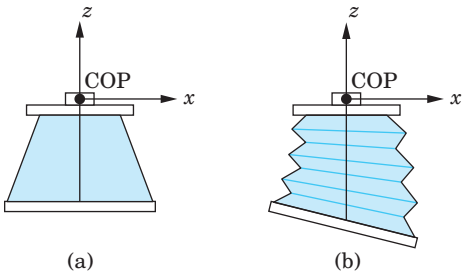

Figure 5.21 Two cameras.

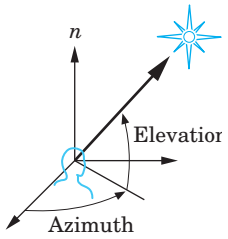

Figure 5.20 Elevation and azimuth.

even though the implementation of the two can use the same pipeline, as we shall see in Sections 5.9 and 5.10.

Just as we did with the model-view matrix, we can set the projection matrix with the `glLoadMatrix` function. Alternatively, we can use OpenGL functions for the most common viewing conditions. First, we consider the mathematics of projection. We can extend our use of homogeneous coordinates to the projection process, which allows us to characterize a particular projection with a 4×4 matrix.

5.4.1 Perspective Projections

Suppose that we are in the camera frame with the camera located at the origin, pointed in the negative z direction. Figure 5.21 shows two possibilities. In Figure 5.21(a), the back of the camera is orthogonal to the z direction and is parallel to the lens. This configuration corresponds to most physical situations, including those of the human visual system and of simple cameras. The situation in Figure 5.21(b) is more general; the back of the camera can have any orientation with respect to the front. We consider the first case in detail because it is simpler. However, the derivation of the general result follows the same steps and should be a direct exercise (Exercise 5.6).

As we saw in Chapter 1, we can place the projection plane in front of the center of projection. If we do so for the configuration of Figure 5.21(a), we get the views shown in Figure 5.22. A point in space (x, y, z) is projected along a projector into the point (x_p, y_p, z_p). All projectors pass through the origin and, because the projection plane is perpendicular to the z-axis,

$$z_p = d.$$

Because the camera is pointing in the negative z direction, the projection plane is in the negative half-space $z < 0$, and the value of d is negative.

From the top view of Figure 5.22(b), we see two similar triangles whose tangents must be same. Hence,

$$\frac{x}{z} = \frac{x_p}{d},$$

and

$$x_p = \frac{x}{z/d}.$$

Using the side view, we obtain a similar result for y_p,

$$y_p = \frac{y}{z/d}.$$

These equations are nonlinear. The division by z describes **nonuniform foreshortening**: The images of objects farther from the center of projection are reduced in size (diminution) compared to the images of objects closer to the COP.

We can look at the projection process as a transformation that takes points (x, y, z) to other points (x_p, y_p, z_p). Although this **perspective transformation** preserves lines, it is not affine. It is also irreversible: Because all points along a projector project into the same point, we cannot recover a point from its projection.[2] We can, however, modify slightly our use of homogeneous coordinates to handle projections.

When we introduced homogeneous coordinates, we represented a point in three dimensions (x, y, z) by the point $(x, y, z, 1)$ in four dimensions. Suppose that, instead, we replace (x, y, z) by the four-dimensional point

$$\mathbf{p} = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}.$$

2. In Sections 5.8 and 5.9 we shall see the advantages of OpenGL's use of an invertible variant of the projection transformation.

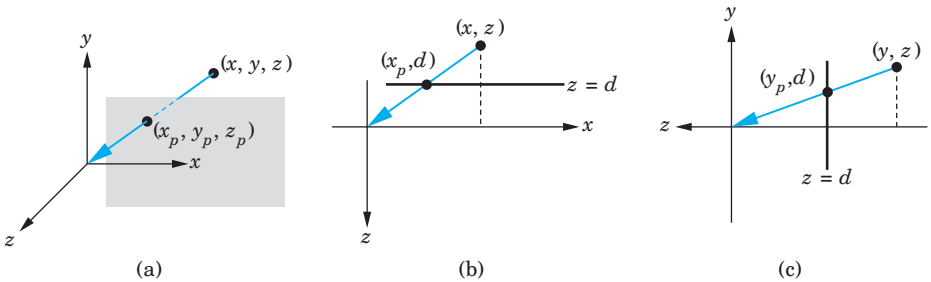

Figure 5.22 Three views of perspective projection. (a) Three-dimensional view. (b) Top view. (c) Side view.

As long as $w \neq 0$, we can recover the three-dimensional point from its four-dimensional representation by dividing the first three components by w. In this new homogeneous-coordinate form, points in three dimensions become lines through the origin in four dimensions. Transformations are again represented by 4×4 matrices, but now the final row of the matrix can be altered—and thus w can be changed by such a transformation.

