Chapter 13

Camera Specifications and Transformations

After this chapter, you will know:

- How cameras are specified in modern graphics systems
- How to convert a camera specification into a projective transformation

13.1 An introductory example

Suppose that we have placed a number of objects into the half-space \( z < 0 \) in the world (see figure 1.1). We’d like to make a picture of these by drawing the shadows they’d cast on a square region of the \( z = 0 \) plane, when illuminated by the setting sun, whose rays happen to shine exactly in the positive-\( z \) direction.

We can take the \( xyz \)-coordinates of each point of each object, and compute the location of the shadow of this point: because the sun’s rays point in the positive-\( z \) direction, we simply increase the \( z \)-value of the point until it’s zero. The point \((x, y, z, 1)\) casts a shadow on our “image plane” \((z = 0)\) at the point \((x, y, 0, 1)\).

The transformation that performs this projection has a matrix that’s very simple:

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}; \quad (13.1)
\]
Figure 13.1: There are several objects in the negative-$z$ half-space; we’d like to make a picture of these objects by finding the shadows that they cast when illuminated by a sun that shines in the positive-$z$ direction onto an imaging region in the $z = 0$ plane in the area $-1 \leq x, y \leq 1$.

If we want to go from 3D coordinates to 2D coordinates, we can actually drop the $z$-coordinate using the matrix

$$M_{\text{proj}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

as you can verify by multiplication. Our “image points” all lie on the $z = 0$ plane, but some of them are outside the range $-1 \leq x, y \leq 1$. We’ll want to ignore these in making our picture; we’ll discuss the further in Section 1.10.

One note about the transformation of Equation 1.1: it sends some nonzero items to zero (e.g., the vector $[0\ 0\ 1\ 0]^T$ to zero). That means that it’s not invertible: given $Mv$, it’s impossible to determine $v$. This corresponds to the physical notion that multiple points in space project to the same point in the image, resulting in occlusion of one object by another, for instance. This is one of the rare non-invertible matrices we’ll encounter.

13.1.1 From imaging rectangle to computer screen

Suppose that we eventually intend to display the image on a computer screen. Perhaps the screen coordinates are expressed in millimeters, perhaps in pixels. We need to transform the rectangle in the image plane (ranging from -1 to 1 in both $x$ and $y$) to millimeter or pixel-coordinates. Following the development in Chapter ??, let’s work out two examples.

In the first case, suppose the display is 300mm wide and 300mm tall, with $(0,0)$ representing the lower-left corner of the screen. We can convert our imaging rectangle to the display rectangle in several steps (see
Figure 13.2: To change from the imaging rectangle to the window coordinates, we can follow several steps: (a) Move the imaging rectangle until its lower-left corner is at the origin, and upper right is at (2, 2). (b) Scale uniformly by \( \frac{1}{2} \) to make it go from (0, 0) to (1, 1), (c) scale uniformly to make it go from (0, 0) to (300, 300). By composing these three transformations, we arrive at the desired transformation.

We first translate by \( \begin{bmatrix} 1 \\ 1 \end{bmatrix} \) to move the lower-left corner to (0, 0). Then we scale uniformly by \( \frac{1}{2} \) to make it into the unit square. Then we scale uniformly to make it the largest square that will fit the display, namely one of side 300mm.

The \( 3 \times 3 \) matrices (operating on homogeneous coordinates in \( \mathbb{R}^2 \)) that effect each of these changes are

\[
M_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \quad M_2 = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad M_3 = \begin{bmatrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 0 & 0 & 1 \end{bmatrix},
\]

and we can multiply them together to get the desired transformation from coordinates in our imaging rectangle to screen coordinates in millimeters:

\[
M_3M_2M_1 = \begin{bmatrix} 150 & 0 & 150 \\ 0 & 150 & 150 \\ 0 & 0 & 1 \end{bmatrix}
\] (13.2)

As we discussed in Chapter ??, a graphics programmer should have a linear algebra package that can compute elementary transformations from their specifications, and can compose transformations. In our package, we would write the windowing transformation above in the form

```java
AffineTransform2 t1 = Translate(new Point(1, 1));
AffineTransform2 t2 = AxisScale(0.5, 0.5);
AffineTransform2 t3 = AxisScale(300, 300);
AffineTransform2 t = t3 * t2 * t1;
```

One could, of course, explicitly build the transform from the entries in the product matrix in equation 1.2, but then the sequence of operations that led to the transformation is less clear. And when one later decides to scale the imaging rectangle nonuniformly so that it covers a different portion of
the display, only a few characters need be changed in the code above, as opposed to changing several entries in the matrix.

Because the transformation $t$ is computed once, but will be then applied to many points, the cost of the multiplications in building $t$ is negligible, while the clarity in the code is priceless.

Sometimes, however, it’s easier to use a slightly higher-level construction to build transformations: rather than saying how to create a transformation as a sequence of elementary transformation, it’s better to say what we want the transformation to accomplish. Using our linear algebra package (section ??), we can simply say that we want a linear map that takes certain points to certain others, and let the package find the unique solution to this problem. We’ll use this approach in our second case.

In the second case, suppose we want to take the square $-1 \leq x, y \leq 1$ to a display with square pixels, 1024 pixels wide, 768 pixels tall. The upper left pixel is called $(0, 0)$; the lower left is $(0, 767)$; the lower right is $(1023, 767)$. We want to find a transformation $T$ with the property that it sends the square $-1 \leq x, y \leq 1$ to a square region on the left-hand side of the display, one that fills as much of the display as possible. To do so, we need coordinates on the plane of the display. Because pixel coordinates refer to the center of each pixel (Figure 1.3), the display coordinates range from $(-0.5, -0.5)$ to $(1023.5, 767.5)$.

That means that we want the point $(-1, -1)$ (the lower left corner of the imaging rectangle) to be sent to $(-0.5, 767.5)$ (the lower left corner of the display) and $(-1, 1)$ (the upper left) to be sent to $(-0.5, -0.5)$. To completely specify an affine transformation on a two-dimensional space, we need to know where three independent points are sent. For our third point, we choose the lower right corner: $(1, -1)$ must go to $(767.5, 767.5)$ (to keep the image square). The code that implements this is

```csharp
Transform t = Transform.PointsToPoints(
    new Point(-1, -1), new Point(-1, 1), new Point(1, -1),
    new Point2(-0.5, 767.5), new Point2(-0.5, -0.5), new Point2(767.5, 767.5));
```

Of course, for this to work we need to know that the “source points” are independent; in two dimensions, this simply means “non-collinear,” which is clearly the case.

In general, this latter form, once one gets used to it, is a remarkably expressive way to describe transformations, and easy to both document and debug. It is this technique that we’ll use to determine the camera transformation that sends a view-volume defined by a virtual camera’s view of the world, together with two “clipping planes” (see figure 1.4). Because this transformation must take the intersecting edges of the view frustum (they intersect at the virtual camera) and map them to parallel lines, the transformation cannot be an affine one; indeed, it will turn out to be a projective transformation instead.
Figure 13.4: A synthetic camera sees objects in a rectangular frustum of space; two “clipping planes” determine a piece of this frustum called the view volume. We’ll find a transformation that takes this frustum to the “standard perspective view volume” shown on the right. The transformation will be a projective transformation rather than an affine one (at least for ‘perspective cameras’ rather than ‘parallel cameras.’)

In the next several sections, we’ll do something quite analogous to the view-plane transformations we just performed, but one dimension higher: we’ll transform a frustum of an infinite pyramid or parallelepiped (this frustum is called a “view volume”) into a standard parallelepiped. The first part of the process is the description of the view volume; the second is computing how it should be transformed. In our 2D case, describing the rectangle was simple; in 3D, it requires more effort.

13.2 Camera Specifications

The virtual camera with which we take a picture of a virtual scene is, in most systems, a pinhole camera: light is assumed to enter the camera (see figure 1.5) through a pinhole and arrive at an imaging plane, upon which we have a coordinate system; the point where the light arrives is described in terms of this coordinate system. We can place a larger imaging rectangle further back, or a smaller one closer to the pinhole and get a scaled version of the same image; if we adjust the coordinates for these rectangles, the coordinate of the imaging point in all three cases will be the same. Indeed, we can even imagine placing the imaging plane in front of the pinhole. This cannot work in a physical camera, but for a virtual one it works fine. To get the same picture, we need only reverse the coordinates on this “out front” imaging plane as shown in figure 1.5. Because of the technology available when graphics was developed, the imaging plane is often called the film plane.

In this chapter, we’ll explain how one describes such a virtual camera, and how this description is converted into a matrix that we’ll use in the
Figure 13.5: (a) The model of the virtual camera is that it's a pinhole camera: light enters a pinhole and reaches a point on the imaging plane behind the pinhole; the imaging point depends solely on the geometry and position of the camera and the line along which the light arrives (the projector). (b) The imaging plane can be placed in front of the pinhole as well, although this doesn't correspond to any real camera. As long as the coordinates on the imaging plane are reversed, the same projector is assigned to the same imaging-plane coordinates in both versions of the camera.

graphics pipeline. The discussion that follows, then, can be thought of as the description of the inputs to a procedure that will produce a matrix for the graphics programmer. The “user” in this case is the graphics programmer who wants to create a picture.

What data do we need to completely specify a virtual camera? First and foremost, we need to know where the camera is. In WPF (and many other graphics systems) this is indicated by giving the Location of the pinhole. In some older systems, it was common to specify a point — the view reference point — on the imaging plane, and determine the pinhole location with respect to this; we’ll discuss this in the exercises. One advantage of this approach is that it can also be used with an orthographic projection — a kind of projection in which the rays arriving at the camera are imagined to be all parallel to each other (see figure ??), rather than converging to a point as they do in a pinhole camera. Such orthographic projections are used in mechanical drawings. In a sense, an orthographic projection is a limit of perspective projections as the pinhole “moves out to infinity.” In orthographic projections in WPF, the Location, instead of being at infinity, serves as the center of the imaging plane.
For now, we’ll concentrate on the perspective projection generated by a pinhole camera. So we’ll denote the location of the pinhole by \( P \).

