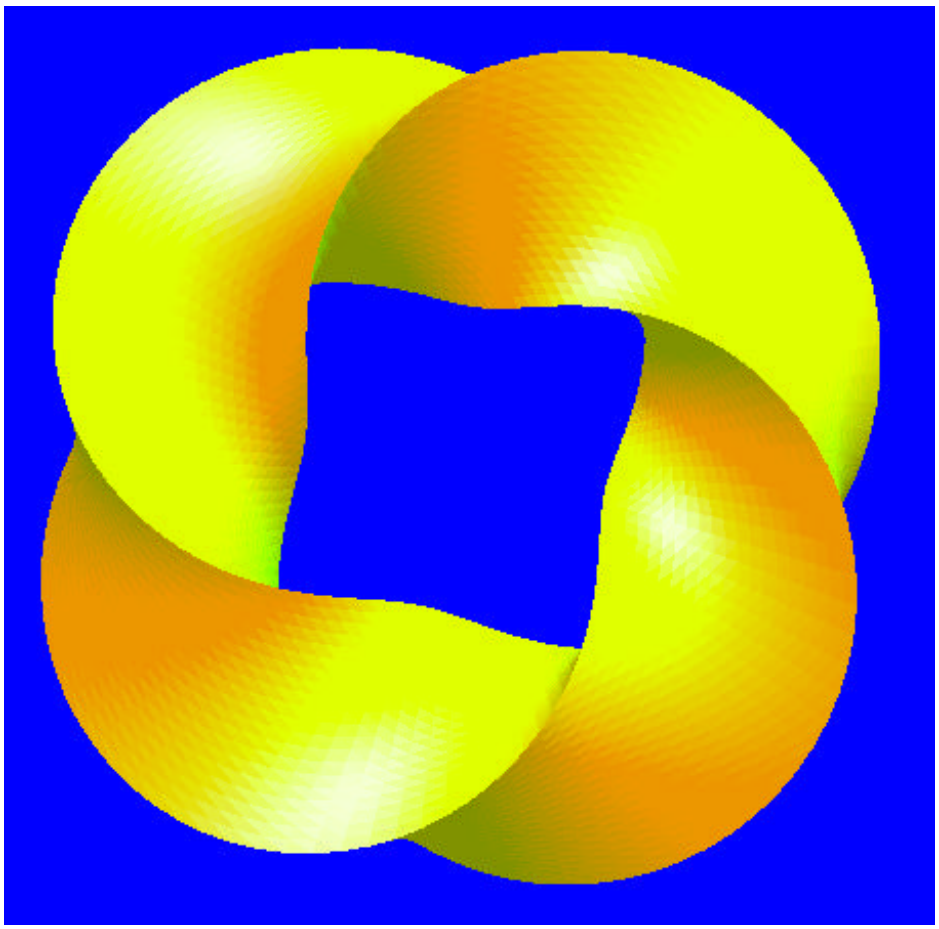
Computer Graphics: Programming, Problem Solving, and Visual Communication



Dr. Steve Cunningham
Computer Science Department
California State University Stanislaus
Turlock, CA 95382

copyright © 2002, Steve Cunningham
All rights reserved

Chapter 0: Getting Started

This book is intended for a beginning course in computer graphics for students with a sound programming background but no previous computer graphics experience. It includes a few features that are not found in most beginning courses:

- The focus is on computer graphics programming with a graphics API, and in particular discusses the OpenGL API. Many of the fundamental algorithms and techniques that are at the root of computer graphics are covered only at the level they are needed to understand questions of graphics programming. This differs from most computer graphics textbooks that place a great deal of emphasis on understanding these algorithms and techniques. We recognize the importance of these for persons who want to develop a deep knowledge of the subject and suggest that a second graphics course can provide that knowledge. We believe that the experience provided by API-based graphics programming will help you understand the importance of these algorithms and techniques as they are developed and will equip you to work with them more fluently than if you met them with no previous background.
- We focus on 3D graphics to the almost complete exclusion of 2D techniques. It has been traditional to start with 2D graphics and move up to 3D because some of the algorithms and techniques have been easier to grasp at the 2D level, but without that concern it seems easier simply to start with 3D and discuss 2D as a special case.
- Because we focus on graphics programming rather than algorithms and techniques, we have fewer instances of data structures and other computer science techniques. This means that these notes can be used for a computer graphics course that can be taken earlier in a student's computer science studies than the traditional graphics course. Our basic premise is that this course should be quite accessible to a student with a sound background in programming a sequential imperative language, particularly C.
- These notes include an emphasis on the scene graph as a fundamental tool in organizing the modeling needed to create a graphics scene. The concept of scene graph allows the student to design the transformations, geometry, and appearance of a number of complex components in a way that they can be implemented quite readily in code, even if the graphics API itself does not support the scene graph directly. This is particularly important for hierarchical modeling, but it provides a unified design approach to modeling and has some very useful applications for placing the eye point in the scene and for managing motion and animation.
- These notes include an emphasis on visual communication and interaction through computer graphics that is usually missing from textbooks, though we expect that most instructors include this somehow in their courses. We believe that a systematic discussion of this subject will help prepare students for more effective use of computer graphics in their future professional lives, whether this is in technical areas in computing or is in areas where there are significant applications of computer graphics.
- Many, if not most, of the examples in these notes are taken from sources in the sciences, and they include two chapters on scientific and mathematical applications of computer graphics. This makes the notes useable for courses that include science students as well as making graphics students aware of the breadth of areas in the sciences where graphics can be used.

This set of emphases makes these notes appropriate for courses in computer science programs that want to develop ties with other programs on campus, particularly programs that want to provide science students with a background that will support development of computational science or scientific visualization work.

What is a graphics API?

The short answer is that an API is an *Application Programming Interface*—a set of tools that allow a programmer to work in an application area. Thus a **graphics** API is a set of tools that allow a programmer to write applications that use computer graphics. These materials are intended to introduce you to the OpenGL graphics API and to give you a number of examples that will help

you understand the capabilities that OpenGL provides and will allow you to learn how to integrate graphics programming into your other work.

Overview of the book

The book is organized along fairly traditional lines, treating projection, viewing, modeling, rendering, lighting, shading, and many other aspects of the field, emphasizing 3D graphics and interactive techniques. It also includes an emphasis on using computer graphics to address real problems and to communicate results effectively to the viewer. As we move through this material, we describe some general principles in computer graphics and show how the OpenGL API provides the graphics programming tools that implement these principles. We do not spend time describing in depth the algorithms behind the techniques or the way the techniques are implemented; your instructor will provide these if he or she finds it necessary. Instead, the book focuses on describing the concepts behind the graphics and on using a graphics API (application programming interface) to carry out graphics operations and create images.

The book will give beginning computer graphics students a good introduction to the range of functionality available in a modern computer graphics API. They are based on the OpenGL API, but we have organized the general outline so that they could be adapted to fit another API as these are developed.

The key concept in the book, and in the computer graphics programming course, is the use of computer graphics to communicate information to an audience. We usually assume that the information under discussion comes from the sciences, and include a significant amount of material on models in the sciences and how they can be presented visually through computer graphics. It is tempting to use the word "visualization" somewhere in the title of this document, but we would reserve that word for material that is fully focused on the science with only a sidelight on the graphics; because we reverse that emphasis, the role of visualization is in the application of the graphics.

We have tried to match the sequence of chapters to the sequence we would expect to be used in an introductory course, and in some cases, the presentation of one module will depend on your knowing the content of an earlier chapter. However, in other cases it will not be critical that earlier chapters have been covered. It should be pretty obvious if other chapters are assumed, and we may make that assumption explicit in some modules.

What is Computer Graphics?

We view computer graphics as the art and science of creating synthetic images by programming the geometry and appearance of the contents of the images, and by displaying the results of that programming on appropriate display devices that support graphical output. The programming may be done (and in these notes, is assumed to be done) with the support of a graphics API that does most of the detailed work of rendering the scene that the programming defines.

The work of the programmer is to develop representations for the geometric entities that are to make up the images, to assemble these entities into an appropriate geometric space where they can have the proper relationships with each other as needed for the image, to define and present the look of each of the entities as part of that scene, to specify how the scene is to be viewed, and to specify how the scene as viewed is to be displayed on the graphic device. These processes are supported by the 3D graphics pipeline, as described below, which will be one of our primary tools in understanding how graphics processes work.

In addition to the work mentioned so far, there are two other important parts of the task for the programmer. Because a static image does not present as much information as a moving image, the

programmer may want to design some motion into the scene, that is, may want to define some animation for the image. And because a user may want to have the opportunity to control the nature of the image or the way the image is seen, the programmer may want to design ways for the user to interact with the scene as it is presented.

All of these topics will be covered as later chapters develop these ideas, using the OpenGL graphics API as the basis for implementing the actual graphics programming.

The 3D Graphics Pipeline

The 3D computer graphics pipeline is simply a process for converting coordinates from what is most convenient for the application programmer into what is most convenient for the display hardware. We will explore the details of the steps for the pipeline in the chapters below, but here we outline the pipeline to help you understand how it operates. The pipeline is diagrammed in Figure 0.1, and we will start to sketch the various stages in the pipeline here, with more detail given in subsequent chapters.

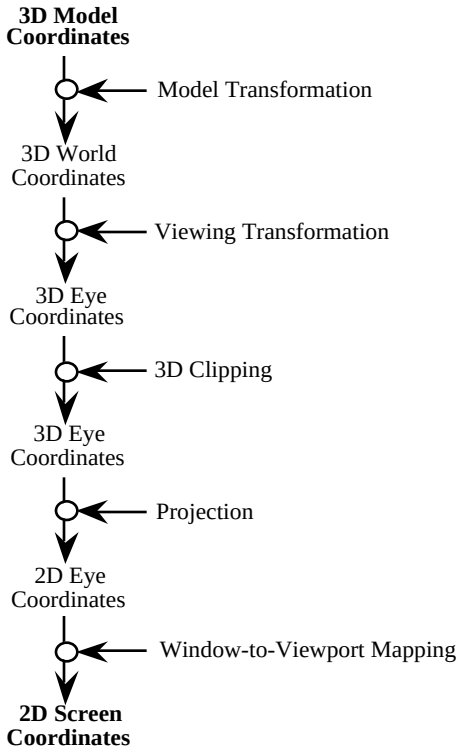


Figure 0.1: The graphics pipeline's stages and mappings

3D model coordinate systems

The application programmer starts by defining a particular object about a local origin that lies somewhere in or around the object. This is what would naturally happen if the object was created with some sort of modeling or computer-aided design system or was defined by a mathematical function. Modeling something about its local origin involves defining it in terms of *model coordinates*, a coordinate system that is used specifically to define a particular graphical object. Because the coordinate system is part of an object's design, it may be different for every part of a scene. In order to integrate each object, built with its own coordinates, into a single overall 3D

world space, the object must be placed in the world space by using an appropriate *modeling transformation*.

Modeling transformations, like all the transformations we will describe throughout the book, are functions that move objects while preserving their geometric properties. The transformations that are available to us in a graphics system are rotations, translations, and scaling. Rotations hold a line through the origin of a coordinate system fixed and rotate all the points in a scene by a fixed angle around the line, translations add a fixed value to each of the coordinates of each point in a scene, and scaling multiplies each coordinate of a point by a fixed value. These will be discussed in much more detail in the chapter on modeling below. All transformations may be represented as matrices, so sometimes in a graphics API you will see a mention of a matrix; this almost always means that a transformation is involved.

In practice, graphics programmers use a relatively small set of simple, built-in transformations and build up the model transformations through a sequence of these simple transformations. Because each transformation works on the geometry it sees, we see the effect of the associative law for functions; in a piece of code represented by metacode such as

```
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

we see that `transformThree` is applied to the original geometry, `transformTwo` to the results of that transformation, and `transformOne` to the results of the second transformation. Letting `t1, t2,` and `t3` be the three transformations, respectively, we see by the application of the associative law for function composition that

```
t1(t2(t3(geometry))) = (t1*t2*t3)(geometry)
```

This shows us that in a product of transformations, applied by multiplying on the left, the transformation nearest the geometry is applied first, and that this principle extends across multiple transformations. This will be very important in the overall understanding of the overall order in which we operate on scenes, as we describe at the end of this section.

The modeling transformation for an object in a scene can change over time to create motion in a scene. For example, in a rigid-body animation, an object can be moved through the scene just by changing its model transformation between frames. This change can be made through standard built-in facilities in most graphics APIs, including OpenGL; we will discuss how this is done later.

3D world coordinate system

The 3D coordinate system shared by all the objects in the scene is called the *world coordinate system*. By considering every component of the scene as sharing this single world, we can treat the scene uniformly as we develop the presentation of the scene through the graphics display device to the user. The scene is a master design element that contains both the geometry of the objects placed in it and the geometry of lights that illuminate it. Note that the world coordinate system often is considered to have actual dimensions as it may well model some real-world environment. This coordinate system exists without any reference to a viewer, however; the viewer is added at the next stage.

3D eye coordinate system

Once the 3D world has been created, an application programmer would like the freedom to allow an audience to view it from any location. But graphics viewing models typically require a specific orientation and/or position for the eye at this stage. For example, the system might require that the eye position be at the origin, looking in $-Z$ (or sometimes $+Z$). So the next step in the pipeline is the *viewing transformation*, in which the coordinate system for the scene is changed to satisfy this

requirement. The result is the 3D eye coordinate system. We can think of this process as grabbing the arbitrary eye location and all the 3D world objects and sliding them around to realign the spaces so that the eye ends up at the proper place and looking in the proper direction. The relative positions between the eye and the other objects have not been changed; all the parts of the scene are simply anchored in a different spot in 3D space. Because standard viewing models may also specify a standard distance from the eyepoint to some fixed "look-at" point in the scene, there may also be some scaling involved in the viewing transformation. The viewing transformation is just a transformation in the same sense as modeling transformations, although it can be specified in a variety of ways depending on the graphics API. Because the viewing transformation changes the coordinates of the entire world space in order to move the eye to the standard position and orientation, we can consider the viewing transformation to be the inverse of whatever transformation placed the eye point in the position and orientation defined for the view. We will take advantage of this observation in the modeling chapter when we consider how to place the eye in the scene's geometry.

Clipping

At this point, we are ready to clip the object against the *3D viewing volume*. The viewing volume is the 3D volume that is determined by the projection to be used (see below) and that declares what portion of the 3D universe the viewer wants to be able to see. This happens by defining how much of the scene should be visible, and includes defining the left, right, bottom, top, near, and far boundaries of that space. Any portions of the scene that are outside the defined viewing volume are clipped and discarded. All portions that are inside are retained and passed along to the projection step. In Figure 0.2, it is clear that some of the world and some of the helicopter lie outside the viewable space, but note how the front of the image of the ground in the figure is clipped—is made invisible in the scene—because it is too close to the viewer's eye. This is a bit difficult to see, but look at the cliffs at the upper left of the scene to see a clipped edge.

Figure 0.2: Clipping on the Left, Bottom, and Right

Clipping is done as the scene is projected to the 2D eye coordinates in projections, as described next. Besides ensuring that the view includes only the things that should be visible, clipping also increases the efficiency of image creation because it eliminates some parts of the geometry from the rest of the display process.

Projections

The 3D eye coordinate system still must be converted into a 2D coordinate system before it can be mapped onto a graphics display device. The next stage of the pipeline performs this operation, called a *projection*. Before discussing the actual projection, we must think about what we will actually see in the graphic device. Imagine your eye placed somewhere in the scene, looking in a particular direction. You do not see the entire scene; you only see what lies in front of your eye and within your field of view. This space is called the *viewing volume* for your scene, and it includes a bit more than the eye point, direction, and field of view; it also includes a front plane, with the concept that you cannot see anything closer than this plane, and a back plane, with the concept that you cannot see anything farther than that plane. In Figure 0.3 we see two viewing volumes for the two kinds of projections that we will discuss in a moment.

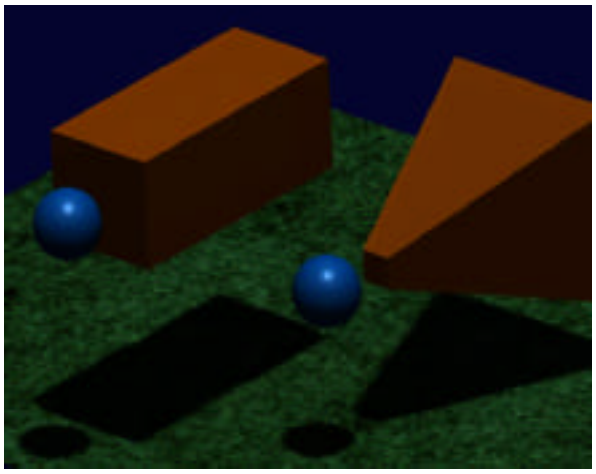


Figure 0.3: Parallel and Perspective Viewing Volumes, with Eyeballs

There are two kinds of projections commonly used in computer graphics. One maps all the points in the eye space to the viewing plane by simply ignoring the value of the z-coordinate, and as a result all points on a line parallel to the direction of the eye are mapped to the same point on the viewing plane. Such a projection is called a *parallel* projection, and it has the effect that the viewer can read accurate dimensions in the x- and y-coordinates. It is common for engineering drawings to present two parallel projections with the second including a 90° rotation of the world space so accurate z-coordinates can also be seen. The other projection acts as if the eye were a single point and each point in the scene is mapped along a line from the eye to that point, to a point on a plane in front of the eye, which is the classical technique of artists when drawing with perspective. Such a projection is called a *perspective* projection. And just as there are parallel and perspective projections, there are *parallel* (also called *orthographic*) and *perspective* viewing volumes. In a parallel projection, objects stay the same size as they get farther away. In a perspective projection, objects get smaller as they get farther away. Perspective projections tend to look more realistic, while parallel projections tend to make objects easier to line up. Each projection will display the geometry within the region of 3-space that is bounded by the right, left, top, bottom, back, and front planes described above. The region that is visible with each projection is often called its *view volume*. As we see in Figure 0.3, the viewing volume of a parallel projection is a rectangular region (here shown as a solid), while the viewing volume of a perspective projection has the shape of a pyramid that is truncated at the top (also shown as a solid). This kind of shape is a truncated pyramid and is sometimes called a *frustum*.

While the viewing volume describes the region in space that is included in the view, the actual view is what is displayed on the front clipping space of the viewing volume. This is a 2D space and is essentially the 2D eye space discussed below. Figure 0.4 presents a scene with both parallel and perspective projections; in this example, you will have to look carefully to see the differences!

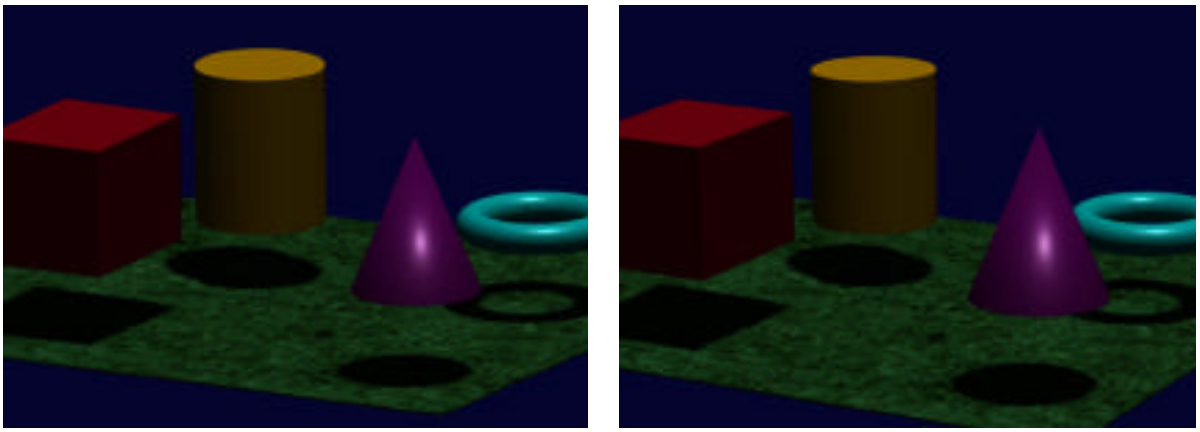


Figure 0.4: the same scene as presented by a parallel projection (left) and by a perspective projection (right)

2D eye coordinates

The space that projection maps to is a two-dimensional real-coordinate space that contains the geometry of the original scene after the projection is applied. Because a single point in 2D eye coordinates corresponds to an entire line segment in the 3D eye space, depth information is lost in the projection and it can be difficult to perceive depth, particularly if a parallel projection was used. Even in that case, however, if we display the scene with a hidden-surface technique, object occlusion will help us order the content in the scene. Hidden-surface techniques are discussed in a later chapter.

2D screen coordinates

The final step in the pipeline is to change units so that the object is in a coordinate system appropriate for the display device. Because the screen is a digital device, this requires that the real numbers in the 2D eye coordinate system be converted to integer numbers that represent screen coordinate. This is done with a proportional mapping followed by a truncation of the coordinate values. It is called the *window-to-viewport mapping*, and the new coordinate space is referred to as *screen coordinates*, or *display coordinates*. When this step is done, the entire scene is now represented by integer screen coordinates and can be drawn on the 2D display device.

Note that this entire pipeline process converts vertices, or *geometry*, from one form to another by means of several different transformations. These transformations ensure that the vertex geometry of the scene is consistent among the different representations as the scene is developed, but computer graphics also assumes that the *topology* of the scene stays the same. For instance, if two points are connected by a line in 3D model space, then those converted points are assumed to likewise be connected by a line in 2D screen space. Thus the geometric relationships (points, lines, polygons, ...) that were specified in the original model space are all maintained until we get to screen space, and are only actually implemented there.

Overall viewing process

Let's look at the overall operations on the geometry you define for a scene as the graphics system works on that scene and eventually displays it to your user. Referring again to Figure 0.1 and omitting the clipping and window-to-viewport process, we see that we start with geometry, apply the modeling transformation(s), apply the viewing transformation, and finally apply the projection to the screen. This can be expressed in terms of function composition as the sequence

```
projection(viewing(transformation(geometry)))
```

or, as we noted above with the associative law for functions and writing function composition as multiplication,

```
(projection * viewing * transformation) (geometry).
```

In the same way we saw that the operations nearest the geometry were performed before operations further from the geometry, then, we will want to define the projection first, the viewing next, and the transformations last before we define the geometry they are to operate on. We will see this sequence as a key factor in the way we structure a scene through the scene graph in the modeling chapter later in these notes.

Different implementation, same result

Warning! This discussion has shown the *concept* of how a vertex travels through the graphics pipeline. There are several ways of implementing this travel, any of which will produce a correct display. Do not be disturbed if you find out a graphics system does not manage the overall graphics pipeline process exactly as shown here. The basic principles and stages of the operation are still the same.

For example, OpenGL combines the modeling and viewing transformations into a single transformation known as the *modelview matrix*. This will force us to take a little different approach to the modeling and viewing process that integrates these two steps. Also, graphics hardware systems typically perform a window-to-normalized-coordinates operation prior to clipping so that hardware can be optimized around a particular coordinate system. In this case, everything else stays the same except that the final step would be normalized-coordinate-to-viewport mapping.

In many cases, we simply will not be concerned about the details of how the stages are carried out. Our goal will be to represent the geometry correctly at the modeling and world coordinate stages, to specify the eye position appropriately so the transformation to eye coordinates will be correct, and to define our window and projections correctly so the transformations down to 2D and to screen space will be correct. Other details will be left to a more advanced graphics course.

Summary of viewing advantages

One of the classic questions beginners have about viewing a computer graphics image is whether to use perspective or orthographic projections. Each of these has its strengths and its weaknesses. As a quick guide to start with, here are some thoughts on the two approaches:

Orthographic projections are at their best when:

- Items in the scene need to be checked to see if they line up or are the same size
- Lines need to be checked to see if they are parallel
- We do not care that distance is handled unrealistically
- We are not trying to move through the scene

Perspective projections are at their best when:

- Realism counts
- We want to move through the scene and have a view like a human viewer would have
- We do not care that it is difficult to measure or align things

In fact, when you have some experience with each, and when you know the expectations of the audience for which you're preparing your images, you will find that the choice is quite natural and will have no problem knowing which is better for a given image.

A basic OpenGL program

Our example programs that use OpenGL have some strong similarities. Each is based on the GLUT utility toolkit that usually accompanies OpenGL systems, so all the sample codes have this fundamental similarity. (If your version of OpenGL does not include GLUT, its source code is available online; check the page at

`http://www.reality.sgi.com/opengl/glut3/glut3.h`

and you can find out where to get it. You will need to download the code, compile it, and install it in your system.) Similarly, when we get to the section on event handling, we will use the MUI (micro user interface) toolkit, although this is not yet developed or included in this first draft release.

Like most worthwhile APIs, OpenGL is complex and offers you many different ways to express a solution to a graphical problem in code. Our examples use a rather limited approach that works well for interactive programs, because we believe strongly that graphics and interaction should be learned together. When you want to focus on making highly realistic graphics, of the sort that takes a long time to create a single image, then you can readily give up the notion of interactive work.

So what is the typical structure of a program that would use OpenGL to make interactive images? We will display this structure-only example in C, as we will with all our examples in these notes. OpenGL is not really compatible with the concept of object-oriented programming because it maintains an extensive set of state information that cannot be encapsulated in graphics classes, while object-oriented design usually calls for objects to maintain their own state. Indeed, as you will see when you look at the example programs, many functions such as event callbacks cannot even deal with parameters and must work with global variables, so the usual practice is to create a global application environment through global variables and use these variables instead of parameters to pass information in and out of functions. (Typically, OpenGL programs use side effects—passing information through external variables instead of through parameters—because graphics environments are complex and parameter lists can become unmanageable.)

In the code below, you will see that the main function involves mostly setting up the system. This is done in two ways: first, setting up GLUT to create and place the system window in which your work will be displayed, and second, setting up the event-handling system by defining the callbacks to be used when events occur. After this is done, main calls the main event loop that will drive all the program operations, as described in the chapter below on event handling.

The full code example that follows this outline also discusses many of the details of these functions and of the callbacks, so we will not go into much detail here. For now, the things to note are that the reshape callback sets up the window parameters for the system, including the size, shape, and location of the window, and defines the projection to be used in the view. This is called first when the main event loop is entered as well as when any window activity happens (such as resizing or dragging). The reshape callback requests a redisplay when it finishes, which calls the display callback function, whose task is to set up the view and define the geometry for the scene. When this is finished, OpenGL is finished and goes back to your computer system to see if there has been any other graphics-related event. If there has, your program should have a callback to manage it; if there has not, then the idle event is generated and the idle callback function is called; this may change some of the geometry parameters and then a redisplay is again called.

```
   #include <GL/glut.h>    // alternately "glut.h" for Macintosh
   // other includes as needed

// typedef and global data section
   // as needed

// function template section
   void doMyInit(void);
   void display(void);
   void reshape(int,int);
   void idle(void);
   // others as defined

// initialization function
   void doMyInit(void) {
      set up basic OpenGL parameters and environment
      set up projection transformation (ortho or perspective)
   }

// reshape function
   void reshape(int w, int h) {
      set up projection transformation with new window
         dimensions w and h
      post redisplay
   }

// display function
   void display(void){
      set up viewing transformation as in later chapters
      define the geometry, transformations, appearance you need
      post redisplay
   }

// idle function
   void idle(void) {
      update anything that changes between steps of the program
      post redisplay
   }

// other graphics and application functions
   // as needed

// main function -- set up the system, turn it over to events
   void main(int argc, char** argv) {
   // initialize system through GLUT and your own initialization
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
      glutInitWindowSize(windW,windH);
      glutInitWindowPosition(topLeftX,topLeftY);
      glutCreateWindow("A Sample Program");
      doMyInit();
   // define callbacks for events as needed; this is pretty minimal
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutIdleFunc(idle);
   // go into main event loop
      glutMainLoop();
   }
```

Now that we have seen a basic structure for an OpenGL program, we will present a complete, working program and will analyze the way it represents the graphics pipeline described earlier in this chapter, while describing the details of OpenGL that it uses. The program is the simple simulation of temperatures in a uniform metal bar that is described in the later chapter on graphical problem-solving in science, and we will only analyze the program structure, not its function. It creates the image shown in Figure 0.5.

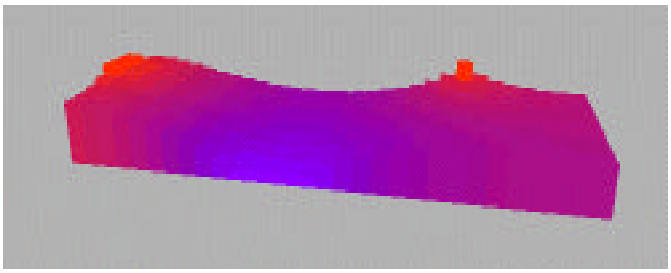


Figure 0.5: heat distribution in a bar

The code we will discuss is given below. We will segment it into components so you may see the different ways the individual pieces contribute to the overall graphics operations, and then we will discuss the pieces after the code listing.