Obviously, we would prefer to keep $w = 1$ to avoid the divisions otherwise necessary to recover the three-dimensional point. However, by allowing w to change, we can represent a broader class of transformations, including perspective projections. Consider the matrix

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

The matrix $\mathbf{M}$ transforms the point

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

to the point

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}.$$

At first glance, $\mathbf{q}$ may not seem sensible; but, when we remember that we have to divide the first three components by the fourth to return to our original three-dimensional space, we obtain the results

$$x_p = \frac{x}{z/d},$$
$$y_p = \frac{y}{z/d},$$
$$z_p = \frac{z}{z/d} = d,$$

which are the equations for a simple perspective projection. In homogeneous coordinates, dividing $\mathbf{q}$ by its w component replaces $\mathbf{q}$ by the equivalent point

$$\mathbf{q}' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}.$$

We have shown that we can do at least a simple perspective projection, by defining a 4×4 projection matrix that we apply after the model-view matrix. However, we must perform a **perspective division** at the end. This division can be made part of the pipeline, as shown in Figure 5.23.

5.4.2 Orthogonal Projections

Orthogonal or **orthographic** projections are a special case of parallel projections, in which the projectors are perpendicular to the view plane. In terms of a camera, orthogonal projections correspond to a camera with a back plane parallel to the lens, which has an infinite focal length. However, rather than using limiting relations as the COP moves to infinity, we can derive the projection equations directly. Figure 5.24 shows an orthogonal projection with the projection plane $z = 0$. As points are projected into this plane, they retain their x and y values, and the equations of projection are

$$x_p = x,$$
$$y_p = y,$$
$$z_p = 0.$$

We can write this result using our original homogeneous coordinates:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

In this case, a division is unnecessary, although, in hardware implementations, we can use the same pipeline for both perspective and orthogonal transformations.

We can expand both our simple projections to general perspective and parallel projections by preceding the projection by a sequence of transformations that converts the general case to one of the two cases that we know how to apply. First, we examine the API that the application programmer uses in OpenGL to specify a projection.

5.5 Projections in OpenGL

The projections that we developed in Section 5.4 did not take into account the properties of the camera—the focal length of its lens or the size of the film

Figure 5.23 Projection pipeline.

plane. Figure 5.25 shows the **angle of view** for a simple pinhole camera, like the one that we discussed in Chapter 1. Only those objects that fit within the angle of view of the camera appear in the image. If the back of the camera is rectangular, only objects within a semi-infinite pyramid—the **view volume**—whose apex is at the COP can appear in the image. Objects not within the view volume are said to be **clipped** out of the scene. Hence, our description of simple projections has been incomplete; we did not include the effects of clipping.

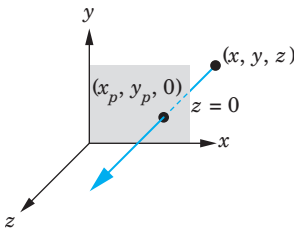

Figure 5.24 Orthogonal projection.

Most graphics APIs define clipping parameters through the specification of a projection. In a computer-graphics system, we allow a finite clipping volume by specifying front and back clipping planes, in addition to the angle of view, as shown in Figure 5.26. The resulting view volume is a **frustum**—a truncated pyramid. We have fixed only one parameter: We have specified that the COP is at the origin in the camera frame. We should be able to define each of the six sides of the frustum to have almost any orientation and position. If we did so, however, we would make it difficult to specify a view, and rarely do we need this flexibility. We examine the OpenGL API. Other APIs differ in their function calls, but incorporate similar restrictions.

5.5.1 Perspective in OpenGL

In OpenGL, we have two functions for specifying perspective views and one for specifying parallel views. Alternatively, we can form the projection matrix directly, either by loading it, or by applying rotations, translations, and scalings to an initial identity matrix. We can specify our camera view by the function

```
glFrustum(left, right, bottom, top, near, far)
```

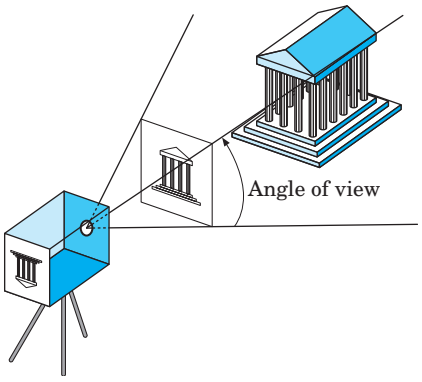

Figure 5.25 Definition of a view volume.