The second thing we need to know is what direction the camera’s looking; WPF describes this with a nonzero vector, the \textit{LookDirection}. We’ll repeatedly need to use a unit-length version of this vector, so we define

\[
\mathbf{w} = -\frac{\text{LookDirection}}{\|\text{LookDirection}\|}.
\]

Note the negative sign; soon we’ll define vectors \( \mathbf{u} \) and \( \mathbf{v} \), and we want \((\mathbf{u}, \mathbf{v}, \mathbf{w})\) to be a right-handed coordinate system. The name \( \mathbf{w} \) fits well with \( \mathbf{u} \) and \( \mathbf{v} \), but clashes slightly with the use of \( \mathbf{w} \) for the fourth coordinate in the homogeneous-coordinate representation of points. Since the latter is italic, while the former is roman boldface, and since we’ll have very little occasion to discuss homogeneous coordinates here, this should not be a problem. The vector we’ve called \( \mathbf{w} \) appears again, in more general camera specifications, as the “view plane normal,” and is traditionally denoted \( \mathbf{n} \). Renamings like this (and switches in handedness of coordinate systems) abound in computer graphics; you’ll need to adapt to this if you ever use programs that you have not written yourself.

Third, we need to know how the camera is oriented relative to this view direction — is it (for a horizontal view vector) held upright or tilted to one side or the other by some amount? This is specified by giving a second vector, the \textit{UpDirection}. While the direction of “up” on a conventional camera is perpendicular to the direction the camera is looking, it’s quite difficult in general to specify a vector perpendicular to \( \mathbf{w} \) unless \( \mathbf{w} \) has some nice form (e.g., \( \mathbf{w} = e_3 \)). Instead, the \textit{UpDirection} can be any nonzero vector that’s not parallel to the \textit{LookDirection} (Figure 1.6). Often in practice the \textit{UpDirection} is set to \texttt{Vector3D(0,1,0)}, i.e., the \textit{y}-direction. As long as the camera does not look straight up or down, this is the most natural direction in which to hold it. In practice, we’ll need a vector that’s actually perpendicular to \( \mathbf{w} \), however, so we project out the \( \mathbf{w} \) component of \textit{UpDirection} and then normalize

\[
\mathbf{v} = \frac{\text{UpDirection} - (\mathbf{w} \cdot \text{UpDirection})\mathbf{w}}{\|\mathbf{w}\|};
\]

\( \square \) **Travis says he prefers the 123 slides explanation here**

All that remains is to specify how wide a \textit{field of view} we have, i.e., do we see everything from 15 degrees to the left of the view vector to 15 degrees to the right of it (a field of view of 30 degrees)? Or do we have a field of view of 90 degrees?. The same question arises in the vertical direction, and the answers together determine the shape of the resulting image (a large vertical field of view and a small horizontal field of view give a tall and thin image). Of course, you might say that you \textit{know} the \textit{aspect ratio} (the ratio of width to height) for your image, and don’t want to have to adjust two
Figure 13.6: Each of the vectors $v_1$, $v_2$, and $v_3$, which all lie in the plane of LookDirection and $v$, all can be used as the UpDirection for the same camera.

angles. Some systems, including WPF, let you specify the aspect ratio and the horizontal field of view, and compute the vertical field of view for you; alternatively, one can specify both fields of view. In fact, in WPF you specify the width and height of the viewport (a rectangle on the display used for exhibiting the image), and the aspect ratio is determined by the ratio of the width to the height\(^1\). In this presentation, we'll specify the view in terms of both horizontal and vertical field of view, and discuss the relationship to aspect ratio afterwards.

**Inline Exercise 13.1:** Determine the horizontal field of view for the displays of the devices you most often use (your desktop display, your cellphone, your television), as seen at their usual viewing distance (for your desktop display, perhaps 0.75m; for your television, perhaps 3m). Compare these to the default field of view for a PerspectiveCamera in WPF.

Everything we've discussed so far assumes a camera in which the view is symmetric about the view direction (i.e., a ray from the pinhole in the view direction corresponds to a point in the center of the resulting image — we see as much to the left of the view direction as to the right, and as much above as below). More complex view cameras have the ability to adjust the

\(^1\)In general, the aspect ratio of a rectangle is the ratio of the longer edge to the shorter; because many desktop displays are oriented so that they are wider than they are tall, for displays it's come to mean "width to height ratio," and that's the sense we'll use here, even when the height is greater than the width.
Figure 13.9: All the parts of a camera specification (shown with the camera itself omitted for clarity).

location of the view rectangle relative to the pinhole and view direction (see Figure 1.7). We’ll discuss this more complex case later in the chapter.

Although the position, look-direction, up-direction, and horizontal and vertical fields of view suffice to completely specify a physical pinhole camera, we also include two non-physical items: so-called clipping planes (see Figure 1.8). These are planes perpendicular to the view direction that limit which objects are rendered: any part of an object nearer to the camera point $P$ than the near clipping plane is not drawn; and part farther away than the far clipping plane is also not drawn. For certain kinds of rendering (like raytracing), these near and far planes can be ignored. But for the polygon-based rendering done by modern graphics cards, they are essential, as we’ll see presently. These planes are specified by giving their distance from $P$ along the LookDirection vector.

In summary, our camera specification (see Figure ??) consists of

- $P$, the camera location,
- LookDirection, the direction in which the camera’s pointed,
- UpDirection, a nonzero vector in the plane determined by the look-direction and the vertical direction on the camera body,
- $\theta_h$, the horizontal field of view angle (in degrees),
- $\theta_v$, the vertical field of view angle (in degrees),
- $n$, the distance from $P$ to the near clipping plane, and
- $f$, the distance from $P$ to the far clipping plane.
13.2.1 Auxiliary computations

We'll now convert this camera specification into a collection of mathematically useful objects (points, vectors, coordinates), and then find an affine transformation to send the view frustum — the portion of the infinite rectangular pyramid viewed by our camera that lies between the clipping planes — to the standard perspective view volume shown in Figure 1.10 in such a way that the far clipping plane is sent to the $z = -1$ plane, and the point $P$ is sent to the origin.

**Inline Exercise 13.2:** What's the horizontal field of view for the standard perspective view volume, in degrees? (The vertical field of view is the same, by symmetry.)

The first step is to convert the `LookDirection` into a unit vector $w$ in the opposite direction, and the `UpDirection` into a unit vector $v$ that's perpendicular to $w$. In equations, we let

$$w = -\frac{\text{LookDirection}}{\|\text{LookDirection}\|};$$
$$t = \text{UpDirection} - (\text{UpDirection} \cdot w)w;$$
$$v = \frac{t}{\|t\|}.$$
We then extend \( w \) and \( v \) to a coordinate frame by defining
\[
u = v \times w.
\]
Now the vectors \( u, v, w \) form a right-handed orthonormal coordinate system. We’ll soon build a transformation that sends these to \( e_1, e_2 \) and \( e_3 \).

**Inline Exercise 13.3:** Alternatively, we can compute the vector \( u \) before computing \( v \) by
\[
\begin{align*}
w &= -\text{LookDirection}/\|\text{LookDirection}\|; \\
t &= \text{UpDirection} - (\text{UpDirection} \cdot w)w; \\
u &= t \times w,
\end{align*}
\]
\[
\begin{align*}
u &= u/\|u\|, \\
v &= w \times u.
\end{align*}
\]
Explain why the results are the same.

The center of the far plane is \( Q = P + fw \). But how wide is the rectangle at the far end of the view frustum? That depends on the field of view. Looking down from above on the view frustum (Figure 1.11), we see that half-width of the far-plane rectangle, divided by the far-plane distance, is the tangent of half the horizontal field of view, i.e.,
\[
\tan \frac{\theta_h}{2} = \frac{w/2}{f} = \frac{w}{2f},
\]
so that
\[
w = 2f \tan \frac{\theta_h}{2}.
\]
The corresponding formula for the height of the view-rectangle is
\[
h = 2f \tan \frac{\theta_v}{2}.
\]

The vector from \( Q \) to the right side of the far clipping rectangle is \( (w/2)u = f \tan(\frac{\theta_h}{2})u \); the vector from \( Q \) to the top is \( (h/2)v = f \tan(\frac{\theta_v}{2})v \). And the vector from \( P \) to \( Q \) is \( fw \).

To transform our view frustum to the standard perspective view volume, we must send \( P \) to the origin, \( f \tan(\frac{\theta_h}{2})u \) to \( e_1 \), \( f \tan(\frac{\theta_v}{2})v \) to \( e_2 \), and \( fw \) to \( e_3 \). We can use our `PointAndVectorsToPointAndVectors` method to generate a matrix that effects such a transform. Alternatively, we can recognize that because \( u, v, w \) are mutually orthogonal and are all unit length, the matrix whose columns are \( u, v, \) and \( w \),
\[
M = [u; v; w],
\]
has the property that \( M^T M = I \), i.e., \( M^T \) sends \( u \) to \( e_1 \), \( v \) to \( e_2 \), and \( w \) to \( e_3 \), which is almost exactly what we’re looking for. By introducing a
diagonal matrix to scale up or down the results, we see that the matrix that transforms the vectors as desired is

$$N = \begin{bmatrix} \frac{1}{f \tan \frac{\theta_h}{2}} & \frac{1}{f \tan \frac{\theta_h}{2}} & \frac{1}{f} \\ \end{bmatrix} M^T.$$  

**Inline Exercise 13.4:** Verify, using the properties of $M$, that the matrix $N$ sends $f \tan(\frac{\theta_h}{2})u$ to $e_1$ and $fw$ to $e_3$.

