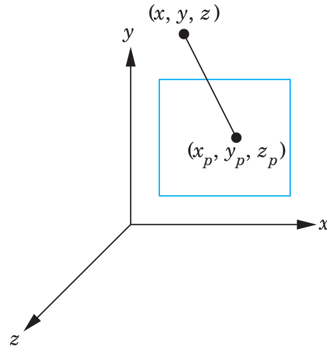

Figure 5.37 Oblique projection.

$$\mathbf{P} = \begin{bmatrix} \frac{2}{left-right} & 0 & 0 & -\frac{left+right}{left-right} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Because the camera is pointing in the negative z direction, the projectors are directed from infinity on the negative z-axis toward the origin.

5.8.3 Oblique Projections

OpenGL provides through `glOrtho` a limited class of parallel projections—namely, only those for which the projectors are orthogonal to the projection plane. As we saw earlier in this chapter, oblique parallel projections are useful in many fields.[4] We could develop an oblique projection matrix directly; instead, however, we follow the process that we used for the general orthogonal projection. We convert the desired projection to a canonical orthogonal projection of distorted objects.

An oblique projection can be characterized by the angle that the projectors make with the projection plane, as shown in Figure 5.37. In APIs that support general parallel viewing, the view volume for an oblique projection has the near and far clipping planes parallel to the view plane, and the right, left, top, and bottom planes parallel to the direction of projection, as shown in Figure 5.38. We can derive the equations for oblique projections

4. Note that without oblique projections we cannot draw coordinate axes in the way that we have been doing in this book; see Exercise 5.15.

by considering the top and side views in Figure 5.39, which shows a projector and the projection plane $z = 0$. The angles θ and ϕ characterize the degree of obliqueness. In drafting, projections such as the cavalier and cabinet projections are determined by specific values of these angles. However, these angles are not the only possible interface (see Exercises 5.9 and 5.10).

If we consider the top view, we can find x_p by noting that

$$\tan\theta = \frac{z}{x - x_p},$$

and thus

$$x_p = x - z\cot\theta.$$

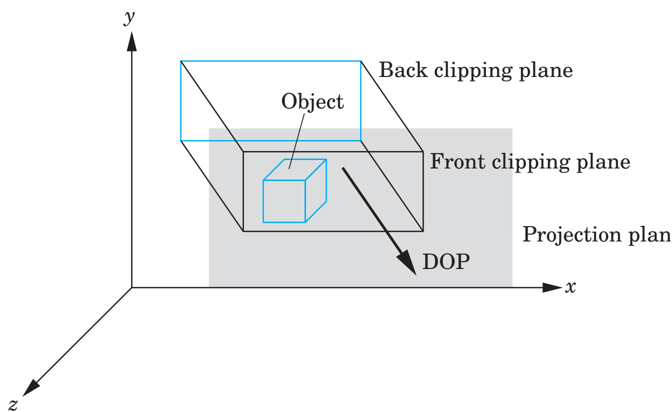

Figure 5.38 Oblique clipping volume.

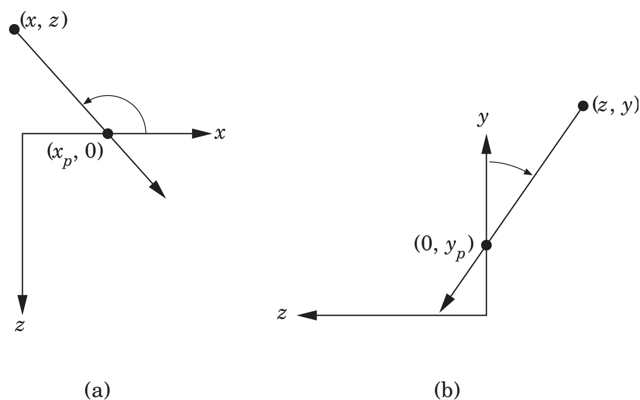

Figure 5.39 Oblique projection. (a) Top view. (b) Side view.

