Chapter 3

OpenGL 1.1: Geometry

It is time to move on to computer graphics in three dimensions, although it won't be until Section 2 of this chapter that we really get into 3D. You will find that many concepts from 2D graphics carry over to 3D, but the move into the third dimension brings with it some new features that take a while to get used to.

Our focus will be OpenGL, a graphics API that was introduced in 1992 and has gone through many versions and many changes since then. OpenGL is a low-level graphics API, similar to the 2D APIs we have covered. It is even more primitive in some ways, but of course it is complicated by the fact that it supports 3D. OpenGL is the basis for WebGL, the current standard for 3D applications on the Web that is covered in Chapter 6 and Chapter 7. There are many competing frameworks for low-level 3D graphics, including Microsoft's Direct3D, Apple's Metal, and Vulkan, which was designed by the creators of OpenGL as a more modern and efficient replacement.

For the next two chapters, the discussion is limited to OpenGL 1.1. OpenGL 1.1 is a large API, and we will only cover a part of it. The goal is to introduce 3D graphics concepts, not to fully cover the API. A significant part of what we cover here has been removed from the most modern versions of OpenGL, including WebGL. However, more modern graphics APIs have a very steep initial learning curve, and they are not really the best starting place for someone who is encountering 3D graphics for the first time. Some additional support is needed—if not OpenGL 1.1 then some similar framework. Since OpenGL 1.1 is still supported, at least by all desktop implementations of OpenGL, it's a reasonable place to start learning about 3D graphics.

This chapter concentrates on the geometric aspects of 3D graphics, such as defining and transforming objects and projecting 3D scenes into 2D images. The images that we produce will look very unrealistic. In the next chapter, we will see how to add some realism by simulating the effects of lighting and of the material properties of surfaces.

3.1 Shapes and Colors in OpenGL 1.1

This section introduces some of the core features of OpenGL. Much of the discussion in this section is limited to 2D. For now, all you need to know about 3D is that it adds a third direction to the *x* and *y* directions that are used in 2D. By convention, the third direction is called *z*. In the default coordinate system, the *x* and *y* axes lie in the plane of the image, and the positive direction of the *z*-axis points in a direction perpendicular to the image.

In the default coordinate system for OpenGL, the image shows a region of 3D space in which

77

x, *y*, and *z* all range from minus one to one. To show a different region, you have to apply a transform. For now, we will just use coordinates that lie between -1 and 1.

A note about programming: OpenGL can be implemented in many different programming languages, but the API specification more or less assumes that the language is C. (See Section A.2 for a short introduction to C.) For the most part, the C specification translates directly into other languages. The main differences are due to the special characteristics of arrays in the C language. My examples will follow the C syntax, with a few notes about how things can be different in other languages. Since I'm following the C API, I will refer to "functions" rather than "subroutines" or "methods." Section 3.6 explains in detail how to write OpenGL programs in C and in Java. You will need to consult that section before you can do any actual programming. The live OpenGL 1.1 demos for this book are written using a JavaScript simulator that implements a subset of OpenGL 1.1. That simulator is discussed in Subsection 3.6.3.

3.1.1 OpenGL Primitives

OpenGL can draw only a few basic shapes, including points, lines, and triangles. There is no built-in support for curves or curved surfaces; they must be approximated by simpler shapes. The basic shapes are referred to as primitives. A primitive in OpenGL is defined by its vertices. A vertex is simply a point in 3D, given by its *x*, *y*, and *z* coordinates. Let's jump right in and see how to draw a triangle. It takes a few steps:

glBegin(GL _TRIANGLES); glVertex2f(-0.7, -0.5); glVertex2f(0.7, -0.5); glVertex2f(0, 0.7); glEnd();

Each vertex of the triangle is specified by a call to the function *glVertex2f*. Vertices must be specified between calls to *glBegin* and *glEnd*. The parameter to *glBegin* tells which type of primitive is being drawn. The *GL TRIANGLES* primitive allows you to draw more than one triangle: Just specify three vertices for each triangle that you want to draw. Note that using *glBegin/glEnd* is not the preferred way to specify primitives, even in OpenGL 1.1. However, the alternative, which is covered in Subsection 3.4.2, is more complicated to use. You should consider *glBegin/glEnd* to be a convenient way to learn about vertices and their properties, but not the way that you will actually do things in modern graphics APIs.

(I should note that OpenGL functions actually just send commands to the GPU. OpenGL can save up batches of commands to transmit together, and the drawing won't actually be done until the commands are transmitted. To ensure that that happens, the function *glFlush*() must be called. In some cases, this function might be called automatically by an OpenGL API, but you might well run into times when you have to call it yourself.)

For OpenGL, vertices have three coordinates. The function *glVertex2f* specifies the *x* and *y* coordinates of the vertex, and the *z* coordinate is set to zero. There is also a function *glVertex3f* that specifies all three coordinates. The “2” or “3” in the name tells how many parameters are passed to the function. The “f” at the end of the name indicates that the parameters are of type float. In fact, there are other “glVertex” functions, including versions that take parameters of type int or double, with named like *glVertex2i* and *glVertex3d*. There are even versions that take four parameters, although it won’t be clear for a while why they should exist. And, as we will see later, there are versions that take an array of numbers instead of individual numbers as parameters. The entire set of vertex functions is often referred to as “glVertex*”, with the “*” standing in for the parameter specification. (The proliferation of names is due to the fact that the C programming language doesn’t support overloading of function names; that is, C distinguishes functions only by their names and not by the number and type of parameters that are passed to the function.)

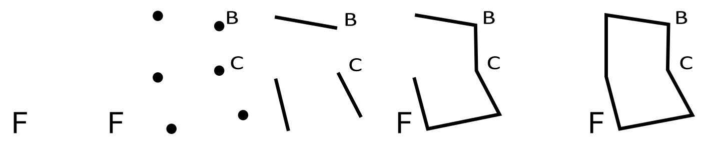
OpenGL 1.1 has ten kinds of primitive. Seven of them still exist in modern OpenGL; the other three have been dropped. The simplest primitive is *GL POINTS*, which simply renders a point at each vertex of the primitive. By default, a point is rendered as a single pixel. The size of point primitives can be changed by calling glPointSize(size);

where the parameter, *size*, is of type float and specifies the diameter of the rendered point, in pixels. By default, points are squares. You can get circular points by calling glEnable(GL POINT SMOOTH);

The functions *glPointSize* and *glEnable* change the OpenGL “state.” The state includes all the settings that affect rendering. We will encounter many state-changing functions. The functions *glEnable* and *glDisable* can be used to turn many features on and off. In general, the rule is that any rendering feature that requires extra computation is turned off by default. If you want that feature, you have to turn it on by calling *glEnable* with the appropriate parameter.

There are three primitives for drawing line segments: *GL LINES*, *GL LINE STRIP*, and *GL LINE _LOOP*. *GL LINES* draws disconnected line segments; specify two vertices for each segment that you want to draw. The other two primitives draw connected sequences of line segments. The only difference is that *GL LINE _LOOP* adds an extra line segment from the final vertex back to the first vertex. Here is what you get if use the same six vertices with the four primitives we have seen so far:

AAAA



ED ED ED ED

GL_POINTS GL_LINES GL_LINE_STRIP GL_LINE_LOOP

The points A, B, C, D, E, and F were specified in that order. In this illustration, all the points lie in the same plane, but keep in mind that in general, points can be anywhere in 3D space.

The width for line primitives can be set by calling *glLineWidth*(*width*). The line width is always specified in pixels. It is **not** subject to scaling by transformations.

Let's look at an example. OpenGL does not have a circle primitive, but we can approximate a circle by drawing a polygon with a large number of sides. To draw an outline of the polygon, we can use a *GL LINE _LOOP* primitive:

glBegin(GL LINE LOOP); for (i = 0; i < 64; i++) { angle = 6.2832 * i / 64; // 6.2832 represents 2*PI x = 0.5 * cos(angle); y = 0.5 * sin(angle); glVertex2f(x, y);

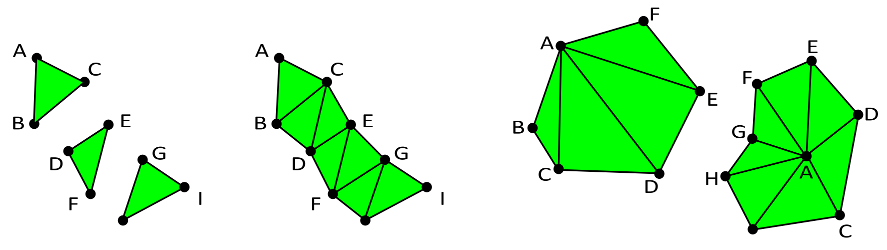
} glEnd();

This draws an approximation for the circumference of a circle of radius 0.5 with center at (0,0). Remember that to learn how to use examples like this one in a complete, running program, you will have to read Section 3.6. Also, you might have to make some changes to the code, depending on which OpenGL implementation you are using.

The next set of primitives is for drawing triangles. There are three of them:

GL TRIANGLES, *GL TRIANGLE _STRIP*, and *GL TRIANGLE FAN*.

_



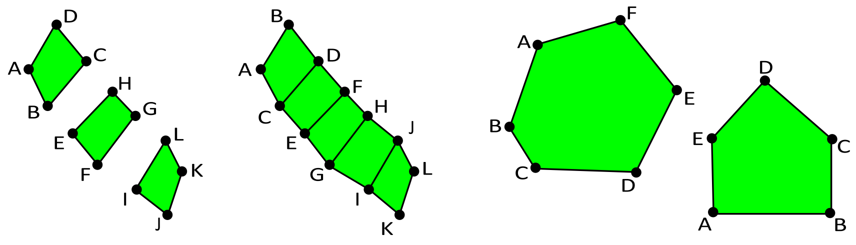
H H B

GL_TRIANGLES GL_TRIANGLE_STRIP GL_TRIANGLE_FAN

The three triangles on the left make up one *GL TRIANGLES* primitive, with nine vertices. With that primitive, every set of three vertices makes a separate triangle. For a *GL*

TRIANGLE _STRIP primitive, the first three vertices produce a triangle. After that, every new vertex adds another triangle to the strip, connecting the new vertex to the two previous vertices. Two *GL TRIANGLE _FAN* primitives are shown on the right. Again for a *GL TRIANGLE _FAN*, the first three vertices make a triangle, and every vertex after that adds anther triangle, but in this case, the new triangle is made by connecting the new vertex to the previous vertex and to the very first vertex that was specified (vertex "A" in the picture). Note that *Gl TRIANGLE _FAN* can be used for drawing filled-in polygons. In this picture, by the way, the dots and lines are not part of the primitive; OpenGL would only draw the filled-in, green interiors of the figures.

The three remaining primitives, which have been removed from modern OpenGL, are *GL QUADS*, *GL QUAD _STRIP*, and *GL POLYGON*. The name "quad" is short for quadrilateral, that is, a four-sided polygon. A quad is determined by four vertices. In order for a quad to be rendered correctly in OpenGL, all vertices of the quad must lie in the same plane. The same is true for polygon primitives. Similarly, to be rendered correctly, quads and polygons must be convex (see Subsection 2.2.3). Since OpenGL doesn't check whether these conditions are satisfied, the use of quads and polygons is error-prone. Since the same shapes can easily be produced with the triangle primitives, they are not really necessary, but here for the record are some examples:



GL_QUADS GL_QUAD_STRIP GL_POLYGON

The vertices for these primitives are specified in the order A, B, C, ... Note how the order differs for the two quad primitives: For *GL QUADS*, the vertices for each individual quad should be specified in counterclockwise order around the quad; for *GL _QUAD STRIP*, the vertices should alternate from one side of the strip to the other.

3.1.2 OpenGL Color

OpenGL has a large collection of functions that can be used to specify colors for the geometry that we draw. These functions have names of the form *glColor**, where the "*" stands for a suffix that gives the number and type of the parameters. I should warn you now that for realistic 3D graphics, OpenGL has a more complicated notion of color that uses a different set of functions. You will learn about that in the next chapter, but for now we will stick to *glColor**.

For example, the function *glColor3f* has three parameters of type float. The parameters give the red, green, and blue components of the color as numbers in the range 0.0 to 1.0. (In fact, values outside this range are allowed, even negative values. When color values are used in computations, out-of-range values will be used as given. When a color actually appears on the screen, its component values are clamped to the range 0 to 1. That is, values less than zero are changed to zero, and values greater than one are changed to one.)

You can add a fourth component to the color by using *glColor4f* (). The fourth component, known as alpha, is not used in the default drawing mode, but it is possible to configure OpenGL to use it as the degree of transparency of the color, similarly to the use of the alpha component in the 2D graphics APIs that we have looked at. You need two commands to turn on transparency:

```
glEnable(GL BLEND);

glBlendFunc(GL SRC ALPHA, GL ONE MINUS _SRC ALPHA);
```

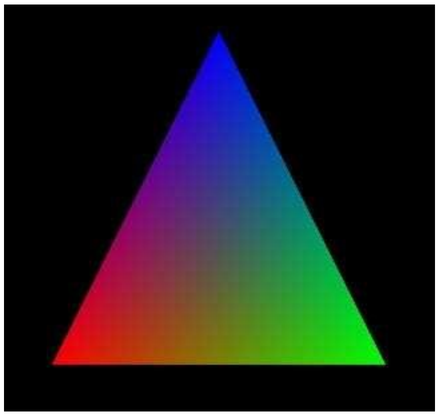
The first command enables use of the alpha component. It can be disabled by calling *glDisable*(*GL BLEND*). When the *GL _BLEND* option is disabled, alpha is simply ignored. The second command tells how the alpha component of a color will be used. The parameters shown here are the most common; they implement transparency in the usual way. I should note that while transparency works fine in 2D, it is much more difficult to use transparency correctly in 3D.

If you would like to use integer color values in the range 0 to 255, you can use *glColor3ub*() or *glColor4ub* to set the color. In these function names, “ub” stands for “unsigned byte.” Unsigned byte is an eight-bit data type with values in the range 0 to 255. Here are some examples of commands for setting drawing colors in OpenGL:

```
    glColor3f(0,0,0);          // Draw in black.

    glColor3f(1,1,1);          // Draw in white.

    glColor3f(1,0,0);          // Draw in full-intensity red.

    glColor3ub(1,0,0);         // Draw in a color just a tiny bit different from

                               // black. (The suffix, "ub" or "f", is important!)

    glColor3ub(255,0,0);// Draw in full-intensity red.
```

glColor4f(1, 0, 0, 0.5); // Draw in transparent red, but only if OpenGL // has been configured to do transparency. By // default this is the same as drawing in plain red.

Using any of these functions sets the value of a “current color,” which is part of the OpenGL state. When you generate a vertex with one of the *glVertex** functions, the current color is saved along with the vertex coordinates, as an attribute of the vertex. We will see that vertices can have other kinds of attribute as well as color. One interesting point about OpenGL is that colors are associated with individual vertices, not with complete shapes. By changing the current color between calls to *glBegin*() and *glEnd*(), you can get a shape in which different vertices have different color attributes. When you do this, OpenGL will compute the colors of pixels inside the shape by interpolating the colors of the vertices. (Again, since OpenGL is extremely configurable, I have to note that interpolation of colors is just the default behavior.) For example, here is a triangle in which the three vertices are assigned the colors red, green, and blue:



This image is often used as a kind of “Hello World” example for OpenGL. The triangle can be drawn with the commands

```
glBegin(GL _TRIANGLES); glColor3f( 1, 0, 0 ); // red glVertex2f( -0.8, -0.8 ); glColor3f( 0, 1, 0 ); // green glVertex2f( 0.8, -0.8 ); glColor3f( 0, 0, 1 ); // blue glVertex2f( 0, 0.9 ); glEnd();
```

Note that when drawing a primitive, you do **not** need to explicitly set a color for each vertex, as was done here. If you want a shape that is all one color, you just have to set the current color once, before drawing the shape (or just after the call to *glBegin*(). For example, we can draw a solid yellow triangle with

```
glColor3ub(255,255,0); // yellow
```

```
glBegin(GL _TRIANGLES); glVertex2f( -0.5, -0.5 ); glVertex2f( 0.5, -0.5 ); glVertex2f( 0, 0.5 ); glEnd();
```

Also remember that the color for a vertex is specified **before** the call to *glVertex** that generates the vertex.

The on-line version of this section has an interactive demo that draws the basic OpenGL triangle, with different colored vertices. That demo is our first OpenGL example. The demo actually uses WebGL, so you can use it as a test to check whether your web browser supports WebGL.

The sample program *jogl/FirstTriangle.java* draws the basic OpenGL triangle using Java. The program *glut/first-triangle.c* does the same using the C programming language. And *glsim/first-triangle.html* is a version that uses my JavaScript simulator, which implements just the parts of OpenGL 1.1 that are covered in this book. Any of those programs could be used to experiment with 2D drawing in OpenGL. And you can use them to test your OpenGL programming environment.

* * *

A common operation is to clear the drawing area by filling it with some background color. It is be possible to do that by drawing a big colored rectangle, but OpenGL has a potentially more efficient way to do it. The function glClearColor(r,g,b,a);

sets up a color to be used for clearing the drawing area. (This only sets the color; the color isn't used until you actually give the command to clear the drawing area.) The parameters are floating point values in the range 0 to 1. There are no variants of this function; you must provide all four color components, and they must be in the range 0 to 1. The default clear color is all zeros, that is, black with an alpha component also equal to zero. The command to do the actual clearing is:

glClear(GL COLOR _BUFFER BIT);

The correct term for what I have been calling the drawing area is the color buffer, where "buffer" is a general term referring to a region in memory. OpenGL uses several buffers in addition to the color buffer. We will encounter the "depth buffer" in just a moment. The *glClear* command can be used to clear several different buffers at the same time, which can be more efficient than clearing them separately since the clearing can be done in parallel. The parameter to *glClear* tells it which buffer or buffers to clear. To clear several buffers at once, combine the constants that represent them with an arithmetic OR operation. For example, glClear(GL COLOR _BUFFER BIT | GL DEPTH BUFFER BIT);

This is the form of *glClear* that is generally used in 3D graphics, where the depth buffer plays an essential role. For 2D graphics, the depth buffer is generally not used, and the appropriate parameter for *glClear* is just *GL COLOR _BUFFER _BIT*.

3.1.3 glColor and glVertex with Arrays

We have see that there are versions of *glColor** and *glVertex** that take different numbers and types of parameters. There are also versions that let you place all the data for the command in a single array parameter. The names for such versions end with "v".

For example: *glColor3fv*, *glVertex2iv*, *glColor4ubv*, and *glVertex3dv*. The “v” actually stands for “vector,” meaning essentially a one-dimensional array of numbers. For example, in the function call *glVertex3fv*(*coords*), *coords* would be an array containing at least three floating point numbers.

The existence of array parameters in OpenGL forces some differences between OpenGL implementations in different programming languages. Arrays in Java are different from arrays in C, and arrays in JavaScript are different from both. Let’s look at the situation in C first, since that’s the language of the original OpenGL API.

