Fundamentals of Computer Graphics

With Java, OpenGL, and Jogl

(Preliminary Partial Version, May 2010)

David J. Eck

Hobart and William Smith Colleges

David J. Eck (eck@hws.edu)
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY 14456

The web site for this book is: http://math.hws.edu/graphicsnotes

Contents

Preface

These notes represent an attempt to develop a new computer graphics course at the advanced undergraduate level. The primary goal, as in any such course, is to cover the fundamental concepts of computer graphics, and the concentration is on graphics in three dimensions. However, computer graphics has become a huge and complex field, and the typical textbook in the field covers much more material than can reasonably fit into a one-semester undergraduate course. Furthermore, the selection of topics can easily bury what should be an exciting and applicable subject under a pile of detailed algorithms and equations. These details are the basis of computer graphics, but they are to a large extent built into graphics systems. While practitioners should be aware of this basic material in a general way, there are other things that are more important for students on an introductory level to learn.

These notes were written over the course of the Spring semester, 2010. More information can be found on the web page for the course at http://math.hws.edu/eck/cs424/. The notes cover computer graphics programming using Java. Jogl is used for three-dimensional graphics programming. Jogl is the Java API for OpenGL; OpenGL is a standard and widely used graphics API. While it is most commonly used with the C programming language, Jogl gives Java programmers access to all the features of OpenGL. The version of Jogl that was used in this course was 1.1.1a. A new version, Jogl 2, was under development as the course was being taught, but Jogl 2 is still listed as a "work in progress" in May 2010. (Unfortunately, it looks like Jogl 1.1.1a will not be upward compatible with Jogl 2, so code written for the older version will not automatically work with the new version. However, the changes that will be needed to adapt code from this book to the new version should not be large.)

* * *

As often happens, not as much is covered in the notes as I had hoped, and even then, the writing gets a bit rushed near the end. A number of topics were covered in the course that did not make it into the notes. Some examples from those topics can be found in Chapter 5 (which is not a real chapter). In addition to OpenGL, the course covered two open-source graphics programs, GIMP briefly and Blender in a little more depth. Some of the labs for the course deal with these programs. Here are the topics covered in the four completed chapters of the book:

- **Chapter 1**: Java Graphics Fundamentals in Two Dimensions. This chapter includes a short general discussion of graphics and the distinction between "painting" and "drawing." It covers some features of the *Graphics2D* class, including in particular the use of geometric transforms for geometric modeling and animation.
- **Chapter 2**: Overview of OpenGL and Jogl. This chapter introduces drawing with OpenGL in both two and three dimensions, with very basic color and lighting. Drawing in this chapter uses the OpenGL routines `glBegin` and `glEnd`. It shows how to use Jogl to write OpenGL applications and applets. It introduces the use of transforms and scene

graphs to do hierarchical modeling and animation. It introduces the topic of optimization of graphics performance by covering display lists.

- **Chapter 3**: Geometric Modeling. This chapter concentrates on composing scenes in three dimensions out of geometric primitives, including the use of vector buffer objects and the OpenGL routines `glDrawArrays` and `glDrawArrayElements`. And it covers viewing, that is, the projection of a 3D scene down to a 2D picture.
- **Chapter 4**: Color, Lighting, and Materials. This chapter discusses how to add color, light, and textures to a 3D scene, including the use of lights in scene graphs.

Note that source code for all the examples in the book can be found in the source directory on-line or in the web site download.

* * *

The web site and the PDF versions of this book are produced from a common set of sources consisting of XML files, images, Java source code files, XSLT transformations, UNIX shell scripts, and a couple other things. These source files require the Xalan XSLT processor and (for the PDF version) the TeX typesetting system. The sources require a fair amount of expertise to use and were not written to be published. However, I am happy to make them available upon request.

* * *

Professor David J. Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, New York 14456, USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/

Chapter 1

Java Graphics Fundamentals in Two Dimensions

The focus of this course will be three-dimensional graphics using OpenGL. However, many important ideas in computer graphics apply to two dimensions in much the same way that they apply to three—often in somewhat simplified form. So, we will begin in this chapter with graphics in two dimensions. For this chapter only, we will put OpenGL to the side and will work with the standard Java two-dimensional graphics API. Some of this will no doubt be review, but you will probably encounter some corners of that API that are new to you.

1.1 Vector Graphics and Raster Graphics

Computer graphics can be divided broadly into two kinds: ***vector graphics*** and ***raster graphics***. In both cases, the idea is to represent an image. The difference is in how the image is represented.

An image that is presented on the computer screen is made up of ***pixels***. The screen consists of a rectangular grid of pixels, arranged in rows and columns. The pixels are small enough that they are not easy to see individually, unless you look rather closely. At a given time, each pixel can show only one color. Most screens these days use ***24-bit color***, where a color can be specified by three 8-bit numbers, giving the levels of red, green, and blue in the color. Other formats are possible, such as ***grayscale***, where a color is given by one number that specifies the level of gray on a black-to-white scale, or even ***monochrome***, where there is a single bit per pixel that tells whether the pixel is on or off. In any case, the color values for all the pixels on the screen are stored in a large block of memory known as a ***frame buffer***. Changing the image on the screen requires changing all the color values in the frame buffer. The screen is redrawn many times per second, so almost immediately after the color values are changed in the frame buffer, the colors of the pixels on the screen will be changed to match, and the displayed image will change.

A computer screen used in this way is the basic model of raster graphics. The term "raster" technically refers to the mechanism used on older vacuum tube computer monitors: An electron beam would move along the rows of pixels, making them glow. The beam could be moved across the screen by powerful magnets that would deflect the path of the electrons. The stronger the beam, the brighter the glow of the pixel, so the brightness of the pixels could be controlled by modulating the intensity of the electron beam. The color values stored in the frame buffer were used to determine the intensity of the electron beam. (For a color screen, each pixel had a red dot, a green dot, and a blue dot, which were separately illuminated by the beam.)

A modern flat-screen computer monitor is not a raster in the same sense. There is no moving electron beam. The mechanism that controls the colors of the pixels is different for different types of screen. But the screen is still made up of pixels, and the color values for all the pixels are still stored in a frame buffer. The idea of an image consisting of a grid of pixels, with numerical color values for each pixel, defines raster graphics.

* * *

Although images on the computer screen are represented using pixels, specifying individual pixel colors is not always the best way to create an image. Another way to create an image is to specify the basic geometric shapes that it contains, shapes such as lines, circles, triangles, and rectangles. This is the idea that defines vector graphics: represent an image as a list of the geometric shapes that it contains. To make things more interesting, the shapes can have ***attributes***, such as the thickness of a line or the color that fills a rectangle. Of course, not every image can be composed from simple geometric shapes. This approach certainly wouldn't work for a picture of a beautiful sunset (or for most any other photographic image). However, it works well for many types of images, such as architectural blueprints and scientific illustrations.

In fact, early in the history of computing, vector graphics were even used directly on computer screens. When the first graphical computer displays were developed, raster displays were too slow and expensive to be practical. Fortunately, it was possible to use vacuum tube technology in another way: The electron beam could be made to directly draw a line on the screen, simply by sweeping the beam along that line. A vector graphics display would store a ***display list*** of lines that should appear on the screen. Since a point on the screen would glow only very briefly after being illuminated by the electron beam, the graphics display would go through the display list over and over, continually redrawing all the lines on the list. To change the image, it would only be necessary to change the contents of the display list. Of course, if the display list became too long, the image would start to flicker because a line would have a chance to visibly fade before its next turn to be redrawn.

But here is the point: For an image that can be specified as a reasonably small number of geometric shapes, the amount of information needed to represent the image is much smaller using a vector representation than using a raster representation. Consider an image made up of one thousand line segments. For a vector representation of the image, you only need to store the coordinates of two thousand points, the endpoints of the lines. This would take up only a few kilobytes of memory. To store the image in a frame buffer for a raster display would require much more memory, even for a monochrome display. Similarly, a vector display could draw the lines on the screen more quickly than a raster display could copy the the same image from the frame buffer to the screen. (As soon as raster displays became fast and inexpensive, however, they quickly displaced vector displays because of their ability to display all types of images reasonably well.)

* * *

The divide between raster graphics and vector graphics persists in several areas of computer graphics. For example, it can be seen in a division between two categories of programs that can be used to create images: ***painting programs*** and ***drawing programs***. In a painting program, the image is represented as a grid of pixels, and the user creates an image by assigning colors to pixels. This might be done by using a "drawing tool" that acts like a painter's brush, or even by tools that draw geometric shapes such as lines or rectangles, but the point is to color the individual pixels, and it is only the pixel colors that are saved. To make this clearer, suppose that you use a painting program to draw a house, then draw a tree in front of the house. If you then erase the tree, you'll only reveal a blank canvas, not a house. In fact, the image never really contained a "house" at all—only individually colored pixels that the viewer might perceive as making up a picture of a house.

In a drawing program, the user creates an image by adding geometric shapes, and the image is represented as a list of those shapes. If you place a house shape (or collection of shapes making up a house) in the image, and you then place a tree shape on top of the house, the house is still there, since it is stored in the list of shapes that the image contains. If you delete the tree, the house will still be in the image, just as it was before you added the tree. Furthermore, you should be able to select any of the shapes in the image and move it or change its size, so drawing programs offer a rich set of editing operations that are not possible in painting programs. (The reverse, however, is also true.)

A practical program for image creation and editing might combine elements of painting and drawing, although one or the other is usually dominant. For example, a drawing program might allow the user to include a raster-type image, treating it as one shape. A painting program might let the user create "layers," which are separate images that can be layered one on top of another to create the final image. The layers can then be manipulated much like the shapes in a drawing program (so that you could keep both your house and your tree, even if in the image the house is in back of the tree).

Two well-known graphics programs are *Adobe Photoshop* and *Adobe Illustrator*. *Photoshop* is in the category of painting programs, while *Illustrator* is more of a drawing program. In the world of free software, the GNU image-processing program, *Gimp* is a good alternative to *Photoshop*, while *Inkscape* is a reasonably capable free drawing program.

* * *

The divide between raster and vector graphics also appears in the field of graphics file formats. There are many ways to represent an image as data stored in a file. If the original image is to be recovered from the bits stored in the file, the representation must follow some exact, known specification. Such a specification is called a graphics file format. Some popular graphics file formats include GIF, PNG, JPEG, and SVG. Most images used on the Web are GIF, PNG, or JPEG, and some web browsers also have support for SVG images.

GIF, PNG, and JPEG are basically raster graphics formats; an image is specified by storing a color value for each pixel. The amount of data necessary to represent an image in this way can be quite large. However, the data usually contains a lot of redundancy, and the data can be ***compressed*** to reduce its size. GIF and PNG use ***lossless data compression***, which means that the original image can be recovered perfectly from the compressed data. (GIF is an older file format, which has largely been superseded by PNG, but you can still find GIF images on the web.)

JPEG uses a ***lossy data compression*** algorithm, which means that the image that is recovered from a JPEG image is not exactly the same as the original image—some information has been lost. This might not sound like a good idea, but in fact the difference is often not very noticeable, and using lossy compression usually permits a greater reduction in the size of the compressed data. JPEG generally works well for photographic images, but not as well for images that have sharp edges between different colors. It is especially bad for line drawings and images that contain text; PNG is the preferred format for such images.

SVG is fundamentally a vector graphics format (although SVG images can contain raster images). SVG is actually an XML-based language for describing two-dimensional vector graphics images. "SVG" stands for "Scalable Vector Graphics," and the term "scalable" indicates one of the advantages of vector graphics: There is no loss of quality when the size of the image is increased. A line between two points can be drawn at any scale, and it is still the same perfect geometric line. If you try to greatly increase the size of a raster image, on the other hand, you will find that you don't have enough color values for all the pixels in the new image; each pixel in the original image will cover a rectangle of images in the scaled image, and you will get large visible blocks of uniform color. The scalable nature of SVG images make them a good choice for web browsers and for graphical elements on your computer's desktop. And indeed, some desktop environments are now using SVG images for their desktop icons.

* * *

When we turn to 3D graphics, the most common techniques are more similar to vector graphics than to raster graphics. That is, images are fundamentally composed out of geometric shapes. Or, rather, a "model" of a three-dimensional scene is built from geometric shapes, and the image is obtained by "projecting" the model onto a two-dimensional viewing surface. The three-dimensional analog of raster graphics **is** used occasionally: A region in space is divided into small cubes called ***voxels***, and color values are stored for each voxel. However, the amount of data can be immense and is wasted to a great extent, since we generally only see the surfaces of objects, and not their interiors, in any case. Much more common is to combine two-dimensional raster graphics with three-dimensional geometry: A two-dimensional image can be projected onto the surface of a three-dimensional object. An image used in this way is referred as a ***texture***.

Both Java and OpenGL have support for both vector-type graphics and raster-type graphics in two dimensions. In this course, we will generally be working with geometry rather than pixels, but you will need to know something about both. In the rest of this chapter, we will be looking at Java's built-in support for two-dimensional graphics.

1.2 Two-dimensional Graphics in Java

Java's support for 2D graphics is embodied primarily in two abstract classes, *Image* and *Graphics*, and in their subclasses. The *Image* class is mainly about raster graphics, while *Graphics* is concerned primarily with vector graphics. This chapter assumes that you are familiar with the basics of the *Graphics* class and the related classes *Color* and *Font*, including such *Graphics* methods as *drawLine*, *drawRect*, *fillRect*, *drawString*, *getColor*, *setColor*, and *setFont*. If you need to review them, you can read Section 6.3 of *Introduction to Programming Using Java*.

The class *java.awt.Image* really represents the most abstract idea of an image. You can't do much with the basic *Image* other than display it on a drawing surface. For other purposes, you will want to use the subclass, *java.awt.image.BufferedImage*. A *BufferedImage* represents a rectangular grid of pixels. It consists of a "raster," which contains color values for each pixel in the image, and a "color model," which tells how the color values are to be interpreted. (Remember that there are many ways to represent colors as numerical values.) In general, you don't have to work directly with the raster or the color model. You can simply use methods in the *BufferedImage* class to work with the image.

Java's standard class *java.awt.Graphics* represents the ability to draw on a two-dimensional drawing surface. A *Graphics* object has the ability to draw geometric shapes on its associated drawing surface. (It can also draw strings of characters and *Images*.) The *Graphics* class is suitable for many purposes, but its capabilities are still fairly limited. A much more complete two-dimensional drawing capability is provided by the class *java.awt.Graphics2D*, which is a subclass of *Graphics*. In fact, all the *Graphics* objects that are provided for drawing in modern Java are actually of type *Graphics2D*, and you can type-cast the variable of type *Graphics* to *Graphics2D* to obtain access to the additional capabilities of *Graphics2D*. For example, a *paintComponent* method that needs access to the capabilities of a *Graphics2D* might look like this:

```
protected void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D)g.create();
       ...  // Use g2 for drawing.
}
```

The method *g.create* creates a new graphics context object that has exactly the same properties as *g*. You can then use *g2* for drawing, and make any changes to it that you like without having any effect on *g*. It is recommended not to make any permanent changes to *g* in the *paintComponent* method, but you are free to do anything that you like to *g2*.

1.2.1 BufferedImages

There are basically two ways to get a *BufferedImage*. You can create a blank image using a constructor from the *BufferedImage* class, or you can get a copy of an existing image by reading the image from a file or some other source. The method

```
public boolean read(File source)
```

in the class *javax.imageio.ImageIO* can be used to read an image from a file. Supported image types include at least PNG and JPEG. Other methods in the *ImageIO* class make it possible to read an image from an *InputStream* or *URL*.

When you want to create a blank image, the constructor that you are most likely to use is

```
public BufferedImage(int width, int height, int imageType)
```

The *width* and *height* specify the size of image, that is, the number of rows and columns of pixels. The *imageType* specifies the color model, that is, what kind of color value is stored for each pixel. The *imageType* can be specified using a constant defined in the *BufferedImage* class. For basic full-color images, the type *BufferedImage.TYPE_INT_RGB* can be used; this specifies a color model that uses three eight-bit numbers for each pixel, giving the red, green, and blue color components. It is possible to add an eight-bit "alpha," or transparency, value for each pixel. To do that, use the image type *BufferedImage.TYPE_INT_ARGB*. Grayscale images can be created using the image type *BufferedImage.TYPE_BYTE_GRAY*.

Once you have a *BufferedImage*, you might want to be able to modify it. The easiest way is to treat the image as a drawing surface and to draw on it using an object of type *Graphics2D*. The method

```
public Graphics2D createGraphics()
```

in the *BufferedImage* class returns a *Graphics2D* object that can be used to draw on the image using the same set of drawing operations that are commonly used to draw on the screen.

It is also possible to manipulate the color of individual pixels in the image. The *BufferedImage* class has methods `public int getRGB(int x, int y)` and `public void setRGB(int x, int y, int rgb)` for getting and setting the color of an individual pixel. There are also methods for getting and setting the color values of a large number of pixels at once. See the API documentation for more information. (For these methods, colors are specified by 32-bit **int** values, with eight bits each for the alpha, red, green, and blue components of the color.)

Once you have your complete image, you can save it to a file using a *write* method from the *ImageIO* class. You can also copy the image to a drawing surface (including another image) using one of the *drawImage* methods from the *Graphics* class, such as

```
public boolean drawImage(Image image, int x, int y, ImageObserver obs)
```

This method draws the *image* at its normal size with its upper left corner at the point (x,y). The *ImageObserver* is meant for use when drawing an image that is still being loaded, for example, over the Internet; it makes it possible to draw a partially loaded image (and the **boolean** return value of the method tells whether the image has been fully drawn when the method returns); the *ImageObserver* is there to make sure that the rest of the image gets drawn when it becomes available. Since a *BufferedImage* is always already in memory, you just pass `null` as the fourth parameter when drawing a *BufferedImage*, and you can ignore the return value.

One important use of *BufferedImages* is to keep a copy of an image that is being displayed on the screen. For example, in a paint program, a *BufferedImage* can be used to store a copy of the image that is being created or edited. I sometimes call an image that is used in this way an ***off-screen canvas***. Any changes that are made to the image are applied to the off-screen canvas, which is then copied to the screen (for example, in the *paintComponent* method of a *JPanel*). You can then draw extra stuff, such as a box around a selected area, over the on-screen image without affecting the image in the off-screen canvas. And you can perform operations on the off-screen canvas, such as reading the color of a given pixel, that you can't do on the on-screen image. Using an off-screen image in this way is referred to as ***double buffering***, where the term "buffer" is used in the sense of a "frame buffer" that stores color values for the pixels in an image. One advantage of double-buffering is that the user doesn't see changes as they are being made to an image; you can compose an image in the off-screen canvas and then copy that image all at once onto the screen, so that the user sees only the completed image. (In fact, Java uses double-buffering automatically for Swing components such as *JPanel*. The *paintComponent* method actually draws to an off-screen canvas and the result is copied onto the screen. However, you don't have direct access to the off-screen canvas that is used for this purpose, and for many applications, you need to create an off-screen canvas of your own.)

1.2.2 Shapes and Graphics2D

You should be familiar with *Graphics* methods such as *drawLine* and *fillRect*. For these methods, points are specified in pixel coordinates, where a pair of integers (x,y) picks out the pixel in column x and in row y. Using pixel coordinates allows you to determine precisely which pixels will be affected by a given drawing operation, and sometimes such precision is essential. Often, however, coordinates are most naturally expressed on some other coordinate system. For example, in an architectural drawing, it would be natural to use actual physical measurements as coordinates. When drawing a mathematical object such as the graph of a function, it would be nice to be able to match the coordinate system to the size of the object that is being displayed. In such cases, the natural coordinates must be laboriously converted into pixel coordinates. Another problem with pixel coordinates is that when the size of the drawing area changes, all the coordinates have to recomputed, at least if the size of the image is to change to match the change in size of the drawing area. For this reason, computer graphics applications usually allow the specification of an arbitrary coordinate system. Shapes can then be drawn using the coordinates that are most natural to the application, and the graphics system will do the work of converting those coordinates to pixel coordinates.

The *Graphics2D* class supports the use of arbitrary coordinate systems. Once you've specified a coordinate system, you can draw shapes using that coordinate system, and the *Graphics2D* object will correctly transform the shape into pixel coordinates for you, before drawing it. Later in the chapter, we will see how to specify a new coordinate system for a *Graphics2D*. For now, we will consider how to draw shapes in such coordinate systems.

Note that the coordinate system that I am calling "pixel coordinates" is usually referred to as ***device coordinates***, since it is the appropriate coordinate system for drawing directly to the display device where the image appears. Java documentation refers to the coordinate system that is actually used for drawing as ***user coordinates***, although the "user" in this case is the programmer. Another name, more appropriate to OpenGL, is ***world coordinates***, corresponding to the idea that there is a natural coordinate system for the "world" that we want to represent in the image.

When using general coordinate systems, coordinates are specified as real numbers rather than integers (since each unit in the coordinate system might cover many pixels or just a small fraction of a pixel). The package *java.awt.geom* provides support for shapes defined using real number coordinates. For example, the class *Line2D* in that package represents line segments whose endpoints are given as pairs of real numbers. (Although the older drawing methods such as *drawLine* use integer coordinates, it's important to note that any shapes drawn using these methods are subject to the same transformation as shapes such as *Line2Ds* that are specified with real number coordinates. For example, drawing a line with *g.drawLine*(1,2,5,7) will have the same effect as drawing a *Line2D* that has endpoints (1.0,2.0) and (5.0,7.0). In fact, all drawing is affected by the transformation of coordinates, even, somewhat disturbingly, the width of lines and the size of characters in a string.)

Java has two primitive real number types: **double** and **float**. The **double** type can represent a larger range of numbers, with a greater number of significant digits, than **float**, and **double** is the more commonly used type. In fact, **doubles** are simply easier to use in Java. There is no automatic conversion from **double** to **float**, so you have to use explicit type-casting to convert a **double** value into a **float**. Also, a real-number literal such as 3.14 is taken to be of type **double**, and to get a literal of type **float**, you have to add an "F": 3.14F. However, **float** values generally have enough accuracy for graphics applications, and they have the advantage of taking up less space in memory. Furthermore, computer graphics hardware often uses float values internally.

So, given these considerations, the *java.awt.geom* package actually provides two versions of each shape, one using coordinates of type **float** and one using coordinates of type **double**. This is done in a rather strange way. Taking *Line2D* as an example, the class *Line2D* itself is an abstract class. It has two subclasses, one that represents lines using **float** coordinates and one using **double** coordinates. The strangest part is that these subclasses are defined as static nested classes inside *Line2D*: *Line2D.Float* and *Line2D.Double*. This means that you can declare a variable of type *Line2D*, but to create an object, you need to use *Line2D.Double* or *Line2D.Float*:

```
Line2D line1 = new Line2D.Double(1,2,5,7); // Line from (1.0,2.0) to (5.0,7.0)
Line2D line2 = new Line2D.Float(2.7F,3.1F,1.5F,7.1F); // (2.7,3.1) to (1.5,7.1)
```

This gives you, unfortunately, a lot of rope with which to hang yourself, and for simplicity, you might want to stick to one of the types *Line2D.Double* or *Line2D.Float*, treating it as a basic type. *Line2D.Double* will be more convenient to use. *Line2D.Float* might give better performance, especially if you also use variables of type **float** to avoid type-casting.

* * *

Let's take a look at some of the classes in package *java.awt.geom*. I will discuss only some of their properties; see the API documentation for more information. The abstract class *Point2D*—

and its concrete subclasses *Point2D.Double* and *Point2D.Float*—represents a point in two dimensions, specified by two real number coordinates. A point can be constructed from two real numbers ("`new Point2D.Double(1.2,3.7)`"). If p is a variable of type *Point2D*, you can use $p.getX()$ and $p.getY()$ to retrieve its coordinates, and you can use $p.setX(x)$, $p.setY(y)$, or $p.setLocation(x,y)$ to set its coordinates. If *pd* is a variable of type *Point2D.Double*, you can also refer directly to the coordinates as *pd.x* and *pd.y* (and similarly for *Point2D.Float*). Other classes in *java.awt.geom* offer a similar variety of ways to manipulate their properties, and I won't try to list them all here.

In addition to *Point2D*, *java.awt.geom* contains a variety of classes that represent geometric shapes, including *Line2D*, *Rectangle2D*, *RoundRectangle2D*, *Ellipse2D*, *Arc2D*, and *Path2D*. All of these are abstract classes, and each of them contains a pair of subclasses such as *Rectangle2D.Double* and *Rectangle2D.Float*. (Note that *Path2D* is new in Java 6.0; in older versions, you can use *GeneralPath*, which is equivalent to *Path2D.Float*.) Each of these classes implements the interface *java.awt.Shape*, which represents the general idea of a geometric shape. Some shapes, such as rectangles, have ***interiors*** that can be ***filled***; such shapes also have outlines that can be ***stroked***. Some shapes, such as lines, are purely one-dimensional and can only be stroked.

Aside from lines, rectangles are probably the simplest shapes. A *Rectangle2D* has a corner point (x,y), a *width*, and a *height*, and can be constructed from that data ("`new Rectangel2D.Double(x,y,w,h)`"). The corner point (x,y) specify the minimum x- and y-values in the rectangle. For the usual pixel coordinate system, (x,y) is the upper left corner. However, in a coordinate system in which the minimum value of y is at the bottom, (x,y) would be the lower left corner. The sides of the rectangle are parallel to the coordinate axes. A variable r of type *Rectangle2D.Double* or *Rectangle2D.Float* has public instance variables *r.x*, *r.y*, *r.width*, and *r.height*. If the width or the height is less than or equal to zero, nothing will be drawn when the rectangle is filled or stroked. A common problem is to define a rectangle by two corner points (*x1*,*y1*) and (*x2*,*y2*). This can be accomplished by creating a rectangle with height and width equal to zero and then *adding* the second point to the rectangle:

```
Rectangle2D.Double r = new Rectangle2D.Double(x1,y1,0,0);
r.add(x2,y2);
```

Adding a point to a rectangle causes the rectangle to grow just enough to include that point.

An *Ellipse2D* that just fits inside a given rectangle can be constructed from the upper left corner, width, and height of the rectangle ("`new Ellipse2D.Double(x,y,w,h)`"). A *RoundRectangle2D* is similar to a plain rectangle, except that an arc of an ellipse has been cut off each corner. The horizontal and vertical radius of the ellipse are part of the data for the round rectangle ("`new RoundRectangle2D.Double(x,y,w,h,r1,r2)`"). An *Arc2D* represents an arc of an ellipse. The data for an *Arc2D* is the same as the data for an *Ellipse2D*, plus the start angle of the arc, the end angle, and a "closure type." The angles are given in degrees, where zero degrees is in the positive direction of the x-axis. The closure types are *Arc2D.CHORD* (meaning that the arc is closed by drawing the line back from its final point back to its starting point), *Arc2D.PIE* (the arc is closed by drawing two line segments, giving the form of a wedge of pie), or *Arc2D.OPEN* (the arc is not closed).

