

(for ECE660, Fall, 1999)

CHAPTER 7 Three-Dimensional Viewing

I am a camera with its shutter open, quite passive, recording, not thinking.

Christopher Isherwood, A Berlin Diary

Goals of the Chapter

- To develop tools for creating and manipulating a “camera” that produces pictures of a 3D scene.
- To see how to “fly” a camera through a scene interactively, and to make animations.
- To learn the mathematics that describe various kinds of projections.
- To see how each operation in the OpenGL graphics pipeline operates, and why it is used.
- To build a powerful clipping algorithm for 3D objects.
- To devise a means for producing stereo views of objects.

Preview

Section 7.1 provides an overview of the additional tools that are needed to build an application that lets a camera “fly” through a scene. Section 7.2 defines a camera that produces perspective views, and shows how to make such a camera using OpenGL. It introduces aviation terminology that helps to describe ways to manipulate a camera. It develops some of the mathematics needed to describe a camera’s orientation through a matrix. Section 7.3 develops the Camera class to encapsulate information about a camera, and develops methods that create and adjust a camera in an application.

Section 7.4 examines the geometric nature of perspective projections, and develops mathematical tools to describe perspective. It shows how to incorporate perspective projections in the graphics pipeline, and describes how OpenGL does it. An additional property of homogeneous coordinates is introduced to facilitate this. The section also develops a powerful clipping algorithm that operates in homogeneous coordinate space, and shows how its efficiency is a result of proper transformations applied to points before clipping begins. Code for the clipper is given for those programmers who wish to develop their own graphics pipeline.

Section 7.5 shows how to produce stereo views of a scene in order to make them more intelligible.

Section 7.6 develops a taxonomy of the many kinds of projections used in art, architecture, and engineering, and shows how to produce each kind of projection in a program. The chapter closes with a number of Case Studies for developing applications that test the techniques discussed.

7.1 Introduction.

We are already in a position to create pictures of elaborate 3D objects. As we saw in Chapter 5, OpenGL provides tools for establishing a camera in the scene, for projecting the scene onto the camera’s viewplane, and for rendering the projection in the viewport. So far our camera only produces parallel projections. In Chapter 6 we described several classes of interesting 3D shapes that can be used to model the objects we want in a scene, and through the Mesh class we have ways of drawing any of them with appropriate shading.

So what’s left to do? For greater realism we want to create and control a camera that produces perspective projections. We also need ways to take more control of the camera’s position and orientation, so that the user can “fly” the camera through the scene in an animation. This requires developing more controls than OpenGL provides. We also need to achieve precise control over the camera’s view volume, which is determined in the perspective case as it was when forming parallel projections: by a certain matrix. This requires a deeper use of homogeneous coordinates than we have used so far, so we develop the mathematics of perspective projections from the beginning, and see how they are incorporated in the OpenGL graphics pipeline. We also describe how clipping is done against the camera’s view volume, which again requires some detailed working with homogeneous coordinates. So we finally see how it is all done, from start to finish! This also provides the underlying theory for those programmers who must develop 3D graphics software without the benefit of OpenGL.

7.2. The Camera Revisited.

It adds a precious seeing to the eye.

In Chapter 5 we used a camera that produces parallel projections. Its view volume is a parallelepiped bounded by six walls, including a near plane and a far plane. OpenGL also supports a camera that creates perspective views of 3D scenes. It is similar in many ways to the camera used before, except that its view volume has a different shape.

Figure 7.1 shows its general form. It has an **eye** positioned at some point in space, and its **view volume** is a portion of a rectangular pyramid, whose apex is at the eye. The opening of the pyramid is set by the **viewangle** θ (see part b of the figure). Two planes are defined perpendicular to the axis of the pyramid: the **near plane** and the **far plane**. Where these planes intersect the pyramid they form rectangular windows. The windows have a certain **aspect ratio**, which can be set in a program. OpenGL clips off any parts of the scene that lie outside the view volume. Points lying inside the view volume are projected onto the **view plane** to a corresponding point P' as suggested in part c. (We shall see that it doesn't matter which plane one uses as the view plane, so for now take it to be the near plane.) With a perspective projection the point P' is determined by finding where a line from the eye to P intersects the view plane. (Contrast this with how a parallel projection operates.)

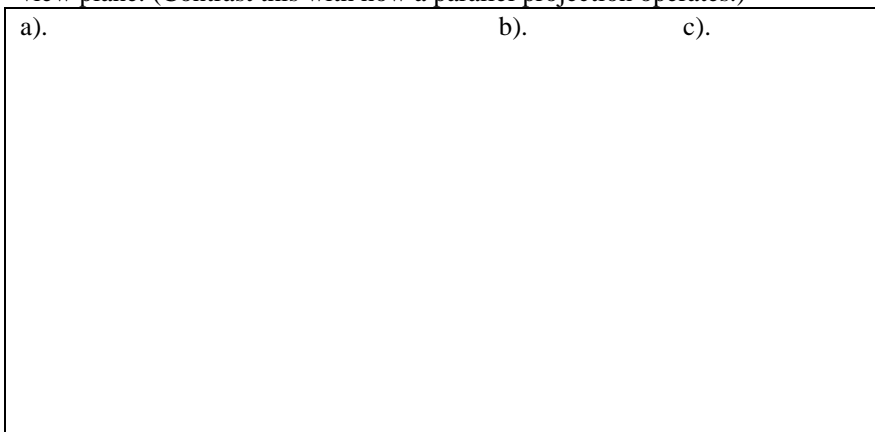


Figure 7.1. A camera to produce perspective views of a scene.

Finally, the image formed on the view plane is mapped into the viewport as shown in part c, and becomes visible on the display device.

7.2.1. Setting the View Volume.

Figure 7.2 shows the camera in its default position, with the eye at the origin and the axis of the pyramid aligned with the z -axis. The eye is “looking” down the negative z -axis.

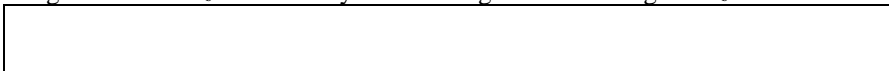


Figure 7.2. the camera in its default position.

OpenGL provides a simple way to set the view volume in a program. Recall that the shape of the camera's view volume is encoded in the **projection matrix** that appears in the graphics pipeline. The projection matrix is set up using the function `gluPerspective()` with four parameters. The sequence to use is:

```
glMatrixMode(GL_PROJECTION); // make the projection matrix current
glLoadIdentity();           // start with a unit matrix
gluPerspective(viewAngle, aspectRatio, N, F); // load the appropriate
values
```

The parameter `viewAngle`, shown as θ in the figure, is given in degrees, and sets the angle between the top and bottom walls of the pyramid. `aspectRatio` sets the aspect ratio of any window parallel to the xy -plane. The value `N` is the distance from the eye to the near plane, and `F` is the distance from the eye to the far plane. `N` and `F` should be positive. For example, `gluPerspective(60.0, 1.5, 0.3, 50.0)` establishes the view volume to have a vertical opening of 60° , with a window that has an aspect

ratio of 1.5. The near plane lies at $z = -0.3$ and the far plane lies at $z = -50.0$. We see later exactly what values this function places in the projection matrix.

7.2.2. Positioning and pointing the camera.

In order to obtain the desired view of a scene, we move the camera away from its default position shown in Figure 7.2, and aim it in a particular direction. We do this by performing a rotation and a translation, and these transformations become part of the **modelview matrix**, as we discussed in Section 5.6.

We set up the camera's position and orientation in *exactly* the same way as we did for the parallel-projection camera. (The only difference between a parallel- and perspective-projection camera resides in the projection matrix, which determines the *shape* of the view volume.) The simplest function to use is again `gluLookAt()`, using the sequence

```
glMatrixMode(GL_MODELVIEW); // make the modelview matrix current
glLoadIdentity();           // start with a unit matrix
gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y,
up.z);
```

As before this moves the camera so its eye resides at point `eye`, and it looks towards the point of interest `look`. The “upward” direction is generally suggested by the vector `up`, which is most often set simply to $(0, 1, 0)$. We took these parameters and the whole process of setting the camera pretty much for granted in Chapter 5. In this chapter we will probe deeper, both to see how it is done and to take finer control over setting the camera. We also develop tools to make *relative* changes to the camera's direction, such as rotating it slightly to the left, tilting it up, or sliding it forward.

The General camera with arbitrary orientation and position.

A camera can have any position in the scene, and any orientation. Imagine a transformation that picks up the camera of Figure 7.2 and moves it somewhere in space, then rotates it around to aim it as desired. We need a way to describe this precisely, and to determine what the resulting modelview matrix will be.

It will serve us well to attach an explicit coordinate system to the camera, as suggested by Figure 7.3. This coordinate system has its origin at the eye, and has three axes, usually called the u -, v -, and n - axes, that define its orientation. The axes are pointed in directions given by the vectors **u**, **v**, and **n** as shown in the figure. Because the camera by default looks down the negative z -axis, we say in general that the camera looks down the negative n -axis, in the direction **-n**. The direction **u** points off “to the right of” the camera, and direction **v** points “upward”. Think of the u -, v -, and n -axes as “clones” of the x -, y -, and z -axes of Figure 7.2, that are moved and rotated as we move the camera into position.



Figure 7.3. Attaching a coordinate system to the camera.

Position is easy to describe, but orientation is difficult. It helps to specify orientation using the flying terms **pitch**, **heading**, **yaw**, and **roll**, as suggested in Figure 7.4. The *pitch* of a plane is the angle that its longitudinal axis (running from tail to nose and having direction **-n**) makes with the horizontal plane. A plane *rolls* by rotating about this longitudinal axis; its *roll* is the amount of this rotation relative to the horizontal. A plane's *heading* is the direction in which it is headed. (Other terms are *azimuth* and *bearing*.) To find the heading and pitch given **n**, simply express **-n** in spherical coordinates, as shown in Figure 7.5. (See Appendix 2 for a review of spherical coordinates.) The vector **-n** has longitude and latitude given by angles θ and ϕ , respectively. The heading of a plane is given by the longitude of **-n**, and the pitch is given by the latitude of **-n**. Formulas for roll, pitch, and heading in terms of the vectors **u** and **n** are developed in the exercises.