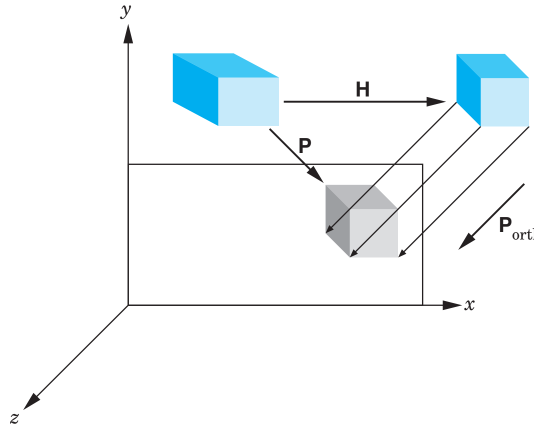

Figure 5.40 Effect of shear transformation.

Likewise,

$$y_p = y - z \cot \phi.$$

Using the equation for the projection plane

$$z_p = 0,$$

we can write these results in terms of a homogeneous-coordinate matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Following our strategy of the previous example, we can break **P** into the product

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{H}(\theta, \phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $\mathbf{H}(\theta, \phi)$ is a shearing matrix. Thus, we can implement an oblique projection by first doing a shear of the objects by $\mathbf{H}(\theta, \phi)$, and then doing an orthographic projection. Figure 5.40 shows the effect of $\mathbf{H}(\theta, \phi)$ on an object—a cube—inside an oblique view volume. The sides of the clipping volume become orthogonal to the view plane, but the sides of the cube become oblique as they are affected by the same shear transformation. However, the orthographic projection of the distorted cube is identical to the oblique projection of the undistorted cube.

We are not finished, because the view volume created by the shear is not our canonical view volume. We have to apply the same scaling and translation matrices that we used in Section 5.8.1. Hence, the transformation

$$\mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{z_{\max}-z_{\min}} & +\frac{far+near}{near-far} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

must be inserted after the shear and before the final orthographic projection, so the final matrix is

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

The values of *left*, *right*, *bottom*, and *top* are the vertices of the right parallelepiped view volume created by the shear. These values depend on how the sides of the original view volume are communicated through the application program; they may have to be determined from the results of the shear to the corners of the original view volume.

5.9 Perspective-Projection Matrices

For perspective projections, we follow a path similar to the one that we used for parallel projections: We find a transformation that, by distorting the vertices of our objects, allows us to do a simple canonical projection to obtain the desired image. Our first step is to decide what this canonical viewing volume should be. We then introduce a new transformation, the **perspective-normalization transformation**, that converts a perspective projection to an orthogonal projection. Finally, we derive the perspective-projection matrix used in OpenGL.

5.9.1 Perspective Normalization

In Section 5.4 we introduced a simple perspective-projection matrix that, for the projection plane at $z = -1$ and the center of projection at the origin, is

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

Suppose that to form an image, we also need to specify a clipping volume. Suppose that we fix the angle of view at 90 degrees by making the sides of the viewing volume intersect the projection plane at a 45-degree angle. Equivalently, the view volume is the semi-infinite view pyramid formed by the planes

$$x = \pm z,$$
$$y = \pm z,$$

shown in Figure 5.41. We can make the volume finite by specifying the near plane to be $z = max$ and the far plane to be $z = z_{\text{min}}$, where both these values are negative (the near and far distances) and

$$z_{\text{max}} > z_{\text{min}}.$$

Consider the matrix

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

which is similar to **M** but is nonsingular. For now, we leave α and β unspecified (but nonzero). If we apply **N** to the homogeneous-coordinate point $\mathbf{p} = [\, x \quad y \quad z \quad 1 \,]^T$, we obtain the new point $\mathbf{q} = [\, x' \quad y' \quad z' \quad w' \,]^T$, where