```
// Example - heat flow in a thin rectangular body

// first section of code is declarations and initialization
// of the variables and of the system.
#include "glut.h" // <GL/glut.h> for windows
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define ROWS 10      //   body is ROWSxCOLS (unitless) squares
#define COLS 30
#define AMBIENT 25.0;//   ambient temperature, degrees Celsius
#define HOT  50.0    //   hot temperature of heat-source cell
#define COLD 0.0     //   cold temperature of cold-sink cell
#define NHOTS 4
#define NCOLDS 5

GLfloat angle = 0.0;
GLfloat temps[ROWS][COLS], back[ROWS+2][COLS+2];
GLfloat theta = 0.0, vp = 30.0;
// set locations of fixed hot and cold spots on the bar
int hotspots[NHOTS][2] =
     { {ROWS/2,0},{ROWS/2-1,0},{ROWS/2-2,0},{0,3*COLS/4} };
int coldspots[NCOLDS][2] =
     { {ROWS-1,COLS/3}, {ROWS-1,1+COLS/3}, {ROWS-1,2+COLS/3},
       {ROWS-1,3+COLS/3}, {ROWS-1,4+COLS/3} };
int myWin;

void myinit(void);
void cube(void);
void display(void);
void reshape(int, int);
void animate(void);
```

```
void myinit(void)
{
   int i,j;

   glEnable (GL_DEPTH_TEST);
   glClearColor(0.6, 0.6, 0.6, 1.0);

// set up initial temperatures in cells
   for (i=0; i<ROWS; i++) {
      for (j=0; j < COLS; j++) {
         temps[i][j] = AMBIENT;
      }
   }
   for (i=0; i<NHOTS; i++) {
      temps[hotspots[i][0]][hotspots[i][1]]=HOT; }
   for (i=0; i<NCOLDS; i++) {
      temps[coldspots[i][0]][coldspots[i][1]]=COLD; }
}
```

```
// This section creates a cube in model coordinates
// Unit cube in first octant
void cube (void)
{
   typedef GLfloat point [3];

   point v[8] = {
      {0.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0}, {0.0, 1.0, 1.0},
      {1.0, 0.0, 0.0}, {1.0, 0.0, 1.0},
      {1.0, 1.0, 0.0}, {1.0, 1.0, 1.0} };

   glBegin (GL_QUAD_STRIP);
      glVertex3fv(v[4]);
      glVertex3fv(v[5]);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
      glVertex3fv(v[6]);
      glVertex3fv(v[7]);
   glEnd();

   glBegin (GL_QUAD_STRIP);
      glVertex3fv(v[1]);
      glVertex3fv(v[3]);
      glVertex3fv(v[5]);
      glVertex3fv(v[7]);
      glVertex3fv(v[4]);
      glVertex3fv(v[6]);
      glVertex3fv(v[0]);
      glVertex3fv(v[2]);
   glEnd();
}
```

```
void display( void )
{
   #define SCALE 10.0
```

```
   int i,j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// This short section defines the viewing transformation
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   //           eye point      center of view       up
   gluLookAt(vp, vp/2., vp/4., 0.0, 0.0, 0.0,  0.0, 0.0, 1.0);

   // Draw the bars
   for (i = 0; i < ROWS; i++) {
      for (j = 0; j < COLS; j++) {
         glColor3f( temps[i][j]/HOT, 0.0, 1.0-temps[i][j]/HOT );
   // hotter redder; colder bluer
// Here is the modeling transformation for each item in the display
         glPushMatrix();
         glTranslatef((float)i-(float)ROWS/2.0,
                      (float)j-(float)COLS/2.0,0.0);
         // 0.1 cold, 4.0 hot
         glScalef(1.0, 1.0, 0.1+3.9*temps[i][j]/HOT);
         cube();
         glPopMatrix();
         }
      }
   glutSwapBuffers();
}

void reshape(int w,int h)
{
// This defines the projection transformation
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (float)w/(float)h, 1.0, 300.0);
}

void animate(void)
{
// This function is called whenever the system is idle; it makes
// changes in the data so that the next image is changed
   int i, j, m, n;
   float filter[3][3]={{ 0.    , 0.125, 0.    },
                       { 0.125 , 0.5,   0.125 },
                       { 0.    , 0.125, 0.    } };

   // increment temperatures throughout the material
   for (i=0; i<ROWS; i++) // backup temps up to recreate it
      for (j=0; j<COLS; j++)
         back[i+1][j+1] = temps[i][j]; // leave boundaries on back
   // fill boundaries with adjacent values from original temps[][]
   for (i=1; i<ROWS+2; i++) {
      back[i][0]=back[i][1];
      back[i][COLS+1]=back[i][COLS];
      }
   for (j=0; j<COLS+2; j++) {
```

```
      back[0][j] = back[1][j];
      back[ROWS+1][j]=back[ROWS][j];
      }
   for (i=0; i<ROWS; i++) // diffusion based on back values
      for (j=0; j<COLS; j++) {
         temps[i][j]=0.0;
         for (m=-1; m<=1; m++)
            for (n=-1; n<=1; n++)
               temps[i][j]+=back[i+1+m][j+1+n]*filter[m+1][n+1];
      }
   for (i=0; i<NHOTS; i++) {
      temps[hotspots[i][0]][hotspots[i][1]]=HOT; }
   for (i=0; i<NCOLDS; i++) {
      temps[coldspots[i][0]][coldspots[i][1]]=COLD; }

   // finished updating temps; now do the display
   glutPostRedisplay();
}

void main(int argc, char** argv)
{
// Initialize the GLUT system and define the window
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(50,50);
      myWin = glutCreateWindow("Temperature in bar");

      myinit();

// define the event callbacks and enter main event loop
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutIdleFunc(animate);
      glutMainLoop(); /* enter event loop */
}
```

The structure of the `main()` program using OpenGL

The `main()` program in an OpenGL-based application looks somewhat different from the programs we probably have seen before. This function has several key operations: it sets up the display mode, defines the window in which the display will be presented, and does whatever initialization is needed by the program. It then does something that may not be familiar to you: it defines a set of event callbacks, which are functions that are called by the system when an event occurs.

When you set up the display mode, you indicate to the system all the special features that your program will use at some point. In the example here,

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

you tell the system that you will be working in double-buffered mode, will use the RGB color model, and will be using depth testing. Some of these have to be enabled before they are actually used, as the depth testing is in the `myInit()` function with

```
      glEnable(GL_DEPTH_TEST).
```

Setting up the window (or windows—OpenGL will let you have multiple windows open) is handled by a set of GLUT calls that positions the window, defines the size of the window, and gives a title to the window. As the program runs, the window may be reshaped by the user, and this is handled by the `reshape()` function.

The way OpenGL handles event-driven programming is described in much more detail in a later chapter, but for now you need to realize that GLUT-based OpenGL (which is all we will describe in this book) operates entirely from events. For each event that the program is to handle, you need to define a callback function here in `main()`. When the main event loop is started, a reshape event is generated to create the window and a display event is created to draw an image in the window. If any other events have callbacks defined, then those callback functions are invoked when the events happen. The reshape callback allows you to move the window or change its size, and is called whenever you do any window manipulation. The idle callback allows the program to create a sequence of images by doing recomputations whenever the system is idle (is not creating an image or responding to another event), and then redisplaying the changed image.

Model space

The function `cube()` above defines a unit cube with sides parallel to the coordinate axes, one vertex at the origin, and one vertex at (1,1,1). This cube is created by defining an array of points that are the eight vertices of such a cube, and then using the `glBegin()...glEnd()` construction to draw the six squares that make up the cube through two quad strips. This is discussed in the chapter on modeling with OpenGL; for now, note that the cube uses its own set of coordinates that may or may not have anything to do with the space in which we will define our metallic strip to simulate heat transfer.

Modeling transformation

The modeling transformation is found in the `display()` function, and is quite simple: it defines the fundamental transformations that are to be applied to the basic geometry units as they are placed into the world. In our example, the basic geometry unit is a cube, and the cube is scaled in Z (but not in X or Y) and is then translated by X and Y (but not Z). The order of the transformations, the way each is defined, and the `pushMatrix()/popMatrix()` operations are described in the later chapter on modeling in OpenGL. For now it suffices to see that the transformations are defined in order to make a cube of the proper height to represent the temperature. If you were observant, you also noted that we also set the color for each cube based on the temperature.

3D world space

The 3D world space for this program is the space in which the graphical objects live after they have been placed by the modeling transformations. The translations give us one hint as to this space; we see that the x-coordinates of the translated cubes will lie between `-ROWS/2` and `ROWS/2`, while the y-coordinates of these cubes will lie between `-COLS/2` and `COLS/2`. Because `ROWS` and `COLS` are 30 and 10, respectively, the x-coordinates will lie between -15 and 15 and the y-coordinates will lie between -5 and 5. The low z-coordinate is 0 because that is never changed when the cubes are scaled, while the high z-coordinate is never larger then 4. Thus the entire bar lies in the region between -15 and 15 in x, -5 and 5 in y, and 0 and 4 in z. (Actually, this is not quite correct, but it is adequate for now; you are encouraged to find the small error.)

Viewing transformation

The viewing transformation is defined at the beginning of the `display()` function above. The code identifies that it is setting up the modelview matrix, sets that matrix to the identity (a

transformation that makes no changes to the world), and then specifies the view. A view is specified in OpenGL with the `gluLookAt()` call:

```
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
```

with parameters that include the coordinates of eye position (`ex, ey, ez`), the coordinates of the point at which the eye is looking (`lx, ly, lz`), and the coordinates of a vector that defines the "up" direction for the view (`ux, uy, uz`). This is discussed in more detail in the chapter below on viewing.

3D eye space

There is no specific representation of the 3D eye space in the program, because this is simply an intermediate stage in the production of the image. We can see, however, that we had set the center of view to the origin, which is the center of our image, and we had set our eye point to look at the origin from a point somewhat above and to the right of the center, so after the viewing transformation the object seems to be tilted up and to the side. This is the representation in the final 3D space that will be used to project the scene to the eye.

Projections

The projection operation is defined here in the `reshape()` function. It may be done in other places, but this is a good location and clearly separates the operation of projection from the operation of viewing.

Projections are specified fairly easily in the OpenGL system. An orthographic (or parallel) projection is defined with the function call:

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

where `left` and `right` are the x-coordinates of the left and right sides of the orthographic view volume, `bottom` and `top` are the y-coordinates of the bottom and top of the view volume, and `near` and `far` are the z-coordinates of the front and back of the view volume. A perspective projection can be defined with the function call:

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

In this function, the first parameter is the field of view in degrees, the second is the aspect ratio for the window, and the near and far parameters are as above. In this projection, it is assumed that your eye is at the origin so there is no need to specify the other four clipping planes; they are determined by the field of view and the aspect ratio.

When the window is reshaped, it is useful to take the width and height from the reshape event and define your projection to have the same aspect ratio (ratio of width to height) that the window has. That way there is no distortion introduced into the scene as it is seen through the newly-shaped window. If you use a fixed aspect ratio and change the window's shape, the original scene will be distorted to be seen through the new window, which can be confusing to the user.

2D eye space

This is the real 2D space to which the 3D world is projected, and it corresponds to the forward plane of the view volume. In order to provide uniform dimensions for mapping to the screen, the eye space is scaled so it has dimension -1 to 1 in each coordinate.

2D screen space

When the system was initialized, the window was defined to be 500x500 pixels with a top corner at (50,50). Thus the screen space is the set of pixels in that area of the screen. In fact, though, the window maintains its coordinate system independently of its location, so the point that had been

(0,0,0) in 3D eye space is now (249,249) in screen space. Note that screen space is discrete, not continuous, and its coordinates start at 0.

Another way to see the program

Another way to see how this program works is to consider it function-by-function instead of by the properties of the graphics pipeline. We will do this briefly here.

The task of the function `myinit()` is to set up the environment for the program so that the scene's fundamental environment is set up. This is a good place to compute values for arrays that define the geometry, to define specific named colors, and the like. At the end of this function you should set up the initial projection specifications.

The task of the function `display()` is to do everything needed to create the image. This can involve manipulating a significant amount of data, but the function does not allow any parameters. Here is the first place where the data for graphics problems must be managed through global variables. As we noted above, we treat the global data as a programmer-created environment, with some functions manipulating the data and the graphical functions using that data (the graphics environment) to define and present the display. In most cases, the global data is changed only through well-documented side effects, so this use of the data is reasonably clean. (Note that this argues strongly for a great deal of emphasis on documentation in your projects, which most people believe is not a bad thing.) Of course, some functions can create or receive control parameters, and it is up to you to decide whether these parameters should be managed globally or locally, but even in this case the declarations are likely to be global because of the wide number of functions that may use them. You will also find that your graphics API maintains its own environment, called its system state, and that some of your functions will also manipulate that environment, so it is important to consider the overall environment effect of your work.

The task of the function `reshape(int, int)` is to respond to user manipulation of the window in which the graphics are displayed. The two parameters are the width and height of the window in screen space (or in pixels) as it is resized by the user's manipulation, and should be used to reset the projection information for the scene. GLUT interacts with the window manager of the system and allows a window to be moved or resized very flexibly without the programmer having to manage any system-dependent operations directly. Surely this kind of system independence is one of the very good reasons to use the GLUT toolkit!

The task of the function `idle()` is to respond to the "idle" event — the event that nothing has happened. This function defines what the program is to do without any user activity, and is the way we can get animation in our programs. Without going into detail that should wait for our general discussion of events, the process is that the `idle()` function makes any desired changes in the global environment, and then requests that the program make a new display (with these changes) by invoking the function `glutPostRedisplay()` that simply requests the display function when the system can next do it by posting a "redisplay" event to the system.

The execution sequence of a simple program with no other events would then look something like is shown in Figure 0.6. Note that `main()` does not call the `display()` function directly; instead `main()` calls the event handling function `glutMainLoop()` which in turn makes the first call to `display()` and then waits for events to be posted to the system event queue. We will describe event handling in more detail in a later chapter.

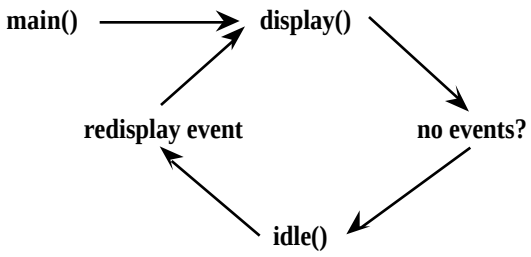


Figure 0.6: the event loop for the idle event

So we see that in the absence of any other event activity, the program will continues to apply the activity of the `idle()` function as time progresses, leading to an image that changes over time—that is, to an animated image.

Now that we have an idea of the graphics pipeline and know what a program can look like, we can move on to discuss how we specify the viewing and projection environment, how we define the fundamental geometry for our image, and how we create the image in the `display()` function with the environment that we define through the viewing and projection.

OpenGL extensions

In this chapter, and throughout these notes, we take a fairly limited view of the OpenGL graphics API. We do not work with most of the advanced features of the system and we only consider the more straightforward uses of the parts we cover. But OpenGL is capable of very sophisticated kinds of graphics, both in its original version and in versions that are available for specific kinds of graphics, and you should know of these because as you develop your graphics skills, you may find that the original "vanilla" OpenGL that we cover here will not do everything you want.

Advanced features of OpenGL include a number of special operation to store or manipulate information on a scene. These include modeling via polygon tessellation, NURBS surfaces, and defining and applying your own special-purpose transformations; the scissor test and the more general stencil buffer and stencil test; rendering in feedback mode to get details on what is being drawn; and facilities for client/server support. Remember that this is a general text, not a detailed presentation of OpenGL, and be ready to look further (see the references) for more information.

In addition to standard OpenGL, there are a number of extensions that support more specialized kinds of operations. These include the ARB imagine subset extension for image processing, the ARB multitexturing extension, vertex shader extensions, and many others. Some of these might have just the tools you need to do the very special things you want, so it would be useful for you to keep up to date on them. You can get information on extensions at the standard OpenGL Web site, `http://www.opengl.org`.

Chapter 1: Viewing and Projection

Prerequisites

An understanding of 2D and 3D geometry and familiarity with simple linear mappings.

Introduction

We emphasize 3D computer graphics consistently in these notes, because we believe that computer graphics should be encountered through 3D processes and that 2D graphics can be considered effectively as a special case of 3D graphics. But almost all of the viewing technologies that are readily available to us are 2D—certainly monitors, printers, video, and film—and eventually even the active visual retina of our eyes presents a 2D environment. So in order to present the images of the scenes we define with our modeling, we must create a 2D representation of the 3D scenes. As we saw in the graphics pipeline in the previous chapter, you begin by developing a set of models that make up the elements of your scene and set up the way the models are placed in the scene, resulting in a set of objects in a common world space. You then define the way the scene will be viewed and the way that view is presented on the screen. In this early chapter, we are concerned with the way we move from the world space to a 2D image with the tools of viewing and projection.

We set the scene for this process in the last chapter, when we defined the graphics pipeline. If we begin at the point where we have the 3D world coordinates—that is, where we have a complete scene fully defined in a 3D world—then this chapter is about creating a view of that scene in our display space of a computer monitor, a piece of film or video, or a printed page, whatever we want. To remind ourselves of the steps in this process, the pipeline is again shown in Figure 1.1.

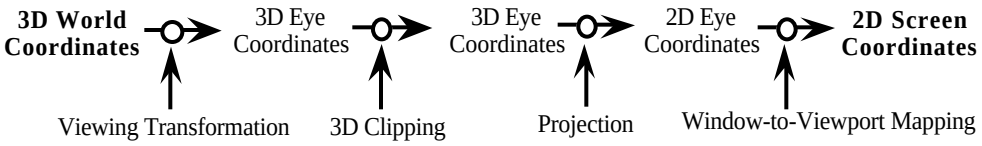


Figure 1.1: the graphics pipeline for creating an image of a scene

Let's consider an example of a world space and look at just what it means to have a view and a presentation of that space. One of the author's favorite places is Yosemite National Park, which is a wonderful example of a 3D world. Certainly there is a basic geometry in the park, made up of stone, wood, and water; this geometry can be seen from a number of points. In Figure 1.2 we see the classic piece of Yosemite geometry, the Half Dome monolith, from below in the valley and from above at Glacier Point. This gives us an excellent example of two views of the same geometry.

If you think about this area shown in these photographs, we can see the essential components of viewing. First, you notice that your view depends first on where you are standing. If you are standing on the valley floor, you see the face of the monolith in the classic view; if you are standing on the rim of Yosemite Valley at about the same height as Half Dome, you get a view that shows the profile of the rock. So your view depends on your position, which we call your *eye point.* Second, the view also depends on the point you are looking at, which we will call the *view reference point.* Both photos look generally towards the Half Dome monolith, or more specifically, towards a point in space behind the dome. This makes a difference not only in the view of the dome, but in the view of the region around the dome. In the classic Half Dome view from the valley, if you look off to the right you see the south wall of the valley; in the view from Glacier Point, if you look off to the right you see Vernal and Nevada falls on the Merced River

and, farther to the right, the high Sierra in the south of the park. The view also depends on the *breadth of field* of your view, whether you are looking at a wide part of the scene or a narrow part; again, the photograph at the left is a view of just Half Dome, while the one at the right is a panoramic view that includes the dome. While both photos are essentially square, you can visualize the left-hand photo as part of a photo that's vertical in layout while the right-hand photo looks more like it would come from a horizontal layout; this represents an *aspect ratio* for the image that can be part of its definition. Finally, although this may not be obvious at first because our minds process images in context, the view depends on your sense of the *up direction* in the scene: whether you are standing with your head upright or tilted (this might be easier to grasp if you think of the view as being defined by a camera instead of by your vision; it's clear that if you tilt a camera at a 45° angle you get a very different photo than one that's taken by a horizontal or vertical camera.) The world is the same in any case, but the determining factors for the image are where your eye is, the point you are looking toward, the breadth of your view, the aspect ratio of your view, and the way your view is tilted. All these will be accounted for when you define an image in computer graphics.

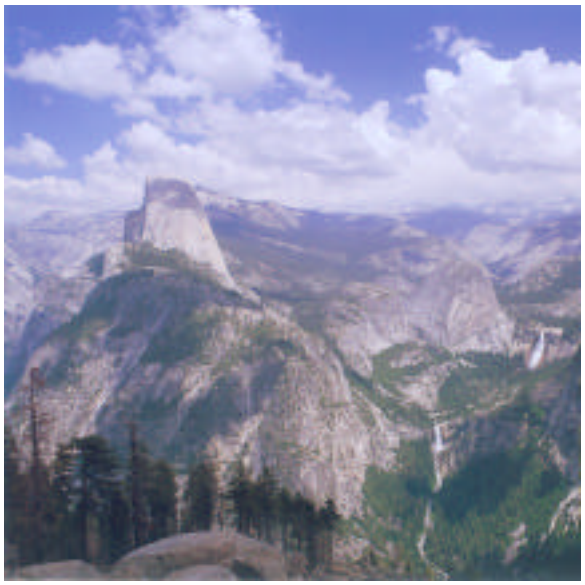



Figure 1.2: two photographs of Half Dome from different positions

Once you have determined your view, it must then be translated into an image that can be presented on your computer monitor. You may think of this in terms of recording an image on a digital camera, because the result is the same: each point of the view space (each pixel in the image) must be given a specific color. Doing that with the digital camera involves only capturing the light that comes through the lens to that point in the camera's sensing device, but doing it with computer graphics requires that we calculate exactly what will be seen at that particular point when the view is presented. We must define the way the scene is transformed into a two-dimensional space, which involves a number of steps: taking into account all the questions of what parts are in front of what other parts, what parts are out of view from the camera's lens, and how the lens gathers light from the scene to bring it into the camera. The best way to think about the lens is to compare two very different kinds of lenses: one is a wide-angle lens that gathers light in a very wide cone, and the other is a high-altitude photography lens that gathers light only in a very tight cylinder and processes light rays that are essentially parallel as they are transferred to the sensor. Finally, once the light from the continuous world comes into the camera, it is recorded on a digital sensor that only captures a discrete set of points.

This model of viewing is paralleled quite closely by a computer graphics system, and it follows the graphics pipeline that we discussed in the last chapter. You begin your work by modeling your scene in an overall world space (you may actually start in several modeling spaces, because you may model the geometry of each part of your scene in its own modeling space where it can be defined easily, then place each part within a single consistent world space to define the scene). This is very different from the viewing we discuss here but is covered in detail in the next chapter. The fundamental operation of viewing is to define an eye within your world space that represents the view you want to take of your modeling space. Defining the eye implies that you are defining a coordinate system relative to that eye position, and you must then transform your modeling space into a standard form relative to this coordinate system by defining, and applying, a viewing transformation. The fundamental operation of projection, in turn, is to define a plane within 3-dimensional space, define a mapping that projects the model into that plane, and displays that plane in a given space on the viewing surface (we will usually think of a screen, but it could be a page, a video frame, or a number of other spaces).

We will think of the 3D space we work in as the traditional X-Y-Z Cartesian coordinate space, usually with the X- and Y-axes in their familiar positions and with the Z-axis coming toward the viewer from the X-Y plane. This is a right-handed coordinate system, so-called because if you orient your right hand with your fingers pointing from the X-axis towards the Y-axis, your thumb will point towards the Z-axis. This orientation is commonly used for modeling in computer graphics because most graphics APIs define the plane onto which the image is projected for viewing as the X-Y plane, and project the model onto this plane in some fashion along the Z-axis. The mechanics of the modeling transformations, viewing transformation, and projection are managed by the graphics API, and the task of the graphics programmer is to provide the API with the correct information and call the API functionality in the correct order to make these operations work. We will describe the general concepts of viewing and projection below and will then tell you how to specify the various parts of this process to OpenGL.

Finally, it is sometimes useful to "cut away" part of an image so you can see things that would otherwise be hidden behind some objects in a scene. We include a brief discussion of clipping planes, a technique for accomplishing this action, because the system must clip away parts of the scene that are not visible in the final image.

Fundamental model of viewing

As a physical model, we can think of the viewing process in terms of looking through a rectangular frame that is held in front of your eye. You can move yourself around in the world, setting your eye into whatever position and orientation from you wish to see the scene. This defines your viewpoint and view reference point. The shape of the frame and the orientation you give it determine the aspect ratio and the up direction for the image. Once you have set your position in the world, you can hold up the frame to your eye and this will set your projection; by changing the distance of the frame from the eye you change the breadth of field for the projection. Between these two operations you define how you see the world in perspective through the hole. And finally, if you put a piece of transparent material that is ruled in very small squares behind the cardboard (instead of your eye) and you fill in each square to match the brightness you see in the square, you will create a copy of the image that you can take away from the original location. Of course, you only have a perspective projection instead of an orthogonal projection, but this model of viewing is a good place to start in understanding how viewing and projection work.

As we noted above, the goal of the viewing process is to rearrange the world so it looks as it would if the viewer's eye were in a standard position, depending on the API's basic model. When we define the eye location, we give the API the information it needs to do this rearrangement. In the next chapter on modeling, we will introduce the important concept of the *scene graph*, which

will integrate viewing and modeling. Here we give an overview of the viewing part of the scene graph.

The key point is that your view is defined by the location, direction, orientation, and field of view of the eye as we noted above. To understand this a little more fully, consider the situation shown in Figure 1.3. Here we have a world coordinate system that is oriented in the usual way, and within this world we have both a (simple) model and an eyepoint. At the eyepoint we have the coordinate system that is defined by the eyepoint-view reference point-up information that is specified for the view, so you may see the eyepoint coordinates in context. From this, you should try to visualize how the model will look once it is displayed with the view.

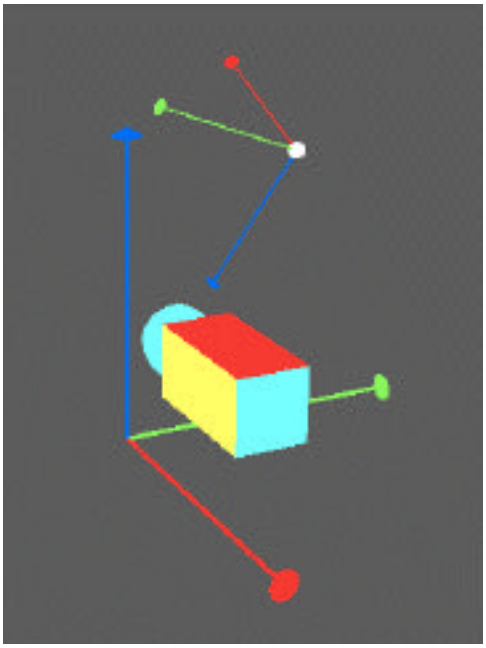


Figure 1.3: the eye coordinate system within the world coordinate system

In effect, you have defined a coordinate system within the world space relative to the eye. There are many ways to create this definition, but basically they all involve specifying three pieces of data in 3D space. Once this eye coordinate system is defined, we can apply an operation that changes the coordinates of everything in the world into equivalent representations in the eye coordinate system. This change of coordinates is a straightforward mathematical operation, performed by creating a change-of-basis matrix for the new system and then applying it to the world-space geometry. The transformation places the eye at the origin, looking along the *Z*-axis, and with the *Y*-axis pointed upwards; this view is similar to that shown in Figure 1.4. The specifications allow us to define the *viewing transformation* needed to move from the world coordinate system to the eye coordinate system. Once the eye is in standard position, and all your geometry is adjusted in the same way, the system can easily move on to project the geometry onto the viewing plane so the view can be presented to the user.

In the next chapter we will discuss modeling, and part of that process is using transformations to place objects that are defined in one position into a different position and orientations in world space. This can be applied to defining the eye point, and we can think of starting with the eye in standard position and applying transformations to place the eye where you want it. If we do that, then the viewing transformation is defined by computing the inverse of the transformation that placed the eye into the world. (If the concept of computing the inverse seems difficult, simply

think of undoing each of the pieces of the transformation; we will discuss this more in the chapter on modeling).

Once you have organized the viewing information as we have described, you must organize the information you send to the graphics system to define the way your scene is projected to the screen. The graphics system provides ways to define the projection and, once the projection is defined, the system will carry out the manipulations needed to map the scene to the display space. These operations will be discussed later in this chapter.

Definitions

There are a small number of things that you must consider when thinking of how you will view your scene. These are independent of the particular API or other graphics tools you are using, but later in the chapter we will couple our discussion of these points with a discussion of how they are handled in OpenGL. The things are:

- Your world must be seen, so you need to say how the view is defined in your model including the eye position, view direction, field of view, and orientation. This defines the viewing transformation that will be used to move from 3D world space to 3D eye space.
- In general, your world must be seen on a 2D surface such as a screen or a sheet of paper, so you must define how the 3D world is projected into a 2D space. This defines the 3D clipping and projection that will take the view from 3D eye space to 2D eye space.
- The region of the viewing device where the image is to be visible must be defined. This is the window, which should not be confused with the concept of window on your screen, though they often will both refer to the same space.
- When your world is seen in the window on the 2D surface, it must be seen at a particular place, so you must define the location where it will be seen. This defines the location of the viewport within the overall 2D viewing space and the window-to-viewport mapping that takes the 2D eye space to screen space.

We will call these three things setting up your viewing environment, defining your projection, and defining your window and viewport, respectively, and they are discussed in that order in the sections below.

Setting up the viewing environment

When you define a scene, you will want to do your work in the most natural world that would contain the scene, which we called the model space in the graphics pipeline discussion of the previous chapter. Objects defined in their individual model spaces are then placed in the world space with modeling transformations, as described in the next chapter on modeling. This world space is then transformed by the viewing transformation into a 3D space with the eye in standard position. To define the viewing transformation, you must set up a view by putting your eyepoint in the world space. This world is defined by the coordinate space you assumed when you modeled your scene as discussed earlier. Within that world, you define four critical components for your eye setup: where your eye is located, what point your eye is looking towards, how wide your field of view is, and what direction is vertical with respect to your eye. When these are defined to your graphics API, the geometry in your modeling is transformed with the viewing transformation to create the view as it would be seen with the environment that you defined.

A graphics API defines the computations that transform your geometric model as if it were defined in a standard position so it could be projected in a standard way onto the viewing plane. Each graphics API defines this standard position and has tools to create the transformation of your geometry so it can be viewed correctly. For example, OpenGL defines its viewing to take place in a right-handed coordinate system and transforms all the geometry in your scene (and we do mean *all* the geometry, including lights and directions, as we will see in later chapters) to place your eye

point at the origin, looking in the negative direction along the *Z*-axis. The eye-space orientation is illustrated in Figure 1.4.

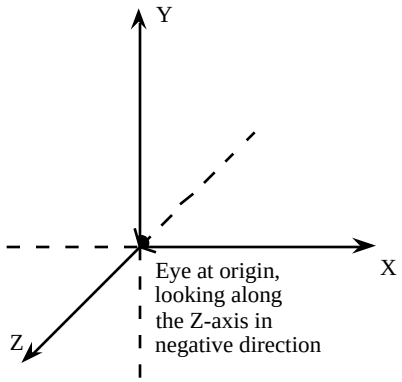


Figure 1.4: the standard OpenGL viewing model

Of course, no graphics API assumes that you can only look at your scenes with this standard view definition. Instead, you are given a way to specify your view very generally, and the API will convert the geometry of the scene so it is presented with your eyepoint in this standard position. This conversion is accomplished through the viewing transformation that is defined from your view definition as we discussed earlier.

The information needed to define your view includes your eye position (its (x,y,z) coordinates), the direction your eye is facing or the coordinates of a point toward which it is facing, and the direction your eye perceives as "up" in the world space. For example, the standard view that would be used unless you define another one has the position at the origin, or (0, 0, 0), the view direction or the "look-at" point coordinates as (0, 0, –1), and the up direction as (0, 1, 0). You will probably want to identify a different eye position for most of your viewing, because this is very restrictive and you probably will not want to define your whole viewable world as lying somewhere behind the *X*-*Y* plane. Your graphics API will give you a function that allows you to set your eye point as you desire.

The viewing transformation, then, is the transformation that takes the scene as you define it in world space and aligns the eye position with the standard model, giving you the eye space we discussed in the previous chapter. The key actions that the viewing transformation accomplishes are to rotate the world to align your personal up direction with the direction of the *Y*-axis, to rotate it again to put the look-at direction in the direction of the negative *Z*-axis (or to put the look-at point in space so it has the same *X*- and *Y*-coordinates as the eye point and a *Z*-coordinate less than the *Z*-coordinate of the eye point), to translate the world so that the eye point lies at the origin, and finally to scale the world so that the look-at point or look-at vector has the value (0, 0, –1). This is a very interesting transformation because what it *really* does is to invert the set of transformations that would move the eye point from its standard position to the position you define with your API function as above. This is very important in the modeling chapter below, and is discussed in some depth later in this chapter in terms of defining the view environment for the OpenGL API.

Defining the projection

The viewing transformation above defines the 3D eye space, but that cannot be viewed on our standard devices. In order to view the scene, it must be mapped to a 2D space that has some correspondence to your display device, such as a computer monitor, a video screen, or a sheet of paper. The technique for moving from the three-dimensional world to a two-dimensional world uses a projection operation that you define based on some straightforward fundamental principles.

When you (or a camera) view something in the real world, everything you see is the result of light that comes to the retina (or the film) through a lens that focuses the light rays onto that viewing surface. This process is a projection of the natural (3D) world onto a two-dimensional space. These projections in the natural world operate when light passes through the lens of the eye (or camera), essentially a single point, and have the property that parallel lines going off to infinity seem to converge at the horizon so things in the distance are seen as smaller than the same things when they are close to the viewer. This kind of projection, where everything is seen by being projected onto a viewing plane through or towards a single point, is called a *perspective projection*. Standard graphics references show diagrams that illustrate objects projected to the viewing plane through the center of view; the effect is that an object farther from the eye are seen as smaller in the projection than the same object closer to the eye.

On the other hand, there are sometimes situations where you want to have everything of the same size show up as the same size on the image. This is most common where you need to take careful measurements from the image, as in engineering drawings. An *orthographic projection* accomplishes this by projecting all the objects in the scene to the viewing plane by parallel lines. For orthographic projections, objects that are the same size are seen in the projection with the same size, no matter how far they are from the eye. Standard graphics texts contain diagrams showing how objects are projected by parallel lines to the viewing plane.

In Figure 1.5 we show two images of a wireframe house from the same viewpoint. The left-hand image of the figure is presented with a perspective projection, as shown by the difference in the apparent sizes of the front and back ends of the building, and by the way that the lines outlining the sides and roof of the building get closer as they recede from the viewer. The right-hand image of the figure is shown with an orthogonal projection, as shown by the equal sizes of the front and back ends of the building and the parallel lines outlining the sides and roof of the building. The differences between these two images is admittedly modest, but you should look for the differences noted above. It could be useful to use both projections on some of your scenes and compare the results to see how each of the projections works in different situations.

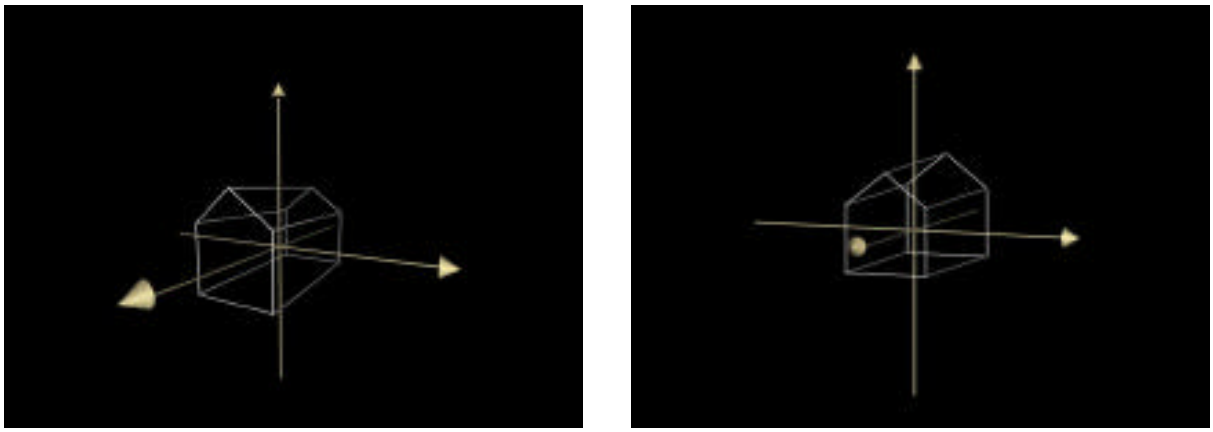


Figure 1.5: perspective image (left) and orthographic image (right) of a simple model

These two projections operate on points in 3D space in rather straightforward ways. For the orthographic projection, all points are projected onto the (X,Y)-plane in 3D eye space by simply omitting the Z-coordinate. Each point in 2D eye space is the image of a line parallel to the Z-axis, so the orthographic projection is sometimes called a *parallel projection*. For the perspective projection, any point is projected onto the plane $Z=1$ in 3D eye space at the point where the line from the point to the origin in 3D eye space meets that plane. Because of similar triangles, if the

point (x,y,z) is projected to the point (x',y'), we must have $x' = x/z$ and $(y' = y/z)$. Here each point in 2D eye space is the image of a line through that point and the origin in 3D eye space.

After a projection is applied, your scene is mapped to 2D eye space, as we discussed in the last chapter. However, the *z*-values in your scene are not lost. As each point is changed by the projection transformation, its *z*-value is retained for later computations such as depth tests or perspective-corrected textures. In some APIs such as OpenGL, the *z*-value is not merely retained but its sign is changed so that positive *z*-values will go away from the origin in a left-handed way. This convention allows the use of positive numbers in depth operations, which makes them more efficient.

View Volumes

A projection is often thought of in terms of its *view volume*, the region of space that is to be visible in the scene after the projection. With any projection, the fact that the projection creates an image on a rectangular viewing device implicitly defines a set of boundaries for the left, right, top, and bottom sides of the scene; these correspond to the left, right, top, and bottom of the viewing space. In addition, the conventions of creating images include not including objects that are too close to or too far from the eye point, and these give us the idea of front and back sides of the region of the scene that can be viewed. Overall, then, the projection defines a region in three-dimensional space that will contain all the parts of the scene that can be viewed. This region is called the *viewing volume* for the projection. The viewing volumes for the perspective and orthogonal projections are shown in Figure 1.6, with the eye point at the origin; this region is the space within the rectangular volume (left, for the orthogonal projection) or the pyramid frustum (right, for the perspective transformation). Note how these view volumes match the definitions of the regions of 3D eye space that map to points in 2D eye space, and note that each is presented in the left-handed viewing coordinate system described in Figure 1.4.

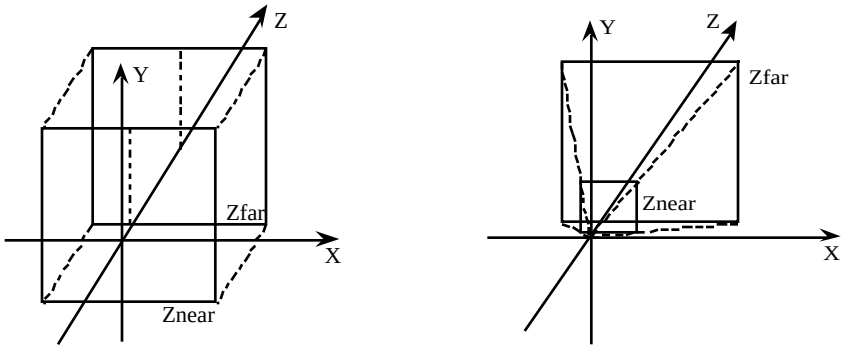


Figure 1.6: the viewing volumes for the orthogonal (left) and perspective (right) projections

While the orthographic view volume is defined only in a specified place in your model space, the orthogonal view volume may be defined wherever you need it because, being independent of the calculation that makes the world appear from a particular point of view, an orthogonal view can take in any part of space. This allows you to set up an orthogonal view of any part of your space, or to move your view volume around to view any part of your model. In fact, this freedom to place your viewing volume for the orthographic projection is not particularly important because you could always use simple translations to center the region you will see.

One of the reasons we pay attention to the view volume is that only objects that are inside the view volume for your projection will be displayed; anything else in the scene will be clipped, that is, be identified in the projection process as invisible, and thus will not be handled further by the graphics system. Any object that is partly within and partly outside the viewing volume will be clipped so

that precisely those parts inside the volume are seen, and we discuss the general concept and process of clipping later in this chapter. The sides of the viewing volume correspond to the projections of the sides of the rectangular space that is to be visible, but the front and back of the volume are less obvious—they correspond to the nearest and farthest space that is to be visible in the projection. These allow you to ensure that your image presents only the part of space that you want, and prevent things that might lie behind your eye from being projected into the visible space.

Calculating the perspective projection

The perspective projection is quite straightforward to compute, and although you do not need to carry this out yourself we will find it very useful later on to understand how it works. Given the general setup for the perspective viewing volume, let's look at a 2D version in Figure 1.7. Here we see that $Y/Y' = Z/1$, or simplifying, $Y' = Y/Z$. Thus with the conventions we have defined, the perspective projection defined on 3D eye space simply divides the original X and Y values by Z.

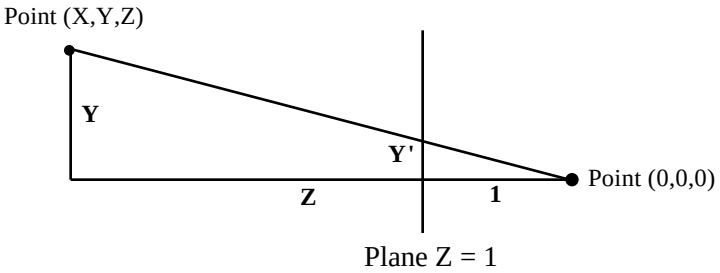


Figure 1.7: the perspective projection calculation

If we were to write this as a matrix, we would have the following:

$$\begin{matrix} 1/Z & 1 & 1 \\ 1 & 1/Z & 1 \\ 1 & 1 & 1 \end{matrix}$$

While this is a matrix, it is not a linear mapping, and that will have some significance later on when we realize that we must perform perspective corrections on some interpolations of object properties. Note here that we do not make any change in the value of Z, so that if we have the transformed values of X' and Y' and keep the original value of Z, we can reconstruct the original values as $X = X'*Z$ and $Y = Y'*Z$.

Defining the window and viewport

The scene as presented by the projection is still in 2D eye space, and the objects are all defined by real numbers. However, the display space is discrete, so the next step is a conversion of the geometry in 2D eye coordinates to discrete coordinates. This required identifying discrete screen points to replace the real-number eye geometry points, and introduces some sampling issues that must be handled carefully, but graphics APIs do this for you. The actual display space used depends on the window and the viewport you have defined for your image.

To a graphics system, a window is a rectangular region in your viewing space in which all of the drawing from your program will be done. It is usually defined in terms of the physical units of the drawing space. The window will be placed in your overall display device in terms of the device's coordinate system, which will vary between devices and systems. The window itself will have its own coordinate system, and the window space in which you define and manage your graphics

content will be called *screen space*, and is identified with integer coordinates. The smallest displayed unit in this space will be called a *pixel*, a shorthand for picture element. Note that the window for drawing is a distinct concept from the window in a desktop display window system, although the drawing window may in fact occupy a window on the desktop; we will be consistently careful to reserve the term window for the graphic display. While the window is placed in screen space, within the window itself—where we will do all our graphics work—we have a separate coordinate system that also has integer coordinates that represent pixel coordinates within the window itself. We will consistently think of the display space in terms of window points and window coordinates because they are all that matter to our image.

You will recall that we have a final transformation in the graphics pipeline from the 2D eye coordinate system to the 2D screen coordinate system. In order to understand that transformation, you need to understand the relation between points in two corresponding rectangular spaces. In this case, the rectangle that describes the scene to the eye is viewed as one space, and the rectangle on the screen where the scene is to be viewed is presented as another. The same processes apply to other situations that are particular cases of corresponding points in two rectangular spaces that we will see later, such as the relation between the position on the screen where the cursor is when a mouse button is pressed, and the point that corresponds to this in the viewing space, or points in the world space and points in a texture space.

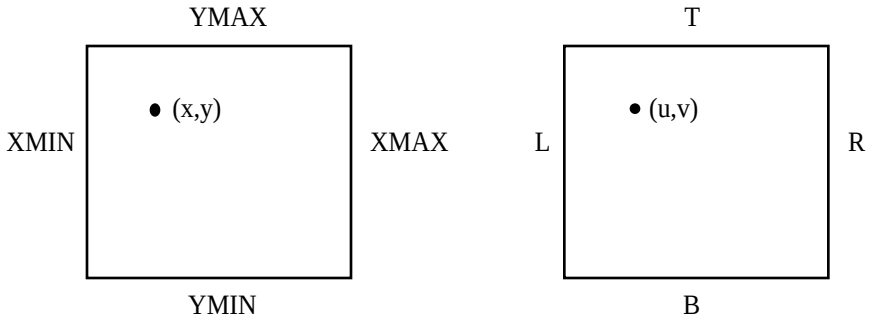


Figure 1.8: correspondences between points in two rectangles

In Figure 1.8, we show two rectangles with boundaries and points named as shown. In this example, we assume that the lower left corner of each rectangle has the smallest coordinate values in the rectangle. So the right-hand rectangle has a smallest X-value of L and a largest X-value of R, and a smallest Y-value of B and a largest Y-value of T, for example (think *left*, *right*, *top*, and *bottom* in this case).

With the names that are used in the figures, we have the proportions

$$X : XMIN :: XMAX : XMIN = u : L :: R : L$$
$$Y : YMIN :: YMAX : YMIN = v : B :: T : B$$

from which we can derive the equations:

$$(x - XMIN)/(XMAX - XMIN) = (u - L)/(R - L)$$
$$(y - YMIN)/(YMAX - YMIN) = (v - B)/(T - B)$$

and finally these two equations can be solved for the variables of either point in terms of the other, giving x and y in terms of u and v as:

$$x = XMIN + (u - L)(XMAX - XMIN)/(R - L)$$
$$y = YMIN + (v - B)(YMAX - YMIN)/(T - B)$$

or the dual equations that solve for (u,v) in terms of (x,y).

This discussion was framed in very general terms with the assumption that all our values are real numbers, because we were taking arbitrary ratios and treating them as exact values. This would

hold if we were talking about 2D eye space, but a moment's thought will show that these relations cannot hold in general for 2D screen space because integer ratios are only rarely exact. In the case of interest to us, one of these is 2D eye space and one is 2D screen space, so we must stop to ask how to modify our work for that case. For this case, to use the equations above for x and y we would regard the ratios for the right-hand rectangle in terms of real numbers and those for the left-hand rectangle as integers, we can get exact values for the ratios $(u-L)/(R-L)$ and $(v-B)/(T-B)$ and calculate real values for x and y, which we then truncate to get the desired integer values. This means that we take the view that an integer coordinate pair represents the unit square with that pair at the lower left of the square.

We noted that the window has a separate coordinate system, but we were not more specific about it. Your graphics API may use either of two conventions for window coordinates. The window may have its origin, or `(0,0)` value, at either the upper left or lower left corner. In the discussion above, we assumed that the origin was at the lower left because that is the standard mathematical convention, but graphics hardware often puts the origin at the top left because that corresponds to the lowest address of the graphics memory. If your API puts the origin at the upper left, you can make a simple change of variable as $Y' = YMAX - Y$ and using the Y' values instead of Y will put you back into the situation described in the figure.

When you create your image, you can choose to present it in a distinct sub-rectangle of the window instead of the entire window, and this part is called a *viewport*. A viewport is a rectangular region within that window to which you can restrict your image drawing. In any window or viewport, the ratio of its width to its height is called its *aspect ratio*. A window can have many viewports, even overlapping if needed to manage the effect you need, and each viewport can have its own image. Mapping an image to a viewport is done with exactly the same calculations we described above, except that the boundaries of the drawing area are the viewport's boundaries instead of the window's. The default behavior of most graphics systems is to use the entire window for the viewport. A viewport is usually defined in the same terms as the window it occupies, so if the window is specified in terms of physical units, the viewport probably will be also. However, a viewport can also be defined in terms of its size relative to the window, in which case its boundary points will be calculated from the window's.

If your graphics window is presented in a windowed desktop system, you may want to be able to manipulate your graphics window in the same way you would any other window on the desktop. You may want to move it, change its size, and click on it to bring it to the front if another window has been previously chosen as the top window. This kind of window management is provided by the graphics API in order to make the graphics window behavior on your system compatible with the behavior on all the other kinds of windows available. When you manipulate the desktop window containing the graphics window, the contents of the window need to be managed to maintain a consistent view. The graphics API tools will give you the ability to manage the aspect ratio of your viewports and to place your viewports appropriately within your window when that window is changed. If you allow the aspect ratio of a new viewport to be different than it was when defined, you will see that the image in the viewport seems distorted, because the program is trying to draw to the originally-defined viewport.

A single program can manage several different windows at once, drawing to each as needed for the task at hand. Individual windows will have different identifiers, probably returned when the window is defined, and these identifiers are used to specify which window will get the drawing commands as they are given. Window management can be a significant problem, but most graphics APIs have tools to manage this with little effort on the programmer's part, producing the kind of window you are accustomed to seeing in a current computing system—a rectangular space that carries a title bar and can be moved around on the screen and reshaped. This is the space in which all your graphical image will be seen. Of course, other graphical outputs such as video will

handle windows differently, usually treating the entire output frame as a single window without any title or border.

Some aspects of managing the view

Once you have defined the basic features for viewing your model, there are a number of other things you can consider that affect how the image is created and presented. We will talk about many of these over the next few chapters, but here we talk about hidden surfaces, clipping planes, and double buffering.

Hidden surfaces

Most of the things in our world are opaque, so we only see the things that are nearest to us as we look in any direction. This obvious observation can prove challenging for computer-generated images, however, because a graphics system simply draws what we tell it to draw in the order we tell it to draw them. In order to create images that have the simple "only show me what is nearest" property we must use appropriate tools in viewing our scene.

Most graphics systems have a technique that uses the geometry of the scene in order to decide what objects are in front of other objects, and can use this to draw only the part of the objects that are in front as the scene is developed. This technique is generally called Z-buffering because it uses information on the *z*-coordinates in the scene, as shown in Figure 1.4. In some systems it goes by other names; for example, in OpenGL this is called the *depth buffer*. This buffer holds the *z*-value of the nearest item in the scene for each pixel in the scene, where the *z*-values are computed from the eye point in eye coordinates. This *z*-value is the depth value after the viewing transformation has been applied to the original model geometry.

This depth value is not merely computed for each vertex defined in the geometry of a scene. When a polygon is processed by the graphics pipeline, an interpolation process is applied as described in the interpolation discussion in the chapter on the pipeline. This process will define a *z*-value, which is also the distance of that point from the eye in the *z*-direction, for each pixel in the polygon as it is processed. This allows a comparison of the *z*-value of the pixel to be plotted with the *z*-value that is currently held in the depth buffer. When a new point is to be plotted, the system first makes this comparison to check whether the new pixel is closer to the viewer than the current pixel in the image buffer and if it is, replaces the current point by the new point. This is a straightforward technique that can be managed in hardware by a graphics board or in software by simple data structures. There is a subtlety in this process that should be understood, however. Because it is more efficient to compare integers than floating-point numbers, the depth values in the buffer are kept as unsigned integers, scaled to fit the range between the near and far planes of the viewing volume with 0 as the front plane. If the near and far planes are far apart you may experience a phenomenon called "*Z*-fighting" in which roundoff errors when floating-point numbers are converted to integers causes the depth buffer shows inconsistent values for things that are supposed to be at equal distances from the eye. This problem is best controlled by trying to fit the near and far planes of the view as closely as possible to the actual items being displayed. This makes each integer value represent a smaller real number and so there is less likelihood of two real depths getting the same integer representation.

There are other techniques for ensuring that only the genuinely visible parts of a scene are presented to the viewer, however. If you can determine the depth (the distance from the eye) of each object in your model, then you may be able to sort a list of the objects so that you can draw them from back to front—that is, draw the farthest first and the nearest last. In doing this, you will replace anything that is hidden by other objects that are nearer, resulting in a scene that shows just the visible content. This is a classical technique called the *painter's algorithm* (because it mimics the way a painter could create an image using opaque paints) that was widely used in more limited

graphics systems, but it sometimes has real advantages over *Z*-buffering because it is faster (it doesn't require the pixel depth comparison for every pixel that is drawn) and because sometimes *Z*-buffering will give incorrect images, as we discuss when we discuss modeling transparency with blending in the color chapter. The painter's algorithm requires that you know the depth of each object in 3D eye space, however, and this can be difficult if your image includes moving parts or a moving eyepoint. Getting depths in eye space is discussed in the modeling chapter in the discussion of scene graphs.

Double buffering

A buffer is a set of memory that is used to store the result of computations, and most graphics APIs allow you to use two image buffers to store the results of your work. These are called the *color buffer* and the *back buffer*; the contents of the color buffer are what you see on your graphics screen. If you use only a single buffer, it is the color buffer, and as you generate your image, is written into the color buffer. Thus all the processes of clearing the buffer and writing new content to the buffer—new parts of your image—will all be visible to your audience.

Because it can take time to create an image, and it can be distracting for your audience to watch an image being built, it is unusual to use a single image buffer unless you are only creating one image. Most of the time you would use both buffers, and write your graphics to the back buffer instead of the color buffer. When your image is completed, you tell the system to switch the buffers so that the back buffer (with the new image) becomes the color buffer and the viewer sees the new image. When graphics is done this way, we say that we are using double buffering.

Because it can be disconcerting to actually watch the pixels changing as the image is created, particularly if you were creating an animated image by drawing one image after another, double buffering is essential to animated images. In fact, is used quite frequently for other graphics because it is more satisfactory to present a completed image instead of a developing image to a user. You must remember, however, that when an image is completed you must specify that the buffers are to be swapped, or the user will never see the new image!

Clipping planes

Clipping is the process of drawing with the portion of an image on one side of a plane drawn and the portion on the other side omitted. Recall from the discussion of geometric fundamentals that a plane is defined by a linear equation

$$Ax + By + Cz + D = 0$$

so it can be represented by the 4-tuple of real numbers (A,B,C,D). The plane divides the space into two parts: those points (x,y,z) for which $Ax + By + Cz + D$ is positive and those points for which it is negative. When you define the clipping plane for your graphics API with the functions it provides, you will probably specify it to the API by giving the four coefficients of the equation above. The operation of the clipping process is that any points for which this value is negative will not be displayed; any points for which it is positive or zero will be displayed.

Clipping defines parts of the scene that you do *not* want to display—parts that are to be left out for any reason. Any projection operation automatically includes clipping, because it must leave out objects in the space to the left, right, above, below, in front, and behind the viewing volume. In effect, each of the planes bounding the viewing volume for the projection is also a clipping plane for the image. You may also want to define other clipping planes for an image. One important reason to include clipping might be to see what is inside an object instead of just seeing the object's surface; you can define clipping planes that go through the object and display only the part of the object on one side or another of the plane. Your graphics API will probably allow you to define other clipping planes as well.

While the clipping process is handled for you by the graphics API, you should know something of the processes it uses. Because we generally think of graphics objects as built of polygons, the key point in clipping is to clip line segments (the boundaries of polygons) against the clipping plane. As we noted above, you can tell what side of a plane contains a point (x,y,z) by testing the algebraic sign of the expression $Ax + By + Cz + D$. If this expression is negative for both endpoints of a line segment, the entire line must lie on the "wrong" side of the clipping plane and so is simply not drawn at all. If the expression is positive for both endpoints, the entire line must lie on the "right" side and is drawn. If the expression is positive for one endpoint and negative for the other, then you must find the point for which the equation $Ax + By + Cz + D = 0$ is satisfied and then draw the line segment from that point to the point whose value in the expression is positive. If the line segment is defined by a linear parametric equation, the equation becomes a linear equation in one variable and so is easy to solve.

In actual practice, there are often techniques for handling clipping that are even simpler than that described above. For example, you might make only one set of comparisons to establish the relationship between a vertex of an object and a set of clipping planes such as the boundaries of a standard viewing volume. You would then be able to use these tests to drive a set of clipping operations on the line segment. We could then extend the work of clipping on line segments to clipping on the segments that are the boundaries of a polygon in order to clip parts of a polygon against one or more planes. We leave the details to the standard literature on graphics techniques.

Stereo viewing

Stereo viewing gives us an opportunity to see some of these viewing processes in action. Let us say quickly that this should not be your first goal in creating images; it requires a bit of experience with the basics of viewing before it makes sense. Here we describe binocular viewing—viewing that requires you to converge your eyes beyond the computer screen or printed image, but that gives you the full effect of 3D when the images are converged. Other techniques are described in later chapters.

Stereo viewing is a matter of developing two views of a model from two viewpoints that represent the positions of a person's eyes, and then presenting those views in a way that the eyes can see individually and resolve into a single image. This may be done in many ways, including creating two individual printed or photographed images that are assembled into a single image for a viewing system such as a stereopticon or a stereo slide viewer. (If you have a stereopticon, it can be very interesting to use modern technology to create the images for this antique viewing system!) Later in this chapter we describe how to present these as two viewports in a single window on the screen with OpenGL.

When you set up two viewpoints in this fashion, you need to identify two eye points that are offset by a suitable value in a plane perpendicular to the up direction of your view. It is probably simplest is you define your up direction to be one axis (perhaps the z-axis) and your overall view to be aligned with one of the axes perpendicular to that (perhaps the x-axis). You can then define an offset that is about the distance between the eyes of the observer (or perhaps a bit less, to help the viewer's eyes converge), and move each eyepoint from the overall viewpoint by half that offset. This makes it easier for each eye to focus on its individual image and let the brain's convergence create the merged stereo image. It is also quite important to keep the overall display small enough so that the distance between the centers of the images in the display is not larger than the distance between the viewer's eyes so that he or she can focus each eye on a separate image. The result can be quite startling if the eye offset is large so the pair exaggerates the front-to-back differences in the view, or it can be more subtle if you use modest offsets to represent realistic

views. Figure 1.9 shows the effect of such stereo viewing with a full-color shaded model. Later we will consider how to set the stereo eyepoints in a more systematic fashion.

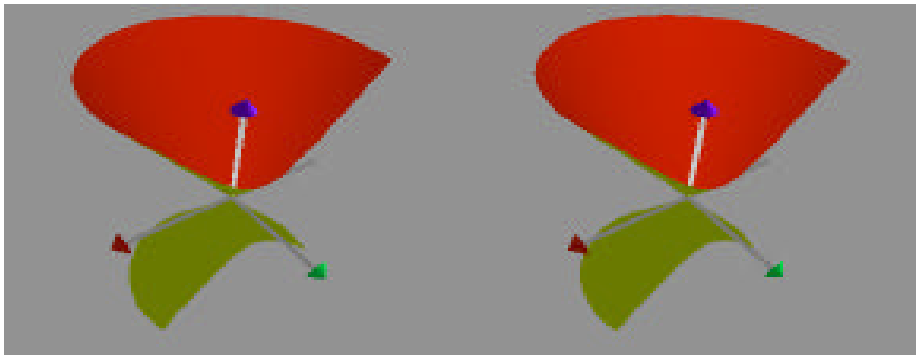


Figure 1.9: A stereo pair, including a clipping plane

Many people have physical limitations that do not allow their eyes to perform the kind of convergence that this kind of stereo viewing requires. Some people have general convergence problems which do not allow the eyes to focus together to create a merged image, and some simply cannot seem to see beyond the screen to the point where convergence would occur. In addition, if you do not get the spacing of the stereo pair right, or have the sides misaligned, or allow the two sides to refresh at different times, or ... well, it can be difficult to get this to work well for users. If some of your users can see the converged image and some cannot, that's probably as good as it's going to be.

There are other techniques for doing 3D viewing. When we discuss texture maps later, we will describe a technique that colors 3D images more red in the nearer parts and more blue in the more distant parts, as shown in Figure 1.10. This makes the images self-converge when you view them through a pair of ChromaDepth™ glasses, as we will describe there, so more people can see the spatial properties of the image, and it can be seen from anywhere in a room. There are also more specialized techniques such as creating alternating-eye views of the image on a screen with an overscreen that can be given alternating polarization and viewing them through polarized glasses that allow each eye to see only one screen at a time, or using dual-screen technologies such as head-mounted displays. The extension of the techniques above to these more specialized technologies is straightforward and is left to your instructor if such technologies are available.

Figure 1.10: a ChromaDepth™ display, courtesy of Michael J. Bailey, SDSC

Another technique involves preparing images in a way that will allow them to be picked up and displayed by special hardware such as the CrystalEyes® glasses with alternating eye vision from StereoGraphics Corp. There are a variety of ways that a left-eye and right-eye image can be combined so they can be picked up by special display hardware; these include side-by-side images, above-and-below images, or interlaced images. These combinations may require some distortion of the images that will have to be handled by the display hardware, as suggested by the distortions in the images in Figure 1.11 below, and the video stream may have to be manipulated in other ways to accommodate these approaches. In these cases, the display hardware manipulates the video output stream, separating the stream into two images that are displayed in synchronization with alternating polarized blanking of one eye, allowing the two eyes to see two distinct images and thus see the stereo pair naturally.

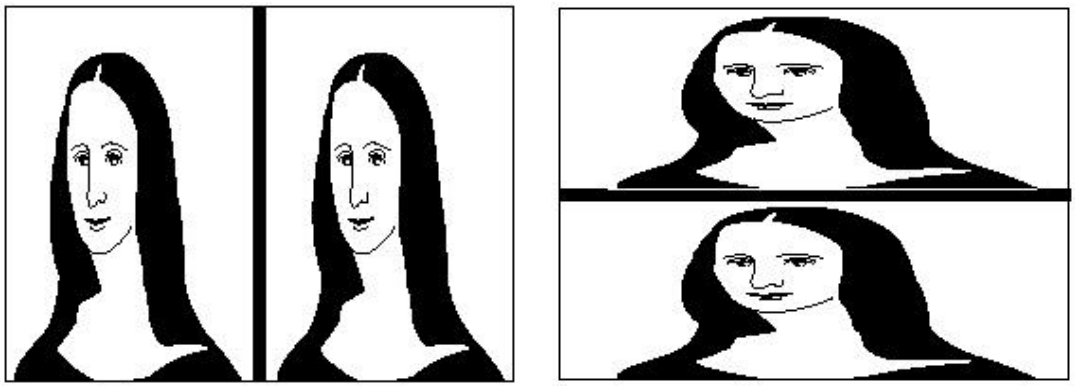


Figure 1.11: side-by-side images and above-below images, respectively.

Implementation of viewing and projection in OpenGL

The OpenGL code below captures much of the code needed in the discussion that follows in this section. It could be taken from a single function or could be assembled from several functions; in the sample structure of an OpenGL program in the previous chapter we suggested that the viewing and projection operations be separated, with the first part being at the top of the `display()` function and the latter part being at the end of the `init()` and `reshape()` functions.