The *Path2D* shape is the most interesting, since it allows the creation of shapes consisting of arbitrary sequences of lines and curves. The curves that can be used are quadratic and cubic ***Bezier curves***, which are defined by polynomials of degree two or three. A *Path2D* p is empty when it is first created ("`p = new Path2D.Double()`"). You can then construct the path by moving an imaginary "pen" along the path that you want to create. The method *p.moveTo*(*x*,*y*) moves the pen to the point (*x*,*y*) without drawing anything. It is used to specify the initial point of the path or the starting point of a new segment of the path. The method *p.lineTo*(*x*,*y*) draws a line from the current pen position to (*x*,*y*), leaving the pen at (*x*,*y*). The method *p.close*() can be used to close the path (or the current segment of the path) by drawing a line back to its starting point. (Note that curves don't have to be closed.) For example, the following code creates a triangle with vertices at (0,5), (2,−3), and (−4,1):

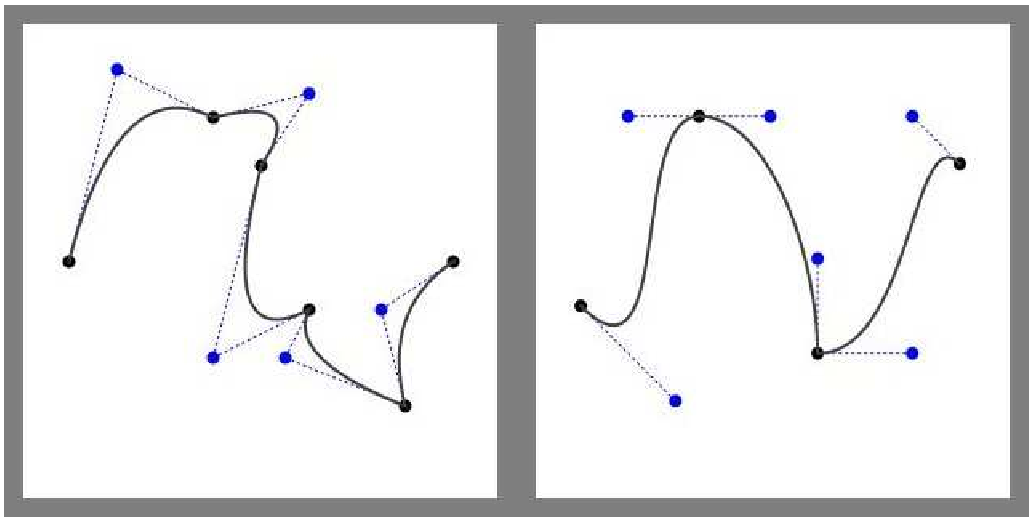
```
Path2D p = new Path2D.Double();
p.moveTo(0,5);
p.lineTo(2,-3);
p.lineTo(-4,1);
p.close();
```

For Bezier curves, you have to specify more than just the endpoints of the curve. You also have to specify ***control points***. Control points don't lie on the curve, but they determine the velocity or tangent of the curve at the endpoints. A quadratic Bezier curve has one control point. You can add a quadratic curve to a *Path2D* *p* using the method *p.quadTo*(*cx*,*cy*,*x*,*y*). The quadratic curve has endpoints at the current pen position and at (*x*,*y*), with control point (*cx*,*cy*). As the curve leaves the current pen position, it heads in the direction of (*cx*,*cy*), with a speed determined by the distance between the pen position and (*cx*,*cy*). Similarly, the curve heads into the point (*x*,*y*) from the direction of (*cx*,*cy*), with a speed determined by the distance from (*cx*,*cy*) to (*x*,*y*). Note that the control point (*cx*,*cy*) is **not** on the curve—it just controls the direction of the curve. A cubic Bezier curve is similar, except that it has two control points. The first controls the velocity of the curve as it leaves the initial endpoint of the curve, and the second controls the velocity as it arrives at the final endpoint of the curve. You can add a Bezier curve to a *Path2D* *p* with the method

```
p.curveTo( cx1, cy1, cx2, xy2, x, y )
```

This adds a Bezier curve that starts at the current pen position and ends at (*x*,*y*), using (*cx1*,*cy1*) and (*cx2*,*cy2*) as control points.

To make this clearer, the picture on the left below shows a path consisting of five quadratic Bezier curves, with their control points. Note that at each endpoint of a curve, the curve is tangent to the line from that endpoint to the control point. (In the picture, these lines are shown as dotted lines.) The picture on the right shows a path that consists of four cubic Bezier curves, with their control points. Note that by choosing appropriate control points, you can always ensure that one cubic Bezier curve joins smoothly to the next, as is done at the point where the first curve joins the second in the picture here.



The source code files *QuadraticBezierEdit.java* and *CubicBezierEdit.java* define applications that let you edit the curves shown in this image by dragging the control points and endpoints of the curve. You can also find applet versions of these applications in the web-site version of these notes.

* * *

Once you have a *Path2D* or a *Line2D*, or indeed any object of type *Shape*, you can use it with a *Graphics2D*. The *Graphics2D* class defines the methods

```
public void draw(Shape s)
public void fill(Shape s)
```

for drawing and filling *Shapes*. A *Graphics2D* has an instance variable of type *Paint* that tells how shapes are to be filled. *Paint* is just an interface, but there are several standard classes that implement that interface. One example is class *Color*. If the current paint in a *Graphics2D* *g2* is a *Color*, then calling *g2.fill*(*s*) will fill the *Shape* *s* with solid color. However, there are other types of *Paint*, such as *GradientPaint* and *TexturePaint*, that represent other types of fill. You can set the *Paint* in a *Graphics2D* *g2* by calling *g2.setPaint*(*p*). For a *Color* *c*, calling *g2.setColor*(*c*) also sets the paint.

Calling *g2.draw*(*s*) will "stroke" the shape by moving an imaginary pen along the lines or curves that make up the shape. You can use this method, for example, to draw a line or the boundary of a rectangle. By default, the shape will be stroked using a pen that is one unit wide. In the usual pixel coordinates, this means one pixel wide, but in a non-standard coordinate system, the pen will be one unit wide and one unit high, according the units of that coordinate system. The pen is actually defined by an object of type *Stroke*, which is another interface. To set the stroke property of a *Graphics2D* *g2*, you can use the method *g2.setStroke*(*stroke*). The *stroke* will almost certainly be of type *BasicStroke*, a class that implements the *Stroke* interface. For example to draw lines with *g2* that are 2.5 times as wide as usual, you would say

```
g2.setStroke( new BasicStroke(2.5F) );
```

Note that the parameter in the constructor is of type **float**. *BasicStroke* has several alternative constructors that can be used to specify various characteristics of the pen, such as how an endpoint should be drawn and how one piece of a curve should be joined to the next. Most interesting, perhaps, is the ability to draw dotted and dashed lines. Unfortunately, the details are somewhat complicated, but as an example, here is how you can create a pen that draws a pattern of dots and dashes:

```
float[] pattern = new float[] { 1, 3, 5, 3 };  // pattern of lines and spaces
BasicStroke pen = new BasicStroke( 1, BasicStroke.CAP_BUTT,
                                        BasicStroke.JOIN_ROUND, 0, pattern, 0 );
```

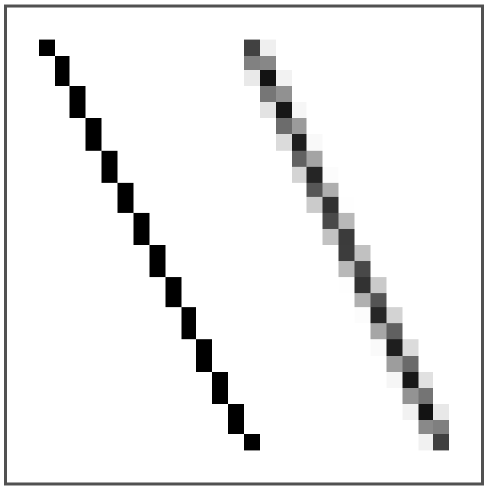
This pen would draw a one-unit-long dot, followed by a 3-unit-wide space, followed by a 5-unit-long dash, followed by a 3-unit-wide space, with the pattern repeating after that. You can adjust the pattern to get different types of dotted and dashed lines.

Note that in addition to the current *Stroke*, the effect of *g2.draw*(s) also depends on the current *Paint* in *g2*. The area covered by the pen is filled with the current paint.

* * *

Before leaving the topic of drawing with *Graphics2D*, I should mention antialiasing. ***Aliasing*** is a problem for raster-type graphics in general, caused by the fact that raster images are made up of small, solid-colored pixels. It is not possible to draw geometrically perfect pictures by coloring pixels. A diagonal geometric line, for example, will cover some pixels only partially, and it is not possible to make a pixel half black and half white. When you try to draw a line with black and white pixels only, the result is a jagged staircase effect. This effect is an example of aliasing. Aliasing can also be seen in the outlines of characters drawn on the screen and in diagonal or curved boundaries between any two regions of different color. (The term aliasing likely comes from the fact that most pictures are naturally described in real-number coordinates. When you try to represent the image using pixels, many real-number coordinates will map to the same integer pixel coordinates; they can all be considered as different names or "aliases" for the same pixel.)

Antialiasing is a term for techniques that are designed to mitigate the effects of aliasing. The idea is that when a pixel is only partially covered by a shape, the color of the pixel should be a mixture of the color of the shape and the color of the background. When drawing a black line on a white background, the color of a partially covered pixel would be gray, with the shade of gray depending on the fraction of the pixel that is covered by the line. (In practice, calculating this area exactly for each pixel would be too difficult, so some approximate method is used.) Here, for example, are two lines, greatly magnified so that you can see the individual pixels. The one on the right is drawn using antialiasing, while the one on the left is not:



Note that antialiasing does not give a perfect image, but it can reduce the "jaggies" that are caused by aliasing.

You can turn on antialiasing in a *Graphics2D*, and doing so is generally a good idea. To do so, simply call

```
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
```

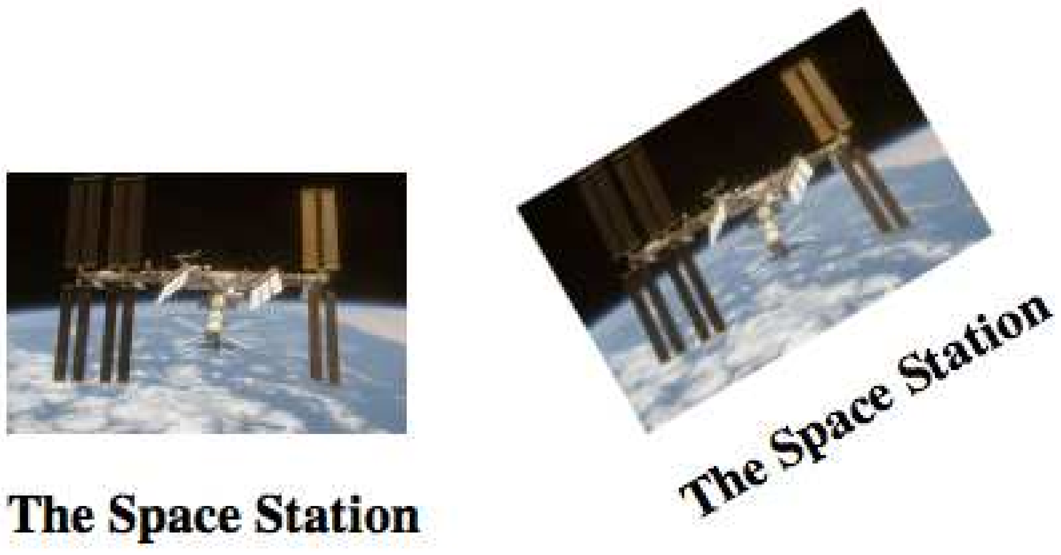
before using the *Graphics2D g2* for drawing. Antialiasing can be applied to lines, to text, and to the curved shapes such as ellipses. Note that turning on antialiasing is considered to be a "hint," which means that its exact effect is not guaranteed, and it might even have no effect at all. In general, though, it does give an improved image. Turning antialiasing on does slow down the drawing process, and the slow-down might be noticeable when drawing a very complex image.

1.3 Transformations and Modeling

We now turn to another aspect of two-dimensional graphics in Java: geometric transformations. The material in this section will carry over nicely to three dimensions and OpenGL. It is an important foundation for the rest of the course.

1.3.1 Geometric Transforms

Being able to draw geometric shapes is important, but it is only part of the story. Just as important are ***geometric transforms*** that allow you to modify the shapes that you draw, such as by moving them, changing their size, or rotating them. Some of this, you can do by hand, by doing your own calculations. For example, if you want to move a shape three units to the right, you can simply add 3 to each of the horizontal coordinates that are used to draw the shape. This would quickly become tedious, however, so its nice that we can get the computer to do it for us simply by specifying an appropriate geometric transform. Furthermore, there are things that you simply can't do in Java without transforms. For example, when you draw a string of text without a transform, the baseline of the text can only be horizontal. There is no way to draw the string tilted to a 30-degree angle. Similarly, the only way to draw a tilted image is by applying a transform. Here is an example of a string and an image drawn by Java in their normal orientation and with a counterclockwise 30-degree rotation:



Every *Graphics2D* graphics context has a current transform that is applied to all drawing that is done in that context. You can change the transform, and the new value will apply to all **subsequent** drawing operations; it will not affect things that have already been drawn in the graphics context. Generally, in a newly created *Graphics2D*, the current transform is the ***identity transform***, which has no effect at all on drawing. (There are some exceptions to this. For example, when drawing to a printer, the pixels are very small. If the same coordinates are used when drawing to the printer as are used on the screen, the resulting picture would be tiny. So, when drawing to a printer, Java provides a default transform that magnifies the picture to make it look about the same size as it does on the screen.)

The transform in a *Graphics2D* can be any ***affine transform***. An affine transform has the property that when it is applied to any set of parallel lines, the result is also a set of parallel lines (or, possibly, a single line or even a single point). An affine transform in two dimensions can be specified by six numbers a, b, c, d, e, and f, which have the property that when the transform is applied to a point (x,y), the resulting point $(x1,y1)$ is given by the formulas:

```
x1 = a*x + b*y + e
y1 = c*x + d*y + f
```

Affine transforms have the important property that if you apply one affine transform and then follow it with a second affine transform, the result is another affine transform. For example, moving each point three units to the right is an affine transform, and so is rotating each point by 30 degrees about the origin (0,0). Therefore the combined transform that first moves a point to the right and then rotates it is also an affine transform. Combining two affine transformations in this way is called ***multiplying*** them (although it is not multiplication in the usual sense). Mathematicians often use the term ***composing*** rather than multiplying.

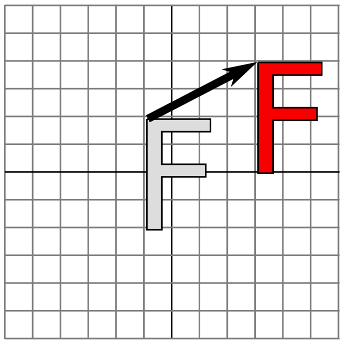
It's possible to build up any two-dimensional affine transformation from a few basic kinds of simple transforms: ***translation***, ***rotation***, and ***scaling***. It's good to have an intuitive understanding of these basic transforms and how they can be used, so we will go through them in some detail. The effect of a transform depends on what coordinate system you are using for drawing. For this discussion, we assume that the origin, (0,0), is at the center of the picture, with the positive direction of the x-axis pointing to the right and the positive direction of the y-axis pointing up. Note that this orientation for the y-axis is the opposite of the usual orientation in Java; it is, however, the more common orientation in mathematics and in OpenGL.

* * *

A **translation** simply moves each point by a certain amount horizontally and a certain amount vertically. The formula for a translation is

```
x1 = x + e
y1 = y + f
```

where the point (x,y) is moved e units horizontally and f units vertically. (Thus for a translation, $a = d = 1$, and $b = c = 0$.) If you wanted to apply this translation to a *Graphics2D* g, you could simply say *g.translate*(e,f). This would mean that for all subsequent drawing operations, e would be added to the x-coordinate and f would be added to the y-coordinate. Let's look at an example. Suppose that you are going to draw an "F" centered at (0,0). If you say *g.translate*(4,2) **before** drawing the "F", then every point of the "F" will be moved over 4 units and up 2 units, and the "F" that appears on the screen will actually be centered at (4,2). Here is a picture:



The light gray "F" in this picture shows what would be drawn without the translation; the dark red "F" shows the same "F" drawn after applying a translation by (4,2). The arrow shows that the upper left corner of the "F" has been moved over 4 units and up 2 units. Every point in the "F" is subjected to the same displacement.

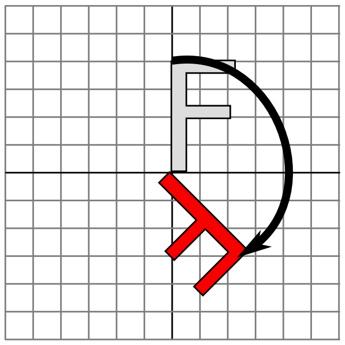
Remember that when you say *g.translate*(e,f), the translation applies to **all** the drawing that you do after that, not just to the next shape that you draw. If you apply another transformation to the same g, the second transform will not replace the translation. It will be multiplied by the translation, so that subsequent drawing will be affected by the combined transformation. This is an important point, and there will be a lot more to say about it later.

* * *

A **rotation** rotates each point about the origin, (0,0). Every point is rotated through the same angle, called the angle of rotation. For this purpose, angles in Java are measured in radians, not degrees. For example, the measure of a right angle is $\pi/2$, not 90. Positive angles move the positive x-axis in the direction of the positive y-axis. (This is counterclockwise in the coordinate system that we are using here, but it is clockwise in the usual pixel coordinates, where the y-axis points down rather than up.) Although it is not obvious, when rotation through an angle of r radians about the origin is applied to the point (x,y), then the resulting point $(x1,y1)$ is given by

```
x1 = cos(r) * x - sin(r) * y
y1 = sin(r) * x + cos(r) * y
```

That is, in the general formula for an affine transform, $e = f = 0$, $a = d = \cos(r)$, $b = -\sin(r)$, and $c = \sin(r)$. Here is a picture that illustrates a rotation about the origin by the angle $-3\pi/4$:

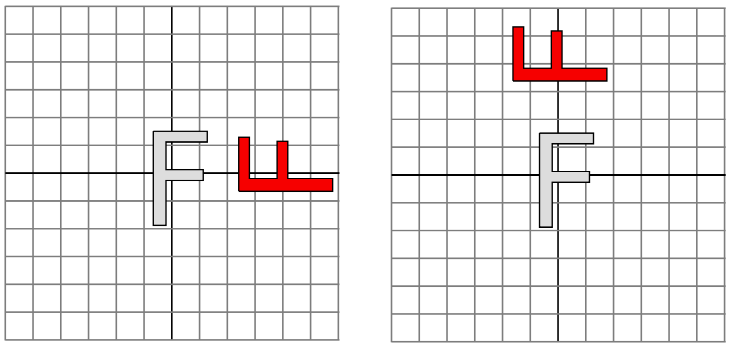


Again, the light gray "F" is the original shape, and the dark red "F" is the shape that results if you apply the rotation. The arrow shows how the upper left corner of the original "F" has been moved.

In a *Graphics2D* g, you can apply a rotation through an angle r by saying $g.rotate(r)$ **before** drawing the shapes that are to be rotated. We are now in a position to see what can happen when you combine two transformations. Suppose that you say

```
    g.translate(4,0);
    g.rotate(Math.PI/2);
```

before drawing some shape. The translation applies to all subsequent drawing, and the thing that you draw after the translation is a **rotated** shape. That is, the translation applies to a shape to which a rotation has already been applied. An example is shown on the **left** in the illustration below, where the light gray "F" is the original shape. The dark red "F" shows the result of applying the two transforms. The "F" has first been rotated through a 90 degree angle, and then moved 4 units to the right.



Transforms are applied to shapes in the reverse of the order in which they are given in the code (because the first transform in the code is applied to a shape that has already been affected by the second transform). And note that the order in which the transforms are applied is important. If we reverse the order in which the two transforms are applied in this example, by saying

```
    g.rotate(Math.PI/2);
    g.translate(4,0);
```

then the result is as shown on the **right** in the above illustration. In that picture, the original "F" is first moved 4 units to the right and the resulting shape is then rotated through an angle of $\pi/2$ about the origin to give the shape that actually appears on the screen.

For a useful application of using several transformations, suppose that we want to rotate a shape through an angle r about a point (p,q) instead of about the point (0,0). We can do this by first moving the point (p,q) to the origin with *g.translate*$(-p,-q)$. Then we can do a standard rotation about the origin by calling *g.rotate*(r). Finally, we can move the origin back to the point (p,q) using *g.translate*(p,q). Keeping in ming that we have to apply the transformations in the reverse order, we can say

```
g.translate(p,q);
g.rotate(r);
g.translate(-p,-q);
```

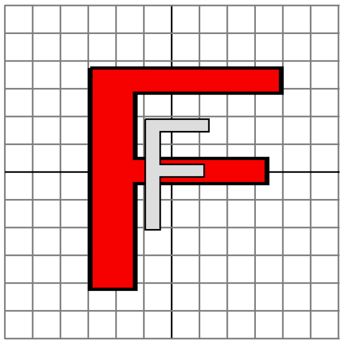
before drawing the shape. In fact, though, you can do the same thing in Java with one command: *g.rotate*(r,p,q) will apply a rotation of r radians about the point (p,q) to subsequent drawing operations.

* * *

A **scaling** transform can be used to make objects bigger or smaller. Mathematically, a scaling transform simply multiplies each x-coordinate by a given amount and each y-coordinate by a given amount. That is, if a point (x,y) is scaled by a factor of a in the x direction and by a factor of d in the y direction, then the resulting point $(x1,y1)$ is given by

```
x1 = a * x
y1 = d * y
```

If you apply this transform to a shape that is centered at the origin, it will stretch the shape by a factor of a horizontally and d vertically. Here is an example, in which the original light gray "F" is scaled by a factor of 3 horizontally and 2 vertically to give the final dark red "F":



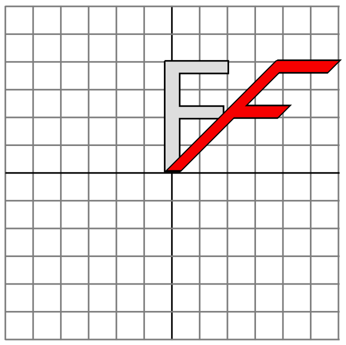
The common case where the horizontal and vertical scaling factors are the same is called ***uniform scaling***. Uniform scaling stretches or shrinks a shape without distorting it. Note that negative scaling factors are allowed and will result in reflecting the shape as well as stretching or shrinking it.

When scaling is applied to a shape that is not centered at (0,0), then in addition to being stretched or shrunk, the shape will be moved away from 0 or towards 0. In fact, the true description of a scaling operation is that it pushes every point away from (0,0) or pulls them towards (0,0). (If you want to scale about a point other than (0,0), you can use a sequence of three transforms, similar to what was done in the case of rotation. Java provides no shorthand command for this operation.)

To scale by (a,d) in *Graphics2D* g, you can call $g.scale(a,d)$. As usual, the transform applies to all x and y coordinates in subsequent drawing operations.

* * *

We will look at one more type of basic transform, a ***shearing transform***. Although shears can in fact be built up out of rotations and scalings if necessary, it is not really obvious how to do so. A shear will "tilt" objects. A horizontal shear will tilt things towards the left (for negative shear) or right (for positive shear). A vertical shear tilts them up or down. Here is an example of horizontal shear:



A horizontal shear does not move the x-axis. Every other horizontal line is moved to the left or to the right by an amount that is proportional to the y-value along that line. When a horizontal shear is applied to a point (x,y), the resulting point ($x1$,$y1$) is given by

```
x1 = x + b * y
y1 = y
```

for some constant shearing factor b. Similarly, a vertical shear by a shearing factor c has equations

```
x1 = x
y1 = c * x + y
```

In Java, the method for applying a shear to a *Graphics2D* g allows you to specify both a horizontal shear factor b and a vertical shear factor c: $g.shear(b,c)$. For a pure horizontal shear, you can set c to zero; for a pure vertical shear, set b to zero.

* * *

In Java, an affine transform is represented by an object belonging to the class *java.awt.geom.AffineTransform*. The current transform in a *Graphics2D* g is an object of type *AffineTransform*. Methods such as *g.rotate* and *g.shear* multiply that current transform object by another transform. There are also methods in the *AffineTransform* class itself for multiplying the transform by a translation, a rotation, a scaling, or a shear. Usually, however, you will just use the corresponding methods in the *Graphics2D* class.

You can retrieve the current affine transform from a *Graphics2D* g by calling $g.getTransform()$, and you can replace the current transform by calling $g.setTransform(t)$. In general, it is recommended to use *g.setTransform* only for restoring a previous transform in g, after temporarily modifying it.

One use for an *AffineTransform* is to compute the ***inverse transform***. The inverse transform for a transform t is a transform that exactly reverses the effect of t. Applying t followed by the inverse of t has no effect at all—it is the same as the identity transform.

Not every transform has an inverse. For example, there is no way to undo a scaling transform that has a scale factor of zero. Most transforms, however, do have inverses. For the basic transforms, the inverses are easy. The inverse of a translation by (e,f) is a translation by $(-e,-f)$. The inverse of rotation by an angle r is rotation by $-r$. The inverse of scaling by (a,d) is scaling by $(1/a,1/d)$, provided a and d are not zero.

For a general *AffineTransform* t, you can call *t.createInverse*() to get the inverse transform. This method will throw an exception of type *NoninvertableTransformException* if t does not have an inverse. This is a checked exception, which requires handling, for example with a `try..catch` statement.

Sometimes, you might want to know the result of applying a transform or its inverse to a point. For an *AffineTransform* t and points *p1* and *p2* of type *Point2D*, you can call

```
t.transform( p1, p2 );
```

to apply t to the point *p1*, storing the result of the transformation in *p2*. (It's OK for *p1* and *p2* to be the same object.) Similarly, to apply the inverse transform of t, you can call

```
t.inverseTransform( p1, p2 );
```

This statement will throw a *NoninvertibleTransformException* if t does not have an inverse.

* * *

You have probably noticed that we have discussed two different uses of transforms: ***Coordinate transforms*** change the coordinate system that is used for drawing. ***Modeling transforms*** modify shapes that are drawn. In fact, these two uses are two sides of the same coin. The same transforms that are used for one purpose can be used for the other. Let's see how to set up a coordinate system on a component such as a *JPanel*. In the standard coordinate system, the upper left corner is (0,0), and the component has a width and a height that give its size in pixels. The width and height can be obtained by calling the panel's *getWidth* and *getHeight* methods. Suppose that we would like to draw on the panel using a real-number coordinate system that extends from *x1* on the left to *x2* on the right and from *y1* at the top to *y2* at the bottom. A point (x,y) in these coordinates corresponds to pixel coordinates

```
( (x-x1)/(x2-x1) * width, (y-y1)/(y2-y1) * height )
```

To see this, note that `(x-x1)/(x2-x1)` is the distance of *x1* from the left edge of the panel, given as fraction of the total width, and similarly for the height. If we rewrite this as

```
( (x-x1) * (width/(x2-x1)), (y-y1) * (height/(y2-y1)) )
```

we see that the change in coordinates can be accomplished by first translating by `(-x1,-y1)` and then scaling by `(width/(x2-x1),height/(y2-y1))`. Keeping in mind that transforms are applied to coordinates in the reverse of the order in which they are given in the code, we can implement this coordinate transform in the panel's *paintComponent* method as follows

```
protected void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D)g.create();
   g2.scale( getWidth() / ( x2 - x1 ), getHeight() / ( y2 - y1) );
   g2.translate( -x1, -y1 );
      ...   // draw the content of the panel, using the new coordinate system
}
```

The scale and translate in this example set up the coordinate transformation. Any further transforms can be thought of as modeling transforms that are used to modify shapes. However, you can see that it's really just a nominal distinction.