$$x' = x,$$
$$y' = y,$$
$$z' = \alpha z + \beta,$$
$$w' = -z.$$

After dividing by w', we have the three-dimensional point

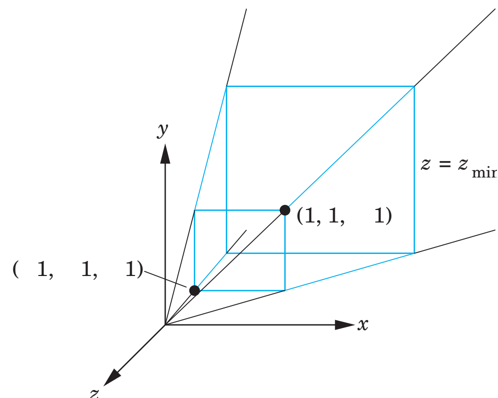

Figure 5.41 Simple perspective projection.

$$x'' = -\frac{x}{z},$$
$$y'' = -\frac{y}{z},$$
$$z'' = -\left(\alpha + \frac{\beta}{z}\right).$$

If we apply an orthographic projection along the z-axis to $\mathbf{N}$, we obtain the matrix

$$\mathbf{M}_{\text{orth}}\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

which is a simple perspective-projection matrix, and the projection of the arbitrary point $\mathbf{p}$ is

$$\mathbf{p}' = \mathbf{M}_{\text{orth}}\mathbf{N}\mathbf{p} = \begin{bmatrix} x \\ y \\ 0 \\ -z \end{bmatrix}.$$

After we do the perspective division, we obtain the desired values for x_p and y_p:

$$x_p = -\frac{x}{z},$$
$$y_p = -\frac{y}{z}.$$

We have shown that we can apply a transformation $\mathbf{N}$ to points, and, after an orthogonal projection, we obtain the same result as we would have for a perspective projection. This process is similar to how we converted oblique projections to orthogonal projections by first shearing the objects.

The matrix $\mathbf{N}$ is nonsingular and transforms the original viewing volume into a new volume. We choose α and β such that the new volume is the canonical clipping volume. Consider the sides

$$x = \pm z.$$

They are transformed by $x'' = -x/z$ to the planes

$$x'' = \pm 1.$$

Likewise, the sides $y = \pm z$ are transformed to

$$y'' = \pm 1.$$

The front of the view volume $z = z_{\text{max}}$ is transformed to the plane

$$z'' = -\left(\alpha + \frac{\beta}{z_{\text{max}}}\right).$$

Finally, the far plane $z = z_{\text{min}}$ is transformed to the plane

$$z'' = -\left(\alpha + \frac{\beta}{z_{\text{min}}}\right).$$

If we select

$$\alpha = \frac{z_{\text{max}} + z_{\text{min}}}{z_{\text{max}} - z_{\text{min}}},$$

$$\beta = -\frac{2z_{\text{max}}z_{\text{min}}}{z_{\text{max}} - z_{\text{min}}},$$

then the plane $z = z_{\text{min}}$ is mapped to the plane $z'' = -1$, the plane $z = z_{\text{max}}$ is mapped to the plane $z'' = 1$, and we have our canonical clipping volume. Figure 5.42 shows this transformation and the distortion to a cube within the volume. Thus, **N** has transformed the viewing frustum to a right parallelepiped, and an orthographic projection in the transformed volume yields the same image as does the perspective projection. The matrix **N** is called the **perspective-normalization matrix**. The mapping

$$z'' = -\left(\alpha + \frac{\beta}{z}\right)$$

is nonlinear but preserves the ordering of depths. Thus, if z_1 and z_2 are the depths of two points within the original viewing volume and

$$z_1 > z_2,$$

then their transformations satisfy

$$z_1'' > z_2''.$$

Consequently, hidden-surface removal works in the normalized volume, although the nonlinearity of the transformation can cause numerical problems because the depth buffer has a limited resolution, usually 24 or 32 bits. Note