Of course, $N$ represents only the vector part of the transformation; to send the point $P = (P_x, P_y, P_z)$ to the origin as well, we must place $N$ in the upper-left $3 \times 3$ portion of a $4 \times 4$ matrix thus:

$$M_{\text{perspective}} = \begin{bmatrix} N & -P_x \\ -P_y & -P_z \\ 0 & 0 & 0 & 1 \\ \end{bmatrix}.$$  

### 13.2.2 Pseudocode and remarks

We'll now pull together the entire sequence of steps into a single piece of pseudocode, where we've assumed the existence of a $4 \times 4$ matrix class, with various operations like indexing and matrix multiplication being defined, starting from the view specification and producing the matrix transformation that takes the specified view volume and maps it to the standard perspective view volume (see Figure 1.15).
function mat44 PerspectiveCameraTransformationMatrix(
    Point3 P, // camera pinhole location
    Vector3 LookDirection, // the gaze direction
    Vector3 UpDirection, // (see text)
    double theta_h, // horizontal field of view in degrees
    double theta_v, // vertical field of view in degrees
    double n, // near clipping plane distance
    double f) // far clipping plane distance
{
    theta_h = theta_h * Math.PI/180;
    theta_v = theta_v * Math.PI/180;
    Vector3 w = -LookDirection / Length(LookDirection);
    Vector3 v = UpDirection - Dot(UpDirection, w) * w;
    v = v / Length(v);
    Vector3 u = Cross(v, w);

    // Version 1
    return Transform3.PointAndVectorsToPointAndVectors(
        P, f*Math.Tan(theta_h/2)*u, f*Math.Tan(theta_v/2)*v, f*w,
        new Point3(0,0,0), new Vector3(1,0,0), new Vector3(0,1,0)
            new Vector3(0, 0, 1));

    // Version 2
    Mat44 M = new Mat44();
    for (int i = 0; i < 3; i++)
    { M[0, i] = u[i]; M[1, i] = v[i]; M[2, i] = w[i]; M[i, 3] = -P[i]; }
    M[3,3] = 1;
    mat44 D = Mat44.Diag(1/(f*Math.Tan(theta_h/2)), 1/(f*Math.Tan(theta_v/2)),
        1/f, 1);
    return D * M;
}

Figure 13.12: Two versions of the code for transforming a view specification into a matrix that takes the specified view volume into the standard perspective view volume.
Exercise 13.1: The second version of the code given in Figure 1.15 does not exactly match the development in the text: there's no $3 \times 3$ matrix built and inserted into a larger $4 \times 4$ matrix, for example. Explain why the code produces the same matrix as the one described in the text.

Coordinate choices

We chose the standard perspective view volume so that as seen from the eye-point (the origin), $x$ increases to the right, $y$ increased as we move up, and $z$ increases as we move from faraway objects (large negative $z$-values) to nearby objects (small negative $z$-values). These choices describe a right-handed coordinate system.

We can think of various camera operations in terms of this standard view volume. Camera roll, for instance, would be a positive rotation about the $z$-axis, i.e., a positive rotation in the $xy$-plane, rotating $+x$ towards $+y$, i.e., rotating the camera counterclockwise as we held it in front of you. Camera yaw is a positive rotation about $y$ (rotating $+x$ towards $+z$); if you were holding the camera in your hands and looking through the viewfinder, you’d yaw by pulling with your right hand and pushing with your left. Camera pitch is a positive rotation about the $x$-axis.

Inline Exercise 13.5: Describe how you’d adjust camera pitch with a physical camera, in analogy with our descriptions of roll and yaw.

In some systems, during the “clipping” phase of the viewing process, the $z$-axis is flipped, so that the view volume lies between $z = 0$ and $z = 1$, instead of $z = -1$. This can lead to considerable confusion about the meaning of “less than” and “greater than” when certain data is kept or rejected based on a test involving $z$-coordinates.

The near plane

Under the transformation we just computed, the camera position (or eye-point, or pinhole) is sent to the origin, hence has $z = 0$; the far plane is sent to the $z = -1$ plane. The near plane is also sent to some $z = c$ plane; because it’s $n/f$ of the way from the eye to the far plane before transformation, it must be $n/f$ of the way from the eye to the far plane after transformation as well. Thus the near plane is transformed to the plane $z = -n/f$. 
Exercise 13.2: In our discussion of the transformation of the near plane, we claimed that if the plane $F$ is at a distance $f > 0$ from the point $P$ (distance measured along the direction normal to $F$), and if the plane $N$ is at distance $n > 0$ from $P$, and parallel to $F$, and on the same side of $P$ as $F$, then for a nondegenerate affine transformation $T$, $T(N)$ is on the same side of $T(P)$ as $T(F)$, and the distance from $T(P)$ to $T(N)$ is $n/f$ of the distance from $T(P)$ to $T(F)$. Let $w$ denote the unit normal vector to $F$ that points to the half-space not containing $P$. Then $P_N = P + nw$ lies on the near plane $N$, and $P_F = P + fw$ lies on the far plane. Use these points to test the claim that the distances of the transformed planes from $T(P)$ have the claimed ratio.

Coordinates

If $Q$ is a point in the view frustum, and $T$ is the transformation that takes the view frustum to the standard perspective view volume, then $T(Q) = (a, b, c)$ is a point in the standard perspective view volume. The coordinates of $T(Q)$ tell us how much $Q$ differed from $P$, but these differences are expressed in terms of $u$, $v$, and $w$. In particular, we know that

$$Q = P + a(\tan(\theta_h/2)u) + b(\tan(\theta_v/2)v) + c(fw).$$

Thus $a$, $b$, and $c$ can be see as the coordinates of $Q$ in a coordinates system whose origin is $P$ and whose basis vectors are $\tan(\theta_h/2)u$, $\tan(\theta_v/2)v$, and $fw$. We thus sometimes refer to $a$, $b$, and $c$ as the camera coordinates of $Q$. Sometimes you'll see the numbers $a\tan(\theta_h/2)$, $b\tan(\theta_v/2)$ and $cf$ also called the camera coordinates, so that the coordinates are expressed with respect to an orthonormal coordinate system. It's important to know which kind of camera coordinates someone's referring to. (Note that in both cases, the third (camera) coordinate of a point in the view frustum is negative; occasionally the camera coordinates will use $wvec$ instead of $w$ so that all three camera coordinates are positive; once again, you need to know what coordinate system is in use before you can make sense of the meaning of coordinates.)

Some tasks are a great deal easier in camera coordinates than in world coordinates — the $xyz$-coordinates in which we describe the camera’s location, etc. For instance, it’s easy to tell whether one point in the view frustum can possibly obscure another: it can only do so if (in camera coordinates) its $z$-coordinate is closer to zero (i.e., it’s on a plane closer to the eye).

13.3 The parallel case

We’ll now put aside the perspective camera for a moment and discuss an orthographic camera (represented in WPF by the OrthographicCamera class). An orthographic camera is an abstraction, and does not correspond
Figure 13.13: An orthographic camera. Instead of a view pyramid, we have an infinite rectangular parallelepiped; two clipping planes again define a portion of interest. Once again there's a \texttt{LookDirection} and an \texttt{UpDirection}. And once again there is a \texttt{Position} (labelled $P$); its use in the orthographic camera is to serve as a reference point for defining the near and far clipping planes by their distances from the \texttt{Position} along the \texttt{LookDirection}, and to define the location of the infinite box: the \texttt{Position} lies at the center of one of the box's cross-sections. Finally, there are the width and height of the parallelepiped.

To any physical camera. In an orthographic camera, we trace rays from objects onto a projection plane, with all the rays being perpendicular to the projection plane, and hence parallel rather than meeting at a point as they did in the pinhole camera (see figure ??).

To specify an orthographic camera, we use information similar to that of the perspective camera:

- A \texttt{Position}, used as a point of reference for defining the near and far clipping planes and the sides of the box;
- A nonzero vector called the \texttt{LookDirection}; the edges of the infinite box are all parallel to this vector;
- A nonzero vector \texttt{UpDirection}, which must not be parallel to the \texttt{LookDirection}; the projection of the \texttt{UpDirection} onto the plane perpendicular to the \texttt{LookDirection} gives the “vertical” direction on the projection plane;
- The distances $n$ and $f$ from the \texttt{Position} to the near and far clipping planes, measured in the \texttt{LookDirection}; and
- The width and height of the view volume.

Once again, we can (with almost exactly the same code) find the affine transformation that sends the view frustum to a standard view volume, in
Figure 13.14: The standard parallel view volume; like the standard perspective view-volume, the parallel volume is in the negative-$z$ half-space, and ranges from $-1$ to $1$ in both $x$ and $y$.

this case the box defined by $-1 \leq x, y \leq 1$ and $0 \geq z \geq -1$, which we call the “standard parallel view volume” (see figure 1.14).

Note that in this case, the point sent to the origin is the center of the near clipping plane, $P + nw$, and that the vector that’s sent to $e_3$ is the one that goes from center of the near clipping plane to the center of the far clipping plane, i.e., $(f - n)w$.

Exercise 13.3: Write a second version of OrthographicCameraTransformationMatrix that’s analogous to the second version of PerspectiveCameraTransformationMatrix.

Exercise 13.4: In computing the orthographic camera transformation in Figure 1.15, suppose that we’d chosen to let $w$ be a unit vector in the same directions as LookDirection, but had altered the code so that $u$ and $v$ remained unchanged. We could then have built a transformation to send $(f - n)w$ to $-e_3$. That is to say, we could have send a left-handed coordinate system to a left-handed version of the standard coordinate frame (i.e., one in which $-e_3$ replaces $e_3$. Show that although the coordinate frames are changed, the resulting transformation matrix remains exactly the same.
function mat44 OrthographicCameraTransformationMatrix(
    Point3 P, // camera location
    Vector3 LookDirection, // the gaze direction
    Vector3 UpDirection, // (see text)
    double width, // horizontal field of view, in degrees
    double height, // vertical field of view, in degrees
    double n, // near clipping plane distance
    double f) // far clipping plane distance
{
    Vector3 w = -LookDirection / Length(LookDirection);
    Vector3 v = UpDirection - Dot(UpDirection, w) * w;
    v = v / Length(v);
    Vector3 u = Cross(v, w);
    return Transform3.PointAndVectorsToPointAndVectors(
        P + n*w, (width/2.0)*u, (height/2.0)*v, (f-n)*w,
        new Point3(0,0,0), new Vector3(1,0,0), new Vector3(0,1,0),
        new Vector3(0,0,1));
}

Figure 13.15: The first version of the code for transforming an orthographic camera view specification into a matrix that takes the specified view volume into the standard parallel view volume.

code:persp-view

13.4 Projection and mapping to a viewport: the parallel case

Having transformed our orthographic view volume to the standard parallel view volume, it’s easy to project points from this standard view volume along the standard direction of projection (the $z$-direction): we simply eliminate the $z$ coordinate (i.e., multiply by $M_{\text{proj}}$). The result (starting with a point that’s in the orthographic view volume) is a pair of coordinates that lie between $-1$ and 1. If the starting point is outside the specified view-volume, then either (a) the transformed $z$-coordinate lies outside the range $-1 \leq z \leq 0$, or (b) the $x$ or $y$ coordinate is greater than 1 or less than $-1$.