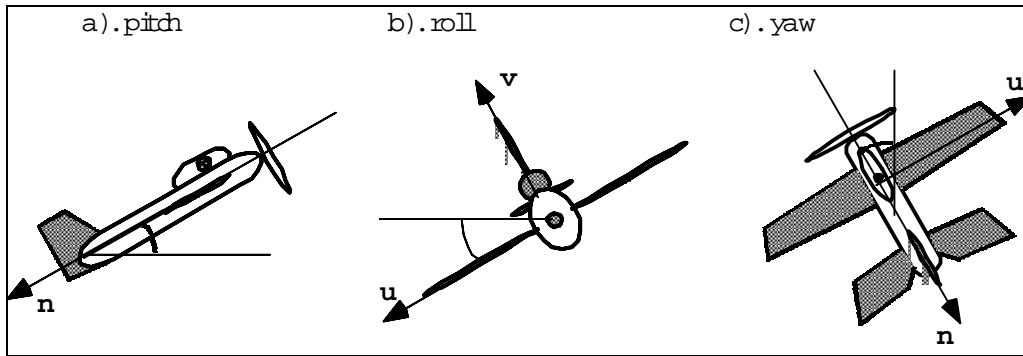


Figure 7.4. A plane's orientation relative to the "world".

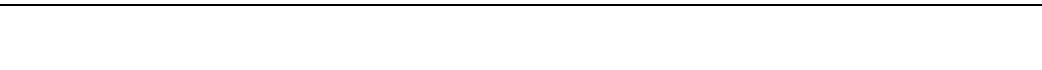


Figure 7.5. The heading and pitch of a plane.

Pitch and *roll* are both nouns and verbs: when used as verbs they describe a change in the plane's orientation. You can say a plane "pitches up" when it increases its pitch (rotates about its u -axis), and that it "rolls" when it rotates about its n -axis. The common term for changing heading is *yaw*: to yaw left or right it rotates about its v -axis.

These terms can be used with a camera as well. Figure 7.6a shows a camera with the same coordinate system attached: it has u , v , and n -axes, and its origin is at position *eye*. The camera in part b has some non-zero roll, whereas the one in part c has zero roll. We most often set a camera to have zero roll, and call it a "**no-roll**" camera. The u -axis of a no-roll camera is horizontal: that is, perpendicular to the y -axis of the world. Note that a no-roll camera can still have an arbitrary n direction, so it can have any pitch or heading.

How do we control the roll, pitch, and heading of a camera? `gluLookAt()` is handy for setting up an initial camera, since we usually have a good idea of how to choose *eye* and *look*. But it's harder to visualize how to choose **up** to obtain a certain roll, and it's hard to make later relative adjustments to the camera using only `gluLookAt()`. (`gluLookAt()` works with Cartesian coordinates, whereas orientation deals with angles and rotations about axes.) OpenGL doesn't give direct access to the u , v , and n directions, so we'll maintain them ourselves in a program. This will make it much easier to describe and adjust the camera.

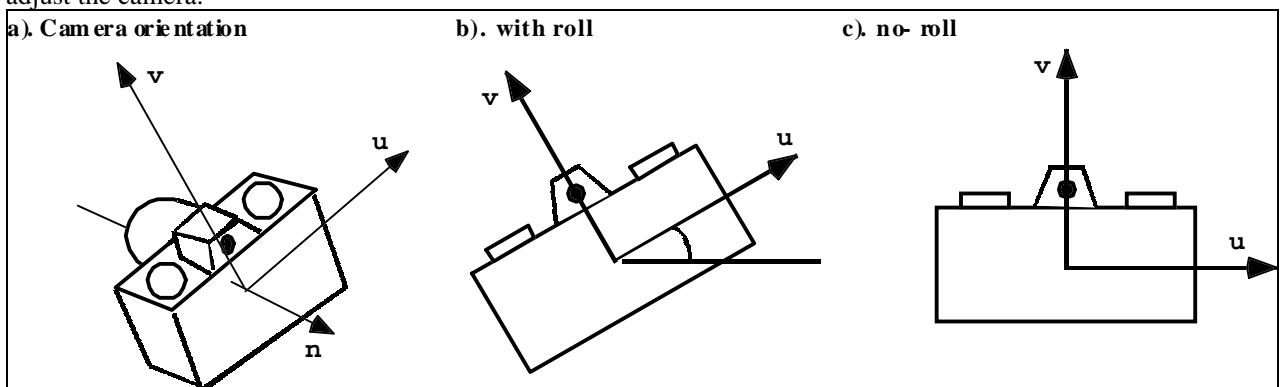


Figure 7.6. Various camera orientations.

What `gluLookAt()` does: some mathematical underpinnings.

What then are the directions u , v , and n when we execute `gluLookAt()` with given values for *eye*, *look*, and **up**? Let's see exactly what `gluLookAt()` does, and why it does it.

As shown in Figure 7.7a, we are given the locations of *eye* and *look*, and the **up** direction. We immediately know that n must be parallel to the vector *eye* - *look*, as shown in Figure 7.7b, so we set $n = \text{eye} - \text{look}$. (We'll normalize this and the other vectors later as necessary.)



Figure 7.7. Building the vectors \mathbf{u} , \mathbf{v} , and \mathbf{n} .

We now need to find \mathbf{u} and \mathbf{v} that are perpendicular to \mathbf{n} and to each other. The \mathbf{u} direction points “off to the side” of a camera, so it is (fairly) natural to make it perpendicular to \mathbf{up} , which the user has said is the “upward” direction. This is the assumption `gluLookAt()` makes in any case, and so the direction \mathbf{u} is made perpendicular to \mathbf{n} and \mathbf{up} . An excellent way to build a vector that is perpendicular to two given vectors is to form their cross product, so we set $\mathbf{u} = \mathbf{up} \times \mathbf{n}$. (The user should not choose an \mathbf{up} direction that is parallel to \mathbf{n} , as then \mathbf{u} would have zero length - why?) We choose $\mathbf{u} = \mathbf{up} \times \mathbf{n}$ rather than $\mathbf{n} \times \mathbf{up}$ so that \mathbf{u} will point “to the right” as we look along $-\mathbf{n}$.

With \mathbf{u} and \mathbf{n} in hand it is easy to form \mathbf{v} : it must be perpendicular to both \mathbf{u} and \mathbf{n} so use a cross product again: $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. Notice that \mathbf{v} will usually not be aligned with \mathbf{up} : \mathbf{v} must be aimed perpendicular to \mathbf{n} , whereas the user provides \mathbf{up} as a suggestion of “upwardness”, and the only property of it that is used is its cross product with \mathbf{n} .

Summarizing: given *eye*, *look*, and \mathbf{up} , we form

$$\mathbf{n} = \text{eye} - \text{look}$$

$$\mathbf{u} = \mathbf{up} \times \mathbf{n} \tag{7.1}$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$

and then normalize all three to unit length.

Note how this plays out for the common case where $\mathbf{up} = (0, 1, 0)$. Convince yourself that in this case $\mathbf{u} = (n_x, 0, -n_z)$ and $\mathbf{v} = (-n_x n_y, n_x^2 + n_z^2, -n_z n_y)$. Notice that \mathbf{u} does indeed have a y-component of 0, so it is “horizontal”. Further, \mathbf{v} has a positive y-component, so it is pointed more or less “upward”.

Example 7.2.1. Find the camera coordinate system. Consider a camera with *eye* = (4, 4, 4) that “looks down” on a look-at point *look* = (0, 1, 0). Further suppose that \mathbf{up} is initially set to (0, 1, 0). Find \mathbf{u} , \mathbf{v} , and \mathbf{n} . Repeat for $\mathbf{up} = (2, 1, 0)$.

Solution: From Equation 7.1 we find: $\mathbf{u} = (4, 0, -4)$, $\mathbf{v} = (-12, 32, -12)$, $\mathbf{n} = (4, 3, 4)$, which are easily normalized to unit length. (Sketch this situation.) Note that \mathbf{u} is indeed horizontal. Check that these are mutually perpendicular. For the case of $\mathbf{up} = (2, 1, 0)$ (try to visualize this camera before working out the arithmetic), $\mathbf{u} = (4, -8, 2)$, $\mathbf{v} = (38, 8, -44)$, and $\mathbf{n} = (4, 3, 4)$. Sketch this situation. Check that these vectors are mutually perpendicular.

Example 7.2.2. Building intuition with cameras. To assist in developing geometric intuition when setting up a camera, Figure 7.8 shows two example cameras - each depicted as a coordinate system with a view volume - positioned above the world coordinate system, which is made more visible by grids drawn in the *xz*-plane. The view volume of both cameras has an aspect ratio of 2. One camera is set with *eye* = (-2, 2, 0), *look* = (0, 0, 0), and $\mathbf{up} = (0, 1, 0)$. For this camera we find from Equation 7.1 that $\mathbf{n} = (-2, 2, 0)$, $\mathbf{u} = (0, 0, 2)$, and $\mathbf{v} = (4, 4, 0)$. The figure shows these vectors (\mathbf{v} is drawn the darkest) as well as the \mathbf{up} vector. The second camera uses *eye* = (2, 2, 0), *look* = (0, 0, 0), and $\mathbf{up} = (0, 0, 1)$. In this case $\mathbf{u} = (-2, -2, 0)$ and $\mathbf{v} = (0, 0, 8)$. The direction \mathbf{v} is parallel to \mathbf{up} here. Note that this camera appears to be “on its side”: (Check that all of these vectors appear drawn in the proper directions.)

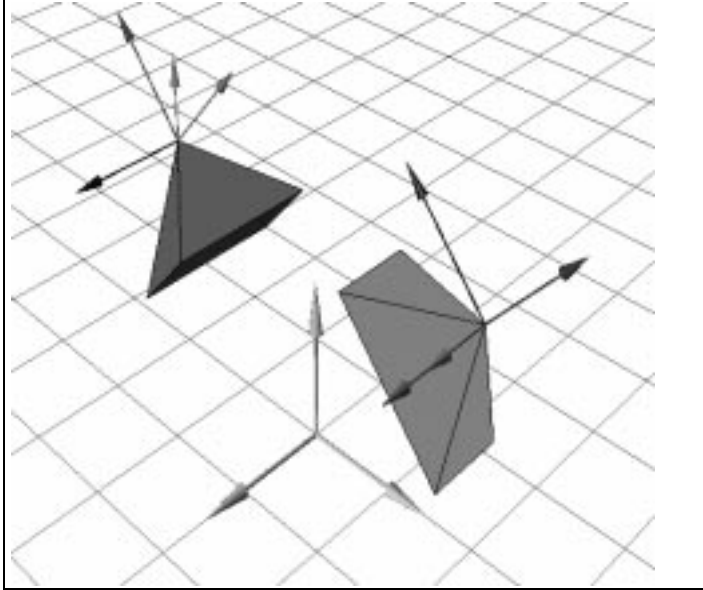


Figure 7.8. Two example settings of the camera.