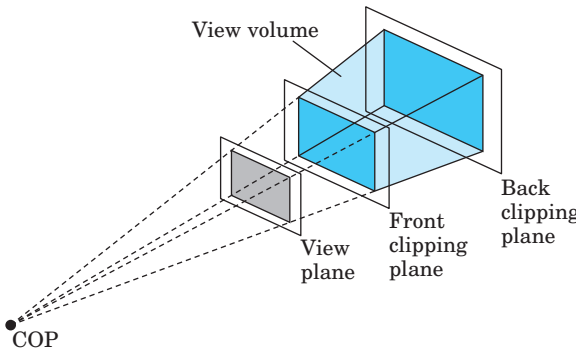

Figure 5.26 Front and back clipping planes.

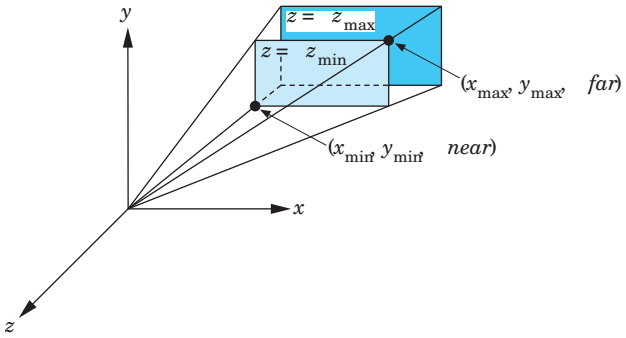

Figure 5.27 Specification of a frustum.

These parameters are shown in Figure 5.27. The near and far distances must be positive and are measured from the COP to these planes, both of which are parallel to the plane $z = 0$. Note that, because the camera is pointing in the negative z direction, the front (near) clipping plane is the plane $z = -near$, and the back (far) clipping plane is the plane $z = -far$.[3] The plane $x = left$ is to the left of the camera as viewed from the COP in the direction the camera is pointing. Similar statements hold for `right`, `bottom`, and `top`.

Because the projection matrix determined by these specifications multiplies the present matrix, we must first select the matrix mode. A typical sequence is

```
glMatrixMode(GL_PROJECTION);
```

3. Measuring distances from the camera to the objects is equivalent to switching from the right-handed world frame that we used for specifying our objects to a left-handed camera frame.

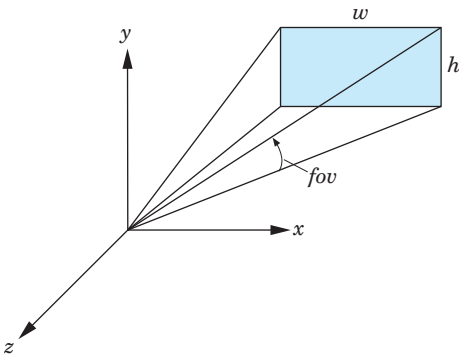

Figure 5.28 Specification using the field of view.

```
glLoadIdentity();
glFrustum(left, right, bottom, top, near, far);
```

Note that these specifications do not have to be symmetric with respect to the z-axis, and that the resulting frustum also does not have to be symmetric (a right frustum). In Section 5.9, we show how the projection matrix for this projection can be derived from the simple perspective-projection matrix of Section 5.4.

In many applications, it is natural to specify the angle or field of view. However, if the projection plane is rectangular, rather than square, then we see a different angle of view in the top and side views (Figure 5.28). The angle *fov* is the angle between the top and bottom planes of the clipping volume. The OpenGL utility function

```
gluPerspective(fovy, aspect, near, far)
```

allows us to specify the angle of view in the up (y) direction, as well as the aspect ratio—width divided by height—of the projection plane. The near and far planes are specified as in `glFrustum`. This matrix also alters the present matrix, so we must again select the matrix mode, and usually must load an identity matrix, before invoking this function.

5.5.2 Parallel Viewing in OpenGL

The only parallel-viewing function provided by OpenGL is the orthogonal (orthographic) viewing function

```
glOrtho(left, right, bottom, top, near, far)
```

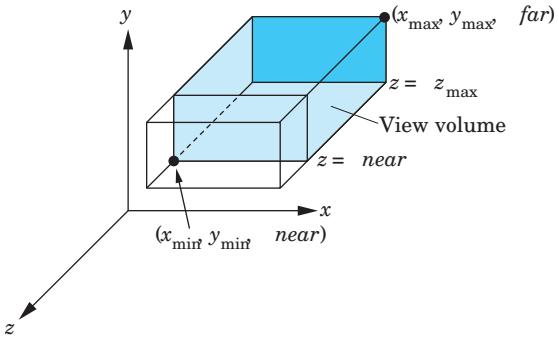

Figure 5.29 Orthographic viewing.