In C, array variables are a sort of variation on pointer variables, and arrays and pointers can be used interchangeably in many circumstances. In fact, in the C API, array parameters are actually specified as pointers. For example, the parameter for *glVertex3fv* is of type “pointer to float.” The actual parameter in a call to *glVertex3fv* can be an array variable, but it can also be any pointer that points to the beginning of a sequence of three floats. As an example, suppose that we want to draw a square. We need two coordinates for each vertex of the square. In C, we can put all 8 coordinates into one array and use *glVertex2fv* to pull out the coordinates that we need:

```
float coords[] = { -0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5 };
```

```
glBegin(GL _TRIANGLE _FAN); glVertex2fv(coords); // Uses coords[0] and coords[1].
glVertex2fv(coords + 2); // Uses coords[2] and coords[3]. glVertex2fv(coords + 4); //
Uses coords[4] and coords[5]. glVertex2fv(coords + 6); // Uses coords[6] and coords[7].
glEnd();
```

This example uses “pointer arithmetic,” in which *coords* + *N* represents a pointer to the N-th element of the array. An alternative notation would be &*coords*[*N*], where “&” is the address operator, and &*coords*[*N*] means “a pointer to *coords*[*N*]”. This will all seem very alien to people who are only familiar with Java or JavaScript. In my examples, I will avoid using pointer arithmetic, but I will occasionally use address operators.

As for Java, the people who designed JOGL wanted to preserve the ability to pull data out of the middle of an array. However, it’s not possible to work with pointers in Java. The solution was to replace a pointer parameter in the C API with a pair of parameters in the JOGL API—one parameter to specify the array that contains the data and one to specify the starting index of the data in the array. For example, here is how the square-drawing code translates into Java:

```
float[] coords = { -0.5F, -0.5F, 0.5F, -0.5F, 0.5F, 0.5F, -0.5F, 0.5F };
```

```
gl2.glBegin(GL2.GL _TRIANGLES); gl2.glVertex2fv(coords, 0); // Uses coords[0] and
coords[1]. gl2.glVertex2fv(coords, 2); // Uses coords[2] and coords[3].
```

gl2.glVertex2fv(coords, 4); // Uses coords[4] and coords[5]. gl2.glVertex2fv(coords, 6); // Uses coords[6] and coords[7]. gl2.glEnd();

There is really not much difference in the parameters, although the zero in the first *glVertex2fv* is a little annoying. The main difference is the prefixes “gl2” and “GL2”, which are required by the object-oriented nature of the JOGL API. I won’t say more about JOGL here, but if you need to translate my examples into JOGL, you should keep in mind the extra parameter that is required when working with arrays.

For the record, here are the *glVertex** and *glColor** functions that I will use in this book.

This is not the complete set that is available in OpenGL:

```
glVertex2f( x, y );            glVertex2fv( xyArray );
glVertex2d( x, y );            glVertex2dv( xyArray );
glVertex2i( x, y );            glVertex2iv( xyArray );
glVertex3f( x, y, z );         glVertex3fv( xyzArray );
glVertex3d( x, y, z );         glVertex3dv( xyzArray );
glVertex3i( x, y, z );         glVertex3iv( xyzArray );
glColor3f( r, g, b );          glColor3f( rgbArray );
glColor3d( r, g, b );          glColor3d( rgbArray );
glColor3ub( r, g, b );         glColor3ub( rgbArray );
```

glColor4f(r, g, b, a);   glColor4f(rgbaArray); glColor4d(r, g, b, a);   glColor4d(rgbaArray); glColor4ub(r, g, b, a);   glColor4ub(rgbaArray);

For *glColor**, keep in mind that the “ub” variations require integers in the range 0 to 255, while the “f” and “d” variations require floating-point numbers in the range 0.0 to 1.0.

3.1.4 The Depth Test

An obvious point about viewing in 3D is that one object can be behind another object. When this happens, the back object is hidden from the viewer by the front object. When we create an image of a 3D world, we have to make sure that objects that are supposed to be hidden behind other objects are in fact not visible in the image. This is the hidden surface problem.

The solution might seem simple enough: Just draw the objects in order from back to front. If one object is behind another, the back object will be covered up later when the front object is drawn. This is called the painter’s algorithm. It’s essentially what you are used to doing in 2D. Unfortunately, it's not so easy to implement. First of all, you can have objects that intersect, so that part of each object is hidden by the other. Whatever order you draw the objects in, there will be some points where the wrong object is visible. To fix this, you would have to cut the objects into pieces, along the intersection, and treat the pieces as separate objects. In fact, there can be problems even if there are no intersecting objects: It's possible to have three non-intersecting objects where the first object hides part of the second, the second hides part of the third, and the third hides part of the first. The painter's algorithm will fail regardless of the order in which the three objects are drawn. The solution again is to cut the objects into pieces, but now it's not so obvious where to cut.

Even though these problems can be solved, there is another issue. The correct drawing order can change when the point of view is changed or when a geometric transformation is applied, which means that the correct drawing order has to be recomputed every time that happens. In an animation, that would mean for every frame.

So, OpenGL does not use the painter's algorithm. Instead, it uses a technique called the depth test. The depth test solves the hidden surface problem no matter what order the objects are drawn in, so you can draw them in any order you want! The term "depth" here has to do with the distance from the viewer to the object. Objects at greater depth are farther from the viewer. An object with smaller depth will hide an object with greater depth. To implement the depth test algorithm, OpenGL stores a depth value for each pixel in the image. The extra memory that is used to store these depth values makes up the depth buffer that I mentioned earlier. During the drawing process, the depth buffer is used to keep track of what is currently visible at each pixel. When a second object is drawn at that pixel, the information in the depth buffer can be used to decide whether the new object is in front of or behind the object that is currently visible there. If the new object is in front, then the color of the pixel is changed to show the new object, and the depth buffer is also updated. If the new object is behind the current object, then the data for the new object is discarded and the color and depth buffers are left unchanged.

By default, the depth test is **not** turned on, which can lead to very bad results when drawing in 3D. You can enable the depth test by calling glEnable(GL DEPTH TEST);

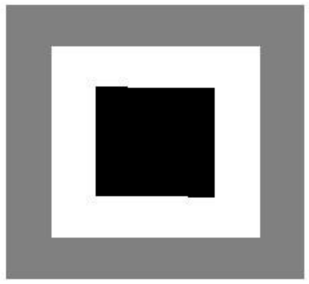
It can be turned off by calling *glDisable*(*GL DEPTH _TEST*). If you forget to enable the depth test when drawing in 3D, the image that you get will likely be confusing and will make no sense physically. You can also get quite a mess if you forget to clear the depth buffer, using the *glClear* command shown earlier in this section, at the same time that you clear the color buffer.

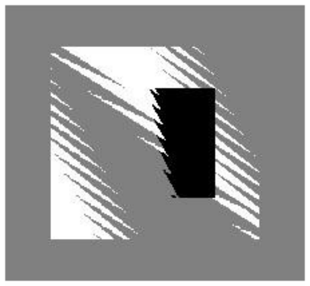
The demo *c3/first-cube.html* in the online version of this section lets you experiment with the depth test. It also lets you see what happens when part of your geometry extends outside the visible range of *z*-values.

Here are a few details about the implementation of the depth test: For each pixel, the depth buffer stores a representation of the distance from the viewer to the point that is currently visible at that pixel. This value is essentially the *z*-coordinate of the point, after any transformations have been applied. (In fact, the depth buffer is often called the "z-buffer".) The range of possible *z*-coordinates is scaled to the range 0 to 1. The fact that there is only a limited range of depth buffer values means that OpenGL can only display objects in a limited range of distances from the viewer. A depth value of 0 corresponds to the minimal distance; a depth value of 1 corresponds to the maximal distance. When you clear the depth buffer, every depth value is set to 1, which can be thought of as representing the background of the image.

You get to choose the range of *z*-values that is visible in the image, by the transformations that you apply. The default range, in the absence of any transformations, is -1 to 1. Points with *z*-values outside the range are not visible in the image. It is a common problem to use too small a range of *z*-values, so that objects are missing from the scene, or have their fronts or backs cut off, because they lie outside of the visible range. You might be tempted to use a huge range, to make sure that the objects that you want to include in the image are included within the range. However, that's not a good idea: The depth buffer has a limited number of bits per pixel and therefore a limited amount of accuracy. The larger the range of values that it must represent, the harder it is to distinguish between objects that are almost at the same depth. (Think about what would happen if all objects in your scene have depth values between 0.499999 and 0.500001—the depth buffer might see them all as being at exactly the same depth!)

There is another issue with the depth buffer algorithm. It can give some strange results when two objects have exactly the same depth value. Logically, it's not even clear which object should be visible, but the real problem with the depth test is that it might show one object at some points and the second object at some other points. This is possible because numerical calculations are not perfectly accurate. Here an actual example:





In the two pictures shown here, a gray square was drawn, followed by a white square, followed by a black square. The squares all lie in the same plane. A very small rotation was applied, to force the computer do some calculations before drawing the objects. The picture on the left was drawn with the depth test disabled, so that, for example, when a pixel of the white square was drawn, the computer didn't try to figure out whether it lies in front of or behind the gray square; it simply colored the pixel white. On the right, the depth test was enabled, and you can see the strange result.

Finally, by the way, note that the discussion here assumes that there are no transparent objects. Unfortunately, the depth test does not handle transparency correctly, since transparency means that two or more objects can contribute to the color of the pixel, but the depth test assumes that the pixel color is the color of the object nearest to the viewer at that point. To handle 3D transparency correctly in OpenGL, you pretty much have to resort to implementing the painter's algorithm by hand, at least for the transparent objects in the scene.

3.2 3D Coordinates and Transforms

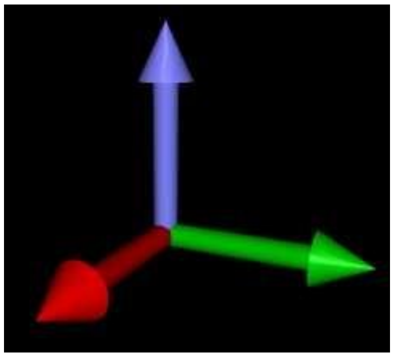
In Chapter 2, we looked fairly closely at coordinate systems and transforms in twodimensional computer graphics. In this section and the next, we will move that discussion into 3D. Things are more complicated in three dimensions, but a lot of the basic concepts remain the same.

3.2.1 3D Coordinates

A coordinate system is a way of assigning numbers to points. In two dimensions, you need a pair of numbers to specify a point. The coordinates are often referred to as *x* and *y*, although of course, the names are arbitrary. More than that, the assignment of pairs of numbers to points is itself arbitrary to a large extent. Points and objects are real things, but coordinates are just numbers that we assign to them so that we can refer to them easily and work with them mathematically. We have seen the power of this when we discussed transforms, which are defined mathematically in terms of coordinates but which have real, useful physical meanings.

In three dimensions, you need three numbers to specify a point. (That's essentially what it means to be three dimensional.) The third coordinate is often called *z*. The *z*-axis is perpendicular to both the *x*-axis and the *y*-axis.

This image illustrates a 3D coordinate system. The positive directions of the *x*, *y*, and *z* axes are shown as big arrows. The *x*-axis is green, the *y*-axis is blue, and the *z*-axis is red. The on-line version of this section has a demo version of this image in which you drag on the axes to rotate the image.



This example is a 2D image, but it has a 3D look. (The illusion is much stronger if you can rotate the image.) Several things contribute to the effect. For one thing, objects that are farther away from the viewer in 3D look smaller in the 2D image. This is due to the way that the 3D scene is "projected" onto 2D. We will discuss projection in the next section. Another factor is the "shading" of the objects. The objects are shaded in a way that imitates the interaction of objects with the light that illuminates them. We will put off a discussion of lighting until Chapter 4. In this section, we will concentrate on how to construct a scene in 3D—what we have referred to as modeling.

OpenGL programmers usually think in terms of a coordinate system in which the *x*- and *y*-axes lie in the plane of the screen, and the *z*-axis is perpendicular to the screen with the positive direction of the *z*-axis pointing **out of** the screen towards the viewer. Now, the default coordinate system in OpenGL, the one that you are using if you apply no transformations at all, is similar but has the positive direction of the *z*-axis pointing **into** the screen. This is not a contradiction: The coordinate system that is actually used is arbitrary. It is set up by a transformation. The convention in OpenGL is to work with a coordinate system in which the positive *z*-direction points toward the viewer and the negative *z*-direction points away from the viewer. The transformation into default coordinates reverses the direction of the *z*-axis.

This conventional arrangement of the axes produces a right-handed coordinate system. This means that if you point the thumb of your right hand in the direction of the positive *z*-axis, then when you curl the fingers of that hand, they will curl in the direction from the positive *x*-axis towards the positive *y*-axis. If you are looking at the tip of your thumb, the curl will be in the counterclockwise direction. Another way to think about it is that if you curl the figures of your right hand from the positive *x* to the positive *y*-axis, then your thumb will point in the direction of the positive *z*-axis. The default OpenGL coordinate system (which, again, is hardly ever used) is a left-handed system. You should spend some time trying to visualize right- and left-handed coordinates systems. Use your hands!

All of that describes the natural coordinate system from the viewer's point of view, the so-called "eye" or "viewing" coordinate system. However, these eye coordinates are not necessarily the natural coordinates on the world. The coordinate system on the world—the coordinate system in which the scene is assembled—is referred to as world coordinates.

Recall that objects are not usually specified directly in world coordinates. Instead, objects are specified in their own coordinate system, known as object coordinates, and then modeling transforms are applied to place the objects into the world, or into more complex objects. In OpenGL, object coordinates are the numbers that are used in the *glVertex** function to specify the vertices of the object. However, before the objects appear on the screen, they are usually subject to a sequence of transformations, starting with a modeling transform.

3.2.2 Basic 3D Transforms

The basic transforms in 3D are extensions of the basic transforms that you are already familiar with from 2D: rotation, scaling, and translation. We will look at the 3D equivalents and see how they affect objects when applied as modeling transforms. We will also discuss how to use the transforms in OpenGL.

Translation is easiest. In 2D, a translation adds some number onto each coordinate. The same is true in 3D; we just need three numbers, to specify the amount of motion in the direction of each of the coordinate axes. A translation by (*dx,dy,dz*) transforms a point (*x,y,z*) to the point (*x+dx, y+dy, z+dz*). In OpenGL, this translation would be specified by the command glTranslatef(dx, dy, dz);

or by the command glTranslated(dx, dy, dz);

The translation will affect any drawing that is done after the command is given. Note that there are two versions of the command. The first, with a name ending in "f", takes three float values as parameters. The second, with a name ending in "d", takes parameters of type double.

As an example, glTranslatef(0, 0, 1);

would translate objects by one unit in the *z* direction.

Scaling works in a similar way: Instead of one scaling factor, you need three. The OpenGL command for scaling is *glScale**, where the "*" can be either "f" or "d". The command glScalef(sx, sy, sz);

transforms a point (*x,y,z*) to (*x*sx, y*sy, z*sz*). That is, it scales by a factor of *sx* in the *x* direction, *sy* in the *y* direction, and *sz* in the *z* direction. Scaling is about the origin; that is, it moves points farther from or closer to the origin, (0,0,0). For uniform scaling, all three factors would be the same. You can use scaling by a factor of minus one to apply a reflection. For example, glScalef(1, 1, -1);

reflects objects through the *xy*-plane by reversing the sign of the *z* coordinate. Note that a reflection will convert a right-handed coordinate system into a left-handed coordinate system, and *vice versa*. Remember that the left/right handed distinction is not a property of the world, just of the way that one chooses to lay out coordinates on the world.

Rotation in 3D is harder. In 2D, rotation is rotation about a point, which is usually taken to be the origin. In 3D, rotation is rotation about a line, which is called the axis of rotation. Think of the Earth rotating about its axis. The axis of rotation is the line that passes through the North Pole and the South Pole. The axis stays fixed as the Earth rotates around it, and points that are not on the axis move in circles about the axis. Any line can be an axis of rotation, but we generally use an axis that passes through the origin. The most common choices for axis of rotation are the coordinates axes, that is, the *x*-axis, the *y*-axis, or the *z*-axis. Sometimes, however, it's convenient to be able to use a different line as the axis.

There is an easy way to specify a line that passes through the origin: Just specify one other point that is on the line, in addition to the origin. That's how things are done in OpenGL: An axis of rotation is specified by three numbers, (*ax,ay,az*), which are not all zero. The axis is the line through (0,0,0) and (*ax,ay,az*). To specify a rotation transformation in 3D, you have to specify an axis and the angle of rotation about that axis.

We still have to account for the difference between positive and negative angles. We can't just say clockwise or counterclockwise. If you look down on the rotating Earth from above the North pole, you see a counterclockwise rotation; if you look down on it from above the South pole, you see a clockwise rotation. So, the difference between the two is not well-defined. To define the direction of rotation in 3D, we use the right-hand rule, which says: Point the thumb of your right hand in the direction of the axis — from the point (0,0,0) towards the point (*ax,ay,az*) that determines the axis. Then the direction of rotation for positive angles is given by the direction in which your fingers curl. I should emphasize that the right-hand rule only works if you are working in a right-handed coordinate system. If you have switched to a left-handed coordinate system, then you need to use a left-hand rule to determine the positive direction of rotation.

The demo *c3/rotation-axis.html* can help you to understand rotation in three-dimensional space.

The rotation function in OpenGL is *glRotatef* (*r,ax,ay,az*). You can also use *glRotated*. The first parameter specifies the angle of rotation, measured in degrees. The other three parameters specify the axis of rotation, which is the line from (0,0,0) to (*ax,ay,az*).

Here are a few examples of scaling, translation, and scaling in OpenGL: glScalef(2,2,2); // Uniform scaling by a factor of 2.

```
    glScalef(0.5,1,1);      // Shrink by half in the x-direction only.

    glScalef(-1,1,1);       // Reflect through the yz-plane.
```

// Reflects the positive x-axis onto negative x.

glTranslatef(5,0,0); // Move 5 units in the positive x-direction. glTranslatef(3,5,-7.5); // Move each point (x,y,z) to (x+3, y+5, z-7.5).

```
    glRotatef(90,1,0,0);  // Rotate 90 degrees about the x-axis.
```

// Moves the +y axis onto the +z axis

```
    //      and the +z axis onto the -y axis.
```

glRotatef(-90,-1,0,0); // Has the same effect as the previous rotation.

```
    glRotatef(90,0,1,0);      // Rotate 90 degrees about the y-axis.

                              // Moves the +z axis onto the +x axis

                              //      and the +x axis onto the -z axis.

    glRotatef(90,0,0,1);      // Rotate 90 degrees about the z-axis.

                              // Moves the +x axis onto the +y axis

                              //      and the +y axis onto the -x axis.
```

glRotatef(30,1.5,2,-3); // Rotate 30 degrees about the line through //   the points (0,0,0) and (1.5,2,-3).

Remember that transforms are applied to objects that are drawn after the transformation function is called, and that transformations apply to objects in the opposite order of the order in which they appear in the code.

Of course, OpenGL can draw in 2D as well as in 3D. For 2D drawing in OpenGL, you can draw on the *xy*-plane, using zero for the *z* coordinate. When drawing in 2D, you will probably want to apply 2D versions of rotation, scaling, and translation. OpenGL does not have 2D transform functions, but you can just use the 3D versions with appropriate parameters:

- For translation by (*dx,dy*) in 2D, use *glTranslatef* (*dx, dy, 0*). The zero translation in the *z* direction means that the transform doesn't change the *z* coordinate, so it maps the *xy*-plane to itself. (Of course, you could use *glTranslated* instead of *glTranslatef*.)
- For scaling by (*sx,sy*) in 2D, use *glScalef* (*sx, sy, 1*), which scales only in the *x* and *y* directions, leaving the *z* coordinate unchanged.
- For rotation through an angle *r* about the origin in 2D, use *glRotatef* (*r, 0, 0, 1*). This is rotation about the *z*-axis, which rotates the *xy*-plane into itself. In the usual OpenGL coordinate system, the *z*-axis points out of the screen, and the right-hand rule says that rotation by a positive angle will be in the counterclockwise direction in the *xy*-plane. Since the *x*-axis points to the right and the *y*-axis points upwards, a counterclockwise rotation rotates the positive *x*-axis in the direction of the positive *y*-axis. This is the same convention that we have used previously for the positive direction of rotation.