Typically we want to produce $xy$-coordinates that are suitable for use in our 2D graphics system; in WPF, these are specified in a Viewport3D, which is a 2D element in which 3D renderings are displayed. It has a width and height, and coordinates run from 0 to width and from 0 to height. To go from the standard parallel view volume to the viewport, we need to both project and to perform a windowing transformation that takes the $-1 \leq x,y \leq 1$ rectangle to the viewport. The matrix that does the projection
is simply
\[
M_{\text{proj}} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix};
\] (13.4)

while the windowing transform is
\[
M_{\text{wind}} = \begin{bmatrix}
\text{width} & 0 & 0 & 0 \\
0 & \text{height} & 0 & 0 \\
0 & 0.5 & 0.5 & 0
\end{bmatrix};
\] (13.5)

The entire sequence of transformations converts the homogeneous-coordinate quadruple of a point in world-space into a pair of WPF coordinates in the viewport.

Of course, it's important that the width and height of the viewport be proportional to the width and height of the view volume; otherwise the objects in the view volume might appear stretched or compressed vertically in the final rendering. Because of this, in WPF the OrthographicCamera lets the user specify only the \textit{width} of the view volume; the height of the view volume is computed to ensure that the aspect ratio of the view volume matches that of the viewport (i.e., internally it would set \textit{height} = \textit{width}/\textit{ViewportAspectRatio}).

### 13.5 Converting the perspective case to the parallel case

We've now seen how to transform objects within a parallel view volume to the standard parallel view volume, and from there project them to the \(xy\)-plane and viewport-map them into a specified viewport. For the perspective case, however, we stopped at the standard perspective view volume. We'll now complete the perspective case by reducing it to a previously solved problem: we'll transform the portion of the standard perspective view volume between the transformed near- and far-planes (i.e., the portion between \(z = -n/f\) and \(z = -1\)) to the standard parallel view volume; we can then apply the remainder of the work we did in the parallel case. The transformation use will be a \textit{projective} transformation in which all the rays passing from the view volume towards the origin are transformed into rays passing from the view volume towards the \(xy\)-plane in the positive-\(z\) direction (see Figure 1.16).

This works because the perspective view of a shape in the pre-transformed volume is the same as the parallel view of the transformed shape in the post-transformed volume. This is easy to see if we look at a two-dimensional slice of the situation, just the \(yz\)-plane. Consider, for instance, the small square shown in Figure 1.17 occupies the middle half of a perspective view of the scene. Occlusion (which points are obscured by others) is determined by the ordering of points along rays from the viewpoint into the scene, so
that the point $B$ is obscured by the near edge of the square. After transformation, that ray from the viewpoint into the scene becomes a ray in the $-z$ direction; once again, the point $B'$ is obscured by the front edge of the square. And once again, the transformed square ends up filling the middle half of the parallel view of the scene. The essential underlying fact is that light (the underlying agent of vision) travels in straight lines, and the transformation we’re using converts straight lines to straight lines (and in particular, the projection rays from the perspective view to projection rays for the parallel view).

To compute the actual transformation, we first consider a special case: suppose that the transformed front-clipping plane is at $z = -1/2$. Because of the symmetry in the situation, we can examine the $yz$-plane only; the $x$-coordinate will transform in the same way as the $y$-coordinate. Recall from chapter ?? that a projective transformation from $\mathbb{R}^2$ to $\mathbb{E}^2$ is completely determined by where it sends four points, no three of which are collinear. We want to send the point $(y, z) = (1, -1)$ to itself, and $(y, z) = (-1, 1)$ to itself (i.e., have the far-clipping plane stay at $z = -1$). We also want to send the near clipping plane to the $z = 0$ plane; in particular, we’ll send the point $(y, z) = (1/2, 1/2)$ to $(1, 0)$ and $(-1/2, 1/2)$ to $(-1, 0)$. The matrix
Figure 13.17: The standard perspective view volume at left (with a near clipping plane at $z = -1/4$) contains a small square. The point $A$ of the square is transformed to the point $A'$ of the distorted square in the standard parallel view volume. The same goes for the points $B$ and $B'$. And just as $B$ appears lower than $A$ in the perspective view of this “scene” (and is obscured from the viewpoint by the front face of the square), the point $B'$ ends up below the point $A'$ and is obscured from the projection plane by the front edge of the (distorted) square.

The linear transformation represented by this transformation (expressed in $yzw$-coordinates) is

$$M_{pp} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & -1 & 0 \end{bmatrix}.$$ 

The corresponding matrix, when we include the $x$-coordinate, is

$$M_{pp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

The linear transformation represented by this matrix sends the point $(x, y, z)$ to

$$M_{pp} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 2z + 1 \\ -z \end{bmatrix}.$$ 

(13.6)
To complete the *projective* transformation, we have to divide through by the last coordinate \((-z)\) to get the map

\[
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
-x/z \\
-y/z \\
(-2z - 1)/z \\
1
\end{bmatrix} = \begin{bmatrix}
-x/z \\
-y/z \\
-2 - 1/z \\
1
\end{bmatrix}.
\]

**Inline Exercise 13.6:** Confirm that this really does send the specified \((y, z)\) to the desired locations.

The matrix for the \(yz\)-transformation we derived is the very one we discussed in section ??.

Examining the derivation of this matrix, it’s easy to alter it to handle the general case. The new constraints are

- the point \((y, z) = (c, c)\) must map to \((-1, 0)\) (recall that \(c\) is negative!)
- the point \((-c, c)\) must map to \((1, 0)\)
- Any point in the \(z = -1\) plane must remain fixed; in particular, \((0, -1) \rightarrow (0, -1)\), and
- \((1, -1) \rightarrow (1, -1)\).

The resulting new matrix is

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1/(1 + c) & -c/(1 + c) & 0 \\
0 & -1 & 0 & 0
\end{bmatrix},
\]

or, in the \(xyzw\)-case,

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1/(1 + c) & -c/(1 + c) & 0 \\
0 & -1 & 0 & 0
\end{bmatrix}.
\]

(13.8)

So the entire sequence of transformations (from “world space” to viewport) becomes (Figure 1.18 Figure 1.22)

- Multiply by \(M_{\text{perspective}}\), from Equation 1.3;
- Multiply by \(M_{\text{pp}}\) (using \(c = -n/f\)) from Equation 1.7, to transform from the standard perspective view volume to the standard parallel view volume;
- Apply the homogenizing transformation \(H(x, y, z, w) = (x/w, y/w, z/w, 1)\);
- Multiply by \(M_{\text{proj}}\) (Equation 1.4) to project to the \(xy\)-plane; and finally
Multiply by $M_{\text{wind}}$ (Equation 1.5) to transform to window or viewport coordinates.

Note that for the sake of efficiency, the first three matrices can be multiplied together, as can the last two; transforming a point then amounts to two matrix-multiplies and one homogenization.

This section needs a comment about factoring out the homogenization operation, etc.

13.5.1 Observations

Aside from the projection to $xy$, this sequence of transformations is generally invertible. With a little more work, we can make this precise. [FINISH THIS]

**Inline Exercise 13.7:** (a) Write down a matrix $M'_{\text{pp}}$ for the linear transformation $(x, y, z, w) \mapsto (x, y, 0, w)$. (b) Write down a $4 \times 4$ matrix $M'_{\text{wind}}$ that transforms $x$ and $y$ just as does $M_{\text{wind}}$ from Equation 1.5, but which acts on homogeneous coordinates in 3-space rather than in 2-space; it should act on $z$ by leaving it unchanged. (c) Verify that these two matrices commute, i.e., that when they are multiplied, the result doesn't depend on the order in which they are multiplied.

With the results of this exercise, we can see that instead of performing the sequence as described, we could instead do the first three steps, then the fifth, and then the fourth.

13.5.2 Practical considerations: why not make $n = 0$ and $f = \infty$?

The choice of the near and far clipping planes can have a substantial impact on the images computed from a camera specification, even if there are no objects in the scene that are closer than the near-plane or farther than the far-plane. At the level of mathematics, changes in the near- and far-plane distances in such situations should have no effect, but in practice, where we use finite-precision representations of numbers, there can be important consequences. To see how this can happen, we’ll consider a couple of special cases.

Let’s re-examine the situation in Figure ??, where we saw a rectangle in the standard perspective view volume become a quadrilateral in the standard parallel view volume. In that case, the quadrilateral was slightly lengthened in the $z$-direction. But this is by no means always the case. Consider a very similar situation shown in Figure 1.23, in which there are now several squares, and several possible near-far clipping plane pairs to consider.

It’s not hard to work out the entire sequence of transformations in this case (or one can determine them directly by asking where the four corners
Figure 13.23: A 2D scene consisting of several squares along the $z$-axis; we show three different front clipping planes, at $z = -1/4$ (so $n_1 = 1/4$), $z = -1/10$ ($n_2 = 1/10$), and $z = -1/50$ ($n_3 = 1/50$), and consider three different far clipping planes, at $z = -1$ (so $f_1 = 1$), $z = -4$ ($f_2 = 4$), and $z = -400$ ($f_3 = 400$), of which only the first two are shown. The field-of-view half-angle in all cases is 45 degrees.

of the view frustum must be sent); it turns out to be

\[ y \mapsto -\frac{y}{z} \]

(13.9)

\[ z \mapsto f \frac{z + n}{(n - f)z} \]

(13.10)

for front and back clipping distances $f$ and $n$.

The results, for each of our three sets of clipping planes, are shown in Figure 1.24. You can see that as the clipping volume is extended, there’s more and more severe compression along $z$. Since many visibility approaches depend on sorting in $z$ (after transformation), this is a problem: with finite resolution in our representation of numbers, it’s easy to have objects at different depths end up with the same (discretized) transformed $z$-values, and hence be unsortable (i.e., we can no longer determine which one is in front).