Its parameters are identical to those of `glFrustum`. The view volume is a right parallelepiped, as shown in Figure 5.29. The near and far clipping planes are again at $z = -near$ and $z = -far$, respectively.

In perspective viewing, we require the distances to both the near and far planes to be positive, because all projectors pass through the COP at the origin, and objects behind the COP are projected upside down, as compared with objects in front of the COP. Points in the plane $z = 0$ cannot be projected at all, and lead to division by zero. This problem does not exist in parallel viewing, and there are thus no restrictions on the sign of the near and far distances in `glOrtho`.

5.6 Hidden-Surface Removal

We can now return to our rotating-cube program of Section 4.9 and add perspective viewing and movement of the camera. First, we can use our development of viewing to understand the hidden-surface–removal process that we used in our first version of the program. When we look at a cube that has opaque sides, we see only its three front-facing sides. From the perspective of our basic viewing model, we can say that we see only these faces because they block the projectors from reaching any other surfaces.

From the perspective of computer graphics, however, all the faces of the cube have been specified and travel down the graphics pipeline; thus, the graphics system must be careful about which surfaces it displays. Conceptually, we seek algorithms that either remove those surfaces that should not be visible to the viewer, called **hidden-surface–removal algorithms**, or find which surfaces are visible, called **visible-surface algorithms**. There are many approaches to the problem, several of which we investigate in Chapter 8. OpenGL has a particular algorithm associated with it, the **z-buffer algorithm**, to which we can interface through three function calls. Hence, we introduce that algorithm here, and we return to the topic in Chapter 8.

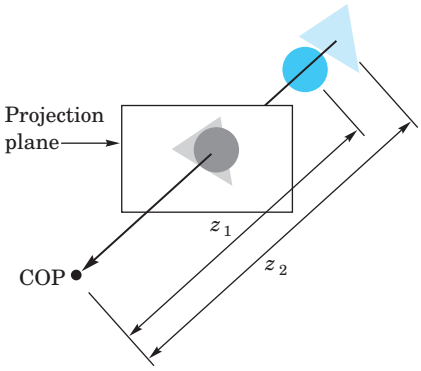

Figure 5.30 The z-buffer algorithm.

Hidden-surface–removal algorithms can be divided into two broad classes. **Object-space algorithms** attempt to order the surfaces of the objects in the scene such that drawing surfaces in a particular order provides the correct image. For example, for our cube, if we were to draw the back-facing surfaces first, we could "paint" over them with the front surfaces and would produce the correct image. This class of algorithms does not work well with pipeline architectures in which objects are passed down the pipeline in an arbitrary order. In order to decide on a proper order in which to render the objects, the graphics system must have all the objects available so it can sort them into the desired order.

Image-space algorithms work as part of the projection process and seek to determine the relationship among object points on each projector. The z-buffer algorithm is of the latter type and fits in well with the rendering pipeline in most graphics systems because we can save partial information as each object is rendered.

The basic idea of the z-buffer algorithm is shown in Figure 5.30. A projector from the COP passes through two surfaces. Because the circle is closer to the viewer than the triangle, it is the circle's color that determines the color that is placed in the color buffer at the location corresponding to where the projector pierces the projection plane. The difficulty is determining how we can make this idea work regardless of the order in which the triangle and the circle pass through the pipeline.

Let's assume that all the objects are polygons. If, as the polygons are rasterized, we can keep track of the distance from the COP or the projection plane to the closest point on each projector, then we can update this information as successive polygons are projected and filled. Ultimately, we display only the closest point on each projector. The algorithm requires a **depth** or **z buffer** to store the necessary depth information as polygons are rasterized. Because we must keep depth information for each pixel in the color buffer,

the z buffer has the same spatial resolution as the color buffers. Its depth corresponds to the amount of depth resolution that is supported, usually 16, 24, or 32 bits. This buffer can come from the standard memory in the system, or special memory can be added at the end of a hardware pipeline. In OpenGL, the z buffer is one of the buffers that constitute the frame buffer.

The depth buffer is initialized to a value that corresponds to the farthest distance from the viewer. When each polygon that is inside the clipping volume is rasterized, the depth of each pixel—how far the corresponding point on the polygon is from the viewer—is calculated. If this distance is greater than the value at that pixel's location in the depth buffer, then a polygon that has already been rasterized is closer to the viewer along the projector corresponding to the pixel. Hence, for this pixel we ignore the color of the polygon and go on to the next pixel for this polygon, where we make the same test. If, however, the depth is less than what is already in the z buffer, then along this projector the polygon being rendered is closer than any one we have seen so far. Thus, we use the color of the polygon to replace the color of the pixel in the color buffer and update the depth in the z buffer.