```
// Define the projection for the scene
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);

// Define the viewing environment for the scene
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   //           eye point     center of view       up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

Defining a window and viewport: The window was defined in the previous chapter by a set of functions that initialize the window size and location and create the window. The details of window management are intentionally hidden from the programmer so that an API can work across many different platforms. In OpenGL, it is easiest to delegate the window setup to the GLUT toolkit where much of the system-dependent parts of OpenGL are defined; the functions to do this are:

```
glutInitWindowSize(width,height);
glutInitWindowPosition(topleftX,topleftY);
thisWindow = glutCreateWindow("Your window name here");
```

The integer value `thisWindow` that is returned by the glutCreateWindow can be used later to set the window you just created as the active window to which you will draw. This is done with the glutSetWindow function, as in

```
glutSetWindow(thisWindow);
```

which sets the window identified with `thisWindow` as the current window. If you are need to check which window is active, you can use the `glutGetWindow()` function that returns the window's value. In any case, no window is active until the main event loop is entered, as described in the previous chapter.

A viewport is defined by the `glViewport` function that specifies the lower left coordinates and the upper right coordinates for the portion of the window that will be used by the display. This function will normally be used in your initialization function for the program.

```
glViewport(VPLowerLeftX,VPLowerLeftY,VPUpperRightX,VPUpperRightY);
```

You can see the use of the viewport in the stereo viewing example below to create two separate images within one window.

Reshaping the window: The window is reshaped when it initially created or whenever is moved it to another place or made larger or smaller in any of its dimensions. These reshape operations are handled easily by OpenGL because the computer generates an event whenever any of these window reshapes happens, and there is an event callback for window reshaping. We will discuss events and event callbacks in more detail later, but the reshape callback is registered by the function `glutReshapeFunc(reshape)` which identifies a function `reshape(GLint w,GLint h)` that is to be executed whenever the window reshape event occurs and that is to do whatever is necessary to regenerate the image in the window.

The work that is done when a window is reshaped can involve defining the projection and the viewing environment and updating the definition of the viewport(s) in the window, or can delegate some of these to the display function. The reshape callback gets the dimensions of the window as it has been reshaped, and you can use these to control the way the image is presented in the reshaped window. For example, if you are using a perspective projection, the second parameter of the projection definition is the aspect ratio, and you can set this with the ratio of the width and height you get from the callback, as

```
gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);
```

This will let the projection compensate for the new window shape and retain the proportions of the original scene. On the other hand, if you really only want to present the scene in a given aspect ratio, then you can simply take the width and height and define a viewport in the window that has the aspect ratio you want. If you want a square presentation, for example, then simply take the smaller of the two values and define a square in the middle of the window as your viewport, and then do all your drawing to that viewport.

Any viewport you may have defined in your window probably needs either to be defined inside the reshape callback function so it can be redefined for resized windows or to be defined in the display function where the changed window dimensions can be taken into account when it is defined. The viewport probably should be designed directly in terms relative to the size or dimensions of the window, so the parameters of the reshape function should be used. For example, if the window is defined to have dimensions `(width, height)` as in the definition above, and if the viewport is to comprise the right-hand side of the window, then the viewport's coordinates are

```
(width/2, 0, width, height)
```

and the aspect ratio of the window is `width/(2*height)`. If the window is resized, you will probably want to make the width of the viewport no larger than the larger of half the new window width (to preserve the concept of occupying only half of the window) or the new window height

times the original aspect ratio. This kind of calculation will preserve the basic look of your images, even when the window is resized in ways that distort it far from its original shape.

Defining a viewing environment: To define what is usually called the viewing projection, you must first ensure that you are working with the `GL_MODELVIEW` matrix, then setting that matrix to be the identity, and finally define the viewing environment by specifying two points and one vector. The points are the eye point, the center of view (the point you are looking at), and the vector is the up vector—a vector that will be projected to define the vertical direction in your image. The only restrictions are that the eye point and center of view must be different, and the up vector must not be parallel to the vector from the eye point to the center of view. As we saw earlier, sample code to do this is:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
//            eye point     center of view       up
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

The `gluLookAt` function may be invoked from the `reshape` function, or it may be put inside the `display` function and variables may be used as needed to define the environment. In general, we will lean towards including the `gluLookAt` operation at the start of the `display` function, as we will discuss below. See the stereo view discussion below for an idea of what that can do.

The effect of the `gluLookAt(...)` function is to define a transformation that moves the eye point from its default position and orientation. That default position and orientation has the eye at the origin and looking in the negative z-direction, and oriented with the y-axis pointing upwards. This is the same as if we invoked the `gluLookAt` function with the parameters

```
gluLookAt(0., 0., 0., 0., 0., -1., 0., 1., 0.).
```

When we change from the default value to the general eye position and orientation, we define a set of transformations that give the eye point the position and orientation we define. The overall set of transformations supported by graphics APIs will be discussed in the modeling chapter, but those used for defining the eyepoint are:

1. a rotation about the *Z*-axis that aligns the *Y*-axis with the up vector,
2. a scaling to place the center of view at the correct distance along the negative *Z*-axis,
3. a translation that moves the center of view to the origin,
4. two rotations, about the *X*- and *Y*-axes, that position the eye point at the right point relative to the center of view, and
5. a translation that puts the center of view at the right position.

In order to get the effect you want on your overall scene, then, the viewing transformation must be the inverse of the transformation that placed the eye at the position you define, because it must act on all the geometry in your scene to return the eye to the default position and orientation. Because functions have the property that the inverse of a product is the product of the inverses in reverse order, as in

$$(f \quad g)^{-1} = g^{-1} \quad f^{-1}$$

for any *f* and *g*, the viewing transformation is built by inverting each of these five transformations in the reverse order. And because this must be done on all the geometry in the scene, it must be applied last, so it must be specified before any of the geometry is defined. Because of this we will usually see the `gluLookAt(...)` function as one of the first things to appear in the `display()` function, and its operation is the same as applying the transformations

1. translate the center of view to the origin,
2. rotate about the *X*- and *Y*-axes to put the eye point on the positive *Z*-axis,
3. translate to put the eye point at the origin,
4. scale to put the center of view at the point (0., 0., –1.), and
5. rotate around the *Z*-axis to restore the up vector to the *Y*-axis.

You may wonder why we are discussing at this point how the `gluLookAt(...)` function defines the viewing transformation that goes into the modelview matrix, but we will need to know about this later when we need to control the eye point as part of our modeling in more advanced kinds of scenes.

Defining perspective projection

A perspective projection is defined by first specifying that you want to work on the GL_PROJECTION matrix, and then setting that matrix to be the identity. You then specify the properties that will define the perspective transformation. In order, these are the field of view (an angle, in degrees, that defines the width of your viewing area), the aspect ratio (a ratio of width to height in the view; if the window is square this will probably be 1.0 but if it is not square, the aspect ratio will probably be the same as the ratio of the window width to height), the `zNear` value (the distance from the viewer to the plane that will contain the nearest points that can be displayed), and the `zFar` value (the distance from the viewer to the plane that will contain the farthest points that can be displayed). This sounds a little complicated, but once you've set it up a couple of times you'll find that it's very simple. It can be interesting to vary the field of view, though, to see the effect on the image.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
```

It is also possible to define your perspective projection by using the `glFrustum` function that defines the projection in terms of the viewing volume containing the visible items, as was shown in Figure 1.4 above. This call is

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

Perhaps the gluPerspective function is more natural, so we will not discuss the glFrustum function further and leave it to the student who wants to explore it.

Defining an orthogonal projection: an orthogonal projection is defined much like a perspective projection except that the parameters of the projection itself are different. As you can see in the illustration of a parallel projection in Figure 1.3, the visible objects lie in a box whose sides are parallel to the *X*-, *Y*-, and *Z*-axes in the viewing space. Thus to define the viewing box for an orthogonal projection, we simply define the boundaries of the box as shown in Figure 1.3 and the OpenGL system does the rest.

```
glOrtho(xLow,xHigh,yLow,yHigh,zNear,zFar);
```

The viewing space is still the same left-handed space as noted earlier, so the `zNear` and `zFar` values are the distance from the *X*-*Y* plane in the negative direction, so that negative values of `zNear` and `zFar` refer to positions behind the eye (that is, in positive *Z*-space). There is no alternate to this function in the way that the `glFrustum(...)` is an alternative to the `gluLookAt(...)` function for parallel projections.

Managing hidden surface viewing

In the Getting Started chapter, we introduced the structure of a program that uses OpenGL and saw the `glutInitDisplayMode` function, called from `main`, is a way to define properties of the display. This function also allows the use of hidden surfaces if you specify `GLUT_DEPTH` as one of its parameters.

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

You must also enable the depth test. Enabling is a standard property of OpenGL; many capabilities of the system are only available after they are enabled through the `glEnable` function, as shown below.

```
glEnable(GL_DEPTH_TEST);
```

From that point the depth buffer is in use and you need not be concerned about hidden surfaces. OpenGL uses integer values for the depth test and so is vulnerable to *Z*-fighting. The default behavior of the depth test is that a point passes the depth test (and so is recorded in the scene) if its *z*-value is less than the *z*-value stored in the depth buffer, but this can be changed by using the `glDepthFunc(value)` function, where value is a symbolic constant. We will only use the depth test in its default form, but you can see OpenGL reference sources for more details.

If you want to turn off the depth test, there is a `glDisable` function as well as the `glEnable` function. Note the use of the enable and disable functions in enabling and disabling the clipping plane in the example code for stereo viewing.

Setting double buffering

Double buffering is a standard facility, and you will note that the function above that initializes the display mode includes a parameter `GLUT_DOUBLE` to set up double buffering. This indicates that you will use two buffers, called the *back buffer* and the *front buffer*, in your drawing. The content of the front buffer is displayed, and all drawing will take place to the back buffer. So in your `display()` function, you will need to call `glutSwapBuffers()` when you have finished creating the image; that will cause the back buffer to be exchanged with the front buffer and your new image will be displayed. An added advantage of double buffering is that there are a few techniques that use drawing to the back buffer and examination of that buffer's contents without swapping the buffers, so the work done in the back buffer will not be seen.

Defining clipping planes

In addition to the clipping OpenGL performs on the standard view volume in the projection operation, OpenGL allows you to define at least six clipping planes of your own, named `GL_CLIP_PLANE0` through `GL_CLIP_PLANE5`. The clipping planes are defined by the function `glClipPlane(plane, equation)` where `plane` is one of the pre-defined clipping planes above and `equation` is a vector of four `GLfloat` values. Once you have defined a clipping plane, it is enabled or disabled by a `glEnable(GL_CLIP_PLANEn)` function or equivalent `glDisable(...)` function. Clipping is performed when any modeling primitive is called when a clip plane is enabled; it is not performed when the clip plane is disabled. They are then enabled or disabled as needed to take effect in the scene. Specifically, some example code looks like

```
GLfloat myClipPlane[] = { 1.0, 1.0, 0.0, -1.0 };
glClipPlane(GL_CLIP_PLANE0, myClipPlane);
glEnable(GL_CLIP_PLANE0);
...
glDisable(GL_CLIP_PLANE0);
```

The stereo viewing example at the end of this chapter includes the definition and use of clipping planes.

Implementing a stereo view

In this section we describe the implementation of binocular viewing as described earlier in this chapter. The technique we will use is to generate two views of a single model as if they were seen from the viewer's separate eyes, and present these in two viewports in a single window on the screen. These two images are then manipulated together by manipulating the model as a whole, while viewer resolves these into a single image by focusing each eye on a separate image.

This latter process is fairly simple. First, create a window that is twice as wide as it is high, and whose overall width is twice the distance between your eyes. Then when you display your model,

do so twice, with two different viewports that occupy the left and right half of the window. Each display is identical except that the eye points in the left and right halves represent the position of the left and right eyes, respectively. This can be done by creating a window with space for both viewports with the window initialization function

```
#define W 600
#define H 300
width = W; height = H;
glutInitWindowSize(width,height);
```

Here the initial values set the width to twice the height, allowing each of the two viewports to be initially square. We set up the view with the overall view at a distance of `ep` from the origin in the *x*-direction and looking at the origin with the *z*-axis pointing up, and set the eyes to be at a given offset distance from the overall viewpoint in the *y*-direction. We then define the left- and right-hand viewports in the `display()` function as follows

```
//    left-hand viewport
      glViewport(0,0,width/2,height);
      ...
//                  eye point      center of view       up
      gluLookAt(ep, -offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
      ... code for the actual image goes here
      ...
//    right-hand viewport
      glViewport(width/2,0,width/2,height);
      ...
//                  eye point      center of view       up
      gluLookAt(ep,  offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
      ... the same code as above for the actual image goes here
      ...
```

This particular code example responds to a `reshape(width,height)` operation because it uses the window dimensions to set the viewport sizes, but it is susceptible to distortion problems if the user does not maintain the 2:1 aspect ratio as he or she reshapes the window. It is left to the student to work out how to create square viewports within the window if the window aspect ratio is changed.

Questions

This set of questions covers your recognition of issues in viewing and projections as you see them in your personal environment. They will help you see the effects of defining views and applying projections and the other topics in this chapter

1. Find a comfortable environment and examine the ways your view of that environment depend on your eyepoint and your viewing direction. Note how objects seem to move in front of and behind other objects as you move your eyepoing, and notice how objects move into the view from one side and out of the view on the other side as you rotate your viewing direction. (It may help if you make a paper or cardboard rectangle to look through as you do this.)

2. Because of the way our eyes work, we cannot see an orthogonal view of a scene. However, if we keep our eyes oriented in a fixed direction and move around in a scene, the view directly ahead of us will approximate a piece of an orthogonal view. For your familiar environment as above, try this and see if you can sketch what you see at each point and put them together into a single image.

3. Consider a painter's algorithm approach to viewing your environment; write down the objects you see in the order of farthest to nearest to your eye. Now move to another position in the environment and imagine drawing the things you see in the same order you wrote them down from the other viewpoint. What things are out of order and so would have the farther thing

drawn on top of the nearer thing? What conclusions can you draw about the calculations you would need to do for the painter's algorithm?

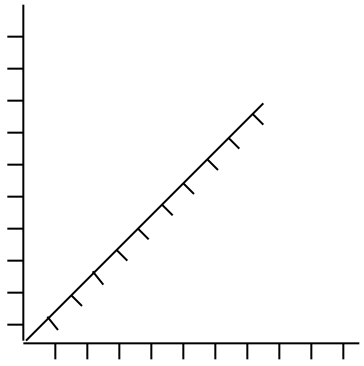
4. Imagine defining a plane through the middle of your environment so that everything on one side of the plane is not drawn. Make this plane go through some of the the objects you would see, so that one part of the object would be visible and another part invisible. What would the view of the environment look like? What would happen to the view if you switched the visibility of the two sides of the plane?

Exercises

These exercises ask you to carry out some calculations that are involved in creating a view of a scene and in doing the projection of the scene to the screen.

5. Take a standard perspective viewing definition with, say, a 45° field of view, an aspect ratio of 1.0, a distance to the front plane of the viewing frustum of 1.0, and a distance to the back plane of the viewing frustum of 20.0. For a point P=(x,y,1.) in the front plane, derive the parametric equation for the line segment within the frustum that all projects to P. Hint: the line segment goes through both the origin and P, and these two points can serve to define the segment's equation.

6. Create an X-Y-Z grid on a piece of paper using the convention that X is to the right, Y is up, and Z is into the page; an example is shown at the right.
   (a) In your familiar environment from question 1, place everything (or a number of things) from that environment into the grid with measured coordinates, to get familiar with coordinate systems to define positions. For this example, put everything into the space with non-negative coordinates (the first octant in 3D Cartesian coordinates) to make it easier to deal with the coordinates.
   (b) Define a position and direction for your eye in that same space and visualize what will be seen with that viewing definition, and then go to your space and see if your visualization was accurate. If not, then work out what was causing the inaccuracy.



7. In the numerically-modeled environment above, place your eyepoint in the (X,Z)-center of the space (the middle of the space left to right and front to back), and have your eye face the origin at floor height. Calculate the coordinates of each point in the space relative to the eye coordinate system, and try to identify a common process for each of these calculations.

Experiments

In these experiments, we will work with the very simple model of the house in Figure 1.5, though you are encouraged to replace that with a more interesting model of your own. The code for a function to create the house centered around the origin that you can call from your `display()` is given below to help you get started.

```
void drawHouse( void )
{
    point3 myHouse[10]={ { -1.0, -1.0,  2.0 }, { -1.0,  1.0,  2.0 },
                         {  0.0,  2.0,  2.0 }, {  1.0,  1.0,  2.0 },
                         {  1.0, -1.0,  2.0 }, { -1.0, -1.0, -2.0 },
                         { -1.0,  1.0, -2.0 }, {  0.0,  2.0, -2.0 },
                         {  1.0,  1.0, -2.0 }, {  1.0, -1.0, -2.0 }  };
    int i;