The transforms used in this example will work even in the case where *y1* is greater than *y2*. This allows you to introduce a coordinate system in which the minimal value of y is at the bottom rather than at the top.

One issue that remains is ***aspect ratio***, which refers to the ratio between the height and the width of a rectangle. If the aspect ratio of the coordinate rectangle that you want to display does not match the aspect ratio of the component in which you will display it, then shapes will be distorted because they will be stretched more in one direction than in the other. For example, suppose that you want to use a coordinate system in which both x and y range from 0 to 1. The rectangle that you want to display is a square, which has aspect ratio 1. Now suppose that the component in which you will draw has width 800 and height 400. The aspect ratio for the component is 0.5. If you simply map the square onto the component, shapes will be stretched twice as much in the horizontal direction as in the vertical direction. If this is a problem, a solution is to use only part of the component for drawing the square. We can do this, for example, by padding the square on both sides with some extra x-values. That is, we can map the range of x values between -0.5 and 1.5 onto the component, while still using range of y values from 0 to 1. This will place the square that we really want to draw in the middle of the component. Similarly, if the aspect ratio of the component is greater than 1, we can pad the range of y values.

The general case is a little more complicated. The *applyLimits* method, shown below, can be used to set up the coordinate system in a *Graphics2D*. It should be called in the *paintCompoent* method before doing any drawing to which the coordinate system should apply. This method is used in the sample program *CubicBezierEdit.java*, so you can look there for an example of how it is used. When using this method, the parameter *limitsRequested* should be an array containing the *left*, *right*, *top*, and *bottom* edges of the coordinate rectangle that you want to display. If the *preserveAspect* parameter is *true*, then the actual limits will be padded in either the horizontal or vertical direction to match the aspect ratio of the coordinate rectangle to the width and height of the "viewport." (The viewport is probably just the entire component where *g2* draws, but the method could also be used to map the coordinate rectangle onto a different viewport.)

```
/**
 * Applies a coordinate transform to a Graphics2D graphics context.  The upper
 * left corner of the viewport where the graphics context draws is assumed to
 * be (0,0).  This method sets the global variables pixelSize and transform.
 *
 * @param g2 The drawing context whose transform will be set.
 * @param width The width of the viewport where g2 draws.
 * @param height The height of the viewport where g2 draws.
 * @param limitsRequested  Specifies a rectangle that will be visible in the
 *   viewport. Under the transform, the rectangle with corners (limitsRequested[0],
 *   limitsRequested[1]) and (limitsRequested[2],limitsRequested[3]) will just
 *   fit in the viewport.
 * @param preserveAspect if preserveAspect is false, then the limitsRequested
 *   rectangle will exactly fill the viewport; if it is true, then the limits
 *   will be expanded in one direction, horizontally or vertically, to make
 *   the aspect ratio of the displayed rectangle match the aspect ratio of the
 *   viewport.  Note that when preserveAspect is false, the units of measure in
 *   the horizontal and vertical directions will be different.
 */
private void applyLimits(Graphics2D g2, int width, int height,
                       double[] limitsRequested, boolean preserveAspect) {
   double[] limits = limitsRequested;
   if (preserveAspect) {
      double displayAspect = Math.abs((double)height / width);
      double requestedAspect = Math.abs(( limits[3] - limits[2] )
                                           / ( limits[1] - limits[0] ));
      if (displayAspect > requestedAspect) {
         double excess = (limits[3] - limits[2])
                               * (displayAspect/requestedAspect - 1);
         limits = new double[] { limits[0], limits[1],
                             limits[2] - excess/2, limits[3] + excess/2 };
      }
      else if (displayAspect < requestedAspect) {
         double excess = (limits[1] - limits[0])
                               * (requestedAspect/displayAspect - 1);
         limits = new double[] { limits[0] - excess/2, limits[1] + excess/2,
                                 limits[2], limits[3] };
      }
   }
   g2.scale( width / (limits[1]-limits[0]), height / (limits[3]-limits[2]) );
   g2.translate( -limits[0], -limits[2] );
   double pixelWidth = Math.abs(( limits[1] - limits[0] ) / width);
   double pixelHeight = Math.abs(( limits[3] - limits[2] ) / height);
   pixelSize = (float)Math.min(pixelWidth,pixelHeight);
   transform = g2.getTransform();
}
```

The last two lines of this method assign values to *pixelSize* and *transform*, which are assumed to be global variables. These are values that might be useful elsewhere in the program. For example, *pixelSize* might be used for setting the width of a stroke to some given number of pixels:

```
g2.setStroke( new BasicStroke(3*pixelSize) );
```

Remember that the unit of measure for the width of a stroke is the same as the unit of measure in the coordinate system that you are using for drawing. If you want a stroke—or anything else—that is measured in pixels, you need to know the size of a pixel.

As for the *transform*, it can be used for transforming points between pixel coordinates and drawing coordinates. In particular, suppose that you want to implement mouse interaction. The methods for handling mouse events will tell you the pixel coordinates of the mouse position. Sometimes, however, you need to know the drawing coordinates of that point. You can use the *transform* variable to make the transformation. Suppose that *evt* is a *MouseEvent*. You can use the *transform* variable from the above method in the following code to transform the point from the *MouseEvent* into drawing coordinates:

```
Point2D p = new Point2D.Double( evt.getX(), evt.getY() );
transform.inverseTransform( p, p );
```

The point *p* will then contain the drawing coordinates corresponding to the mouse position.

1.3.2 Hierarchical Modeling

A major motivation for introducing a new coordinate system is that it should be possible to use the coordinate system that is most natural to the scene that you want to draw. We can extend this idea to individual objects in a scene: When drawing an object, it should be possible to use the coordinate system that is most natural to the object. Geometric transformations allow us to specify the object using whatever coordinate systeme we want, and to apply a transformation to the object to place it wherever we want it in the scene. Transformations used in this way are called modeling transformations.

Usually, we want an object in its natural coordinates to be centered at the origin, (0,0), or at least to use the origin as a convenient reference point. Then, to place it in the scene, we can use a scaling transform, followed by a rotation, followed by a translation to set its size, orientation, and position in the scene. Since scaling and rotation leave the origin fixed, those operations don't move the reference point for the object. A translation can then be used to move the reference point, and the object along with it, to any other point. (Remember that in the code, the transformations are specified in the opposite order from the order in which they are applied to the object and that the transformations are specified before drawing the object.)

The modeling transformations that are used to place an object in the scene should not affect other objects in the scene. To limit their application to just the one object, we can save the current transformation before starting work on the object and restore it afterwards. The code could look something like this, where g is a *Graphics2D*:

```
AffineTransform saveTransform = g.getTransform();
g.translate(a,b); // move object into position
g.rotate(r); // set the orientation of the object
g.scale(s,s); // set the size of the object
  ...  // draw the object, using its natural coordinates
g.setTransform(saveTransform);  // restore the previous transform
```

Note that we can't simply assume that the original transform is the identity transform. There might be another transform in place, such as a coordinate transform, that affects the scene as a whole. The modeling transform for the object is effectively applied in addition to any other transform that was specified previously. The modeling transform moves the object from its natural coordinates into its proper place in the scene. Then on top of that, another transform is applied to the scene as a whole, carrying the object along with it.

Now let's extend this a bit. Suppose that the object that we want to draw is itself a complex picture, made up of a number of smaller objects. Think, for example, of a potted flower made up of pot, stem, leaves, and bloom. We would like to be able to draw the smaller component objects in their own natural coordinate systems, just as we do the main object. But this is easy: We draw each small object in its own coordinate system, and use a modeling transformation to move the small object into position *within the main object*. On top of that, we can apply *another* modeling transformation to the main object, to move it into the completed scene; its component objects are carried along with it. In fact, we can build objects that are made up of smaller objects which in turn are made up of even smaller objects, to any level. This type of modeling is known as ***hierarchical modeling***.

Let's look at a simple example. Suppose that we want to draw a simple 2D image of a cart with two wheels. We will draw the body of the cart as a rectangle. For the wheels, suppose that we have written a method

```
private void drawWheel(Graphics2D g2)
```

that draws a wheel. The wheel is drawn using its own natural coordinate system. Let's say that in this coordinate system, the wheel is centered at (0,0) and has radius 1.

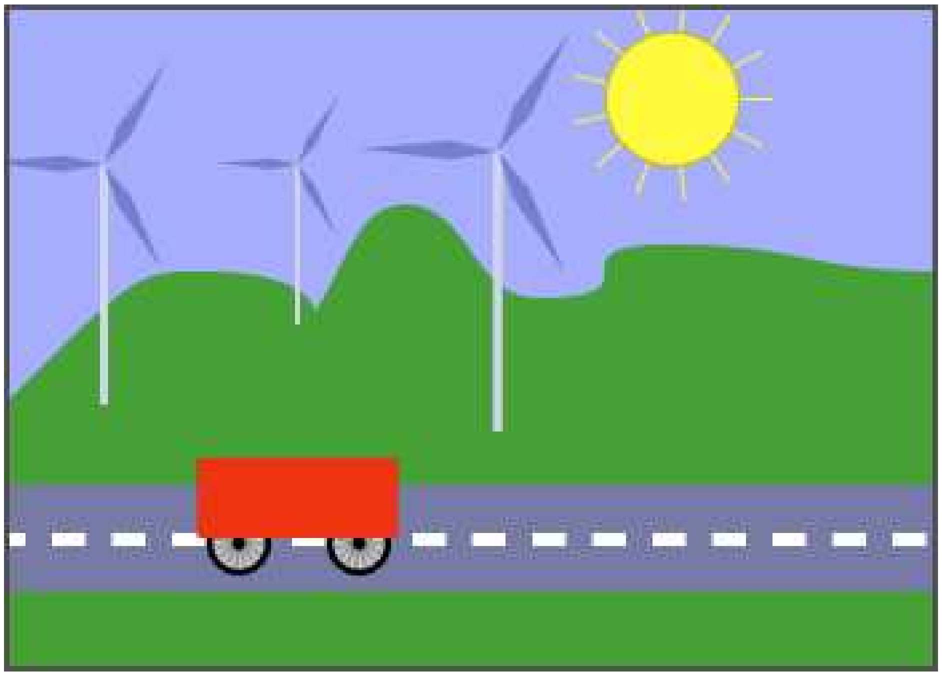
We will draw the cart with the center of its rectangular body at the point (0,0). The rectangle has width 5 and height 2, so its corner is at (−2.5,0). To complete the cart, we need *two* wheels. To make the size of the wheels fit the cart, we will probably have to scale them. To place them in the correct positions relative to body of the cart, we have to translate one wheel to the left and one wheel to the right. When I coded this example, I had to play around with the numbers to get the right sizes and positions for the wheels, and I also found that the wheels looked better if I also moved them down a bit. Using the usual techniques of hierarchical modeling, we have to remember to save the current transform and to restore it after drawing each wheel. Here is a subroutine that can be used to draw the cart:

```
private void drawCart(Graphics2D g2) {
    AffineTransform tr = g2.getTransform();  // save the current transform
    g2.translate(-1.5,-0.1);  // center of first wheel will be at (-1.5,-0.1)
    g2.scale(0.8,0.8);  // scale to reduce radius from 1 to 0.8
    drawWheel(g2); // draw the first wheel
    g2.setTransform(tr);  // restore the transform
    g2.translate(1.5,-0.1);  // center of second wheel will be at (1.5,-0.1)
    g2.scale(0.8,0.8); // scale to reduce radius from 1 to 0.8
    drawWheel(g2); // draw the second wheel
    g2.setTransform(tr);  // restore the transform
    g2.setColor(Color.RED);
    g2.fill(new Rectangle2D.Double(-2.5,0,5,2) ); // draw the body of the cart
}
```

It's important to note that the same subroutine is used to draw both wheels. The reason that two wheels appear in the picture is that different modeling transformations are in effect for the two subroutine calls. Once we have this cart-drawing subroutine, we can use it to add a cart to a scene. When we do this, we can apply another modeling transformation to the cart as a whole. Indeed, we could add several carts to the scene, if we want, by calling the *cart* subroutine several times with different modeling transformations.

You should notice the analogy here: Building up a complex scene out of objects is similar to building up a complex program out of subroutines. In both cases, you can work on pieces of the problem separately, you can compose a solution to a big problem from solutions to smaller problems, and once you have solved a problem, you can reuse that solution in several places.

Here is our cart used in a scene. The scene shows the cart on a road, with hills, windmills, and a sun in the background. The on-line applet version of this example is an ***animation*** in which the windmills turn and the cart rolls down the road.



You can probably see how hierarchical modeling is used to draw the windmills in this example. There is a *drawWindmill* method that draws a windmill in its own coordinate system. Each of the windmills in the scene is then produced by applying a different modeling transform to the standard windmill.

It might not be so easy to see how different parts of the scene can be animated. In fact, animation is just another aspect of modeling. A computer animation consists of a sequence of frames. Each frame is a separate image, with small changes from one frame to the next. From our point of view, each frame is a separate scene and has to be drawn separately. The same object can appear in many frames. To animate the object, we can simply apply a different modeling transformation to the object in each frame. The parameters used in the transformation can be computed from the current time or from the frame number. To make a cart move from left to right, for example, we might apply a translation

```
g2.translate(frameNumber * 0.1);
```

to the cart, where *frameNumber* is the frame number, which increases by 1 from one frame to the next. (The animation is driven by a *Timer* that fires every 30 milliseconds. Each time the timer fires, 1 is added to *frameNumber* and the scene is redrawn.) In each frame, the cart will be 0.1 units farther to the right than in the previous frame. (In fact, in the actual program, the translation that is applied to the cart is

```
g2.translate(-3 + 13*(frameNumber % 300) / 300.0, 0);
```

which moves the reference point of the cart from -3 to 13 along the horizontal axis every 300 frames.)

The really neat thing is that this type of animation works with hierarchical modeling. For example, the *drawWindmill* method doesn't just draw a windmill—it draws an *animated* windmill, with turning vanes. That just means that the rotation applied to the vanes depends on the frame number. When a modeling transformation is applied to the windmill, the rotating vanes are scaled and moved as part of the object as a whole. This is actually an example of hierarchical modeling. The vanes are sub-objects of the windmill. The rotation of the vanes is part of the modeling transformation that places the vanes into the windmill object. Then a further modeling transformation can be applied to the windmill object to place it in the scene.

The file *HierarchicalModeling2D.java* contains the complete source code for this example. You are strongly encouraged to read it, especially the *paintComponent* method, which composes the entire scene.

Chapter 2

The Basics of OpenGL and Jogl

The step from two dimensions to three dimensions in computer graphics is a big one. Three-dimensional objects are harder to visualize than two, and three-dimensional transformations take some getting used to. Three-dimensional scenes have to be projected down onto a two-dimensional surface to be viewed, and that introduces its own complexities.

Furthermore, there is the problem that realistic 3D scenes require the effects of lighting to make them appear three-dimensional to the eye. Without these effects, the eye sees only flat patches of color. To simulate lighting in 3D computer graphics, you have to know what light sources illuminate the scene, and you have to understand how the light from those sources interacts with the surface material of the objects in the scene.

As we pursue our study of OpenGL, we will cover all of this and more. We will start in this chapter with an overview that will allow you to use OpenGL to create fairly complex 3D scenes. In later chapters, we will cover more of the details and advanced features of OpenGL, and we will consider the general theory of 3D graphics in more depth.

2.1 Basic OpenGL 2D Programs

The first version of OpenGL was introduced in 1992. A sequence of new versions has gradually added features to the API, with the latest version being OpenGL 3.2, in 2009. In 2004, the power of OpenGL was greatly enhanced in version 2.0 with the addition of GLSL, the OpenGL Shading Language, which makes it possible to replace parts of the standard OpenGL processing with custom programs written in a C-like programming language. This course will cover nothing beyond OpenGL 2.1. (Indeed, I have no graphics card available to me that supports a later version.)

OpenGL is an API for creating images. It does not have support for writing complete applications. You can't use it to open a window on a computer screen or to support input devices such as a keyboard and mouse. To do all that, you need to combine OpenGL with a general-purpose programming language, such as Java. While OpenGL is not part of standard Java, libraries are available that add support for OpenGL to Java. The libraries are referred to as Jogl or as the Java Binding for OpenGL. We will be using Jogl version 1.1.1a (although Jogl 2.0 will be available soon).

In this section, we'll see how to to create basic two-dimensional scenes using Jogl and OpenGL. We will draw some simple shapes, change the drawing color, and apply some geometric transforms. This will already give us enough capabilities to do some hierarchical modeling and animation.

2.1.1 A Basic Jogl App

To use OpenGL to display an image in a Java program, you need a GUI component in which you can draw. Java's standard AWT and Swing components don't support OpenGL drawing directly, but the Jogl API defines two new component classes that you can use, *GLCanvas* and *GLJPanel*. These classes, like most of the Jogl classes that we will use in this chapter, are defined in the package *javax.media.opengl*. *GLCanvas* is a subclass of the standard AWT *Canvas* class, while *GLJPanel* is a subclass of Swing's *JPanel* class. Both of these classes implement an interface, *GLAutoDrawable*, that represents the ability to support OpenGL drawing.

While *GLCanvas* might offer better performance, it doesn't fit as easily into Swing applications as does *GLJPanel*. For now, we will use *GLJPanel*.

OpenGL drawing is not done in the *paintComponent* method of a *GLJPanel*. Instead, you should write an event listener object that implements the *GLEventListener* interface, and you should register that object to listen for events from the panel. It's the *GLEventListener* that will actually do the drawing, in the *GLEventListener* method

```
public void display(GLAutoDrawable drawable)
```

The *GLAutoDrawable* will be the *GLJPanel* (or other OpenGL drawing surface) that needs to be drawn. This method plays the same role as the *paintComponent* method of a *JPanel*. The *GLEventListener* class defines three other methods:

```
public void init(GLAutoDrawable drawable)
public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height)
public void displayChanged(GLAutoDrawable drawable, boolean modeChanged,
                                                    boolean deviceChanged)
```

The *init* method is called once, when the drawable is first created. It is analogous to a constructor. The *reshape* method is called when the drawable changes shape. The third method can always be left empty; it has to be there to satisfy the definition of a *GLEventListener*, but it will not be called (and in fact will not even exist in Jogl 2.0).

To actually do anything with OpenGL, you need an ***OpenGL context***, which is represented in Jogl by an object of type *GL*. You can get a context from a *GLAutoDrawable* by saying

```
GL gl = drawable.getGL()
```

Generally, this statement is the first line of the *init* method, of the *display* method, and of the *reshape* method in a *GLEventListener*. An object of type *GL* includes a huge number of methods, and the *GL* class defines a huge number of constants for use as parameters in those methods. We will spend much of our time in the rest of this course investigating the *GL* class. In fact, most of the OpenGL API is contained in this class.

* * *

We are just about ready to look at our first program! Here is a very simple but functional Jogl Application:

```
import java.awt.*;
import javax.swing.*;
import javax.media.opengl.*;

public class BasicJoglApp2D extends JPanel implements GLEventListener {

    public static void main(String[] args) {
        JFrame window = new JFrame("Basic JOGL App 2D");
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
            window.setContentPane(new BasicJoglApp2D());
            window.pack();
            window.setVisible(true);
        }

        public BasicJoglApp2D() {
            GLJPanel drawable = new GLJPanel();
            drawable.setPreferredSize(new Dimension(600,600));
            setLayout(new BorderLayout());
            add(drawable, BorderLayout.CENTER);
            drawable.addGLEventListener(this); // Set up events for OpenGL drawing!
        }

        public void init(GLAutoDrawable drawable) {
        }

        public void display(GLAutoDrawable drawable) {
            GL gl = drawable.getGL();

            gl.glClear(GL.GL_COLOR_BUFFER_BIT);  // Fill with background color.

            gl.glBegin(GL.GL_POLYGON);  // Start drawing a polygon.
            gl.glVertex2f(-0.5f,-0.3f); // First vertex of the polygon.
            gl.glVertex2f(0.5f,-0.3f);  // Second vertex
            gl.glVertex2f(0,0.6f);      // Third vertex
            gl.glEnd();                 // Finish the polygon.
        }

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

        public void displayChanged(GLAutoDrawable drawable, boolean modeChanged,
                boolean deviceChanged) {
        }
    }
```

The constructor in this class creates a *GLJPanel*, *drawable*, and adds it to the main panel. The main panel itself acts as a *GLEventListener*, and registers itself to listen for OpenGL events from *drawable*. This is done with the command `drawable.addGLEventListener(this)`. The *display* method is then the place where the content of the *GLJPanel* is drawn.

Let's look at the *display* method, since it's our first example of OpenGL drawing. This method starts, as discussed above, by getting an OpenGL context, *gl*. The rest of the method uses that context for all OpenGL drawing operations. The goal is to draw a triangle. Default settings are used for the background color (black), the drawing color (white), and the coordinate system (x coordinates ranging from -1 on the left to 1 on the right; y coordinates, from -1 on the bottom to 1 at the top).

The first step is to fill the entire component with the background color. This is done with the command

```
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
```

This command "clears" the drawing area to its background color. It's actually clearing the color buffer, where the color value for each pixel is stored. There are several buffers that can be cleared with this command, and the parameter tells which of them should be cleared. We'll see later how to set the background color.

The rest of the *display* method draws the triangle. Each of the vertices of the triangle is specified by calling the method *gl.glVertex2f*. This method just specifies points. To say what is *done* with the points, the calls to *gl.glVertex2f* have to come between *gl.glBegin* and *gl.glEnd*. The parameter to *gl.glBegin* gives the meaning of the points—in this case the fact that they are the vertices of a filled polygon. If we were to replace `GL.GL_POLYGON` with `GL.GL_LINE_LOOP` in the *glBegin* method, then the method would draw the outline of the triangle instead of a filled triangle.

You are probably thinking that "gl.glVertex2f" is a pretty strange name for a method. The names of the methods and constants come from the OpenGL API, which was designed with the C programming language in mind. It would have been possible to simplify the names in Java (and in particular to leave off the extra "gl"), but it was thought that programmers would be more comfortable with the more familiar names. The name "glVertex2f" actually has a very specific format. There is a basic name, "glVertex". Then the number 2 tells how many parameters there will be, and the "f" at the end says that the parameters are of type **float**. There is a version of the basic method that has two parameters of type **double**, and its name is "glVertex2**d**". When we move into 3D, the 2 will change to a 3, giving "glVertex**3**f" and "glVertex**3**d", since one needs three coordinates to specify a point in three dimensions. The names make sense. A lot of method names in OpenGL work the same way. Unfortunately, you can't always predict which possible variations of a name are actually defined.

Now that we have a sample Jogl application, we can use the same outline for other programs. We just need to change the *init*, *display*, and *reshape* methods.

2.1.2 Color

To set the drawing color in an OpenGL context *gl*, you can use one of the *glColor* methods, such as `gl.glColor3f(red,green,blue)`. For this variation, the parameters are three values of type **float** in the range 0 to 1. For example, `gl.glColor3f(1,1,0)` will set the color for subsequent drawing operations to yellow. (There is also a version, *glColor3d*, that takes parameters of type **double**.)

Suppose that we want to draw a yellow triangle with a black border. This can be done with a display method

```
public void display(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glColor3f(1.0f, 1.0f, 0.0f);      // Draw with yellow.
    gl.glBegin(GL.GL_POLYGON);          // Draw a filled triangle.
    gl.glVertex2f(-0.5f,-0.3f);
    gl.glVertex2f(0.5f,-0.3f);
    gl.glVertex2f(0,0.6f);
    gl.glEnd();
    gl.glColor3f(0,0,0);                 // Draw with black.
    gl.glBegin(GL.GL_LINE_LOOP);        // Draw the outline of the triangle.
    gl.glVertex2f(-0.5f,-0.3f);
    gl.glVertex2f(0.5f,-0.3f);
    gl.glVertex2f(0,0.6f);
    gl.glEnd();
}
```

That takes care of changing the drawing color. The background color is a little more complicated. There is no background color as such; there is a "clear color," which is used by the *glClear* method to fill the color buffer. The clear color can be set by calling `gl.glClearColor(red,blue,green,alpha)`. This method has only one version, and the parameters must be **float** values in the range 0 to 1. The parameters specify the color components of the color that will be used for clearing the color buffer, including an alpha component, which should ordinarily be set equal to 1. (Alpha components can be used for transparency, but alphas are a more complicated topic in OpenGL than they are in standard Java graphics, and we will avoid that topic for now.)

The clear color is often set once and for all, and then does not change between one call of *display* and the next. In that case, it is not necessary to set the clear color in *display*; it can be set in *init* instead. This illustrates the use of the *init* method for setting the values of OpenGL state variables that will remain the same throughout the program. Here is an example that sets the clear color to light blue:

```
public void init(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();
    gl.glClearColor(0.8f, 0.8f, 1, 1);
}
```

Note that this uses an important fact about OpenGL. An OpenGL context keeps many internal values in state variables. This includes, among many other things, the clear color, the drawing color, the lighting setup, and the current geometric transform. All these state variables **retain their values** between calls to *init*, *display*, and *reshape*. This is different from Java graphics, where every call to *paintComponent* starts with a new graphics context, initialized to default values. In OpenGL, if you are drawing in red at the end of one call to *display*, then you will be drawing in red at the beginning of the next call. If you want to be sure of drawing with some default drawing color at the beginning of *display*, you have to put in an explicit call to *glColor3f* to do so.

2.1.3 Geometric Transforms and Animation

Naturally, we don't want to spend too much time looking at still images. We need some action. We can make a simple animation by introducing a *frameNumber* instance variable to our program and adding *Timer* to drive the animation. When the timer fires, we will increment the frame number and call the *repaint* method of the *GLJPanel*. Calling *repaint* will in turn trigger a call to the *display* method where the actual OpenGL drawing is done. If we make that drawing depend on the frame number, we get an animation.

We can use modeling transforms for animation in OpenGL, just as we did with *Graphics2D* in Section 1.3. The main difference is that in OpenGL, transforms are meant to work in three dimensions rather than two. In three dimensions, there is a z coordinate, in addition to the usual x and y coordinates. We can still use 3D transforms to operate in 2D, as long as we make sure to leave the z coordinate unchanged.

For example, **translation** in OpenGL is done with *gl.glTranslatef(dx,dy,dz)* where the parameters specify the amount of displacement in the x, y, and z directions. By setting the third parameter, *dz*, to 0, we get a 2D translation. The parameters are of type **float**, as indicated by the "f" at the end of "translatef". If you want to use parameters of type **double**, you can use *gl.translated*. The names for the other transform methods work in the same way.