Finally, we want to see what values `gluLookAt()` places in the modelview matrix. From Chapter 5 we know that the modelview matrix is the product of two matrices, the matrix V that accounts for the transformation of world points into camera coordinates, and the matrix M that embodies all of the modeling transformations applied to points. `gluLookAt()` builds the V matrix and postmultiplies the current matrix by it. Because the job of the V matrix is to convert world coordinates to camera coordinates, it must transform the camera's coordinate system into the generic position for the camera as shown in Figure 7.9. This means it must transform eye into the origin, \mathbf{u} into the vector \mathbf{i} , \mathbf{v} into \mathbf{j} , and \mathbf{n} into \mathbf{k} . There are several ways to derive what V must be, but it's easiest to check that the following matrix does the trick:



Figure 7.9. The transformation which `gluLookAt()` sets up.

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (7.2)$$

where $(d_x, d_y, d_z) = (-eye \cdot \mathbf{u}, -eye \cdot \mathbf{v}, -eye \cdot \mathbf{n})$ ¹. Check that in fact

$$V \begin{pmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

¹ A technicality: since it's not legal to dot a point and a vector, eye should be replaced here by the vector $(eye - (0,0,0))$.

as desired, where we have extended point *eye* to homogeneous coordinates. Also check that

$$V \begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

and that V maps \mathbf{v} into $(0,1,0,0)^T$ and maps \mathbf{n} into $(0,0,1,0)^T$. The matrix V is created by `gluLookAt()` and is postmultiplied with the current matrix. We will have occasion to do this same operation later when we maintain our own camera in a program.

Practice Exercises.

7.2.1. Finding roll, pitch, and heading given vectors \mathbf{u} , \mathbf{v} , and \mathbf{n} . Suppose a camera is based on a coordinate system with axes in the directions \mathbf{u} , \mathbf{v} , and \mathbf{n} , all unit vectors. The heading and pitch of the camera is found by representing $-\mathbf{n}$ in spherical coordinates. Using Appendix 2, show that

$$\text{heading} = \arctan(-n_z, -n_x)$$

$$\text{pitch} = \sin^{-1}(-n_y)$$

Further, the roll of the camera is the angle its u -axis makes with the horizontal. To find it, construct a vector \mathbf{b} that is horizontal and lies in the uv -plane. Show that $\mathbf{b} = \mathbf{j} \times \mathbf{n}$ has these properties. Show that the angle between \mathbf{b} and \mathbf{u} is given by

$$\text{roll} = \cos^{-1} \left(\frac{u_x n_z - u_z n_x}{n_x^2 + n_z^2} \right)$$

7.2.2. Using \mathbf{up} sets \mathbf{v} to a “best approximation” to \mathbf{up} . Show that using \mathbf{up} as in Equation 7.1 to set \mathbf{u} and \mathbf{v} is equivalent to making \mathbf{v} the closest vector to \mathbf{up} that is perpendicular to vector \mathbf{n} . Use these steps:

- Show that $\mathbf{v} = \mathbf{n} \times (\mathbf{up} \times \mathbf{n})$;
- Use a property of the “triple vector product”, that says $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$.
- Show that \mathbf{v} is therefore the projection of \mathbf{up} onto the plane with normal \mathbf{n} (see Chapter 4), and therefore is the closest vector in this plane to \mathbf{up} .

7.3 Building a Camera in a Program.

It is as interesting and as difficult to say a thing well as to paint it.
Vincent van Gogh

In order to have fine control over camera movements, we create and manipulate our own camera in a program. After each change to this camera is made, the camera “tells” OpenGL what the new camera is.

We create a `Camera` class that knows how to do all the things a camera does. It’s very simple and the payoff is high. In a program we create a `Camera` object called, say, `cam`, and adjust it with functions such as:

```
cam.set(eye, look, up); // initialize the camera - similar to
gluLookAt()
cam.slide(-1,0,-2); // slide the camera forward and to the left
cam.roll(30); // roll it through 30°
cam.yaw(20); // yaw it through 20°
etc.
```

Figure 7.10 shows the basic definition of the `Camera` class. It contains fields for the *eye* and the directions \mathbf{u} , \mathbf{v} , and \mathbf{n} . (`Point3` and `Vector3` are the basic data types defined in Appendix 3.) It also has fields that describe the shape of the view volume: `viewAngle`, `aspect`, `nearDist`, and `farDist`.

```
class Camera{
```

```

private:
    Point3 eye;
    Vector3 u,v,n;
    double viewAngle, aspect, nearDist, farDist; // view volume shape
    void setModelviewMatrix(); // tell OpenGL where the camera is

public:
    Camera(); // default constructor
    void set(Point3 eye, Point3 look, Vector3 up); // like gluLookAt()
    void roll(float angle); // roll it
    void pitch(float angle); // increase pitch
    void yaw(float angle); // yaw it
    void slide(float delU, float delV, float delN); // slide it
    void setShape(float vAng, float asp, float nearD, float farD);
};

```

Figure 7.10. The Camera class definition.

The utility routine `setModelviewMatrix()` communicates the modelview matrix to OpenGL. It is used only by member functions of the class, and needs to be called after each change is made to the camera's position or orientation. Figure 7.11 shows a possible implementation. It computes the matrix of Equation 7.2 based on current values of *eye*, *u*, *v*, and *n*, and loads the matrix directly into the modelview matrix using `glLoadMatrixf()`.

The method `set()` acts just like `gluLookAt()`: it uses the values of *eye*, *look*, and *up* to compute *u*, *v*, and *n* according to Equation 7.1. It places this information in the camera's fields and communicates it to OpenGL. Figure 7.11 shows a possible implementation.

```

void Camera::setModelViewMatrix(void)
{ // load modelview matrix with existing camera values
    float m[16];
    Vector3 eVec(eye.x, eye.y, eye.z); // a vector version of eye
    m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -eVec.dot(u);
    m[1] = v.x; m[5] = v.y; m[9] = v.z; m[13] = -eVec.dot(v);
    m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -eVec.dot(n);
    m[3] = 0; m[7] = 0; m[11] = 0; m[15] = 1.0;
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(m); // load OpenGL's modelview matrix
}

void Camera::set(Point3 Eye, Point3 look, Vector3 up)
{
    // create a modelview matrix and send it to OpenGL
    eye.set(Eye); // store the given eye position
    n.set(eye.x - look.x, eye.y - look.y, eye.z - look.z); // make n
    u.set(up.cross(n)); // make u = up X n
    n.normalize(); u.normalize(); // make them unit length
    v.set(n.cross(u)); // make v = n X u
    setModelViewMatrix(); // tell OpenGL
}

```

Figure 7.11. The utility routines `set()` and `setModelViewMatrix()`.

The routine `setShape()` is even simpler: It puts the four argument values into the appropriate camera fields, and then calls `gluPerspective(viewangle,aspect,nearDist, farDist)` (along with `glMatrixMode(GL_PROJECTION)` and `glLoadIdentity()`) to set the projection matrix.

The central camera methods are `slide()`, `roll()`, `yaw()`, and `pitch()`, which make *relative* changes to the camera's position and orientation. (The whole reason for maintaining the *eye*, *u*, *v*, and *n* fields in our Camera data structure is so that we have a record of the "current" camera, and can therefore alter it.) We examine how the camera methods operate next.

7.3.1. "Flying" the Camera.

The user flies the camera through a scene interactively by pressing keys or clicking the mouse. For instance, pressing 'u' might slide the camera "up" some amount, pressing 'y' might yaw it to the left, and pressing 'f' might slide it forward. The user can see how the scene looks from one point of view, then

change the camera to a better viewing spot and direction and produce another picture. Or the user can fly around a scene taking different snapshots. If the snapshots are stored and then played back rapidly, an animation is produced of the camera flying around the scene.

There are six degrees of freedom for adjusting a camera: it can be “slid” in three dimensions, and it can be rotated about any of three coordinate axes. We first develop the `slide()` function.

Sliding the camera.

Sliding a camera means to move it along one its *own* axes, that is, in the **u**, **v**, or **n** direction, without rotating it. Since the camera is looking along the negative **n**-axis, movement along **n** is “forward” and “back”. Similarly, movement along **u** is “left” or “right”, and along **v** is “up” or “down”.

It is simple to move the camera along one of its axes. To move it distance D along its u -axis, set eye to $eye + D \mathbf{u}$. For convenience we can combine the three possible slides in a single function. `slide(delU, delV, delN)` slides the camera amount `delU` along **u**, `delV` along **v**, and `delN` along **n**:

```
void Camera:: slide(float delU, float delV, float delN)
{
    eye.x += delU * u.x + delV * v.x + delN * n.x;
    eye.y += delU * u.y + delV * v.y + delN * n.y;
    eye.z += delU * u.z + delV * v.z + delN * n.z;
    setModelViewMatrix();
}
```

Rotating the Camera.

We want to roll, pitch, or yaw the camera. This involves a rotation of the camera about one of its own axes. We look at rolling in detail; the other two types of rotation are similar.

To roll the camera is to rotate it about its own **n** axis. This means that both the directions **u** and **v** must be rotated, as shown in Figure 7.12. We form two new axes **u'** and **v'** that lie in the same plane as **u** and **v** yet have been rotated through the angle α degrees.

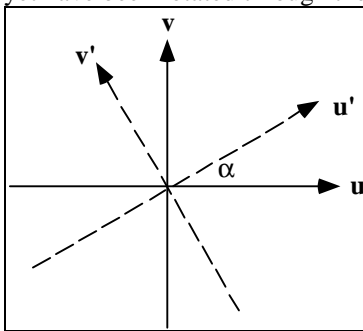


Figure 7.12. Rolling the camera.

So we need only form **u'** as the appropriate linear combination of **u** and **v**, and do similarly for **v'**:

$$\begin{aligned}\mathbf{u}' &= \cos(\alpha) \mathbf{u} + \sin(\alpha) \mathbf{v} \\ \mathbf{v}' &= -\sin(\alpha) \mathbf{u} + \cos(\alpha) \mathbf{v}\end{aligned}\tag{7.3}$$

The new axes **u'** and **v'** then replace **u** and **v** in the camera. This is straightforward to implement.

```
void Camera :: roll(float angle)
{ // roll the camera through angle degrees
    float cs = cos(3.14159265/180 * angle);
    float sn = sin(3.14159265/180 * angle);
    Vector3 t(u); // remember old u
    u.set(cs*t.x - sn*v.x, cs*t.y - sn*v.y, cs*t.z - sn*v.z);
    v.set(sn*t.x + cs*v.x, sn*t.y + cs*v.y, sn*t.z + cs*v.z);
    setModelViewMatrix();
}
```

The functions `pitch()` and `yaw()` are implemented in a similar fashion. See the exercises.

Putting it all together.