Suppose for the moment that we keep $n$ fixed, and we have some fixed set of objects (the rectangles in our example) that we wish to transform. What happens to

\[ f \frac{z + n}{(n - f)z} \]

for a fixed $z$ and $n$, as $f$ grows larger? For very large values of $f$, $n$ becomes
Figure 13.24: The results of transforming each of the three view volumes to the standard parallel view volume. Notice how as the near and far distances approach zero and infinity, respectively, the objects in the scene get more and more compressed along the $z$-axis, after transformation.


 negligible, and the expression is approximately

$$\approx f \frac{z}{(−f)z} = −1,$$

In other words, almost every object, after transformation, is squeezed against the back plane of the standard parallel view volume. You might have expected the opposite: if you doubled $f$, you might expect that “all the stuff between the old $f$ and the new one would transform to the back half of the view volume, while the stuff we had already would end up in the front half,” but that’s not true at all. In fact, if we look at the point $z = −f$, it’s transformed (when we set the rear clipping plane to $z = −2f$) to

$$2f \frac{−f + n}{(n − 2f)(−f)} = 2 \frac{n − f}{(2f − n)};$$

once again, we observe that as $f$ gets large, this is very close to $z = −1$.

In short: if you let the rear clipping plane move away to infinity, your post-transform $z$ values will all end up clustered very near to -1; discretization of the values will cause occlusion information to be lost.

There’s a possible way to work around this problem. Suppose that instead of having the standard parallel view volume extend from 0 to $−1$ in $z$, we had it extend from 1 to 0 in $z$ (i.e., we simply added one to each transformed $z$-value). Then although most transformed values would cluster near zero, there would not be as large a problem, because if we store them
Figure 13.25: Two polygons at different depths can end up with the same $z$-value (after transformation to a parallel view volume, and discretization); when this happens, some pixels may be mis-sorted, so that the back polygon shows through the front one; this often happens in stripes, as shown here.

as floating point numbers, there are many more floating point numbers near zero than near one. This does in fact improves matters somewhat.

An alternative desire is to not clip out anything close to the camera; after all, if something is right in front of your face, you definitely see it! So people are sometimes tempted to let $n$ be a very small value. If we let $n$ go to zero, the resulting matrix is again the one given in Equation ??; you can verify this directly, or you can observe that this matrix depends only on $c = -n/f$, and that as $f$ goes to infinity or $n$ goes to zero, this ratio tends to the same value (namely zero).

To summarize: setting $n$ to a very small value or $f$ to a very large value will cause all the transformed $z$-values to be very nearly equal; visibility-testing methods that rely on discretized versions of these $z$-values will tend to fail badly.

One common artifact of this discretization error is that a polygon that's supposed to be in front of another will sometimes not be recognized as in front. The failure to sort properly tends to have a certain spatial coherence (i.e., all the pixels in one area of the polygon will sort incorrectly, those in another will sort correctly; and so on). The phenomenon is known as $z$-fighting, and its visual manifestation, shown in Figure 1.25, sometimes has a striped appearance.
Figure 13.26: A more general perspective view volume specification: in addition to the pinhole location \( P \) and the look- and up-directions, we specify a view-plane distance \( d \) along the \( \text{LookDirection} \), and then specify a view rectangle in \( uv \)-coordinates on that view-plane.

13.6 Generalized cameras

As we mentioned earlier, there are more general camera specifications than the ones predefined by WPF. In particular, a view camera allows one to select any rectangle on the imaging plane rather than requiring that the view rectangle be centered around the “gaze line” defined by the position and \( \text{LookDirection} \). It’s easy to experience this by standing in front of a wall with a window in it, facing the window. You see through the center of the window (if you’re the right height) to the world outside. You could even trace, on the window panes, what you see, thus “rendering” a 2D image. This corresponds to WPF’s \text{PerspectiveCamera}. But if you bend your knees and look out the window, or move one step to the right and look out the window, and again draw on the glass to make a rendering, you have a kind of view that’s not available from the \text{PerspectiveCamera}, because the point that’s nearest to you on the imaging plane is not the center of the window. You can use this closest point (sometimes called the \textit{view reference point, and denoted } \text{VRP}) to define the window, using the vectors \( u \) and \( v \): you can specify numbers \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, \text{ and } v_{\text{max}} \) and use these to say that the window extends from \( \text{VRP} + u_{\text{min}}u + v_{\text{min}}v \) to \( \text{VRP} + u_{\text{max}}u + v_{\text{max}}v \) in the imaging plane, as shown in Figure 1.26. Of course, to do so unambiguously, you also need to specify how far away the imaging plane actually is. This view-plane distance, and the bounds on \( u \) and \( v \), (in addition to all our prior view-specification data) suffice to determine the most general possible pinhole perspective camera.

To find a transformation matrix for this more general view volume to the standard perspective view volume follows the same pattern as before, except that:
• Instead of specifying that \( f w \) is sent to \(-e_3\), we compute the vector from \( P \) to the center of the far rectangle of the view frustum and send \textit{that} to \(-e_3\), and

• There’s no need to compute the width and height from the field-of-view angles, since they’re determined by \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, \text{ and } v_{\text{max}} \); one must, however, do the geometry to compute the width and height of the far rectangle of the view frustum from the specified rectangle on the view-plane.

In the case of parallel projections, the situation is similar: again we specify an imaging plane and a rectangle on that plane, but the projection along parallel lines onto that plane may be along parallel lines in \textit{any} direction, \( d \), and not just in the direction of the view-plane normal vector. Thus a more general \textit{parallel camera} can be specified by

• A point \( P \) on the imaging plane;

• The unit normal vector, \( w \), to the imaging plane, pointing towards the camera;

• An up-direction vector \( k \), from which we compute a vector \( v = k - (k \cdot w)w \), which we then normalize, and the vector \( u = v \times w \);

• The limits of the view rectangle \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, v_{\text{max}} \); and

• A unit direction-of-projection vector \( d \),

as shown in Figure ??.

Again, the computations are quite similar to those in the case of the orthographic camera. Because parallel, but non-orthographic, views of objects are fairly commonplace — the “cavalier”, “cabinet”, “dimetric,” “trimetric,” and “isometric” views are all common in mechanical drawing (see [?]) — we’ll consider one example in detail.

### 13.6.1 An example: the cabinet projection

The “cabinet projection” shown in Figure 1.28, takes an object whose faces are aligned with the \( xy \), \( yz \), and \( zx \)-planes and renders it so that lengths in the \( xy \)-plane are preserved, while lengths along the \( z \)-axis are shortened by a factor of two.

What direction of projection \( d \) will give this shortening-by-two property? We can find out by examining a single vector: the projection of \( e_3 \) in the \( d \) direction onto the \( xy \)-plane must have length \( 1/2 \). Any projection of \( e_3 \) along the \( d \) direction has the form

\[
e_3 + td
\]

for some value of \( t \). The constraint that this point lie on the \( xy \)-plane means that the third coordinate must be zero, i.e., that

\[
e_3 \cdot (e_3 + td) = 0.
\]
Figure 13.27: A more general parallel camera. We have a view reference point $P$, on the imaging plane, the view-plane normal $w$, the up-vector $k$ (which gives rise to the vectors $u$ and $v$). We also have the direction of projection $d$; rays in direction $d$ leave objects and meet the imaging plane to determine the imaging point. (We’ve omitted the near and far clipping planes to simplify the figure.)

Figure 13.28: A cabinet projection of a desk whose right-hand side is square. Lengths in the $xy$-plane (the plane of the front of the desk) are preserved, while those in the $z$-direction are reduced by one-half. If the projection were along the $z$-axis, we’d see only the front of the desk; instead, the projection is along a ray that’s about 27 degrees from the $z$-axis, which results in the view we see.
Letting $d_z$ denote the $z$-coordinate of $d$, this simplifies to $t = -1/d_z$.

**Inline Exercise 13.8**: Verify that the expression given for $t$ is correct.

In addition, this point must be at distance one-half from the origin, i.e., the vector $e_3 - (1/d_z)d$ must have length one-half. In coordinates, this says that

$$(-d_x/d_z)^2 + (-d_y/d_z)^2 = (1/2)^2,$$

i.e., that

$$\frac{d_x^2 + d_y^2}{d_z^2} = \frac{1}{4}.$$

Because we're working with a unit vector $d$ as our direction of projection, we have $d_x^2 + d_y^2 + d_z^2 = 1$, we can simplify to

$$1 - \frac{d_z^2}{d_z^2} = \frac{1}{4},$$

whose solutions are $d_z = \pm 2/\sqrt{5}$. That means that the dot product of $d$ with $e_3$ is $2/\sqrt{5} \approx 0.894$, which is the cosine of 26.6 degrees. Hence any direction of projection that's about 27 degrees from the $e_3$ direction will produce a cabinet projection. Picking

$$d = \frac{1}{\sqrt{5}} \begin{bmatrix} -4/5 \\ 3/5 \\ 2 \end{bmatrix}$$

gives a projection in which $e_3$ projects in a direction that's parallel to neither the $x$- or $y$-axis. We've chosen to make the $x$-coordinate larger because that matches conventional drawing styles; with this value of $d$, the projection of $e_3$ is

$$e_3 - (1/d_z)d = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} -4/10 \\ 3/10 \\ 1 \end{bmatrix} = \begin{bmatrix} -4/10 \\ 3/10 \\ 0 \end{bmatrix},$$

so $-e_3$ projects to a point in the fourth quadrant, as is conventional.