For the example in Figure 5.30, we see that if the triangle passes through the pipeline first, its colors and depths will be placed in the color and z buffers. When the circle passes through the pipeline, its colors and depths will replace the colors and depths of the triangle where they overlap. If the circle is rendered first, its colors and depths will be placed in the buffers. When the triangle is rendered, in the areas where there is overlap, the depths of the triangle are greater than the depth of the circle, and at the corresponding pixels no changes will be made to the frame or z buffers.

Major advantages of this algorithm are that its worst-case complexity is proportional to the number of polygons and that it can be implemented with a small number of additional calculations over what we have to do anyway to project and display polygons. We shall return to this issue in Chapter 7.

From the application programmer's perspective, she must initialize the depth buffer and enable hidden-surface removal by using

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glEnable(GL_DEPTH_TEST);
```

Here we use the GLUT library for the initialization and specify a depth buffer in addition to our usual RGB color and double buffering. The programmer can clear the buffer as necessary for a new rendering by using

```
glClear(GL_DEPTH_BUFFER_BIT);
```

5.6.1 Culling

For a convex object, such as the cube, we could simply remove all the faces pointing away from the viewer, and we could render only the ones facing the

viewer. We consider this special case further in Chapter 7. We can turn on culling in OpenGL by simply enabling it

```
glEnable(GL_CULL);
```

However, culling works only if we have a convex object. Often we can use culling in addition to the z-buffer algorithm (which works with any collection of polygons). For example, suppose that we have a scene composed of a collection of n cubes. If we use only the z-buffer algorithm, we pass $6n$ polygons through the pipeline. If we enable culling, half the polygons can be eliminated early in the pipeline, and thus only $3n$ polygons pass through all stages of the pipeline.

5.7 Interactive Mesh Displays

We can now combine our understanding of projections and modeling three-dimensional concepts to build an interactive application. We will use a simple mesh model that has many of the features of complex CAD models.

5.7.1 Meshes

We now have the tools to walk through scene interactively by have the camera parameters change in response to user input. Before introducing the interface, let's consider another example of data display: mesh plots. A **mesh** is a set of polygons that share vertices and edges. A general mesh, as shown in Figure 5.31, may contain polygons with any number of vertices and require a moderately sophisticated data structure to store and display efficiently. Rectangular and triangular meshes, such as we introduced in Chapter 2 for modeling a sphere, are much simpler with which to work and are useful for a wide variety of applications. Here, we introduce rectangular meshes for the display of height data.

One way to represent surfaces is though a function of the form

$$z = f(x, y).$$

Thus, for each x, y we get exactly one z as in Figure x. Such surfaces are sometimes called **2 1/2 dimensional surfaces**. Although not all surfaces can be represented this way, these surfaces have many applications. For example, if we use an x, y coordinate system to give positions on the surface of the earth, then we can use a function to represent the height or altitude at each location. In many situations, such as when we discussed contour maps in Chapter 2, the function f is known only discretely and we have a set of samples or measurements of experimental data of the form

$$z_{ij} = f(x_i, y_j).$$

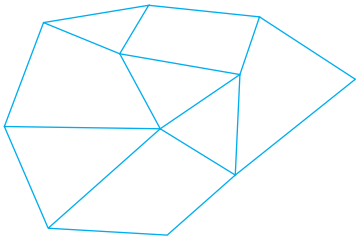

Figure 5.31 Mesh.

We assume that these data points are equally spaced such that

$$x_i = x_0 + i\Delta x, \quad i = 0, \ldots, N,$$
$$y_j = y_0 + j\Delta y, \quad j = 0, \ldots, M,$$

where Δx and Δy are the spacing between the samples in the x and y directions, respectively. If f is known analytically, then we can sample it to obtain a set of discrete data with which to work.

One simple way to generate a surface is through either a triangular or a quadrilateral mesh. We can use the four points z_{ij}, $z_{i+1,\,j}$, $z_{i+1,\,j+1}$, and $z_{i,\,j+1}$ to generate either a quadrilateral or two triangles. Thus, the data define a mesh of either NM quadrilaterals or $2NM$ triangles. The corresponding OpenGL programs are simple. The display callback need only go through the array forming quads, triangles, quads, or quad strips from adjacent rows. For quads, the heart of the display callback is simply