We show in Figure 7.13 how the `Camera` class can be used with OpenGL to fly a camera through a scene. The scene consists of the lone teapot here. The camera is a global object, and is set up in `main()` with a good starting view and shape. When a key is pressed `myKeyboard()` is called, and the camera is slid or rotated depending on which key was pressed. For instance, if ‘P’ is pressed the camera is pitched up by 1 degree. If CTRL F is pressed² (hold down the control key and press ‘f’), the camera is pitched down by 1 degree. After the keystroke has been processed `glutPostRedisplay()` causes `myDisplay()` to be called again to draw the new picture.

[illegible]

Figure 7.13. Application to fly a camera around the teapot.

² On most keyboards pressing CTRL and a letter key returns an ASCII value that is 64 less than the ASCII value returned by the letter itself.

Notice the call to `glutSwapBuffers()`³. This application uses **double-buffering** to produce a rapid and smooth transition between one picture and the next. There are two memory buffers used to store the generated pictures. The display switches from showing one buffer to showing the other under the control of `glutSwapBuffers()`. Each new picture is drawn in the invisible buffer, and when the drawing is complete the display switches to it. Thus the viewer doesn't see the screen erased and the new picture slowly emerge line-by-line, which is visually annoying. Instead the "old" picture is displayed steadily while the picture is being composed off-screen, and then the display switches very rapidly to the newly completed picture.

Drawing SDL scenes using a camera.

It is just as easy to incorporate a camera in an application that reads SDL files, as described in Chapter 5. There are then two global objects:

```
Camera cam;
Scene scn;
```

and in `main()` an SDL file is read and parsed using `scn.read("myScene.dat")`. Finally, in `myDisplay(void)`, simply replace `glutWireTeapot(1.0)` with `scn.drawSceneOpenGL()`;

Practice Exercises.

7.3.1. Implementing `pitch()` and `yaw()`. Write the functions `void Camera::pitch(float angle)` and `void Camera::yaw(float angle)` that respectively pitch and yaw the camera. Arrange matters so that a positive yaw yaws the camera to the "left" and a positive pitch pitches the camera "up".

7.3.2. Building a universal `rotate()` method. Write the functions `void Camera::rotate(Vector3 axis, float angle)` that rotates the camera through `angle` degrees about `axis`. It rotates all three axes **u**, **v**, and **n** about the eye.

7.4. Perspective Projections of 3D Objects

Treat them in terms of the cylinder, the sphere, the cone, all in perspective.

Ashanti proverb

With the Camera class in hand we can navigate around 3D scenes and readily create pictures. Using OpenGL each picture is created by passing vertices of objects (such as a mesh representing a teapot or chess piece) down the graphics pipeline, as we described in Chapter 5. Figure 7.14 shows the graphics pipeline again, with one new element.

has perspective division too

Figure 7.14. The graphics pipeline revisited.

Recall that each vertex v is multiplied by the modelview matrix (VM). The modeling part (M) embodies all of the modeling transformations for the object; the viewing part (V) accounts for the transformation set by the camera's position and orientation. When a vertex emerges from this matrix it is in **eye coordinates**, that is, in the coordinate system of the eye. Figure 7.15 shows this system, for which the eye is at the origin, and the near plane is perpendicular to the z -axis residing at $z = -N$. A vertex located at P in eye coordinates is passed through the next stages of the pipeline where it is (somehow) projected to a certain point (x^*, y^*) on the near plane, clipping is carried out, and finally the surviving vertices are mapped to the viewport on the display.

Figure 7.15. Perspective projection of vertices expressed in eye coordinates.

³ `glutInitDisplayMode()` must have an argument of `GLUT_DOUBLE` to enable double-buffering.

At this point we must look more deeply into the process of forming perspective projections. We need answers to a number of questions. What operations constitute forming a perspective projection, and how does the pipeline do these operations? What's the relationship between perspective projections and matrices. How does the projection map the view volume into a "canonical view volume" for clipping? How is clipping done? How do homogeneous coordinates come into play in the process? How is the "depth" of a point from the eye retained so that proper hidden surface removal can be done? And what is that "perspective division" step?

We start by examining the nature of perspective projection, independent of specific processing steps in the pipeline. Then we see how the steps in the pipeline are carefully crafted to produce the numerical values required for a perspective projection.

7.4.1. Perspective Projection of a Point.

The fundamental operation is projecting a 3D point to a 2D point on a plane. Figure 7.16 elaborates on Figure 7.15 to show point $P = (P_x, P_y, P_z)$ projecting onto the near plane of the camera to a point (x^*, y^*) . We erect a local coordinate system on the near plane, with its origin on the camera's z -axis. Then it is meaningful to talk about the point x^* units over to the right of the origin, and y^* units above the origin.



Figure 7.16. Finding the projection of a point P in eye coordinates.

The first question is then, what are x^* and y^* ? It's simplest to use similar triangles, and say x^* is in the same ratio to P_x as the distance N is to the distance $|P_z|$. Or since P_z is negative, we can say

$$\frac{x^*}{P_x} = \frac{N}{-P_z}$$

or $x^* = NP_x/(-P_z)$. Similarly $y^* = NP_y/(-P_z)$. So we have that P projects to the point on the viewplane:

$$(x^*, y^*) = \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z} \right) \quad \text{(the projection of } P \text{)} \quad (7.4)$$

An alternate (analytical) method for arriving at this result is given in the exercises.

Example 7.4.1: Where on the viewplane does $P = (1, 0.5, -1.5)$ lie for the camera having a near plane at $N = 1$? **Solution:** Direct use of Equation 7.4 yields $(x^*, y^*) = (0.666, 0.333)$.

We can make some preliminary observations about how points are projected.

- 1). Note the denominator term $-P_z$. It is larger for more remote points (those farther along the negative z -axis), which reduces the values of x^* and y^* . This introduces **perspective foreshortening**, and makes remote parts of an object appear smaller than nearer parts.
- 2). Denominators have a nasty way of evaluating to zero, and P_z becomes 0 when P lies in the "same plane" as the eye: the $z = 0$ plane. Normally we use clipping to remove such offending points before trying to project them.
- 3). If P lies "behind the eye" there is a reversal in sign of P_z , which causes further trouble, as we see later. These points, too, are usually removed by clipping.
- 4). The effect of the near plane distance N is simply to *scale* the picture (both x^* and y^* are proportional to N). So if we choose some other plane (still parallel to the near plane) as the view plane onto which to project pictures, the projection will differ only in size with the projection onto the near plane. Since we ultimately map this projection to a viewport of a fixed size on the display, the size of the projected image makes no difference. This shows that any viewplane parallel to the near plane would work just as well, so we might as well use the near plane itself.

5). Straight lines project to straight lines. Figure 7.17 provides the simplest proof. Consider the line in 3D space between points A and B . A projects to A' and B projects to B' . But do points between A and B project to points directly between A' and B' ? Yes: just consider the plane formed by A , B and the origin. Since any two planes intersect in a straight line, this plane intersects the near plane in a straight line. Thus line segment AB projects to *line segment* $A'B'$.



Figure 7.17. Proof that a straight line projects to a straight line.

Example 7.4.2. Three projections of the barn.

A lot of intuition can be acquired by seeing how a simple object is viewed by different cameras. Here we examine how the edges of the barn defined in Chapter 6 and repeated in Figure 7.18 are projected onto three cameras. The barn has 10 vertices, 15 edges and seven faces.



Figure 7.18. The basic barn revisited.

• **View #1:** We first set the camera's *eye* at $(0, 0, 3)$ and have it look down the negative z -axis, with $\mathbf{u} = (1, 0, 0)$ and $\mathbf{n} = (-1, 0, 0)$. We will set the near plane at distance 1 from the eye. (The near plane happens therefore to coincide with the front of the barn.) In terms of camera coordinates all points on the front wall of the barn have $P_z = -1$ and those on the back wall have $P_z = -2$. So from Equation 7.4 any point (P_x, P_y, P_z) on the front wall projects to:

$$P' = (P_x, P_y) \quad \{\text{projection of a point on the front wall}\}$$

and any point on the back wall projects to

$$P' = (P_x / 2, P_y / 2). \quad \{\text{projection of a point on the back wall}\}$$

The foreshortening factor is two for those points on the back wall. Figure 7.19a shows the projection of the barn for this view. Note that edges on the rear wall project at half their true length. Also note that edges of the barn that are actually parallel in 3D *need not* project as parallel. (We shall see that parallel edges that are parallel to the viewplane do project as parallel, but parallel edges that are not parallel to the viewplane are not parallel: they recede to a “vanishing point”.)



Figure 7.19. Projections of the barn for views #1 and #2.

View #2: Here the camera has been slid over so that *eye* = $(0.5, 0, 2)$, but \mathbf{u} , and \mathbf{n} are the same as in view #1. Figure 7.19b shows the projection.

View #3 Here we use the camera with *Eye* = $(2, 5, 2)$ and *look* = $(0, 0, 0)$, resulting in Figure 7.20. The world axes have been added as a guide. This shows the barn from an informative point of view. From a wireframe view it is difficult to discern which faces are where.

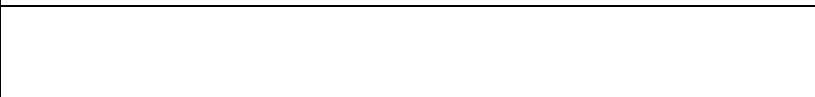


Figure 7.20. A third view of the barn.

Practice Exercises.

7.4.1. Sketch a Cube in Perspective. Draw (by hand) the perspective view of a **cube** C (axis-aligned and centered at the origin, with sides of length 2) when the eye is at $E = 5$ on the z -axis. Repeat when C is shifted so that its center is at $(1, 1, 1)$.

7.4.2. Where does the ray hit the viewplane? (Don't skip this one.)

We want to derive Equation 7.4 by finding where the ray from the origin to P intersects the near plane.

- a). Show that if this ray is at the origin at $t = 0$ and at P at time $t = 1$, then it has parametric representation $r(t) = Pt$.
- b). Show that it hits the near plane at $t = N/(-P_z)$;
- c). Show that the “hit point” is $(x^*, y^*) = (NP_x/(-P_z), NP_y/(-P_z))$.

7.4.2. Perspective Projection of a Line.

We develop here some interesting properties of perspective projections by examining how straight lines project.

- 1). Lines that are parallel in 3D project to lines, but these lines aren't necessary parallel. If not parallel, they meet at some “vanishing point.”
- 2). Lines that pass behind the eye of the camera cause a catastrophic “passage through infinity”. (Such lines should be clipped off.)
- 3). Perspective projections usually produce geometrically realistic pictures. But realism is strained for very long lines parallel to the viewplane.

1). Projecting Parallel Lines.