With this as our chosen vector $d$, let's work out the transformation to the standard parallel view volume. We'll choose $P$ to be the origin, and a LookDirection of $-e_3$, with the near plane at 0 and the far plane at 1. We'll pick the UpDirection to be $e_2$. While these choices simplify our computations, they're also quite typical. We'll choose a width and height of 2. These choices ensure that points on the $z = 0$ coordinate plane will transform to themselves, so that distances in the $xy$-plane will be preserved. Knowing that $P$, $e_1$, and $e_2$ must be sent to themselves, and that some multiple of $d$ must be sent to $e_3$, we could use the PointAndVectorsToPointAndVectors method to determine the matrix. But it's easy to do it directly as well.
Since we know that the origin, $e_1$, and $e_2$ must be sent to themselves, the matrix must have the form

$$\begin{bmatrix}
1 & 0 & a & 0 \\
0 & 1 & b & 0 \\
0 & 0 & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.$$ 

This maps the entire near plane to itself. What about the far plane? The point at $P - (1/d_z)d$ must be sent to the center point of the far plane of the standard parallel view volume, i.e. $P + e_3$, so $-(1/d_z)d$ must be sent to $e_3$, i.e. we must have

$$\begin{bmatrix}
1 & 0 & a & 0 \\
0 & 1 & b & 0 \\
0 & 0 & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
-4/10 \\
3/10 \\
1 \\
0
\end{bmatrix} = \begin{bmatrix}
n0 \\
0 \\
1 \\
0
\end{bmatrix}$$

which implies that $a = 4/10$, $b = -3/10$, and $c = 0$, so that the final transformation is

$$\begin{bmatrix}
1 & 0 & 4/10 & 0 \\
0 & 1 & -3/10 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.$$ 

Other kinds of parallel projection can be determined similarly; the details are given in the paper by Carlbom and Paciorek [2].

### 13.7 Real-world cameras

Real-world cameras do not use a pinhole, in part because the energy passing through an ideal infinitesimal pinhole is zero; it would be impossible for the light to affect a sensor. On the other hand, when the pinhole is enlarged, light from a point source no longer creates a single illuminated point on the image plane (Figure ??).

Modern cameras use lenses to address this problem; a lens has the characteristic that light from a source at a certain distance can hit many points of the lens (see figure 1.30) and all be bent to hit a single point on the imaging plane. Unfortunately, light from sources at different distances will converge either in front of or behind the imaging plane, so objects at those distances appear out of focus. Other problems arise as well: light of different wavelengths gets focused differently, too, for instance, and reflected from the lens-surface in different ways. Correct modeling of lens effects can be incorporated into rendering of scenes [?], but is beyond the scope of this discussion of virtual cameras.
Figure 13.29: A camera with a pinhole of nonzero radius. Light from a point source no longer appears on the image plane as a single illuminated point, but rather as an illuminated disk. The entire image formed on the image plane is blurry compared to the one produced by the idealized pinhole camera.

Figure 13.30: A lens is designed so that light at some distance can hit the whole lens and be focused back to a single point. Light further away will be focused to a further point. Thus a lens in a camera causes objects at certain distances to be in focus on the imaging plane, while others appear blurry.
13.8 Putting it all together

At this point, we've described in detail how cameras are specified in graphics systems; you have almost all the tools needed to go from a geometric description of a scene and a view of that scene to a basic rendering of the scene from that view. This “basic rendering” consists of transforming every point of the scene to the imaging plane, regardless of whether it's obscured by some other object; if we apply this to vertexes and edges of objects represented as triangle meshes, the result is a wire-frame rendering of the scene, as we saw in Chapter 3. The steps are these:

- Convert the view specification into a pair of matrix transformations with matrices $M_{\text{perspective}}$, transforming the view frustum to the standard view volume, and $M_{\text{pp}}$, transforming part of the standard perspective view volume to the standard parallel view volume $-1 \leq x, y \leq 1, 0 \geq z \geq 1$.

- For each vertex $P_i$ of each object, compute $P'_i = T_{M_{\text{perspective}}}(P_i)$.

- For each edge $ij$ from vertex $i$ to vertex $j$
  - “Clip” the edge that goes from $P'_i$ to $P'_j$ so that only the part within the standard view frustum remains.
  - Transform the endpoints of this clipped edge by the matrix $M_{\text{pp}}$.
  - Project the resulting points to $xy$-coordinates (by ignoring $z$).
  - Transform the resulting $xy$-points to viewport coordinates via the windowing transform, with matrix $M_{\text{wind}}$.
  - Draw the resulting line segment in screen coordinates.

If you re-examine the code in Listing 3.4 for the Dürer etching of Chapter 3, you'll see that it follows these steps quite closely, omitting the clipping step. In particular, the vertexes of the model are transformed by this piece of code:

```csharp
double scale = 100;
for (int i = 0; i < nPoints; i++)
{
    double x = vtable[i, 0];
    double y = vtable[i, 1];
    double z = vtable[i, 2];
    double xprime = -x / z;
    double yprime = y / z;
    pictureVertexes[i].X = scale * (xprime - xmin) / (xmax - xmin);
    pictureVertexes[i].Y = scale * (yprime - ymin) / (ymax - ymin);
    gp.Children.Add(new Dot(pictureVertexes[i].X, pictureVertexes[i].Y));
}
```
This may not seem to follow the pattern of building matrices as described in the general formulation, but let’s examine it more carefully. In the Dürer etching, we chose coordinates so that the LookDirection is $-e_3$ and the camera Position $P$ is $(0,0,0)$. We chose the field of view so that at a distance of 1, we saw a square of width one; that makes both field of view angles $\theta = 2 \arctan(1/2)$, so that $\tan(\theta/2) = 1/2$. The up-vector $k$ is $e_2$. The only missing items in the standard virtual camera are the near- and far-plane distances, $n$ and $f$. As it turns out, neither of these will matter, so we'll take $n = 1$ and $f = 2$. Finally, the viewport extended from $(0,0)$ to $(100,100)$.

With these in hand, let’s walk through the steps above.

- Convert the view specification into a pair of matrix transformations with matrices $M_{\text{persp}}$, transforming the view frustum to the standard view volume, and $M_{\text{pp}}$, transforming part of the standard perspective view volume to the standard parallel view volume $-1 \leq x,y \leq 1, 0 \geq z \geq 1$.

To do this, we compute $w = e_3, v = k - (k \cdot w)w = e_2 - 0e_3 = e_2, u = w \times v = -e_1$. Then, since $f \tan(\theta/2) = 2 \cdot 1/2 = 1$, according to version 2 of the code in Figure 1.15, we have

$$
M_{\text{persp}} = \begin{bmatrix}
1 & 1 & -1/2 \\
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & -1/2 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

which sends

$$
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
-x \\
y \\
-z/2 \\
1
\end{bmatrix}.
$$

To compute $M_{\text{pp}}$, we note that $c = -n/f = -1/2$, so according to Equation 1.8,

$$
M_{\text{pp}} = \begin{bmatrix}
1 & 1 & -1 \\
2 & -1 & 0
\end{bmatrix},
$$

which sends

$$
\begin{bmatrix}
-x \\
y \\
-z/2 \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
-x \\
y \\
-(z + 1) \\
-z/2
\end{bmatrix}.
$$

- For each vertex $P_i$ of each object, compute $P'_i = T_{M_{\text{persp}}}(P_i)$. 
As noted, this step transforms
\[
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]
to
\[
\begin{bmatrix}
    -x \\
    y \\
    -z/2 \\
    1
\end{bmatrix}
\]
, which we do not see directly in the Dürer code.

Now for each edge we do the following:

• “Clip” the edge that goes from \(P'_i\) to \(P'_j\) so that only the part within the standard view frustum remains.

Because all of the edges in our Dürer model project within the viewport, we skipped this step.

• Transform the endpoints of this clipped edge by the matrix \(M_{pp}\).

We thus transform, by \(M_{pp}\), resulting in
\[
\begin{bmatrix}
    -x \\
    y \\
    -(z+1) \\
    z/2
\end{bmatrix}
\]
, which we then homogenize to \((-2x/z, 2y/z, -(z+1)/z)\).

• Project the resulting points to \(xy\)-coordinates (by ignoring \(z\)).

This gives us \((-2x/z, 2y/z)\).

• Transform the resulting \(xy\)-points to viewport coordinates via the windowing transform, with matrix \(M_{wind}\).

The windowing transformation is
\[
M_{wind} = \begin{bmatrix}
100 & 0 & 1/2 & 1/2 \\
0 & 100 & 0 & 1/2 & 1/2
\end{bmatrix}
\]

Applied to the point \((-2x/z, 2y/z, 1)\), we get
\[
(100(-x/z + 1/2), 100(-y/z + 1/2)).
\]

• Draw the resulting line segment in screen coordinates.

How does this compare to the Dürer code? In that code, \(xmin\) is \(-1/2\), \(xmax\) is \(1/2\), and \(scale\) is 100, so the assignments

\[
\begin{align*}
pictureVertexes[i].X &= scale \times (xprime - xmin) / (xmax - xmin); \\
pictureVertexes[i].Y &= scale \times (yprime - ymin) / (ymax - ymin);
\end{align*}
\]

amount to

\[
\begin{align*}
pictureVertexes[i].X &= 100 \times (xprime - 1/2) / (1); \\
pictureVertexes[i].Y &= scale \times (yprime - 1/2) / (1);
\end{align*}
\]
Remembering that \( x' = -x/z \) and \( y' = y/z \), respectively, we see that the two computations are identical.

In practice, rather than multiplying each vertex by \( M_{\text{persp}} \) and then by \( M_{pp} \), we would compute the product

\[
M_{pp} M_{\text{persp}} = \begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & -1 \\
0 & 0 & 0.5 & 0
\end{bmatrix}
\]

and multiply each point by this product. We would then homogenize the points.

Following this, we would project to the \( xy \)-plane and apply the windowing transformation, i.e., we’d multiply by the single matrix

\[
M_{\text{wind}} M_{xy} = \begin{bmatrix}
100 \\
100 \\
100 \\
100
\end{bmatrix}
\begin{bmatrix}
1/2 & 0 & 1/2 \\
0 & 1/2 & 1/2 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
50 & 0 & 0 & 50 \\
0 & 50 & 0 & 50
\end{bmatrix}.
\]

**Inline Exercise 13.9:** (a) Transform the point \((x, y, z)\) using the first product matrix; write your answer in terms of \(x, y,\) and \(z\). Then homogenize, and write your answer. Then transform the homogenized point by the second matrix, and write down your answer. The results should all be familiar to you. (b) Transform the point \((x, y, z)\) by the product of the two matrices; write out the homogeneous coordinates of the result in terms of \(x\) and \(y\); then homogenize. (c) Now show that in general, as long as we homogenize at the end of a sequence of projective transformations, we get the same result as if we homogenize after each one.