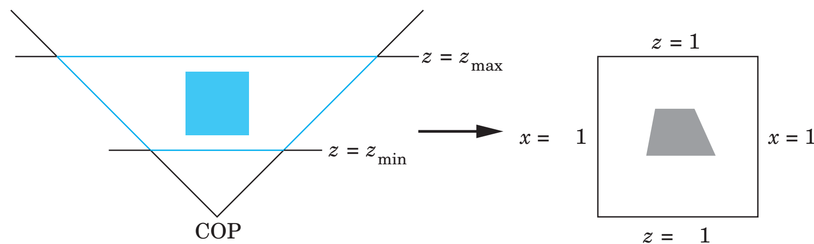

Figure 5.42 Perspective normalization of view volume.

that, although the original projection plane we placed at $z = -1$ has been transformed by $\mathbf{N}$ to the plane $z'' = \beta - \alpha$, there is little consequence to this result because we follow $\mathbf{N}$ by an orthographic projection.

Although we have shown that both perspective and parallel transformations can be converted to orthographic transformations, the effects of this conversion are greatest in implementation. As long as we can put a carefully chosen projection matrix in the pipeline before the vertices are defined, we need only one viewing pipeline for all possible views. In Chapter 8, where we discuss implementation in detail, we shall see how converting all view volumes to right parallelepipeds by our normalization process simplifies both clipping and hidden-surface removal.

5.9.2 OpenGL Perspective Transformations

The OpenGL function `glFrustum` does not restrict the view volume to a symmetric (or right) frustum. The parameters are as shown in Figure 5.43. We can form the OpenGL perspective matrix by first converting this frustum to the symmetric frustum with 45-degree sides (see Figure 5.41). The process is similar to the conversion of an oblique parallel view to an orthogonal view. First, we do a shear to convert the asymmetric frustum to a symmetric one. Figure 5.43 shows the desired transformation. The shear angle is determined by our desire to skew (shear) the point $((\mathit{left} + \mathit{right})/2, (\mathit{top} + \mathit{bottom})/2, -\mathit{far})$ to $(0, 0, -\mathit{near})$. The required shear matrix is

$$\mathbf{H}(\theta, \phi) = \mathbf{H}\left(\cot^{-1}\left(\frac{\mathit{left} + \mathit{right}}{2z_{\max}}\right), \cot^{-1}\left(\frac{\mathit{top} + \mathit{bottom}}{2z_{\max}}\right)\right).$$

The resulting frustum is described by the planes

$$x = \pm\frac{\mathit{right} - \mathit{left}}{2z_{\max}},$$

$$y = \pm\frac{\mathit{top} - \mathit{bottom}}{2z_{\max}},$$

$$z = z_{\min},$$

$$z = z_{\max}.$$

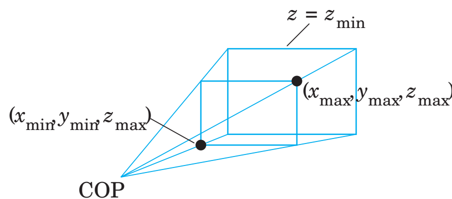

Figure 5.43 OpenGL perspective.

The next step is to scale the sides of this frustum to

$$x = \pm z,$$

$$y = \pm z,$$

without changing either the near plane or the far plane. The required scaling matrix is $\mathbf{S}(2z_{\max}/(right - left), 2z_{\max}/(top - bottom), 1)$. Note that this transformation is determined uniquely without reference to the location of the far plane $z = z_{\min}$, because, in three dimensions, an affine transformation is determined by the results of the transformation on four points. In this case, these points are the four vertices where the sides of the frustum intersect the near plane.