    glBegin(GL_LINE_STRIP);
      for ( i=0; i<5; i++)
        glVertex3fv(myHouse[i]);
      glVertex3fv(myHouse[0]);
    glEnd();
    glBegin(GL_LINE_STRIP);
      for ( i=0; i<5; i++)
        glVertex3fv(myHouse[i+5]);
      glVertex3fv(myHouse[5]);
    glEnd();
    for ( i=0; i<5; i++) {
      glBegin(GL_LINE_STRIP);
        glVertex3fv(myHouse[i]);
        glVertex3fv(myHouse[i+5]);
      glEnd();
    }
}
```

8. Create a program to draw the house with this function or to draw your own scene, and note what happens to the view as you move your eyepoint around the scene, always looking at the origin (0,0,0).

9. With the same program as above and a fixed eye point, experiment with the other parameters of the perspective view: the front and back view planes, the aspect ratio of the view, and the field of view of the projection. For each, note the effect so you can control these when you create more sophisticated images later.

Chapter 2: Principles of Modeling

Prerequisites

This chapter requires an understanding of simple 3-dimensional geometry, knowledge of how to represent points in 3-space, enough programming experience to be comfortable writing code that calls API functions to do required tasks, ability to design a program in terms of simple data structures such as stacks, and an ability to organize things in 3D space.

Introduction

Modeling is the process of defining the geometry that makes up a scene and implementing that definition with the tools of your graphics API. This chapter is critical in developing your ability to create graphical images and takes us from quite simple modeling to fairly complex modeling based on hierarchical structures, and discusses how to implement each of these different stages of modeling in OpenGL. It is fairly comprehensive for the kinds of modeling one would want to do with a basic graphics API, but there are other kinds of modeling used in advanced API work and some areas of computer graphics that involve more sophisticated kinds of constructions than we include here, so we cannot call this a genuinely comprehensive discussion. It is, however, a good enough introduction to give you the tools to start creating interesting images.

The chapter has three distinct parts because there are three distinct levels of modeling that you will use to create images. We begin with simple geometric modeling: modeling where you define the coordinates of each vertex of each component you will use at the point where that component will reside in the final scene. This is straightforward but can be very time-consuming to do for complex scenes, so we will also discuss importing models from various kinds of modeling tools that can allow you to create parts of a scene more easily.

The second section describes the next step in modeling: extending the simple objects defined in standard positions so they can have any size, any orientation, and any position. This extends the utility of your simple modeling by providing a set of primitive transformations you can apply to your simple geometry in order to create more general model components in your scene. This is a very important part of the modeling process because it allows you to use appropriate transformations that allow you to start with standard templates for a modest number of kinds of graphic objects and then generalize them and place them in your scene as needed. These transformations are also critical to the ability to define and implement motion in your scenes because it is typical to move objects, lights, and the eyepoint with transformations that are controlled by parameters that change with time. This can allow you to extend your modeling to define animations that can represent such concepts as changes over time.

In the third section of the chapter we give you an important tool to organize complex images by introducing the concept of the scene graph, a modeling tool that gives you a unified approach to defining all the objects and transformations that are to make up a scene and to specifying how they are related and presented. We then describe how you work from the scene graph to write the code that implements your model. This concept is new to the introductory graphics course but has been used in some more advanced graphics tools, and we believe you will find it to make the modeling process much more straightforward for anything beyond a very simple scene. In the second level of modeling discussed in this section, we introduce hierarchical modeling in which objects are designed by assembling other objects to make more complex structures. These structures can allow you to simulate actual physical assemblies and develop models of structures like physical machines. Here we develop the basic ideas of scene graphs introduced earlier to get a structure that allows individual components to move relative to each other in ways that would be difficult to define from first principles.

Simple Geometric Modeling

Introduction

Computer graphics deals with geometry and its representation in ways that allow it to be manipulated and displayed by a computer. Because these notes are intended for a first course in the subject, you will find that the geometry will be simple and will use familiar representations of 3-dimensional space. When you work with a graphics API, you will need to work with the kinds of object representations that API understands, so you must design your image or scene in ways that fit the API's tools. For most APIs, this means using only a few simple graphics primitives, such as points, line segments, and polygons.

The space we will use for our modeling is simple Euclidean 3-space with standard coordinates, which we will call the *X*-, *Y*-, and *Z*-coordinates. Figure 2.1 below illustrates a point, a line segment, a polygon, and a polyhedron—the basic elements of the computer graphics world that you will use for most of your graphics. In this space a *point* is simply a single location in 3-space, specified by its coordinates and often seen as a triple of real numbers such as (*px*, *py*, *pz*). A point is drawn on the screen by lighting a single pixel at the screen location that best represents the location of that point in space. To draw the point you will specify that you want to draw points and specify the point's coordinates, usually in 3-space, and the graphics API will calculate the coordinates of the point on the screen that best represents that point and will light that pixel. Note that a point is usually presented as a square, not a dot, as indicated in the figure. A *line segment* is determined by its two specified endpoints, so to draw the line you indicate that you want to draw lines and define the points that are the two endpoints. Again, these endpoints are specified in 3-space and the graphics API calculates their representations on the screen, and draws the line segment between them. A *polygon* is a region of space that lies in a plane and is bounded in the plane by a collection of line segments. It is determined by a sequence of points (called the *vertices* of the polygon) that specify a set of line segments that form its boundary, so to draw the polygon you indicate that you want to draw polygons and specify the sequence of vertex points. A *polyhedron* is a region of 3-space bounded by polygons, called the faces of the polyhedron. A polyhedron is defined by specifying a sequence of faces, each of which is a polygon. Because figures in 3-space determined by more than three vertices cannot be guaranteed to line in a plane, polyhedra are often defined to have triangular faces; a triangle always lies in a plane (because three points in 3-space determine a plane. As we will see when we discuss lighting and shading in subsequent chapters, the direction in which we go around the vertices of each face of a polygon is very important, and whenever you design a polyhedron, you should plan your polygons so that their vertices are ordered in a sequence that is counterclockwise as seen from outside the polyhedron (or, to put it another way, that the angle to each vertex as seen from a point inside the face is increasing rather than decreasing as you go around each face).

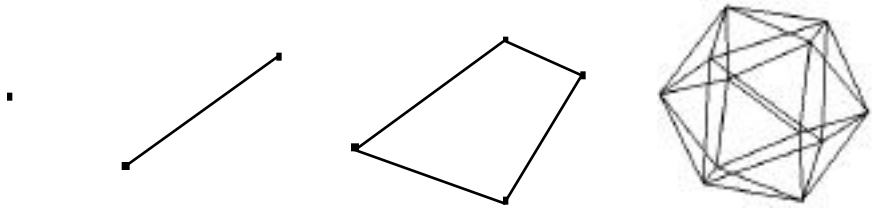


Figure 2.1: a point, a line segment, a polygon, and a polyhedron

Before you can create an image, you must define the objects that are to appear in that image through some kind of modeling process. Perhaps the most difficult—or at least the most time-consuming—part of beginning graphics programming is creating the models that are part of the

image you want to create. Part of the difficulty is in designing the objects themselves, which may require you to sketch parts of your image by hand so you can determine the correct values for the points used in defining it, for example, or it may be possible to determine the values for points from some other technique. Another part of the difficulty is actually entering the data for the points in an appropriate kind of data structure and writing the code that will interpret this data as points, line segments, and polygons for the model. But until you get the points and their relationships right, you will not be able to get the image right.

Definitions

We need to have some common terminology as we talk about modeling. We will think of modeling as the process of defining the objects that are part of the scene you want to view in an image. There are many ways to model a scene for an image; in fact, there are a number of commercial programs you can buy that let you model scenes with very high-level tools. However, for much graphics programming, and certainly as you are beginning to learn about this field, you will probably want to do your modeling by defining your geometry in terms of relatively simple primitive terms so you may be fully in control of the modeling process.

Besides defining a single point, line segment, or polygon, graphics APIs provide modeling support for defining larger objects that are made up of several simple objects. These can involve disconnected sets of objects such as points, line segments, quads, or triangles, or can involve connected sets of points, such as line segments, quad strips, triangle strips, or triangle fans. This allows you to assemble simpler components into more complex groupings and is often the only way you can define polyhedra for your scene. Some of these modeling techniques involve a concept called *geometry compression,* which allow you to define a geometric object using fewer vertices than would normally be needed. The OpenGL support for geometry compression will be discussed as part of the general discussion of OpenGL modeling processes. The discussions and examples below will show you how to build your repertoire of techniques you can use for your modeling.

Before going forward, however, we need to mention another way to specify points for your models. In some cases, it can be helpful to think of your 3-dimensional space as embedded as an affine subspace of 4-dimensional space. If we think of 4-dimensional space as having X, Y, Z, and W components, this embedding identifies the three-dimensional space with the subspace $W=1$ of the four-dimensional space, so the point (x, y, z) is identified with the four-dimensional point $(x, y, z, 1)$. Conversely, the four-dimensional point (x, y, z, w) is identified with the three-dimensional point $(x/w, y/w, z/w)$ whenever $w \neq 0$. The four-dimensional representation of points with a non-zero `w` component is called *homogeneous coordinates*, and calculating the three-dimensional equivalent for a homogeneous representation by dividing by `w` is called *homogenizing* the point. When we discuss transformations, we will sometimes think of them as 4x4 matrices because we will need them to operate on points in homogeneous coordinates.

Not all points in 4-dimensional space can be identified with points in 3-space, however. The point $(x, y, z, 0)$ is not identified with a point in 3-space because it cannot be homogenized, but it is identified with the direction defined by the vector $\langle x, y, z \rangle$. This can be thought of as a "point at infinity" in a certain direction. This has an application in the chapter below on lighting when we discuss directional instead of positional lights, but in general we will not encounter homogeneous coordinates often in these notes.

Some examples

In this section we will describe the kinds of simple objects that are directly supported by most graphics APIs. We begin with very simple objects and proceed to more complex ones, but you

will find that both simple and complex objects will be needed in your work. With each kind of primitive object, we will describe how that object is specified, and in later examples, we will create a set of points and will then show the function call that draws the object we have defined.

Point and points

To draw a single point, we will simply define the coordinates of the point and give them to the graphics API function that draws points. Such a function can typically handle one point or a number of points, so if we want to draw only one point, we provide only one vertex; if we want to draw more points, we provide more vertices. Points are extremely fast to draw, and it is not unreasonable to draw tens of thousands of points if a problem merits that kind of modeling. On a very modest-speed machine without any significant graphics acceleration, a 50,000 point model can be re-drawn in a small fraction of a second.

Line segments

To draw a single line segment, we must simply supply two vertices to the graphics API function that draws lines. Again, this function will probably allow you to specify a number of line segments and will draw them all; for each segment you simply need to provide the two endpoints of the segment. Thus you will need to specify twice as many vertices as the number of line segments you wish to produce.

The simple way that a graphics API handles lines hides an important concept, however. A line is a continuous object with real-valued coordinates, and it is displayed on a discrete object with integer screen coordinates. This is, of course, the difference between model space and eye space on one hand and screen space on the other. While we focus on geometric thinking in terms that overlook the details of conversions from eye space to screen space, you need to realize that algorithms for such conversions lie at the foundation of computer graphics and that your ability to think in higher-level terms is a tribute to the work that has built these foundations.

Connected lines

Connected lines—collections of line segments that are joined "head to tail" to form a longer connected group—are shown in Figure 2.2. These are often called line strips and line loops, and your graphics API will probably provide a function for drawing them. The vertex list you use will define the line segments by using the first two vertices for the first line segment, and then by using

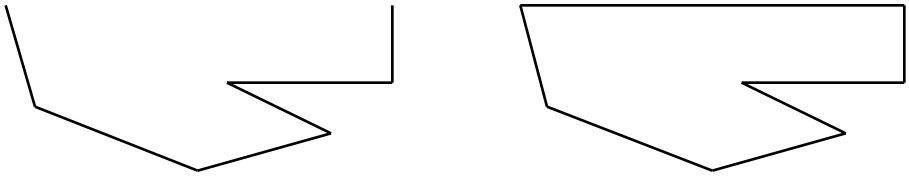


Figure 2.2: a line strip and a line loop

each new vertex and its predecessor to define each additional segment. The difference between a line strip and a line loop is that the former does not connect the last vertex defined to the first vertex, leaving the figure open; the latter includes this extra segment and creates a closed figure. Thus the number of line segments drawn by the a line strip will be one fewer than the number of vertices in the vertex list, while a line loop will draw the same number of segments as vertices. This is a geometry compression technique because to define a line strip with N segments you only specify $N+1$ vertices instead of $2N$ vertices; instead of needing to define two points per line segment, each segment after the first only needs one vertex to be defined.

Triangle

To draw one or more unconnected triangles, your graphics API will provide a simple triangle-drawing function. With this function, each set of three vertices will define an individual triangle so that the number of triangles defined by a vertex list is one third the number of vertices in the list. The humble triangle may seem to be the most simple of the polygons, but as we noted earlier, it is probably the most important because no matter how you use it, and no matter what points form its vertices, it always lies in a plane. Because of this, most polygon-based modeling really comes down to triangle-based modeling in the end, and almost every kind of graphics tool knows how to manage objects defined by triangles. So treat this humblest of polygons well and learn how to think about polygons and polyhedra in terms of the triangles that make them up.

Sequence of triangles

Triangles are the foundation of most truly useful polygon-based graphics, and they have some very useful capabilities. Graphics APIs often provide two different geometry-compression techniques to assemble sequences of triangles into your image: triangle strips and triangle fans. These techniques can be very helpful if you are defining a large graphic object in terms of the triangles that make up its boundaries, when you can often find ways to include large parts of the object in triangle strips and/or fans. The behavior of each is shown in Figure 2.3 below. Note that this figure and similar figures that show simple geometric primitives are presented as if they were drawn in 2D space. In fact they are not, but in order to make them look three-dimensional we would need to use some kind of shading, which is a separate concept discussed in a later chapter (and which is used to present the triangle fan of Figure 2.18). We thus ask you to think of these as three-dimensional, even though they look flat.

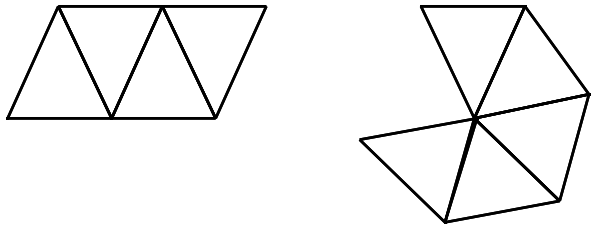


Figure 2.3: triangle strip and triangle fan

Most graphics APIs support both techniques by interpreting the vertex list in different ways. To create a triangle strip, the first three vertices in the vertex list create the first triangle, and each vertex after that creates a new triangle with the two vertices immediately before it. We will see in later chapters that the order of points around a polygon is important, and we must point out that these two techniques behave quite differently with respect to polygon order; for triangle fans, the orientation of all the triangles is the same (clockwise or counterclockwise), while for triangle strips, the orientation of alternate triangles is reversed. This may require some careful coding when lighting models are used. To create a triangle fan, the first three vertices create the first triangle and each vertex after that creates a new triangle with the point immediately before it and the first point in the list. In each case, the number of triangles defined by the vertex list is two less than the number of vertices in the list, so these are very efficient ways to specify triangles.

Quadrilateral

A convex quadrilateral, often called a “quad” to distinguish it from a general quadrilateral because the general quadrilateral need not be convex, is any convex 4-sided figure. The function in your graphics API that draws quads will probably allow you to draw a number of them. Each

quadrilateral requires four vertices in the vertex list, so the first four vertices define the first quadrilateral, the next four the second quadrilateral, and so on, so your vertex list will have four times as many points as there are quads. The sequence of vertices is that of the points as you go around the perimeter of the quadrilateral. In an example later in this chapter, we will use six quadrilaterals to define a cube that will be used in later examples.

Sequence of quads

You can frequently find large objects that contain a number of connected quads. Most graphics APIs have functions that allow you to define a sequence of quads. The vertices in the vertex list are taken as vertices of a sequence of quads that share common sides. For example, the first four vertices can define the first quad; the last two of these, together with the next two, define the next quad; and so on The order in which the vertices are presented is shown in Figure 2.4. Note the order of the vertices; instead of the expected sequence around the quads, the points in each pair have the same order. Thus the sequence 3-4 is the opposite order than would be expected, and this same sequence goes on in each additional pair of extra points. This difference is critical to note when you are implementing quad strip constructions. It might be helpful to think of this in terms of triangles, because a quad strip acts as though its vertices were specified as if it were really a triangle strip — vertices 1/2/3 followed by 2/3/4 followed by 3/4/5 etc.

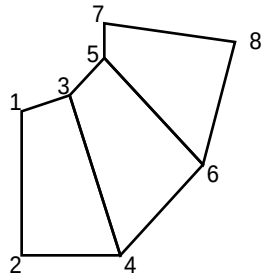


Figure 2.4: sequence of points in a quad strip

As an example of the use of quad strips and triangle fans, let's create your a model of a sphere. As we will see in the next chapter, both the GLU and GLUT toolkits include pre-built sphere models, but the sphere is a familiar object and it can be helpful to see how to create familiar things with new tools. There may also be times when you need to do things with a sphere that are difficult with the pre-built objects, so it is useful to have this example in your "bag of tricks."

In the chapter on mathematical fundamentals, we will describe the use of spherical coordinates in modeling. We can use spherical coordinates to model the sphere at first, and then we can later convert to Cartesian coordinates as we describe in that chapter to present the model to the graphics system for actual drawing. Let's think of creating a model of the sphere with N divisions around the equator and $N/2$ divisions along the prime meridian. In each case, then, the angular division will be *theta* = $360/N$ degrees. Let's also think of the sphere as having a unit radius, so it will be easier to work with later when we have transformations. Then the basic structure would be:

```
// create the two polar caps with triangle fans
doTriangleFan()      // north pole
   set vertex at (1, 0, 90)
   for i = 0 to N
      set vertex at (1, 360/i, 90-180/N)
endTriangleFan()
doTriangleFan()      // south pole
   set vertex at (1, 0, -90)
   for i = 0 to N
```

```
      set vertex at (1, 360/i, -90+180/N)
endTriangleFan()
// create the body of the sphere with quad strips
for j = -90+180/N to 90 - 180/2N
   // one quad strip per band around the sphere at a given latitude
   doQuadStrip()
      for i = 0 to 360
         set vertex at (1, i, j)
         set vertex at (1, i, j+180/N)
         set vertex at (1, i+360/N, j)
         set vertex at (1, i+360/N, j+180/N)
   endQuadStrip()
```

Note the order in which we set the points in the triangle fans and in the quad strips, as we described when we introduced these concepts; this is not immediately an obvious order and you may want to think about it a bit. Because we're working with a sphere, the quad strips as we have defined them are planar, so there is no need to divide each quad into two triangles to get planar surfaces as we might want to do for other kinds of objects.

General polygon

Some images need to include more general kinds of polygons. While these can be created by constructing them manually as collections of triangles and/or quads, it might be easier to define and display a single polygon. A graphics API will allow you to define and display a single polygon by specifying its vertices, and the vertices in the vertex list are taken as the vertices of the polygon in sequence order. As we will note in the chapter on mathematical fundamentals, many APIs can only handle *convex* polygons—polygons for which any two points in the polygon also have the entire line segment between them in the polygon. We refer you to that earlier discussion for more details.

Polyhedron

In Figure 2.1 we saw that a polyhedron is one of the basic objects we use in our modeling, especially when we will focus almost exclusively on 3D computer graphics. We specify a polyhedron by specifying all the polygons that make up its boundary. In general, most graphics APIs leave the specification of polyhedrons up to the user, which can make them fairly difficult objects to define as you are learning the subject. With experience, however, you will develop a set of polyhedra that you're familiar with and can use them with comfort.

While a graphics API may not have a general set of polyhedra, however, some provide a set of basic polyhedra that can be very useful to you. These depend on the API so we cannot be more specific here, but the next chapter includes a description of the polyhedra provided by OpenGL.

Aliasing and antialiasing

When you create a point, line, or polygon in your image, the system will define the pixels on the screen that represent the geometry within the discrete integer-coordinate 2D screen space. The standard way of selecting pixels is all-or-none: a pixel is computed to be either in the geometry, in which case it is colored as the geometry specifies, or not in the geometry, in which case it is left in whatever color it already was. Because of the relatively coarse nature of screen space, this all-or-nothing approach can leave a great deal to be desired because it created jagged edges along the space between geometry and background. This appearance is called *aliasing*, and it is shown in the left-hand image of Figure 2.4.

There are a number of techniques to reduce the effects of aliasing, and collectively the techniques are called *antialiasing*. They all work by recognizing that the boundary of a true geometry can go through individual pixels in a way that only partially covers a pixel. Each technique finds a way to account for this varying coverage and then lights the pixel according to the amount of coverage of the pixel with the geometry. Because the background may vary, this variable lighting is often managed by controlling the blending value for the pixel's color, using the color (R, G, B, A) where (R, G, B) is the geometry color and A is the proportion of the pixel covered by the object's geometry. An image that uses antialiasing is shown in the right-hand image of Figure 2.4. For more detail on color blending, see the later chapter on color.

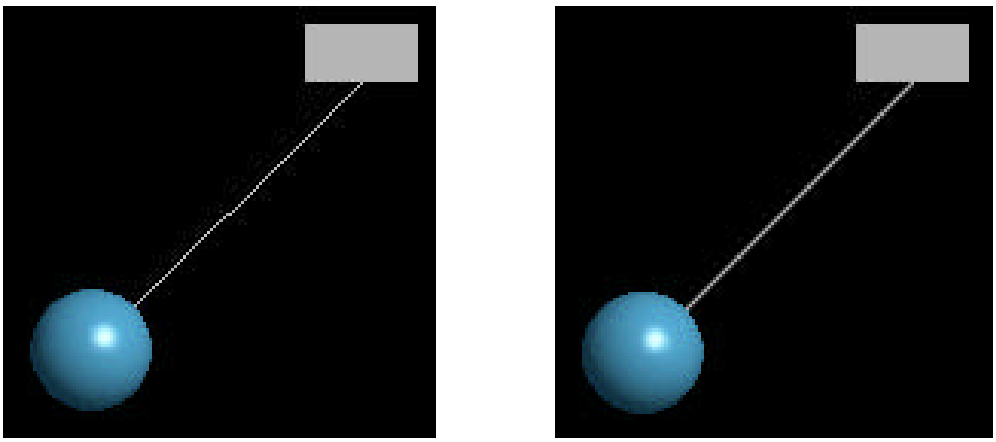


Figure 2.4: aliased lines (left) and antialiased lines (right)

As we said, there are many ways to determine the coverage of a pixel by the geometry. One way that is often used for very high-quality images is to *supersample* the pixel, that is, to assume a much higher image resolution than is really present and to see how many of these "subpixels" lie in the geometry. The proportion of subpixels that would be covered will serve as the proportion for the antialiasing value. However, supersampling is not an ordinary function of a graphics API, so we would expect a simpler approach to be used. Because APIs use linear geometries—all the basic geometry is polygon-based—it is possible to calculate exactly how the 2D world space line intersects each pixel and then how much of the pixel is covered. This is a more standard kind of API computation, though the details will certainly vary between APIs and even between different implementations of an API. You may want to look at your API's manuals for more details.

Normals

When you define the geometry of an object, you may also want or need to define the direction the object faces as well as the coordinate values for the point. This is done by defining a normal for the object. Normals are often fairly easy to obtain. In the appendix to this chapter you will see ways to calculate normals for plane polygons fairly easily; for many of the kinds of objects that are available with a graphics API, normals are built into the object definition; and if an object is defined by mathematical formulas, you can often get normals by doing some straightforward calculations.

The sphere described above is a good example of getting normals by calculation. For a sphere, the normal to the sphere at a given point is the radius vector at that point. For a unit sphere with center at the origin, the radius vector to a point has the same components as the coordinates of the point. So if you know the coordinates of the point, you know the normal at that point.

To add the normal information to the modeling definition, then, you can simply use functions that set the normal for a geometric primitive, as you would expect to have from your graphics API, and you would get code that looks something like the following excerpt from the example above:

```
for j = -90+180/M to 90-180/M  // latitude without sphere caps
   doQuadStrip()
   // one quad strip per band around the sphere at any latitude
      for i = 0 to 360          // longitude
         set normal to (1, i, j)
         set vertex at (1, i,       j      )
         set vertex at (1, i,       j+180/M)
         set vertex at (1, i+360/N, j      )
         set vertex at (1, i+360/N, j+180/M)
   endQuadStrip()
```

Data structures to hold objects

When you define a polyhedron for your graphics work, as we discussed above, there are many ways you can hold the information that describes a polyhedral graphics object. One of the simplest is the *triangle list*—an array of triples, with each set of three triples representing a separate triangle. Drawing the object is then a simple matter of reading three triples from the list and drawing the triangle. A good example of this kind of list is the STL graphics file format discussed in the chapter below on graphics hardcopy and whose formal specifications are in the Appendix.

A more effective, though a bit more complex, approach is to create three lists. The first is a *vertex list*, and it is simply an array of triples that contains all the vertices that would appear in the object. If the object is a polygon or contains polygons, the second list is an edge list that contains an entry for each edge of the polygon; the entry is an ordered pair of numbers, each of which is an index of a point in the vertex list. If the object is a polyhedron, the third is a *face list*, containing information on each of the faces in the polyhedron. Each face is indicated by listing the indices of all the edges that make up the face, in the order needed by the orientation of the face. You can then draw the face by using the indices as an indirect reference to the actual vertices. So to draw the object, you loop across the face list to draw each face; for each face you loop across the edge list to determine each edge, and for each edge you get the vertices that determine the actual geometry.

As an example, let's consider the classic cube, centered at the origin and with each side of length two. For the cube let's define the vertex array, edge array, and face array that define the cube, and let's outline how we could organize the actual drawing of the cube. We will return to this example later in this chapter and from time to time as we discuss other examples throughout the notes.

We begin by defining the data and data types for the cube. The vertices are points, which are arrays of three points, while the edges are pairs of indices of points in the point list and the faces are quadruples of indices of faces in the face list. The normals are vectors, one per face, but these are also given as arrays of three points. In C, these would be given as follows:

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];    // each face of a cube has four edges

point3 vertices[8] = {{-1.0, -1.0, -1.0},
                      {-1.0, -1.0,  1.0},
                      {-1.0,  1.0, -1.0},
                      {-1.0,  1.0,  1.0},
                      { 1.0, -1.0, -1.0},
                      { 1.0, -1.0,  1.0},
```

```
                        { 1.0,  1.0, -1.0},
                        { 1.0,  1.0,  1.0} };

point3 normals[6]   ={{ 0.0, 0.0, 1.0},
                        {-1.0, 0.0, 0.0},
                        { 0.0, 0.0,-1.0},
                        { 1.0, 0.0, 0.0},
                        { 0.0,-1.0, 0.0},
                        { 0.0, 1.0, 0.0} };

edge   edges[24]    = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                        { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                        { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                        { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                        { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                        { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

face   cube[6]      = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                        { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                        {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

Notice that in our edge list, each edge is actually listed twice—once for each direction the in which the edge can be drawn. We need this distinction to allow us to be sure our faces are oriented properly, as we will describe in the discussion on lighting and shading in later chapters. For now, we simply ensure that each face is drawn with edges in a counterclockwise direction as seen from outside that face of the cube. Drawing the cube, then, proceeds by working our way through the face list and determining the actual points that make up the cube so they may be sent to the generic (and fictitious) `setVertex(...)` and `setNormal(...)` functions. In a real application we would have to work with the details of a graphics API, but here we sketch how this would work in a pseudocode approach. In this pseudocode, we assume that there is no automatic closure of the edges of a polygon so we must list both the vertex at both the beginning and the end of the face when we define the face; if this is not needed by your API, then you may omit the first `setVertex` call in the pseudocode for the function `cube()` below.

```
void cube(void) {
   for faces 1 to 6
      start face
         setNormal(normals[i]);
         setVertex(vertices[edges[cube[face][0]][0]);
         for each edge in the face
            setVertex(vertices[edges[cube[face][edge]][1]);
      end face
}
```

We added a simple structure for a list of normals, with one normal per face, which echos the structure of the faces. This supports what is often called flat shading, or shading where each face has a single color. In many applications, though, you might want to have smooth shading, where colors blend smoothly across each face of your polygon. For this, each vertex needs to have its individual normal representing the perpendicular to the object at that vertex. In this case, you often need to specify the normal each time you specify a vertex, and a normal list that follows the vertex list would allow you to do that easily. For the code above, for example, we would not have a per-face normal but instead each `setVertex` operation could be replaced by the pair of operations