### 13.9 Projection and parameterization

The projective transformation \( T_{pp} \) that takes the standard perspective view volume to the standard parallel view volume generally transforms lines to lines. We’ve already mentioned that lines passing through the plane on which \( T_{pp} \) is undefined are transformed to lines missing a single point. Let’s now closely examine a particular instance of \( T_{pp} \), where the near-to-far ratio is \( c = 1/4 \), so that the \( z = -1/4 \) plane is transformed to the \( z = 0 \) plane.

We’ll look at just the \( yz \)-plane to make it easier to draw things. The matrix for this projective transformation is

\[
M = \begin{bmatrix}
1 & 0 & 0 \\
0 & 4/3 & 1/3 \\
0 & -1 & 0
\end{bmatrix}.
\]
When you multiply this matrix by \[
\begin{bmatrix}
y \\
z \\
1
\end{bmatrix},
\]
you get
\[
\begin{bmatrix}
y \\
\frac{1}{3}(4z + 1) \\
-z
\end{bmatrix}.
\]

The transformation $T_{pp}$, however, includes a homogenization step, which takes this to
\[
\begin{bmatrix}
-y/z \\
-\frac{4z+1}{3z} \\
1
\end{bmatrix}.
\]

Let’s take a diagonal line in the standard perspective view volume, and see how it transforms (see Figure 1.31). The endpoints of our diagonal are sent to two opposite corners of the standard parallel view volume. The midpoint $M$ is sent to a point $M'$ of the diagonal line between $A'$ and $B'$, but $M'$ is not the midpoint of the transformed segment. Generalizing, we see that taking equal steps before transforming is different, in general, from taking equal steps after transforming.

**Inline Exercise 13.10:** Show that if $T$ is a linear transformation, then linear interpolation of a greyscale value before or after transformations generates the same result. Part of the challenge in this exercise is to express this informal statement clearly; once you’ve done so, the result is obvious. This exercise shows that it’s the homogenizing part of the transformation to the standard parallel view volume that “ruins” linear interpolation.

Now imagine that you have some value — say a greyscale value — associated to each end of the original segment, and interpolated linearly over the segment. If the grey-value is 0.2 at $A$, and 0.8 at $B$, then it’s 0.5 at $M$. In many situations, it’s convenient to be able to compute this interpolated grey-value (or other values) after transformation, i.e., by knowing that the grey-value at $A'$ is 0.1 and at $B'$ is 0.9, we’d like to be able to determine that the grey-value at $M'$ is 0.5. But $M'$ is 4/5 of the way from $A'$ to $B'$; ordinary linear interpolation would assign it a value of
\[
0.2 + (4/5)(0.9 - 0.1) = 0.2 + .64 = .84.
\]

Evidently ordinary linear interpolation will not suffice.

There is a simple insight that lets us address this problem, however. Although (letting $G$ denote the greyscale value) the expression $G$ does not “interpolate linearly,” the expression $G/w$ does, where $w$ is the $w$-coordinate of a point after it’s multiplied by the matrix, but before it’s homogenized; in our case, the $w$-value is $-z$, although this approach works for any projective transformation, not just $T_{pp}$. So when we transform $A$ and $B$ to $A'$ and $B'$,
Figure 13.31: Within the standard perspective view volume with near-plane at \( z = -1/4 \), we draw the segment from \( A = (-1/4, -1/4) \) to \( B = (1, -1) \). The midpoint \( M \) is at \( (3/8, -5/8) \). Under the perspective-to-parallel transformation, this segment transforms to a new segment going from \( A' = (-1, 0) \) to \( B' = (1, -1) \). The midpoint, \( M' \), transforms to \( (3/5, -4/5) \), which is not the midpoint of the transformed segment.

We’ll retain a \( w \)-value associated with each. \( A = (-1/4, -1/4) \) transforms to the pair \((A'; w_A) = (-1, 0; 1/4)\), while \( B \) transforms to the pair \((B'; w_B) = (1, -1; 1)\). The value of \( 1/w \) at \( A' \) is 4, while at \( B' \) it’s 1; hence the value of \( G/w \) at \( A' \) is \( 4 \cdot 0.1 = 0.4 \), and at \( B' \) it’s \( 1 \cdot 0.9 = 0.9 \). If we interpolate linearly between these values to find the value of \( G/w \) at \( M' \) (which is \( 4/5 \) of the way from \( A' \) to \( B' \)), we get

\[
0.4 + \left( \frac{4}{5} \right)(0.9 - 0.4) = 0.8
\]

We can similarly interpolate \( 1/w \) linearly to get its value at \( M' \), namely

\[
4 + \left( \frac{4}{5} \right)(1 - 4) = 8/5.
\]

Since \( G/w \) at \( M' \) is \( 8/10 \), and \( 1/w \) is \( 8/5 \), we can compute the ratio \( (G/w)/(1/w) = G \) as \( (8/10)/(8/5) = 0.5 \).

Conclusion: If we know the values of a linear function \( f \) at the ends of a line segment \( AB \), and \( T \) is a projective transformation defined by a matrix \( M \), we can compute the value of \( f \circ T^{-1} \) at the point \((1 - t)A + tB\) by (a) computing \( w_A = \) homogeneous coordinate of \( MA \) and \( w_B = \) homogeneous coordinate of \( MB \), (b) computing \( w_t = (1 - t)(1/w_A) + t(1/w_B) \) and \( s = (1-t)f(A/w_A)+tf(B/w_B) \), and (c) computing \( s/w_t \), which is the interpolated value.

We’ll see, in Chapter 39, how this can be used in various ways in hardware. In particular, it’s often applied to the projective transformation that sends line-segments in world-space to line-segments in pixel-space. When
we want to compute interpolated colors, for instance, taking constant-distance steps from pixel to pixel along a line, this technique becomes essential.

13.10 Transformations and Clipping

One of the steps in the graphics pipeline is clipping, in which segments (or triangles) that extend outside the view frustum are truncated in some way before conversion to pixels. In the Dürer example, all segments fit comfortably inside the view frustum, so we ignored clipping. Let's consider it now. In Figure 1.32 you can see a scene consisting of many segments, as seen through a small viewport. To render a segment, without clipping, requires that we transform the endpoints of the segment as described in this chapter (a constant number of operations) and then compute values for the pixels that the segment meets—a number of operations that's proportional to the projected length of the segment. If a segment is very long, this “fill” operation dominates the computation cost. Imagine a segment that's a million pixels long, but which happens to not intersect the viewport at all; processing this segment wastes a huge amount of time. Suppose, instead, we first determine, for each segment, whether it intersects the viewport at all, and if it does, we shorten it to just the part that's visible in the viewport. Then the fill time is bounded by a constant multiple of the viewport dimensions: in a $1000 \times 1000$ pixel viewport, no (infinitely thin) line intersects more than 2000 pixels, for instance. By including this clipping step, we can gain an enormous speedup.

But what about a segment that's entirely contained within the viewport? The time spent clipping that segment is wasted. In trade for putting a bound on worst-case performance, we suffer somewhat in best-case performance. We'll discuss the details of how one clips a particular line-segment against a half-space (i.e., shorten the segment to consist of only the portion lying in the half-space) shortly; for now, assume that this operation is easy.

Clipping also addresses another important problem, one that arises only in the perspective projection case: suppose that after transformation to the standard perspective view volume, we have a line segment that goes from the center of the back plane, i.e., $(0, 0, -1)$ to a point in the positive-$z$ half-space, say $(2, 0, 1)$. This segment crosses the $z = 0$ plane at $(1, 0, 0)$, and when we transform to the standard parallel view volume, that point gets transformed to infinity! In fact, the segment becomes a pair of rays pointing in opposite directions. We have two choices:

- Keep track of such line segments and carefully handle their transformed versions as a pair of rays
- Clip such line segments before performing the transformation, thus transforming only line segments whose ends are both in the negative-$z$ half-space.
Figure 13.32: A view of a scene in which there are many line segments. The (very small) viewport has been divided into pixels. Many of the line segments in the scene either miss the viewport entirely, or lie mostly outside the viewport.
The second approach is much simpler, and a variant of it is the one used in practice: the viewing transformation with matrix $M_{\text{persp}}$ is applied to the segment’s endpoints, and then the perspective-to-orthographic transformation matrix $M_{\text{pp}}$ is multiplied in as well. Only the homogenization step is left undone. These two steps can therefore be combined into a single step in which we multiply the endpoints by the product matrix $M_{\text{pp}} \cdot M_{\text{persp}}$.

The resulting segments have endpoints with four coordinates, $x', y', z', w'$ with $w'$ not necessarily being 1. Because the perspective-to-orthographic transform has $\begin{bmatrix} 0 & 0 & -1 & 0 \end{bmatrix}$ as its last row, points which were in the positive-$z$ half-space before that transform end up with $w' < 0$. So we can clip our segments against the $w' > 0$ half-space. But in fact no point whose pre-perspective-to-orthographic $z$ coordinate is greater than $-n/f$ will ever appear in our viewport, so we can in fact clip against the $w \geq n/f$ half-space. This has the advantage that after clipping, all points have positive $w$ coordinates; homogenizing never produces a divide-by-zero error.

After homogenization, it’s easy to clip against the $x \leq 1$ and $x \geq -1$ half-spaces, and similarly for $y$.

In practice, the exact choice of what kind of clipping to do, and where to do it, often depends on the hardware architecture of a graphics processor; such choices are discussed in Chapter 39. It may be easier in some cases, for instance, to rasterize (i.e., convert to pixel values) a segment that extends slightly outside a viewport rather than clipping it, by simply throwing away pixels that fall outside the viewport. This approach is called scissoring.

Even so, if you are writing a software renderer, you may need to clip segments to half-planes or half-spaces. One easy approach is to treat a segment $AB$ as a parameterized ray, $\gamma(t) = (1 - t)A + tB$. It turns out to be easy to determine at what $t$-value such a ray enters or leaves a half-space (or a half-plane, in the 2D case). If it enters at, say, $t = 0.2$, then we can say “the clipped segment goes from $t = 0.2$ to $t = 1.0$, so its new endpoints are $0.8A + 0.2B$ and $B$. By repeatedly clipping the segment against each half-space defining a convex object, we can determine the segment clipped against the entire object. For instance, to clip against a unit square in the plane, we could clip against $x/\geq 0$, then $x \leq 1$, then $y \geq 0$, and finally $y \leq 1$. If, after clipping by repeated halfspaces, the range of valid $t$-values becomes empty, then the segment is outside the clipping region.