3.2.3 Hierarchical Modeling

Modeling transformations are often used in hierarchical modeling, which allows complex objects to be built up out of simpler objects. See Section 2.4. To review briefly: In hierarchical modeling, an object can be defined in its own natural coordinate system, usually using (0,0,0) as a reference point. The object can then be scaled, rotated, and translated to place it into world coordinates or into a more complex object. To implement this, we need a way of limiting the effect of a modeling transformation to one object or to part of an object. That can be done using a stack of transforms. Before drawing an object, push a copy of the current transform onto the stack. After drawing the object and its sub-objects, using any necessary temporary transformations, restore the previous transform by popping it from the stack.

OpenGL 1.1 maintains a stack of transforms and provides functions for manipulating that stack. (In fact it has several transform stacks, for different purposes, which introduces some complications that we will postpone to the next section.) Since transforms are represented as matrices, the stack is actually a stack of matrices. In OpenGL, the functions for operating on the stack are named *glPushMatrix*() and *glPopMatrix*().

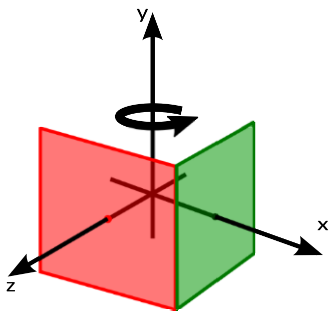
These functions do not take parameters or return a value. OpenGL keeps track of a current matrix, which is the composition of all transforms that have been applied. Calling a function such as *glScalef* simply modifies the current matrix. When an object is drawn, using the *glVertex** functions, the coordinates that were specified for the object are transformed by the current matrix. There is another function that affects the current matrix: *glLoadIdentity*(). Calling *glLoadIdentity* sets the current matrix to be the identity transform, which represents no change of coordinates at all and is the usual starting point for a series of transformations.

When the function *glPushMatrix*() is called, a copy of the current matrix is pushed onto the stack. Note that this does not change the current matrix; it just saves a copy on the stack. When *glPopMatrix*() is called, the matrix on the top of the stack is popped from the stack, and that matrix replaces the current matrix. Note that *glPushMatrix* and *glPopMatrix* must always occur in corresponding pairs; *glPushMatrix* saves a copy of the current matrix, and a corresponding call to *glPopMatrix* restores that copy. Between a call to *glPushMatrix* and the corresponding call to *glPopMatrix*, there can be additional calls of these functions, as long as they are properly paired. Usually, you will call *glPushMatrix* before drawing an object and *glPopMatrix* after finishing that object. In between, drawing sub-objects might require additional pairs of calls to those functions.

As an example, suppose that we want to draw a cube. It's not hard to draw each face using glBegin/glEnd, but let's do it with transformations. We can start with a function that draws a square in the position of the front face of the cube. For a cube of size 1, the front face would sit one-half unit in front of the screen, in the plane *z* = 0.5, and it would have vertices at (-0.5, -0.5, 0.5), (0.5, -0.5, 0.5), (0.5, 0.5, 0.5), and (-0.5, 0.5, 0.5). Here is a function that draws the square. The function's parameters are floating point numbers in the range 0.0 to 1.0 that give the RGB color of the square:

```
void square( float r, float g, float b ) { glColor3f(r,g,b); // Set the color for the square.

glBegin(GL_TRIANGLE_FAN); glVertex3f(-0.5, -0.5, 0.5); glVertex3f(0.5, -0.5, 0.5);
glVertex3f(0.5, 0.5, 0.5); glVertex3f(-0.5, 0.5, 0.5); glEnd();

}
```

To make a red front face for the cube, we just need to call *square*(1,0,0). Now, consider the right face, which is perpendicular to the *x*-axis, in the plane *x* = 0.5. To make a right face, we can start with a front face and rotate it 90 degrees about the *y*-axis. Think about rotating the front face (red) to the position of the right face (green) in this illustration by rotating the red square about the *y-axis*:



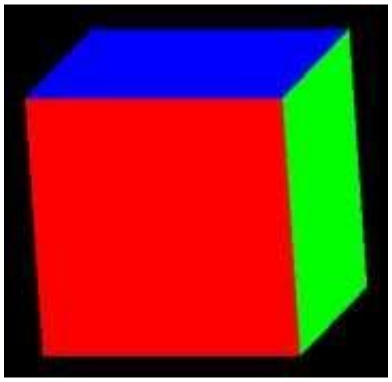
So, we can draw a green right face for the cube with

```
glPushMatrix(); glRotatef(90, 0, 1, 0); square(0, 1, 0); glPopMatrix();
```

The calls to *glPushMatrix* and *glPopMatrix* ensure that the rotation that is applied to the square will not carry over to objects that are drawn later. The other four faces can be made in a similar way, by rotating the front face about the coordinate axes. You should try to visualize the rotation that you need in each case. We can combine it all into a function that draws a cube. To make it more interesting, the size of the cube is a parameter:

```
void cube(float size) { // Draws a cube with side length = size.

glPushMatrix(); // Save a copy of the current matrix. glScalef(size,size,size); // Scale unit cube to desired size.

square(1, 0, 0); // red front face

glPushMatrix(); glRotatef(90, 0, 1, 0); square(0, 1, 0); // green right face glPopMatrix();

glPushMatrix(); glRotatef(-90, 1, 0, 0);

square(0, 0, 1); // blue top face glPopMatrix();

glPushMatrix(); glRotatef(180, 0, 1, 0);

square(0, 1, 1); // cyan back face glPopMatrix();

glPushMatrix(); glRotatef(-90, 0, 1, 0); square(1, 0, 1); // magenta left face
glPopMatrix();

glPushMatrix(); glRotatef(90, 1, 0, 0); square(1, 1, 0); // yellow bottom face
glPopMatrix(); glPopMatrix(); // Restore matrix to its state before cube() was called.

}
```

The sample program *glut/unlit-cube.c* uses this function to draw a cube, and lets you rotate the cube by pressing the arrow keys. A Java version is *jogl/UnlitCube.java*, and a web version is *glsim/unlit-cube.html*. Here is an image of the cube, rotated by 15 degrees about the *x*-axis and -15 degrees about the *y*-axis to make the top and right sides visible:



For a more complex example of hierarchical modeling with *glPushMatrix* and *glPopMatrix*, you can check out an OpenGL equivalent of the "cart and windmills" animation that was used as an example in Subsection 2.4.1. The three versions of the example are: *glut/opengl-cart-and-windmill-2d.c*, *jogl/CartAndWindmillJogl2D.java*, and *glsim/opengl-cartand-windmill.html*. This program is an example of hierarchical 2D graphics in OpenGL.

* * *

Keep in mind that transformation and matrix functions such as *glRotated*() and *glPushMatrix*() are old-fashioned OpenGL. In WebGL and other modern graphics APIs, you will be responsible for managing transforms and matrices on your own. You are quite likely to do that using a software library that provides functions very similar to those that are built into OpenGL 1.1.

3.3 Projection and Viewing

In the previous section, we looked at the modeling transformation, which transforms from object coordinates to world coordinates. However, for 3D computer graphics, you need to know about several other coordinate systems and the transforms between them. We discuss them in this section.

We start with an overview of the various coordinate systems. Some of this is review, and some of it is new.

3.3.1 Many Coordinate Systems

The coordinates that you actually use for drawing an object are called object coordinates. The object coordinate system is chosen to be convenient for the object that is being drawn. A modeling transformation can then be applied to set the size, orientation, and position of the object in the overall scene (or, in the case of hierarchical modeling, in the object coordinate system of a larger, more complex object). The modeling transformation is the first that is applied to the vertices of an object.

The coordinates in which you build the complete scene are called world coordinates. These are the coordinates for the overall scene, the imaginary 3D world that you are creating. The modeling transformation maps from object coordinates to world coordinates.

In the real world, what you see depends on where you are standing and the direction in which you are looking. That is, you can't make a picture of the scene until you know the position of the "viewer" and where the viewer is looking—and, if you think about it, how the viewer's head is tilted. For the purposes of OpenGL, we imagine that the viewer is attached to their own individual coordinate system, which is known as eye coordinates. In this coordinate system, the viewer is at the origin, (0,0,0), looking in the direction of the negative *z*-axis; the positive direction of the *y*-axis is pointing straight up; and the *x*-axis is pointing to the right. This is a viewer-centric coordinate system. In other words, eye coordinates are (almost) the coordinates that you actually want to **use** for drawing on the screen. The transform from world coordinates to eye coordinates is called the viewing transformation.

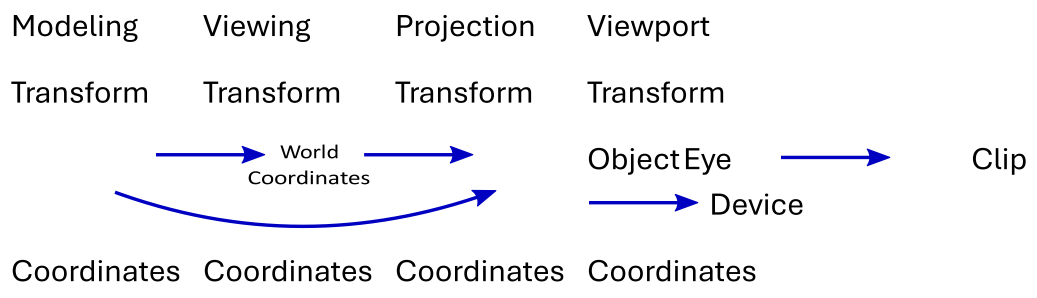
If this is confusing, think of it this way: We are free to use any coordinate system that we want on the world. Eye coordinates are the natural coordinate system for making a picture of the world as seen by a viewer. If we used a different coordinate system (world coordinates) when building the world, then we have to transform those coordinates to eye coordinates to find out what the viewer actually sees. That transformation is the viewing transform.

Note, by the way, that OpenGL doesn't keep track of separate modeling and viewing transforms. They are combined into a single transform, which is known as the modelview transformation. In fact, even though world coordinates might seem to be the most important and natural coordinate system, OpenGL doesn't have any representation for them and doesn't use them internally. For OpenGL, only object and eye coordinates have meaning. OpenGL goes directly from object coordinates to eye coordinates by applying the modelview transformation.

We are not done. The viewer can't see the entire 3D world, only the part that fits into the viewport, which is the rectangular region of the screen or other display device where the image will be drawn. We say that the scene is "clipped" by the edges of the viewport. Furthermore, in OpenGL, the viewer can see only a limited range of *z*-values in the eye coordinate system. Points with larger or smaller *z*-values are clipped away and are not rendered into the image. (This is not, of course, the way that viewing works in the real world, but it's required by the use of the depth test in OpenGL. See Subsection 3.1.4.) The volume of space that is actually rendered into the image is called the view volume. Things inside the view volume make it into the image; things that are not in the view volume are clipped and cannot be seen. For purposes of drawing, OpenGL applies a coordinate transform that maps the view volume onto a **cube**. The cube is centered at the origin and extends from -1 to 1 in the x-direction, in the y-direction, and in the z-direction. The coordinate system on this cube is referred to as clip coordinates. The transformation from eye coordinates to clip coordinates is called the projection transformation. At this point, we haven't quite projected the 3D scene onto a 2D surface, but we can now do so simply by discarding the z-coordinate. (The z-coordinate, however, is still needed to provide the depth information that is needed for the depth test.) Note that clip coordinates are the coordinates will be used if you apply no transformation at all, that is if both the modelview and the projection transforms are the identity. It is a lefthanded coordinate system, with the positive direction of the *z*-axis pointing into the screen.

We **still** aren't done. In the end, when things are actually drawn, there are device coordinates, the 2D coordinate system in which the actual drawing takes place on a physical display device such as the computer screen. Ordinarily, in device coordinates, the pixel is the unit of measure. The drawing region is a rectangle of pixels. This is the rectangle that is called the viewport. The viewport transformation takes x and y from the clip coordinates and scales them to fit the viewport.

Let's go through the sequence of transformations one more time. Think of a primitive, such as a line or triangle, that is part of the scene and that might appear in the image that we want to make of the scene. The primitive goes through the following sequence of operations:



Modelview Transform

1. The points that define the primitive are specified in object coordinates, using methods such as *glVertex3f*.

2. The points are first subjected to the modelview transformation, which is a combination of the modeling transform that places the primitive into the world and the viewing transform that maps the primitive into eye coordinates.

3. The projection transformation is then applied to map the view volume that is visible to the viewer onto the clip coordinate cube. If the transformed primitive lies outside that cube, it will not be part of the image, and the processing stops. If part of the primitive lies inside and part outside, the part that lies outside is clipped away and discarded, and only the part that remains is processed further.

4. Finally, the viewport transform is applied to produce the device coordinates that will actually be used to draw the primitive on the display device. After that, it's just a matter of deciding how to color the individual pixels that are part of the primitive.

We need to consider these transforms in more detail and see how to use them in OpenGL 1.1.

3.3.2 The Viewport Transformation

The simplest of the transforms is the viewport transform. It transforms *x* and *y* clip coordinates to the coordinates that are used on the display device. To specify the viewport transform, it is only necessary to specify the rectangle on the device where the scene will be rendered. This is done using the *glViewport* function.

OpenGL must be provided with a drawing surface by the environment in which it is running, such as JOGL for Java or the GLUT library for C. That drawing surface is a rectangular grid of pixels, with a horizontal size and a vertical size. OpenGL uses a coordinate system on the drawing surface that puts (0,0) at the lower left, with y increasing from bottom to top and x increasing from left to right. When the drawing surface is first given to OpenGL, the viewport is set to be the entire drawing surface. However, it is possible for OpenGL to draw to a different rectangle by calling

```
glViewport( x, y, width, height );
```

where (*x,y*) is the lower left corner of the viewport, in the drawing surface coordinate system, and *width* and *height* are the size of the viewport. Clip coordinates from -1 to 1 will then be mapped to the specified viewport. Note that this means in particular that drawing is limited to the viewport. It is not an error for the viewport to extend outside of the drawing surface, though it would be unusual to set up that situation deliberately.

When the size of the drawing surface changes, such as when the user resizes a window that contains the drawing surface, OpenGL does not automatically change the viewport to match the new size. However, the environment in which OpenGL is running might do that for you.

(See Section 3.6 for information about how this is handled by JOGL and GLUT.) *glViewport* is often used to draw several different scenes, or several views of the same scene, on the same drawing surface. Suppose, for example, that we want to draw two scenes, side-byside, and that the drawing surface is 600-by-400 pixels. An outline for how to do that is very simple:

```
glViewport(0,0,300,400); // Draw to left half of the drawing surface.

.

        .       // Draw the first scene.

. glViewport(300,0,300,400); // Draw to right half of the drawing surface.

.

        .       // Draw the second scene.

.
```

The first *glViewport* command establishes a 300-by-400 pixel viewport with its lower left corner at (0,0). That is, the lower left corner of the viewport is at the lower left corner of the drawing surface. This viewport fills the left half of the drawing surface. Similarly, the second viewport, with its lower left corner at (300,0), fills the right half of the drawing surface.

3.3.3 The Projection Transformation

We turn next to the projection transformation. Like any transform, the projection is represented in OpenGL as a matrix. OpenGL keeps track of the projection matrix separately from the matrix that represents the modelview transformation. The same transform functions, such as *glRotatef*, can be applied to both matrices, so OpenGL needs some way to know which matrix those functions apply to. This is determined by an OpenGL state property called the matrix mode. The value of the matrix mode is a constant such as *GL _PROJECTION* or *GL MODELVIEW*. When a function such as *glRotatef* is called, it modifies a matrix; which matrix is modified depends on the current value of the matrix mode. The value is set by calling the function *glMatrixMode*. The initial value is *GL _MODELVIEW*. This means that if you want to work on the projection matrix, you must first call glMatrixMode(GL _PROJECTION);

If you want to go back to working on the modelview matrix, you must call glMatrixMode(GL _MODELVIEW);

In my programs, I generally set the matrix mode to *GL PROJECTION*, set up the projection transformation, and then immediately set the matrix mode back to *GL MODELVIEW*. This means that anywhere else in the program, I can be sure that the matrix mode is *GL MODELVIEW*.

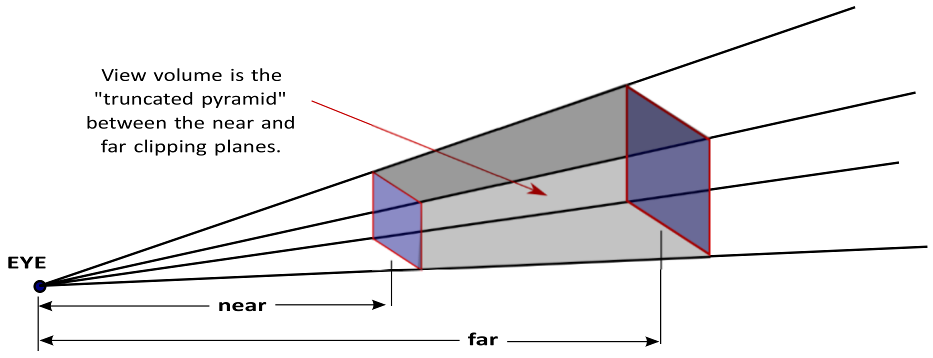
* * *

To help you to understand projection, remember that a 3D image can show only a part of the infinite 3D world. The view volume is the part of the world that is visible in the image. The view volume is determined by a combination of the viewing transformation and the projection transformation. The viewing transform determines where the viewer is located and what direction the viewer is facing, but it doesn't say how much of the world the viewer can see. The projection transform does that: It specifies the shape and extent of the region that is in view. Think of the viewer as a camera, with a big invisible box attached to the front of the camera that encloses the part of the world that that camera has in view. The inside of the box is the view volume. As the camera moves around in the world, the box moves with it, and the view volume changes. But the shape and size of the box don't change. The shape and size of the box correspond to the projection transform. The position and orientation of the camera correspond to the viewing transform.

This is all just another way of saying that, mathematically, the OpenGL projection transformation transforms eye coordinates to clip coordinates, mapping the view volume onto the 2-by-2-by-2 clipping cube that contains everything that will be visible in the image. To specify a projection just means specifying the size and shape of the view volume, relative to the viewer.

There are two general types of projection, perspective projection and orthographic projection. Perspective projection is more physically realistic. That is, it shows what you would see if the OpenGL display rectangle on your computer screen were a window into an actual 3D world (one that could extend in front of the screen as well as behind it). It shows a view that you could get by taking a picture of a 3D world with an ordinary camera. In a perspective view, the apparent size of an object depends on how far it is away from the viewer. Only things that are in front of the viewer can be seen. In fact, ignoring clipping in the *z*direction for the moment, the part of the world that is in view is an infinite pyramid, with the viewer at the apex of the pyramid, and with the sides of the pyramid passing through the sides of the viewport rectangle.