We suppose the line in 3D passes (using camera coordinates) through point $A = (A_x, A_y, A_z)$ and has direction vector $\mathbf{c} = (c_x, c_y, c_z)$. It therefore has parametric form $P(t) = A + \mathbf{c}t$. Substituting this form in Equation 7.4 yields the parametric form for the projection of this line:

$$p(t) = \left(N \frac{A_x + c_x t}{-A_z - c_z t}, N \frac{A_y + c_y t}{-A_z - c_z t} \right) \quad (7.5)$$

(This may not look like the parametric form for a straight line, but it is. See the exercises.) Thus the point A in 3D projects to the point $p(0)$, and as t varies the projected point $p(t)$ moves across the screen (in a straight line). We can discern several important properties directly from this formula.

Suppose the line $A + \mathbf{c}t$ is parallel to the viewplane. Then $c_z = 0$ and the projected line is given by:

$$p(t) = \frac{N}{-A_z} (A_x + c_x t, A_y + c_y t)$$

This is the parametric form for a line with slope c_y/c_x . This slope does not depend on the position of the line, only its direction \mathbf{c} . Thus all lines in 3D with direction \mathbf{c} will project with this slope, so their projections are parallel. We conclude:

If two lines in 3D are parallel to each other *and* to the viewplane, they project to two parallel lines.

Now consider the case where the direction \mathbf{c} is not parallel to the viewplane. For convenience suppose $c_z < 0$, so that as t increases the line recedes further and further from the eye. At very large values of t , Equation 7.5 becomes:

$$p(\infty) = \left(N \frac{c_x}{-c_z}, N \frac{c_y}{-c_z} \right) \quad (\text{the vanishing point for the line}) \quad (7.6)$$

This is called the “**vanishing point**” for this line: it's the point towards which the projected line moves as t gets larger and larger. Notice that it depends only on the direction \mathbf{c} of the line and not its position (which is embodied in A). Thus all parallel lines share the same vanishing point. In particular, these lines project to lines that are *not* parallel.

Figure 7.21a makes this more vivid for the example of a cube. Several edges of the cube are parallel: there are those that are horizontal, those that are vertical, and those that recede from the eye. This picture

was made with the camera oriented so that its near plane was parallel to the front face of the cube. Thus in camera coordinates the z -component of \mathbf{c} for the horizontal and vertical edges is 0. The horizontal edges therefore project to parallel lines, and so do the vertical edges. The receding edges, however, are not parallel to the view plane, and hence converge onto a vanishing point (VP). Artists often set up drawings of objects this way, choosing the vanishing point and sketching parallel lines as pointing at the VP . We shall see more on vanishing points as we proceed.

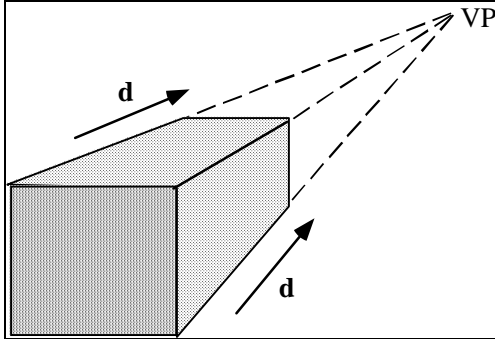


Figure 7.21. The vanishing point for parallel lines.

Figure 7.22 suggests what a vanishing point is geometrically. Looking down onto the camera's xz -plane from above, we see the eye viewing various points on the line AB . A projects to A' , B projects to B' , etc. Very remote points on the line project to VP as shown. The point VP is situated so that the line from the eye through VP is *parallel* to AB (why?).

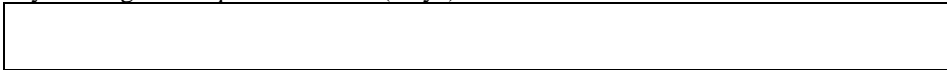


Figure 7.22. The geometry of a vanishing point.

2). Lines that Pass Behind the Eye.

We saw earlier that trying to project a point that lies in the plane of the eye ($z = 0$ in eye coordinates) results in a denominator of 0, and would surely spell trouble if we try to project it. We now examine the projection of a line segment where one endpoint lies in front of the eye, and one endpoint lies behind.

Figure 7.23 again looks down on the camera from above. Point A lies in front of the eye and projects to A' in a very reasonable manner. Point B , on the other hand, lies behind the eye, and projects to B' , which seems to end up on the wrong side of the viewplane! Consider a point C that moves from A to B , and sketch how its projection moves. As C moves back towards the plane of the eye, its projection slides further and further along the viewplane to the right. As C approaches the plane of the eye its projection spurts off to infinity, and as C moves behind the eye its projection reappears from far off to the left on the viewplane! You might say that the projection has “wrapped around infinity” and come back from the opposite direction [blinn96]. If we tried to draw such a line there would most likely be chaos. Normally all parts of the line closer to the eye than the near plane are clipped off before the projection is attempted.



Figure 7.23. Projecting the line segment AB , with B “behind the eye.”

Example 7.4.3. The Classic Horizontal Plane in Perspective.

A good way to gain insight into vanishing points is to view a grid of lines in perspective, as in Figure 7.24. Grid lines here lie in the xz -plane, and are spaced 1 unit apart. The eye is perched 1 unit above the xz -plane, at $(0, 1, 0)$, and looks along $-\mathbf{n}$, where $\mathbf{n} = (0, 0, 1)$. As usual we take $\mathbf{up} = (0, 1, 0)$. N is chosen to be 1.

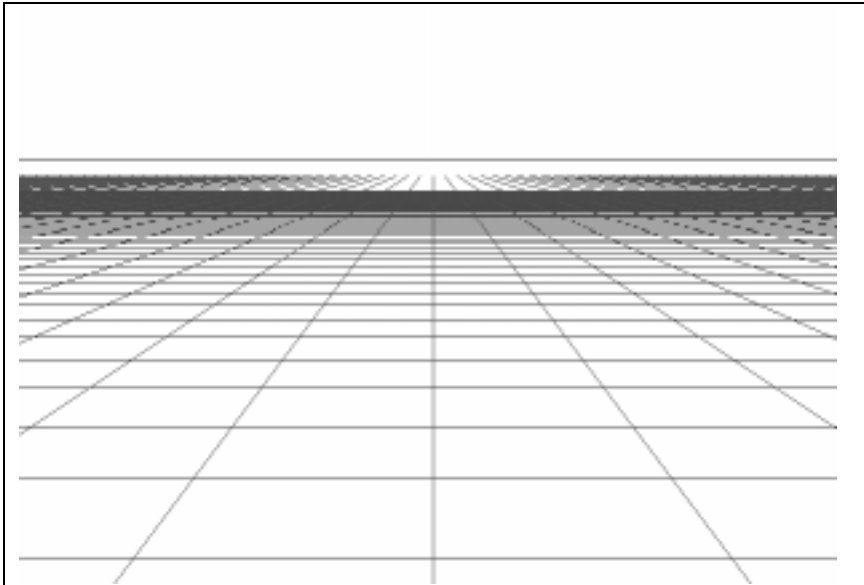


Figure 7.24. Viewing a horizontal grid on the xz -plane.

The grid lines of constant x have parametric form in eye coordinates of $(i, 0, t)$, where $i = \dots, -2, -1, 0, 1, 2, \dots$ and t varies from 0 to ∞ . By Equation 7.4 the i -th line projects to $(-i/t, 2/t)$, which is a line through the vanishing point $(0, 0)$, so all of these lines converge on the same vanishing point, as expected.

The grid lines of constant z are given by $(t, 0, -i)$, where $i = 1, 2, \dots, N$ for some N , and t varies from $-\infty$ to ∞ . These project to $(-t/i, -2/i)$, which appear as horizontal straight lines (check this). Their projections are parallel since the gridlines themselves are parallel to the viewplane. The more remote ones (larger values of i) lie closer together, providing a vivid example of perspective foreshortening. Many of the remote contours are not drawn here, as they become so crowded they cannot be drawn clearly. The **horizon** consists of all the contours where z is very large and negative; it is positioned at $y = 0$.

3). The anomaly of viewing long parallel lines.

Perspective projections seem to be a reasonable model for the way we see. But there are some anomalies, mainly because our eyes do not have planar “view screens”. The problem occurs for very long objects. Consider an experiment, for example, where you look up at a pair of parallel telephone wires, as suggested in Figure 7.25a.

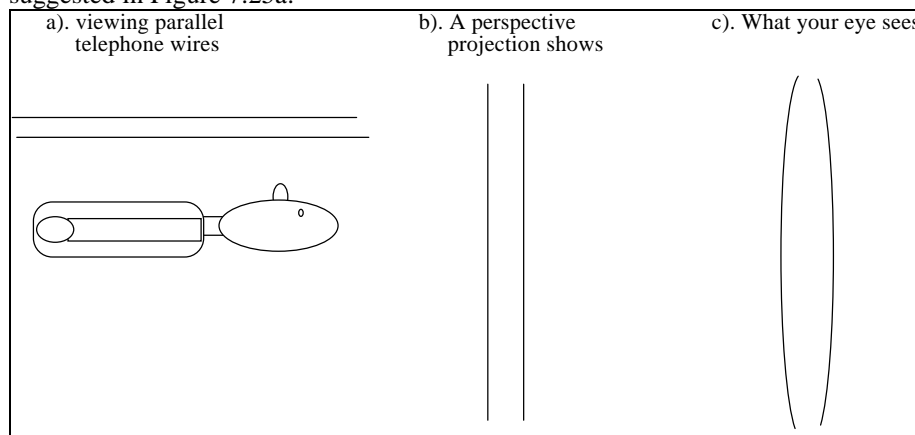


Figure 7.25. Viewing very long parallel wires. (use old 12.14).

For the perspective view, if we orient the viewplane to be parallel to the wires, we know the image will show two straight and parallel lines (part b). But what you see is quite different. The wires appear curved as they converge to “vanishing points” in both directions (part c)! In Practice this anomaly is barely

visible because the window or your eye limits the field of view to a reasonable region. (To see different parts of the wires you have to roll your eyes up and down, which of course rotates your “view planes”.)

Practice Exercises.

7.4.3. Straight lines project as straight lines: the parametric form. Show that the parametric form in Equation 7.5 is that of a straight line. Hint: For the x -component divide the denominator into the numerator to get $-A_x N/A_z + R g(t)$ where R depends on the x -components of A and \mathbf{c} , but not the y -components, and $g(t)$ is some function of t that depends on neither the x nor y -components. Repeat for the y -component, obtaining $-A_y N/A_z + S g(t)$ with similar properties. Argue why this is the parametric representation of a straight line, (albeit one for which the point does not move with constant speed as t varies).

7.4.4. Derive results for horizontal grids. Derive the parametric forms for the projected grid lines in Example 7.4.3.

7.4.3. Incorporating Perspective in the Graphics Pipeline.

Only a fool tests the depth of the river with both feet.