Let’s specify a half-space by a point $P$ on the boundary and a unit vector $n$ that points into the half-space. Thus the $z \geq 0$ halfspace could be specified by the point $(3, 4, 0)$ (or any other points whose $z$-coordinate is zero) and the vector $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

Where does the ray $A + td$ (where $d = B - A$) meet the half-plane defined by $P$ and $n$? Assuming for the moment that the ray is not parallel to
the boundary of the half-space, it meets the half-space at some point of the boundary, i.e., we seek a value \( t \) with

\[
(A + td) - P \cdot n = 0.
\]

A little algebra converts this to

\[
td \cdot n = (P - A) \cdot n
\]

and hence

\[
t = \frac{(P - A) \cdot n}{d \cdot n}.
\]

Furthermore, if \( d \cdot n \) is positive, this is the \( t \) value at which the ray enters the half-space (as \( t \) increases). If \( d \cdot n \) is negative, then as \( t \) increases, this is the \( t \)-value at which the ray leaves the half-space. And if the dot product is zero, then the ray is parallel to the boundary of the half-space, and either lies entirely inside or entirely outside the half-space.

With this in mind, we can write a procedure for clipping a segment of a ray against a half-space:

```csharp
public bool clip(Point A, Vector d, out double tmin, out double tmax, Point P, Vector n)
// Clip the portion of the ray A + t d lying between tmin and tmax // against the half-plane defined by (X - P) dot n >= 0.
// Return true if the clipped segment is nonempty, else false.
{
    double d = Dot(P - A, n);
    if (d == 0){
        // ray is parallel to bounding plane
        // if basepoint of ray is in halfspace, whole ray is, too.
        if (Dot(A - X, n) >= 0) {
            return true;
        }
        else{
            tmax = tmin - 1; // make sure empty segment is returned.
            return false;
        }
    }
    else {
        double t = Dot(P - A, n)/d;
        if (d > 0) { // we’re entering the half-plane
            tmin = Math.Max(tmin, t);
        }
        else {
            tmax = Math.Min(tmax, t);
        }
        return (tmax >= tmin);
    }
}
```

This code works for both the 2D and the 3D case (provided we replace `Point` with `Point3D`, and `Vector` with `Vector3D`). To clip a line segment in 2D against the square defined by \(-1 \leq x, y \leq 1\), we’d invoke the procedure above four times.
public bool segmentClip(Point A, Point B, out double tmin, out double tmax)
// clip the segment AB against the viewport -1 <= x, y <= 1.
{
    tmin = 0; tmax = 1; // initialize the segment.
    Vector d = B - A;
    return
    clip(A, d, tmin, tmax, new Point(1,0), new Vector(-1, 0)) &&
    clip(A, d, tmin, tmax, new Point(-1,0), new Vector(1, 0)) &&
    clip(A, d, tmin, tmax, new Point(0, 1), new Vector(0, -1)) &&
    clip(A, d, tmin, tmax, new Point(0,-1), new Vector(0, 1));
}

Here we are using the short-circuiting of the conjunction operator to ensure that if the segment lies entirely in the region \( x > 1 \), then no further clipping is done, and similarly for the other regions. This algorithm is essentially due to Cyrus and Beck [?], and was later made more efficient for the special case of the standard view volumes by Liang and Barsky [?].

What about other shapes that we might have to clip, like triangles? Most graphics processors are well-tuned to work with triangles, and not more general polygons. One insight is that when we clip a triangle against a frustum or parallelepiped, the result is either empty or a convex polygon. But a convex polygon can easily be broken into a set of triangles, to which the graphics processor is well-suited.

**Exercise 13.5:** Consider the following algorithm for clipping a non-degenerate triangle against a viewport in 2D: for each edge \( e \) of the viewport, compare all three vertexes of the triangle to the edge. If one vertex \( A \) is inside and two, say \( B \) and \( C \), are outside, clip the two crossing triangle edges \( AB \) and \( AC \) and return the triangle \( AB'C' \), where \( B' \) and \( C' \) are the clipped endpoints; we then clip this returned triangle against the remaining viewport edges. If two vertexes are inside, we get a quadrilateral \( BCC'B' \), which we break into two triangles, \( BCC' \) and \( C'B'B \), which we return and then clip against the remaining viewport edges. What is the largest number of triangles that this algorithm can produce after clipping against all four edges? How would you handle the case where a triangle vertex lies exactly on a clip edge?

### 13.11 Discussion and further reading

The idea of interpolating after projective transformation is an application of a more general principle: if we have a a projective transformation \( T \) and two points \( A \) and \( B \) in its domain, then we can parameterize the line between them as \( \gamma(t) = (1-t)A + tB \). We can also parameterize the line between the image points, writing \( \eta(s) = (1-s)T(A) + sT(B) \). In general, \( T(\gamma(t)) \) is not the same as \( \eta(t) \), as we observed. But \( T(\gamma(t)) = \eta(S(t)) \) for some function...
S that reparameterizes the line, i.e., that associates an \( s \)-value to every \( t \)-value. It turns out that the reparameterization \( S \) is always a projective transformation, i.e., has the form

\[
S(t) = \frac{at + b}{ct + d}
\]

for some four real numbers \( a, b, c \) and \( d \). The study of such re-parameterizations is at the heart of projective geometry; the book by Samuel ?? is an excellent reference for the mathematically inclined.

Clipping algorithms constituted a small industry when software rendering dominated. Several algorithms for computing visibility of one point from another in a polygonal environment depended on generalized clipping algorithms, in which one clipped one polygonally-bounded region against another. One of the early and aesthetically pleasing of these algorithms is the Sutherland-Hodgman algorithm [?] for clipping a polygon against an arbitrary convex clipping region bounded by planes (in 3-space) or lines (in the plane). The polygon is clipped against each boundary in sequence, in a divide-and-conquer approach. One of the subtleties is that a connected polygon may be clipped into multiple disconnected polygons.

Our discussion of line segments was motivated by their simplicity. In practice, line segments are less and less important. Some graphics systems implement line segments by drawing them as two very long thin triangles, so that the only primitive object the hardware must handle is the triangle.

**Exercise 13.6:** Just as a projective transformation of the plane is determined by its value on 4 points, a projective transformation of the line is determined by its value on three points. Such a projective transformation always has the form \( t \mapsto \frac{at + b}{ct + d} \), where \( a, b, c \), and \( d \) are real numbers with \( ad - bc \geq 0 \). (a) Suppose you want to send the points \( t = 0, 1, \infty \) to 3, 7, and 2, respectively. Find values of \( a, b, c \), and \( d \) that make this happen. The value at \( t = \infty \) is defined as the limit of values as \( t \to \infty \), and turns out to be \( a/c \). (b) Generalize: if we want \( t = 0, 1, \infty \) to be sent to \( A, B, \) and \( C \), find the appropriate values of \( a, b, c \) and \( d \).

**Exercise 13.7:** Create examples to show that a connected \( n \)-sided polygon in the plane, when clipped against a square, can produce up to \( \lfloor n/2 \rfloor \) disconnected pieces. What is the largest number of disconnected pieces that can be produced of the polygon is convex? Explain.
Exercise 13.8: Suppose that $T : \mathbb{R}^n \to \mathbb{R}^n : x \mapsto Mx$ is a linear transformation. (a) Recalling that the squared-length of a vector $v$ is just $v^t v$, express the length of $T(x)$ as a product of vectors and matrices involving $M$ and $x$. (b) Suppose that $M^t M$ is the identity; what can you say about the squared-length of $T(x)$? (c) Because $S = M^t M$ is symmetric, is it guaranteed, by Sylvester’s Law of Inertia, to have positive eigenvalues $\lambda_1, \ldots, \lambda_n$ (not necessarily distinct), for which one can find an orthonormal set of eigenvectors $v_1, v_2, \ldots, v_n$. Since these vectors are orthogonal, they’re also independent, so they form a basis of $\mathbb{R}^n$, i.e., any vector $x$ can be written as a linear combination $x = \alpha_1 v_1 + \ldots + \alpha_n v_n$. Remembering that $v_1, \ldots, v_n$ are unit vectors, what is the length of $x$ in terms of the $\alpha$s? (d) Express the length of $T(x)$ in terms of the $\alpha$s and the $\lambda$s. (e) Conclude that if $x$ is a vector that is a linear combination of eigenvectors for the eigenvalue 1, then $T(x)$ and $x$ have the same length. (f) Apply this result to a shearing operation in $\mathbb{R}^3$ that’s parallel to the $xy$-plane. What vectors have lengths that remain unchanged by such a shear?

Exercise 13.9: Construct a pinhole camera from a shoe-box and a sheet of tissue paper by cutting off one end of the shoe-box and replacing it with tissue paper, punching a tiny hole in the other end, and taping the top of the box in place. Stand inside a darkened room that looks out on a bright outdoor scene; look at the tissue paper, pointing the pinhole-end of the box towards the window. You should see a faint inverted view of the outdoor scene appear on the tissue paper. Now enlarge the hole somewhat, and again view the scene; notice how much blurrier the image is.

Exercise 13.10: We described two ways to change the view angle (varying $u_{\min} \ldots v_{\max}$ and varying $\alpha_3$). Do these result in exactly the same changes in the resulting image? In particular, suppose we (a) double each of $u_{\min} \ldots v_{\max}$ or (b) double $\alpha_3$. Will the two resulting projections be the same? Explain why, or create a scene in which they’ll be different and show why they are different.

Exercise 13.11: Find a photograph of a person, and estimate the distance from the camera to the subject – let’s say it’s 3 meters. Have a friend stand at that distance, and determine at what distance you would have to place the photograph so that the person in the photo occupies about the same visual area as your friend. Is this in fact the distance at which you are likely to view the photo? Try to explain what your brain might be doing when it views such a photo at a distance other than this “ideal.”