However, OpenGL can't actually show everything in this pyramid, because of its use of the depth test to solve the hidden surface problem. Since the depth buffer can only store a finite range of depth values, it can't represent the entire range of depth values for the infinite pyramid that is theoretically in view. Only objects in a certain range of distances from the viewer can be part of the image. That range of distances is specified by two values, *near* and *far*. For a perspective transformation, both of these values must be positive numbers, and *far* must be greater than *near*. Anything that is closer to the viewer than the *near* distance or farther away than the *far* distance is discarded and does not appear in the rendered image. The volume of space that is represented in the image is thus a "truncated pyramid." This pyramid is the view volume for a perspective projection:



The view volume is bounded by six planes—the four sides plus the top and bottom of the truncated pyramid. These planes are called clipping planes because anything that lies on the wrong side of each plane is clipped away. The projection transformation maps the six sides of the truncated pyramid in eye coordinates to the six sides of the clipping cube in clip coordinates.

In OpenGL, setting up the projection transformation is equivalent to defining the view volume. For a perspective transformation, you have to set up a view volume that is a truncated pyramid. A rather obscure term for this shape is a frustum. A perspective transformation can be set up with the *glFrustum* command:

glFrustum(xmin, xmax, ymin, ymax, near, far);

The last two parameters specify the *near* and *far* distances from the viewer, as already discussed. The viewer is assumed to be at the origin, (0,0,0), facing in the direction of the negative z-axis. (This is the eye coordinate system.) So, the near clipping plane is at $z = -near$, and the far clipping plane is at $z = -far$. (Notice the minus signs!) The first four parameters specify the sides of the pyramid: *xmin*, *xmax*, *ymin*, and *ymax* specify the horizontal and vertical limits of the view volume **at the near clipping plane**. For example, the coordinates of the upper-left corner of the small end of the pyramid are (*xmin*, *ymax*, *-near*). The *x* and *y* limits at the far clipping plane are larger, usually much larger, than the limits specified in the *glFrustum* command.

Note that *x* and *y* limits in *glFrustum* are usually symmetrical about zero. That is, *xmin* is usually equal to the negative of *xmax* and *ymin* is usually equal to the negative of *ymax*. However, this is not required. It is possible to have asymmetrical view volumes where the z-axis does not point directly down the center of the view.

Since the matrix mode must be set to *GL PROJECTION* to work on the projection transformation, *glFrustum* is often used in a code segment of the form

glMatrixMode(GL_PROJECTION); glLoadIdentity();

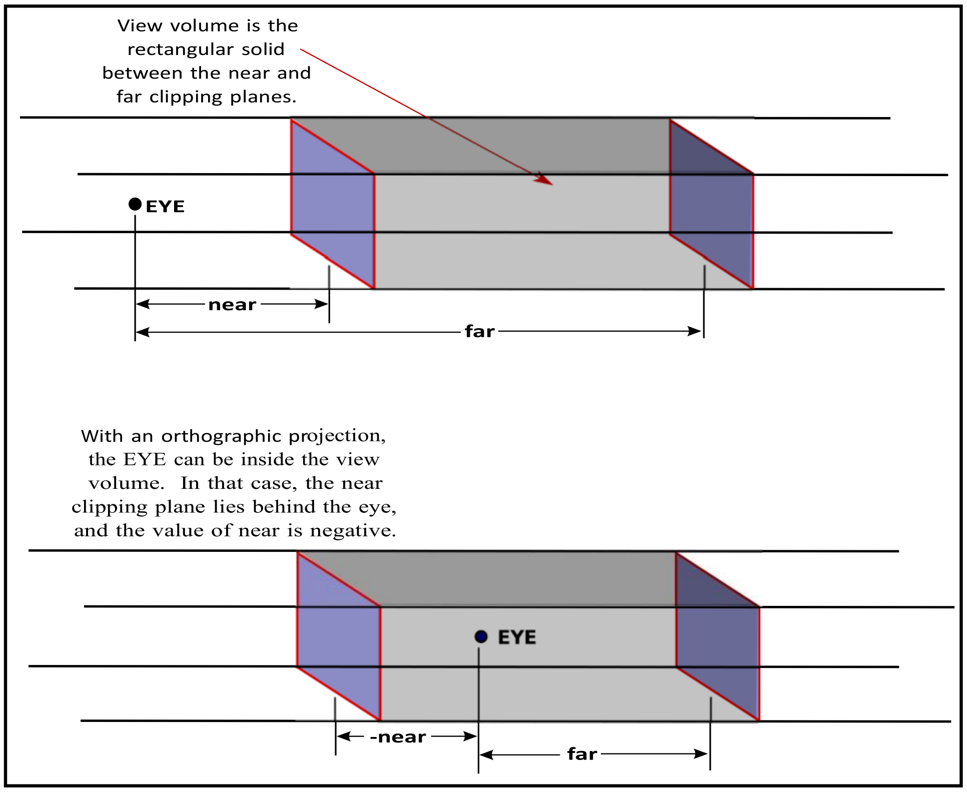
glFrustum(xmin, xmax, ymin, ymax, near, far); glMatrixMode(GL_MODELVIEW);

The call to *glLoadIdentity* ensures that the starting point is the identity transform. This is important since *glFrustum* modifies the existing projection matrix rather than replacing it, and although it is theoretically possible, you don't even want to try to think about what would happen if you combine several projection transformations into one.

* * *

Compared to perspective projections, orthographic projections are easier to understand: In an orthographic projection, the 3D world is projected onto a 2D image by discarding the *z*-coordinate of the eye-coordinate system. This type of projection is unrealistic in that it is not what a viewer would see. For example, the apparent size of an object does not depend on its distance from the viewer. Objects in back of the viewer as well as in front of the viewer can be visible in the image. Orthographic projections are still useful, however, especially in interactive modeling programs where it is useful to see true sizes and angles, undistorted by perspective.

In fact, it's not really clear what it means to say that there is a viewer in the case of orthographic projection. Nevertheless, for orthographic projection in OpenGL, there is considered to be a viewer. The viewer is located at the eye-coordinate origin, facing in the direction of the negative z-axis. Theoretically, a rectangular corridor extending infinitely in both directions, in front of the viewer and in back, would be in view. However, as with perspective projection, only a finite segment of this infinite corridor can actually be shown in an OpenGL image. This finite view volume is a parallelepiped—a rectangular solid—that is cut out of the infinite corridor by a *near* clipping plane and a *far* clipping plane. The value of *far* must be greater than *near*, but for an orthographic projection, the value of *near* is allowed to be negative, putting the "near" clipping plane behind the viewer, as shown in the lower section of this illustration:


View volume is the rectangular solid between the near and far clipping planes.
EYE
near
far
With an orthographic projection, the EYE can be inside the view volume. In that case, the near clipping plane lies behind the eye, and the value of near is negative.
EYE
-near
far

Note that a negative value for *near* puts the near clipping plane on the **positive** *z*-axis, which is behind the viewer.

An orthographic projection can be set up in OpenGL using the *glOrtho* method, which is has the following form:

glOrtho(xmin, xmax, ymin, ymax, near, far);

The first four parameters specify the *x*- and *y*-coordinates of the left, right, bottom, and top of the view volume. Note that the last two parameters are *near* and *far*, not *zmin* and *zmax*.

In fact, the minimum z-value for the view volume is −*far* and the maximum z-value is −*near*. However, it is often the case that *near* = −*far*, and if that is true then the minimum and maximum z-values turn out to be *near* and *far* after all!

As with *glFrustum*, *glOrtho* should be called when the matrix mode is *GL PROJECTION*. As an example, suppose that we want the view volume to be the box centered at the origin containing *x*, *y*, and *z* values in the range from -10 to 10. This can be accomplished with

glMatrixMode(GL_PROJECTION); glLoadIdentity();

glOrtho(-10, 10, -10, 10, -10, 10); glMatrixMode(GL_MODELVIEW);

Now, as it turns out, the effect of *glOrtho* in this simple case is exactly the same as the effect of *glScalef* (0.1, 0.1, -0.1), since the projection just scales the box down by a factor of 10. But it's usually better to think of projection as a different sort of thing from scaling. (The minus sign on the *z* scaling factor is there because projection reverses the direction of the *z*-axis, transforming the conventionally right-handed eye coordinate system into OpenGL's left-handed default coordinate system.)

* * *

The *glFrustum* method is not particularly easy to use. There is a library known as GLU that contains some utility functions for use with OpenGL. The GLU library includes the method *gluPerspective* as an easier way to set up a perspective projection. The command gluPerspective(fieldOfViewAngle, aspect, near, far);

can be used instead of *glFrustum*. The *fieldOfViewAngle* is the vertical angle, measured in degrees, between the upper side of the view volume pyramid and the lower side. Typical values are in the range 30 to 60 degrees. The *aspect* parameter is the aspect ratio of the view, that is, the width of a cross-section of the pyramid divided by its height. The value of *aspect* should generally be set to the aspect ratio of the viewport. The *near* and *far* parameters in *gluPerspective* have the same meaning as for *glFrustum*.

3.3.4 The Modelview Transformation

"Modeling" and "viewing" might seem like very different things, conceptually, but OpenGL combines them into a single transformation. This is because there is no way to distinguish between them in principle; the difference is purely conceptual. That is, a given transformation can be considered to be either a modeling transformation or a viewing transformation, depending on how you think about it. (One significant difference, conceptually, is that the same viewing transformation usually applies to every object in the 3D scene, while each object can have its own modeling transformation. But this is not a difference in principle.) We have already seen something similar in 2D graphics (Subsection 2.3.1), but let's think about how it works in 3D.

For example, suppose that there is a model of a house at the origin, facing towards the direction of the positive *z*-axis. Suppose the viewer is on the positive *z*-axis, looking back towards the origin. The viewer is looking directly at the front of the house. Now, you might apply a modeling transformation to the house, to rotate it by 90 degrees about the *y*-axis. After this transformation, the house is facing in the positive direction of the *x*-axis, and the viewer is looking directly at the **left** side of the house. On the other hand, you might rotate the **viewer** by **minus** 90 degrees about the *y*-axis. This would put the viewer on the negative *x*-axis, which would give it a view of the **left** side of the house. The net result after either transformation is that the viewer ends up with exactly the same view of the house. Either transformation can be implemented in OpenGL with the command

glRotatef(90,0,1,0);

That is, this command represents either a modeling transformation that rotates an object by 90 degrees or a viewing transformation that rotates the viewer by -90 degrees about the *y*-axis. Note that the effect on the viewer is the inverse of the effect on the object. Modeling and viewing transforms are always related in this way. For example, if you are looking at an object, you can move yourself 5 feet to the **left** (viewing transform), or you can move the object 5 feet to the **right** (modeling transform). In either case, you end up with the same view of the object. Both transformations would be represented in OpenGL as glTranslatef(5,0,0);

This even works for scaling: If the viewer *shrinks*, it will look to the viewer exactly the same as if the world is expanding, and vice-versa.

* * *

Although modeling and viewing transformations are the same in principle, they remain very different conceptually, and they are typically applied at different points in the code. In general when drawing a scene, you will do the following: (1) Load the identity matrix, for a well-defined starting point; (2) apply the viewing transformation; and (3) draw the objects in the scene, each with its own modeling transformation. Remember that OpenGL keeps track of several transformations, and that this must all be done while the modelview transform is current; if you are not sure of that then before step (1), you should call *glMatrixMode*(*GL MODELVIEW*). During step (3), you will probably use *glPushMatrix*() and *glPopMatrix*() to limit each modeling transform to a particular object.

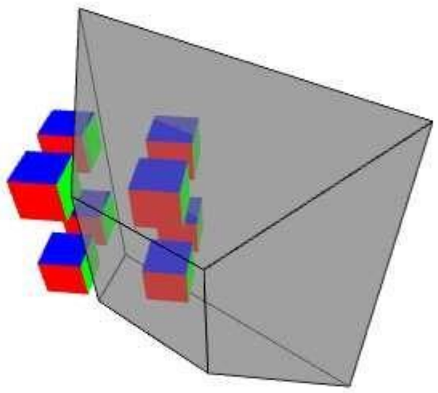
After loading the identity matrix, the viewer is in the default position, at the origin, looking down the negative *z*-axis, with the positive *y*-axis pointing upwards in the view. Suppose, for example, that we would like to move the viewer from its default location at the origin back along the positive z-axis to the point (0,0,20). This operation has exactly the same effect as moving the world, and the objects that it contains, 20 units in the negative direction along the z-axis. Whichever operation is performed, the viewer ends up in exactly the same position relative to the objects. Both operations are implemented by the same OpenGL command, *glTranslatef* (0,0,-20). For another example, suppose that we use two commands

```
glRotatef(90,0,1,0); glTranslatef(10,0,0);
```

to establish the viewing transformation. As a modeling transform, these commands would first translate an object 10 units in the positive *x*-direction, then rotate the object 90 degrees about the *y*-axis. This would move an object originally at (0,0,0) to (0,0,-10), placing the object 10 units directly in front of the viewer. (Remember that modeling transformations are applied to objects in the order opposite to their order in the code.) What do these commands do as a viewing transformation? The effect on the view is the inverse of the effect on objects. The inverse of “translate 90 then rotate 10” is “rotate -10 then translate -90.” That is, to do the inverse, you have to undo the rotation **before** you undo the translation. The effect as a viewing transformation is first to rotate the view by -90 degrees about the *y*-axis (which would leave the viewer at the origin, but now looking along the positive *x*-axis), then to translate the viewer by -10 along the *x*-axis (backing up the viewer to the point (-10,0,0)). An object at the point (0,0,0) would thus be 10 units directly in front of the viewer. (You should think about how the two interpretations affect the view of a house that starts out at (0,0,0). The transformation affects which side of the house the viewer is looking at, as well as how far away from the house the viewer is located).

Note, by the way, that the order in which viewing transformations are applied is the **same as** the order in which they occur in the code.

The on-line version of this section includes the live demo *c3/transform-equivalence-3d.html* that can help you to understand the equivalence between modeling and viewing. This picture, taken from that demo, visualizes the view volume as a translucent gray box. The scene contains eight cubes, but not all of them are inside the view volume, so not all of them would appear in the rendered image:



In this case, the projection is a perspective projection, and the view volume is a frustum. This picture might have been made either by rotating the frustum towards the right (viewing transformation) or by rotating the cubes towards the left (modeling transform). Read the help text in the demo for more information

It can be difficult to set up a view by combining rotations, scalings, and translations, so OpenGL provides an easier way to set up a typical view. The command is not part of OpenGL itself but is part of the GLU library.

The GLU library provides the following convenient method for setting up a viewing transformation:

gluLookAt(eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ);

This method places the viewer at the point (*eyeX*,*eyeY*,*eyeZ*), looking towards the point (*refX*,*refY*,*refZ*). The viewer is oriented so that the vector (*upX*,*upY*,*upZ*) points upwards in the viewer's view. For example, to position the viewer on the negative *x*-axis, 10 units from the origin, looking back at the origin, with the positive direction of the *y*-axis pointing up as usual, use gluLookAt(-10,0,0, 0,0,0, 0,1,0);

* * *

With all this, we can give an outline for a typical display routine for drawing an image of a 3D scene with OpenGL 1.1:

// possibly set clear color here, if not set elsewhere glClear(GL COLOR _BUFFER BIT | GL DEPTH BUFFER BIT);

```
// possibly set up the projection here, if not done elsewhere glMatrixMode( GL
MODELVIEW ); glLoadIdentity(); gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ
); // Viewing transform

glPushMatrix(); .

. // apply modeling transform and draw an object

. glPopMatrix();

glPushMatrix(); .

. // apply another modeling transform and draw another object .

glPopMatrix();

.

.

.
```

3.3.5 A Camera Abstraction

Projection and viewing are often discussed using the analogy of a camera. A real camera is used to take a picture of a 3D world. For 3D graphics, it useful to imagine using a virtual camera to do the same thing. Setting up the viewing transformation is like positioning and pointing the camera. The projection transformation determines the properties of the camera: What is its field of view, what sort of lens does it use? (Of course, the analogy breaks for OpenGL in at least one respect, since a real camera doesn't do clipping in its *z*-direction.)

I have written a camera utility to implement this idea. The camera is meant to take over the job of setting the projection and view. Instead of doing that by hand, you set properties of the camera. The API is available for both C and Java. The two versions are somewhat different because the Java version is object-oriented. I will discuss the C implementation first. (See Section 3.6 for basic information about programming OpenGL in C and Java. For an example of using a camera in a program, see the polyhedron viewer example in the next section. Note also that there is a version of the camera for use with my JavaScript simulator for OpenGL; it is part of the simulator library *glsim/glsim.js* and has an API almost identical to the Java API.)

In C, the camera is defined by the sample .c file, *glut/camera.c* and a corresponding header file, *glut/camera.h*. Full documentation for the API can be found in the header file. To use the camera, you should #include "camera.h" at the start of your program, and when you compile the program, you should include *camera.c* in the list of files that you want to compile. The camera depends on the GLU library and on C's standard math library, so you have to make sure that those libraries are available when it is compiled. To use the camera, you should call cameraApply();

to set up the projection and viewing transform before drawing the scene. Calling this function replaces the usual code for setting up the projection and viewing transformations. It leaves the matrix mode set to *GL MODELVIEW*.

The remaining functions in the API are used to configure the camera. This would usually be done as part of initialization, but it is possible to change the configuration at any time. However, remember that the settings are not used until you call *cameraApply*(). Available functions include:

cameraLookAt(eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ); // Determines the viewing transform, just like gluLookAt // Default is cameraLookAt(0,0,30, 0,0,0, 0,1,0);

cameraSetLimits(xmin, xmax, ymin, ymax, zmin, zmax);

// Sets the limits on the view volume, where zmin and zmax are

// given with respect to the view reference point, NOT the eye, // and the xy limits are measured at the distance of the // view reference point, NOT the near distance.

// Default is cameraSetLimits(-5,5, -5,5, -10,10);

cameraSetScale(limit);

// a convenience method, which is the same as calling

// cameraSetLimits(-limit,limit, -limit,limit, -2*limit, 2*limit);

cameraSetOrthographic(ortho);

// Switch between orthographic and perspective projection. // The parameter should be 0 for perspective, 1 for // orthographic. The default is perspective.

cameraSetPreserveAspect(preserve);

// Determine whether the aspect ratio of the viewport should

// be respected. The parameter should be 0 to ignore and // 1 to respect the viewport aspect ratio. The default // is to preserve the aspect ratio.

In many cases, the default settings are sufficient. Note in particular how *cameraLookAt* and *cameraSetLimits* work together to set up the view and projection. The parameters to *cameraLookAt* represent three points in world coordinates. The view reference point, (*refX,refY,refZ*), should be somewhere in the middle of the scene that you want to render. The parameters to *cameraSetLimits* define a box about that view reference point that should contain everything that you want to appear in the image.

* * *

For use with JOGL in Java, the camera API is implemented as a class named Camera, defined in the file *jogl/Camera.java*. The camera is meant for use with a GLPanel or GLCanvas that is being used as an OpenGL drawing surface. To use a camera, create an object of type Camera as an instance variable:

camera = new Camera();

Before drawing the scene, call camera.apply(gl2);

where *gl2* is the OpenGL drawing context of type *GL2*. (Note the presence of the parameter *gl2*, which was not necessary in C; it is required because the OpenGL drawing context in JOGL is implemented as an object.) As in the C version, this sets the viewing and projection transformations and can replace any other code that you would use for that purpose. The functions for configuring the camera are the same in Java as in C, except that they become methods in the *camera* object, and true/false parameters are *boolean* instead of *int*:

camera.lookAt(eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ); camera.setLimits(xmin,xmax, ymin,ymax, zmin,zmax); camera.setScale(limit);

camera.setOrthographic(ortho);   // ortho is of type boolean
camera.setPreserveAspect(preserve); // preserve is of type boolean

* * *

The camera comes with a simulated "trackball." The trackball allows the user to rotate the view by clicking and dragging the mouse on the display. To use it with GLUT in C, you just need to install a mouse function and a mouse motion function by calling

glutMouseFunc(trackballMouseFunction); glutMotionFunc(trackballMotionFunction);

The functions *trackballMouseFunction* and *trackballMotionFunction* are defined as part of the camera API and are declared and documented in *camera.h*. The trackball works by modifying the viewing transformation associated with the camera, and it only works if *cameraApply*() is called at the beginning of the display function to set the viewing and projection transformations. To install a trackball for use with a Camera object in JOGL, call camera.installTrackball(drawSurface);

where *drawSurface* is the component on which the camera is used.