Similarly, **scaling** can be done with *gl.glScalef*(*sx*,*sy*,*sz*), where the parameters specify the scaling factor in the *x*, *y*, and *z* directions. To avoid changing the scale in the *z* direction, the third parameter, *sz*, should be 1.

Rotation transforms are a little more complicated, since rotation in 3D is more complicated than rotation in 2D. In two dimensions, rotation is rotation about a point, and by default, the rotation is about the origin, (0,0). In three dimensions, rotation is about a line called the ***axis of rotation***. Think of the rotating Earth spinning about its axis. To specify a rotation in OpenGL, you must specify both an angle of rotation and an axis of rotation. This can be done with *gl.glRotatef*(*d*,*x*,*y*,*z*), where *d* is the angle of rotation, measured in degrees, and the axis of rotation is the line through the origin (0,0,0) and the point (*x*,*y*,*z*). For a 2D rotation, you can set (*x*,*y*,*z*) to (0,0,1). That is, for a rotation of *d* degrees about the 2D origin (0,0), you should call *gl.glRotatef*(*d*,0,0,1). The direction of rotation is counterclockwise when *d* is positive.

As with *Graphics2D*, transforms apply to drawing that is done after the transform is specified, not to things that have already been drawn. And when there are multiple transforms, then they are applied to shapes in the reverse of the order in which they are specified in the code. For example, if we want to scale a 2D object by a factor of 1.8, rotate it 30 degrees, and then move it 3 units over and 1.5 units up, we could say

```
gl.glTranslatef( 3, 1.5f, 0 );
gl.glRotatef( 30, 0, 0, 1 );
gl.glScalef( 1.8f, 1.8f, 1 );
  ...  // draw the object
```

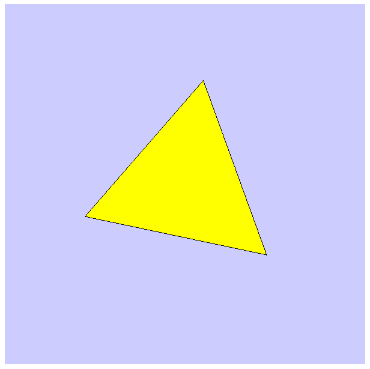
Keep in mind that the transforms apply to all subsequent drawing, not just to the next object drawn, unless you do something to restore the original transform. In fact, since OpenGL state persists across calls to *display*, the transform will **still** be in effect the next time *display* is called, unless you do something to prevent it! To avoid this problem, it's common to set the transform at the beginning of the *display* method to be the identity transform. This can be done by calling the method *gl.glLoadIdentity*().

Here, for example, is a *display* method that draws a rotated triangle:

```
public void display(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glLoadIdentity();                 // Start from the identity transform!
    gl.glRotated(frameNumber, 0, 0, 1);  // Apply a rotation of frameNumber degrees.
    gl.glColor3f(1.0f, 1.0f, 0.0f);
    gl.glBegin(GL.GL_POLYGON);           // Draw a yellow filled triangle.
    gl.glVertex2f(-0.5f,-0.3f);
    gl.glVertex2f(0.5f,-0.3f);
    gl.glVertex2f(0,0.6f);
    gl.glEnd();
    gl.glColor3f(0,0,0);
    gl.glBegin(GL.GL_LINE_LOOP);         // Outline the triangle with black.
    gl.glVertex2f(-0.5f,-0.3f);
    gl.glVertex2f(0.5f,-0.3f);
    gl.glVertex2f(0,0.6f);
    gl.glEnd();
}
```

Since the rotation depends on *frameNumber*, we can use this method in an animation of a spinning triangle. You can find the full source code for this simple animation program in

BasicJoglAnimation2D.java You can find an applet version of the program on-line. Here is one frame from the animation:



2.1.4 Hierarchical Modeling

When we do geometric modeling, we need a way to limit the effect of a transform to one object. That is, we should save the current transform before drawing the object, and restore that transform after the object has been drawn. OpenGL has methods that are meant to do precisely that. The command *gl.glPushMatrix*() saves a copy of the current transform. It should be matched by a later call to *gl.glPopMatrix*(), which restores the transform that was saved by *gl.glPushMatrix* (thus discarding any changes that were made to the current transform between the two method calls). So, the general scheme for geometric modeling is:

```
gl.glPushMatrix();
   ...  // Apply the modeling transform of the object.
   ...  // Draw the object.
gl.glPopMatrix();
```

("Matrix," by the way, is really just another name for "transform." It refers to the way in which transforms are represented internally.)

As you might suspect from the method names, transforms are actually saved on a **stack** of transforms. (Unfortunately, you cannot assume that this stack can grow to arbitrary size. However, OpenGL implementations must allow at least 32 transforms on the stack, which should be plenty for most purposes.) *glPushMatrix* adds a copy of the current transform to the stack; *glPopMatrix* removes the transform from the top of the stack and uses it to replace the current transform in the OpenGL context.

The use of a stack makes it easy to implement hierarchical modeling. You can draw a scene using *glPushMatrix* and *glPopMatrix* when placing each object into the scene, to limit transforms that are applied to an object to just that object. But each of the objects could itself be a complex object that uses *glPushMatrix* and *glPopMatrix* to draw its sub-objects. As long as each "push" is matched with a "pop," drawing a sub-object will not make any permanent change to the transform stack. In outline, the process would go something like this:

```
pushMatrix
apply overall object modeling transform
   pushMatrix
   apply first sub-object modeling transform
   draw first sub-object
   popMatrix
```

```
      pushMatrix
      apply second sub-object modeling transform
      draw second sub-object
      popMatrix
        .
        .
        .
   popMatrix
```

And keep in mind that the sub-objects could themselves be complex objects, with their own nested calls to *glPushMatrix* and *glPopMatrix*.

In *HierarchicalModeling2D.java*, in Subsection 1.3.2, we did some hierarchical modeling using Java *Graphics2D*. We're now in a position to do the same thing with Jogl. The sample program *JoglHierarchicalModeling2D.java* is an almost direct port of the *Graphics2D* example. It uses similar subroutines to draw a sun, a wheel, a cart, and a windmill. For example, here is the cart-drawing subroutine in OpenGL:

```
private void drawCart(GL gl) {

    gl.glPushMatrix();               // Draw the first wheel as a sub-object.
    gl.glTranslatef(-1.5f, -0.1f, 0);
    gl.glScalef(0.8f,0.8f,1);
    drawWheel(gl);
    gl.glPopMatrix();

    gl.glPushMatrix();               // Draw the second wheel as a sub-object.
    gl.glTranslatef(1.5f, -0.1f, 0);
    gl.glScalef(0.8f,0.8f,1);
    drawWheel(gl);
    gl.glPopMatrix();

    gl.glColor3f(1,0,0);            // Draw a rectangle for the body of the cart
    gl.glBegin(GL.GL_POLYGON);
    gl.glVertex2f(-2.5f,0);
    gl.glVertex2f(2.5f,0);
    gl.glVertex2f(2.5f,2);
    gl.glVertex2f(-2.5f,2);
    gl.glEnd();
}
```

When the *display* method draws the complete scene, it calls the *drawCart* method to add a cart to the scene. The cart requires its own modeling transformation to place it into the scene, and that transform requires its own calls to *glPushMatrix* and *glPopMatrix*:

```
gl.glPushMatrix();
gl.glTranslated(-3 + 13*(frameNumber % 300) / 300.0, 0, 0);
gl.glScaled(0.3,0.3,1);
drawCart(gl); // does its own pushes and pops internally!
gl.glPopMatrix();
```

I encourage you to look through the program, even though it does use a few techniques that we haven't covered yet.

2.1.5 Introduction to Scene Graphs

So far, we have been doing hierarchical modeling using subroutines to draw the objects. This works well, but there is another approach that goes well with object-oriented programming. That is, we can represent the graphical objects in a scene with software objects in the program. Instead of having a cart-drawing subroutine, we might have an object of type *Cart*, where *Cart* is a class that we have written to represent carts in the scene. (Or, as I will do in the actual example, we could have a *cart* object belonging to a more general class *ComplexObject2D*, which represents objects that are made up of sub-objects.) Note that the cart object would include references to the cart's sub-objects, and the scene as a whole would be represented by a linked collection of software objects. This linked data structure is an example of a ***scene graph***. A scene graph is a data structure that represents the contents of a scene. When you need to draw the scene, you can get all the information that you need by traversing the scene graph.

The sample program *JoglHMWithSceneGraph2D.java* is another version of our hierarchical modeling animation, this one using a scene graph. In this version, an object in the scene is represented in the program by an object of type *SceneNode2D*. *SceneNode2D* is an abstract class (defined in *scenegraph2D/SceneNode2D.java*). It has an abstract method, `public void draw(GL gl)`, which draws the object in an OpenGL context. A *SceneNode2D* can have a color, and it can have a transformation that is specified by separate scaling, rotation, and translation factors. The color in this case is an example of an ***attribute*** of a node in a scene graph. Attributes contain properties of objects beyond the purely geometric properties.

SceneNode2D has a subclass *ComplexObject2D* that represents an object that is made up of sub-objects. Each sub-object is of type *SceneNode2D* and can itself be a *ComplexObject2D*. There is an *add* method for adding a sub-object to the list of sub-objects in the complex object. The *draw* method in the *ComplexObject2D* class simply draws the sub-objects, with their proper colors and transformations. Note that the complex object can have its own transformation, which is applied to the complex object as a whole, on top of the transformations that are applied to the sub-objects. (The treatment of color for complex objects is also interesting. I have decided that the color of an object can be `null`, which is the default value. If the color of a sub-object is `null`, then its color will be inherited from the main object in which it is contained. The general question of how to handle attributes in scene graphs is an important design issue.)

SceneNode2D also has several subclasses to represent geometric primitives. For example, there is a class *LineNode* that represents a single line segment between two points. There is also *PolygonNode* and *DiskNode*. These geometric primitive classes represent the "bottom level" of the scene graph. They are objects that are not composed of simpler objects, and they are the fundamental building blocks of complex objects.

In the sample animation program, an instance variable named *theWorld* of type *ComplexObject2D* represents the entire content of the scene. This means that in the *display* method, drawing the content of the image requires only one line of code:

```
theWorld.draw(gl);
```

Most of the work has been moved into a method, *buildTheWorld*(), that constructs the scene graph. This method is called only once, at the beginning of the program. It's interesting to see how objects in the scene are constructed. Here, for example, is the code the creates a *wheel* for the cart:

```
ComplexObject2D wheel = new ComplexObject2D();
wheel.setColor(Color.BLACK);   // Color for most of the wheel.
wheel.add( new DiskNode(1) );  // The parameter is the radius of the disk.
DiskNode hubcap = new DiskNode(0.8);
hubcap.setColor(new Color(0.75f, 0.75f, 0.75f)); // Override color of hubcap.
wheel.add(hubcap);
wheel.add( new DiskNode(0.2) );
for (int i = 0; i < 15; i++) { // Add lines representing the wheel's spokes.
    double angle = (2*Math.PI/15) * i;
    wheel.add(new LineNode(0,0,Math.cos(angle),Math.sin(angle)));
}
```

Once this object is created, we can add it to the *cart* object. In fact, we can add it **twice**, with two different transformations. When the scene graph is traversed, the wheel will appear twice. The two occurrences will be in different locations because of the different transformations that are applied. To make things like this easier, I made a subclass *TransformedObject* of *SceneNode2D* to represent an object with an extra transform to be applied to that object. Here, then, is how the *cart* is created:

```
ComplexObject2D cart = new ComplexObject2D();
cart.setColor(Color.RED); // Overall color; in fact, will apply only to the body.
cart.add( new TransformedObject(wheel, 0.8, 0, -1.5, -0.1) ); // First wheel.
cart.add( new TransformedObject(wheel, 0.8, 0, 1.5, -0.1) );  // Second wheel.
cart.add( new PolygonNode(
               new double[] { -2.5, 0, 2.5, 0, 2.5, 2, -2.5, 2} ) ); // The body.
cart.setScaling(0.3);  // Suitable scaling for the overall scene.
```

We can now add the cart to the scene simply by calling `theWorld.add(cart)`. You can read the program to see how the rest of the scene graph is created and used. The classes that are used to build the scene graph can be found in the source directory *scenegraph2D*.

This leaves open the question of how to do animation when the objects are stored in a scene graph. For the type of animation that is done in this example, we just have to be able to change the transforms that are applied to the object. Before drawing each frame, we will set the correct transforms for that frame.

In the sample program, I store references to the objects representing the wheel, the cart, and the rotating vanes of the windmill in instance variables *wheel*, *cart*, and *vanes*. To animate these objects, the display methods includes commands to set the rotation or translation of the object to a value that depends on the current frame number. This is done just before the world is drawn:

```
wheel.setRotation(frameNumber*20);
vanes.setRotation(frameNumber * (180.0/46));
cart.setTranslation(-3 + 13*(frameNumber % 300) / 300.0, 0);

theWorld.draw(gl);
```

The scene graph framework that is used in this example is highly simplified and is meant as an example only. More about scene graphs will be coming up later. By the way, a scene graph is called a "graph" because it's generally in the form of a data structure known as a "directed acyclic graph."

2.1.6 Compiling and Running Jogl Apps

Since Jogl is not a standard part of Java, you can't run programs that use it unless you make the Jogl classes available to that program. The classes that you need for Jogl 1.1.1 are distributed in two jar files, *jogl.jar* and *gluegen-rt.jar*. These jar files must be on the Java classpath when you compile or run any program that uses Jogl.

Furthermore, Jogl uses "native code libraries." Native code is code that is written in the machine language that is appropriate to a particular kind of computer, rather than in Java. The interface to OpenGL requires some native code, which means that you have to make the appropriate native code libraries available to Java when you run (but not when you compile) any program that uses Jogl. Unfortunately, you need different native code libraries for Linux, for MacOS, and for Windows. That's why there is a different Jogl download for each platform. The download for a given platform includes the appropriate native code libraries for that platform. As I write this, the downloads for Jogl 1.1.1a for various platforms can be found at:

```
http://download.java.net/media/jogl/builds/archive/jsr-231-1.1.1a/
```

You will need to get the download for your platform if you want to develop Jogl programs on the command line or in the Eclipse IDE. However, if you use Netbeans (http://netbeans.org) as your development environment, you can avoid the whole issue. There is a Netbeans plugin that adds full support for Jogl/OpenGL development. If you install the plugin, you can develop and run your programs in Netbeans with no further action. As I write this, the current version of the plugin supports Jogl 1.1.1a, and information about it can be found at

```
http://kenai.com/projects/netbeans-opengl-pack/pages/Home
```

If Eclipse (http://eclipse.org) is your development environment, it's just a little harder. You need to get the Jogl 1.1.1a download for your platform. Unzip the zip file and put the directory in some convenient place. The directory has a *lib* subdirectory that contains the jar files and native code libraries that you need. While you are downloading, you might also want to get the Jogl documentation download, jogl-1.1.1a-docs.zip, and unzip it as well.

Then, when you want to use Jogl in an Eclipse project, you have to do the following setup in that project: Right-click the name of the project in the "Project Explorer" pane. In the pop-up menu, go to the "Build Path" submenu and select "Configure Build Path...". A dialog box will open. Click on the "Libraries" tab in the "Java Build Path" pane. Add the two required Jogl jar files to the build path. [Click the button "Add External Jars...". In the file dialog box, browse to the *lib* directory in the Jogl download. Select *jogl.jar* and click "OK". Now do the same thing for *gluegen-rt.jar*.] Finally, set up the jar files to use the Jogl native code libraries. [Click the triangle to the left of *jogl.jar* in the list of libraries. This will reveal a setting for "Native Library Location". Click it, then click the "Edit" button. Browse to the Jogl *lib* directory, and click OK. Do the same thing for *gluegen-rt.jar*.] If you downloaded the Jogl docs, you will also want to set up the "Javadoc Location" under "jogl.jar". Set it to point to the unzipped documentation directory.

If you've gotten all the steps right, you can then compile and run Jogl programs in your Eclipse project.

* * *

If you like to work on the command line, you can still do that with Jogl. When using *javac* to compile a class that uses Jogl, you will need to add the Jogl jar file, *jogl.jar*, to the classpath. For example:

```
javac -cp /home/eck/jogl-1.1.1/lib/jogl.jar:. packagename/*.java
```

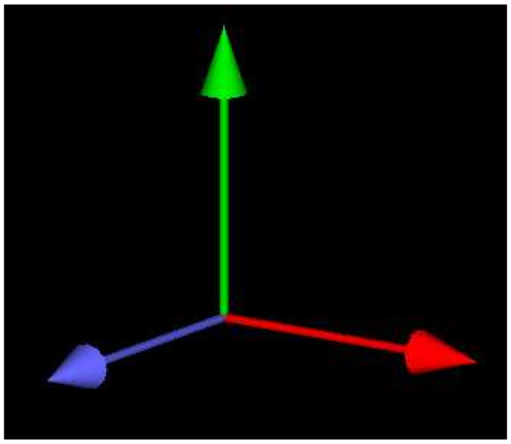
Of course, you need to replace */home/eck/jogl-1.1.1/lib* with the correct path to the jar file on your computer. (This example assumes a Linux or Mac OS computer; on Windows, paths look a little different.) When using the *java* command for running a program that uses Jogl, you need to add both *jogl.jar* and *gluegen-rt.jar* to the class path. Furthermore, Java must be able to find the native library files that are also in the *lib* directory of the Jogl download. For Linux, that directory should be added to the `LIBRARY_PATH` environment variable; for Mac OS, it should be added to `DYLD_LIBRARY_PATH`; and for Windows, it should be added to `PATH`. Here for example is a sequence of commands that might be used on Linux to run a class, *packagename.MainClass*, that uses Jogl:

```
JOGL_LIB_DIR="/home/eck/jogl-1.1.1/lib"
JOGL_JAR_FILES="$JOGL_LIB_DIR/jogl.jar:$JOGL_LIB_DIR/gluegen-rt.jar"
export LIBRARY_PATH="$JOGL_LIB_DIR"
java -cp $JOGL_JAR_FILES:. packagename.MainClass
```

All that typing would quickly get annoying, so you will want to use scripts to do the compiling and running. The source code directory for this book contains two simple scripts, *gljavac.sh* and *gljava.sh*, that can be used to compile and run Jogl programs on Mac OS or Linux. You will just need to edit the scripts to use the correct path to the Jogl *lib* directory. (Sorry, but I do not currently have scripts for Windows.)

2.2 Into the Third Dimension

As we move our drawing into the third dimension, we have to deal with the addition of a third coordinate, a z-axis in addition to an x-axis and a y-axis. There's quite a bit more to it than that, however, since our goal is not just to define shapes in 3D. The goal is to make 2D images of 3D scenes that look like pictures of three-dimensional objects. Take a look, for example, at this picture of a set of three coordinate axes:



It's not too hard to see this as a picture of three arrows pointing in different directions in space. If we just drew three straight lines to represent the axes, it would look like nothing more than three lines on a 2D surface. In the above picture, on the other hand, each axis is actually built from a long, thin cylinder and a cone. Furthermore, the shapes are shaded in a way that imitates the way that light would reflect from curved shapes. Without this simulated lighting, the shapes would just look like flat patches of color instead of curved 3D shapes. (This picture was drawn with a short OpenGL program. Take a look at the sample program *Axes3D.java* if you are interested.)

In this section and in the rest of the chapter, we will be looking at some of the fundamental ideas of working with OpenGL in 3D. Some things will be simplified and maybe even oversimplified as we try to get the big picture. Later chapters will fill in more details.

2.2.1 Coordinate Systems

Our first task is to understand 3D coordinate systems. In 3D, the x-axis and the y-axis lie in a plane called the xy-plane. The z-axis is perpendicular to the xy-plane. But you see that we already have a problem. The origin, (0,0,0), divides the z-axis into two parts. One of these parts is the positive direction of the z-axis, and we have to decide which one. In fact, either choice will work.

In OpenGL, the default coordinates system identifies the xy-plane with the computer screen, with the positive direction of the x-axis pointing to the right and the positive direction of the y-axis pointing upwards. The z-axis is perpendicular to the screen. The positive direction of the z-axis points **out** of the screen, towards the viewer, and the negative z-axis points into the screen. This is a ***right-handed coordinate system***: If you curl the fingers of your right hand in the direction from the positive x-axis to the positive y-axis, then your thumb points in the direction of the positive z-axis. (In a left-handed coordinate system, you would use your left hand in the same way to select the positive z-axis.)

This is only the default coordinate system. Just as in 2D, you can set up a different coordinate system for use in drawing. However, OpenGL will transform everything into the default coordinate system before drawing it. In fact, there are several different coordinate systems that you should be aware of. These coordinates systems are connected by a series of transforms from one coordinate system to another.

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 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. Once we have this world, we want to produce an image of it.

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 (and the positive direction of the y-axis is pointing straight up). This is a viewer-centric coordinate system, and it's important because it determines what exactly is seen in the image. 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 transform***.

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 ***modelview transform***. In fact, although the distinction between modeling transforms and viewing transforms is important conceptually, the distinction is for convenience only. The same transform can be thought of as a modeling transform, placing objects into the world, or a viewing transform, placing the viewer into the world. In fact, OpenGL doesn't even use world coordinates internally—it 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***, 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. Objects with larger or smaller z-values are also 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 way that OpenGL works internally.) 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 ***normalized device coordinates.*** The transformation from eye coordinates to normalized device 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.

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 we have called the viewport. The ***viewport transformation*** takes x and y from the normalized device 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 polygon, that is part of the world and that might appear in the image that we want to make of the world. The primitive goes through the following sequence of operations:

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 normalized device 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 individual pixels to draw the primitive on the device.

* * *

All this still leaves open the question of how you actually work with this complicated series of transformations. Remember that you just have to **set up** the viewport, modelview, and projection transforms. OpenGL will do all the calculations that are required to implement the transformations.

With Jogl, the viewport is automatically set to what is usually the correct value, that is, to use the entire available drawing area as the viewport. This is done just before calling *reshape* method of the *GLEventListener*. It is possible, however, to set up a different viewport by calling *gl.glViewport*(*x*,*y*,*width*,*height*), where (*x*,*y*) is the lower left corner of the rectangle that you want to use for drawing, *width* is the width of the rectangle, and *height* is the height. These values are given in device (pixel) coordinates. Note that in OpenGL device coordinates, the minimal y value is at the bottom, and y increases as you move **up**; this is the opposite of the convention in Java *Graphics2D*. You might use this, for example, to draw two or more views of the same scene in different parts of the drawing area, as follows: In your *display*i> method, you would use *glViewport* to set up a viewport in part of the drawing area, and draw the first view of the scene. You would then use *glViewport* again to set up a different viewport in another part of the drawing area, and draw the scene again, from a different point of view.

The modelview transform is a combination of modeling and viewing transforms. Modeling is done by applying the basic transform methods *glScalef*, *glRotatef*, and *glTranslatef* (or their **double** precision equivalents). These methods can also be used for viewing. However, it can be clumsy to get the exact view that you want using these methods. In the next chapter, we'll look at more convenient ways to set up the view. For now, we will just use the default view, in which the viewer is on the z-axis, looking in the direction of the negative z-axis.

Finally, to work with the projection transform, you need to know a little more about how OpenGL handles transforms. Internally, transforms are represented as matrices (two-dimensional arrays of numbers), and OpenGL uses the terms ***projection matrix*** and ***modelview matrix*** instead of projection transform and modelview transform. OpenGL keeps track of these two matrices separately, but it only lets you work on one or the other of these matrices at a time. You select the matrix that you want to work on by setting the value of an OpenGL state variable. To select the projection matrix, use

```
gl.glMatrixMode(GL.GL_PROJECTION);
```

To select the modelview matrix, use

```
gl.glMatrixMode(GL.GL_MODELVIEW);
```

All operations that affect the transform, such as *glLoadIdentity*, *glScalef*, and *glPushMatrix*, affect only the currently selected matrix. (OpenGL keeps separate stacks of matrices for use with *glPushMatrix* and *glPopMatrix*, one for use with the projection matrix and one for use with the modelview matrix.)

The projection matrix is used to establish the view volume, the part of the world that is rendered onto the display. The view volume is expressed in eye coordinates, that is, from the point of view of the viewer. (Remember that the projection transform is the transform from eye coordinates onto the standard cube that is used for normalized device coordinates.) OpenGL has two methods for setting the view volume, *glOrtho* and *glFrustum*. These two methods represent two different kinds of projection, ***orthographic projection*** and ***perspective projection***. Although *glFrustum* gives more realistic results, it's harder to understand, and we will put it aside until the next chapter. For now, we consider a simple version of *glOrtho*. We assume that we want to view objects that lie in a cube centered at the origin. We can use *glOrtho* to establish this cube as the view volume. If the cube stretches from $-s$ to s in the x, y, and z directions, then the projection can be set up by calling

```
gl.glOrtho(-s,s,-s,s,-s,s);
```

In general, though, we want the view volume to have the same aspect ratio as the viewport, so we need to expand the view volume in either the x or the y direction to match the aspect ratio of the viewport. This can be done most easily in the *reshape* method, where we know the aspect ratio of the viewport. Remember that we must call *glMartixMode* to switch to the projection matrix. Then, after setting up the projection, we call *glMatrixMode* again to switch back to the modelview matrix, so that all further transform operations will affect the modelview transform. Putting this all together, we get the following *reshape* method:

```
public void reshape(GLAutoDrawable drawable,
                                            int x, int y, int width, int height) {
    GL gl = drawable.getGL();
    double s = 1.5;  // limits of cube that we want to view go from -s to s.
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();  // Start with the identity transform.
    if (width > height) {  // Expand x limits to match viewport aspect ratio.
        double ratio = (double)width/height;
        gl.glOrtho(-s*ratio, s*ratio, -s, s, -s, s);
    }
    else {  // Expand y limits to match viewport aspect ratio.
        double ratio = (double)height/width;
        gl.glOrtho(-s, s, -s*ratio, s*ratio, -s, s);
    }
    gl.glMatrixMode(GL.GL_MODELVIEW);
}
```

This method is used in the sample program *Axes3D.java*. Still, it would be nice to have a better way to set up the view, and so I've written a helper class that you can use without really understanding how it works. The class is *Camera*, and you can find it in the source package *glutil* along with several other utility classes that I've written. A *Camera* object takes responsibility for setting up both the projection transform and the view transform. By default, it uses a perspective projection, and the region with x, y, and z limits from -5 to 5 is in view. If *camera* is a *Camera*, you can change the limits on the view volume by calling *camera.setScale*(s). This will change the x, y, and z limits to range from $-s$ to s. To use the camera, you should call *camera.apply*(gl) at the beginning of the *display* method to set up the projection and view. In this case, there is no need to define the *reshape* method. Cameras are used in the remaining sample programs in this chapter.