Paul Cezanne, 1925

We want the graphics pipeline to project vertices of 3D objects onto the near plane, then map them to the viewport. After passing through the modelview matrix, the vertices are represented in the camera’s coordinate system, and Equation 7.4 shows the values we need to compute for the proper projection. We need to do clipping, and then map what survives to the viewport. But we need a little more as well.

Adding Pseudodepth.

Taking a projection discards depth information; that is, how far the point is from the eye. But we mustn’t discard this information completely, or it will be impossible to do hidden surface removal later.

The actual distance of a point P from the eye in camera coordinates is $\sqrt{P_x^2 + P_y^2 + P_z^2}$, which would be cumbersome and slow to compute for each point of interest. All we really need is some measure of distance that tells when two points project to the *same* point on the near plane, which is the closer. Figure 7.26 shows points P_1 and P_2 that both lie on a line from the eye, and therefore project to the same point. We must be able to test whether P_1 obscures P_2 or vice versa. So for each point P that we project we compute a value called the **pseudodepth** that provides an adequate measure of depth for P . We then say that P projects to (x^*, y^*, z^*) , where (x^*, y^*) is the value provided in Equation 7.4 and z^* is its pseudodepth.

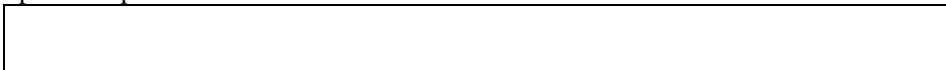


Figure 7.26. Is P_1 closer than P_2 or farther away?

What is a good choice for the pseudodepth function? Notice that if two points project to the same point the farther one always has a more negative value of P_z , so we might use $-P_z$ itself for pseudodepth. But it will turn out to be very harmonious and efficient to choose a function with the *same denominator* ($-P_z$) as occurs with x^* and y^* . So we try a function that has this denominator, and a numerator that is linear in P_z , and say that P “projects to”

$$(x^*, y^*, z^*) = \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \quad (7.7)$$

for some choice of the constants a and b . Although many different choices for a and b will do, we choose them so that the pseudodepth varies between -1 and 1 (we see later why these are good choices). Since depth increases as a point moves further down the negative z -axis, we decide that the pseudodepth is -1 when $P_z = -N$, and is $+1$ when $P_z = -F$. With these two conditions we can easily solve for a and b , obtaining:

$$a = -\frac{F + N}{F - N}, b = \frac{-2FN}{F - N} \quad (7.8)$$

Figure 7.27 plots pseudodepth versus $(-P_z)$. As we insisted it grows from -1 for a point on the near plane up to +1 for a point on the far plane. As P_z approaches 0 (so that the point is just in front of the eye) pseudodepth plummets to $-\infty$. For a point just behind the eye, the pseudodepth is huge and positive. But we will clip off points that lie closer than the near plane, so this catastrophic behavior will never be encountered.

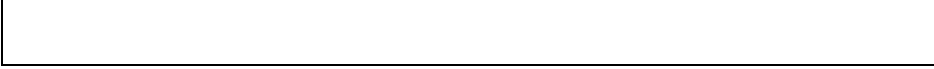


Figure 7.27. Pseudodepth grows as P_z becomes more negative.

Also notice that pseudodepth values bunch together as $(-P_z)$ gets closer to F . Given the finite precision arithmetic of a computer, this can cause a problem when you must distinguish the pseudodepth of two points during hidden surface removal: the points have different true depths from the eye, but their pseudodepths come out with the same value!

Note that defining pseudodepth as in Equation 7.7 causes it to become more positive as P_z becomes more negative. This seems reasonable since depth from the eye increases as P_z moves further along the negative z -axis.

Example 7.4.4. Pseudodepth varies slowly as $(-P_z)$ approaches F .

Suppose $N = 1$ and $F = 100$. This produces $a = -101/99$ and $b = -200/99$, so pseudodepth is given by

$$pseudodepth|_{N=1, F=100} = \frac{101P_z + 200}{99P_z}$$

This maps appropriately to -1 at $P_z = -N$, and 1 at $P_z = -F$. But close to $-F$ it varies quite slowly with $(-P_z)$. For $(-P_z)$ values of 97, 98, and 99, for instance, this evaluates to 1.041028, 1.040816, and 1.040608.

A little algebra (see the exercises) shows that when N is much smaller than F as it normally will be, pseudodepth can be approximated by

$$pseudodepth \approx 1 + \frac{2N}{P_z} \quad (7.9)$$

Again you see that it varies more and more slowly as $(-P_z)$ approaches F . But its variation is increased by using large values of N . N should be set as large as possible (but of course not so large that objects nearest to the camera are clipped off!).

Using Homogeneous Coordinates.

Why was there consideration given to having the same denominator for each term in Equation 7.7? As we now show, this makes it possible to represent all of the steps so far in the graphics pipeline as matrix multiplications, offering both processing efficiency and uniformity. (Chips on some graphics cards can multiply a point by a matrix in hardware, making this operation extremely fast!) Doing it this way will also allow us to set things up for a highly efficient and reliable clipping step.

The new approach requires that we represent points in homogeneous coordinates. We have been doing that anyway, since this makes it easier to transform a vertex by the modelview matrix. But we are going to expand the notion of the homogeneous coordinate representation beyond what we have needed before now, and therein find new power. In particular, a matrix will not only be able to perform an affine transformation, it will be able to perform a “perspective transformation.”

Up to now we have said that a point $P = (P_x, P_y, P_z)$ has the representation $(P_x, P_y, P_z, 1)$ in homogeneous coordinates, and that a vector $\mathbf{v} = (v_x, v_y, v_z)$ has the representation $(v_x, v_y, v_z, 0)$. We have simply appended a 1 or 0. This made it possible to use coordinate frames as a basis for representing the points and vectors of interest, and it allowed us to represent an affine transformation by a matrix.

Now we extend the idea, and say that a point $P = (P_x, P_y, P_z)$ has a whole family of homogeneous representations (wP_x, wP_y, wP_z, w) for any value of w except 0. For example, the point $(1, 2, 3)$ has the representations $(1, 2, 3, 1)$, $(2, 4, 6, 2)$, $(0.003, 0.006, 0.009, 0.001)$, $(-1, -2, -3, -1)$, and so forth. If

someone hands you a point in this form, say (3, 6, 2, 3) and asks what point is it, just divide through by the last component to get (1, 2, 2/3, 1), then discard the last component: the point in “ordinary” coordinates is (1, 2, 2/3). Thus:

- To convert a point from *ordinary coordinates* to *homogeneous coordinates*, append a 1⁴;
- To convert a point from *homogeneous coordinates* to *ordinary coordinates*, divide all components by the last component, and discard the fourth component.

The additional property of being able to scale all the components of a point without changing the point is really the basis for the name “homogeneous”. Up until now we have always been working with the special case where the final component is 1.

We examine homogeneous coordinates further in the exercises, but now focus on how they operate when transforming points. Affine transformations work fine when homogeneous coordinates are used. Recall that the matrix for an affine transformation always has (0,0,0,1) in its fourth row. Therefore if we multiply a point P in homogeneous representation by such a matrix M , to form $MP = Q$ (recall Equation 5.24), as in the example

$$\begin{pmatrix} 2 & -1 & 3 & 1 \\ 6 & .5 & 1 & 4 \\ 0 & 4 & 2 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wQ_x \\ wQ_y \\ wQ_z \\ w \end{pmatrix}$$

the final component of Q will always be unaltered: it is still w . Therefore we can convert the Q back to ordinary coordinates in the usual fashion.

But something new happens if we deviate from a fourth row of (0,0,0,1). Consider the important example that has a fourth row of (0, 0, -1, 0), (which is close to what we shall later call the “projection matrix”):

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad \text{(the projection matrix - version 1)} \quad (7.10)$$

for any choices of N , a , and b . Multiply this by a point represented in homogeneous coordinates with an arbitrary w :

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z + b) \\ -wP_z \end{pmatrix}$$

This corresponds to an ordinary point, but which one? Divide through by the fourth component and discard it, to obtain

$$\left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right)$$

⁴ and, if you wish, multiply all four components by any nonzero value.

which is precisely what we need according to Equation 7.7. Thus using homogeneous coordinates allows us to capture perspective using a matrix multiplication! To make it work we must always divide through by the fourth component, a step which is called **perspective division**.

A matrix that has values other than (0,0,0,1) for its fourth row does not perform an affine transformation. It performs a more general class of transformation called a **perspective transformation**. It is a transformation not a projection. A projection reduces the dimensionality of a point, to a 3-tuple or a 2-tuple, whereas a perspective transformation takes a 4-tuple and produces a 4-tuple.

Consider the algebraic effect of putting nonzero values in the fourth row of the matrix, such as (A,B,C,D) . When you multiply the matrix by $(P_x, P_y, P_z, 1)$ (or any multiple thereof) the fourth term in the resulting point becomes $AP_x + BP_y + CP_z + D$, making it linearly dependent on each of the components of P . After perspective division this term appears in the denominator of the point. Such a denominator is exactly what is needed to produce the geometric effect of perspective projection onto a general plane, as we show in the exercises.

The perspective transformation therefore carries a 3D point P into another 3D point P' , according to:

$$(P_x, P_y, P_z) \rightarrow \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \quad \text{“the perspective transformation”} \quad (7.11)$$

Where does the projection part come into play? Further along the pipeline the first two components of this point are used for drawing: to locate in screen coordinates the position of the point to be drawn. The third component is “peeled off” to be used for depth testing. As far as locating the point on the screen is concerned, ignoring the third component is equivalent to replacing it by 0, as in:

$$\left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \rightarrow \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, 0 \right) \quad \text{“the projection”} \quad (7.12)$$

This is just what we did in Chapter 5 to project a point “orthographically” (meaning perpendicularly to the viewplane) when setting up a camera for our first efforts at viewing a 3D scene. We will study orthographic projections in full detail later. For now we can conclude:

$$(\text{perspective projection}) = (\text{perspective transformation}) + (\text{orthographic projection})$$

This decomposition of a perspective projection into a specific transformation followed by a (trivial) projection will prove very useful, both algorithmically and for understanding better what each point actually experiences as it passes through the graphics pipeline. OpenGL does the transformation step separately from the projection step. In fact it inserts clipping, perspective division, and one additional mapping between them. We next look deeper into the transformation part of the process.

The Geometric Nature of the Perspective Transformation.

The perspective transformation alters 3D point P into another 3D point according to Equation 7.11, to “prepare it” for projection. It is useful to think of it as causing a “warping” of 3D space, and to see how it warps one shape into another. Very importantly, it preserves straightness and “flatness”, so lines transform into lines, planes into planes, and polygonal faces into other polygonal faces. It also preserves “in-between-ness”, so if point a is inside an object, the transformed point will also be inside the transformed object. (Our choice of a suitable pseudodepth function was guided by the need to preserve these properties.) The proof of these properties is developed in the exercises.