3.4 Polygonal Meshes and glDrawArrays

We have drawn only very simple shapes with OpenGL. In this section, we look at how more complex shapes can be represented in a way that is convenient for rendering in OpenGL, and we introduce a new, more efficient way to draw OpenGL primitives.

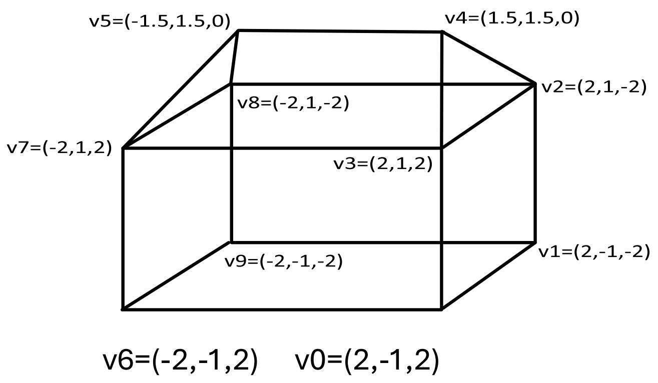
OpenGL can only directly render points, lines, and polygons. (In fact, in modern OpenGL, the only polygons that are used are triangles.) A polyhedron, the 3D analog of a polygon, can be represented exactly, since a polyhedron has faces that are polygons. On the other hand, if only polygons are available, then a curved surface, such as the surface of a sphere, can only be approximated. A polyhedron can be represented, or a curved surface can be approximated, as a polygonal mesh, that is, a set of polygons that are connected along their edges. If the polygons are small, the approximation can look like a curved surface. (We will see in the next chapter how lighting effects can be used to make a polygonal mesh look more like a curved surface and less like a polyhedron.)

So, our problem is to represent a set of polygons—most often a set of triangles. We start by defining a convenient way to represent such a set as a data structure.

3.4.1 Indexed Face Sets

The polygons in a polygonal mesh are also referred to as “faces” (as in the faces of a polyhedron), and one of the primary means for representing a polygonal mesh is as an indexed face set, or IFS.

The data for an IFS includes a list of all the vertices that appear in the mesh, giving the coordinates of each vertex. A vertex can then be identified by an integer that specifies its index, or position, in the list. As an example, consider this “house,” a polyhedron with 10 vertices and 9 faces:



The vertex list for this polyhedron has the form

Vertex #0. (2, -1, 2)

Vertex #1. (2, -1, -2)

Vertex #2. (2, 1, -2)

Vertex #3. (2, 1, 2)

Vertex #4. (1.5, 1.5, 0)

Vertex #5. (-1.5, 1.5, 0)

Vertex #6. (-2, -1, 2)

Vertex #7. (-2, 1, 2)

Vertex #8. (-2, 1, -2)

Vertex #9. (-2, -1, -2)

The order of the vertices is completely arbitrary. The purpose is simply to allow each vertex to be identified by an integer.

To describe one of the polygonal faces of a mesh, we just have to list its vertices, in order going around the polygon. For an IFS, we can specify a vertex by giving its index in the list. For example, we can say that one of the triangular faces of the pyramid is the polygon formed by vertex #3, vertex #2, and vertex #4. So, we can complete our data for the mesh by giving a list of vertex indices for each face. Here is the face data for the house. Remember that the numbers in parentheses are indices into the vertex list:

Face #0: (0, 1, 2, 3)

Face #1: (3, 2, 4)

Face #2: (7, 3, 4, 5)

Face #3: (2, 8, 5, 4)

Face #4: (5, 8, 7)

Face #5: (0, 3, 7, 6)

Face #6: (0, 6, 9, 1)

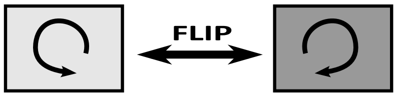
Face #7: (2, 1, 9, 8)

Face #8: (6, 7, 8, 9)

Again, the order in which the faces are listed in arbitrary. There is also some freedom in how the vertices for a face are listed. You can start with any vertex. Once you've picked a starting vertex, there are two possible orderings, corresponding to the two possible directions in which you can go around the circumference of the polygon. For example, starting with vertex 0, the first face in the list could be specified either as (0,1,2,3) or as (0,3,2,1). However, the first possibility is the right one in this case, for the following reason. A polygon in 3D can be viewed from either side; we can think of it as having two faces, facing in opposite directions. It turns out that it is often convenient to consider one of those faces to be the "front face" of the polygon and one to be the "back face." For a polyhedron like the house, the front face is the one that faces the outside of the polyhedron. The usual rule is that the vertices of a polygon should be listed in counterclockwise order when looking at the front face of the polygon. When looking at the back face, the vertices will be listed in clockwise order. This is the default rule used by OpenGL.

Front face. Back face.

2 1 1 2



3 0 0 3

Vertex order 0,1,2,3 is counter-clockwise from the front and clockwise from the back

The vertex and face data for an indexed face set can be represented as a pair of twodimensional arrays. For the house, in a version for Java, we could use

```
double[][] vertexList =

{ {2,-1,2}, {2,-1,-2}, {2,1,-2}, {2,1,2}, {1.5,1.5,0},

{-1.5,1.5,0}, {-2,-1,2}, {-2,1,2}, {-2,1,-2}, {-2,-1,-2} };

int[][] faceList =

{ {0,1,2,3}, {3,2,4}, {7,3,4,5}, {2,8,5,4}, {5,8,7},

{0,3,7,6}, {0,6,9,1}, {2,1,9,8}, {6,7,8,9} };
```

In most cases, there will be additional data for the IFS. For example, if we want to color the faces of the polyhedron, with a different color for each face, then we could add another array, *faceColors*, to hold the color data. Each element of *faceColors* would be an array of three double values in the range 0.0 to 1.0, giving the RGB color components for one of the faces. With this setup, we could use the following code to draw the polyhedron, using Java and JOGL:

```
for (int i = 0; i < faceList.length; i++) { gl2.glColor3dv( faceColors[i], 0 ); // Set color for
face number i.

gl2.glBegin(GL2.GL _TRIANGLE _FAN); for (int j = 0; j < faceList[i].length; j++) { int
vertexNum = faceList[i][j]; // Index for vertex j of face i. double[] vertexCoords =
vertexList[vertexNum]; // The vertex itself.

gl2.glVertex3dv( vertexCoords, 0 );

}

gl2.glEnd();

}
```

Note that every vertex index is used three or four times in the face data. With the IFS representation, a vertex is represented in the face list by a single integer. This representation uses less memory space than the alternative, which would be to write out the vertex in full each time it occurs in the face data. For the house example, the IFS representation uses 64 numbers to represent the vertices and faces of the polygonal mesh, as opposed to 102 numbers for the alternative representation.

Indexed face sets have another advantage. Suppose that we want to modify the shape of the polygon mesh by moving its vertices. We might do this in each frame of an animation, as a way of "morphing" the shape from one form to another. Since only the positions of the vertices are changing, and not the way that they are connected together, it will only be necessary to update the 30 numbers in the vertex list. The values in the face list will remain unchanged.

* * *

There are other ways to store the data for an IFS. In C, for example, where two-dimensional arrays are more problematic, we might use one dimensional arrays for the data. In that case, we would store all the vertex coordinates in a single array. The length of the vertex array would be three times the number of vertices, and the data for vertex number *N* will begin at index 3**N* in the array. For the face list, we have to deal with the fact that not all faces have the same number of vertices. A common solution is to add a -1 to the array after the data for each face. In C, where it is not possible to determine the length of an array, we also need variables to store the number of vertices and the number of faces. Using this representation, the data for the house becomes:

```
int vertexCount = 10; // Number of vertices.

double vertexData[] =

{ 2,-1,2, 2,-1,-2, 2,1,-2, 2,1,2, 1.5,1.5,0,

-1.5,1.5,0, -2,-1,2, -2,1,2, -2,1,-2, -2,-1,-2 };

int faceCount = 9; // Number of faces.

int[][] faceData =

{ 0,1,2,3,-1, 3,2,4,-1, 7,3,4,5,-1, 2,8,5,4,-1, 5,8,7,-1,

0,3,7,6,-1, 0,6,9,1,-1, 2,1,9,8,-1, 6,7,8,9,-1 };
```

After adding a *faceColors* array to hold color data for the faces, we can use the following C code to draw the house:

```
int i,j;

j = 0; // index into the faceData array for (i = 0; i < faceCount; i++) { glColor3dv(
&faceColors[ i*3 ] ); // Color for face number i.

glBegin(GL _TRIANGLE _FAN); while ( faceData[j] != -1) { // Generate vertices for face
number i. int vertexNum = faceData[j]; // Vertex number in vertexData array.

glVertex3dv( &vertexData[ vertexNum*3 ] ); j++;

}

j++; // increment j past the -1 that ended the data for this face. glEnd();

}
```

Note the use of the C address operator, &. For example, &faceColors[i*3] is a pointer to element number *i*3* in the *faceColors* array. That element is the first of the three color component values for face number *i*. This matches the parameter type for *glColor3dv* in C, since the parameter is a pointer type.

* * *

We could easily draw the edges of the polyhedron instead of the faces simply by using *GL LINE _LOOP* instead of *GL _TRIANGLE _FAN* in the drawing code (and probably leaving out the color changes). An interesting issue comes up if we want to draw both the faces and the edges. This can be a nice effect, but we run into a problem with the depth test: Pixels along the edges lie at the same depth as pixels on the faces. As discussed in Subsection 3.1.4, the depth test cannot handle this situation well. However, OpenGL has a solution: a feature called “polygon offset.” This feature can adjust the depth, in clip coordinates, of a polygon, in order to avoid having two objects exactly at the same depth. To apply polygon offset, you need to set the amount of offset by calling

```
glPolygonOffset(1,1);
```

The second parameter gives the amount of offset, in units determined by the first parameter. The meaning of the first parameter is somewhat obscure; a value of 1 seems to work in all cases. You also have to enable the *GL _POLYGON _OFFSET _FILL* feature while drawing the faces. An outline for the procedure is

```
glPolygonOffset(1,1); glEnable( GL POLYGON OFFSET FILL ); .

. // Draw the faces.
```

.

glDisable(GL POLYGON OFFSET _FILL); .

. // Draw the edges.

.

There is a sample program that can draw the house and a number of other polyhedra. It uses drawing code very similar to what we have looked at here, including polygon offset. The program is also an example of using the camera and trackball API that was discussed in Subsection 3.3.5, so that the user can rotate a polyhedron by dragging it with the mouse. The program has menus that allow the user to turn rendering of edges and faces on and off, plus some other options. The Java version of the program is *jogl/IFSPolyhedronViewer.java*, and the C version is *glut/ifs-polyhedron-viewer.c*. To get at the menu in the C version, right-click on the display. The data for the polyhedra are created in *jogl/Polyhedron.java* and *glut/polyhedron.c*. There is also a live demo version of the program in this section on line.

3.4.2 glDrawArrays and glDrawElements

All of the OpenGL commands that we have seen so far were part of the original OpenGL 1.0. OpenGL 1.1 added some features to increase performance. One complaint about the original OpenGL was the large number of function calls needed to draw a primitive using functions such as *glVertex2d* and *glColor3fv* with *glBegin/glEnd*. To address this issue, OpenGL 1.1 introduced the functions *glDrawArrays* and *glDrawElements*. These functions are still used in modern OpenGL, including WebGL. We will look at *glDrawArrays* first. There are some differences between the C and the Java versions of the API. We consider the C version first and will deal with the changes necessary for the Java version in the next subsection.

When using *glDrawArrays*, all of the data that is needed to draw a primitive, including vertex coordinates, colors, and other vertex attributes, can be packed into arrays. Once that is done, the primitive can be drawn with a single call to *glDrawArrays*. Recall that a primitive such as a *GL LINE LOOP* or a *GL TRIANGLES* can include a large number of vertices, so that the reduction in the number of function calls can be substantial.

To use *glDrawArrays*, you must store all of the vertex coordinates for a primitive in a single one-dimensional array. You can use an array of int, float, or double, and you can have 2, 3, or 4 coordinates for each vertex. The data in the array are the same numbers that you would pass as parameters to a function such as *glVertex3f*, in the same order.

You need to tell OpenGL where to find the data by calling void glVertexPointer(int size, int type, int stride, void* array)

The *size* parameter is the number of coordinates per vertex. (You have to provide the same number of coordinates for each vertex.) The *type* is a constant that tells the data type of each of the numbers in the array. The possible values are *GL FLOAT*, *GL _INT*, and *GL _DOUBLE*. The constant that you provide here must match the data type of the numbers in the array. The *stride* is usually 0, meaning that the data values are stored in consecutive locations in the array; if that is not the case, then *stride* gives the distance **in bytes** between the location of the data for one vertex and location for the next vertex. (This would allow you to store other data, along with the vertex coordinates, in the same array.) The final parameter is the array that contains the data. It is listed as being of type "*void**", which is a C data type for a pointer that can point to any type of data. (Recall that an array variable in C is a kind of pointer, so you can just pass an array variable as the fourth parameter.) For example, suppose that we want to draw a square in the *xy*-plane. We can set up the vertex array with float coords[8] = { -0.5,-0.5, 0.5,-0.5, 0.5,0.5, -0.5,0.5 }; glVertexPointer(2, GL FLOAT, 0, coords);

In addition to setting the location of the vertex coordinates, you have to enable use of the array by calling glEnableClientState(GL _VERTEX ARRAY);

OpenGL ignores the vertex pointer except when this state is enabled. You can use *glDisableClientState* to disable use of the vertex array. Finally, in order to actually draw the primitive, you would call the function void glDrawArrays(int primitiveType, int firstVertex, int vertexCount)

This function call corresponds to one use of glBegin/glEnd. The *primitiveType* tells which primitive type is being drawn, such as *GL QUADS* or *GL TRIANGLE _STRIP*. The same ten primitive types that can be used with *glBegin* can be used here. The parameter *firstVertex* is the number of the first vertex that is to be used for drawing the primitive. Note that the position is given in terms of vertex number; the corresponding array index would be the vertex number times the number of coordinates per vertex, which was set in the call to *glVertexPointer*. The *vertexCount* parameter is the number of vertices to be used, just as if *glVertex** were called *vertexCount* times. Often, *firstVertex* will be zero, and *vertexCount* will be the total number of vertices in the array. The command for drawing the square in our example would be glDrawArrays(GL TRIANGLE _FAN, 0, 4);

Often there is other data associated with each vertex in addition to the vertex coordinates. For example, you might want to specify a different color for each vertex.

The colors for the vertices can be put into another array. You have to specify the location of the data by calling void glColorPointer(int size, int type, int stride, void* array)

which works just like *gVertexPointer*. And you need to enable the color array by calling glEnableClientState(GL _COLOR ARRAY);

With this setup, when you call *glDrawArrays*, OpenGL will pull a color from the color array for each vertex at the same time that it pulls the vertex coordinates from the vertex array. Later, we will encounter other kinds of vertex data besides coordinates and color that can be dealt with in much the same way.

Let's put this together to draw the standard OpenGL red/green/blue triangle, which we drew using *glBegin/glEnd* in Subsection 3.1.2. Since the vertices of the triangle have different colors, we will use a color array in addition to the vertex array.

float coords[6] = { -0.9,-0.9, 0.9,-0.9, 0,0.7 }; // two coords per vertex. float colors[9] = { 1,0,0, 0,1,0, 1,0,0 }; // three RGB values per vertex.

glVertexPointer(2, GL FLOAT, 0, coords); // Set data type and location. glColorPointer(3, GL FLOAT, 0, colors);

glEnableClientState(GL VERTEX _ARRAY); // Enable use of arrays. glEnableClientState(GL COLOR ARRAY); glDrawArrays(GL TRIANGLES, 0, 3); // Use 3 vertices, starting with vertex 0.

In practice, not all of this code has to be in the same place. The function that does the actual drawing, *glDrawArrays*, must be in the display routine that draws the image. The rest could be in the display routine, but could also be done, for example, in an initialization routine.

* * *

The function *glDrawElements* is similar to *glDrawArrays*, but it is designed for use with data in a format similar to an indexed face set. With *glDrawArrays*, OpenGL pulls data from the enabled arrays in order, vertex 0, then vertex 1, then vertex 2, and so on. With *glDrawElements*, you provide a list of vertex numbers. OpenGL will go through the list of vertex numbers, pulling data for the specified vertices from the arrays. The advantage of this comes, as with indexed face sets, from the fact that the same vertex can be reused several times.

To use *glDrawElements* to draw a primitive, you need an array to store the vertex numbers. The numbers in the array can be 8, 16, or 32 bit integers. (They are supposed to be unsigned integers, but arrays of regular positive integers will also work.) You also need arrays to store the vertex coordinates and other vertex data, and you must enable those arrays in the same way as for *glDrawArrays*, using functions such as *glVertexArray* and *glEnableClientState*. To actually draw the primitive, call the function void glDrawElements(int primitiveType, vertexCount, dataType, void *array)

Here, *primitiveType* is one of the ten primitive types such as *GL LINES*, *vertexCount* is the number of vertices to be drawn, *dataType* specifies the type of data in the array, and *array* is the array that holds the list of vertex numbers. The *dataType* must be given as one of the constants *GL UNSIGNED _BYTE*, *GL _UNSIGNED SHORT*, or *GL _UNSIGNED _INT* to specify 8, 16, or

32 bit integers respectively.

As an example, we can draw a cube. We can draw all six faces of the cube as one primitive of type *GL QUADS*. We need the vertex coordinates in one array and the vertex numbers for the faces in another array. I will also use a color array for vertex colors. The vertex colors will be interpolated to pixels on the faces, just like the red/green/blue triangle. Here is code that could be used to draw the cube. Again, all this would not necessarily be in the same part of a program:

```
float vertexCoords[24] = { // Coordinates for the vertices of a cube.

        1,1,1,  1,1,-1, 1,-1,-1,1,-1,1,

-1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1 };

float vertexColors[24] = { // An RGB color value for each vertex

        1,1,1,  1,0,0,  1,1,0,  0,1,0,

        0,0,1,  1,0,1,  0,0,0,  0,1,1 };

int elementArray[24] = { // Vertex numbers for the six faces.

0,1,2,3, 0,3,7,4, 0,4,5,1,

6,2,1,5, 6,5,4,7, 6,7,3,2 };

glVertexPointer( 3, GL FLOAT, 0, vertexCoords ); glColorPointer( 3, GL FLOAT, 0,
vertexColors );

glEnableClientState( GL VERTEX _ARRAY ); glEnableClientState( GL COLOR ARRAY );
glDrawElements( GL QUADS, 24, GL UNSIGNED INT, elementArray );
```

Note that the second parameter is the number of vertices, not the number of quads.