```
for(i=0;i<N-2;i++) for(j=1;j<M-2;j++)

    glBegin(GL_QUADS)
      glVertex3i(i,z[i][j],j);
      glVertex3i(i+1,z[i+1][j],j);
      glVertex3i(i+1,z[i+1][j+1],j+1);
      glVertex3i(i,z[i][j+1],j+1);
    glEnd();
```

Figure 5.32 shows a rectangular mesh from height data for a part of Honolulu, Hawaii. These data are available on the web site for the book.

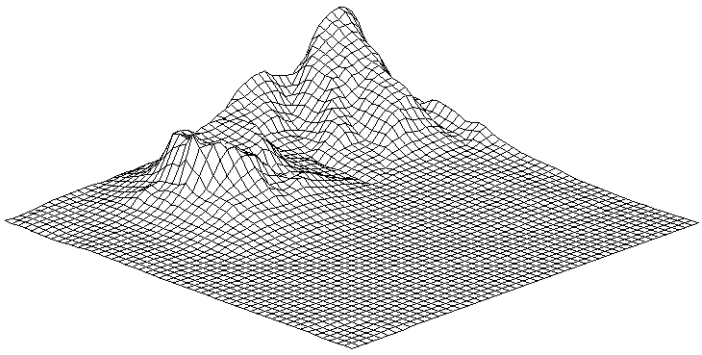

Figure 5.32 Mesh plot of Honolulu data.

5.7.2 Walking Through a Scene

The next step is to specify the camera andadd interactivity. In this version, we use perspective viewing, and we allow the viewer to move the camera by depressing the x, X, y, Y, z, and Z keys on the keyboard, but we have the camera always pointing at the center of the cube. The `gluLookAt` function provides a simple way to reposition and reorient the camera.

The changes that we have to make to our previous program (Section 4.9) are minor. We define an array `viewer[3]` to hold the camera position. Its contents are altered by the keyboard callback function `keys`

```
void keys(unsigned char key, int x, int y)
{
   if(key == 'x') viewer[0]-= 1.0;
   if(key == 'X') viewer[0]+= 1.0;
   if(key == 'y') viewer[1]-= 1.0;
   if(key == 'Y') viewer[1]+= 1.0;
   if(key == 'z') viewer[2]-= 1.0;
   if(key == 'Z') viewer[2]+= 1.0;
   glutPostRedisplay();
}
```

The display function calls `LookAt` using `viewer` for the camera position, and uses the origin for the "at" position. The cube is rotated, as before, based on the mouse input. Note the order of the function calls in `display` that alter the model-view matrix:

```
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
   gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0,
                                            0.0, 1.0, 0.0);
```

```
   glRotatef(theta[0], 1.0, 0.0, 0.0);
   glRotatef(theta[1], 0.0, 1.0, 0.0);
   glRotatef(theta[2], 0.0, 0.0, 1.0);

/* draw mesh or other objects here */
   mesh();

   glFlush();
   glutSwapBuffers();
}
```

We can invoke `glFrustum` from the reshape callback to specify the camera lens through the following code:

```
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if(w<=h) glFrustum(-2.0, 2.0, -2.0 * (GLfloat) h/ (GLfloat)
        w,2.0* (GLfloat) h / (GLfloat) w, 2.0, 20.0);
   else glFrustum(-2.0, 2.0, -2.0 * (GLfloat) w/ (GLfloat) h,
        2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
}
```

Note that we chose the values of the parameters in `glFrustum` based on the aspect ratio of the window. Other than the added specification of a keyboard callback function in `main`, the rest of the program is the same as the program in Section 4.9. The complete program is given in Appendix A. If you run this program, you should note the effects of moving the camera, the lens, and the sides of the viewing frustum. Note what happens as you move toward the mesh. You should also consider the effect of always having the viewer look at the center of the mesh as she is moving.

Note that we could have used the mouse buttons to move the viewer. We could use the mouse buttons to move the user forward, or to turn her right or left (see Exercise 5.14). However, by using the keyboard for moving the viewer, we can use the mouse to move the object as with the rotating cube in Chapter 4.

In this example, we are using direct positioning of the camera through `gluLookAt`. There are other possibilities. One is to use rotation and translation matrices to alter the model-view matrix incrementally. If we want to move the viewer through the scene without having her looking at a fixed point, this option may be more appealing. We could also keep a position variable in the program, and change it as the viewer moves. In this case, the model-view matrix would be computed from scratch, rather than changed incrementally. Which option we choose depends on the particular application, and often on other factors, such as the possibility that numerical errors might accumulate if we were to change the model-view matrix incrementally many times.

5.7.3 Polygon Offset

There are interesting aspects to and modifications we can make to the OpenGL program. First, if we use all the data, the resulting plot may contain many small polygons. The resulting density of lines in the display may be annoying and can contain moiré patterns. Hence, we might prefer to subsample the data either by using every kth point for some k or by averaging groups of data points to obtain a new set of samples with smaller N and M.

Second, the data in Figure 5.32 were drawn both as black lines and as white filled polygons. The lines are necessary to display the mesh. The polygons are necessary so that data in the front hide the data in the back. Although we can use OpenGL's hidden-surface–removal algorithm to display the polygons correctly, because these data are given in a structured order, we do not need to carry out a standard hidden-surface–removal process. If we display the data by first drawing those in back and then proceeding toward the front, the front polygons automatically hide the polygons farther back. In a system in which hidden-surface removal is done in software, not using a software z buffer and displaying the data in this manner may be faster. Data that are structured in such a manner have been called $2\frac{1}{2}$ **dimensional data sets** because, although the data exist in a three-dimensional world, this special organization leads to efficient rendering algorithms. This structure is not sufficient to represent all three-dimensional surfaces.

There is one additional trick that we used in the display of Figure 5.32. If we draw both a polygon and a line loop with the same data, such as in the code