To get the far plane to the plane $z = -1$ and the near plane to $z = 1$ after applying a projection normalization, we use the projection-normalization matrix $\mathbf{N}$:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

with α and β as in Section 5.9.1. The resulting projection matrix is in terms of the near and far distances,

$$\mathbf{P} = \mathbf{NSH} = \begin{bmatrix} \frac{-2far}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{-2far}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

5.10 Projections and Shadows

The creation of simple shadows is an interesting application of projection matrices. Although shadows are not geometric objects, they are important components of realistic images and give many visual clues to the spatial relationships among the objects in a scene. Starting from a physical point of view, shadows require one or more light sources to be present. A point is in shadow if it is not illuminated by any light source, or equivalently if a viewer at that point cannot see any light sources. However, if the only light source is at the center of projection, there are no visible shadows, because any shadows are behind visible objects. This lighting strategy has been called the "flashlight in the eye" model and corresponds to the simple lighting we have used thus far.

To add physically correct shadows, we must understand the interaction between light and materials, a topic that we investigate in Chapter 6. There,

we show that such calculations are difficult; normally, they cannot be done in real time.

Nevertheless, the importance of shadows in applications such as flight simulators led to a number of special approaches that can be used in many circumstances. Consider the shadow generated by the point source in Figure 5.44. We assume for simplicity that the shadow falls on the ground or the surface

$$y = 0.$$

Not only is the shadow a flat polygon, called a **shadow polygon**, but also it is the projection of the original polygon onto the surface. Specifically, the shadow polygon is the projection of the polygon onto the surface with the center of projection at the light source. Thus, if we do a projection from a frame in which the light source is at the origin, we obtain the vertices of the shadow polygon. These vertices must then be converted back to a representation in the world frame. Rather than do the work as part of an application program, we can find a suitable projection matrix and use OpenGL to compute the vertices of the shadow polygon.

Suppose that we start with a light source at (x_l, y_l, z_l), as in Figure 5.45(a). If we reorient the figure such that the light source is at the origin, as in Figure 5.45(b), by a translation matrix $\mathbf{T}(-x_l, -y_l, -z_l)$, then we have a simple perspective projection through the origin. The projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$

Finally, we translate everything back with $\mathbf{T}(x_l, y_l, z_l)$. The concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l},$$

$$y_p = 0,$$

$$z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}.$$

However, with an OpenGL program, we can alter the model-view matrix to form the desired polygon,

```
GLfloat m[16];              /* Shadow Projection Matrix */
for(i=0;i<15;i++) m[i]=0.0;
m[0]=m[5]=m[10]=1.0;
m[7]= -1.0/yl;

glColor3fv(polygon_color)
```

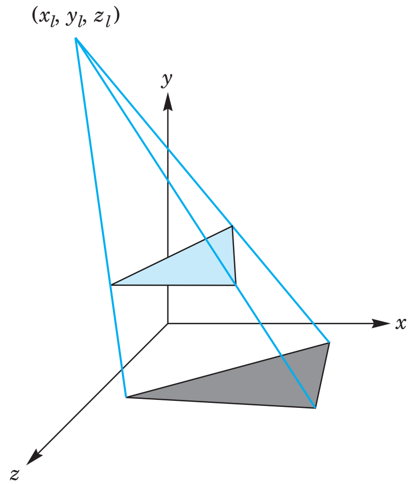

Figure 5.44 Shadow from a single polygon.