The sample program *glut/cubes-with-vertex-arrays.c* uses this code to draw a cube. It draws a second cube using *glDrawArrays*. The Java version is *jogl/CubesWithVertexArrays.java*, but you need to read the next subsection before you can understand it. There is also a JavaScript version, *glsim/cubes-with-vertex-arrays.html*.

3.4.3 Data Buffers in Java

Ordinary Java arrays are not suitable for use with *glDrawElements* and *glDrawArrays*, partly because of the format in which data is stored in them and partly because of inefficiency in transfer of data between Java arrays and the Graphics Processing Unit. These problems are solved by using direct nio buffers. The term “nio” here refers to the package *java.nio*, which contains classes for input/output. A “buffer” in this case is an object of the class java.nio.Buffer or one of its subclasses, such as FloatBuffer or IntBuffer. Finally, “direct” means that the buffer is optimized for direct transfer of data between memory and other devices such as the GPU. Like an array, an nio buffer is a numbered sequence of elements, all of the same type. A FloatBuffer, for example, contains a numbered sequence of values of type float. There are subclasses of Buffer for all of Java’s primitive data types except boolean.

Nio buffers are used in JOGL in several places where arrays are used in the C API. For example, JOGL has the following *glVertexPointer* method in the GL2 class:

public void glVertexPointer(int size, int type, int stride, Buffer buffer)

Only the last parameter differs from the C version. The buffer can be of type FloatBuffer, IntBuffer, or DoubleBuffer. The type of buffer must match the *type* parameter in the method. Functions such as *glColorPointer* work the same way, and *glDrawElements* takes the form

public void glDrawElements(int primitiveType, vertexCount, dataType, Buffer buffer)

where the *buffer* can be of type IntBuffer, ShortBuffer, or ByteBuffer to match the *dataType UNSIGNED _INT*, *UNSIGNED _SHORT*, or *UNSIGNED _BYTE*.

The class com.jogamp.common.nio.Buffers contains static utility methods for working with direct nio buffers. The easiest to use are methods that create a buffer from a Java array. For example, the method *Buffers.newDirectFloatBuffer*(*array*) takes a float array as its parameter and creates a FloatBuffer of the same length and containing the same data as the array. These methods are used to create the buffers in the sample program *jogl/CubesWithVertexArrays.java*.

For example,

float[] vertexCoords = { // Coordinates for the vertices of a cube.

1,1,1,   1,1,-1, 1,-1,-1,1,-1,1,

-1,1,1, -1,1,-1, -1,-1,-1, -1,-1,1 };

int[] elementArray = { // Vertex numbers for the six faces.

0,1,2,3, 0,3,7,4, 0,4,5,1,

6,2,1,5, 6,5,4,7, 6,7,3,2 };

// Buffers for use with glVertexPointer and glDrawElements:

FloatBuffer vertexCoordBuffer = Buffers.newDirectFloatBuffer(vertexCoords); IntBuffer elementBuffer = Buffers.newDirectIntBuffer(elementArray); The buffers can then be used when drawing the cube:

gl2.glVertexPointer(3, GL2.GL FLOAT, 0, vertexCoordBuffer); gl2.glDrawElements(GL2.GL _QUADS, 24, GL2.GL _UNSIGNED INT, elementBuffer);

There are also methods such as *Buffers.newDirectFloatBuffer*(*n*), which creates a FloatBuffer of length *n*. Remember that an nio Buffer, like an array, is simply a linear sequence of elements of a given type. In fact, just as for an array, it is possible to refer to items in a buffer by their index or position in that sequence. Suppose that *buffer* is a variable of type FloatBuffer, *i* is an int and *x* is a float. Then buffer.put(i,x);

copies the value of *x* into position number *i* in the buffer. Similarly, *buffer.get*(*i*) can be used to retrieve the value at index *i* in the buffer. These methods make it possible to work with buffers in much the same way that you can work with arrays.

3.4.4 Display Lists and VBOs

All of the OpenGL drawing commands that we have considered so far have an unfortunate inefficiency when the same object is going be drawn more than once: The commands and data for drawing that object must be transmitted to the GPU each time the object is drawn. It should be possible to store information on the GPU, so that it can be reused without retransmitting it. We will look at two techniques for doing this: display lists and vertex buffer objects (VBOs). Display lists were part of the original OpenGL 1.0, but they are not part of the modern OpenGL API. VBOs were introduced in OpenGL 1.5 and are still important in modern OpenGL; we will discuss them only briefly here and will consider them more fully when we get to WebGL.

Display lists are useful when the same sequence of OpenGL commands will be used several times. A display list is a list of graphics commands and the data used by those commands. A display list can be stored in a GPU. The contents of the display list only have to be transmitted once to the GPU. Once a list has been created, it can be “called.” The key point is that calling a list requires only one OpenGL command. Although the same list of commands still has to be executed, only one command has to be transmitted from the CPU to the graphics card, and then the full power of hardware acceleration can be used to execute the commands at the highest possible speed.

Note that calling a display list twice can result in two different effects, since the effect can depend on the OpenGL state at the time the display list is called. For example, a display list that generates the geometry for a sphere can draw spheres in different locations, as long as different modeling transforms are in effect each time the list is called. The list can also produce spheres of different colors, as long as the drawing color is changed between calls to the list.

If you want to use a display list, you first have to ask for an integer that will identify that list to the GPU. This is done with a command such as listID = glGenLists(1);

The return value is an int which will be the identifier for the list. The parameter to *glGenLists* is also an int, which is usually 1. (You can actually ask for several list IDs at once; the parameter tells how many you want. The list IDs will be consecutive integers, so that if *listA* is the return value from *glGenLists*(3), then the identifiers for the three lists will be *listA*, *listA* + 1, and *listA* + 2.)

Once you’ve allocated a list in this way, you can store commands into it. If *listID* is the ID for the list, you would do this with code of the form:

glNewList(listID, GL COMPILE);

... // OpenGL commands to be stored in the list. glEndList();

The parameter *GL COMPILE* means that you only want to store commands into the list, not execute them. If you use the alternative parameter *GL COMPILE AND EXECUTE*, then the commands will be executed immediately as well as stored in the list for later reuse.

Once you have created a display list in this way, you can call the list with the command glCallList(listID);

The effect of this command is to tell the GPU to execute a list that it has already stored. You can tell the graphics card that a list is no longer needed by calling gl.glDeleteLists(listID, 1);

The second parameter in this method call plays the same role as the parameter in *glGenLists*; that is, it allows you delete several sequentially numbered lists. Deleting a list when you are through with it allows the GPU to reuse the memory that was used by that list.

* * *

Vertex buffer objects take a different approach to reusing information. They only store data, not commands. A VBO is similar to an array. In fact, it is essentially an array that can be stored on the GPU for efficiency of reuse. There are OpenGL commands to create and delete VBOs and to transfer data from an array on the CPU side into a VBO on the GPU. You can configure *glDrawArrays*() and *glDrawElements*() to take the data from a VBO instead of from an ordinary array (in C) or from an nio Buffer (in JOGL). This means that you can send the data once to the GPU and use it any number of times.

I will not discuss how to use VBOs here, since it was not a part of OpenGL 1.1. However, there is a sample program that lets you compare different techniques for rendering a complex image. The C version of the program is *glut/color-cube-of-spheres.c*, and the Java version is *jogl/ColorCubeOfSpheres.java*. The program draws 1331 spheres, arranged in an 11-by-11-by11 cube. The spheres are different colors, with the amount of red in the color varying along one axis, the amount of green along a second axis, and the amount of blue along the third. Each sphere has 66 vertices, whose coordinates can be computed using the math functions *sin* and *cos*. The program allows you to select from five different rendering methods, and it shows the time that it takes to render the spheres using the selected method. (The Java version has a drop-down menu for selecting the method; in the C version, right-click the image to get the menu.) You can use your mouse to rotate the cube of spheres, both to get a better view and to generate more data for computing the average render time. The five rendering techniques are:

- *Direct Draw, Recomputing Vertex Data* — A remarkably foolish way to draw 1331 spheres, by recomputing all of the vertex coordinates every time a sphere is drawn.
- *Direct Draw, Precomputed Data* — The vertex coordinates are computed once and stored in an array. The spheres are drawn using *glBegin/glEnd*, but the data used in the calls to *glVertex** are taken from the array rather than recomputed each time they are needed.

- *Display List* — A display list is created containing all of the commands and data needed to draw a sphere. Each sphere can then be drawn by a single call of that display list.
- *DrawArrays with Arrays* — The data for the sphere is stored in a vertex array (or, for Java, in an nio buffer), and each sphere is drawn using a call to *glDrawArrays*, using the techniques discussed earlier in this section. The data has to be sent to the GPU every time a sphere is drawn.
- *DrawArrays with VBOs* — Again, *glDrawArrays* is used to draw the spheres, but this time the data is stored in a VBO instead of in an array, so the data only has to be transmitted to the GPU once.

In my own experiments, I found, as expected, that display lists and VBOs gave the shortest rendering times, with little difference between the two. There were some interesting differences between the results for the C version and the results for the Java version, which seem to be due to the fact that function calls in C are more efficient than method calls in Java. You should try the program on your own computer, and compare the rendering times for the various rendering methods.

3.5 Some Linear Algebra

Linear algebra is a branch of mathematics that is fundamental to computer graphics. It studies vectors, linear transformations, and matrices. We have already encountered these topics in Subsection 2.3.8 in a two-dimensional context. In this section, we look at them more closely and extend the discussion to three dimensions.

It is not essential that you know the mathematical details that are covered in this section, since they can be handled internally in OpenGL or by software libraries. However, you will need to be familiar with the concepts and the terminology. This is especially true for modern OpenGL, which leaves many of the details up to your programs. Even when you have a software library to handle the details, you still need to know enough to use the library. You might want to skim this section and use it later for reference.

3.5.1 Vectors and Vector Math

A vector is a quantity that has a length and a direction. A vector can be visualized as an arrow, as long as you remember that it is the length and direction of the arrow that are relevant, and that its specific location is irrelevant. Vectors are often used in computer graphics to represent directions, such as the direction from an object to a light source or the direction in which a surface faces. In those cases, we are more interested in the direction of a vector than in its length.

If we visualize a 3D vector *V* as an arrow starting at the origin, (0,0,0), and ending at a point *P*, then we can, to a certain extent, identify *V* with *P*—at least as long as we remember that an arrow starting at any other point could also be used to represent *V*. If *P* has coordinates (*a,b,c*), we can use the same coordinates for *V*. When we think of (*a,b,c*) as a vector, the value of *a* represents the **change** in the *x*-coordinate between the starting point of the arrow and its ending point, *b* is the change in the *y*-coordinate, and *c* is the change in the *z*-coordinate. For example, the 3D point (*x,y,z*) = (3,4,5) has the same coordinates as the vector (*dx,dy,dz*) = (3,4,5). For the point, the coordinates (3,4,5) specify a position in space in the *xyz* coordinate system. For the vector, the coordinates (3,4,5) specify the change in the *x*, *y*, and *z* coordinates along the vector. If we represent the vector with an arrow that starts at the origin (0,0,0), then the head of the arrow will be at (3,4,5). But we could just as well visualize the vector as an arrow that starts at the point (1,1,1), and in that case the head of the arrow would be at the point (4,5,6).

The distinction between a point and a vector is subtle. For some purposes, the distinction can be ignored; for other purposes, it is important. Often, all that we have is a sequence of numbers, which we can treat as the coordinates of either a vector or a point, whichever is more appropriate in the context.

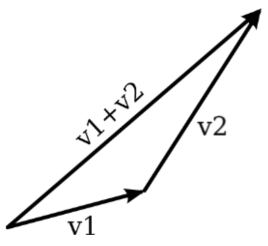
One of the basic properties of a vector is its length. In terms of its coordinates, the length of a 3D vector (*x,y,z*) is given by *sqrt*(x^2+y^2+z^2). (This is just the Pythagorean theorem in three dimensions.) If *v* is a vector, its length is denoted by |*v*|. The length of a vector is also called its norm. (We are considering 3D vectors here, but concepts and formulas are similar for other dimensions.)

Vectors of length 1 are particularly important. They are called unit vectors. If *v* = (*x,y,z*) is any vector other than (0,0,0), then there is exactly one unit vector that points in the same direction as *v*. That vector is given by

(x/length, y/length, z/length) where *length* is the length of *v*. Dividing a vector by its length is said to normalize the vector: The result is a unit vector that points in the same direction as the original vector.

Two vectors can be added. Given two vectors *v1* = (*x1,y1,z1*) and *v2* = (*x2,y2,z2*), their sum is defined as v1 + v2 = (x1+x2, y1+y2, z1+z2); The sum has a geometric meaning:

The vector sum of v1 and v2 can be obtained by placing the starting point of v2 at the ending point of v1. The sum is the vector from the starting point of v1 to the ending point of v2.



Remember that vectors have length and direction, but no set position.

Multiplication is more complicated. The obvious definition of the product of two vectors, similar to the definition of the sum, does not have geometric meaning and is rarely used. However, there are three kinds of vector multiplication that are used: the scalar product, the dot product, and the cross product.

If *v* = (*x,y,z*) is a vector and *a* is a number, then the scalar product of *a* and *v* is defined as

av = (a*x, a*y, a*z);

Assuming that *a* is positive and *v* is not zero, *av* is a vector that points in the same direction as *v*, whose length is *a* times the length of *v*. If *a* is negative, *av* points in the opposite direction from *v*, and its length is |*a*| times the length of *v*. This type of product is called a scalar product because a number like *a* is also referred to as a “scalar,” perhaps because multiplication by *a* scales *v* to a new length.

Given two vectors *v1* = (*x1,y1,z1*) and *v2* = (*x2,y2,z2*), the dot product of *v1* and *v2* is denoted by *v1·v2* and is defined by v1·v2 = x1*x2 + y1*y2 + z1*z2

Note that the dot product is a number, not a vector. The dot product has several very important geometric meanings. First of all, note that the length of a vector *v* is just the square root of *v·v*. Furthermore, the dot product of two non-zero vectors *v1* and *v2* has the property that cos(angle) = v1·v2 / (|v1|*|v2|)

where *angle* is the measure of the angle between *v1* and *v2*. In particular, in the case of two unit vectors, whose lengths are 1, the dot product of two unit vectors is simply the cosine of the angle between them. Furthermore, since the cosine of a 90-degree angle is zero, two non-zero vectors are perpendicular if and only if their dot product is zero. Because of these properties, the dot product is particularly important in lighting calculations, where the effect of light shining on a surface depends on the angle that the light makes with the surface.

The scalar product and dot product are defined in any dimension. For vectors in 3D, there is another type of product called the cross product, which also has an important geometric meaning. For vectors *v1* = (*x1,y1,z1*) and *v2* = (*x2,y2,z2*), the cross product of *v1* and *v2* is denoted *v1×v2* and is the vector defined by v1×v2 = (y1*z2 - z1*y2, z1*x2 - x1*z2, x1*y2 - y1*x2)

If *v1* and *v2* are non-zero vectors, then *v1×v2* is zero if and only if *v1* and *v2* point in the same direction or in exactly opposite directions. Assuming *v1×v2* is non-zero, then it is perpendicular both to *v1* and to *v2*; furthermore, the vectors *v1*, *v2*, *v1×v2* follow the right-hand rule (in a right-handed coordinate system); that is, if you curl the fingers of your right hand from *v1* to *v2*, then your thumb points in the direction of *v1×v2*. If *v1* and *v2* are perpendicular unit vectors, then the cross product *v1×v2* is also a unit vector, which is perpendicular both to *v1* and to *v2*.

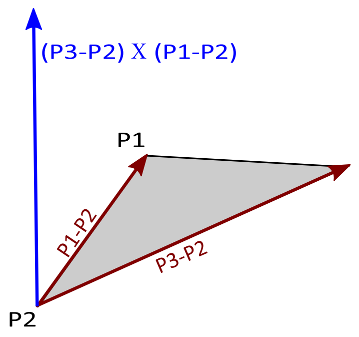
Finally, I will note that given two points *P1* = (*x1,y1,z1*) and *P2* = (*x2,y2,z2*), the difference *P2−P1* is defined by

P2 − P1 = (x2 − x1, y2 − y1, z2 − z1)

This difference is a vector that can be visualized as an arrow that starts at *P1* and ends at *P2*.

Now, suppose that *P1*, *P2*, and *P3* are vertices of a polygon. Then the vectors *P1−P2* and *P3−P2* lie in the plane of the polygon, and so the cross product

(P3−P2) × (P1−P2) is a vector that is perpendicular to the polygon.



Try to visualize this in 3D! A vector that is perpendicular to the triangle is obtained by taking P3 the cross product of P3-P2 and P1-P2, which are vectors that lie along two sides of the triangle.

This vector is said to be a normal vector for the polygon. A normal vector of length one is called a unit normal. Unit normals will be important in lighting calculations, and it will be useful to be able to calculate a unit normal for a polygon from its vertices.

3.5.2 Matrices and Transformations

A matrix is just a two-dimensional array of numbers. A matrix with *r* rows and *c* columns is said to be an *r*-by-*c* matrix. If *A* and *B* are matrices, and if the number of columns in *A* is equal to the number of rows in *B*, then *A* and *B* can be multiplied to give the matrix product *AB*. If *A* is an *n*-by-*m* matrix and *B* is an *m*-by-*k* matrix, then *AB* is an *n*-by-*k* matrix. In particular, two *n*-by-*n* matrices can be multiplied to give another *n*-by-*n* matrix.

An *n*-dimensional vector can be thought of an *n*-by-*1* matrix. If *A* is an *n*-by-*n* matrix and *v* is a vector in *n* dimensions, thought of as an *n*-by-*1* matrix, then the product *Av* is again an *n*-dimensional vector. The product of a 3-by-3 matrix *A* and a 3D vector *v* = (*x,y,z*) is often displayed like this:

a1 a2 a3a1*x + a2*y + a3*z x b1 b2 b3=b1*x + b2*y
+b3*z c1 c2 c3c1*x +c2*y y + c3*z
Note that the *i*-th z coordinate in the product *Av* is simply the dot product of the *i*-th row of the matrix *A* and the vector *v*.

Using this definition of the multiplication of a vector by a matrix, a matrix defines a transformation that can be applied to one vector to yield another vector. Transformations that are defined in this way are linear transformations, and they are the main object of study in linear algebra. A linear transformation *L* has the properties that for two vectors *v* and *w*, *L(v+w) = L(v) + L(w)*, and for a number *s*, *L(sv) = sL(v)*.

Rotation and scaling are linear transformations, but translation is **not** a linear transformation. To include translations, we have to widen our view of transformation to include affine transformations. An affine transformation can be defined, roughly, as a linear transformation followed by a translation. Geometrically, an affine transformation is a transformation that preserves parallel lines; that is, if two lines are parallel, then their images under an affine transformation will also be parallel lines. For computer graphics, we are interested in affine transformations in three dimensions. However—by what seems at first to be a very odd trick— we can narrow our view back to the linear by moving into the fourth dimension.

