\mathbf{q} + t(\mathbf{M}\mathbf{Q} - \mathbf{M}\mathbf{q}) \equiv \mathbf{r} + t(\mathbf{R} - \mathbf{r}).$$

The homogenized 3D line segment is

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

If Equation (7.6) can be rewritten in a form

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

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

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

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

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

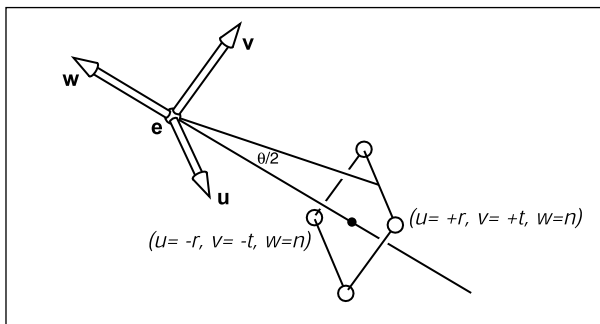


Figure 7.14. The field-of-view θ is the angle from the bottom of the screen to the top of the screen as measured from the eye.

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:

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

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 judgements 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 result 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.