Note that even when drawing in 2D, you need to set up a projection transform. Otherwise, you can only use coordinates between -1 and 1. When setting up the projection matrix for 2D drawing, you can use *glOrtho* to specify the range of x and y coordinates that you want to use. The range of z coordinates is not important, as long as it includes 0. For example:

```
gl.glMatrixMode(GL.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrtho(xmin, xmax, ymin, ymax, -1, 1);
gl.glMatrixMode(GL.GL_MODELVIEW);
```

2.2.2 Essential Settings

Making a realistic picture in 3D requires a lot of computation. Certain features of this computation are turned off by default, since they are not always needed and certainly not for 2D drawing. When working in 3D, there are a few essential features that you will almost always want to enable. Often, you will do this in the *init* method.

Perhaps the most essential feature for 3D drawing is the ***depth test***. When one object lies behind another, only one of the objects can be seen, and which is seen depends on which one is closer to the viewer, and not on the order in which the objects are drawn. Now, OpenGL always draws objects in the order in which they are generated in the code. However, for each pixel that it draws in an object, it firsts tests whether there is already another object at that pixel that is closer to the viewer than the object that is being drawn. In that case, it leaves the pixel unchanged, since the object that is being drawn is hidden from sight by the object that is already there. This is the depth test. "Depth" here really means distance from the user, and it is essentially the z-coordinate of the object, expressed in eye coordinates. In order to implement the depth test, OpenGL uses a ***depth buffer***. This buffer stores one value for each pixel, which represents the eye-coordinate z-value of the object that is drawn at that pixel, if any. (There is a particular value that represents "no object here yet.") By default, the depth test is disabled. If you don't enable it, then things might appear in your picture that should really be hidden behind other objects. The depth test is enabled by calling

```
gl.glEnable(GL.GL_DEPTH_TEST);
```

This is usually done in the *init* method. You can disable the depth test by calling

```
gl.glDisable(GL.GL_DEPTH_TEST);
```

and you might even want to do so in some cases. (Note that any feature that can be enabled can also be disabled. In the future, I won't mention this explicitly.)

To use the depth test correctly, it's not enough to enable it. Before you draw anything, the depth buffer must be set up to record the fact that no objects have been drawn yet. This is called clearing the depth buffer. You can do this by calling *glClear* with a flag that indicates that it's the depth buffer that you want to clear:

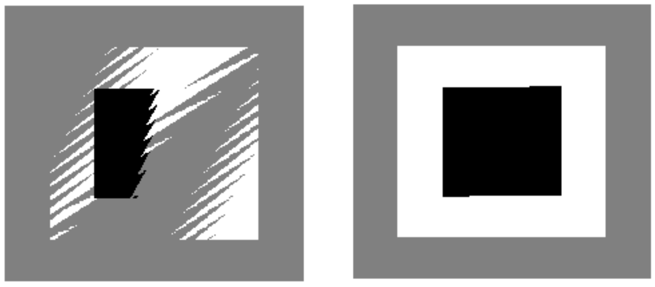
```
gl.glClear(GL.GL_DEPTH_BUFFER_BIT);
```

This should be done at the beginning of the *display* method, before drawing anything, at the same time that you clear the color buffer. In fact, the two opererations can be combined into one method call, by "or-ing" together the flags for the two buffers:

```
gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
```

This might be faster than clearing the two buffers separately, depending on how the clear operations are implemented by the graphics hardware.

There is one issue with the depth test that you should be aware of. What happens when two objects are actually at the same distance from the user? Suppose, for example, that you draw one square inside another, lying in the same plane. Does the second square appear or not? You might expect that the second square that is drawn might appear, as it would if the depth test were disabled. The truth is stranger. Because of the inevitable inexactness of real-number computations, the computed z-values for the two squares at a given pixel might be different— and which one is greater might be different for different pixels. Here is a real example in which a black square is drawn inside a white square which is inside a gray square. The whole picture is rotated a bit to force OpenGL to do some real-number computations. The picture on the left is drawn with the depth test enabled, while it is disabled for the picture on the right:



When the depth test is applied, there is no telling which square will end up on top at a given pixel. A possible solution would be to move the white square a little bit forward and the black square forward a little bit more—not so much that the change will be visible, but enough to clear up the ambiguity about which square is in front. This will enable the depth test to produce the correct result for each pixel, in spite of small computational errors.

* * *

The depth test ensures that the right object is visible at each pixel, but it's not enough to make a 3D scene look realistic. For that, you usually need to simulate ***lighting*** of the scene. Lighting is disabled by default. It can be enabled by calling

```
gl.glEnable(GL.GL_LIGHTING);
```

This is possibly the single most significant command in OpenGL. Turning on lighting changes the rendering algorithm in fundamental ways. For one thing, if you turn on lighting and do nothing else, you won't see much of anything! This is because, by default, no lights are turned on, so there is no light to illuminate the objects in the scene. You need to turn on at least one light:

```
gl.glEnable(GL.GL_LIGHT0);
```

This command turns on light number zero, which by default is a white light that shines on the scene from the direction of the viewer. It is sufficient to give decent, basic illumination for many scenes. It's possible to add other lights and to set properties of lights such as color and position. However, we will leave that for Chapter 4.

Turning on lighting has another major effect: If lighting is on, then the current drawing color, as set for example with *glColor3f*, is not used in the rendering process. Instead, the current ***material*** is used. Material is more complicated than color, and there are special commands for setting material properties. The default material is a rather ugly light gray. We will consider material properties in Chapter 4.

To get the best effect from lighting, you will also want to use the following command:

```
gl.glShadeModel(GL.GL_SMOOTH);
```

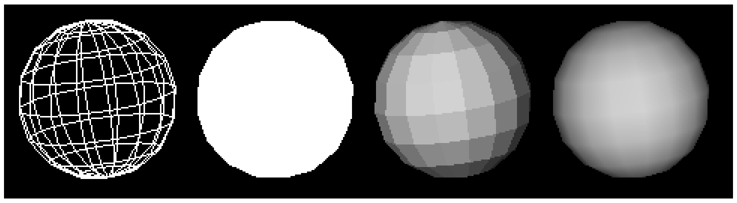
This has to do with the way that interiors of polygons are filled in. When drawing a polygon, OpenGL does many calculations, including lighting calculations, only at the vertices of the polygons. The results of these calculations are then interpolated to the pixels inside the polygon. This can be much faster than doing the full calculation for each pixel. OpenGL computes a color for each vertex, taking lighting into account if lighting is enabled. It then has to decide how to use the color information from the vertices to color the pixels in the polygon. (OpenGL does the same thing for lines, interpolating values calculated for the endpoints to the rest of the line.) The default, which is called ***flat shading***, is to simply copy the color from the first vertex of the polygon to every other pixel. This results in a polygon that is a uniform color, with no shading at all. A better solution, called ***smooth shading*** smoothly varies the colors from the vertices across the face of the polygon. Setting the shade model to *GL_SMOOTH* tells OpenGL to use smooth shading. You can return to flat shading by calling

```
gl.glShadeModel(GL.GL_FLAT);
```

Putting all the essential settings that we have talked about here, we get the following *init* method for 3D drawing with basic lighting:

```
public void init(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();
    gl.glClearColor(0,0,0,1);          // Set background color.
    gl.glEnable(GL.GL_LIGHTING);       // Turn on lighting.
    gl.glEnable(GL.GL_LIGHT0);         // Turn on light number 0.
    gl.glEnable(GL.GL_DEPTH_TEST);     // Turn on the depth test.
    gl.glShadeModel(GL.GL_SMOOTH);     // Use smooth shading.
}
```

Let's look at a picture that shows some of the effects of these commands. OpenGL has no sphere-drawing command, but we can approximate a sphere with polygons. In this case, a rather small number of polygons is used, giving only a rough approximation of a sphere. Four spheres are shown, rendered with different settings:



In the sphere on the left, only the outlines of the polygons are drawn. This is called a ***wireframe model***. Lighting doesn't work well for lines, so the wireframe model is drawn with lighting turned off. The color of the lines is the current drawing color, which has been set to white using *glColor3f*.

The second sphere from the left is drawn using filled polygons, but with lighting turned off. Since no lighting or shading calculations are done, the polygons are simply filled with the current drawing color, white. Nothing here looks like a 3D sphere; we just see a flat patch of color.

The third and fourth spheres are drawn with lighting turned on, using the default light gray material color. The difference between the two spheres is that flat shading is used for the third sphere, while smooth shading is used for the fourth. You can see that drawing the sphere using lighting and smooth shading gives the most realistic appearance. The realism could be increased even more by using a larger number of polygons to draw the sphere.

* * *

There are several other common, but not quite so essential, settings that you might use for 3D drawing. As noted above, when lighting is turned on, the color of an object is determined by its material properties. However, if you want to avoid the complications of materials and still be able to use different colors, you can turn a feature that causes OpenGL to use the current drawing color for the material:

```
gl.glEnbable(GL.GL_MATERIAL_COLOR);
```

This causes the basic material color to be taken from the color set by *glColor3f* or similar commands. (Materials have other aspects besides this basic color, but setting the basic color is often sufficient.) Examples in this chapter and the next will use this feature.

Another useful feature is ***two-sided lighting***. OpenGL distinguishes the two sides of a polygon. One side is the front side, and one is the back side. Which is the front side and which is the back is determined by the order in which the vertices of the polygon are specified when it is drawn. (By default, the order is counterclockwise if you are looking at the front face and is clockwise if you are looking at the back face.) In the default lighting model, the back faces are not properly lit. This is because in many 3D scenes, the back faces of polygons face the insides of objects and will not be visible in the scene; the calculation that is required to light them would be wasted. However, if the back sides of some polygons might be visible in your scene, then you can turn on two-sided lighting, at least when you are drawing those polygons. This forces the usual lighting calculations to be done for the back sides of polygons. To turn on two-sided lighting, use the command:

```
gl.glLightModeli(GL.GL_LIGHT_MODEL_TWO_SIDE, GL.GL_TRUE);
```

You can turn it off using the same command with parameter *GL.GL_FALSE* in place of *GL.GL_TRUE*.

Finally, I will mention the *GL_NORMALIZE* option, which can be enabled with

```
gl.glEnable(GL.GL_NORMALIZE);
```

This has to do with the "normal vectors" that will be discussed in Section 2.4. (For now, just think of a vector as an arrow that has a length and a direction.) This option should be enabled in two circumstances: if you supply normal vectors that do not have length equal to 1, or if you apply scaling transforms. (Rotation and translation are OK without it.) Correct lighting calculations require normal vectors of length 1, but nothing forces you to supply vectors of proper length. Furthermore, scaling transforms are applied to normal vectors as well as to geometry, and they can increase or decrease the lengths of the vectors. *GL_NORMALIZE* forces OpenGL to adjust the length of all normal vectors to length 1 before using them in lighting calculations. This adds some significant computational overhead to the rendering process, so this feature is turned off by default. However, if you fail to turn it on when it is needed, the lighting calculations for your image will be incorrect.

2.3 Drawing in 3D

Finally, we are ready to create some three-dimensional scenes. We will look first at how to create three-dimensional objects and how to place them into the world. This is geometric modeling in 3D, and is not so different from modeling in 2D. You just have to deal with the extra coordinate.

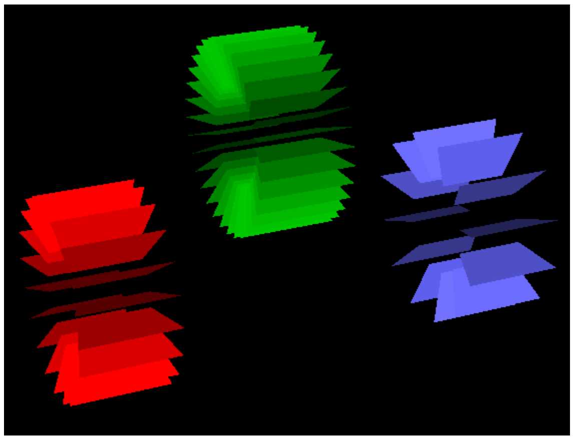
2.3.1 Geometric Modeling

We have already seen how to use *glBegin/glEnd* to draw primitives such as lines and polygons in two dimensions. These commands can be used in the same way to draw primitives in three dimensions. To specify a vertex in three dimensions, you can call *gl.glVertex3f(x,y,z)* (or *gl.glVertex3d(x,y,z)*). You can also continue to use *glVertex2f* and *glVertex2d* to specify points with z-coordinate equal to zero. You can use *glPushMatrix* and *glPopMatrix* to implement hierarchical modeling, just as in 2D.

Modeling transformations are an essential part of geometric modeling. Scaling and translation are pretty much the same in 3D as in 2D. Rotation is more complicated. Recall that the OpenGL command for rotation is *gl.glRotatef(d,x,y,z)* (or *glRotated*), where d is the measure of the angle of rotation in degrees and x, y, and z specify the axis of rotation. The axis passes through (0,0,0) and the point (x,y,z). However, this does not fully determine the rotation, since it does not specify the direction of rotation. Is it clockwise? Counterclockwise? What would that even mean in 3D?

The direction of rotation in 3D follows the ***right-hand rule*** (assuming that you are using a right-handed coordinate system). To determine the direction of rotation for *gl.glRotatef(d,x,y,z)*, place your right hand at (0,0,0), with your thumb pointing at (x,y,z). The direction in which your fingers curl will be the direction of a positive angle of rotation; a negative angle of rotation will produce a rotation in the opposite direction. (If you happen to use a left-handed coordinate system, you would use your left hand in the same way to determine the positive direction of rotation.) When you want to apply a rotation in 3D, you will probably find yourself literally using your right hand to figure out the direction of rotation.

Let's do a simple 3D modeling example. Think of a paddle wheel, with a bunch of flat flaps that rotate around a central axis. For this example, we will just draw the flaps, with nothing physical to support them. Three paddle wheels, with different numbers of flaps, are drawn in the following picture:



To produce a model of a paddle wheel, we can start with one flap, drawn as a trapezoid lying in the xy-plane above the origin:

```
gl.glBegin(GL.GL_POLYGON);
gl.glVertex2d(-0.7,1);
gl.glVertex2d(0.7,1);
gl.glVertex2d(1,2);
gl.glVertex2d(-1,2);
gl.glEnd();
```

We can make the complete wheel by making copies of this flap, rotated by various amounts around the x-axis. If *paddles* is an integer giving the number of flaps that we want, then the angle between each pair of flaps will be 360/*paddles*. Recalling that successive transforms are multiplied together, we can draw the whole set of paddles with a for loop:

```
for (int i = 0; i < paddles; i++) {
    gl.glRotated(360.0/paddles, 1, 0, 0);
    gl.glBegin(GL.GL_POLYGON);
    gl.glVertex2d(-0.7,1);
    gl.glVertex2d(0.7,1);
    gl.glVertex2d(1,2);
    gl.glVertex2d(-1,2);
    gl.glEnd();
}
```

The first paddle is rotated by 360/*paddles* degrees. By the time the second paddle is drawn, two such rotations have been applied, so the second paddle is rotated by 2*(360/(*paddles*). As we continue, the paddles are spread out to cover a full circle. If we want to make an animation in which the paddle wheel rotates, we can apply an additional rotation, depending on the frameNumber, to the wheel as a whole.

To draw the set of three paddle wheels, we can write a method to draw a single wheel and then call that method three times, with different colors and translations in effect for each call. You can see how this is done in the source code for the program that produced the above image, *PaddleWheels.java*. By the way, in this sample program, you can use mouse to rotate the view. The mouse is enabled by an object of type *TrackBall*, another of the classes defined in the *glutil* source directory. You can find an applet version of the program in the on-line version of this section.

2.3.2 Some Complex Shapes

Complex shapes can be built up out of large numbers of polygons, but it's not always easy to see how to do so. OpenGL has no built-in complex shapes, but there are utility libraries that provide subroutines for drawing certain shapes. One of the most basic libraries is ***GLUT***, the OpenGL Utilities Toolkit. Jogl includes a partial implementation of GLUT in the class *com.sun.opengl.util.GLUT*. GLUT includes methods for drawing spheres, cylinders, cones, and regular polyhedra such as cubes and dodecahedra. The curved shapes are actually just approximations made out of polygons. To use the drawing subroutines, you should create an object, *glut*, of type *GLUT*. You will only need one such object for your entire program, and you can create it in the *init* method. Then, you can draw various shapes by calling methods such as

```
glut.glutSolidSphere(0.5, 40, 20);
```

This draws a polyhedral approximation of a sphere. The sphere is centered at the origin, with its axis lying along the z-axis. The first parameter gives the radius of the sphere. The second gives the number of "slices" (like lines of longitude or the slices of an orange), and the third gives the number of "stacks" (divisions perpendicular to the axis of the sphere, like lines of latitude). Forty slices and twenty stacks give quite a good approximation for a sphere. You can check the *GLUT* API documentation for more information.

Because of some limitations to the GLUT drawing routines (notably the fact that they don't support textures), I have written a few shape classes of my own. Three shapes—sphere, cylinder, and cone—are defined by the classes *UVSphere*, *UVCylinder*, and *UVCone* in the sample source package *glutil*. To draw a sphere, for example, you should create an object of type *UVSphere*. For example,

```
    sphere = new UVSphere(0.5, 40, 20);
```

Similarly to the GLUT version, this represents a sphere with radius 0.5, 40 slices, and 20 stacks, centered at the origin and with its axis along the z-axis. Once you have the object, you can draw the sphere in an OpenGL context *gl* by saying

```
    sphere.render(gl);
```

Cones and cylinders are used in a similar way, except that they have a height in addition to a radius:

```
    cylinder = new UVCylinder(0.5,1.5,15,10); // radius, height, slices, stacks
```

The cylinder is drawn with its base in the xy-plane, centered at the origin, and with its axis along the positive z-axis. By default, both the top and bottom ends of the cylinder are drawn, but you can turn this behavior off by calling *cylinder.setDrawTop(false)* and/or *cylinder.setDrawBottom(false)*. Using a *UVCone* is very similar, except that a cone doesn't have a top, only a bottom.

I will use these three shape classes in several examples throughout the rest of this chapter.

2.3.3 Optimization and Display Lists

The drawing commands that we have been using work fine, but they have a problem with efficiency. To understand why, you need to understand something of how OpenGL works. The calculations involved in computing and displaying an image of a three-dimensional scene can be immense. Modern desktop computers have ***graphics cards***—dedicated, specialized hardware that can do these calculations at very high speed. In fact, the graphics cards that come even in inexpensive computers today would have qualified as supercomputers a decade or so ago.

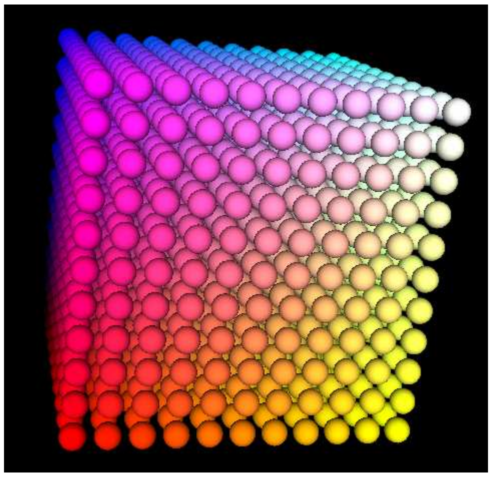
It is possible, by the way, to implement OpenGL on a computer without any graphics card at all. The computations and drawing can be done, if necessary, by the computer's main CPU. When this is done, however, graphics operations are much slower—slow enough to make many 3D programs unusable. Using a graphics card to perform 3D graphics operations at high speed is referred to as ***3D hardware acceleration***.

OpenGL is built into almost all modern graphics cards, and most OpenGL commands are executed on the card's specialized hardware. Although the commands can be executed very quickly once they get to the graphics card, there is a problem: The commands, along with all the data that they require, have to transmitted from your computer's CPU to the graphics card. This can be a significant bottleneck, and the time that it takes to transmit all that information can be a real drag on graphics performance. Newer versions of OpenGL have introduced various techniques that can help to overcome the bottleneck. In this section, we will look at one of these optimization strategies, ***display lists***. Display lists have been part of OpenGL from the beginning.

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 that can be stored on a graphics card (though there is no guarantee that they actually will be). The contents of the display list only have to be transmitted once from the CPU to the card. Once a list has been created, it can be "called," very much like a subroutine. 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 the full power of hardware acceleration can be used to execute the commands at the highest possible speed.

One difference between display lists and subroutines is that display lists don't have parameters. This limits their usefulness, since its not possible to customize their behavior by calling them with different parameter values. However, 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.

Here, for example is an image that shows 1331 spheres, arranged in a cube that has eleven spheres along each edge.



Each sphere in the picture is a different color and is in a different location. Exactly the same OpenGL commands are used to draw each sphere, but before drawing each sphere, the transform and color are changed. This is a natural place to use a display list. We can store the commands for drawing one sphere in a display list, and call that list once for each sphere that we want to draw. The graphics commands that draw the sphere only have to be sent to the card once. Then each of the 1331 spheres can be drawn by sending a single command to the card to call the list (plus a few commands to change the color and transform). Note that we are also saving some significant time on the main CPU, since it only has to generate the sphere-drawing commands once instead of 1331 times.

In the program that drew the image, you can rotate the cube of spheres by dragging the mouse over the image. You can turn the use of display lists on and off using a checkbox below the image. You are likely to see a significant change in the time that it takes to render the image, depending on whether or not display lists are used. However, the exact results will depend very much on the OpenGL implementation that you are using. On my computer, with a good graphics card and hardware acceleration, the rendering with display lists is fast enough to produce very smooth rotation. Without display lists, the rendering is about five times slower, and the rotation is a bit jerky but still usable. Without hardware acceleration, the rendering might take so long that you get no feeling at all of natural rotation. You can try the applet in the on-line version of these notes.

* * *

I hope that this has convinced you that display lists can be very worthwhile. Fortunately, they are also fairly easy to use.

Remember that display lists are meant to be stored on the graphics card, so it is the graphics card that has to manage the collection of display lists that have been created. Display lists are identified by integer code numbers. If you want to use a display list, you first have to ask the graphics card for an integer to identify the list. (You can't just make one up, since the one that you pick might already be in use.) This is done with a command such as

```
listID = gl.glGenLists(1);
```

The return value is an **int** which will be the identifier for the list. The parameter to *glGenLists* is also an **int**. 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 *gl.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:

```
gl.glNewList(listID, GL.GL_COMPILE);
   ...  // Generate OpenGL commands to store in the list.
gl.glEndList();
```

Note that the previous contents of the list, if any, will be deleted and relaced. The parameter *GL.GL_COMPILE* means that you only want to store commands into the list, not execute them. If you use the alternative parameter *GL.GL_COMPILE_AND_EXECUTE*, then the commands will be executed immediately as well as stored in the list for later reuse.

Most, but not all, OpenGL commands can be placed into a display list. In particular, commands for generating geometry, applying transforms, changing the color, and enabling and disabling various features can be placed into lists. You can even add commands for calling other lists. In addition to OpenGL commands, you can have other Java code between *glNewList* and *glEndList*. For example, you can use a *for* loop to generate a large number of vertices. But remember that the list only stores OpenGL commands, not the Java code that generates them For example, if you use a *for* loop to call *glVertex3f* 10,000 times, then the display list will contain 10,000 individual *glVertex3f* commands, not a *for* loop. Furthermore, as the parameters to these commands, the list only stores numerical values, not references to variables. So if you call

```
gl.glVertex3d(x,y,z);
```

between *glNewList* and *glEndList*, only the values of x, y, and z at the time *glVertex3d* is called are stored into the list. If you change the values of the variables later, the change has no effect on the list or what it does.

Once you have a list, you can call the list with the command

```
gl.glCallList(listID);
```

The effect of this command, remember, is to tell the graphics card to execute a list that it has already stored.

If you are going to reuse a list many times, it can make sense to create it in the *init* method and keep it around as long as your program will run. You can also, of course, make them on the fly, as needed, in the *display* method. However, a graphics card has only a limited amount of memory for storing display lists, so you shouldn't keep a list around longer than you need it. 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 one list at a time is probably the most likely use. Deleting a list when you are through with it allows the graphics card to reuse the memory used by that list.

* * *

As an example of using display lists, you can look at the source code for the program that draws the "cube of spheres," *ColorCubeOfSpheres.java*. That program is complicated somewhat by the option to use or not use the display list. Let's look at some slightly modified code that always uses lists.

The program uses an object of type *UVSphere* to draw a sphere. The display list is created in the *init* method, and the *UVSphere* object is used to generate the commands that go into the list. The integer identifier for the list is *displayList*:

```
UVSphere sphere = new UVSphere(0.4);  // For drawing a sphere of radius 0.4.
displayList = gl.glGenLists(1);  // Allocate the list ID.
gl.glNewList(displayList, GL.GL_COMPILE);
sphere.render(gl);
gl.glEndList();
```

When *sphere.render*(*gl*) is called, all the OpenGL commands that it generates are channeled into the list, instead of being executed immediately. In the *display* method, the display list is called 1331 times in a set of triply nested *for* loops:

```
for (int i = 0; i <= 10; i++) {
     float r = i/10.0f;  // Red component of sphere color.
     for (int j = 0; j <= 10; j++) {
          float g = j/10.0f;  // Green component of sphere color.
          for (int k = 0; k <= 10; k++) {
               float b = k/10.0f;  // Blue component of sphere color.
               gl.glColor3f(r,g,b);  // Set the color for the sphere.
               gl.glPushMatrix();
               gl.glTranslatef(i-5,j-5,k-5);  // Translate the sphere.
               gl.glCallList(displayList);  // Call display list to draw sphere.
               gl.glPopMatrix();
          }
     }
}
```

The *glCallList* in this code replaces a call to *sphere.render* that would otherwise be used to draw the sphere directly.

2.4 Normals and Textures

OpenGL associates several quantities with every vertex that is generated. These quantities are called ***attributes*** of the vertex. One attribute is color. It is possible to assign a different color to every vertex in a polygon. The color that is assigned to a vertex is the current drawing color at the time the vertex is generated, which can be set, for example, with *glColor3f*. (This color is used only if lighting is off or if the *GL_COLOR_MATERIAL* option is on.) The drawing color can be changed between calls to *glBegin* and *glEnd*. For example:

```
gl.glBegin(GL.GL_POLYGON);
gl.glColor3f(1,0,0);
gl.glVertex(-0.5,-0.5);  // The color associated with this vertex is red.
gl.glColor3f(0,1,0);
gl.glVertex(0.5,-0.5);   // The color associated with this vertex is green.
gl.glColor3f(0,0,1);
gl.glVertex(0,1);        // The color associated with this vertex is blue.
gl.glEnd();
```

Assuming that the shade model has been set to *GL_SMOOTH*, each vertex of this triangle will be a different color, and the colors will be smoothly interpolated to the interior of the triangle.

In this section, we will look at two other important attributes that can be associated with a vertex. One of them, the normal vector, is an essential of lighting calculations. Another, the texture coordinates, is used when applying texture images to surfaces.