Of particular interest is how it transforms the camera’s view volume, because if we are going to do clipping in the warped space, we will be clipping against the warped view volume. The perspective transformation shines in this regard: the warped view volume is a perfect shape for simple and efficient clipping! Figure 7.28 suggests how the view volume and other shapes are transformed. The near plane W at $z = -N$ maps into the plane W' at $z = -1$, and the far plane maps to the plane at $z = +1$. The top wall T is “tilted” into the horizontal plane T' so that is parallel to the z -axis. The bottom wall S becomes the

horizontal S' , and the two side walls become parallel to the z -axis. The camera's view volume is transformed into a parallelepiped!



Figure 7.28. The view volume warped by the perspective transformation.

It's easy to prove how these planes map, because they all involve lines that are either parallel to the near plane or that pass through the eye. Check these carefully:

Fact: Lines through the eye map into lines parallel to the z -axis. **Proof:** All points of such a line project to a single point, say (x^*, y^*) , on the viewplane. So all of the points along the line transform to all of the points (x, y, z) with $x = x^*$, $y = y^*$, and z taking on all pseudodepth values between -1 and 1 .

Fact: Lines perpendicular to the z -axis map to lines perpendicular to the z -axis. **Proof:** All points along such a line have the same z -coordinate, so they all map to points with the same pseudodepth value.

Using these facts it is straightforward to derive the exact shape and dimensions of the warped view volume.

The transformation also “warps” objects, like the blocks shown, into new shapes. Figure 7.28b shows a block being projected onto the near plane. Suppose the top edge of its front face projects to $y = 2$ and the top edge of its back face projects to $y = 1$. When this block is transformed, it becomes a truncated pyramid: the top edge of its front face *lies at* $y = 2$, and the top edge of its back face lies at $y = 1$. Things closer to the eye than the near plane become bigger, and things beyond the near plane become smaller. The transformed object is smaller at the back than the front because the original object projects that way. The x and y -coordinates of the transformed object are the x - and y -coordinates of the *projection* of the original object. These are the coordinates you would encounter upon making an orthographic projection of the transformed object. In a nutshell:

The perspective transformation “warps” objects so that, when viewed with an orthographic projection, they appear the same as the original objects do when viewed with a perspective projection.

So all objects are warped into properly foreshortened shapes according to the rules of perspective projection. Thereafter they can be viewed with an orthographic projection, and the correct picture is produced.

We look more closely at the specific shape and dimensions of the transformed view volume.

Details of the Transformed View Volume, and mapping into the Canonical View Volume.

We want to put some numbers on the dimensions of the view volume before and after it is warped. Consider the top plane, and suppose it passes through the point (*left*, *top*, $-N$) at $z = -N$ as shown in Figure 7.29. Because it is composed of lines that pass through the eye and through points in the near plane all of which have a y -coordinate of *top*, it must transform to the plane $y = \text{top}$. Similarly,

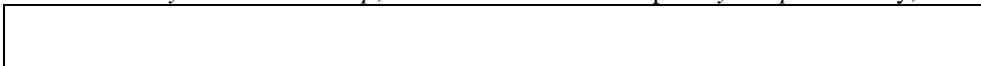


Figure 7.29. Details of the perspective transformation.

- the bottom plane transforms to the $y = \text{bott}$ plane;
- the left plane transforms to the $x = \text{left}$ plane;
- the right plane transforms to the $x = \text{right}$.

We now know the transformed view volume precisely: a parallelepiped with dimensions that are related to the camera's properties in a very simple way. This is a splendid shape to clip against as we shall see, because its walls are parallel to the coordinate planes. But it would be an even better shape for clipping if its dimensions didn't depend on the particular camera being used. OpenGL composes the perspective transformation with another mapping that scales and shifts this parallelepiped into the **canonical view volume**, a cube that extends from -1 to 1 in each dimension. Because this scales things differently in the

x - and y - dimensions as it “squashes” the scene into a fixed volume it introduces some distortion, but the distortion will be eliminated in the final viewport transformation.

The transformed view volume already extends from -1 to 1 in z , so it only needs to be scaled in the other two dimensions. We therefore include a scaling and shift in both x and y to map the parallelepiped into the canonical view volume. We first shift by $-(right + left)/2$ in x and by $-(top + bott)/2$ in y . Then we scale by $2/(right - left)$ in x and by $2/(top - bott)$ in y . When the matrix multiplications are done (see the exercises) we obtain the final matrix:

$$R = \begin{pmatrix} \frac{2N}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2N}{top - bott} & \frac{top + bott}{top - bott} & 0 \\ 0 & 0 & \frac{-(F + N)}{F - N} & \frac{-2FN}{F - N} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad \text{(the projection matrix)} \quad (7.13)$$

This is known as the **projection matrix**, and it performs the perspective transformation plus a scaling and shifting to transform the camera’s view volume into the canonical view volume. It is precisely the matrix that OpenGL creates (and by which it multiplies the current matrix) when `glFrustum(left, right, bott, top, N, F)` is executed. Recall that `gluPerspective(viewAngle, aspect, N, F)` is usually used instead, as its parameters are more intuitive. This sets up the same matrix, after computing values for top , $bott$, etc. using

$$top = N \tan\left(\frac{\pi}{180} \text{viewAngle} / 2\right)$$

$bott = -top$, $right = top * aspect$, and $left = -right$.

Clipping Faces against the View Volume.

Recall from Figure 7.14 that clipping is performed after vertices have passed through the projection matrix. It is done in this warped space because the canonical view volume is particularly well suited for efficient clipping. Here we show how to exploit this, and we develop the details of the clipping algorithm.

Clipping in the warped space works because a point lies inside the camera’s view volume if and only if its transformed version lies inside the canonical view volume. Figure 7.30a shows an example of clipping in action. A triangle has vertices v_1 , v_2 , and v_3 . Vertex v_3 lies outside the canonical view volume, CVV. The clipper works on edges: it first clips edge v_1v_2 , and finds that the entire edge lies inside CVV. Then it clips edge v_2v_3 , and records the new vertex a formed where the edge exits from the CVV. Finally it clips edge v_3v_1 and records the new vertex where the edge enters the CVV. At the end of the process the original triangle has become a quadrilateral with vertices v_1v_2ab . (We will see later that in addition to identifying the locations of the new vertices, the pipeline also computes new color and texture parameters at these new vertices.)

a). clip a triangle

b). clip an edge

Figure 7.30. Clipping against the canonical view volume.

The clipping problem is basically the problem of clipping a line segment against the CVV. We examined such an algorithm, the Cyrus-Beck clipper, in Section 4.8.3. The clipper we develop here is similar to that one, but of course it works in 3D rather than 2D.

Actually, it works in 4D. We will clip in the 4D homogeneous coordinate space called “clip coordinates” in Figure 7.14. This is easier than it might seem, and it will nicely distinguish between points in front of, and behind, the eye.

Suppose we want to clip the line segment AC shown in Figure 7.30b against the CVV. This means we are given two points in homogeneous coordinates, $A = (a_x, a_y, a_z, a_w)$ and $C = (c_x, c_y, c_z, c_w)$, and we want to

determine which part of the segment lies inside the CVV. If the segment intersects the boundary of the CVV we will need to compute the intersection point $I = (I_x, I_y, I_z, I_w)$.

As with the Cyrus-Beck algorithm we view the CVV as six infinite planes, and consider where the given edge lies relative to each plane in turn. We can represent the edge parametrically as $A + (C-A)t$. It lies at A when t is 0, and at C when t is 1. For each wall of the CVV we first test whether A and C lie on the same side of a wall: if they do there is no need to compute the intersection of the edge with the wall. If they lie on opposite sides we locate the intersection point and clip off the part of the edge that lies outside.

So we must be able to test whether a point is on the “outside” or “inside” of a plane. Take the plane $x = -1$, for instance, which is one of the walls of the CVV. The point A lies to the right of it (on the “inside”) if

$$\frac{a_x}{a_w} > -1 \quad \text{or} \quad a_x > -a_w \quad \text{or} \quad (a_w + a_x) > 0. \quad (7.14)$$

(When you multiply both sides of an inequality by a negative term you must reverse the direction of the inequality. But we are ultimately dealing with only positive values of a_w here - see the exercises.) Similarly A is inside the plane $x = 1$ if

$$\frac{a_x}{a_w} > 1 \quad \text{or} \quad (a_w - a_x) > 0$$

Blinn [blinn96] calls these quantities the “boundary coordinates” of point A , and he lists the six such quantities that we work with as in Figure 7.31:

<i>boundary coordinate</i>	<i>homogeneous value</i>	<i>clip plane</i>
BC_0	$w+x$	$X=-1$
BC_1	$w-x$	$X=1$
BC_2	$w+y$	$Y=-1$
BC_3	$w-y$	$Y=1$
BC_4	$w+z$	$Z=-1$
BC_5	$w-z$	$Z=1$

Figure 7.31. The boundary codes computed for each end point of an edge.

We form these six quantities for A and again for C . If all six are positive the point lies inside the CVV. If any are negative the point lies outside. If both points lie inside we have the same kind of “trivial accept” we had in the Cohen Sutherland clipper of Section 3.3. If A and C lie outside on the same side (corresponding BC ’s are negative) the edge must lie wholly outside the CVV.

Trivial accept: both endpoints lie inside the CVV (all 12 BC’s are positive)

Trivial reject: both endpoints lie outside the same plane of the CVV.

If neither condition prevails we must clip segment AC against each plane individually. Just as with the Cyrus-Beck clipper, we keep track of a **candidate interval** (CI) (see Figure 4.45), an interval of time during which the edge might still be inside the CVV. Basically we know the converse: if t is outside the CI we know for sure the edge is *not* inside the CVV. The CI extends from $t = t_{in}$ to t_{out} .

We test the edge against each wall in turn. If the corresponding boundary codes have opposite signs we know the edge hits the plane at some t_{hit} , which we then compute. If the edge “is entering” (is moving into the “inside” of the plane as t increases) we update $t_{in} = \max(\text{old } t_{in}, t_{hit})$, since it could not possibly be entering at an earlier time than t_{hit} . Similarly, if the edge is exiting, we update $t_{out} = \min(\text{old } t_{out}, t_{hit})$. If at any time the CI is reduced to the empty interval (t_{out} becomes $> t_{in}$) we know the entire edge is clipped off and we have an “early out”, which saves unnecessary computation.