```
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_QUADS)
  glVertex3i(i,z[i][j],j);
  glVertex3i(i+1,z[i+1][j],j);
  glVertex3i(i+1,z[i+1][j+1],j+1);
  glVertex3i(i,z[i][j+1],j+1);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_LINE_LOOP)
  glVertex3i(i,z[i][j],j);
  glVertex3i(i+1,z[i+1][j],j);
  glVertex3i(i+1,z[i+1][j+1],j+1);
  glVertex3i(i,z[i][j+1],j+1);
glEnd();
```

then the polygon and line loop are in the same plane. Even though the line loop is rendered after the polygon, numerical inaccuracies in the renderer often cause parts of a line loop to be blocked by the polygon with the same vertices. Although OpenGL lacks a mode to draw a filled polygon with its edges displayed in a different color, we can enable the **polygon offset mode** and set the offset parameters as in

```
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 0.5);
```

The first parameter determines the offset; the second is an implementation-dependent scale factor. These functions move the lines slightly toward the viewer relative to the polygon, so all the desired lines are now visible.

The basic mesh plot can be extended in many ways. In Chapter 6 we shall learn to add lights and surface properties to creat a more realistic image and in Chapter 9 we shall learn to add a texture to the surface. The texture map might be an image of the terrain from a photograph or other information that might be obtained by digitization of a map. If we combine these techniques, we can generate a display in which, by changing the position of the light source, we can make the image depend on the time of day. It is also possible to obtain smoother surfaces by using the data to define a smoother surface using one of the surface types that we shall introduce in Chapter 12.

5.8 Parallel-Projection Matrices

The OpenGL projection matrices are not quite as simple as the projection matrices that we derived in Section 5.4. In this section and the next, we derive the OpenGL projection matrices. Because projections are such a key part of three-dimensional computer graphics, understanding projections is crucial both for writing user applications and for implementing a graphics system. Furthermore, although the OpenGL projection functions that we introduced are sufficient for most viewing situations, views such as parallel oblique are not provided directly by the OpenGL API. We can obtain such views by setting up a projection matrix from scratch, or by modifying one of the standard views. We can apply either of these approaches using the matrices that we derive.

5.8.1 Projection Normalization

Our approach is based on a technique called **projection normalization**, which converts all projections into orthogonal projections by first distorting the objects such that the orthogonal projection of the distorted objects is the same as the desired projection of the original objects. This technique is shown in Figure 5.33. However, because the distortion of the objects is described by a homogeneous-coordinate matrix, we can, rather than distorting the objects, concatenate this matrix with a simple orthogonal-projection matrix to form the desired projection matrix, as shown in Figure 5.34.

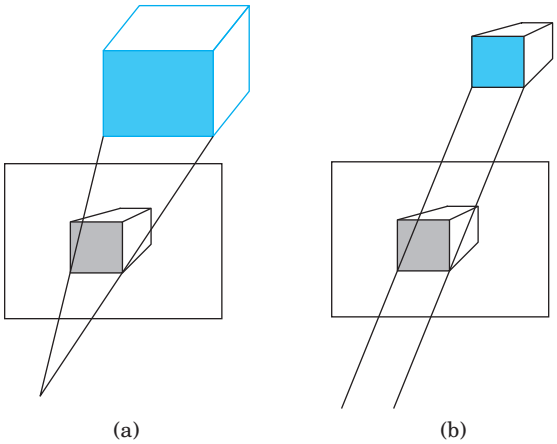

Figure 5.33 Predistortion of objects. (a) Perspective view. (b) Orthographic projection of distorted object.

5.8.2 Orthogonal-Projection Matrices

Although parallel viewing is a special case of perspective viewing, we start with orthogonal parallel viewing, and extend the normalization technique to perspective viewing. We have shown that projection converts points in three-dimensional space to points on the projection plane, and that the transformation that does this operation is singular. All points along a projector project into the same point on the projection plane.

Our development breaks projection into two parts. The first converts the specified viewing volume to standard volume by a nonsingular homogeneous-coordinate transformation. We apply the transformation that does this conversion to all our objects by concatenating the transformation matrix with the model-view matrix. Objects are distorted in a manner that yields the desired projection through the second step, which is an orthogonal projection on the transformed objects and volume

$$x_p = x,$$
$$y_p = y,$$
$$z_p = 0.$$

Figure 5.34 Normalization transformation.

Note that carrying out this orthographic projection requires only setting the z value to zero, or equivalently just neglecting it, because it is not needed. The real work in the projection process is in the first transformation. The reasons for separating the projection process into two parts have to do with many of the other tasks that we do as part of the viewing pipeline. In particular, we shall see in Chapter 7 that clipping must be done in three dimensions, and that the use of the nonsingular transformation matrix allows us to retain depth information along projectors that is necessary for hidden-surface removal and shading (Chapter 6). The first part of the process defines what most systems, including OpenGL, call the **projection matrix**. OpenGL also distinguishes between *screen coordinates*, which are two-dimensional and lack depth information, and *window coordinates*, which are three-dimensional and retain the depth information. In OpenGL, the projection matrix and the subsequent perspective division convert vertices to window coordinates.

For orthographic projections, the simplest clipping volume to deal with is a cube whose center is at the origin and whose sides are given by the six planes

$$x = \pm 1,$$
$$y = \pm 1,$$
$$z = \pm 1.$$

This cube is the default OpenGL view volume; equivalently, we can use the function calls

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

We call this volume the **canonical view volume**. The final two parameters in `glOrtho` are distances to the near and far planes measured from a camera at the origin pointed in the negative z direction. The near plane is at $z = 1.0$, which is behind the camera; the far plane is at $z = -1.0$, which is in front of the camera. Although the projectors are parallel and an orthographic projection is conceptually akin to having a camera with a long telephoto lens located far from the objects, the importance of the near and far distances in `glOrtho` is that they determine which objects are clipped out.

Now suppose that, instead, we set the `glOrtho` parameters by the function call