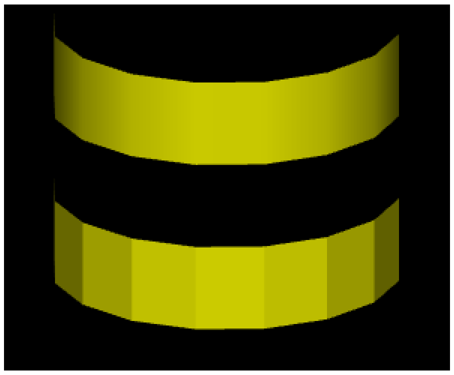
2.4.1 Introduction to Normal Vectors

The visual effect of a light shining on a surface depends on the properties of the surface and of the light. But it also depends to a great extent on the angle at which the light strikes the surface. That's why a curved, lit surface looks different at different points, even if its surface is a uniform color. To calculate this angle, OpenGL needs to know the direction in which the surface is facing. That direction can be specified by a ***vector*** that is perpendicular to the surface. A vector can be visualized as an arrow. It has a direction and a length. It doesn't have a particular location, so you can visualize the arrow as being positioned anywhere you like, such as sticking out of a particular point on a surface. A vector in three dimensions is given by three numbers that specify the change in x, the change in y, and the change in z along the vector. If you position the vector (x,y,z) with its start point at the origin, (0,0,0), then the end point of the vector is at the point with coordinates (x,y,z).

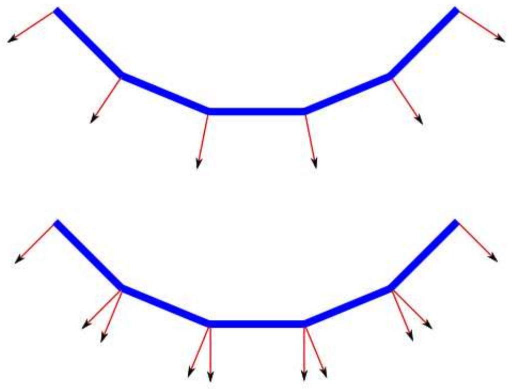
Another word for "perpendicular" is "normal," and a vector that is perpendicular to a surface at a given point is called a ***normal*** to that surface. When used in lighting calculations, a normal vector must have length equal to one. A normal vector of length one is called a ***unit normal***. For proper lighting calculations in OpenGL, a unit normal must be specified for each vertex. (Actually, if you turn on the option *GL_NORMALIZE*, then you can specify normal vectors of any length, and OpenGL will convert them to unit normals for you; see Subsection 2.2.2.)

Just as OpenGL keeps track of a current drawing color, it keeps track of a current normal vector, which is part of the OpenGL state. When a vertex is generated, the value of the current normal vector is copied and is associated to that vertex as an attribute. The current normal vector can be set by calling *gl.glNormal3f*(x,y,z) or *gl.glNormal3d*(x,y,z). This can be done at any time, including between calls to *glBegin* and *glEnd*. This means that it's possible for different vertices of a polygon to have different associated normal vectors.

Now, you might be asking yourself, "Don't all the normal vectors to a polygon point in the same direction?" After all, a polygon is flat; the perpendicular direction to the polygon doesn't change from point to point. This is true, and if your objective is to display a polyhedral object whose sides are flat polygons, then in fact, all the normals of one of those polygons should point in the same direction. On the other hand, polyhedra are often used to approximate curved surfaces such as spheres. If your real objective is to make something that looks like a curved surface, then you want to use normal vectors that are perpendicular to the actual surface, not to the polyhedron that approximates it. Take a look at this example:



The two objects in this picture are made up of bands of rectangles. The two objects have exactly the same geometry, yet they look quite different. This is because different normal vectors are used in each case. For the top object, I was using the band of rectangles to approximate a smooth surface (part of a cylinder, in fact). The vertices of the rectangles are points on that surface, and I really didn't want to see the rectangles at all—I wanted to see the curved surface, or at least a good approximation. So for the top object, when I specified a normal vector at one of the vertices, I used a vector that is perpendicular to the surface rather than one perpendicular to the rectangle. For the object on the bottom, on the other hand, I was thinking of an object that really **is** a band of rectangles, and I used normal vectors that were actually perpendicular to the rectangles. Here's a two-dimensional illustration that shows the normal vectors:



The thick blue lines represent the rectangles. Imagine that you are looking at them edge-on. The arrows represent the normal vectors. Two normal vectors are shown for each rectangle, one on each end.

In the bottom half of this illustration, the vectors are actually perpendicular to the rectangles. There is an abrupt change in direction as you move from one rectangle to the next, so where one rectangle meets the next, the normal vectors to the two rectangles are different. The visual effect on the rendered image is an abrupt change in shading that is perceived as a corner or edge between the two rectangles.

In the top half, on the other hand, the vectors are perpendicular to a curved surface that passes through the endpoints of the rectangles. When two rectangles share a vertex, they also share the same normal at that vertex. Visually, this eliminates the abrupt change in shading, resulting in something that looks more like a smoothly curving surface.

The upshot of this is that in OpenGL, a normal vector at a vertex is whatever you say it is, and it does not have to be literally perpendicular to your polygons. The normal vector that you choose should depend on the object that you are trying to model.

There is one other issue in choosing normal vectors: There are always two possible unit normal vectors at a vertex, pointing in opposite directions. Recall that a polygon has two faces, a front face and a back face, which are distinguished by the order in which the vertices are generated. (See Section 2.2.) A normal vector should point out of the front face of the polygon. That is, when you are looking at the front face of a polygon, the normal vector should be pointing towards you. If you are looking at the back face, the normal vector should be pointing away from you.

Unfortunately, it's not always easy to compute normal vectors, and it can involve some non-trivial math. I'll have more to say about that in Chapter 4. For now, you should at least understand that the solid surfaces that are defined in the GLUT library come with the correct normal vectors already built-in. So do the shape classes in my *glutil* package.

We can look at one simple case of supplying normal vectors by hand: drawing a cube. Let's say that we want to draw a cube centered at the origin, where the length of each side of the cube is one. Let's start with a method to draw the face of the cube that is perpendicular to the z-axis with center at (0,0,0.5). The unit normal to that face is (0,0,1), which points directly out of the screen along the z-axis.:

```
private static void drawFace1(GL gl) {
    gl.glBegin(GL.GL_POLYGON);

    gl.glNormal3d(0,0,1);  // Unit normal vector, which applies to all vertices.

    gl.glVertex3d(-0.5, -0.5, 0.5);  // The vertices, in counterclockwise order.
    gl.glVertex3d(0.5, -0.5, 0.5);
    gl.glVertex3d(0.5, 0.5, 0.5);
    gl.glVertex3d(-0.5, 0.5, 0.5);

    gl.glEnd();
}
```

We could draw the other faces similarly. For example, the bottom face, which has normal vector (0,$-$1,0), could be created with

```
gl.glBegin(GL.GL_POLYGON);
gl.glNormal3d(0, -1, 0);
gl.glVertex3d(-0.5, -0.5, 0.5);
gl.glVertex3d(0.5, -0.5, 0.5);
gl.glVertex3d(0.5, -0.5, -0.5);
gl.glVertex3d(-0.5, -0.5, -0.5);
gl.glEnd();
```

However, getting all the vertices correct and in the right order is by no means trivial. Another approach is to use the method for drawing the front face for each of the six faces, with appropriate rotations to move the front face into the desired position:

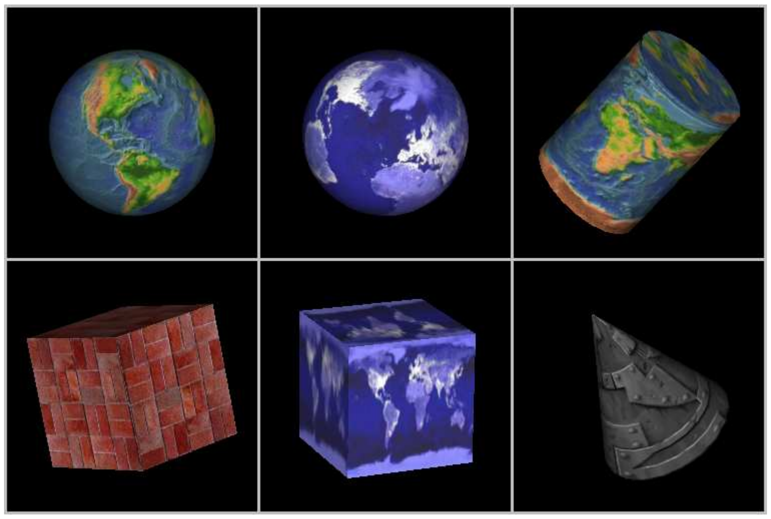
```
public static void drawUnitCube(GL gl) {

   drawFace1(gl);  // the front face

   gl.glPushMatrix();  // the bottom face
   gl.glRotated( 90, 1, 0, 0);  // rotate 90 degrees about the x-axis
   drawFace1(gl);
```

```
      gl.glPopMatrix();

      gl.glPushMatrix();  // the back face
      gl.glRotated( 180, 1, 0, 0);  // rotate 180 degrees about the x-axis
      drawFace1(gl);
      gl.glPopMatrix();

      gl.glPushMatrix();  // the top face
      gl.glRotated( -90, 1, 0, 0);  // rotate -90 degrees about the x-axis
      drawFace1(gl);
      gl.glPopMatrix();

      gl.glPushMatrix();  // the right face
      gl.glRotated( 90, 0, 1, 0);  // rotate 90 degrees about the y-axis
      drawFace1(gl);
      gl.glPopMatrix();

      gl.glPushMatrix();  // the left face
      gl.glRotated( -90, 0, 1, 0);  // rotate -90 degrees about the y-axis
      drawFace1(gl);
      gl.glPopMatrix();

   }
```

Whether this looks easier to you might depend on how comfortable you are with transformations, but it really does require a lot less work!

2.4.2 Introduction to Textures

The 3D objects that we have created so far look nice enough, but they are a little bland. Their uniform colors don't have the visual appeal of, say, a brick wall or a plaid couch. Three-dimensional objects can be made to look more interesting and more realistic by adding a ***texture*** to their surfaces. A texture—or at least the kind of texture that we consider here—is a 2D image that can be applied to the surface of a 3D object. Here is a picture that shows six objects with various textures:



(Topographical Earth image, courtesy NASA/JPL-Caltech, http://maps.jpl.nasa.gov/. Brick and metal textures from http://www.grsites.com/archive/textures/. EarthAtNight image taken from the Astronomy Picture of the Day web site, http://apod.nasa.gov/apod/ap001127.html; it is also a NASA/JPL image. Some nicer planetary images for use on a sphere can be found at http://evildrganymede.net/art/maps.htm and http://planetpixelemporium.com/earth.html; they are free for you to use, but I can't distribute them on my web site. The textures used in this program are in the folder named *textures* in the source directory.)

This picture comes from the sample program *TextureDemo.java*. You can try the applet version on-line. In the applet version, you can rotate each individual object by dragging the mouse on it. The rotation makes it look even more realistic.

Textures are one of the more complex areas in the OpenGL API, both because of the number of options involved and the new features that have been introduced in various versions. The Jogl API has a few classes that make textures easier to use. For now, we will work mostly with the Jogl classes, and we will cover only a few of the many options. You should understand that the Jogl texture classes are not themselves part of the OpenGL API.

To use a texture in OpenGL, you first of all need an image. In Jogl, the *Texture* class represents 2D images that can be used as textures. (This class and other texture-related classes are defined in the package *com.sun.opengl.util.texture*.) To create a *Texture* object, you can use the helper class *TextureIO*, which contains several static methods for reading images from various sources. For example, if *img* is a *BufferedImage* object, you can create a *Texture* from that image with

```
Texture texture = TextureIO.newTexture( img, true );
```

The second parameter is a **boolean** value that you don't need to understand just now. (It has to do with optimizing the texture to work on objects of different sizes.)

This means that you can create a *BufferedImage* any way you like, even by creating the image from scratch using Java's drawing commands, and then use that image as a texture in OpenGL. More likely, you will want to read the image from a file or from a program resource. *TextureIO* has methods that can do that automatically. For example, if *file* is of type *java.io.File* and represents a file that contains an image, you can create a texture from that image with

```
Texture texture = Texture.newTexture( file, true );
```

The case of reading an image from a resource is probably more common. A resource is, more or less, a file that is part of your program. It might be packaged into the same jar file as the rest of your program, for example. I can't give you a full tutorial on using resources here, but the idea is to place the image files that you want to include as resources in some package (that is, folder) in your project. Let's say that you store the images in a package named *textures*, and let's say that "brick001.jpg" is the name of one of the image files in that package. Then, to retrieve and use the image as a texture, you can use the following code in any instance method in your program:

```
Texture texture;  // Most likely, an instance variable.
try {
    URL textureURL;  // URL class from package java.net.
    textureURL = getClass().getClassLoader().getResource("texture/brick001.jpg");
    texture = TextureIO.newTexture(textureURL, true, "jpg");
                // The third param above is the extension from the file name;
                // "jpg", "png", and probably "gif" files should be OK.
}
catch (Exception e) {
```

```
            // Won’t get here if the file is properly packaged with the program!
        e.printStackTrace();
    }
```

All this makes it reasonably easy to prepare images for use as textures. But there are a couple of issues. First of all, in the original versions of OpenGL, the width and height of texture images had to be powers of two, such as 128, 256, 512, or 1024. Although this limitation has been eliminated in more recent versions, it’s probably still a good idea to resize your texture images, if necessary, so that their dimensions are powers of two. Another issue is that Java and OpenGL have different ideas about where the origin of an image should be. Java puts it at the top right corner, while OpenGL puts it at the bottom left. This means that images loaded by Java might be upside down when used as textures in OpenGL. The *Texture* class has a way of dealing with this, but it complicates things. For now, it’s easy enough to flip an image vertically if necessary. Jogl even provides a way of doing this for a *BufferedImage*. Here’s some alternative code that you can use to load an image resource as a texture, if you want to flip the image:

```
Texture texture;
try {
    URL textureURL;
    textureURL = getClass().getClassLoader().getResource(textureFileName);
    BufferedImage img = ImageIO.read(textureURL); // read file into BufferedImage
    ImageUtil.flipImageVertically(img);
    texture = TextureIO.newTexture(img, true);
}
catch (Exception e) {
    e.printStackTrace();
}
```

* * *

We move on now to the question of how to actually use textures in a program. First, note that textures work best if the material of the textured object is pure white, since the colors from the texture are actually multiplied by the material color, and if the color is not white then the texture will be “tinted” with the material color. To get white-colored objects, you can use use *glColor3f* and the *GL_MATERIAL_COLOR* option, as discussed in Subsection 2.2.2.

Once you have a *Texture* object, three things are necessary to apply that texture to an object: The object must have ***texture coordinates*** that determine how the image will be mapped onto the object. You must enable texturing, or else textures are simply ignored. And you have to specify which texture is to be used.

Two of these are easy. To enable texturing with a 2D image, you just have to call the method

```
texture.enable();
```

where *texture* is any object of type *Texture*. It’s important to understand that this is not selecting that particular texture for use; it’s just turning on texturing in general. (This is true at least as long as you stick to texture images whose dimensions are powers of two. In that case, it doesn’t even matter which *Texture* object you use to call the *enable* method.) To turn texturing off, call *texture.disable*(). If you are planning to texture all the objects in your scene, you could enable texturing method once, in the *init* method, and leave it on. Otherwise, you can enable and disable texturing as needed.

To get texturing to work, you must also specify that some particular texture is to be used. This is called ***binding*** the texture, and you can do it by calling

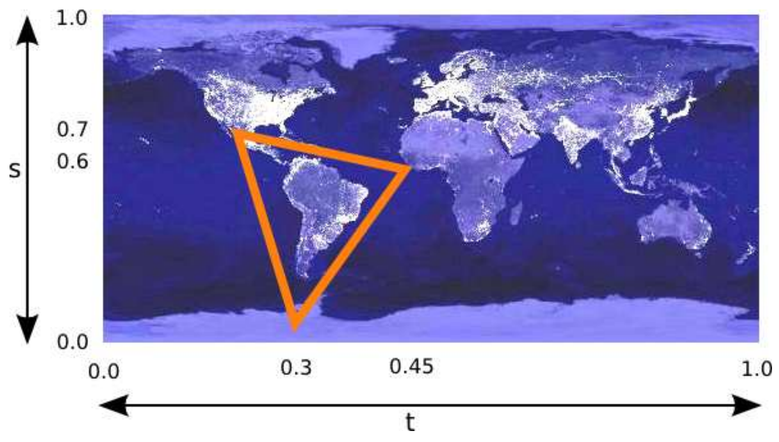
```
texture.bind()
```

where *texture* is the *Texture* object that you want to use. That texture will be used until you bind a different texture, but of course only when texturing is enabled. If you're just using one texture, you could bind it in the *init* method. It will be used whenever texturing is enabled. If you are using several textures, you can bind different textures in different parts of the *display* method.

Just remember: Call *texture.enable*() and *texture.bind*() before drawing the object that you want to texture. Call *texture.disable*() to turn texturing off. Call *texture2.bind*(), if you want to switch to using a different texture, *texture2*.

That leaves us with the problem of texture coordinates for objects to which you want to apply textures. A texture image comes with its own 2D coordinate system. Traditionally, *s* used for the horizontal coordinate on the image and *t* is used for the vertical coordinate. *s* is a real-number coordinate that ranges from 0 on the left of the image to 1 on the right, while *t* ranges from 0 at the bottom to 1 at the top. Values of *s* or *t* outside of the range 0 to 1 are not inside the image.

To apply a texture to a polygon, you have to say which point in the texture should correspond to each vertex of the polygon. For example, suppose that we want to apply part of the *EarthAtNight* image to a triangle. Here's the area in the image that I want to map onto the triangle, shown outlined in thick orange lines:



The vertices of this area have (*s*,*t*) coordinates (0.3,0.05), (0.45,0.6), and (0.25,0.7). These coordinates in the image, expressed in terms of *s* and *t* are called texture coordinates. When I generate the vertices of the triangle that I want to draw, I have to specify the corresponding texture coordinate in the image. This is done by calling *gl.glTexCoord2d*(*s*,*t*). (Or you can use *glTexCoord2f*.) You can call this method just before generating the vertex. Usually, every vertex of a polygon will have different texture coordinates. To draw the triangle in this case, I could say:

```
gl.glBegin(GL.GL_POLYGON);
gl.glNormal3d(0,0,1);
gl.glTexCoord2d(0.3,0.05);  // Texture coords for vertex (0,0)
gl.glVertex2d(0,0);
gl.glTexCoord2d(0.45,0.6);  // Texture coords for vertex (0,1)
```

```
    gl.glVertex2d(0,1);
    gl.glTexCoord2d(0.25,0.7);   // Texture coords for vertex (1,0)
    gl.glVertex2d(1,0);
    gl.glEnd();
```

Note that there is no particular relationship between the (x,y) coordinates of a vertex, which give its position in space, and the (s,t) texture coordinate associated with the vertex. which tell what point in the image is mapped to the vertex. In fact, in this case, the triangle that I am drawing has a different shape from the triangular area in the image, and that piece of the image will have to be stretched and distorted to fit.

Note that texture coordinates are attributes of vertices. Like other vertex attributes, values are only specified at the vetex. OpenGL will interpolate the values between vertices to calculate texture coordinates for points inside the polygon.

Sometimes, it's difficult to decide what texture coordinates to use. One case where it's easy is applying a complete texture to a rectangle. Here is a method from *Cube.java* that draws a square in the xy-plane, with appropriate texture coordinates to map the entire image onto the square:

```
    /**
     * Draws a square in the xy-plane, with given radius,
     * where radius is half the length of the side.
     */
    private void square(GL gl, double radius) {
        gl.glBegin(GL.GL_POLYGON);
        gl.glNormal3f(0,0,1);
        gl.glTexCoord2d(0,0);
        gl.glVertex2d(-radius,-radius);
        gl.glTexCoord2d(1,0);
        gl.glVertex2d(radius,-radius);
        gl.glTexCoord2d(1,1);
        gl.glVertex2d(radius,radius);
        gl.glTexCoord2d(0,1);
        gl.glVertex2d(-radius,radius);
        gl.glEnd();
    }
```

At this point, you will probably be happy to know that the shapes in the package *glutil* come with reasonable texture coordinates already defined. To use textures on these objects, you just have to create a *Texture* object, enable it, and bind it. For *Cube.java*, for example, a copy of the texture is mapped onto each face. For *UVCylinder.java* the entire texture wraps once around the cylinder, and circular cutouts from the texture are applied to the top and bottom. For a *UVCSphere.java*, the texture is wrapped once around the sphere. The flat texture has to be distorted to fit onto the sphere, but some textures, using what is called a cylindrical projection, are made to work precisely with this type of texture mapping. The textures that are used on the spheres in the *TextureDemo* example are of this type.

One last question: What happens if you supply texture coordinates that are not in the range from 0 to 1? It turns out that such values are legal, but exactly what they mean depends on the setting of two parameters in the *Texture* object. The most desirable outcome is probably that the texture is copied over and over to fill the entire plane, so that using s and t values outside the usual range will simply cause the texture to be repeated. In that case, for example, if you supply texture coordinates (0,0), (0,2), (2,2), and (2,0) for the four vertices of a square, then the square will be covered with four copies of the texture. For a repeating texture, such as a brick wall image, this can be much more effective than stretching a single copy of the image to cover the entire square. This behavior is not the default. To get this behavior for a *Texture* object *tex*, you can use the following code:

```
    tex.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
    tex.setTexParameteri(GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
```

There are two parameters to control this behavior because you can enable the behavior separately for the horizontal and the vertical directions. OpenGL, as I have said, has many parameters to control how textures are used. This is an example of how these parameters are set using the *Texture* class.

* * *

Chapter 3

Geometry

In the previous chapter, we surveyed some of the basic features of OpenGL. This chapter will fill in details about the geometric aspects of OpenGL. It will cover the full range of geometric primitives and the various ways of specifying them, and it will cover geometric transformations in more depth, including some of the mathematics behind them.

So far, everything that we have covered is part of OpenGL 1.1, and so should be universally available. This chapter will introduce some features that were introduced in later versions. It will explain why you might want to use those features and how you can test for their presence before you use them.

3.1 Vectors, Matrices, and Homogeneous Coordinates

Before moving on with OpenGL, we look at the mathematical background in a little more depth. The mathematics of computer graphics is primarily ***linear algebra***, which is the study of vectors and linear transformations. A vector, as we have seen, 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. If we visualize a vector V as starting at the origin and ending at a point P, then we can to a certain extent identify V with P—at least to the extent that both V and P have coordinates, and their coordinates are the same. 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).

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 at will.

Matrices are rectangular arrays of numbers. A matrix can be used to apply a transformation to a vector (or to a point). The geometric transformations that are so important in computer graphics are represented as matrices.

In this section, we will look at vectors and matrices and at some of the ways that they can be used. The treatment is not very mathematical. The goal is to familiarize you with the properties of vectors and matrices that are most relevant to OpenGL.

3.1.1 Vector Operations

We assume for now that we are talking about vectors in three dimensions. A 3D vector can be specified by a triple of numbers, such as (0,1,0) or (3.7,−12.88,0.02). Most of the discussion, except for the "cross product," carries over easily into other dimensions.

One of the basic properties of a vector is its ***length***. In terms of its coordinates, the length of a 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 can be denoted by $|v|$. The length of a vector is also called its ***norm***.

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.

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

```
v1·v2 = x1*x2 + x2*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 $v \cdot v$. 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 from $v1$ to $v2$. In particular, in the case of 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 dot product is 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 $v1 \times v2$ 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 $v1 \times v2$ is zero if and only if $v1$ and $v2$ point in the same direction or in exactly opposite directions. Assuming $v1 \times v2$ is non-zero, then it is perpendicular both to $v1$ and to $v2$; furthermore, the vectors $v1$, $v2$, $v1 \times v2$ follow the right-hand rule; that is, if you curl the fingers of your right hand from $v1$ to $v2$, then your thumb points in the direction of $v1 \times v2$. If $v1$ and $v2$ are unit vectors, then the cross product $v1 \times v2$ 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*** $P2-P1$ which is defined by

```
P2 − P1 = ( x2 − x1, y2 − y1, z2 − z1 )
```

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 either zero or is a vector that is perpendicular to the polygon. This fact allows us to use the vertices of a polygon to produce a normal vector to the polygon. Once we have that, we can normalize the vector to produce a unit normal. (It's possible for the cross product to be zero. This will happen if *P1*, *P2*, and *P3* lie on a line. In that case, another set of three vertices might work. Note that if all the vertices of a polygon lie on a line, then the polygon degenerates to a line segment and has no interior points at all. We don't need unit normals for such polygons.)

3.1.2 Matrices and Transformations

A matrix is just a two-dimensional array of numbers. Suppose that a matrix M has r rows and c columns. Let v be a c-dimensional vector, that is, a vector of c numbers. Then it is possible to multiply M by v to yield another vector, which will have dimension r. For a programmer, it's probably easiest to define this type of multiplication with code. Suppose that we represent M and *v by the arrays*

```
double[][] M = new double[r][c];
double[] v = new double[c];
```

Then we can define the product $w = M{*}v$ as follows:

```
double w = new double[r];
for (int i = 0; i < r; i++) {
    w[i] = 0;
    for (int j = 0; j < c; j++) {
       w[i] = w[i] + M[i][j] * v[j];
    }
}
```

If you think of a row, $M[i]$, of M as being a c-dimensional vector, then $w[i]$ is simply the dot product $M[i]{\cdot}v$.

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 called ***linear transformations***, and they are the main object of study in the field of mathematics known as linear algebra.

Rotation, scaling, and shear are linear transformations, but translation is not. To include translations, we have to widen our view to include ***affine transformations***. An affine transformation can be defined, roughly, as a linear transformation followed by a translation. 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 the precisely the vector (*x2*,*y2*,*z2*,1). That is, instead of applying the 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 OpenGL does: It represents affine transformations as 4-by-4 matrices, in which the bottom row is (0,0,0,1), and it converts three-dimensional vectors into four dimensional vectors 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 matrix multiplication.

One advantage of using matrices to represent transforms is that matrices can be multiplied. In particular, if A and B are 4-by-4 matrices, then their matrix product $A*B$ is another 4-by-4 matrix. Each of the matrices A, B, and $A*B$ represents a linear transformation. The important fact is that applying the single transformation $A*B$ to a vector v has the same effect as first applying B to v and then applying A to the result. Mathematically, this can be said very simply: $(A*B)*v = A*(B*v)$. For computer graphics, it means that the operation of following one transform by another simply means multiplying their matrices. This allows OpenGL to keep track of a single modelview matrix, rather than a sequence of individual transforms. Transform commands such as *glRotatef* and *glTranslated* are implemented as matrix multiplication—the current modelview matrix is multiplied by a matrix representing the transform that is being applied, yielding a matrix that represents the combined transform. You might compose your modelview transform as a long sequence of modeling and viewing transforms, but when the transform is actually applied to a vertex, only a single matrix multiplication is necessary. The matrix that is used represents the entire sequence of transforms, all multiplied together. It's really a very neat system.

3.1.3 Homogeneous Coordinates

There is one 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.

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. 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 three-dimensional vectors, and 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 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

```
gl.glVertex4d(x,y,z,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 required when setting the position of a light, as we'll see in the next chapter.