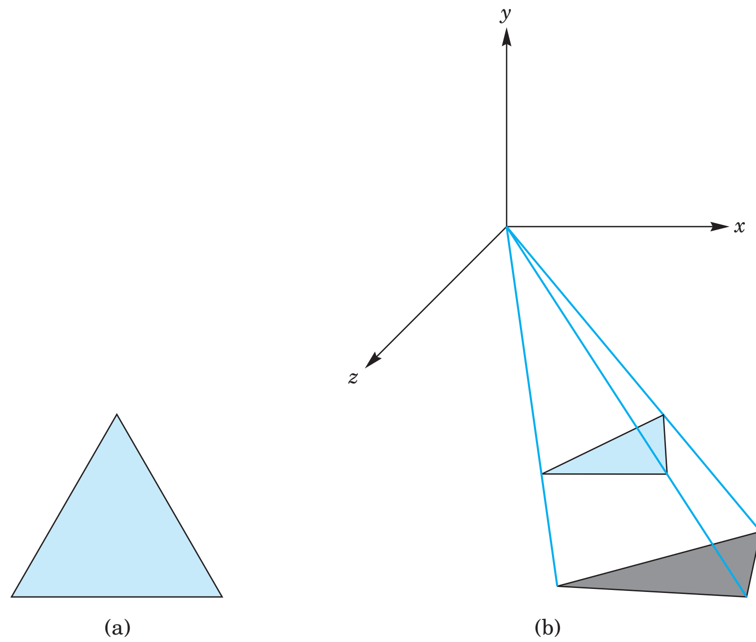

Figure 5.45 Shadow polygon projection (a) from a light source, and (b) with source moved to the origin.

```
glBegin(GL_POLYGON)
.
.                            /* Draw the polygon normally */
.
glEnd();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();              /* Save state */
glTranslatef(xl,yl,zl);      /* Translate back */
glMultMatrixf(m);            /* Project */
glTranslate(-xl,-yl,-zl);    /* Move light to origin */
glColor3fv(shadow_color);
glBegin(GL_POLYGON);
.
.                            /* Draw the polygon again */
.
glEnd();
glPopMatrix();               /* Restore state */
```

Note that although we are performing a projection with respect to the light source, the matrix that we use is the model-view matrix. We render the same polygon twice: the first time as usual and the second time with an altered model-view matrix that transforms the vertices. The same viewing conditions are applied to both the polygon and its shadow polygon. The results of computing shadows for the cube are shown in Back Plate 3. The code is in the program cubes.c.

For a simple environment, such as an airplane flying over terrain casting a single shadow, this technique works well. It is also easy to convert from point sources to distant (parallel) light sources (see Exercise 5.17). However, when objects can cast shadows on other objects, this method becomes impractical. In Chapter 13 we address more general, but slower, rendering methods that will create shadows automatically as part of the rendering process.

5.11 Summary and Notes

We have come a long way. We can now write complete, nontrivial, three-dimensional applications. Probably the most instructive activity that you can do now is to write such an application. Facility with manipulating the model-view and projection functions takes practice.

We have presented the mathematics of the standard projections. Although most APIs obviate the user from writing projection functions, understanding the mathematics leads to understanding a pipeline implementation based on concatenation of 4×4 matrices. Until recently, user programs had to do the projections within the applications, and most hardware systems did not support perspective projections.

There are three major themes in the remainder of this book. First, we explore modeling further by expanding our basic set of primitives. In Chapter 9, we incorporate more complex relationships between simple objects through

hierarchical models. In Chapter 10, we leave the world of flat objects, adding curves and curved surfaces. These objects are defined by vertices, and we can implement them by breaking them into small flat primitives, so we can use the same viewing pipeline. Then, in Chapter 11, we explore approaches to modeling that do not force us to describe objects through procedures rather than as geometric objects. This approach allows us to model objects with only as much detail as is needed, to incorporate physical laws into our models, and to model natural phenomena that cannot be described by polygons.

The second major theme is realism. Although more complex objects allow us to build more realistic models, we also explore more complex rendering options. In Chapter 6 we consider the interaction of light with the materials that characterize our objects. We look more deeply at hidden-surface–removal methods, at shading models, and, in Chapter 7, at techniques such as texture mapping that allow us to create complex images from simple objects using advanced rendering techniques.

Third, we look more deeply at implementation in Chapter 8. At this point, we have introduced the major functional units of the graphics pipeline. We discuss the details of the algorithms used in each unit. We shall also see additional possibilities for creating images by working directly in the frame buffer.

After reading Chapter 6, you should be able to read the remaining chapters in any order.