```
setNormal(normals[edges[cube[face][0]][0]);
setVertex(vertices[edges[cube[face][0]][0]);
```

Neither the simple triangle list nor the more complex structure of vertex, normal, edge, and face lists takes into account the very significant savings in memory you can get by using geometry compression techniques. There are a number of these techniques, but we only talked about line strips, triangle strips, triangle fans, and quad strips above because these are more often supported by a graphics API. Geometry compression approaches not only save space, but are also more effective for the graphics system as well because they allow the system to retain some of the information it generates in rendering one triangle or quad when it goes to generate the next one.

Additional sources of graphic objects

Interesting and complex graphic objects can be difficult to create, because it can take a lot of work to measure or calculate the detailed coordinates of each vertex needed. There are more automatic techniques being developed, including 3D scanning techniques and detailed laser rangefinding to measure careful distances and angles to points on an object that is being measured, but they are out of the reach of most college classrooms. So what do we do to get interesting objects? There are four approaches.

The first way to get models is to buy them: to go is to the commercial providers of 3D models. There is a serious market for some kinds of models, such as medical models of human structures, from the medical and legal worlds. This can be expensive, but it avoids having to develop the expertise to do professional modeling and then putting in the time to create the actual models. If you are interested, an excellent source is viewpoint.com; they can be found on the Web.

A second way to get models is to find them in places where people make them available to the public. If you have friends in some area of graphics, you can ask them about any models they know of. If you are interested in molecular models, the protein data bank (with URL `http://www.pdb.bnl.gov`) has a wide range of structure models available at no charge. If you want models of all kinds of different things, try the site `avalon.viewpoint.com`; this contains a large number of public-domain models contributed to the community by generous people over the years.

A third way to get models is to digitize them yourself with appropriate kinds of digitizing devices. There are a number of these available with their accuracy often depending on their cost, so if you need to digitize some physical objects you can compare the cost and accuracy of a number of possible kinds of equipment. The digitizing equipment will probably come with tools that capture the points and store the geometry in a standard format, which may or may not be easy to use for your particular graphics API. If it happens to be one that your API does not support, you may need to convert that format to one you use or to find a tool that does that conversion.

A fourth way to get models is to create them yourself. There are a number of tools that support high-quality interactive 3D modeling, and it is perfectly reasonable to create your models with such tools. This has the same issue as digitizing models in terms of the format of the file that the tools produce, but a good tool should be able to save the models in several formats, one of which you should be able to use fairly easily with your graphics API. It is also possible to create interesting models analytically, using mathematical approaches to generate the vertices. This is perhaps slower than getting them from other sources, but you have final control over the form and quality of the model, so perhaps it might be worth the effort. This will be discussed in the chapter on interpolation and spline modeling, for example.

If you get models from various sources, you will probably find that they come in a number of different kinds of data format. There are a large number of widely used formats for storing graphics information, and it sometimes seems as though every graphics tool uses a file format of its own. Some available tools will open models with many formats and allow you to save them in

a different format, essentially serving as format converters as well as modeling tools. In any case, you are likely to end up needing to understand some model file formats and writing your own tools to read these formats and produce the kind of internal data that you need for your models, and it may take some work to write filters that will read these formats into the kind of data structures you want for your program. Perhaps things that are "free" might cost more than things you buy if you can save the work of the conversion, but that's up to you to decide. An excellent resource on file formats is the *Encyclopedia of Graphics File Formats*, published by O'Reilly Associates, and we refer you to that book for details on particular formats.

A word to the wise

As we said above, modeling can be the most time-consuming part of creating an image, but you simply aren't going to create a useful or interesting image unless the modeling is done carefully and well. If you are concerned about the programming part of the modeling for your image, it might be best to create a simple version of your model and get the programming (or other parts that we haven't talked about yet) done for that simple version. Once you are satisfied that the programming works and that you have gotten the other parts right, you can replace the simple model—the one with just a few polygons in it—with the one that represents what you really want to present.

Transformations and Modeling

This section requires some understanding of 3D geometry, particularly a sense of how objects can be moved around in 3-space. You should also have some sense of how the general concept of stacks works.

Introduction

Transformations are probably the key point in creating significant images in any graphics system. It is extremely difficult to model everything in a scene in the place where it is to be placed, and it is even worse if you want to move things around in real time through animation and user control. Transformations let you define each object in a scene in any space that makes sense for that object, and then place it in the world space of a scene as the scene is actually viewed. Transformations can also allow you to place your eyepoint and move it around in the scene.

There are several kinds of transformations in computer graphics: projection transformations, viewing transformations, and modeling transformations. Your graphics API should support all of these, because all will be needed to create your images. Projection transformations are those that specify how your scene in 3-space is mapped to the 2D screen space, and are defined by the system when you choose perspective or orthogonal projections; viewing transformations are those that allow you to view your scene from any point in space, and are set up when you define your view environment, and modeling transformations are those you use to create the items in your scene and are set up as you define the position and relationships of those items. Together these make up the graphics pipeline that we discussed in the first chapter of these notes.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. These all maintain the basic geometry of any object to which they may be applied, and are fundamental tools to build more general models than you can create with only simple modeling techniques. Later in this chapter we will describe the relationship between objects in a scene and how you can build and maintain these relationships in your programs.

The real power of modeling transformation, though, does not come from using these simple transformations on their own. It comes from combining them to achieve complete control over your modeled objects. The individual simple transformations are combined into a composite modeling transformation that is applied to your geometry at any point where the geometry is specified. The modeling transformation can be saved at any state and later restored to that state to allow you to build up transformations that locate groups of objects consistently. As we go through the chapter we will see several examples of modeling through composite transformations.

Finally, the use of simple modeling and transformations together allows you to generate more complex graphical objects, but these objects can take significant time to display. You may want to store these objects in pre-compiled display lists that can execute much more quickly.

Definitions

In this section we outline the concept of a geometric transformation and describe the fundamental transformations used in computer graphics, and describe how these can be used to build very general graphical object models for your scenes.

Transformations

A transformation is a function that takes geometry and produces new geometry. The geometry can be anything a computer graphics systems works with—a projection, a view, a light, a direction, or

an object to be displayed. We have already talked about projections and views, so in this section we will talk about projections as modeling tools. In this case, the transformation needs to preserve the geometry of the objects we apply them to, so the basic transformations we work with are those that maintain geometry, which are the three we mentioned earlier: rotations, translations, and scaling. Below we look at each of these transformations individually and together to see how we can use transformations to create the images we need.

Our vehicle for looking at transformations will be the creation and movement of a rugby ball. This ball is basically an ellipsoid (an object that is formed by rotating an ellipse around its major axis), so it is easy to create from a sphere using scaling. Because the ellipsoid is different along one axis from its shape on the other axes, it will also be easy to see its rotations, and of course it will be easy to see it move around with translations. So we will first discuss scaling and show how it is used to create the ball, then discuss rotation and show how the ball can be rotated around one of its short axes, then discuss translations and show how the ball can be moved to any location we wish, and finally will show how the transformations can work together to create a rotating, moving ball like we might see if the ball were kicked. The ball is shown with some simple lighting and shading as described in the chapters below on these topics.

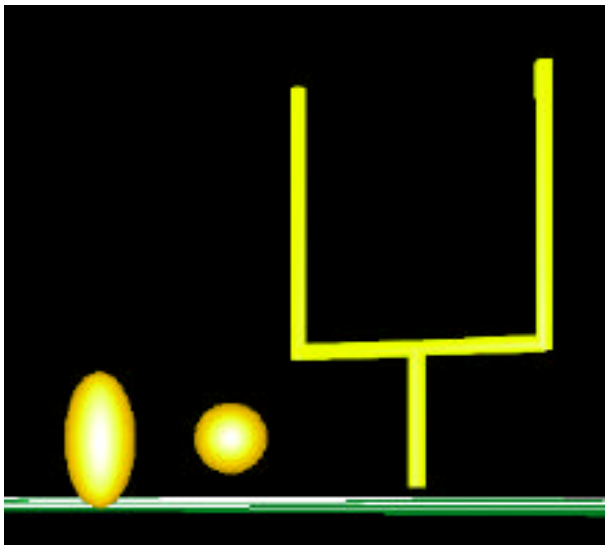


Figure 2.5: a sphere a scaled by 2.0 in the y-direction to make a rugby ball (left) and the same sphere is shown unscaled (right)

Scaling changes the entire coordinate system in space by multiplying each of the coordinates of each point by a fixed value. Each time it is applied, this changes each dimension of everything in the space. A scaling transformation requires three values, each of which controls the amount by which one of the three coordinates is changed, and a graphics API function to apply a scaling transformation will take three real values as its parameters. Thus if we have a point (x, y, z) and specify the three scaling values as Sx, Sy, and Sz, then applying the scaling transformation changes the point to $(x \quad Sx, y \quad Sy, z \cdot Sz)$. If we take a simple sphere that is centered at the origin and scale it by 2.0 in one direction (in our case, the y-coordinate or vertical direction), we get the rugby ball that is shown in Figure 2.5 next to the original sphere. It is important to note that this scaling operates on everything in the space, so if we happen to also have a unit sphere at position farther out along the axis, scaling will move the sphere farther away from the origin and will also multiply each of its coordinates by the scaling amount, possibly distorting its shape. This shows that it is most useful to apply scaling to an object defined at the origin so only the dimensions of the object will be changed.

Rotation takes everything in your space and changes each coordinate by rotating it around the origin of the geometry in which the object is defined. The rotation will always leave a line through the origin in the space fixed, that is, will not change the coordinates of any point on that line. To define a rotation transformation, you need to specify the amount of the rotation (in degrees or radians, as needed) and the line about which the rotation is done. A graphics API function to apply a rotation transformation, then, will take the angle and the line as its parameters; remember that a line through the origin can be specified by three real numbers that are the coordinates of the direction vector for that line. It is most useful to apply rotations to objects centered at the origin in order to change only the orientation with the transformation.

Translation takes everything in your space and changes each point's coordinates by adding a fixed value to each coordinate. The effect is to move everything that is defined in the space by the same amount. To define a translation transformation, you need to specify the three values that are to be added to the three coordinates of each point. A graphics API function to apply a translation, then, will take these three values as its parameters. A translation shows a very consistent treatment of everything in the space, so a translation is usually applied after any scaling or rotation in order to take an object with the right size and right orientation and place it correctly in space.

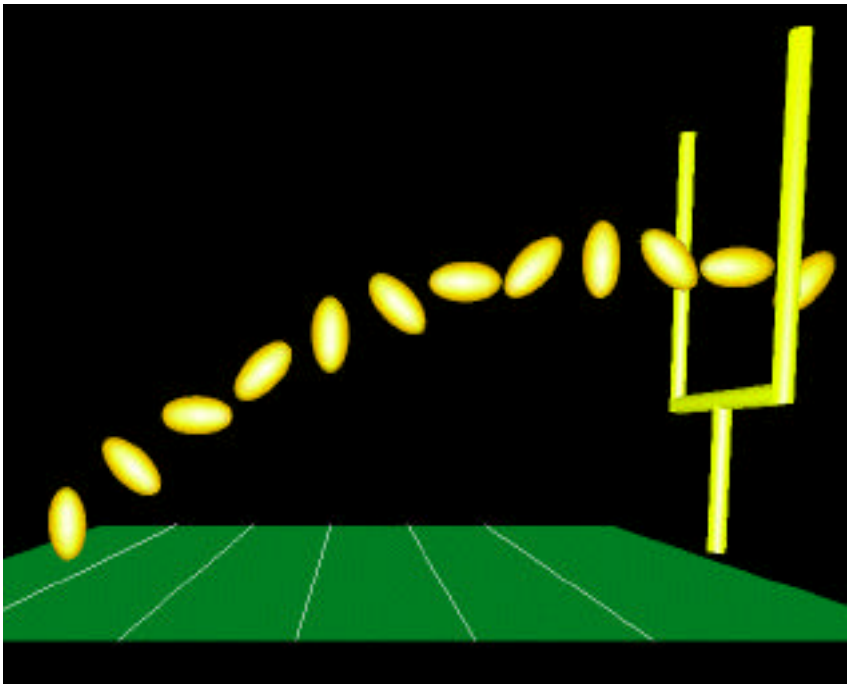


Figure 2.6: a sequence of images of the rugby ball as transformations move it through space

Finally, we put these three kinds of transformations together to create a sequence of images of the rugby ball as it moves through space, rotating as it goes, shown in Figure 2.6. This sequence was created by first defining the rugby ball with a scaling transformation and a translation putting it on the ground appropriately, creating a composite transformation as discussed in the next section. Then rotation and translation values were computed for several times in the flight of the ball, allowing us to rotate the ball by slowly-increasing amounts and placing it as if it were in a standard gravity field. Each separate image was created with a set of transformations that can be generically described by

```
translate( Tx, Ty, Tz )
rotate( angle, x-axis )
scale( 1., 2., 1. )
drawBall()
```

where the operation `drawBall()` was defined as

```
translate( Tx, Ty, Tz )
scale( 1., 2., 1. )
drawSphere()
```

Notice that the ball rotates in a slow counterclockwise direction as it travel from left to right, while the position of the ball describes a parabola as it moves, modeling the effect of gravity on the ball's flight. This kind of composite transformation constructions is described in the next section, and as we point out there, the order of these transformation calls is critical in order to achieve the effect we need.

Transformations are mathematical operations that map 3D space to 3D space, and so mathematics has standard ways to represent them. This is discussed in the next chapter, and processes such as composite transformations are linked to the standard operations on these objects.

Composite transformations

In order to achieve the image you want, you may need to apply more than one simple transformation to achieve what is called a composite transformation. For example, if you want to create a rectangular box with height *A*, width *B*, and depth *C*, with center at (*C*1, *C*2, *C*3), and oriented at an angle α relative to the *Z*-axis, you could start with a cube one unit on a side and with center at the origin, and get the box you want by applying the following sequence of operations:

first, scale the cube to the right size to create the rectangular box with dimensions *A*, *B*, and *C*,
second, rotate the cube by the angle α to the right orientation, and
third, translate the cube to the position *C*1, *C*2, *C*3.

This sequence is critical because of the way transformations work in the whole space. For example, if we rotated first and then scaled with different scale factors in each dimension, we would introduce distortions in the box. If we translated first and then rotated, the rotation would move the box to an entirely different place. Because the order is very important, we find that there are certain sequences of operations that give predictable, workable results, and the order above is the one that works best: apply scaling first, apply rotation second, and apply translation last.

The order of transformations is important in ways that go well beyond the translation and rotation example above. In general, transformations are an example of *noncommutative* operations, operations for which $f*g \neq g*f$ (that is, $f(g(x)) \neq g(f(x))$). Unless you have some experience with noncommutative operations from a course such as linear algebra, this may be a new idea. But let's look at the operations we described above: if we take the point (1, 1, 0) and apply a rotation by 90° around the *Z*-axis, we get the point (-1, 1, 0). If we then apply a translation by (2, 0, 0) we get the point (1, 1, 0) again. However, if we start with (1, 1, 0) and first apply the translation, we get (3, 1, 0) and if then apply the rotation, we get the point (-1, 3, 0) which is certainly not the same as (1, 1, 0). That is, using some pseudocode for rotations, translations, and point setting, the two code sequences

```
rotate(90, 0, 0, 1)          translate(2, 0, 0)
translate (2, 0, 0)          rotate(90, 0, 0, 1)
setPoint(1, 1, 0)            setPoint(1, 1, 0)
```

produce very different results; that is, the rotate and translate operations are not commutative.

This behavior is not limited to different kinds of transformations. Different sequences of rotations can result in different images as well. Again, if you consider the sequence of rotations in two different orders

```
rotate(60, 0, 0, 1)          rotate(90, 0, 1, 0)
rotate(90, 0, 1, 0)          rotate(60, 0, 0, 1)
scale(3, 1, .5)              scale(3, 1, .5)
cube()                       cube()
```

then the results are quite different, as is shown in Figure 2.7.

Figure 2.7: the results from two different orderings of the same rotations

Transformations are implemented as matrices for computational purposes. Recall that we are able to represent points as 4-tuples of real numbers; transformations are implemented as 4x4 matrices that map the space of 4-tuples into itself. Although we will not explicitly use this representation in our work, it is used by graphics APIs and helps explain how transformations work; for example, you can understand why transformations are not commutative by understanding that matrix multiplication is not commutative. (Try it out for yourself!) And if we realize that a 4x4 matrix is equivalent to an array of 16 real numbers, we can think of transformation stacks as stacks of such matrices. While this book does not require matrix operations for transformations, there may be times when you'll need to manipulate transformations in ways that go beyond your API, so be aware of this.

When it comes time to apply transformations to your models, we need to think about how we represent the problem for computational purposes. Mathematical notation can be applied in many ways, so your previous mathematical experience may or may not help you very much in deciding how you can think about this problem. In order to have a good model for thinking about complex transformation sequences, we will define the sequence of transformations as *last-specified, first-applied*, or in another way of thinking about it, we want to apply our functions so that the function nearest to the geometry is applied first. We can also think about this is in terms of building composite functions by multiplying the individual functions, and with the convention above we want to compose each new function by multiplying it on the right of the previous functions. So the standard operation sequence we see above would be achieved by the following algebraic sequence of operations:

```
translate * rotate * scale * geometry
```

or, thinking of multiplication as function composition, as

```
translate(rotate(scale(geometry)))
```

This might be implemented by a sequence of function calls like that below that is not intended to represent any particular API:

```
translate(C1,C2,C3); // translate to the desired point
rotate(A, Z);        // rotate by A around the Z-axis
scale(A, B, C);      // scale by the desired amounts
cube();              // define the geometry of the cube
```

At first glance, this sequence looks to be exactly the opposite of the sequence noted above. In fact, however, we readily see that the scaling operation is the function closest to the geometry (which is expressed in the function `cube()`) because of the last-specified, first-applied nature of transformations. In Figure 2.8 we see the sequence of operations as we proceed from the plain cube (at the left), to the scaled cube next, then to the scaled and rotated cube, and finally to the cube that uses all the transformations (at the right). The application is to create a long, thin, rectangular bar that is oriented at a 45° angle upwards and lies above the definition plane.

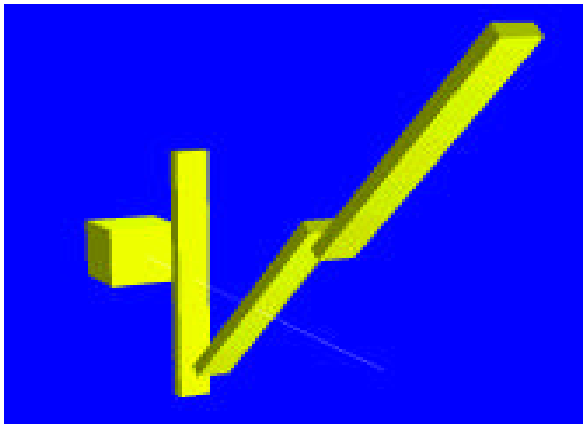


Figure 2.8: the sequence of figures as a cube is transformed

In general, the overall sequence of transformations that are applied to a model by considering the total sequence of transformations in the order in which they are specified, as well as the geometry on which they work:

```
P  V  T0 T1 T2 …  Tn Tn+1  …  Tlast  …  geometry
```

Here, `P` is the projection transformation, `V` is the viewing transformation, and `T0`, `T1`, … `Tlast` are the transformations specified in the program to model the scene, in order (`T1` is first, `Tlast` is last). The projection transformation is defined in the `reshape` function; the viewing transformation is defined in the `init` function or at the beginning of the `display` function so it is defined at the beginning of the modeling process. But the sequence is actually applied in reverse: `Tlast` is actually applied first, and `V` and finally `P` are applied last. The code would then have the definition of `P` first, the definition of `V` second, the definitions of `T0`, `T1`, … `Tlast` next in order, and the definition of the geometry last. You need to understand this sequence very well, because it's critical to understand how you build complex hierarchical models.

Transformation stacks and their manipulation

In defining a scene, we often want to define some standard pieces and then assemble them in standard ways, and then use the combined pieces to create additional parts, and go on to use these parts in additional ways. To do this, we need to create individual parts through functions that do not pay any attention to ways the parts will be used later, and then be able to assemble them into a whole. Eventually, we can see that the entire image will be a single whole that is composed of its various parts.

The key issue is that there is some kind of transformation in place when you start to define the object. When we begin to put the simple parts of a composite object in place, we will use some transformations but we need to undo the effect of those transformations when we put the next part in place. In effect, we need to save the state of the transformations when we begin to place a new part, and then to return to that transformation state (discarding any transformations we may have added past that mark) to begin to place the next part. Note that we are always adding and discarding at the end of the list; this tells us that this operation has the computational properties of a stack. We may define a stack of transformations and use it to manage this process as follows:

- as transformations are defined, they are multiplied into the current transformation in the order noted in the discussion of composite transformations above, and

- when we want to save the state of the transformation, we make a copy of the current version of the transformation and push that copy onto the stack, and apply all the subsequent transformations to the copy at the top of the stack. When we want to return to the original transformation state, we can pop the stack and throw away the copy that was removed, which gives us the original transformation so we can begin to work again at that point. Because all transformations are applied to the one at the top of the stack, when we pop the stack we return to the original context.

Designing a scene that has a large number of pieces of geometry as well as the transformations that define them can be difficult. In the next section we introduce the concept of the scene graph as a design tool to help you create complex and dynamic models both efficiently and effectively.

Compiling geometry

It can take a fair amount of time to calculate the various components of a piece of an image when that piece involves vertex lists and transformations. If an object is used frequently, and if it must be re-calculated each time it is drawn, it can make a scene quite slow to display. As a way to save time in displaying the image, many graphics APIs allow you to "compile" your geometry in a way that will allow it to be displayed much more quickly. Geometry that is to be compiled should be carefully chosen so that it is not changed between displays. If changes are needed, you will need to re-compile the object. Once you have seen what parts you can compile, you can compile them and use the compiled versions to make the display faster. We will discuss how OpenGL compiles geometry later in this chapter. If you use another API, look for details in its documentation.

A word to the wise

As we noted above, you must take a great deal of care with transformation order. It can be very difficult to look at an image that has been created with mis-ordered transformations and understand just how that erroneous example happened. In fact, there is a skill in what we might call "visual debugging"—looking at an image and seeing that it is not correct, and figuring out what errors might have caused the image as it is seen. It is important that anyone working with images become skilled in this kind of debugging. However, obviously you cannot tell than an image is wrong unless you know what a correct image should be, so you must know in general what you should be seeing. As an obvious example, if you are doing scientific images, you must know the science well enough to know when an image makes no sense.

Scene Graphs and Modeling Graphs

Introduction

In this chapter, we have defined modeling as the process of defining and organizing a set of geometry that represents a particular scene. While modern graphics APIs can provide you with a great deal of assistance in rendering your images, modeling is usually supported less well and programmers may find considerable difficulty with modeling when they begin to work in computer graphics. Organizing a scene with transformations, particularly when that scene involves hierarchies of components and when some of those components are moving, involves relatively complex concepts that need to be organized very systematically to create a successful scene. This is even more difficult when the eye point is one of the moving or hierarchically-organized parts. Hierarchical modeling has long been done by using trees or tree-like structures to organize the components of the model, and we will find this kind of approach to be very useful.

Recent graphics systems, such as Java3D and VRML 2, have formalized the concept of a *scene graph* as a powerful tool both for modeling scenes and for organizing the drawing process for those scenes. By understanding and adapting the structure of the scene graph, we can organize a careful and formal tree approach to both the design and the implementation of hierarchical models. This can give us tools to manage not only modeling the geometry of such models, but also animation and interactive control of these models and their components. In this section we will introduce the scene graph structure and will adapt it to a slightly simplified *modeling graph* that you can use to design scenes. We will also identify how this modeling graph gives us the three key transformations that go into creating a scene: the projection transformation, the viewing transformation, and the modeling transformation(s) for the scene's content. This structure is very general and lets us manage all the fundamental principles in defining a scene and translating it into a graphics API. Our terminology is based on with the scene graph of Java3D and should help anyone who uses that system understand the way scene graphs work there.

A brief summary of scene graphs

The fully-developed scene graph of the Java3D API has many different aspects and can be complex to understand fully, but we can abstract it somewhat to get an excellent model to help us think about scenes that we can use in developing the code to implement our modeling. A brief outline of the Java3D scene graph in Figure 2.9 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to beginning computer graphics. Remember that we will be simplifying some aspects of this graph before applying it to our modeling.

A *virtual universe* holds one or more (usually one) *locales*, which are essentially positions in the universe to put scene graphs. Each scene graph has two kinds of branches: *content branches*, which are to contain shapes, lights, and other content, and *view branches*, which are to contain viewing information. This division is somewhat flexible, but we will use this standard approach to build a framework to support our modeling work.

The *content branch* of the scene graph is organized as a collection of nodes that contains group nodes, transform groups, and shape nodes. A *group node* is a grouping structure that can have any number of children; besides simply organizing its children, a group can include a switch that selects which children to present in a scene. A *transform group* is a collection of modeling transformations that affect all the geometry that lies below it. The transformations will be applied to any of the transform group's children with the convention that transforms "closer" to the geometry (geometry that is defined in shape nodes lower in the graph) are applied first. A *shape node* includes both geometry and appearance data for an individual graphic unit. The geometry data includes standard 3D coordinates, normals, and texture coordinates, and can include points,

lines, triangles, and quadrilaterals, as well as triangle strips, triangle fans, and quadrilateral strips. The appearance data includes color, shading, or texture information. Lights and eye points are included in the content branch as a particular kind of geometry, having position, direction, and other appropriate parameters. Scene graphs also include shared groups, or groups that are included in more than one branch of the graph, which are groups of shapes that are included indirectly in the graph, and any change to a shared group affects all references to that group. This allows scene graphs to include the kind of template-based modeling that is common in graphics applications.

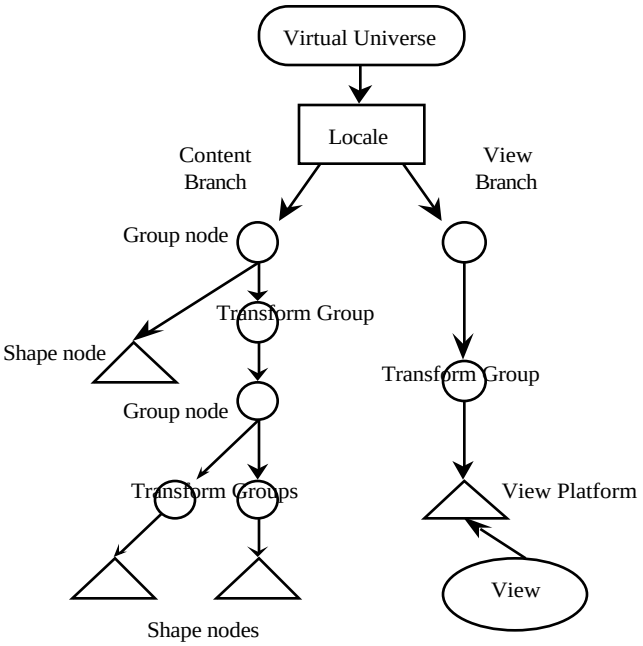


Figure 2.9: the structure of the scene graph as defined in Java3D

The *view branch* of the scene graph includes the specification of the display device, and thus the projection appropriate for that device. It also specifies the user's position and orientation in the scene and includes a wide range of abstractions of the different kinds of viewing devices that can be used by the viewer. It is intended to permit viewing the same scene on any kind of display device, including sophisticated virtual reality devices. This is a much more sophisticated approach than we need for our relatively simple modeling. We will simply consider the eye point as part of the geometry of the scene, so we set the view by including the eye point in the content branch and get the transformation information for the eye point in order to create the view transformations in the view branch.

In addition to the modeling aspect of the scene graph, Java3D also uses it to organize the processing as the scene is rendered. Because the scene graph is processed from the bottom up, the content branch is processed first, followed by the viewing transformation and then the projection transformation. However, the system does not guarantee any particular sequence in processing the node's branches, so it can optimize processing by selecting a processing order for efficiency, or can distribute the computations over a networked or multiprocessor system. Thus the Java3D programmer must be careful to make no assumptions about the state of the system when any shape node is processed. We will not ask the system to process the scene graph itself, however, because we will only use the scene graph to develop our modeling code.

An example of modeling with a scene graph

We will develop a scene graph to design the modeling for an example scene to show how this process can work. To begin, we present an already-completed scene so we can analyze how it can be created, and we will take that analysis and show how the scene graph can give us other ways to present the scene. Consider the scene as shown in Figure 2.10, where a helicopter is flying above a landscape and the scene is viewed from a fixed eye point. (The helicopter is the small green object toward the top of the scene, about 3/4 of the way across the scene toward the right.) This scene contains two principal objects: a helicopter and a ground plane. The helicopter is made up of a body and two rotors, and the ground plane is modeled as a single set of geometry with a texture map. There is some hierarchy to the scene because the helicopter is made up of smaller components, and the scene graph can help us identify this hierarchy so we can work with it in rendering the scene. In addition, the scene contains a light and an eye point, both at fixed locations. The first task in modeling such a scene is now complete: to identify all the parts of the scene, organize the parts into a hierarchical set of objects, and put this set of objects into a viewing context. We must next identify the relationship among the parts of the landscape way so we may create the tree that represents the scene. Here we note the relationship among the ground and the parts of the helicopter. Finally, we must put this information into a graph form.

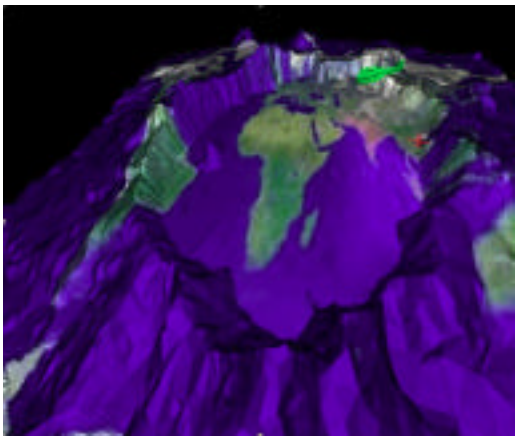


Figure 2.10: a scene that we will describe with a scene graph

The initial analysis of the scene in Figure 2.10, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 2.11. The content branch

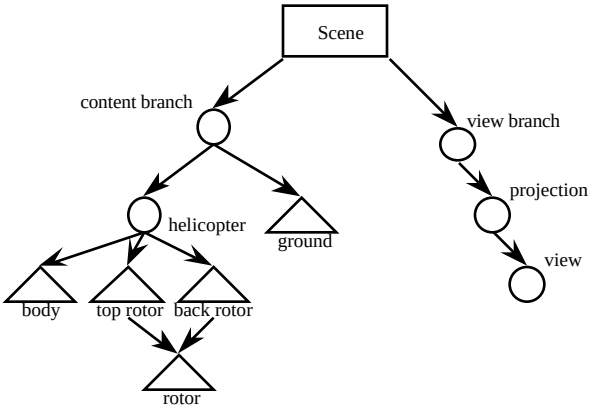


Figure 2.11: a scene graph that organizes the modeling of our simple scene

of this graph captures the organization of the components for the modeling process. This describes how content is assembled to form the image, and the hierarchical structure of this branch helps us organize our modeling components. The view branch of this graph corresponds roughly to projection and viewing. It specifies the projection to be used and develops the projection transformation, as well as the eye position and orientation to develop the viewing transformation.

This initial structure is compatible with the simple OpenGL viewing approach we discussed in the previous chapter and the modeling approach earlier in this chapter, where the view is implemented by using built-in function that sets the viewpoint, and the modeling is built from relatively simple primitives. This approach only takes us so far, however, because it does not integrate the eye into the scene graph. It can be difficult to compute the parameters of the viewing function if the eye point is embedded in the scene and moves with the other content, and later we will address that part of the question of rendering the scene.

While we may have started to define the scene graph, we are not nearly finished. The initial scene graph of Figure 2.11 is incomplete because it merely includes the parts of the scene and describes which parts are associated with what other parts. To expand this first approximation to a more complete graph, we must add several things to the graph:

- the transformation information that describes the relationship among items in a group node, to be applied separately on each branch as indicated,
- the appearance information for each shape node, indicated by the shaded portion of those nodes,
- the light and eye position, either absolute (as used in Figure 2.10 and shown Figure 2.12) or relative to other components of the model (as described later in the chapter), and
- the specification of the projection and view in the view branch.

These are all included in the expanded version of the scene graph with transformations, appearance, eyepoint, and light shown in Figure 2.12.

The content branch of this graph handles all the scene modeling and is very much like the content branch of the scene graph. It includes all the geometry nodes of the graph in Figure 2.11 as well as appearance information; includes explicit transformation nodes to place the geometry into correct sizes, positions, and orientations; includes group nodes to assemble content into logical groupings;

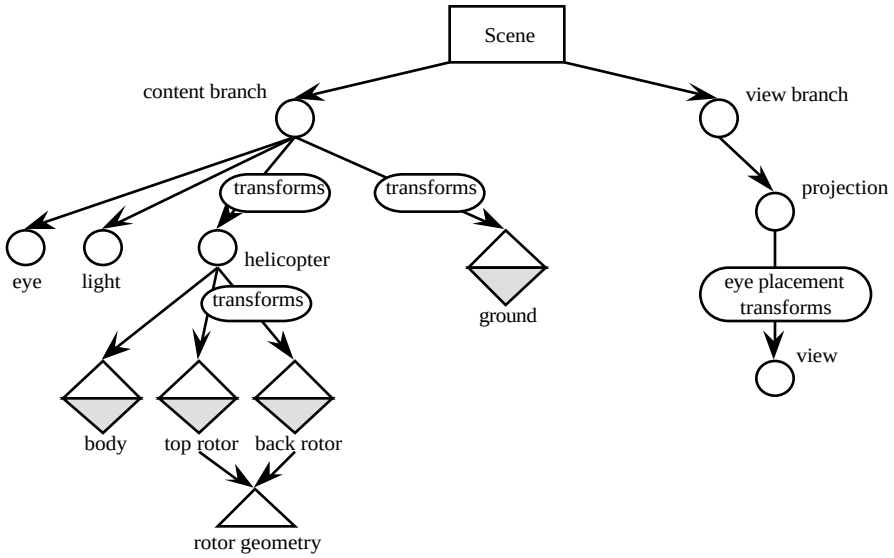


Figure 2.12: the more complete graph including transformations and appearance

and includes lights and the eye point, shown here in fixed positions. It is, of course, quite possible that in some models a light or the eye might be attached to a group instead of being positioned independently, and this can lead to some interesting examples that we describe later. In the example above, it identifies the geometry of the shape nodes such as the rotors or individual trees as shared. This might be implemented, for example, by defining the geometry of the shared shape node in a function and calling that from each of the rotor or tree nodes that uses it.

The view branch of this graph is similar to the view branch of the scene graph but is treated much more simply, containing only projection and view components. The projection component includes the definition of the projection (orthogonal or perspective) for the scene and the definition of the window and viewport for the viewing. The view component includes the information needed to create the viewing transformation, and because the eye point is placed in the content branch, this is simply a copy of the set of transformations that position the eye point in the scene as represented in the content branch.

The appearance part of the shape node is built from color, lighting, shading, texture mapping, and several other kinds of operations. Eventually each vertex of the geometry will have not only geometry, in terms of its coordinates, but also normal components, texture coordinates, and several other properties. Here, however, we are primarily concerned with the geometry content of the shape node; much of the rest of these notes is devoted to building the appearance properties of the shape node, because the appearance content is perhaps the most important part of graphics for building high-quality images.

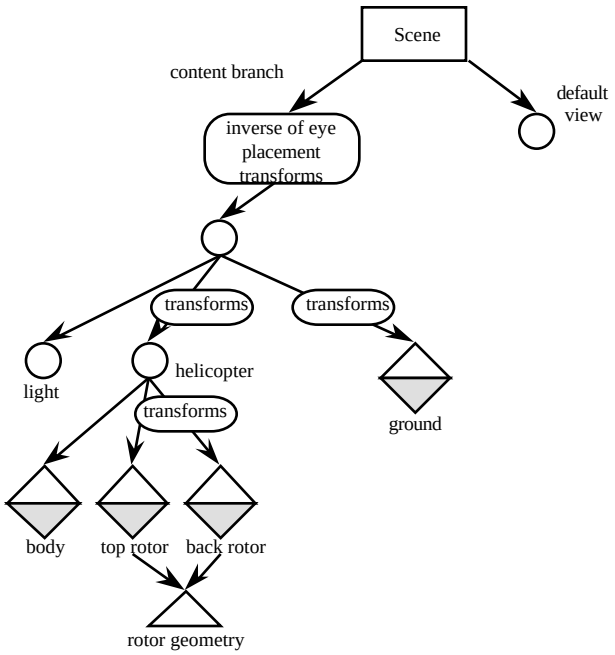


Figure 2.13: the scene graph after integrating the viewing transformation into the content branch

The scene graph for a particular image is not unique because there are many ways to organize a scene. When you have a well-defined set of transformation that place the eye point in a scene, we saw in the earlier chapter on viewing how you can take advantage of that information to organize the scene graph in a way that can define the viewing transformation explicitly and simply use the default view for the scene. As we noted there, the real effect of the viewing transformation is to be the inverse of the transformation that placed the eye. So we can explicitly compute the viewing

transformation as the inverse of the placement transformation ourselves and place that at the top of the scene graph. Thus we can restructure the scene graph of Figure 2.12 as shown in Figure 2.13 so it may take any arbitrary eye position. This will be the key point below as we discuss how to manage the eyepoint when it is a dynamic part of a scene.

It is very important to note that the scene graph need not describe a static geometry. Callbacks for user interaction and other events can affect the graph by controlling parameters of its components, as noted in the re-write guidelines in the next section. This can permit a single graph to describe an animated scene or even alternate views of the scene. The graph may thus be seen as having some components with external controllers, and the controllers are the event callback functions.

We need to extract information on the three key kinds of transformations from this graph in order to create the code that implements our modeling work. The projection transformation is straightforward and is built from the projection information in the view branch, and this is easily managed from tools in the graphics API. Because this is so straightforward, we really do not need to include it in our graph. The viewing transformation is readily created from the transformation information in the view by analyzing the eye placement transformations as we saw above, so it is straightforward to extract this and, more important, to create this transformation from the inversed of the eyepoint transformations. This is discussed in the next section of the chapter. Finally, the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn, and are discussed later in this chapter.

Because all the information we need for both the primitive geometry and all the transformations is held in this simple graph, we will call it the *modeling graph* for our scene. This modeling graph, basically a scene graph without a view branch but with the viewing information organized at the top as the inverse of the eyepoint placement transformations, will be the basis for the coding of our scenes as we describe in the remainder of the chapter.

The viewing transformation

In a scene graph with no view specified, we assume that the default view puts the eye at the origin looking in the negative z-direction with the y-axis upward. If we use a set of transformations to position the eye differently, then the viewing transformation is built by inverting those transformations to restore the eye to the default position. This inversion takes the sequence of transformations that positioned the eye and inverts the primitive transformations in reverse order, so if $T_1T_2T_3\ldots T_K$ is the original transformation sequence, the inverse is $T_K^{u}\ldots T_3^{u}T_2^{u}T_1^{u}$ where the superscript u indicates inversion, or "undo" as we might think of it.

Each of the primitive scaling, rotation, and translation transformations is easily inverted. For the scaling transformation `scale(Sx, Sy, Sz)`, we note that the three scale factors are used to multiply the values of the three coordinates when this is applied. So to invert this transformation, we must divide the values of the coordinates by the same scale factors, getting the inverse as `scale(1/Sx, 1/Sy, 1/Sz)`. Of course, this tells us quickly that the scaling function can only be inverted if none of the scaling factors are zero.

For the rotation transformation `rotate(angle, line)` that rotates space by the value `angle` around the fixed line `line`, we must simply rotate the space by the same angle in the reverse direction. Thus the inverse of the rotation transformation is `rotate(-angle, line)`.

For the translation transformation `translate(Tx, Ty, Tz)` that adds the three translation values to the three coordinates of any point, we must simply subtract those same three translation

values when we invert the transformation. Thus the inverse of the translation transformation is `translate(-Tx, -Ty, -Tz)`.

Putting this together with the information on the order of operations for the inverse of a composite transformation above, we can see that, for example, the inverse of the set of operations (written as if they were in your code)