3.1.4 Vector Forms of OpenGL Commands

Some OpenGL commands take parameters that are vectors (or points, if you want to look at it that way) given in the form of arrays of numbers. For some commands, such *glVertex* and *glColor*, vector parameters are an option. Others, such as the commands for setting material properties, only work with vectors.

Commands that take parameters in array form have names that end in "v". For example, you can use the command *gl.glVertex3fv*(A,offset) to generate a vertex that is given by three numbers in an array A of type **float[]**. The second parameter is an integer offset value that specifies the starting index of the vector in the array. For example, you might use an array to store the coordinates of the three vertices of a triangle—say (1,0,0), (−1,2,2), and (1,1,−3)—and use that array to draw the triangle:

```
float[] vert = new float[] { 1, 0, 0, -1, 2, 2, 1, 1, -3};
gl.glBegin(GL.GL_POLYGON);
gl.glVertex3fv(vert, 0); // Equivalent to gl.glVertex3f(vert[0],vert[1],vert[2]).
gl.glVertex3fv(vert, 3); // Equivalent to gl.glVertex3f(vert[3],vert[4],vert[5]).
gl.glVertex3fv(vert, 6); // Equivalent to gl.glVertex3f(vert[6],vert[7],vert[8]).
gl.glEnd();
```

(This will make a lot more sense if you already have the vertex data in an array for some reason.)

Similarly, there are methods such as *glVertex2dv* and *glColor3fv*. The color command can be useful when working with Java *Color* objects, since a color object c has a method *c.getColorComponents*(*null*), that returns an array containing the red, green, blue, and alpha components of the color as **float** values in the range 0.0 to 1.0. (The *null* parameter tells the method to create and return a new array; the parameter could also be an array into which the method would place the data.) You can use this method and the command *gl.glColor3fv* or *gl.glColor4fv* to set OpenGL's color from the Java *Color* object c:

```
gl.glColor3fv( c.getColorComponents(null), 0);
```

(I should note that the versions of these commands in the OpenGL API for the C programming language do not have a second parameter. In C, the only parameter is a pointer, and you can pass a pointer to any array element. If *vert* is an array of floats, you can call *glVertex3fv*(*vert*) when the vertex is given by the first three elements of the array. You can call *glVertex3fv*(&*vert*[3]), or equivalently *glVertex3fv*(*vert*+3), when the vertex is given by the next three array elements, and so on. I should also mention that Java has alternative forms for these methods that use things called "buffers" instead of arrays; these forms will be covered later in the chapter.)

OpenGL has a number of methods for reading the current values of OpenGL state variables. Many of these values can be retrieved using four generic methods that take an array as a parameter:

```
gl.glGetFloatv( propertyCodeNumber, destinationArray, offset );
gl.glGetDoublev( propertyCodeNumber, destinationArray, offset );
gl.glGetIntegerv( propertyCodeNumber, destinationArray, offset );
gl.glGetBooleanv( propertyCodeNumber, destinationArray, offset);
```

In these methods, the first parameter is an integer constant such as *GL.GL_CURRENT_COLOR* or *GL.GL_MODELVIEW_MATRIX* that specifies which state variable you want to read. The second parameter is an array of appropriate type into which the retrieved value of the state variable will be placed. And offset tells the starting index in the array where the data should be placed; it will probably be 0 in most cases. (For *glGetBooleanv*, the array type is **byte[]**, and the numbers 0 and 1 are used to represent the **boolean** values *false* and *true*.) For example, to retrieve the current color, you could say

```
float[] saveColor = new float[4];  // Space for red,blue,green, and alpha.
gl.glGetFloatv( GL.GL_CURRENT_COLOR, saveColor, 0 );
```

You might use this to save the current color while you do some drawing that requires changing to another color. Later, you could restore the current color to its previous state by saying

```
gl.glColor4fv( saveColor, 0 );
```

You could use *glGetDoublev* instead of *glGetFloatv*, if you prefer. OpenGL will convert the data to the type that you request. However, as usual, **float** values might be the most efficient. There are many, many state variables that you can read in this way. I will mention just a few more as examples. You can read the current viewport with

```
int[] viewport = new int[4];  // Space for x, y, width, height.
gl.glGetIntegerv( GL.GL_VIEWPORT, viewport, 0 );
```

For boolean state variables such as whether lighting is currently enabled, you can call *glGet-Booleanv* with an array of length 1:

```
byte[] lit = new byte[1];  // Space for the single return value
gl.glGetBooleanv( GL.GL_LIGHTING, lit );
```

I might mention that if you want to retrieve and save a value so that you can restore it later, there are better ways to do so, which will be covered later in the chapter.

Perhaps most interesting for us now, you can retrieve the current transformation matrices. A transformation matrix is represented by a one-dimensional array of length 16. The 4-by-4 transformation matrix is stored in the array in column-major order, that is, the entries in the first column from top to bottom, followed by the entries in the second column, and so on. You could retrieve the current modelview transformation with

```
float[] transform = new float[16];
gl.glGetFloatv( GL.GL_MODELVIEW, transform, 0 );
```

It is possible to set the transformation matrix to a value given in the same form, using the *glLoadMatrixf* method. For example, you could restore the modelview matrix to the value that was retrieved by the previous command using

```
gl.glLoadMatrixf( transform, 0 );
```

But again, if you just want to save a transform so that you can restore it later, there is a better way to do it—in this case by using *glPushMatrix* and *glPopMatrix*.

Just as you can multiply the current transform matrix by a rotation matrix by calling *glRotatef* or by a translation matrix by calling *glTranslatef*, you can multiply the current transform matrix by an arbitrary matrix using *gl.glMultMatrixf*(*matrix*,*offset*). As with the other matrix methods, the *matrix* is given by a one-dimensional array of 16 **floats**. One situation in which this can be useful is to do shear transformations. For example, consider a shear that transforms the vector (*x1*,*y1*,*z1*) to the vector (*x2*,*y2*,*z2*) given by

```
x2 = x1 + s * z1
y2 = y1
z2 = z1
```

where *s* is a constant shear amount. The 4-by-4 matrix for this transformation is hard to construct from scaling, rotation, and translation, but it is very simple:

```
1  0  s  0
0  1  0  0
0  0  1  0
0  0  0  1
```

To use this shear transformation as a modeling transform, you can use *glMultMatrixf* as follows:

```
float[] shear = new float[] { 1,0,0,0,  0,1,0,0,  s,0,1,0,  0,0,0,1 };
gl.glMultMatrixf( shear, 0 );
```

3.2 Primitives

The term "primitive" in computer graphics refers to a geometry element that is not made up of simpler elements. In OpenGL, the primitives are points, lines (meaning line segments), and polygons. Other graphics systems might have additional primitives such as Bezier curves, circles, or spheres, but in OpenGL, such shapes have to be approximated by lines or polygons.

Primitives can be generated using *glBegin/glEnd* or by more efficient methods that we will cover in Section 3.3. In any case, though, the type of primitive that you are generating is indicated by a constant such as *GL.GL_POLYGON* that could be passed as a parameter to *gl.glBegin*. There are ten such constants, which are said to define ten different ***primitive types*** in OpenGL. (For most of these primitive types, a single use of *glBegin/glEnd* can produce several basic primitives, that is, several points, lines, or polygons.)

In this section, we'll look at the ten primitive type constants and how they are used to generate primitives. We'll also cover some of the attributes that can be applied to primitives to modify their appearance.

3.2.1 Points

Individual points can be generated by using the primitive type *GL.GL_POINTS*. With this primitive type, each vertex that is specified generates an individual point. For example, the following code generates 10 points lying along the circumference of a circle of radius 1 in the xy-plane:

```
gl.glBegin(GL.GL_POINTS);
for (int i = 0; i <  10; i++){
   double angle = (2*Math.PI/10)*i;  // Angle in radians, for Java's functions.
   gl.glVertex3d( Math.cos(angle), Math.sin(angle), 0);
}
gl.glEnd();
```

By default, a point is drawn as a single pixel. If lighting is off, the pixel takes its color from the current drawing color as set by one of the *glColor* methods. If lighting is on, then the same lighting calculations are done for the point as would be done for the vertex of a polygon, using the current material, normal vector, and so on; this can have some interesting effects, but generally it makes more sense to draw points with lighting disabled.

An individual pixel is just barely visible. The size of a point can be increased by calling *gl.glPointSize(size)*. The parameter is a **float** that specifies the diameter of the point, in pixels. The range of sizes that is supported depends on the implementation.

A point that has size larger than 1 appears by default as a square, which might be a surprise. To get circular points, you have to turn on point smoothing by calling

```
gl.glEnable(GL.GL_POINT_SMOOTH);
```

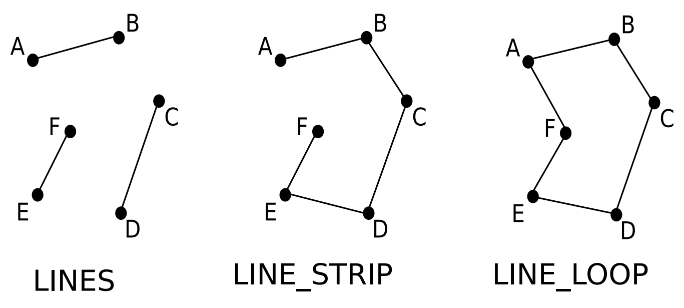
Just turning on this option will leave the edges of the point looking a little jagged. You can get nicely antialiased points by also using ***blending***. Recall that antialiasing can be implemented by using partial transparency for pixels that are only partially covered by the object that is being drawn. When a partially transparent pixel is drawn, its color is blended with the previous color of the pixel, rather than replacing it. Unfortunately, blending and transparency are fairly complicated topics in OpenGL. We will discuss them in more detail in the next chapter. For now, I will just note that you can generally get good results for antialiased points by setting the following options, in addition to *GL_POINT_SMOOTH*:

```
gl.glEnable(GL.GL_BLEND);
gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA);
```

However, you should remember to turn off the *GL_BLEND* option when drawing polygons.

3.2.2 Lines

There are three primitive types for generating lines: *GL.GL_LINES*, *GL.GL_LINE_STRIP*, and *GL.GL_LINE_LOOP*. With *GL_LINES*, the vertices that are specified are paired off, and each pair is joined to produce a line segment. If the number of vertices is odd, the last vertex is ignored. When *GL_LINE_STRIP* is used, a line is drawn from the first specified vertex to the second, from the second to the third, from the third to the fourth, and so on. *GL_LINE_LOOP* is similar, except that one additional line segment is drawn from the final vertex back to the first. This illustration shows the line segments that would be drawn if you specified the points A, B, C, D, E, and F, in that order, using the three primitive types:



When a line segment is drawn, a color is first assigned to each vertex, as usual, just as if it were a vertex in a polygon, including all lighting calculations when lighting is enabled. If the current shade model is *GL_SMOOTH*, then the vertex colors are interpolated between the two vertices to assign colors to the points along the line. If the shade model is *GL_FLAT*, then the color of the second vertex is used for the entire segment.

By default, the width of a line is one pixel. The width can be changed by calling *gl.glLineWidth(width)*, where *width* is a value of type **float** that specifies the line width in pixels. To get antialiased lines, you should call

```
gl.glEnable(GL.GL_LINE_SMOOTH);
```

and you should set up blending with the same two commands that were given above for antialiased points.

A line can also have an attribute called a ***stipple pattern*** which can be used to produce dotted and dashed lines. The stipple pattern is set by calling

```
gl.glLineStipple( multiplier, pattern );
```

where *pattern* is a value of type **short** and *multiplier* is an integer. The 16 bits in *pattern* specify how drawing is turned on and off along the line, with 0 turning drawing off and 1 turning it on. For example, the value `(short)0x0F33` represents the binary number `0000111100110011`, and drawing is tuned off for four steps along the line, on for four steps, off for two, on for two, off for two, and finally on for the final two steps along the line. The pattern then repeats. The *multiplier* tells the size of the steps. When *multiplier* is one, each bit in *pattern* corresponds to one pixel-length along the line; when *multiplier* is two, each bit in *pattern* corresponds to two pixel-lengths; and so on. The pattern value `(short)0x0F33` would draw the line with a long-dash / short-dash / short-dash stippling, and the multiplier value would determine the length of the dashes. (Exactly the same stippling could be produced by using *pattern* equal to `(short)0x3535` and doubling the multiplier.)

The line stipple pattern is only applied if the *GL_LINE_STIPPLE* option is enabled by calling

```
gl.glEnable(GL.GL_LINE_STIPPLE);
```

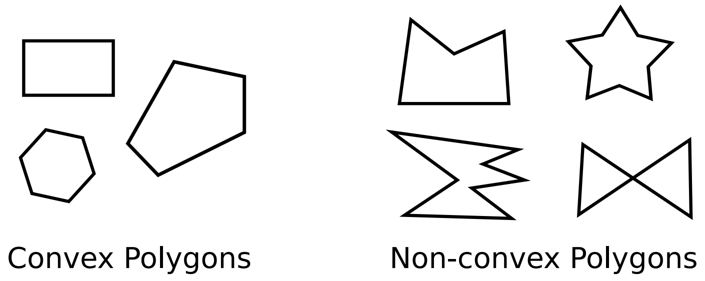
3.2.3 Polygons

The remaining six primitive types are for drawing polygons. The familiar *GL.GL_POLYGON*, which we used almost exclusively before this section, is used to draw a single polygon with any number of sides. The other five polygon primitives are for drawing triangles and quadrilaterals and will be covered later in this section.

Up until now, I haven't bothered to mention an important fact about polygons in OpenGL: It is correct to draw a polygon in OpenGL only if the polygon satisfies certain requirements. OpenGL will attempt to draw a polygon using any sequence of vertices that you give it, by following its usual rendering algorithms. However, for polygons that don't meet the requirements, there is no reason to expect that the results will be geometrically correct or meaningful, and they might vary from one OpenGL implementation to another.

First of all, very reasonably, any polygon drawn by OpenGL should be ***planar***; that is, all of its vertices should lie in the same plane. It is not even clear what a non-planar polygon would mean. But you should be aware that OpenGL will not report an error if you specify a non-planar polygon. It will just draw whatever its algorithms produce from the data you give them. (Note that polygons that are very close to being planar will come out OK, so you don't have to be too obsessive about it.)

The other requirement is less reasonable: Polygons drawn by OpenGL must be ***convex***. The technical definition of convex is that whenever two points lie in the polygon, then the entire line segment between the two points also lies in the polygon. Informally, convexity means that the polygon does not have any indentations along its edge. Note that a convex polygon can never be self-intersecting, like the non-convex polygon on the lower right in the illustration below.



In order to draw a non-convex polygon with OpenGL, you have to sub-divide it into several smaller convex polygons and draw each of those smaller polygons.

When drawing polygons, you should keep in mind that a polygon has a front side and a back side, and that OpenGL determines which side is the front by the order in which the vertices of the polygon are generated. In the default setup, the vertices have to be specified in counterclockwise order when viewed from the front (and therefore in clockwise order when viewed from the back). You can reverse this behavior by calling *gl.glFrontFace*(*GL.GL_CW*). You might do this, for example, if you read your polygon data from a file that lists vertices in clockwise order as seen from the front.

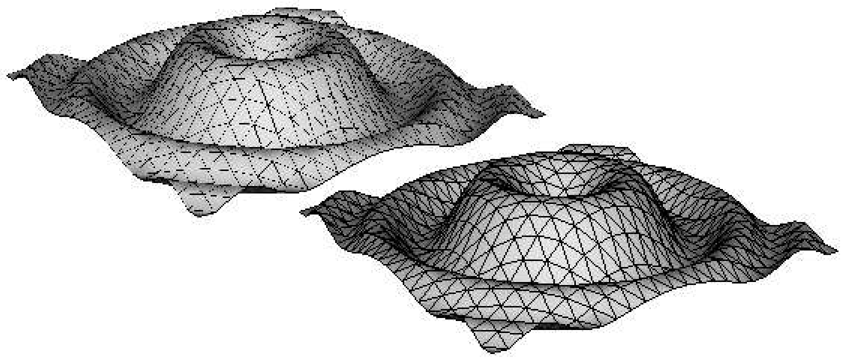
The front/back distinction is used in two cases. First, as we will see in the next chapter, it is possible to assign different material properties to the front faces and to the back faces of polygons, so that they will have different colors. Second, OpenGL has an option to do ***back-face culling***. This option can be turned on with *gl.glEnable*(*GL.GL_CULL_FACE*). It tells OpenGL to ignore a polygon when the user is looking at the back face of that polygon. This can be done for efficiency when the polygon is part of a solid object, and the back faces of the objects that make up the object are all facing the interior of the object. In that case, the back faces will not be visible in the final image, so any time spent rendering them would be wasted.

When drawing polygons, whether with *GL_POLYGON* or with the more specialized primitive types discussed below, the default is to to draw the interior of the polygons. Sometimes, it is useful just to draw the edges of the polygon. For example, you might want to draw a "wire-frame" representation of a solid object, showing just the polygon edges. Or you might want to fill a polygon with one color and then outline it with another color. You can use *glPolygonMode* to determine how polygons are rendered. You can actually apply different settings to front faces and back faces at the same time. Assuming that you want to treat both faces the same, the three possible settings are:

```
gl.glPolygonMode( GL.GL_FRONT_AND_BACK, GL.GL_FILL )
gl.glPolygonMode( GL.GL_FRONT_AND_BACK, GL.GL_LINE )
gl.glPolygonMode( GL.GL_FRONT_AND_BACK, GL.GL_POINT )
```

The default mode is *GL_FILL*, where the interior of the polygon is drawn. The *GL_LINE* mode produces an outline of the polygon. And the *GL_POINT* mode just places a point at each vertex of the polygon.

To draw both the interior and the outline, you have to draw the polygon twice, once using the *GL_FILL* polygon mode and once using *GL_LINE*. When you do this, however, you run into a problem with the depth test that was discussed in Subsection 2.2.2: Along an edge of the polygon, the interior and the outline of the polygon are at the same depth, and because of computational inaccuracy, the depth test can give erratic results when comparing objects that have the same depth values. At some points, you see the edge; at others, you see the interior color. OpenGL has a fix for this problem, called "polygon offset." Here are two pictures of the same surface. The surface is rendered by drawing a large number of triangles. The picture on the top left does not use polygon offset; the one on the bottom right does:



Polygon offset can be used to add a small amount to the depth of the pixels that make up the polygon. In this case, I applied polygon offset while filling the triangles but not while drawing their outlines. This eliminated the problem with drawing things that have the same depths, since the depth values for the polygon and its outline are no longer exactly the same.

Here is some code for drawing the above picture with polygon offset, assuming that the method *drawSurface* generates the polygons that make up the surface.

```
gl.glEnable(GL.GL_LIGHTING);   // Draw polygon interiors with lighting enabled.
gl.glPolygonMode(GL.GL_FRONT_AND_BACK, GL.GL_FILL);  // Draw polygon interiors.

gl.glEnable(GL.GL_POLYGON_OFFSET_FILL);  // Turn on offset for filled polygons.
gl.glPolygonOffset(1,1);      // Set polygon offset amount.

drawSurface(gl);      // Fill the triangles that make up the surface.

gl.glDisable(GL.GL_LIGHTING);  // Draw polygon outlines with lighting disabled.
gl.glPolygonMode(GL.GL_FRONT_AND_BACK, GL.GL_LINE);  // Draw polygon outlines.

gl.glColor3f(0,0,0);  // Draw the outlines in black.
drawSurface(gl);      // Outline the triangles that make up the surface again.
```

Polygon offset can be enabled separately for the *FILL*, *LINE*, and *POINT* polygon modes. Here, it is enabled only for the *FILL* mode, so it does not apply when the outlines are drawn. The *glPolygonOffset* method specifies the amount of the offset. The parameters used here, (1,1), produce a minimal change in depth, just enough to make the offset work. (The first parameter, which I don't completely understand, compensates for the fact that a polygon that is tilted relative to the screen will require a bigger change in depth to be effective.)

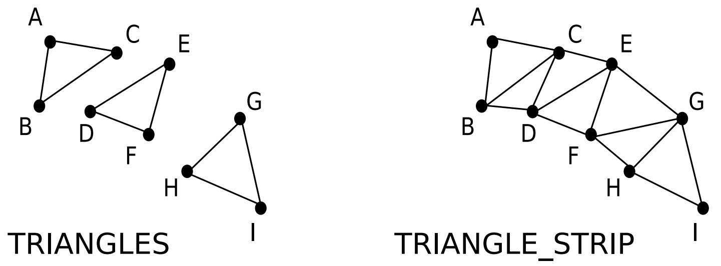
3.2.4 Triangles

Triangles are particularly common in OpenGL. When you build geometry out of triangles, you don't have to worry about planarity or convexity, since every triangle is both planar and convex. OpenGL has three primitive types for drawing triangles: *GL.GL_TRIANGLES*, *GL.GL_TRIANGLE_STRIP*, and *GL.GL_TRIANGLE_FAN*.

With *GL_TRIANGLES*, the vertices that you specify are grouped into sets of three, and every set of three generates a triangle. If the number of vertices is not a multiple of three, then the one or two extra vertices are ignored. You can, of course, use *GL_TRIANGLES* to specify a single triangle by providing exactly three vertices.

With *GL_TRIANGLE_STRIP*, the first three vertices specify a triangle, and each additional vertex adds another triangle, which is formed from the new vertex and the two previous vertices. The triangles form a "strip" or "band" with the vertices alternating from one side of the strip to the other.

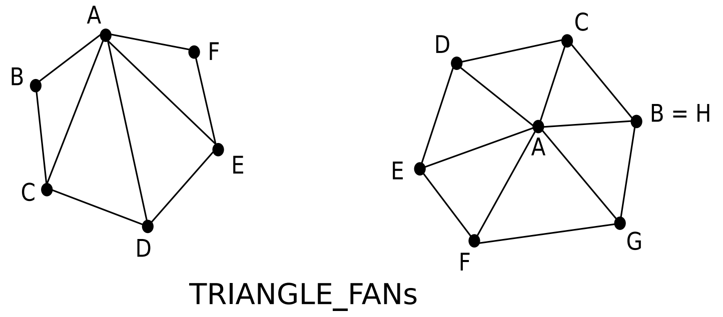
This illustration shows the triangles produced using *GL_TRIANGLES* and *GL_TRIANGLE_STRIP* with the same set of vertices A, B, . . . , I, specified in that order:



Note that the vertices of triangle DEF in the picture on the left are specified in clockwise order, which means that you are looking at the back face of this triangle. On the other hand, you are looking at the front face of triangles ABC and GHI. When using *GL_TRIANGLES*, each triangle is treated as a separate polygon, and the usual rule for determining the front face apply.

In the picture that is made by *GL_TRIANGLE_STRIP*, on the other hand, you are looking at the front faces of all the triangles. To make this true, the orientation of every second triangle is reversed; that is, the order of the vertices is considered to be the reverse of the order in which they are specified. So, in the first triangle, the ordering is ABC; in the second, the order is DCB; in the third, CDE; in the fourth, FED, and so on. This might look complicated, but you rarely have to think about it, since it's the natural order to ensure that the front faces of all the triangles are on the same side of the strip.

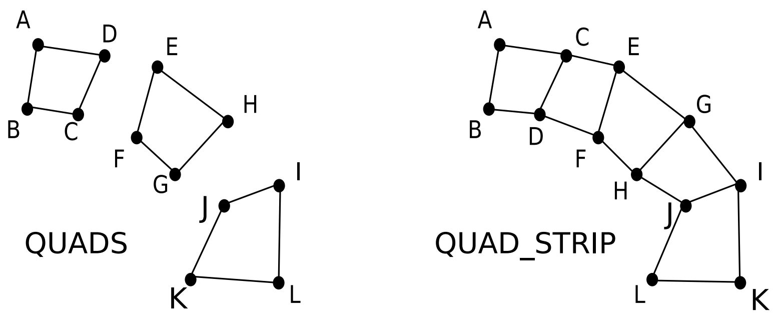
The third primitive for drawing triangles is *GL_TRIANGLE_FAN*. It is less commonly used than the other two. In a triangle fan, the first vertex that is specified becomes one of the vertices of every triangle that is generated, and those triangles form a "fan" around that initial vertex. This illustration shows typical uses of *GL_TRIANGLE_FAN*:



3.2.5 Quadrilaterals

Finally, OpenGL has two primitives for drawing quadrilaterals: *GL.GL_QUADS* and *GL.GL_QUAD_STRIP*. A quadrilateral is a four-sided polygon. As with any polygon in OpenGL, to be valid, a quadrilateral must be planar and convex. Because of this restriction, it's more common to use triangles than quads. However, there are many common objects for which quads are appropriate, including all the shape classes in my *glutil* package.

GL_QUADS, as you can probably guess, draws a sequence of independent quadrilaterals, using groups of four vertices. *GL_QUAD_STRIP* draws a strip or band made up of quadrilaterals. Here is an illustration showing the quadrilaterals produced by *GL_QUADS* and *GL_QUAD_STRIP*, using the same set of vertices—although **not** generated in the same order:



For *GL_QUADS*, the vertices of each quadrilateral should be specified in counterclockwise order, as seen from the front. For *GL_QUAD_STRIP*, the vertices should alternate from one side of the strip to the other, just as they do for *GL_TRIANGLE_STRIP*. In fact, *GL_TRIANGLE_STRIP* can always be used in place of *GL_QUAD_STRIP*, with the same vertices in the same order. (The reverse is not true, since the quads that are produced when *GL_QUAD_STRIP* is substituted for *GL_TRIANGLE_STRIP* might not be planar.)

As an example, let's approximate a cylinder—without a top or bottom—using a quad strip. Assume that the cylinder has radius r and height h and that its base is in the xy-plane, centered at (0,0). If we use a strip of 32 quads, then the vertices along the bottom of the cylinder have coordinates ($r{*}cos(d{*}i)$,$r{*}sin(d{*}i)$,0), for i = 0, ...,31, where d is 1/32 of a complete circle. Similarly, the vertices along the top edge of the cylinder have coordinates ($r{*}cos(d{*}i)$,$r{*}sin(d{*}i)$,h). Furthermore, the unit normal to the cylinder at the point ($r{*}cos(d{*}i)$,$r{*}sin(d{*}i)$,z) is ($cos(d{*}i)$,$sin(d{*}i)$,0) for both $z = 0$ and $z = h$. So, we can use the following code to generate the quad strip:

```
double d = (1.0/32) * (2*Math.PI);  // 1/32 of a circle, for Java's functions.
gl.glBegin(GL.GL_QUAD_STRIP);
for (int i = 0; i <= 32; i++) {
    // Generate a pair of points, on top and bottom of the strip.
    gl.glNormal3d( Math.cos(d*i), Math.sin(d*i, 0);  // Normal for BOTH points.
    gl.glVertex3d( r*Math.cos(d*i), r*Math.sin(d*i), h );  // Top point.
    gl.glVertex3d( r*Math.cos(d*i), r*Math.sin(d*i), 0 );  // Bottom point.
}
gl.glEnd();
```

The order of the points, with the top point before the bottom point, is chosen to put the front faces of the quads on the outside of the cylinder. Note that if *GL_TRIANGLE_STRIP* were substituted for *GL_QUAD_STRIP* in this example, the results would be identical as long as the usual *GL_FILL* polygon mode is selected. If the polygon mode is set to *GL_LINE*, then *GL_TRIANGLE_STRIP* would produce an extra edge across the center of each quad.