Note first of all that an affine transformation in three dimensions transforms a vector (*x1,y1,z1*) into a vector (*x2,y2,z2*) given by formulas

x2 = a1*x1 + a2*y1 + a3*z1 + t1 y2 = b1*x1 + b2*y1 + b3*z1 + t2 z2 = c1*x1 + c2*y1 + c3*z1 + t3

These formulas express a linear transformation given by multiplication by the 3-by-3 matrix

a1 a2 a3 b1 b2 b3 c1 c2 c3

followed by translation by *t1* in the *x* direction, *t2* in the *y* direction and *t3* in the *z* direction. The trick is to replace each three-dimensional vector (*x,y,z*) with the four-dimensional vector (*x,y,z,* 1), adding a "1" as the fourth coordinate. And instead of the 3-by-3 matrix, we use the 4-by-4 matrix

a1 a2 a3 t1 b1 b2 b3 t2 c1 c2 c3 t3

0 0 0 1

If the vector (*x1,y1,z1*,1) is multiplied by this 4-by-4 matrix, the result is precisely the vector (*x2,y2,z2,*1). That is, instead of applying an *affine* transformation to the 3D vector (*x1,y1,z1*), we can apply a *linear* transformation to the 4D vector (*x1,y1,z1*,1).

This might seem pointless to you, but nevertheless, that is what is done in OpenGL and other 3D computer graphics systems: An affine transformation is represented as a 4by-4 matrix in which the bottom row is (0,0,0,1), and a three-dimensional vector is changed into a four dimensional vector by adding a 1 as the final coordinate. The result is that all the affine transformations that are so important in computer graphics can be implemented as multiplication of vectors by matrices.

The identity transformation, which leaves vectors unchanged, corresponds to multiplication by the identity matrix, which has ones along its descending diagonal and zeros elsewhere. The OpenGL function *glLoadIdentity*() sets the current matrix to be the 4-by-4 identity matrix. An OpenGL transformation function, such as *glTranslatef* (*tx,ty,tz*), has the effect of multiplying the current matrix by the 4-by-4 matrix that represents the transformation. Multiplication is on the right; that is, if *M* is the current matrix and *T* is the matrix that represents the transformation, then the current matrix will be set to the product matrix *MT*. For the record, the following illustration shows the identity matrix and the matrices corresponding to various OpenGL transformation functions:

1 0 0 1 0 0

0 txsx 0 0 ty0

01 0 0 0 0 00 1 sy 0

0 0 1 00 0 1 tz0 0 sz 0 0 0 0

10 0 0 10 0 0 1

Identity Matrix glTranslatef(tx,ty,tz) glScalef(sx,sy,sz)

1 0 0 0cos(d) 0 sin(d) 0cos(d) -sin(d) 0
0 0 cos(d) - sin(d) 00 1 0 0sin(d) cos(d) 0 0 0
sin(d) cos(d) 0-sin(d) 0 cos(d) 00 0 1 0
0 0 0 10 0 0
10 0 0 1

glRotatef(d,1,0,0) glRotatef(d,0,1,0) glRotatef(d,0,0,1)

It is even possible to use an arbitrary transformation matrix in OpenGL, using the function *glMultMatrixf* (*T*) or *glMultMatrixd*(*T*). The parameter, *T*, is an array of numbers of type float or double, representing a transformation matrix. The array is a one-dimensional array of length 16. The items in the array are the numbers from the transformation matrix, stored in column-major order, that is, the numbers in the fist column, followed by the numbers in the second column, and so on. These functions multiply the current matrix by the matrix *T*, on the right. You could use them, for example, to implement a shear transform, which is not easy to represent as a sequence of scales, rotations, and translations.

3.5.3 Homogeneous Coordinates

We finish this section with a bit of mathematical detail about the implementation of transformations. There is one common transformation in computer graphics that is not an affine transformation: In the case of a perspective projection, the projection transformation is not affine. In a perspective projection, an object will appear to get smaller as it moves farther away from the viewer, and that is a property that no affine transformation can express, since affine transforms preserve parallel lines and parallel lines will seem to converge in the distance in a perspective projection.

Surprisingly, we can still represent a perspective projection as a 4-by-4 matrix, provided we are willing to stretch our use of coordinates even further than we have already. We have already represented 3D vectors by 4D vectors in which the fourth coordinate is 1. We now allow the fourth coordinate to be anything at all, except for requiring that at least one of the four coordinates is non-zero. When the fourth coordinate, *w*, is non-zero, we consider the coordinates (*x,y,z,w*) to represent the three-dimensional vector (*x/w,y/w,z/w*). Note that this is consistent with our previous usage, since it considers (*x,y,z,1*) to represent (*x,y,z*), as before. When the fourth coordinate is zero, there is no corresponding 3D vector, but it is possible to think of (*x,y,z*,0) as representing a 3D "point at infinity" in the direction of (*x,y,z*).

Coordinates (*x,y,z,w*) used in this way are referred to as homogeneous coordinates. If we use homogeneous coordinates, then any 4-by-4 matrix can be used to transform threedimensional vectors, including matrices whose bottom row is not (0,0,0,1). Among the transformations that can be represented in this way is the projection transformation for a perspective projection. And in fact, this is what OpenGL does internally. It represents all three-dimensional points and vectors using homogeneous coordinates, and it represents all transformations as 4-by-4 matrices. You can even specify vertices using homogeneous coordinates. For example, the command glVertex4f(x,y,z,w);

with a non-zero value for *w*, generates the 3D point (*x/w,y/w,z/w*). Fortunately, you will almost never have to deal with homogeneous coordinates directly. The only real exception to this is that homogeneous coordinates are used, surprisingly, when configuring OpenGL lighting, as we'll see in the next chapter.

3.6 Using GLUT and JOGL

OpenGL is an API for graphics only, with no support for things like windows or events. OpenGL depends on external mechanisms to create the drawing surfaces on which it will draw. Windowing APIs that support OpenGL often do so as one library among many others that are used to produce a complete application. We will look at two cross-platform APIs that make it possible to use OpenGL in applications, one for C/C++ and one for Java.

For simple applications written in C or C++, one possible windowing API is GLUT (OpenGL Utility Toolkit). GLUT is a small API. It is used to create windows that serve as simple frames for OpenGL drawing surfaces. It has support for handling mouse and keyboard events, and it can do basic animation. It does not support controls such as buttons or input fields, but it does allow for a menu that pops up in response to a mouse action. The original version of GLUT is no longer actively supported, and a version called freeglut (http://freeglut.sourceforge.net/) is recommended instead. For example, the version included in Linux is actually freeglut. For details of the freeglut API, see

http://freeglut.sourceforge.net/docs/api.php

JOGL (Java OpenGL) is a collection of classes that make it possible to use OpenGL in Java applications. JOGL is integrated into Swing and AWT, the standard Java graphical user interface APIs. With JOGL, you can create Java GUI components on which you can draw using OpenGL. These OpenGL components can be used in any Java application, in much the same way that you would use a Canvas or JPanel as a drawing surface. Like many things Java, JOGL is immensely complicated. We will use it only in fairly simple applications. JOGL is not a standard part of Java. It's home web site is http://jogamp.org/jogl/www/

This section contains information to get you started using GLUT and JOGL, assuming that you already know the basics of programming with C and Java. It also briefly discusses *glsim.js*, a JavaScript library that I have written to simulate the subset of OpenGL 1.1 that is used in this book.

3.6.1 Using GLUT

To work with GLUT, you will need a C compiler and copies of the OpenGL and GLUT (or freeglut) development libraries. I can't tell you exactly that means on your own computer. On my computer, which runs Linux Mint, for example, the free C compiler gcc is already available. To do OpenGL development, I installed several packages, including *freeglut3-dev* and *libgl1mesa-dev*. (Mesa is a Linux implementation of OpenGL.) If *glutprog.c* contains a complete C program that uses GLUT, I can compile it using a command such as gcc -o glutprog glutprog.c -lGL -lglut

The "-o glutprog" tells the compiler to use "glutprog" as the name of its output file, which can then be run as a normal executable file; without this option, the executable file would be named "a.out". The "-lglut" and "-lGL" options tell the compiler to link the program with the GLUT and OpenGL libraries. (The character after the "-" is a lower case "L".) Without these options, the linker won't recognize any GLUT or OpenGL functions. If the program also uses the GLU library, compiling it would require the option "-lGLU, and if it uses the math library, it would need the option "-lm". If a program requires additional .c files, they should be included as well. For example, the sample program *glut/color-cube-of-spheres.c* depends on *camera.c*, and it can be compiled with the Linux gcc compiler using the command:

```
gcc -o cubes color-cube-of-spheres.c camera.c -lGL -lglut -lGLU -lm
```

The sample program *glut/glut-starter.c* can be used as a starting point for writing programs that use GLUT. While it doesn't do anything except open a window, the program contains the framework needed to do OpenGL drawing, including doing animation, responding to mouse and keyboard events, and setting up a menu. The source code contains comments that tell you how to use it.

On Windows, you might consider installing the WSL, or Windows Subsystem for Linux, (https://docs.microsoft.com/en-us/windows/wsl/), which as I write this should soon include the ability to work with GUI programs. WSL is an official Microsoft system lets you install a version of Linux inside Windows. Another option is the older open source project, Cygwin (https://cygwin.com/). (Using Cygwin, I installed the packages gcc-core, xinit, xorg-server, libglut-devel, libGLU-devel, and libGL-devel. After starting the X11 window system with the startxwin command, I was able to compile and run OpenGL examples from this textbook in a Cygwin terminal window using the same commands that I would use in Linux.)

For MacOS, the situation is more complicated, because OpenGL has been deprecated in favor of Metal, Apple's own proprietary API. However, as I write this, OpenGL can still be used on MacOS with Apple's XCode developer tools. The examples from this textbook require some modification to work with XCode tools, since the OpenGL and GLUT libraries are not loaded in the same way on Mac as they are on Linux. Modified programs for use on MacOS can be found in the source folder *glut/glut-mac*. See the README.txt file in that folder for more information.

* * *

The GLUT library makes it easy to write basic OpenGL applications in C. GLUT uses event-handling functions. You write functions to handle events that occur when the display needs to be redrawn or when the user clicks the mouse or presses a key on the keyboard.

To use GLUT, you need to include the header file *glut.h* (or *freeglut.h*) at the start of any source code file that uses it, along with the general OpenGL header file, *gl.h*. The header files should be installed in a standard location, in a folder named *GL*. (But note that the folder name could be different, or omitted entirely.) So, the program usually begins with something like

```
#include <GL/gl.h>

#include <GL/glut.h>
```

On my computer, saying *#include <GL/glut.h>* actually includes the subset of FreeGLUT that corresponds to GLUT. To get access to all of FreeGLUT, I would substitute *#include <GL/freeglut.h>*. Depending on the features that it uses, a program might need other header files, such as *#include <GL/glu.h>* and *#include <math.h>*.

The program's *main*() function must contain some code to initialize GLUT, to create and open a window, and to set up event handling by registering the functions that should be called in response to various events. After this setup, it must call a function that runs the GLUT event-handling loop. That function waits for events and processes them by calling the functions that have been registered to handle them. The event loop runs until the program ends, which happens when the user closes the window or when the program calls the standard *exit*() function.

To set up the event-handling functions, GLUT uses the fact that in C, it is possible to pass a function name as a parameter to another function. For example, if *display*() is the function that should be called to draw the content of the window, then the program would use the command glutDisplayFunc(display);

to install this function as an event handler for display events. A display event occurs when the contents of the window need to be redrawn, including when the window is first opened. Note that *display* must have been previously defined, as a function with no parameters:

```
void display() { .

. // OpenGL drawing code goes here! .

}
```

Keep in mind that it's not the name of this function that makes it an OpenGL display function. It has to be set as the display function by calling *glutDisplayFunc*(*display*). All of the GLUT event-handling functions work in a similar way (except many of them do need to have parameters).

There are a lot of possible event-handling functions, and I will only cover some of them here. Let's jump right in and look at a possible *main*() routine for a GLUT program that uses most of the common event handlers:

```
int main(int argc, char** argv) { glutInit(&argc, argv); // Required initialization!

glutInitDisplayMode(GLUT _DOUBLE | GLUT DEPTH); glutInitWindowSize(500,500);   //
size of display area, in pixels glutInitWindowPosition(100,100);  // location in screen
coordinates glutCreateWindow("OpenGL Program"); // the parameter is the window title

glutDisplayFunc(display);     // called when window needs to be redrawn
glutReshapeFunc(reshape);// called when size of the window changes
glutKeyboardFunc(keyFunc);           // called when user types a character
glutSpecialFunc(specialKeyFunc);// called when user presses a special key
glutMouseFunc(mouseFunc);          // called for mousedown and mouseup events
glutMotionFunc(mouseDragFunc); // called when mouse is dragged
glutIdleFunc(idleFun);          // called when there are no other events
```

glutMainLoop(); // Run the event loop! This function never returns.

return 0; // (This line will never actually be reached.) }

The first five lines do some necessary initialization, the next seven lines install event handlers, and the call to *glutMainLoop*() runs the GLUT event loop. I will discuss all of the functions that are used here. The first GLUT function call must be *glutInit*, with the parameters as shown. (Note that *argc* and *argv* represent command-line arguments for the program. Passing them to *glutInit* allows it to process certain command-line arguments that are recognized by GLUT. I won't discuss those arguments here.) The functions *glutInitWindowSize* and *glutInitWindowPosition* do the obvious things; size is given in pixels, and window position is given in terms of pixel coordinates on the computer screen, with (0,0) at the upper left corner of the screen. The function *glutCreateWindow* creates the window, but note that nothing can happen in that window until *glutMainLoop* is called. Often, an additional, user-defined function is called in *main*() to do whatever initialization of global variables and OpenGL state is required by the program. OpenGL initialization can be done after calling *glutCreateWindow* and before calling *glutMainLoop*. Turning to the other functions used in *main*(), glutInitDisplayMode(GLUT _DOUBLE | GLUT _DEPTH) — Must be called to define some characteristics of the OpenGL drawing context. The parameter specifies features that you would like the OpenGL context to have. The features are represented by constants that are OR'ed together in the parameter. *GLUT _DEPTH* says that a depth buffer should be created; without it, the depth test won't work. If you are doing 2D graphics, you wouldn't include this option. *GLUT DOUBLE* asks for double buffering, which means that drawing is actually done off-screen, and the off-screen copy has to copied to the screen to be seen. The copying is done by glutSwapBuffers(), which **must** be called at the end of the display function. (You can use *GLUT SINGLE* instead of *GLUT _DOUBLE* to get single buffering; in that case, you have to call *glFlush*() at the end of the display function instead of *glutSwapBuffers*(). However, all of the examples in this book use *GLUT DOUBLE*.)

glutDisplayFunc(display) — The display function should contain OpenGL drawing code that can completely redraw the scene. This is similar to *paintComponent*() in the Java Swing API. The display function can have any name, but it must be declared as a void function with no parameters: *void display*().

glutReshapeFunc(reshape) — The reshape function is called when the user changes the

size of the window. Its parameters tell the new width and height of the drawing area:

void reshape(int width, int height)

For example, you might use this method to set up the projection transform, if the projection depends only on the window size. A reshape function is not required, but if one is provided, it should always set the OpenGL viewport, which is the part of the window that is used for drawing. Do this by calling glViewport(0,0,width,height);

The viewport is set automatically if no reshape function is specified.

glutKeyboardFunc(keyFunc) — The keyboard function is called when the user types a character such as ’b’ or ’A’ or a space. It is not called for special keys such as arrow keys that do not produce characters when pressed. The keyboard function has a parameter of type unsigned char which represents the character that was typed. It also has two *int* parameters that give the location of the mouse when the key was pressed, in pixel coordinates with (0,0) at the upper left corner of the display area. So, the definition of the key function must have the form:

void keyFunc(unsigned char ch, int x, int y)

Whenever you make any changes to the program’s data that require the display to be redrawn, you should call *glutPostRedisplay*(). This is similar to calling *repaint*() in Java. It is better to call *glutPostRedisplay*() than to call the display function directly. (I also note that it’s possible to call OpenGL drawing commands directly in the event-handling functions, but it probably only makes sense if you are using single buffering; if you do this, call *glFlush*() to make sure that the drawing appears on the screen.)
glutSpecialFunc(specialKeyFunc) — The “special” function is called when the user presses certain special keys, such as an arrow key or the Home key. The parameters are an integer code for the key that was pressed, plus the mouse position when the key was pressed:

void specialKeyFunc(int key, int x, int y)

GLUT has constants to represent the possible key codes, including *GLUT KEY LEFT*, *GLUT KEYRIGHT*, *GLUT _KEY UP*, and *GLUT KEY DOWN* for the arrow keys and *GLUT KEY _HOME* for the Home key. For example, you can check whether the user pressed the left arrow key by testing if(key == GLUT KEY LEFT).

glutMouseFunc(mouseFunc) — The mouse function is called both when the user presses and when the user releases a button on the mouse, with a parameter to tell which of these occurred. The function will generally look like this:

void mouseFunc(int button, int buttonState, int x, int y) { if (buttonState == GLUT DOWN) {

// handle mousePressed event

}

else { // buttonState is GLUT UP

// handle mouseReleased event

}

}

The first parameter tells which mouse button was pressed or released; its value is the constant *GLUT LEFT BUTTON* for the left, *GLUT MIDDLE BUTTON* for the middle, and *GLUT RIGHT BUTTON* for the right mouse button. The other two parameters tell the position of the mouse. The mouse position is given in pixel coordinates with (0,0) in the top left corner of the display area and with y increasing from top to bottom.

glutMotionFunc(mouseDragFunc) — The motion function is called when the user moves the mouse while dragging, that is, while a mouse button is pressed. After the user presses the mouse in the OpenGL window, this function will continue to be called even if the mouse moves outside the window, and the mouse release event will also be sent to the same window. The function has two parameters to specify the new mouse position:

void mouseDragFunc(int x, int y)

glutIdleFunc(idleFunction) — The idle function is called by the GLUT event loop whenever there are no events waiting to be processed. The idle function has no parameters. It is called as often as possible, not at periodic intervals. GLUT also has a timer function, which schedules some function to be called once, after a specified delay. To set a timer, call glutTimerFunc(delayInMilliseconds, timerFunction, userSelectedID)

and define *timerFunction* as void timerFunction(int timerID) { ...

The parameter to *timerFunction* when it is called will be the same integer that was passed as the third parameter to *glutTimerFunc*. If you want to use *glutTimerFunc* for animation, then *timerFunction* should end with another call to *glutTimerFunc*.

* * *

A GLUT window does not have a menu bar, but it is possible to add a hidden popup menu to the window. The menu will appear in response to a mouse click on the display. You can set whether it is triggered by the left, middle, or right mouse button.

A menu is created using the function *glutCreateMenu(menuHandler)*, where the parameter is the name of a function that will be called when the user selects a command from the menu. The function must be defined with a parameter of type int that identifies the command that was selected:

void menuHandler(int commandID) { ...

Once the menu has been created, commands are added to the menu by calling the function *glutAddMenuEntry*(*name,commandID*). The first parameter is the string that will appear in the menu. The second is an int that identifies the command; it is the integer that will be passed to the menu-handling function when the user selects the command from the menu.

Finally, the function *glutAttachMenu*(*button*) attaches the menu to the window. The parameter specifies which mouse button will trigger the menu. Possible values are *GLUT LEFT _BUTTON*, *GLUT _MIDDLE _BUTTON*, and *GLUT _RIGHT _BUTTON*. As far as I can tell, if a mouse click is used to trigger the popup menu, than the same mouse click will **not** also produce a call to the mouse-handler function.

Note that a call to *glutAddMenuEntry* doesn’t mention the menu, and a call to *glutAttachMenu* doesn’t mention either the menu or the window. When you call *glutCreateMenu*, the menu that is created becomes the “current menu” in the GLUT state. When *glutAddMenuEntry* is called, it adds a command to the current menu. When *glutAttachMenu* is called, it attaches the current menu to the current window, which was set by a call to *glutCreateWindow*.