```
glOrtho(left, right, bottom, top, near, far);
```

We now have specified a right parallelepiped view volume whose right side (relative to the camera) is the plane $x = left$, whose left side is the plane $x = right$, whose top is the plane $y = top$, and whose bottom is the plane $y = bottom$. The front is the near clipping plane $z = -near$, and the back

is the far clipping plane $z = -far$. The projection matrix that OpenGL sets up is the matrix that transforms this volume to the cube centered at the origin with sides of length 2 shown in Figure 5.35. This matrix converts the vertices that specify our objects, such as through calls to `glVertex`, to vertices within this canonical view volume, by scaling and translating them. Consequently, vertices are transformed such that vertices within the specified view volume are transformed to vertices within the canonical view volume, and vertices outside the specified view volume are transformed to vertices outside the canonical view volume. Putting everything together, we see that the projection matrix is determined by the type of view and the view volume specified in `glOrtho`, and that these specifications are relative to the camera. The positioning and orientation of the camera are determined by the model-view matrix. These two matrices are concatenated together, and objects have their vertices transformed by this matrix product.

We can use our knowledge of affine transformations to find this projection matrix. There are two tasks that we need to do. First, we must move the center of the specified view volume to the center of the canonical view volume (the origin) by doing a translation. Second, we must scale the sides of the specified view volume to each have a length of 2 (Figure 5.36). Hence, the two transformations are $\mathbf{T}(-(right + left)/2, -(top + bottom)/2, +(far + near)/2)$ and $\mathbf{S}(2/(right - left), 2/(top - bottom), 2/(near - far))$, and can be concatenated together to form the projection matrix

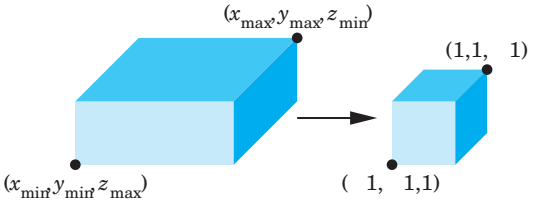

Figure 5.35 Mapping a view volume to the canonical view volume.

Figure 5.36 Affine transformations for normalization.