5.12 Suggested Readings

Carlbom and Paciorek [Car78] discuss the relationships between classical and computer viewing. Rogers and Adams [Rog90] give many examples of the projection matrices corresponding to the standard views used in drafting. Foley [Fol90], Watt [Wat00], and Hearn and Baker [Hea94] derive canonical projection transformations. All follow a PHIGS orientation, so the API is slightly different from the one used here, although Foley derives the most general case. The references differ in whether they use column or row matrices, in where the COP is located, and in whether the projection is in the positive or negative z direction. See the *OpenGL Programmer's Guide* [Ope01a] for a further discussion of the use of the model-view and projection matrices in OpenGL.

Exercises

5.1 Not all projections are planar geometric projections. Give an example of a projection in which the projection surface is not a plane, and another in which the projectors are not lines.

5.2 Consider an airplane whose position is specified by the roll, pitch, and yaw, and by the distance from an object. Find a model-view matrix in terms of these parameters.

5.3 Consider a satellite rotating around the earth. Its position above the earth is specified in polar coordinates. Find a model-view matrix that keeps the viewer looking at the earth. Such a matrix could be used to show the earth as it rotates.

5.4 Show how to compute u and v directions from the VPN, VRP, and VUP using only cross products.

5.5 Can we obtain an isometric of the cube by a single rotation about a suitably chosen axis? Explain your answer.

5.6 Derive the perspective-projection matrix when the COP can be at any point and the projection plane can be at any orientation.

5.7 Show that perspective projection preserves lines.

5.8 Any attempt to take the projection of a point in the same plane as the COP will lead to a division by zero. What is the projection of a line segment that has endpoints on either side of the projection plane?

5.9 Define one or more APIs to specify oblique projections. You do not need write the functions; just decide which parameters the user must specify.

5.10 Derive an oblique-projection matrix from specification of front and back clipping planes, and top-right and bottom-left intersections of the sides of the clipping volume with the front clipping plane.

5.11 Our approach of normalizing all projections seems to imply that we could predistort all objects and support only orthographic projections. Explain any problems we would face if we took this approach to building a graphics system.

5.12 How do the OpenGL projection matrices change if the COP is not at the origin? Assume that the COP is at $(0, 0, d)$ and the projection plane is $z = 0$.

5.13 We can create an interesting class of three-dimensional objects by extending two-dimensional objects into the third dimension by extrusion. For example, a circle becomes a cylinder, a line becomes a quadrilateral, and a quadrilateral in the plane becomes a parallelepiped. Use this technique to convert the two-dimensional maze from Exercise 2.8 to a three-dimensional maze.

5.14 Extend the maze program of Exercise 5.13 to allow the user to walk through the maze. A click on the middle mouse button should move the user forward; a click on the right or left button should turn the user 90 degrees to the right or left, respectively.

5.15 If we were to use orthogonal projections to draw the coordinate axes, the x- and y-axes would lie in the plane of the paper, but the z-axis

would point out of the page. Instead, we can draw the x- and y-axes as meeting at a 90-degree angle, with the z-axis going off at -135 degrees from the x-axis. Find the matrix that projects the original orthogonal-coordinate axes to this view.

5.16 Write a program to display a rotating cube in a box with three light sources. Each light source should project the cube onto one of the three visible sides of the box.

5.17 Find the projection of a point onto the plane $ax + by + cz + d = 0$ from a light source located at infinity in the direction (d_x, d_y, d_z).

5.18 Using one of the three-dimensional interfaces discussed in Chapter 4, write a program to move the camera through a scene composed of simple objects.

5.19 In animation, often we can save work by working with two-dimensional patterns that are mapped onto flat polygons that are always parallel to the camera. Write a program that will keep a simple polygon facing the camera as the camera moves.

5.20 Stereo images are produced by creating two images with the viewer in two slightly different positions. Consider a viewer who is at the origin but whose eyes are separated by Δx units. What are the appropriate viewing specifications to create the two images?