```
translate(Tx, Ty, Tz)
rotate(angle, line)
scale(Sx, Sy, Sz)
```

is the set of operations

```
scale(1/Sx, 1/Sy, 1/Sz)
rotate(-angle, line)
translate(-Tx, -Ty, -Tz)
```

Now let us apply this process to the viewing transformation. Deriving the eye transformations from the tree is straightforward. Because we suggest that the eye be considered one of the content components of the scene, we can place the eye at any position relative to other components of the scene. When we do so, we can follow the path from the root of the content branch to the eye to obtain the sequence of transformations that lead to the eye point. That sequence of transformations is the eye transformation that we may record in the view branch.

In Figure 2.14 we show the change that results in the view of Figure 2.10 when we define the eye to be immediately behind the helicopter, and in Figure 2.15 we show the change in the scene graph of Figure 2.12 that implements the changed eye point. The eye transform consists of the transforms that places the helicopter in the scene, followed by the transforms that place the eye relative to the helicopter. Then as we noted earlier, the viewing transformation is the inverse of the eye positioning transformation, which in this case is the inverse of the transformations that placed the eye relative to the helicopter, followed by the inverse of the transformations that placed the helicopter in the scene.

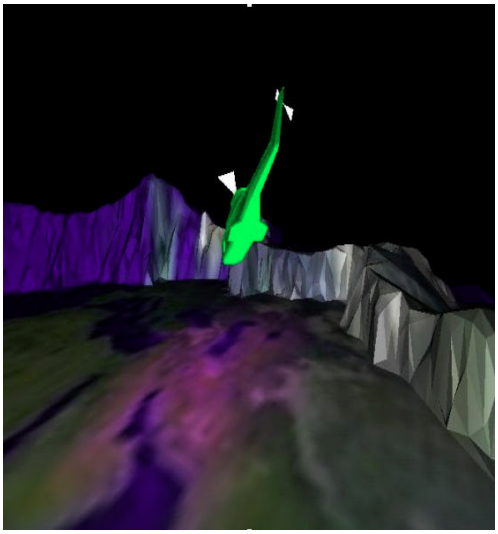


Figure 2.14: the same scene as in Figure 2.10 but with the eye point following directly behind the helicopter

This change in the position of the eye means that the set of transformations that lead to the eye point in the view branch must be changed, but the mechanism of writing the inverse of these transformations before beginning to write the definition of the scene graph still applies; only the actual transformations to be inverted will change. You might, for example, have a menu switch

that specified that the eye was to be at a fixed point or at a point following the helicopter; then the code for inverting the eye position would be a switch statement that implemented the appropriate transformations depending on the menu choice. This is how the scene graph will help you to organize the viewing process that was described in the earlier chapter on viewing.

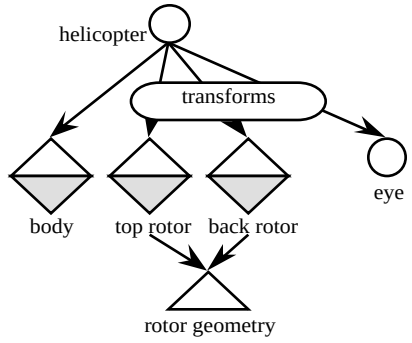


Figure 2.15: the change in the scene graph of Figure 2.10 to implement the view in Figure 2.14

With this scene graph, we can identify the set of transformations $T_a T_b T_c T_d \ldots T_i T_j T_k$ that are applied to put the helicopter in the scene, and the transformations $T_u T_v \ldots T_z$ that place the eye point relative to the helicopter. The implementation of the structure of Figure 2.13, then, is to begin the display code with the standard view, followed by $T_z^{-1} \ldots T_v^{-1} T_u^{-1}$ and then $T_k^{-1} T_j^{-1} T_i^{-1} \ldots T_d^{-1} T_c^{-1} T_b^{-1} T_a^{-1}$, before you begin to write the code for the standard scene as described in Figure 2.16 below.

The process of placing the eye point can readily be generalized. For example, if you should want to design a scene with several possible eye points and allow a user to choose among them, you can design the view branch by creating one view for each eye point and using the set of transformations leading to each eye point as the transformation for the corresponding view. You can then invert each of these sets of transformations to create the viewing transformation for each of the eye points. The choice of eye point will then create a choice of view, and the viewing transformation for that view can then be chosen to implement the user choice.

Because the viewing transformation is performed before the modeling transformations, we see from Figure 2.13 that the inverse transformations for the eye must be applied before the content branch is analyzed and its operations are placed in the code. This means that the display operation must begin with the inverse of the eye placement transformations, which has the effect of moving the eye to the top of the content branch and placing the inverse of the eye path at the front of each set of transformations for each shape node.

The scene graph and depth testing

In almost all of the images we expect to create, we would use the hidden-surface abilities provided by our graphics API. As we described in the last chapter, this will probably use some sort of depth buffer or *Z*-buffer, and the comparisons of depths for hidden surface resolution is done as the parts of the scene are drawn.

However, there may be times when you will want to avoid depth testing and take control of the sequence of drawing your scene components. One such time is described later in the chapter on color and blending, where you need to create a back-to-front drawing sequence in order to simulate transparency with blending operations. In order to do this, you will need to know the depth of each of the pieces of your scene, or the distance of that piece from the eye point. This is easy

enough to do if the scene is totally static, but when you allow pieces to move or the eye to move, it becomes much less simple.

The solution to this problem lies in doing a little extra work as you render your scene. Before you actually draw anything, but after you have updated whatever transformations you will use and whatever choices you will make to draw the current version of the scene, apply the same operations but use a tool called a *projection* that you will find with most graphics APIs. The projection operation allows you to calculate the coordinates of any point in your model space when it is transformed by the viewing and projection transformations into a point in 3D eye space. The depth of that point, then, is simply the *Z*-coordinate of the projected value. You can draw the entire scene, then, using the projection operation instead of the rendering operation, get the depth values for each piece of the scene, and use the depth values to determine the order in which you will draw the parts. The scene graph will help you make sure you have the right transformations when you project each of the parts, ensuring that you have the right depth values.

Using the modeling graph for coding

Because the modeling graph as we defined it above is intended as a learning tool and not a production tool, we will resist the temptation to formalize its definition beyond the terms we used there:

- shape node containing two components
  - geometry content
  - appearance content
- transformation node
- group node
- projection node
- view node

Because we do not want to look at any kind of automatic parsing of the modeling graph to create the scene, we will merely use the graph to help organize the structure and the relationships in the model to help you organize your code to implement your simple or hierarchical modeling. This is quite straightforward and is described in detail below.

Once you know how to organize all the components of the model in the modeling graph, you next need to write the code to implement the model. This turns out to be straightforward, and you can use a simple set of re-write guidelines that allow you to rewrite the graph as code. In this set of rules, we assume that transformations are applied in the reverse of the order they are declared, as they are in OpenGL, for example. This is consistent with your experience with tree handling in your programming courses, because you have usually discussed an expression tree which is parsed in leaf-first order. It is also consistent with the Java3D convention that transforms that are " closer" to the geometry (nested more deeply in the scene graph) are applied first.

The informal rewrite guidelines are as follows, including the rewrites for the view branch as well as the content branch:

- Nodes in the view branch involve only the window, viewport, projection, and viewing transformations. The window, viewport, and projection are handled by simple functions in the API and should be at the top of the display function.
- The viewing transformation is built from the transformations of the eye point within the content branch by copying those transformations and undoing them to place the eye effectively at the top of the content branch. This sequence should be next in the display function.
- The content branch of the modeling graph is usually maintained fully within the display function, but parts of it may be handled by other functions called from within the display, depending on the design of the scene. A function that defines the geometry of an object may be used by one or more shape nodes. The modeling may be affected by parameters set

by event callbacks, including selections of the eye point, lights, or objects to be displayed in the view.

- Group nodes are points where several elements are assembled into a single object. Each separate object is a different branch from the group node. Before writing the code for a branch that includes a transformation group, the student should push the modelview matrix; when returning from the branch, the student should pop the modelview matrix.
- Transformation nodes include the familiar translations, rotations, and scaling that are used in the normal ways, including any transformations that are part of animation or user control. In writing code from the modeling graph, students can write the transformations in the same sequence as they appear in the tree, because the bottom-up nature of the design work corresponds to the last-defined, first-used order of transformations.
- As you work your way through the modeling graph, you will need to save the state of the modeling transformation before you go down any branch of the graph from which you will need to return as the graph is traversed. Because of the simple nature of each transformation primitive, it is straightforward to undo each as needed to create the viewing transformation. This can be handled through a transformation stack that allows you to save the current transformation by pushing it onto the stack, and then restore that transformation by popping the stack.
- Shape nodes involve both geometry and appearance, and the appearance must be done first because the current appearance is applied when geometry is defined.
  - An appearance node can contain texture, color, blending, or material information that will make control how the geometry is rendered and thus how it will appear in the scene.
  - A geometry node will contain vertex information, normal information, and geometry structure information such as strip or fan organization.
- Most of the nodes in the content branch can be affected by any interaction or other event-driven activity. This can be done by defining the content by parameters that are modified by the event callbacks. These parameters can control location (by parametrizing rotations or translations), size (by parametrizing scaling), appearance (by parametrizing appearance details), or even content (by parametrizing switches in the group nodes).

We will give some examples of writing graphics code from a modeling graph in the sections below, so look for these principles as they are applied there.

In the example for Figure 2.14 above, we would use the tree to write code as shown in skeleton form in Figure 2.16. Most of the details, such as the inversion of the eye placement transformation, the parameters for the modeling transformations, and the details of the appearance of individual objects, have been omitted, but we have used indentation to show the pushing and popping of the modeling transformation stack so we can see the operations between these pairs easily. This is straightforward to understand and to organize.

Animation is simple to add to this example. The rotors can be animated by adding an extra rotation in their definition plane immediately after they are scaled and before the transformations that orient them to be placed on the helicopter body, and by updating angle of the extra rotation each time the idle event callback executes. The helicopter's behavior itself can be animated by updating the parameters of transformations that are used to position it, again with the updates coming from the idle callback. The helicopter's behavior may be controlled by the user if the positioning transformation parameters are updated by callbacks of user interaction events. So there are ample opportunities to have this graph represent a dynamic environment and to include the dynamics in creating the model from the beginning.

Other variations in this scene could by developed by changing the position of the light from its current absolute position to a position relative to the ground (by placing the light as a part of the branch group containing the ground) or to a position relative to the helicopter (by placing the light as a part of the branch group containing the helicopter). The eye point could similarly be placed

relative to another part of the scene, or either or both could be placed with transformations that are controlled by user interaction with the interaction event callbacks setting the transformation parameters.

```
display()
   set the viewport and projection as needed
   initialize modelview matrix to identity
   define viewing transformation
      invert the transformations that set the eye location
   set eye through gluLookAt with default values
   define light position         // note absolute location
   push the transformation stack       // ground
      translate
      rotate
      scale
      define ground appearance (texture)
      draw ground
   pop the transformation stack
   push the transformation stack       // helicopter
      translate
      rotate
      scale
      push the transformation stack    // top rotor
         translate
         rotate
         scale
         define top rotor appearance
         draw top rotor
      pop the transformation stack
      push the transformation stack    // back rotor
         translate
         rotate
         scale
         define back rotor appearance
         draw back rotor
      pop the transformation stack
      // assume no transformation for the body
      define body appearance
      draw body
   pop the transformation stack
   swap buffers
```

Figure 2.16: code sketch to implement the modeling in Figure 2.15

We emphasize that you should include appearance content with each shape node. Many of the appearance parameters involve a saved state in APIs such as OpenGL and so parameters set for one shape will be retained unless they are re-set for the new shape. It is possible to design your scene so that shared appearances will be generated consecutively in order to increase the efficiency of rendering the scene, but this is a specialized organization that is inconsistent with more advanced APIs such as Java3D. Thus it is very important to re-set the appearance with each shape to avoid accidentally retaining an appearance that you do not want for objects presented in later parts of your scene.

Example

We want to further emphasize the transformation behavior in writing the code for a model from the modeling graph by considering another small example. Let us consider a very simple rabbit's head as shown in Figure 2.17. This would have a large ellipsoidal head, two small spherical eyes, and two middle-sized ellipsoidal ears. So we will use the ellipsoid (actually a scaled sphere, as we saw earlier) as our basic part and will put it in various places with various orientations as needed.

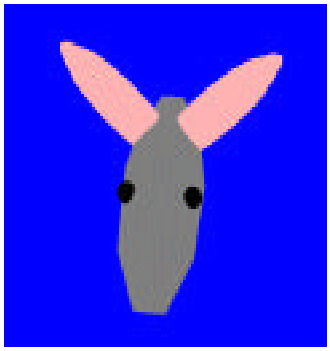


Figure 2.17: the rabbit's head

The modeling graph for the rabbit's head is shown in Figure 2.18. This figure includes all the transformations needed to assemble the various parts (eyes, ears, main part) into a unit. The fundamental geometry for all these parts is the sphere, as we suggested above. Note that the transformations for the left and right ears include rotations; these can easily be designed to use a parameter for the angle of the rotation so that you could make the rabbit's ears wiggle back and forth.

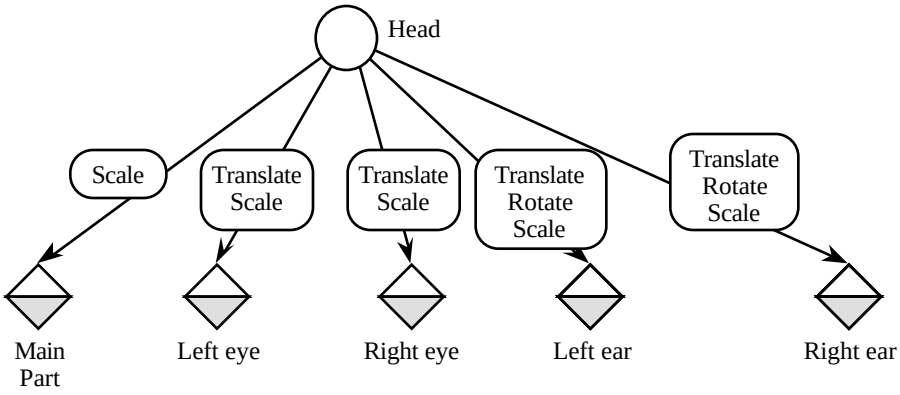


Figure 2.18: the modeling graph for the rabbit's head

To write the code to implement the modeling graph for the rabbit's head, then, we would apply the following sequence of actions on the modeling transformation stack:

- push the modeling transformation stack
- apply the transformations to create the head, and define the head:
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left eye relative to the head, and define the eye:
  translate

scale
draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right eye relative to the head, and define the eye:
  translate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack

You should trace this sequence of operations carefully and watch how the head is drawn. Note that if you were to want to put the rabbit's head on a body, you could treat this whole set of operations as a single function `rabbitHead()` that is called between operations push and pop the transformation stack, with the code to place the head and move it around lying above the function call. This is the fundamental principle of hierarchical modeling — to create objects that are built of other objects, finally reducing the model to simple geometry at the lowest level. In the case of the modeling graph, that lowest level is the leaves of the tree, in the shape nodes.

The transformation stack we have used informally above is a very important consideration in using a scene graph structure. It may be provided by your graphics API or it may be something you need to create yourself; even if it provided by the API, there may be limits on the depth of the stack that will be inadequate for some projects and you may need to create your own. We will discuss this in terms of the OpenGL API later in this chapter.

Using standard objects to create more complex scenes

The example of transformation stacks is, in fact, a larger example—an example of using standard objects to define a larger object. In a program that defined a scene that needed rabbits, we would create the rabbit head with a function `rabbitHead()` that has the content of the code we used (and that is given below) and would apply whatever transformations would be needed to place a rabbit head properly on each rabbit body. The rabbits themselves could be part of a larger scene, and you could proceed in this way to create however complex a scene as you wish.

Questions

1. We know that we can model any polyhedron with triangles, but why can you model a sphere with triangle fans for the polar caps and quad strips for the rest of the object?

2. Put yourself in a familiar environment, but imagine the environment simplified so that it is made up of only boxes, cylinders, and other very basic shapes. Imagine further that your

environment has only one door and that everything in the room has to come in that door. Write out the sequence of transformations that would have to be done to place everything in its place in your environment. Now imagine that each of these basic shapes starts out as a standard shape: a unit cube, a cylinder with diameter one and height one, and the like; write out the sequence of transformations that would have to be done to make each object from these basic objects. Finally, if the door would only admit basic objects, put together these two processes to write out the full transformations to create the objects and place them in the space.

3. Now take the environment above and write a scene graph that describes the whole scene, using the basic shapes and transformations you identified in the previous question. Also place your eye in the scene graph starting with a standard view of you standing in the doorway and facing directly into the room. Now imagine that on a table in the space there is a figure of a ballerina spinning around and around, and identify the way the transformations in the scene graph would handle this moving object.

Exercises

4. Calculate the coordinates of the vertices of the simpler regular polyhedra: the cube, the tetrahedron, and the octagon. For the octagon and tetrahedron, try using spherical coordinates and converting them to rectangular coordinates; see the chapter on mathematics for modeling.

5. Verify that for any x, y, z, and w, the point $(x/w,\ y/w,\ z/w,\ 1)$ is the intersection of the line segment from $(x,\ y,\ z,\ w)$ to $(0,\ 0,\ 0,\ 0)$, and the hyperplane $\{\ (a,\ b,\ c,\ 1)\ |$ arbitrary $a,\ b,\ c\ \}$.

6. Show how you can define a cube as six quads. Show how you can refine that definition to write a cube as two quad strips. Can you write a cube as one quad strip?

7. Show how you can write any polygon, convex or not, as a set of triangles. Show further how you can write any *convex* polygon as a triangle fan.

8. Define a polygon in 2D space that is reasonably large and having a side that is not parallel to one of the axes. Find a unit square in the 2D space that intersects that side, and calculate the proportion of the polygon that lies within the unit square. If the square represents a pixel, draw conclusions about what proportion of the pixel's color should come from the polygon and what proportion from the background.

9. The code for the normals to a quad on page 2.9 is not accurate because it uses the normal at a vertex instead of the normal in the middle of the quad. How should you calculate the normal so that it is the face normal and not a vertex normal?

10. Make a basic object with no symmetries, and apply simple rotation, simple translation, and simple scaling to it; see what you get. Then apply second and third transformations after you have applied the first transformation and again see what you get. Show why the order of the transformations matters by applying the same transformations in different order and seeing the results.

11. Scene graphs are basically trees, though different branches may share common shape objects. As trees, they can be traversed in any way that is convenient. Show how you might choose the way you would traverse a scene graph in order to draw back-to-front if you knew the depth of each object in the tree.

12. Add a mouth and tongue to the rabbit's head, and modify the scene graph for the rabbit's head to have the rabbit stick out its tongue and wiggle it around.

13. Define a scene graph for a carousel, or merry-go-round. This object has a circular disk as its floor, a cone as its roof, a collection of posts that connect the floor to the roof, and a collection of animals in a circle just inside the outside diameter of the floor, each parallel to the tangent to the floor at the point on the edge nearest the animal. The animals will go up and down in a periodic way as the carousel goes around. You may assume that each animal is a primitive and not try to model it, but you should carefully define all the transformations that build the carousel and place the animals.

Experiments

14. Get some of the models from the `avalon.viewpoint.com` site and examine the model file to see how you could present the model as a sequence of triangles or other graphics primitives.

15. Write the code for the scene graph of the familiar space from question 3, including the code that manages the inverse transformations for the eye point. Now identify a simple path for the eye, created by parametrizing some of the transformations that place the eye, and create an animation of the scene as it would be seen from the moving eye point.

Chapter 3: Implementing Modeling in OpenGL

This chapter discusses the way OpenGL implements the general modeling discussion of the last chapter. This includes specifying geometry, specifying points for that geometry in model space, specifying normals for these vertices, and specifying and managing transformations that move these objects from model space into the world coordinate system. It also includes the set of operations that implement polygons, including those that provide the geometry compression that was described in the previous chapter. Finally, it discusses some pre-built models that are provided by the OpenGL and GLUT environments to help you create your scenes more easily.

The OpenGL model for specifying geometry

In defining your model for your program, you will use a single function to specify the geometry of your model to OpenGL. This function specifies that geometry is to follow, and its parameter defines the way in which that geometry is to be interpreted for display:

```
glBegin(mode);
// vertex list: point data to create a primitive object in
// the drawing mode you have indicated
// normals may also be specified here
glEnd();
```

The vertex list is interpreted as needed for each drawing mode, and both the drawing modes and the interpretation of the vertex list are described in the discussions below. This pattern of `glBegin(mode) - vertex list - glEnd` uses different values of the `mode` to establish the way the vertex list is used in creating the image. Because you may use a number of different kinds of components in an image, you may use this pattern several times for different kinds of drawing. We will see a number of examples of this pattern in this module.

In OpenGL, point (or vertex) information is presented to the computer through a set of functions that go under the general name of `glVertex*(…)`. These functions enter the numeric value of the vertex coordinates into the OpenGL pipeline for the processing to convert them into image information. We say that `glVertex*(…)` is a *set* of functions because there are many functions that differ only in the way they define their vertex coordinate data. You may want or need to specify your coordinate data in any standard numeric type, and these functions allow the system to respond to your needs.

- If you want to specify your vertex data as three separate real numbers, or floats (we'll use the variable names `x`, `y`, and `z`, though they could also be float constants), you can use `glVertex3f(x,y,z)`. Here the character `f` in the name indicates that the arguments are floating-point; we will see below that other kinds of data formats may also be specified for vertices.
- If you want to define your coordinate data in an array, you could declare your data in a form such as `glFloat x[3]` and then use `glVertex3fv(x)` to specify the vertex. Adding the letter `v` to the function name specifies that the data is in vector form (actually a pointer to the memory that contains the data, but an array's name is really such a pointer). Other dimensions besides 3 are also possible, as noted below.

Additional versions of the functions allow you to specify the coordinates of your point in two dimensions (`glVertex2*`); in three dimensions specified as integers (`glVertex3i`), doubles (`glVertex3d`), or shorts (`glVertex3s`); or as four-dimensional points (`glVertex4*`). The four-dimensional version uses homogeneous coordinates, as described earlier in this chapter. You will see some of these used in the code examples later in this chapter.

One of the most important things to realize about modeling in OpenGL is that you can call your own functions between a `glBegin(mode)` and `glEnd()` pair to determine vertices for your vertex list. Any vertices these functions define by making a `glVertex*(…)` function call will be

added to the vertex list for this drawing mode. This allows you to do whatever computation you need to calculate vertex coordinates instead of creating them by hand, saving yourself significant effort and possibly allowing you to create images that you could not generate by hand. For example, you may include various kind of loops to calculate a sequence of vertices, or you may include logic to decide which vertices to generate. An example of this way to generate vertices is given among the first of the code examples toward the end of this module.

Another important point about modeling is that a great deal of other information can go between a `glBegin(mode)` and `glEnd()` pair. We will see the importance of including information about vertex normals in the chapters on lighting and shading, and of including information on texture coordinates in the chapter on texture mapping. So this simple construct can be used to do much more than just specify vertices. Although you may carry out whatever processing you need within the `glBegin(mode)` and `glEnd()` pair, there are a limited number of OpenGL operations that are permitted here. In general, the available OpenGL operations here are `glVertex`, `glColor`, `glNormal`, `glTexCoord`, `glEvalCoord`, `glEvalPoint`, `glMaterial`, `glCallList`, and `glCallLists`, although this is not a complete list. Your OpenGL manual will give you additional information if needed.

Point and points mode

The mode for drawing points with the `glBegin` function is named `GL_POINTS`, and any vertex data between `glBegin` and `glEnd` is interpreted as the coordinates of a point we wish to draw. If we want to draw only one point, we provide only one vertex between `glBegin` and `glEnd`; if we want to draw more points, we provide more vertices between them. If you use points and want to make each point more visible, the function `glPointSize(float size)` allows you to set the size of each point, where `size` is any nonnegative real value and the default size is 1.0.

The code below draws a sequence of points in a straight line. This code takes advantage of fact that we can use ordinary programming processes to define our models, showing we need not hand-calculate points when we can determine them by an algorithmic approach. We specify the vertices of a point through a function `pointAt()` that calculates the coordinates and calls the `glVertex*()` function itself, and then we call that function within the `glBegin/glEnd` pair. The function calculates points on a spiral along the z-axis with x- and y-coordinates determined by functions of the parameter `t` that drives the entire spiral.