It is straightforward to calculate the hit time of an edge with a plane. Write the edge parametrically in homogeneous coordinates:

$$edge(t) = (a_x + (c_x - a_x)t, a_y + (c_y - a_y)t, a_z + (c_z - a_z)t, a_w + (c_w - a_w)t)$$

If it's the $X = 1$ plane, for instance, when the x -coordinate of $A + (C-A)t$ is 1:

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - a_w)t} = 1$$

This is easily solved for t , yielding

$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)} \quad (7.15)$$

Note that t_{hit} depends on only two boundary coordinates. Intersection with other planes yield similar formulas.

This is easily put into code, as shown in Figure 7.32. This is basically the Liang Barsky algorithm [liang84], with some refinements suggested by Blinn [blinn96]. The routine `clipEdge(Point4 A, Point4 C)` takes two points in homogeneous coordinates (having fields x, y, z , and w) and returns 0 if no part of AC lies in the CVV, and 1 otherwise. It also alters A and C so that when the routine is finished A and C are the endpoints of the clipped edge.

The routine finds the six boundary coordinates for each endpoint and stores them in `aBC[]` and `cBC[]`. For efficiency it also builds an **outcode** for each point, which holds the *signs* of the six boundary codes for that point. Bit i of A 's outcode holds a 0 if `aBC[i] > 0` (A is inside the i -th wall) and a 1 otherwise. Using these, a trivial accept occurs when both `aOutcode` and `cOutcode` are 0. A trivial reject occurs when the bit-wise AND of the two outcodes is nonzero.

```
int clipEdge(Point4& A, Point4& C)
{
    double tIn = 0.0, tOut = 1.0, tHit;
    double aBC[6], cBC[6];
    int aOutcode = 0, cOutcode = 0;
    <.. find BC's for A and C ..>
    <.. form outcodes for A and C ..>

    if((aOutcode & cOutcode) != 0) // trivial reject
        return 0;
    if((aOutcode | cOutcode) == 0) // trivial accept
        return 1;

    for(int i = 0; i < 6; i++) // clip against each plane
    {
        if(cBC[i] < 0) // exits: C is outside
        {
            tHit = aBC[i]/(aBC[i] - cBC[i]);
            tOut = MIN(tOut, tHit);
        }
        else if(aBC[i] < 0) //enters: A is outside
        {
            tHit = aBC[i]/(aBC[i] - cBC[i]);
            tIn = MAX(tIn, tHit);
        }
        if(tIn > tOut) return 0; //CI is empty early out
    }
    // update the end points as necessary
    Point4 tmp;
    if(aOutcode != 0) // A is out: tIn has changed
    { // find updated A, (but don't change it yet)
        tmp.x = A.x + tIn * (C.x - A.x);
        tmp.y = A.y + tIn * (C.y - A.y);
        tmp.z = A.z + tIn * (C.z - A.z);
        tmp.w = A.w + tIn * (C.w - A.w);
```



```

    }
    if(cOutcode != 0) // C is out: tOut has changed
    { // update C (using original value of A)
        C.x = A.x + tOut * (C.x - A.x);
        C.y = A.y + tOut * (C.y - A.y);
        C.z = A.z + tOut * (C.z - A.z);
        C.w = A.w + tOut * (C.w - A.w);
    }
    A = tmp; // now update A
    return 1; // some of the edge lies inside the CVV
}

```

Figure 7.32. The edge clipper (as refined by Blinn).

In the loop that tests the edge against each plane, at most one of the BC 's can be negative. (Why?) If A has negative BC the edge must be entering at the hit point; if C has a negative BC the edge must be exiting at the hit point. (Why?) (Blinn uses a slightly faster test by incorporating a mask that tests one bit of an outcode.) Each time t_{In} or t_{Out} are updated an early out is taken if t_{In} has become greater than t_{Out} .

When all planes have been tested, one or both of t_{In} and t_{Out} have been altered (why?). A is updated to $A + (i - A) t_{In}$ if t_{In} has changed, and C is updated to $A + (C - A) t_{Out}$ if t_{Out} has changed.

Blinn suggests pre-computing the BC 's and outcode for every point to be processed. This eliminates the need to re-compute these quantities when a vertex is an endpoint of more than one edge, as is often the case.

Why did we clip against the canonical view volume?

Now that we have seen how easy it is to do clipping against the canonical view volume, we can see the value of having transformed all objects of interest into it prior to clipping. There are two important features of the CVV:

1. It is parameter-free: the algorithm needs no extra information to describe the clipping volume. It uses only the values -1 and 1. So the code itself can be highly tuned for maximum efficiency.
2. Its planes are aligned with the coordinate axes (after the perspective transformation is performed). This means that we can determine which side of a plane a point lies on using a single coordinate, as in $a_x > -1$. If the planes were not aligned, an expensive dot product would be needed.

Why did we clip in homogeneous coordinates, rather than after the perspective division step?

This isn't completely necessary, but it makes the clipping algorithm clean, fast, and simple. Doing the perspective divide step destroys information: if you have the values a_x and a_w explicitly you know of course the signs of both of them. But given only the ratio a_x/a_w you can tell only whether a_x and a_w have the same or opposite signs. Keeping values in homogeneous coordinates and clipping points closer to the eye than the near plane automatically removes points that lie behind the eye, such as B in Figure 7.23.

Some "perverse" situations that necessitate clipping in homogeneous coordinates are described in [blinn96, foley90]. They involve peculiar transformations of objects, or construction of certain surfaces, where the original point (a_x, a_y, a_z, a_w) has a negative fourth term, even though the point is in front of the eye. None of the objects we discuss modeling here involve such cases. We conclude that clipping in homogeneous coordinates, although usually not critical, makes the algorithm fast and simple, and brings it almost no cost.

Following the clipping operation **perspective division** is finally done, (as in Figure 7.14), and the 3-tuple (x, y, z) is passed through the viewport transformation. As we discuss next, this transformation sizes and shifts the x - and y - values so they are placed properly in the viewport, and makes minor adjustments on the z - component (pseudodepth) to make it more suitable for depth testing.

The Viewport Transformation.

As we have seen the perspective transformation squashes the scene into the canonical cube, as suggested in Figure 7.33. If the aspect ratio of the camera's view volume (that is, the aspect ratio of the window on the near plane) is 1.5, there is obvious distortion introduced when the perspective transformation scales objects into a window with aspect ratio 1. But the viewport transformation can undo this distortion by mapping a square into a viewport of aspect ratio 1.5. We normally set the aspect ratio of the viewport to be the same as that of the view volume.



Figure 7.33. The viewport transformation restores aspect ratio.

We have encountered the OpenGL function `glViewport(x, y, wid, ht)` often before. It specifies that the viewport will have lower left corner (x, y) in screen coordinates, and will be `wid` pixels wide and `ht` pixels high. It thus specifies a viewport with aspect ratio `wid/ht`. The viewport transformation also maps pseudodepth from the range -1 to 1 into the range 0 to 1.

Review Figure 7.14, which reveals the entire graphics pipeline. Each point P (which is usually one vertex of a polygon) is passed through the following steps:

- P is **extended** to a homogeneous 4-tuple by appending a 1;
- This 4-tuple is multiplied by the **modelview matrix**, producing a 4-tuple giving the position in eye coordinates;
- The point is then multiplied by the **projection matrix**, producing a 4-tuple in clip coordinates;
- The edge having this point as an endpoint is **clipped**;
- **Perspective division** is performed, returning a 3-tuple;
- The **viewport transformation** multiplies the 3-tuple by a matrix: the result (sx, sy, dz) is used for drawing and depth calculations. (sx, sy) is the point in screen coordinates to be displayed; dz is a measure of the depth of the original point from the eye of the camera.

Practice Exercises.

7.4.5. P projects where? Suppose the viewplane is given in camera coordinates by the equation $Ax + By + Cz = D$. Show that any point P projects onto this plane at the point given in homogeneous coordinates

$$P' = (DP_x, DP_y, DP_z, AP_x + BP_y + CP_z)$$

Do it using these steps.

- a). Show that the projected point is the point at which the ray between the eye and P hits the given plane.
- b). Show that the ray is given by Pt , and it hits the plane at $t^* = D/(AP_x + BP_y + CP_z)$.
- c). Show that the projected point - the hit point - is therefore given properly above.
- d). Show that for the near plane we use earlier, we obtain (x^*, y^*) as given by Equation 7.4.

7.4.6. A revealing approximate form for pseudodepth. Show that pseudodepth $a + b/(-P_z)$, where a and b are given in Equation 7.8, is well approximated by Equation 7.9 when N is much smaller than F .

7.4.7. Points at infinity in homogeneous coordinates. Consider the nature of the homogeneous coordinate point (x, y, z, w) as w becomes smaller and smaller. For $w = .01$ it is $(100x, 100y, 100z)$, for $w = 0.0001$ it is $(10000x, 10000y, 10000z)$, etc. It progresses out "toward infinity" in the direction (x, y, z) . The point with representation $(x, y, z, 0)$ is in fact called a "point at infinity". It is one of the advantages of homogeneous coordinates that such an idealized point has a perfectly "finite" representation: in some mathematical derivations this removes many awkward special cases. For instance, two lines will always intersect, even if they are parallel [ayers67, semple52]. But other things don't work as well. What is the difference of two points in homogeneous coordinates?

7.4.8. How does the perspective transformation affect lines and planes?

We must show that the perspective transformation preserves flatness and in-between-ness.

- a). Argue why this is proven if we can show that a point P lying on the line between two points A and B transforms to a point P' that lies between the transformed versions of A and B .
- b). Show that the perspective transformation does indeed produce a point P' with the property just stated.
- c). Show that each plane that passes through the eye maps to a plane that is parallel to the z -axis.
- d). Show that each plane that is parallel to the z -axis maps to a plane parallel to the z -axis.
- e). Show that relative depth is preserved;

7.4.9. The details of the transformed view volume. Show that the warped view volume has the dimensions given in the five points above. You can use the facts developed in the preceding exercise.

7.4.10. Show the final form of the projection matrix. The projection matrix is basically that of Equation 7.10, followed by the shift and scaling described. If the matrix of Equation 7.10 is denoted as M , and T represents the shifting matrix, and S the scaling matrix, show that the matrix product STM is that given in Equation 7.13.

7.4.11. What Becomes of Points Behind the Eye? If the perspective transformation moves the eye off to $-\infty$, what happens to points that lie behind the eye? Consider a line, $P(t)$, that begins at a point in front of the eye at $t = 0$ and moves to one behind the eye at $t = 1$.

- a). Find its parametric form in homogeneous coordinates;
 - b). Find the parametric representation after it undergoes the perspective transformation;
 - c). Interpret it geometrically. Specifically state what the fourth homogeneous coordinate is geometrically.
- A valuable discussion of this phenomenon is given in [blinn78].