All this is consistent with the OpenGL “state machine” philosophy, where functions act by modifying the current state.

As an example, suppose that we want to let the user set the background color for the display. We need a function to carry out commands that we will add to the menu. For example, we might define

```
function doMenu( int commandID ) { if ( commandID == 1) glClearColor(0,0,0,1); //
BLACK

else if ( commandID == 2) glClearColor(1,1,1,1); // WHITE

else if ( commandID == 3) glClearColor(0,0,0.5,1); // DARK BLUE

else if (commandID == 10) exit(0); // END THE PROGRAM

glutPostRedisplay(); // redraw the display, with the new background color }
```

We might have another function to create the menu. This function would be called in *main*(), after calling *glutCreateWindow*:

```
function createMenu() { glutCreateMenu( doMenu ); // Call doMenu() in response to
menu commands.

glutAddMenuEntry( "Black Background", 1 ); glutAddMenuEntry( "White Background", 2
); glutAddMenuEntry( "Blue Background", 3 ); glutAddMenuEntry( "EXIT", 10 );
glutAttachMenu(GLUT _RIGHT BUTTON); // Show menu on right-click. }
```

It’s possible to have submenus in a menu. I won’t discuss the procedure here, but you can look at the sample program *glut/ifs-polyhedron-viewer.c* for an example of using submenus.

* * *

In addition to window and event handling, GLUT includes some functions for drawing basic 3D shapes such as spheres, cones, and regular polyhedra. It has two functions for each shape, a “solid” version that draws the shape as a solid object, and a wireframe version that draws something that looks like it’s made of wire mesh. (The wireframe is produced by drawing just the outlines of the polygons that make up the object.) For example, the function void glutSolidSphere(double radius, int slices, int stacks)

draws a solid sphere with the given radius, centered at the origin. Remember that this is just an approximation of a sphere, made up of polygons. For the approximation, the sphere is divided by lines of longitude, like the slices of an orange, and by lines of latitude, like a stack of disks. The parameters *slices* and *stacks* tell how many subdivisions to use. Typical values are 32 and 16, but the number that you need to get a good approximation for a sphere depends on the size of the sphere on the screen. The function *glutWireframeSphere* has the same parameters but draws only the lines of latitude and longitude. Functions for a cone, a cylinder, and a torus (doughnut) are similar:

void glutSolidCone(double base, double height, int slices, int stacks)

void glutSolidTorus(double innerRadius, double outerRadius,

int slices, int rings)

void glutSolidCylinder(double radius, double height, int slices, int stacks) // NOTE: Cylinders are available in FreeGLUT and in Java, // but not in the original GLUT library.

For a torus, the *innerRadius* is the size of the doughnut hole. The function void glutSolidCube(double size)

draws a cube of a specified size. There are functions for the other regular polyhedra that have no parameters and draw the object at some fixed size: *glutSolidTetrahedron*(), *glutSolidOctahedron*(), *glutSolidDodecahedron*(), and *glutSolidIcosahedron*(). There is also *glutSolidTeapot*(*size*) that draws a famous object that is often used as an example. Here’s what the teapot looks like:

Wireframe versions of all of the shapes are also available. For example, *glutWireTeapot*(*size*) draws a wireframe teapot. Note that GLUT shapes come with normal vectors that are required for lighting calculations. However, except for the teapot, they do not come with texture coordinates, which are required for applying textures to objects.

GLUT also includes some limited support for drawing text in an OpenGL drawing context. I won't discuss that possibility here. You can check the API documentation if you are interested, and you can find an example in the sample program *glut/color-cube-of-spheres.c*.

3.6.2 Using JOGL

JOGL is a framework for using OpenGL in Java programs. It is a large and complex API that supports all versions of OpenGL, but it is fairly easy to use for basic applications. You should use JOGL 2.4 or later. The programs in this book were tested with version 2.4.0.

The sample program *jogl/JoglStarter.java* can be used as a starting point for writing OpenGL programs using JOGL. While it doesn't do anything except open a window, the program contains the framework needed to do OpenGL drawing, including doing animation, responding to mouse and keyboard events, and setting up a menu. The source code contains comments that tell you how to use it.

To use JOGL, you will need two .jar files containing the Java classes for JOGL: *jogl-all.jar* and *gluegen-rt.jar*. In addition, you will need two native library files. A native library is a collection of routines that can be called from Java but are not written in Java. Routines in a native library will work on only one kind of computer; you need a different native library for each type of computer on which your program is to be used. The native libraries for JOGL are stored in additional .jar files, which are available in several versions for different computers. For example, for 64-bit Linux on Intel or AMD CPUs, you need *jogl-all-natives-linux-amd64.jar* and *gluegen-rt-natives-linux-amd64.jar*. It is unfortunate that there are different versions for different platforms, since many people don't know exactly which one they are using. However, if you are in doubt, you can get more than one version; JOGL will figure out which one to use.

JOGL software can be found at https://jogamp.org/. You can download the jar files from the most recent release, which can be found near the end of the list at

https://jogamp.org/deployment/archive/rc/

Click on the release name, then click on the *jar/* link to see the full list of jar files. Find and download *jogl-all.jar* and *gluegen-rt.jar* and the corresponding native library files. I have also made jogl-all.jar and gluegen-rt.jar available on my own web site, along with the native libraries for some of the most common platforms, at http://math.hws.edu/eck/cs424/jogl _2 4 support/

JOGL is open-source, and the files are freely redistributable, according to their license.

To do JOGL development, you should create a directory somewhere on your computer to hold the jar files. Place the two JOGL jar files in that directory, along with the two native library jar files for your platform. (Having extra native library jar files doesn't hurt, as long as you have the ones that you need.)

It is possible to do JOGL development on the command line. You have to tell the *javac* command where to find the two JOGL jar files. You do that in the classpath ("-cp") option to the *javac* command. For example, if you are working in Linux or MacOS, and if the jar files happen to be in the same directory where you are working, you might say:

```
javac -cp jogl-all.jar:gluegen-rt.jar:. MyOpenGLProg.java
```

It's similar for Windows, except that the classpath uses a ";" instead of a ":" to separate the items in the list:

```
javac -cp jogl-all.jar;gluegen-rt.jar;. MyOpenGLProg.java
```

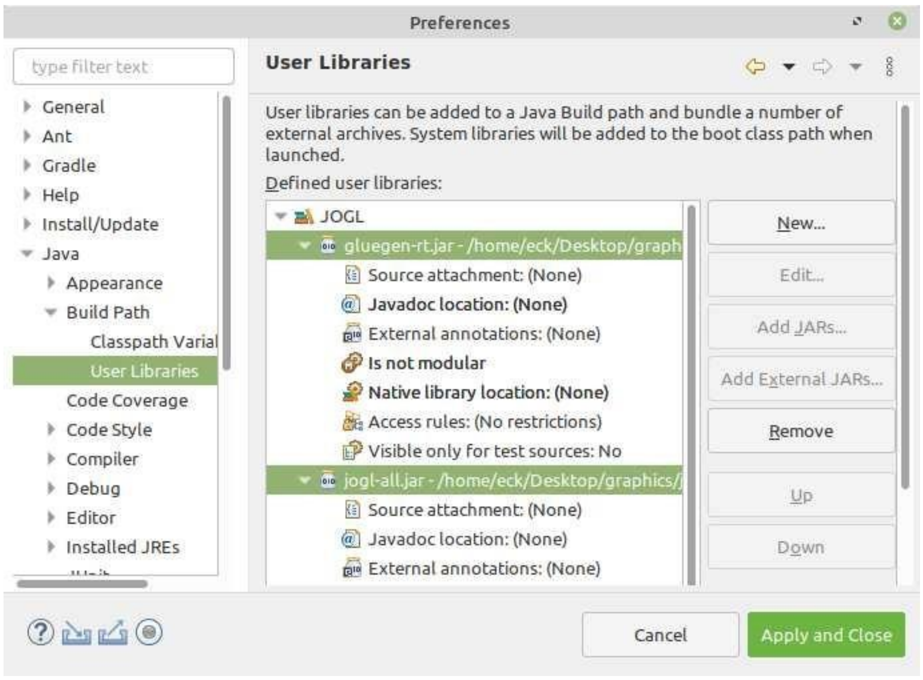
There is an essential period at the end of the classpath, which makes it possible for Java to find .java files in the current directory. If the jar files are not in the current directory, you can use full path names or relative path names to the files. For example, javac -cp ../jogl/jogl-all.jar:../jogl/gluegen-rt.jar:. MyOpenGLProg.java Running a program with the *java* command is exactly similar. For example:

```
java -cp jogl-all.jar:gluegen-rt.jar:. MyOpenGLProg
```

Note that you don't have to explicitly reference the native library jar files. They just have to be in the same directory with the JOGL jar files.

* * *

I do most of my Java development using the Eclipse IDE (http://eclipse.org). To do development with JOGL in Eclipse, you will have to configure Eclipse with information about the jar files. To do that, start up Eclipse. You want to create a "User Library" to contain the jar files: Open the Eclipse Preferences window, and select "Java" / "Build Path" / "User Libraries" on the left. Click the "New" button on the right. Enter "JOGL" (or any name you like) as the name of the user library. Make sure that the new user library is selected in the list of libraries, then click the "Add External Jars" button. In the file selection box, navigate to the directory that contains the JOGL jar files, and select the two jar files that are needed for JOGL, *jogl-all.jar* and *gluegen-rt.jar*. (Again, you do not need to add the native libraries; they just need to be in the same directory as the JOGL jar files.) Click "Open". The selected jars will be added to the user library. (You could also add them one at a time, if you don't know how to select multiple files.) It should look something like this:


Preferences

User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

- JOGL
  - gluegen-rt.jar - /home/eck/Desktop/graph
    - Source attachment: (None)
    - Javadoc location: (None)
    - External annotations: (None)
    - Is not modular
    - Native library location: (None)
    - Access rules: (No restrictions)
    - Visible only for test sources: No
  - jogl-all.jar - /home/eck/Desktop/graphics/
    - Source attachment: (None)
    - Javadoc location: (None)
    - External annotations: (None)

Click "OK." The user library has been created. You will only have to do this once, and then you can use it in all of your JOGL projects.

Now, to use OpenGL in a project, create a new Java project as usual in Eclipse. (If you are asked whether you want to create a module-info.java file for the project, say "Don't Create". Sample programs for this textbook do not use Java modules.) Right-click the new project in the Project Explorer view, and select "Build Path" / "Configure Build Path" from the menu. You will see the project Properties dialog, with "Java Build Path" selected on the left. (You can also access this through the "Properties" command in the "Project" menu.) Select the "Libraries" tab at the top of the window, and then click on "Class Path" in the "Libraries" tab to select it. Click the "Add Library" button, on the right. In the popup window, select "User Library" and click "Next." In the next window, select your JOGL User Library and click "Finish." Finally, click "Apply and Close" in the main Properties window. Your project should now be set up to do JOGL development. You should see the JOGL User Library listed as part of the project in the Project Explorer. Any time you want to start a new JOGL project, you can go through the same setup to add the JOGL User Library to the build path in the project.

* * *

With all that setup out of the way, it's time to talk about actually writing OpenGL programs with Java. With JOGL, we don't have to talk about mouse and keyboard handling or animation, since that can be done in the same way as in any Java Swing program. You will only need to know about a few classes from the JOGL API.

First, you need a GUI component on which you can draw using OpenGL. For that, you can use GLJPanel, which is a subclass of JPanel. (GLJPanel is for use in programs based on the Swing API; an alternative is GLCanvas, which is a subclass of the older AWT class Canvas.) The class is defined in the package *com.jogamp.opengl.awt*. All of the other classes that we will need for basic OpenGL programming are in the package *com.jogamp.opengl*.

JOGL uses Java's event framework to manage OpenGL drawing contexts, and it defines a custom event listener interface, GLEventListener, to manage OpenGL events. To draw on a GLJPanel with OpenGL, you need to create an object that implements the GLEventListener interface, and register that listener with your GLJPanel. The GLEventListener interface defines the following methods:

public void init(GLAutoDrawable drawable) public void display(GLAutoDrawable drawable) public void dispose(GLAutoDrawable drawable) public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height)

The *drawable* parameter in these methods tells which OpenGL drawing surface is involved. It will be a reference to the GLJPanel. (GLAutoDrawable is an interface that is implemented by GLJPanel and other OpenGL drawing surfaces.) The *init*() method is a place to do OpenGL initialization. (According to the documentation, it can actually be called several times, if the OpenGL context needs to be recreated for some reason. So *init*() should not be used to do initialization that shouldn't be done more than once.) The *dispose*() method will be called to give you a chance to do any cleanup before the OpenGL drawing context is destroyed. The *reshape*() method is called when the window first opens and whenever the size of the GLJPanel changes. OpenGL's *glViewport*() function is called automatically before *reshape*() is called, so you won't need to do it yourself. Usually, you won't need to write any code in *dispose*() or *reshape*(), but they have to be there to satisfy the definition of the GLEventListener interface.

The *display*() method is where the actual drawing is done and where you will do most of your work. It should ordinarily clear the drawing area and completely redraw the scene. Take a minute to study an outline for a minimal JOGL program. It creates a GLJPanel which also serves as the GLEventListener:

import com.jogamp.opengl.*; import com.jogamp.opengl.awt.GLJPanel;

import java.awt.Dimension; import javax.swing.JFrame; public class JOGLProgram extends GLJPanel implements GLEventListener {

public static void main(String[] args) {

JFrame window = new JFrame("JOGL Program"); JOGLProgram panel = new JOGLProgram(); window.setContentPane(panel); window.pack(); window.setLocation(50,50);

```
window.setDefaultCloseOperation(JFrame.EXIT _ON CLOSE); window.setVisible(true);

}

public JOGLProgram() { setPreferredSize( new Dimension(500,500) );

addGLEventListener(this);

}

// --------------- Methods of the GLEventListener interface -----------

public void init(GLAutoDrawable drawable) {

// called when the panel is created GL2 gl = drawable.getGL().getGL2(); // Add
initialization code here!

}

public void display(GLAutoDrawable drawable) {

// called when the panel needs to be drawn GL2 gl = drawable.getGL().getGL2(); // Add
drawing code here!

}

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {

// called when user resizes the window

}

public void dispose(GLAutoDrawable drawable) {

// called when the panel is being disposed }

}
```

* * *

At this point, the only other thing you need to know is how to use OpenGL functions in the program. In JOGL, the OpenGL 1.1 functions are collected into an object of type GL2. (There are different classes for different versions of OpenGL; GL2 contains OpenGL 1.1 functionality, along with later versions that are compatible with 1.1.) An object of type GL2 is an OpenGL graphics context, in the same way that an object of type Graphics2D is a graphics context for ordinary Java 2D drawing. The statement

```
GL2 gl = drawable.getGL().getGL2();
```

in the above program obtains the drawing context for the GLAutoDrawable, that is, for the GLJPanel in that program. The name of the variable could, of course, be anything, but *gl* or *gl2* is conventional.

For the most part, using OpenGL functions in JOGL is the same as in C, except that the functions are now methods in the object *gl*. For example, a call to *glClearColor*(*r,g,b,a*) becomes gl.glClearColor(r,g,b,a);

The redundant "gl.gl" is a little annoying, but you get used to it. OpenGL constants such as

GL TRIANGLES are static members of GL2, so that, for example, *GL TRIANGLES* becomes *GL2.GL TRIANGLES* in JOGL. Parameter lists for OpenGL functions are the same as in the C API in most cases. One exception is for functions such as *glVertex3fv*() that take an array/pointer parameter in C. In JOGL, the parameter becomes an ordinary Java array, and an extra integer parameter is added to give the position of the data in the array. Here, for example, is how one might draw a triangle in JOGL, with all the vertex coordinates in one array:

float[] coords = { 0,0.5F, -0.5F,-0.5F, 0.5F,-0.5F };

gl.glBegin(GL2.GL _TRIANGLES);

gl.glVertex2fv(coords, 0);   // first vertex data starts at index 0 gl.glVertex2fv(coords, 2);
        // second vertex data starts at index 2

gl.glVertex2fv(coords, 4);   // third vertex data starts at index 4 gl.glEnd();

The biggest change in the JOGL API is the use of nio buffers instead of arrays in functions such as *glVertexPointer*. This is discussed in Subsection 3.4.3. We will see in Subsection 4.3.9 that texture images also get special treatment in JOGL.

* * *

The JOGL API includes a class named GLUT that makes GLUT's shape-drawing functions available in Java. (Since you don't need GLUT's window or event functions in Java, only the shape functions are included.) Class GLUT is defined in the package *com.jogamp.opengl.util.gl2*. To draw shapes using this class, you need to create an object of type GLUT. It's only necessary to make one of these for use in a program:

GLUT glut = new GLUT();

The methods in this object include all the shape-drawing functions from the GLUT C API, with the same names and parameters. For example:

glut.glutSolidSphere(2, 32, 16); glut.glutWireTeapot(5); glut.glutSolidIcosahedron();

(I don't know why these are instance methods in an object rather than static methods in a class; logically, there is no need for the object.)

The GLU library is available through the class *com.jogamp.opengl.glu.GLU*, and it works similarly to GLUT. That is, you have to create an object of type GLU, and the GLU functions will be available as methods in that object. We have encountered GLU only for the functions *gluLookAt* and *gluPerspective*, which are discussed in Section 3.3. For example,

```
GLU glu = new GLU(); glu.gluLookAt( 5,15,7, 0,0,0, 0,1,0 );
```

3.6.3 About glsim.js

The JavaScript library *glsim.js* was written to accompany and support this textbook. It implements the subset of OpenGL 1.1 that is discussed in Chapter 3 and Chapter 4, except for display lists (Subsection 3.4.4). It is used in the demos that appear in the on-line versions of those chapters. Many of the sample programs that are discussed in those chapters are available in JavaScript versions that use glsim.js.

If you would like to experiment with OpenGL 1.1, but don't want to go through the trouble of setting up a C or Java environment that supports OpenGL programming, you can consider writing your programs as web pages using glsim.js. Note that glsim is meant for experimentation and practice only, not for serious applications.

The OpenGL API that is implemented by glsim.js is essentially the same as the C API, although some of the details of semantics are different. Of course the techniques for creating a drawing surface and an OpenGL drawing context are specific to JavaScript and differ from those used in GLUT or JOGL.

To use glsim.js, you need to create an HTML document with a <canvas> element to serve as the drawing surface. The HTML file has to import the script; if glsim.js is in the same directory as the HTML file, you can do that with

```
<script src="glsim.js"></script>
```

To create the OpenGL drawing context, use the JavaScript command glsimUse(canvas);

where *canvas* is either a string giving the *id* of the <canvas> element or is the JavaScript DOM object corresponding to the <canvas> element. Once you have created the drawing context in this way, any OpenGL commands that you give will apply to the canvas. To run the program, you just need to open the HTML document in a web browser that supports WebGL 1.0.

The easiest way to get started programming is to modify a program that already exists. The sample program *glsim/first-triangle.html*, from Subsection 3.1.2 is a very minimal example of using glsim.js. The sample web page *glsim/glsim-starter.html* can be used as a starting point for writing longer programs that use glsim.js. It provides a framework for doing OpenGL drawing, with support for animation and mouse and keyboard events. The code contains comments that tell you how to use it. Some documentation for the glsim.js library can be found in *glsim/glsimdoc.html*. All of these files are part of the web site for this textbook and can be found in the web site download in a folder named *glsim* inside the *source* folder