```
void pointAt(int i) {
   glVertex3f(fx(t)*cos(g(t)),fy(t)*sin(g(t)),0.2*(float)(5-i));
}

void pointSet( void ) {
   int i;

   glBegin(GL_POINTS);
      for ( i=0; i<10; i++ )
      pointAt(i);
   glEnd();
}
```

Some functions that drive the x- and y-coordinates may be familiar to you through studies of functions of polar coordinates in previous mathematics classes, and you are encouraged to try out some possibilities on your own.

Line segments

To draw line segments, we use the `GL_LINES` mode for `glBegin/glEnd`. For each segment we wish to draw, we define the vertices for the two endpoints of the segment. Thus between `glBegin` and `glEnd` each pair of vertices in the vertex list defines a separate line segment.

Line strips

Connected lines are called *line strips* in OpenGL, and you can specify them by using the mode `GL_LINE_STRIP` for `glBegin/glEnd`. The vertex list defines the line segments as noted in the general discussion of connected lines above, so if you have N vertices, you will have N-1 line segments. With either line segments or connected lines, we can set the line width to emphasize (or de-emphasize) a line. Heavier line widths tend to attract more attention and give more emphasis than lighter line widths. The line width is set with the `glLineWidth(float width)` function. The default value of `width` is 1.0 but any nonnegative width can be used.

As an example of a line strip, let's consider a parametric curve. Such curves in 3-space are often interesting objects of study. The code below define a rough spiral in 3-space that is a good (though simple) example of using a single parameter to define points on a parametric curve so it can be drawn for study.

```
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=10; i++ )
      glVertex3f(2.0*cos(3.14159*(float)i/5.0),
         2.0*sin(3.14159*(float)i/5.0),0.5*(float)(i-5));
glEnd();
```

This can be made much more sophisticated by increasing the number of line segments, and the code can be cleaned up a bit as described in the code fragment below. Simple experiments with the `step` and `zstep` variables will let you create other versions of the spiral as experiments.

```
#define PI 3.14159
#define N 100
step = 2.0*PI/(float)N;
zstep = 2.0/(float)N;
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),2.0*cos(step*(float)i),
      -1.0+zstep*(float)i);
glEnd();
```

If this spiral is presented in a program that includes some simple rotations, you can see the spiral from many points in 3-space. Among the things you will be able to see are the simple sine and cosine curves, as well as one period of the generic shifted sine curve.

Line loops

A line loop is just like a line strip except that an additional line segment is drawn from the last vertex in the list to the first vertex in the list, creating a closed loop. There is little more to be said about line loops; they are specified by using the mode `GL_LINE_LOOP`.

Triangle

To draw unconnected triangles, you use `glBegin/glEnd` with the mode `GL_TRIANGLES`. This is treated exactly as discussed in the previous chapter and produces a collection of triangles, one for each three vertices specified.

Sequence of triangles

OpenGL provides both of the standard geometry-compression techniques to assemble sequences of triangles: triangle strips and triangle fans. Each has its own mode for `glBegin/glEnd`: `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN` respectively. These behave exactly as described in the general section above.

Because there are two different modes for drawing sequences of triangles, we'll consider two examples in this section. The first is a triangle fan, used to define an object whose vertices can be seen as radiating from a central point. An example of this might be the top and bottom of a sphere, where a triangle fan can be created whose first point is the north or south pole of the sphere. The second is a triangle strip, which is often used to define very general kinds of surfaces, because most surfaces seem to have the kind of curvature that keeps rectangles of points on the surface from being planar. In this case, triangle strips are much better than quad strips as a basis for creating curved surfaces that will show their surface properties when lighted.

The triangle fan (that defines a cone, in this case) is organized with its vertex at point `(0.0,1.0,0.0)` and with a circular base of radius 0.5 in the XZ-plane. Thus the cone is oriented towards the y-direction and is centered on the y-axis. This provides a surface with unit diameter and height, as shown in Figure 3.1. When the cone is used in creating a scene, it can easily be defined to have whatever size, orientation, and location you need by applying appropriate modeling transformations in an appropriate sequence. Here we have also added normals and flat shading to emphasize the geometry of the triangle fan, although the code does not reflect this.

```
glBegin(GL_TRIANGLE_FAN);
   glVertex3f(0., 1.0, 0.);  // the point of the cone
   for (i=0; i < numStrips; i++) {
      angle = 2. * (float)i * PI / (float)numStrips;
      glVertex3f(0.5*cos(angle), 0.0, 0.5*sin(angle));
      // code to calculate normals would go here
   }
glEnd();
```

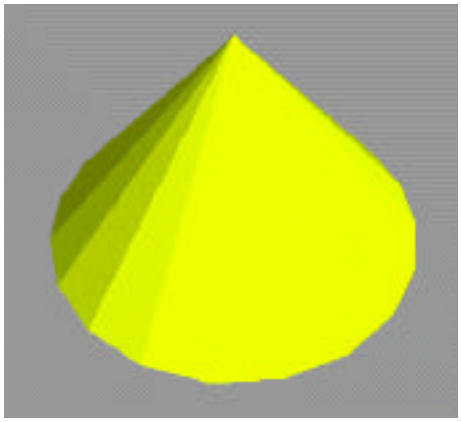


Figure 3.1: the cone produced by the triangle fan

The triangle strip example is based on an example of a function surface defined on a grid. Here we describe a function whose domain is in the X-Z plane and whose values are shown as the Y-value of each vertex. The grid points in the X-Z domain are given by functions `XX(i)` and `ZZ(j)`, and

the values of the function are held in an array, with `vertices[i][j]` giving the value of the function at the grid point `(XX(i),ZZ(j))` as defined in the short example code fragment below.

```
for ( i=0; i<XSIZE; i++ )
   for ( j=0; j<ZSIZE; j++ ) {
      x = XX(i);
      z = ZZ(j);
      vertices[i][j] = (x*x+2.0*z*z)/exp(x*x+2.0*z*z+t);
   }
```

The surface rendering can then be organized as a nested loop, where each iteration of the loop draws a triangle strip that presents one section of the surface. Each section is one unit in the X-direction that extends across the domain in the Z-direction. The code for such a strip is shown below, and the resulting surface is shown in Figure 3.2. Again, the code that calculates the normals is omitted; this example is discussed further and the normals are developed in the later chapter on shading. This kind of surface is explored in more detail in the chapters on scientific applications of graphics.

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      glBegin(GL_TRIANGLE_STRIP);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
      glEnd();
   }
```

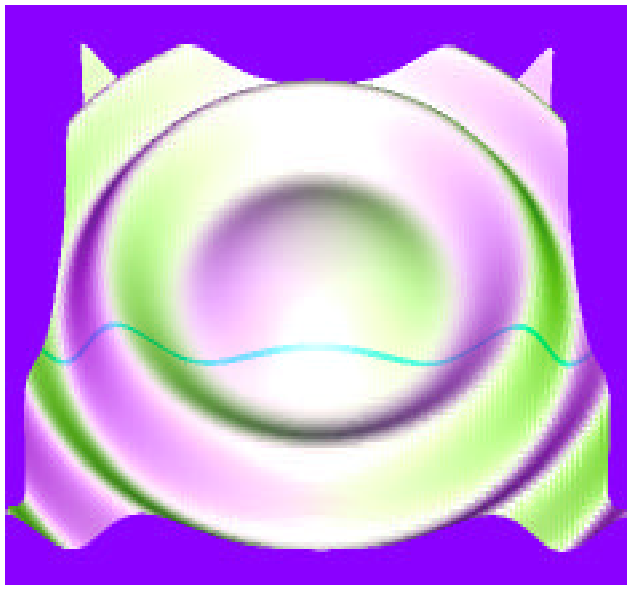


Figure 3.2: the full surface created by triangle strips, with a single strip highlighted in cyan

This example is a white surface lighted by three lights of different colors, a technique we describe in the chapter on lighting. This surface example is also briefly revisited in the quads discussion below. Note that the sequence of points here is slightly different here than it is in the example below because of the way quads are specified. In this example instead of one quad, we will have two triangles—and if you rework the example below to use quad strips instead of simple quads to

display the mathematical surface, it is simple to make the change noted here and do the surface with extended triangle strips.

Quads

To create a set of one or more distinct quads you use `glBegin/glEnd` with the `GL_QUADS` mode. As described earlier, this will take four vertices for each quad. An example of an object based on quadrilaterals would be the function surface discussed in the triangle strip above. For quads, the code for the surface looks like this:

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      // quad sequence: points (i,j),(i+1,j),(i+1,j+1),(i,j+1)
      glBegin(GL_QUADS);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
      glEnd();
   }
```

Note that neither this surface nor the one composed from triangles is going to look very good yet because it does not yet contain any lighting or color information. These will be added in later chapters as this concept of function surfaces is re-visited when we discuss lighting and color.

Quad strips

To create a sequence of quads, the mode for `glBegin/glEnd` is `GL_QUAD_STRIP`. This operates in the way we described at the beginning of the chapter, and as we noted there, the order in which the vertices are presented is different from that in the GL_QUADS mode. Be careful of this when you define your geometry or you may get a very unusual kind of display!

In a fairly common application, we can create long, narrow tubes with square cross-section. This can be used as the basis for drawing 3-D coordinate axes or for any other application where you might want to have, say, a beam in a structure. The quad strip defined below creates the tube oriented along the Z-axis with the cross-section centered on that axis. The dimensions given make a unit tube—a tube that is one unit in each dimension, making it actually a cube. These dimensions will make it easy to scale to fit any particular use.

```
#define RAD 0.5
#define LEN 1.0
glBegin(GL_QUAD_STRIP);
   glVertex3f( RAD, RAD, LEN ); // start of first side
   glVertex3f( RAD, RAD, 0.0 );
   glVertex3f(-RAD, RAD, LEN );
   glVertex3f(-RAD, RAD, 0.0 );
   glVertex3f(-RAD,-RAD, LEN ); // start of second side
   glVertex3f(-RAD,-RAD, 0.0 );
   glVertex3f( RAD,-RAD, LEN ); // start of third side
   glVertex3f( RAD,-RAD, 0.0 );
   glVertex3f( RAD, RAD, LEN ); // start of fourth side
   glVertex3f( RAD, RAD, 0.0 );
glEnd();
```

You can also get the same object by using the GLUT cube that is discussed below and applying appropriate transformations to center it on the Z-axis.

General polygon

The GL_POLYGON mode for glBegin/glEnd is used to allow you to display a single convex polygon. The vertices in the vertex list are taken as the vertices of the polygon in sequence order, and we remind you that the polygon needs to be convex. It is not possible to display more than one polygon with this operation because the function will always assume that whatever points it receives go in the same polygon.

Probably the simplest kind of multi-sided convex polygon is the regular N-gon, an N-sided figure with all edges of equal length and all interior angles between edges of equal size. This is simply created (in this case, for N=7), again using trigonometric functions to determine the vertices.

```
#define PI 3.14159
#define N 7
step = 2.0*PI/(float)N;
glBegin(GL_POLYGON);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),
                 2.0*cos(step*(float)i),0.0);
glEnd();
```

Note that this polygon lives in the XY-plane; all the Z-values are zero. This polygon is also in the default color (white) because we have not specified the color to be anything else. This is an example of a "canonical" object—an object defined not primarily for its own sake, but as a template that can be used as the basis of building another object as noted later, when transformations and object color are available. An interesting application of regular polygons is to create regular polyhedra—closed solids whose faces are all regular N-gons. These polyhedra are created by writing a function to draw a simple N-gon and then using transformations to place these properly in 3-space to be the boundaries of the polyhedron.

Antialiasing

As we saw in the previous chapter, geometry drawn with antialiasing is smoother and less "jaggy" than geometry drawn in the usual "all-or-nothing" pixel mode. OpenGL provides some capabilities for antialiasing by allowing you to enable point smoothing, line smoothing, and/or polygon smoothing. These are straightforward to specify, but they operate with color blending and there may be some issues around the order in which you draw your geometry. OpenGL calculates the coverage factor for antialiasing based on computing the proportion of a pixel that is covered by the geometry being presented, as described in the previous chapter. For more on RGBA blending and the order in which drawing is done, see the later chapter on color and blending.

To use the built-in OpenGL antialiasing, choose the various kinds of point, line, or polygon smoothing with the glEnable(...) function. Each implementation of OpenGL will define a default behavior for smoothing, so you may want to override that default by defining your choice with the glHint(...) function. The appropriate pairs of enable/hint are shown here:

```
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POLYGON_SMOOTH);
glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);
```

There is a more sophisticated kind of polygon smoothing involving entire scene antialiasing, done by drawing the scene into the accumulation buffer with slight offsets so that boundary pixels will be chosen differently for each version. This is a time-consuming process and is generally considered a more advanced use of OpenGL than we are assuming in this book. We do discuss the accumulation buffer in a later chapter when we discuss motion blur, but we will not go into more detail here.

The cube we will use in many examples

Because a cube is made up of six square faces, it is very tempting to try to make the cube from a single quad strip. Looking at the geometry, though, it is impossible to make a single quad strip go around the cube; in fact, the largest quad strip you can create from a cube's faces has only four quads. It is possible to create two quad strips of three faces each for the cube (think of how a baseball is stitched together), but here we will only use a set of six quads whose vertices are the eight vertex points of the cube. Below we repeat the declarations of the vertices, normals, edges, and faces of the cube from the previous chapter. We will use the `glVertex3fv(…)` vertex specification function within the specification of the quads for the faces.

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];  // each face of a cube has four edges

   point3 vertices[8] = {{-1.0, -1.0, -1.0},
                         {-1.0, -1.0,  1.0},
                         {-1.0,  1.0, -1.0},
                         {-1.0,  1.0,  1.0},
                         { 1.0, -1.0, -1.0},
                         { 1.0, -1.0,  1.0},
                         { 1.0,  1.0, -1.0},
                         { 1.0,  1.0,  1.0} };

   point3 normals[6]  = {{ 0.0, 0.0, 1.0},
                         {-1.0, 0.0, 0.0},
                         { 0.0, 0.0,-1.0},
                         { 1.0, 0.0, 0.0},
                         { 0.0,-1.0, 0.0},
                         { 0.0, 1.0, 0.0} };

   edge   edges[24]   = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                         { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                         { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                         { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                         { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                         { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

   face   cube[6]     = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                         { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                         {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

As we said before, drawing the cube proceeds by working our way through the face list and determining the actual points that make up the cube. We will expand the function we gave earlier to write the actual OpenGL code below. Each face is presented individually in a loop within the `glBegin-glEnd` pair, and with each face we include the normal for that face. Note that only the first vertex of the first edge of each face is identified, because the GL_QUADS drawing mode

takes each set of four vertices as the vertices of a quad; it is not necessary to close the quad by including the first point twice.

```
void cube(void) {
   int face, edge;
   glBegin(GL_QUADS);
      for (face = 0; face < 6; face++) {
         glNormal3fv(normals[face];
         for (edge = 0; edge < 4; edge++)
            glVertex3fv(vertices[edges[cube[face][edge]][0]]);
      }
   glEnd();
}
```

This cube is shown in Figure 3.3, presented through the six steps of adding individual faces (the faces are colored in the typical RGBCMY sequence so you may see each added in turn). This approach to defining the geometry is actually a fairly elegant way to define a cube, and takes very little coding to carry out. However, this is not the only approach we could take to defining a cube. Because the cube is a regular polyhedron with six faces that are squares, it is possible to define the cube by defining a standard square and then using transformations to create the faces from this master square. Carrying this out is left as an exercise for the student.

This approach to modeling an object includes the important feature of specifying the normals (the vectors perpendicular to each face) for the object. We will see in the chapters on lighting and shading that in order to get the added realism of lighting on an object, we must provide information on the object's normals, and it was straightforward to define an array that contains a normal for each face. Another approach would be to provide an array that contains a normal for each vertex if you would want smooth shading for your model; see the chapter on shading for more details. We will not pursue these ideas here, but you should be thinking about them when you consider modeling issues with lighting.

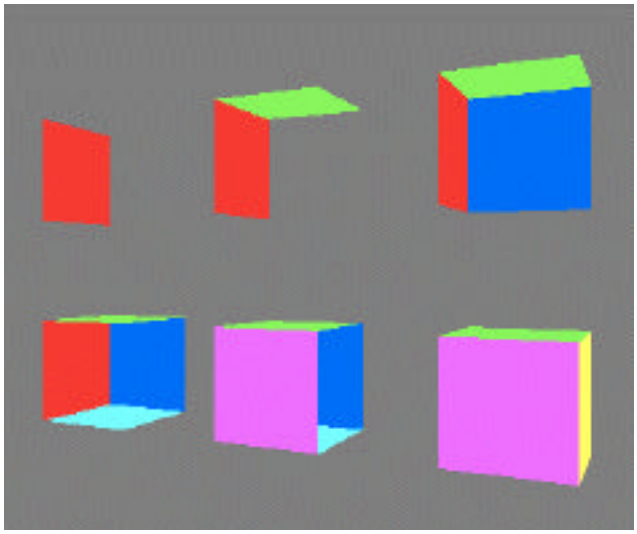


Figure 3.3: the cube as a sequence of quads

Additional objects with the OpenGL toolkits

Modeling with polygons alone would require you to write many standard graphics elements that are so common, any reasonable graphics system should include them. OpenGL includes the

OpenGL Utility Library, GLU, with many useful functions, and most releases of OpenGL also include the OpenGL Utility Toolkit, GLUT. We saw in the first chapter that GLUT includes window management functions, and both GLU and GLUT include a number of built-in graphical elements that you can use. This chapter describes a number of these elements.

The objects that these toolkits provide are defined with several parameters that define the details, such as the resolution in each dimension of the object with which the object is to be presented. Many of these details are specific to the particular object and will be described in more detail when we describe each of these.

GLU quadric objects

The GLU toolkit provides several general quadric objects, which are objects defined by quadric equations (polynomial equations in three variables with degree no higher than two in any term), including Spheres (`gluSphere`), cylinders (`gluCylinder`), and disks (`gluDisk`). Each GLU primitive is declared as a GLUquadric and is allocated with the function

```
GLUquadric* gluNewQuadric( void )
```

Each quadric object is a surface of revolution around the z-axis. Each is modeled in terms of subdivisions around the z-axis, called slices, and subdivisions along the z-axis, called stacks. Figure 3.4 shows an example of a typical pre-built quadric object, a GLUT wireframe sphere, modeled with a small number of slices and stacks so you can see the basis of this definition.

The GLU quadrics are very useful in many modeling circumstances because you can use scaling and other transformations to create many common objects from them. The GLU quadrics are also useful because they have capabilities that support many of the OpenGL rendering capabilities that support creating interesting images. You can determine the drawing style with the `gluQuadricDrawStyle()` function that lets you select whether you want the object filled, wireframe, silhouette, or drawn as points. You can get normal vectors to the surface for lighting models and smooth shading with the `gluQuadricNormals()` function that lets you choose whether you want no normals, or normals for flat or smooth shading. Finally, with the `gluQuadricTexture()` function you can specify whether you want to apply texture maps to the GLU quadrics in order to create objects with visual interest. See later chapters on lighting and on texture mapping for the details.

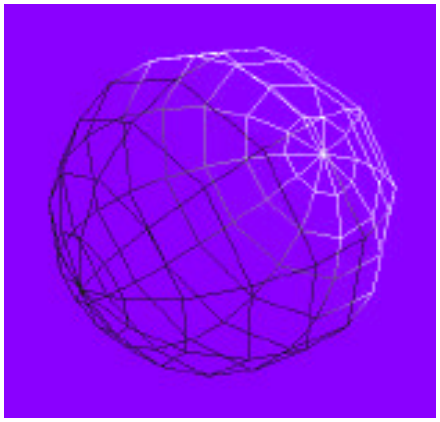


Figure 3.4: A GLUT wireframe sphere with 10 slices and 10 stacks

Below we describe each of the GLU primitives by listing its function prototype; more details may be found in the GLU section of your OpenGL manual.

GLU cylinder:

```
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top,
GLdouble height, GLint slices, GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
base is the radius of the cylinder at z = 0, the base of the cylinder
top is the radius of the cylinder at z = height, and
height is the height of the cylinder.

GLU disk:

The GLU disk is different from the other GLU primitives because it is only two-dimensional, lying entirely within the X-Y plane. Thus instead of being defined in terms of stacks, the second granularity parameter is loops, the number of concentric rings that define the disk.

```
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer,
GLint slices, GLint loops)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
inner is the inner radius of the disk (may be 0).
outer is the outer radius of the disk.

GLU sphere:

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices,
GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
radius is the radius of the sphere.

The GLUT objects

Models provided by GLUT are more oriented to geometric solids, except for the teapot object. They do not have as wide a usage in general situations because they are of fixed shape and many cannot be modeled with varying degrees of complexity. They also do not include shapes that can readily be adapted to general modeling situations. Finally, there is no general way to create a texture map for these objects, so it is more difficult to make scenes using them have stronger visual interest. The GLUT models include a cone (`glutSolidCone`), cube (`glutSolidCube`), dodecahedron (12-sided regular polyhedron, `glutSolidDodecahedron`), icosahedron (20-sided regular polyhedron, `glutSolidIcosahedron`), octahedron (8-sided regular polyhedron, `glutSolidOctahedron`), a sphere (`glutSolidSphere`), a teapot (the Utah teapot, an icon of computer graphics sometimes called the "teapotahedron", `glutSolidTeapot`), a tetrahedron (4-sided regular polyhedron, `glutSolidTetrahedron`), and a torus (`glutSolidTorus`). There are also wireframe versions of each of the GLUT solid objects.

The GLUT primitives include both solid and wireframe versions. Each object has a canonical position and orientation, typically being centered at the origin and lying within a standard volume and, if it has an axis of symmetry, that axis is aligned with the z-axis. As with the GLU standard primitives, the GLUT cone, sphere, and torus allow you to specify the granularity of the primitive's modeling, but the others do not. You should not take the term "solid" for the GLUT objects too seriously, however; they are not actually solid but are simply bounded by polygons. If you clip them you will find that they are, in fact, hollow.

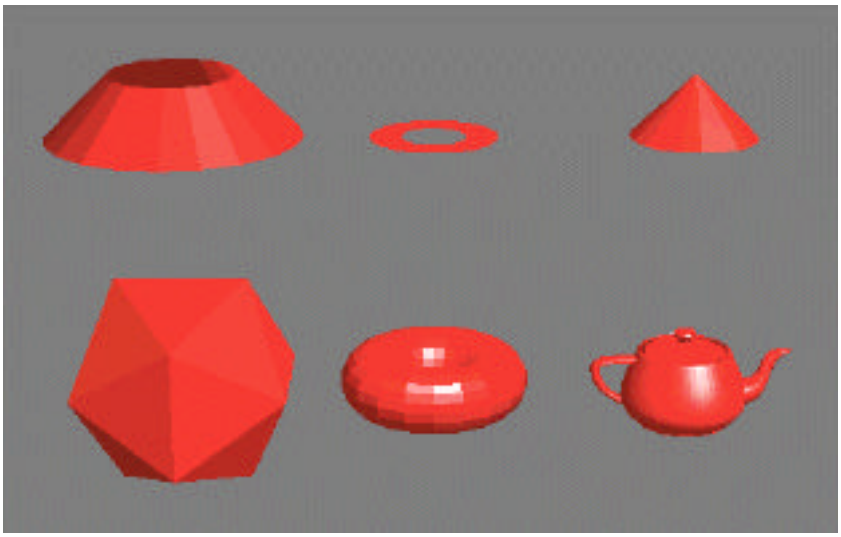


Figure 3.5: several GLU and GLUT objects as described in the text

If you have GLUT with your OpenGL, you should check the GLUT manuals for the details on these solids and on many other important capabilities that GLUT will add to your OpenGL system. If you do not already have it, you can download the GLUT code from the OpenGL Web site for many different systems and install it in your OpenGL area so you may use it readily with your system.

Selections from the overall collection of GLU and GLUT objects are shown in Figure 3.5 to show the range of items you can create with these tools. From top left and moving clockwise, we see a gluCylinder, a gluDisk, a glutSolidCone, a glutSolidIcosahedron, a glutSolidTorus, and a glutSolidTeapot. You should think about how you might use various transformations to create other figures from these basic parts.

An example

Our example for this module is quite simple. It is the heart of the `display()` function for a simple application that displays the built-in sphere, cylinder, dodecahedron, torus, and teapot provided by OpenGL and the GLU and GLUT toolkits. In the full example, there are operations that allow the user to choose the object and to control its display in several ways, but for this example we will only focus on the models themselves, as provided through a switch() statement such as might be used to implement a menu selection. This function is not complete, but would need the addition of viewing and similar functionality that is described in the chapter on viewing and projection.

```
void display( void )
{
   GLUquadric *myQuad;
   GLdouble radius = 1.0;
   GLint slices, stacks;
   GLint nsides, rings;

   ...

   switch (selectedObject) {
      case (1): {
         myQuad=gluNewQuadric();
```

```
            slices = stacks = resolution;
            gluSphere( myQuad , radius , slices , stacks  );
            break;
            }
         case (2): {
            myQuad=gluNewQuadric();
            slices = stacks = resolution;
            gluCylinder( myQuad, 1.0, 1.0, 1.0, slices, stacks );
            break;
            }
         case (3): {
            glutSolidDodecahedron(); break;
            }
         case (4): {
            nsides = rings = resolution;
            glutSolidTorus( 1.0, 2.0, nsides, rings);
            break;
            }
         case (5): {
            glutSolidTeapot(2.0); break;
            }
         }
      ...
}
```

A word to the wise...

One of the differences between student programming and professional programming is that students are often asked to create applications or tools for the sake of learning creation, not for the sake of creating working, useful things. The graphics primitives that are the subject of the first section of this module are the kind of tools that students are often asked to use, because they require more analysis of fundamental geometry and are good learning tools. However, working programmers developing real applications will often find it useful to use pre-constructed templates and tools such as the GLU or GLUT graphics primitives. You are encouraged to use the GLU and GLUT primitives whenever they can save you time and effort in your work, and when you cannot use them, you are encouraged to create your own primitives in a way that will let you re-use them as your own library and will let you share them with others.

Transformations in OpenGL

In OpenGL, there are only two kinds of transformations: projection transformations and modelview transformations. The latter includes both the viewing and modeling transformations. We have already discussed projections and viewing, so here we will focus on the transformations used in modeling.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. In OpenGL, these are applied with the built-in functions (actually function sets) `glRotate`, `glTranslate`, and `glScale`, respectively. As we have found with other OpenGL function sets, there are different versions of each of these, varying only in the kind of parameters they take.

The `glRotate` function is defined as

```
glRotatef(angle, x, y, z)
```

where `angle` specifies the angle of rotation, in degrees, and `x`, `y`, and `z` specify the coordinates of a vector, all as floats (`f`). There is another rotation function `glRotated` that operates in