* * *

In the on-line version of this section, you will find an applet that lets you experiment with the ten primitive types and a variety of options that affect the way they are drawn.

3.3 Polygonal Meshes

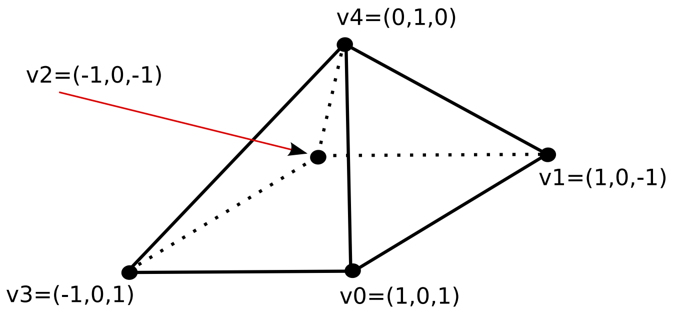
When creating objects that are made up of more than just a few polygons, we need some way to organize the data and some strategy for using it effectively. The problem is how to represent ***polygonal meshes*** for efficient processing. A polygonal mesh is just a collection of connected polygons used to model a two-dimensional surface. The term includes polyhedra such as a cube or a dodecahdron, as well as meshes used to approximate smooth surfaces such as the geometry produced by my *UVSphere* class. In these cases, the mesh represents the surface of a solid object. However, the term is also used to refer to open surfaces with no interior, such as a mesh representing the graph of a function $z = f(x,y)$.

There are many data structures that can be used to represent polygonal meshes. We will look at one general data structure, the indexed face set, that is often used in computer graphics. We will also consider the special case of a polygonal grid.

3.3.1 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 pyramid, a simple five-sided polyhedron:



The vertex list for this polyhedron has the form

```
Vertex #0.  (1,0,1)
Vertex #1.  (1,0,-1)
Vertex #2.  (-1,0,-1)
Vertex #3.  (-1,0,1)
Vertex #4.  (0,1,0)
```

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

To describe a polygon, we have to give its vertices, being careful to enumerate them in counterclockwise order as seen from the front of the polygon. In this case, the fronts of the polygonal faces of the pyramid should be on the outside of the pyramid. Now for an IFS, we can specify a vertex by giving its index in the list. For example, we can say that one of the faces of the pyramid is the polygon formed by vertex #4, vertex #3, and vertex #0. To complete the data for our IFS, we just give the indexes of the vertices for each face:

```
Face #1:  4 3 0
Face #2:  4 0 1
Face #3:  4 1 2
Face #4:  4 2 3
Face #5:  0 3 2 1
```

In Java, we can store this data effectively in two two-dimensional arrays:

```
float[][] vertexList = {  {1,0,1}, {1,0,-1}, {-1,0,-1}, {-1,0,1}, {0,1,0}  };
int[][] faceList   = {  {4,3,0},  {4,0,1},  {4,1,2},  {4,2,3},  {0,3,2,1}  };
```

Note that each vertex is repeated three or four times in the face-list array. With the IFS representation, a vertex is represented in the face list by a single integer, giving an index into the vertex list array. This representation uses significantly less memory space than the alternative, which would be to write out the vertex in full each time it occurs in the face list. For this example, the IFS representation uses 31 numbers to represent the polygonal mesh. as opposed to 48 numbers for the alternative.

IFSs 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 15 numbers in the vertex list. The values in the face list will remain unchanged. So, in this example, we would only be modifying 15 numbers, instead of 48.

Suppose that we want to use the IFS data to draw the pyramid. It's easy to generate the OpenGL commands that are needed to generate the vertices. But to get a correct rendering, we also need to specify normal vectors. When the IFS represents an actual polyhedron, with flat faces, as it does in this case, we can compute a normal vector for each polygon, as discussed in Subsection 3.1.1. (In fact, if *P1*, *P2*, and *P3* are the first three vertices of the polygon, then the cross product $(P3-P2)\times(P1-P2)$ is the desired normal as long as the three points do not lie on a line.) Let's suppose that we have a method, *computeUnitNormalForPolygon*, to compute the normal for us. Then the object represented by an IFS can be generated from the *vertexList* and *faceList* data with the following simple method:

```
static void drawIFS(GL gl, float[][] vertexList, int[][] faceList) {
    for (int i = 0; i < faceList.length; i++) {
       gl.glBegin(GL.GL_POLYGON);
       int[] faceData = faceList[i];  // List of vertex indices for this face.
       float[] normal = computeUnitNormalForPolygon(vertexList, faceData);
       gl.glNormal3fv( normal, 0 );
       for (int j = 0; j < faceData.length; j++) {
           int vertexIndex = faceData[j];  // Location of j-th vertex in list.
           float[] vertex = vertexList[ vertexIndex ];
           gl.glVertex3fv( vertex, 0 );
       }
       gl.glEnd();
    }
}
```

This leaves open the question of what to do about normals for an indexed face set that is meant to approximate a smooth surface. In that case, we want to use normals that are perpendicular to the surface, not to the polygons. One possibility is to expand our data structure to store a normal vector for each vertex, as we will do in the next subsection. Another possibility, which can give nice-looking results, is to calculate an approximate normal for each vertex from the data that we have. To do this for a particular vertex *v*, consider all the polygons in which *v* is a vertex. Compute vectors perpendicular to each of those polygons, and then use the **average** of all those vectors as the normal vector at *v*. Although that vector is probably not exactly perpendicular to the surface at *v*, it is usually good enough to pass visual inspection of the resulting image.

* * *

Before moving on, lets think about making a minor change to the way that we have stored the data for the vertex list. It is common in OpenGL to store this type of data in a one-dimensional array, instead of a two dimensional array. Store the first vertex in the first three array elements, the second vertex in the next three, and so on:

```
float[] vertexList = {  1,0,1, 1,0,-1, -1,0,-1, -1,0,1, 0,1,0  };
```

In general, the data for vertex number n will be stored starting at position $3*n$ in the array. Recall that the second parameter to *gl.glVertex3fv* gives the position in the array where the data for the vertex is stored. So, to use our modified data, we can now draw the IFS as follows:

```
static void drawIFS(GL gl, float[] vertexList, int[][] faceList) {
    for (int i = 0; i < faceList.length; i++) {
       gl.glBegin(GL.GL_POLYGON);
       int[] faceData = faceList[i];  // List of vertex indices for this face.
       float[] normal = computeUnitNormalForPolygon(vertexList, faceData);
       gl.glNormal3fv( normal, 0 );
       for (int j = 0; j < faceData.length; j++) {
           int vertexIndex = faceData[j];
           gl.glVertex3fv( vertexList, 3*vertexIndex );
```

```
            }
            gl.glEnd();
        }
    }
```

This representation will be convenient when we look at new methods for drawing OpenGL primitives in the next section.

3.3.2 OBJ Files

For complex shapes that are not described by any simple mathematical formula, it's not feasible to generate the shape using Java code. We need a way to import shape data into our programs from other sources. The data might be generated by physical measurement, for example, or by an interactive 3D modeling program such as Blender (http://www.blender.org). To make this possible, one program must write data in a format that can be read by another program. The two programs need to use the same graphics file format. One of the most common file formats for the exchange of polygonal mesh data is the ***Wavefront OBJ*** file format. Although the official file format can store other types of geometric data, such as Bezier curves, it is mostly used for polygons, and that's the only use that we will consider here.

An OBJ file (with file extension ".obj") can store the data for an indexed face set, plus normal vectors and texture coordinates for each vertex. The data is stored as plain, human-readable text, using a simple format. Lines that begin with "v", "vn", "vt", or "f", followed by a space, contain data for one vertex, one normal vector, one set of texture coordinates, or one face, respectively. For our purposes here, other lines can be ignored.

A line that specifies a vertex has the form `v` x y z, where x, y, and z are numeric constants giving the coordinates of the vertex. For example:

```
v  0.707  -0.707  1
```

Four numbers, specifying homogeneous coordinates, are also legal but, I believe, rarely used. All the "v" lines in the file are considered to be part of one big list of vertices, and the vertices are assigned indices based on their position in the list. The indices start at one not zero, so vertex 1 is the vertex specified by the first "v" line in the file, vertex 2 is specified by the second "v" line, and so on. Note that there can be other types of lines interspersed among the "v" lines—those extra lines are not counted when computing the index of a vertex.

Lines starting with "vn" or "vt" work very similarly. Each "vn" line specifies a normal vector, given by three numbers. Normal vectors are not required to be unit vectors. All the "vn" lines in the file are considered to be part of one list of normal vectors, and normal vectors are assigned indices based on their order in the list. A "vt" line defines texture coordinates with one, two, or three numbers. (Two numbers would be used for 2D image textures.) All the "vt" lines in the file create a list of texture coordinates, which can be referenced by their indices in the list.

Faces are more complicated. Each "f" line defines one face, that is, one polygon. The data on the "f" line must give the list of vertices for the face. The data can also assign a normal vector and texture coordinates to each vertex. Vertices, texture coordinates, and normals are referred to by giving their indices in the respective lists. (Remember that the numbering starts from one, not from zero; if you've stored the data in Java arrays, you have to subtract 1 from the numbers given in the "f" line to get the correct array indices. There reference numbers can be *negative*. A negative index in an "f" line means to count backwards from the position of the "f" line in the file. For example, a vertex index of -1 refers to the "v" line that was seen most recently in the file, before encountering the “f” line; a vertex index of −2 refers the “v” line that precedes that one, an so on. If you are reading the file sequentially and storing data in arrays as you go, then −1 simply refers to the last item that is currently in the array, −2 refers to the next-to-last item, and so on.)

In the simple case, where there are no normals or texture coordinates, an “f” line can simply list the vertex indices in order. For example, an OBJ file for the pyramid example from the previous subsection could look like this:

```
v 1 0 1
v 1 0 -1
v -1 0 -1
v -1 0 1
v 0 1 0
f 5 4 1
f 5 1 2
f 5 2 3
f 5 3 4
f 1 4 3 2
```

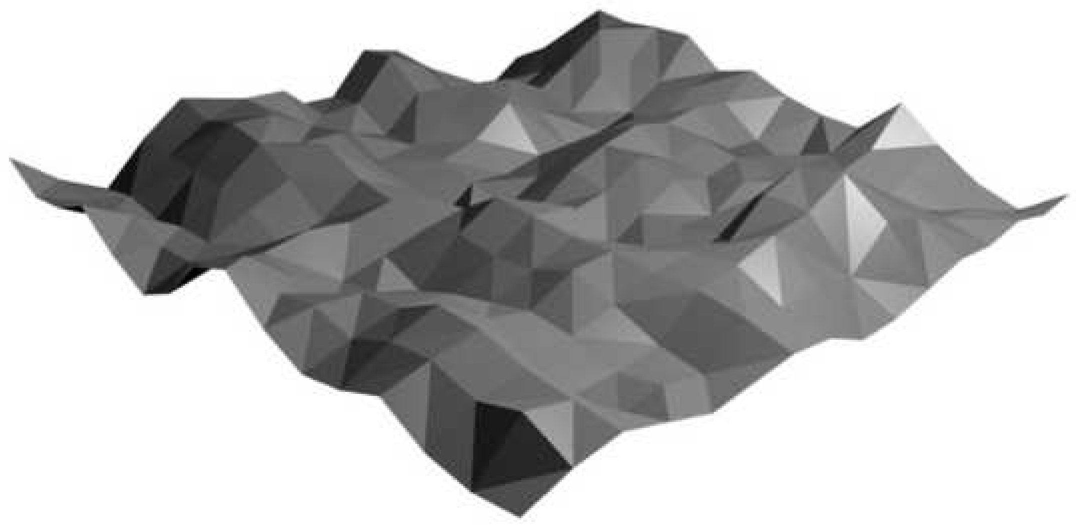
When texture coordinate or normal data is included, a single vertex index such as “5” is replaced by a data element in the format $v/t/n$, where v is a vertex index, t is a texture coordinates index, and n is a normal coordinate index. The texture coordinates index can be left out, but the two slash characters must still be there. For example, “5/3/7” specifies vertex number 5, with texture coordinates number 3, and normal vector number 7. And “2//1” specifies vertex 2 with normal vector 7. As an example, here is complete OBJ file representing a cube, with its normal vectors, exactly as exported from Blender. Note that it contains additional data lines, which we want to ignore:

```
# Blender3D v248 OBJ File:
# www.blender3d.org
mtllib stage.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vn -0.000000 -1.000000 0.000000
vn 0.000000 1.000000 -0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
usemtl Material
s off
f 1//1 2//1 3//1 4//1
f 5//2 8//2 7//2 6//2
f 1//3 5//3 6//3 2//3
f 2//4 6//4 7//4 3//4
f 3//5 7//5 8//5 4//5
f 5//6 1//6 4//6 8//6
```

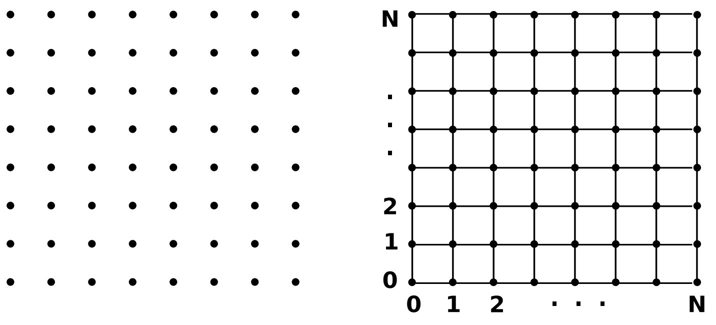
Once you have read a geometric object from an OBJ file and stored the data in arrays, it is easy enough to use the arrays to draw the object with OpenGL.

3.3.3 Terraine and Grids

We look at one more common type of polygonal mesh, which occurs when the vertices in the mesh form a grid or net. A typical example in computer graphics programs would be to represent the ground or terrain by specifying the height of the ground at each point in a grid. That is, for example, we could use a two-dimensional array $height[i][j]$, for 0 <= i <= N and 0 <= j <= N, to give the height of the ground over each of the points $(i/N, j/N)$. (This defines the ground height only over a square that is one unit on a side, but we can easily scale to extend the terrain over a larger square.) The ground would then be approximated by a polygonal mesh containing the points $(i/N, height[i][j], j/N)$, with the points joined by line segments. The result might look something like this (although certainly a larger value of N would be desirable):



Imagine looking down from above at the original grid of points and at the same grid connected by lines:



You can see that the geometry consists of N horizontal "strips." Each of these strips can be drawn using the *GL_TRIANGLE_STRIP* primitive. Again, we have to worry about normal vectors, but let's assume again that we have a way to compute them (although the computation in this case is more difficult than for indexed face sets). If we have a texture that we want to apply to the terrain, we can do so easily. As we have set things up, all the original grid points have xz-coordinates in the unit square, and we can simply use those coordinates as texture coordinates. We can then use the following method to draw the terrain

```
static void drawTerrain(GL gl, float[][] height, int N) {
    for (int row = 0; row < N; row++) {
       gl.glBegin(GL.GL_TRIANGLE_STRIP);
       // Draw strip number row.
       for (int col = 0; col <= N; col++) {
           // Generate one pair of points on the strip, at horizontal position col.
           float x1 = (float)col/N;     // Upper point on strip.
           float y1 = height[row+1][col];
           float z1 = (float)(row+1)/N;
           float[] normal1 = computeTerrainUnitNormal(height,N,row+1,col);
           gl.glNormal3fv( normal1, 0 );
           gl.glTexCoord2f( x1, z1 );
           gl.glVertex3f( x1, y1, z1 );
           float x2 = (float)col/N;     // Lower point on strip.
           float y2 = height[row][col];
           float z2 = (float)row/N;
           float[] normal2 = computeTerrainUnitNormal(height,N,row,col);
           gl.glNormal3fv( normal2, 0 );
           gl.glTexCoord2f( x2, z2 );
           gl.glVertex3f( x2, y2, z2 );
       }
       gl.glEnd();
    }
}
```

* * *

For readers with some background in calculus, this example is actually just a special case of drawing the graph of a mathematical function of two variables by plotting some points on the surface and connecting them to form lines and triangles. In that case, thinking of the function as giving $y = f(x,z)$, we would use the points $(x,f(x,z),z)$ for a grid of points (x,z) in the xz-plane.

The graph of the function is a surface, which will be smooth if f is a differentiable function. We are approximating the smooth surface by a polygonal mesh, so we would like to use normal vectors that are perpendicular to the surface. The normal vectors can be calculated from the partial derivatives of f.

More generally, we can easily draw a polygonal approximation of a ***parametric surface*** given by a set of three functions of two variables, $x(u,v)$, $y(u,v)$, and $z(u,v)$. Points on the surface are given by $(x,y,z) = (x(u,v),\ y(u,v),\ z(u,v))$. The surface defined by a rectangular area in the uv-plane can be approximated by plotting points on the surface for a grid of uv-points in that rectangle and using the resulting points to draw a sequence of triangle strips. The normal at a point on the surface can be computed from partial derivatives. In fact, if we use the notation $\mathbf{D}_u$ to mean the partial derivative with respect to u and $\mathbf{D}_v$ for the partial derivative with respect to v, then a normal at the point $(x(u,v),\ y(u,v),\ z(u,v))$ is given by the cross product

$$(\mathbf{D}_u x(u,v),\ \mathbf{D}_u y(u,v),\ \mathbf{D}_u z(u,v)) \times (\mathbf{D}_v x(u,v),\ \mathbf{D}_v y(u,v),\ \mathbf{D}_v z(u,v))$$

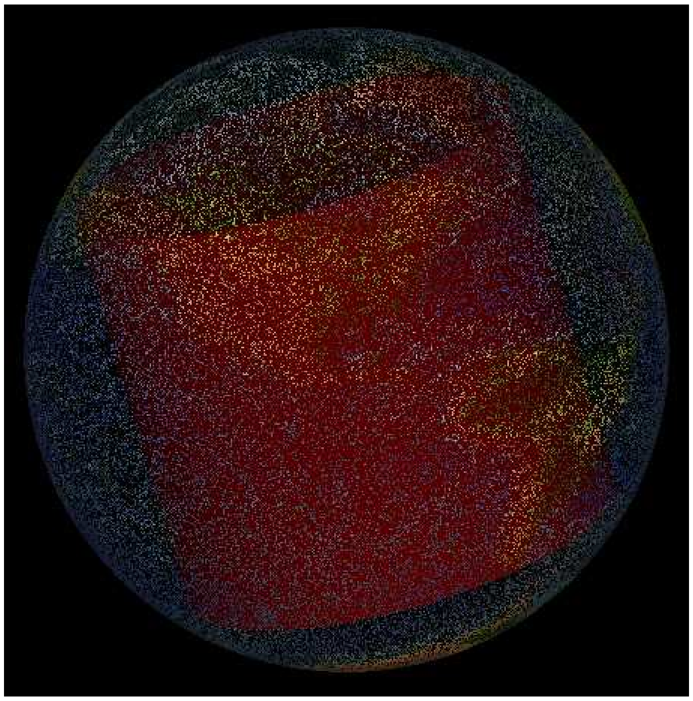
In fact, it's not even necessary to compute exact formulas for the partial derivatives. A rough numerical approximation, using the difference quotient, is good enough to produce nice-looking surfaces.

3.4 Drawing Primitives

We have been exclusively using glBegin/glEnd for drawing primitives, but that is just one of the approaches to drawing that OpenGL makes available. In fact, the glBegin/glEnd paradigm is considered to be a little old-fashioned and it has even been deprecated in the latest versions of OpenGL. This section will discuss alternative approaches.

The alternative approaches are more efficient, because they make many fewer calls to OpenGL commands and because they offer the possibility of storing data in the graphics card's memory instead of retransmitting the data every time it is used. Unfortunately, the alternative drawing methods are more complicated than glBegin/glEnd. This is especially true in Java (as opposed to C), because the implementation of arrays in Java makes Java arrays unsuitable for use in certain OpenGL methods that have array parameters in C. Jogl's solution to the array problem is to use "nio buffers" instead of arrays in those cases were Java arrays are not suitable. The first subsection, below, discusses nio buffers.

To give us something to draw in this section, we'll look at ways to produce the following image. It shows a dark red cylinder inside a kind of cloud of points. The points are randomly selected points on a sphere. The sphere points are lit, with normal vectors and texture coordinates appropriate for the unit sphere. The texture that is used is a topological map of the Earth's surface. The on-line version of this section has an applet that animates the image and allows you to rotate it.



The source code for the program can be found in *VertexArrayDemo.java*.

3.4.1 Java's Data Buffers

The term "buffer" was already overused in OpenGL. Jogl's use of nio buffers makes it even harder to know exactly what type of buffer is being discussed in any particular case. The term buffer ends up meaning, basically, no more than a place where data can be stored—usually a place where data can be stored by one entity and retrieved by another entity.

A Java nio buffer is an object belonging to the class *java.nio.Buffer* or one of its subclasses. The package *java.nio* defines input/output facilities that supplement those in the older *java.io* package. Buffers in this package are used for efficient transfer of data between a Java program and an I/O device such as a hard drive. When used in Jogl, this type of buffer offers the possibility of efficient data transfer between a Java program and a graphics card.

Subclasses of *Buffer* such as *FloatBuffer* and *IntBuffer* represent buffers capable of holding data items of a given primitive type. There is one such subclass for each primitive type except **boolean**.

To make working with nio buffers a little easier, Jogl defines a set of utility methods for creating buffers. They can be found as static methods in class *BufferUtil*, from the package *com.sun.opengl.util*. We will be using *BufferUtil.newFloatBuffer*(n), which creates a buffer that can hold n floating point values, and *BufferUtil.newIntBuffer*(n), which creates a buffer that can hold n integers. (The buffers created by these methods are so-called "direct" buffers, which are required for the Jogl methods that we will be using. It is also possible to create a *Buffer* object that stores its data in a standard Java array, but buffers of that type have the same problems for OpenGL as plain Java arrays. They could be used with some Jogl methods, but not others. We will avoid them.)

A 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.

A buffer differs from an array in that a buffer has an internal position pointer that indicates a "current position" within the buffer. A buffer has ***relative*** *put* and *get* methods that work on the item at the current position, and then advance the position pointer to the next position. That is,

```
buffer.put(x);
```

stores the value of x at the current position, and advances the position pointer to the next element in the buffer. Similarly, *buffer.get*() returns the item at the buffer's current position and advances the pointer.

There are also methods for transferring an entire array of values to or from a buffer. These methods can be much more efficient than transferring values one at a time. For example,

```
buffer.put(data, start, count);
```

copies *count* values from an array, *data*, starting from position *start* in the array. The data is stored in the *buffer*, starting at the buffer's current position. The buffer's position pointer is moved to the first index following the block of transferred data.

The method *buffer.position*(i) can be used to set the buffer's current position pointer to i. To return the position pointer to the start of the buffer, you can use *buffer.position*(0) or *buffer.rewind*().

As an example, suppose that we want to store the vertices of a triangle in a *FloatBuffer*, *buf*. Say the vertices are (1.5,0,0), (0,1.5,0), and (0.5,0.5,−1). Here are three ways to do it:

```
(1) Absolute put, starting at position 0:

      buf.put(0,1.5f);  buf.put(1,0);     buf.put(2,0);   // first vertex
      buf.put(3,0);     buf.put(4,1.5f);  buf.put(5,0);   // second vertex
      buf.put(6,0.5f);  buf.put(7,0.5f);  buf.put(8,-1);  // third vertex

(2) Relative put; starts at current position; no need to specify indices:

      buf.put(1.5f);  buf.put(0);     buf.put(0);   // first vertex
      buf.put(0);     buf.put(1.5f);  buf.put(0);   // second vertex
      buf.put(0.5f);  buf.put(0.5f);  buf.put(-1);  // third vertex

(3) Bulk put, copying the data from an array:

      float[] vert = {  1.5,0,0,  0,1.5,0,  0.5,0.5,-1  };
      buf.put(vert,0,9);
```

You should note that the absolute *put* methods used in case (1) do **not** move the buffer's internal pointer. The relative *puts* in cases (2) and (3) advance the pointer nine positions in each case—and when you use them, you might need to reset the buffer's position pointer later.

3.4.2 Drawing With Vertex Arrays

As an alternative to using glBegin/glEnd, you can place your data into nio buffers and let OpenGL read the data from the buffers as it renders the primitives. (In C or C++, you would use arrays rather than buffers.) We start with a version of this technique that has been available since OpenGL 1.1, and so can be used on all current systems. The technique is referred to as using ***vertex arrays***, although nio buffers, not arrays, are used in Jogl, and you can store other data such as normal vectors in the buffers, not just vertices.

To use vertex arrays, you must store vertices in a buffer, and you have to tell OpenGL that you are using that buffer with the following method from the class *GL*:

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

The *size* here is the number of coordinates per vertex, which can be 2, 3, or 4. (You have to provide the same number of coordinates for each vertex.) The *stride* is usually 0, meaning that the data values are stored in consecutive locations in the buffer; if that is not the case, then *stride* gives the distance **in bytes** between the location of one value and the location of the next value. The *type* is a constant that tells the data type of each of the numbers in the buffer. The possible values are *GL.GL_FLOAT*, *GL.GL_INT*, and *GL.GL_DOUBLE*. The constant that you provide here must match the type of the buffer. For example, if you want to use **float** values for the vertex coordinates, specify *GL.GL_FLOAT* as the *type* and use a buffer of type *FloatBuffer*. The vertex data is assumed to start at the buffer's current position. Usually, that will be at the beginning of the buffer. (In particular, note that you might need to set the buffer's position pointer to the appropriate value by calling *buffer.position*() before calling *gl.glVertexPointer*(); this will certainly be true if you use relative *put* commands to put data into the buffer.)

In addition to calling *glVertexPointer*, you must enable the use of the buffer by calling

```
gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
```

Use *gl.glDisableClientState*(*GL.GL_VERTEX_ARRAY*) to disable the use of the array. OpenGL ignores the vertex pointer except when this state is enabled.

Finally, in order to actually use the vertex data from the buffer, use the following method from class *GL*:

```
void glDrawArrays(int mode, int first, int count);
```

This method call corresponds to one use of glBegin/glEnd. The *mode* tells which primitive type is being drawn, such as *GL.GL_POLYGON* or *GL.GL_TRIANGLE_STRIP*. The same ten primitive types that can be used with *glBegin* can be used here. *first* is the index in the buffer of the first vertex that is to used for drawing the primitive. Note that the position is given in terms of vertex number, not number of floating point values. The *count* is the number of vertices to be used, just as if *glVertex* were called *count* times.

Let's see how this could be used to draw the rectangle in the xy-plane with corners at (−1,−1) and (1,1). We need a variable of type *FloatBuffer*, which would probably be an instance variable:

```
private FloatBuffer rectVertices;
```

The buffer itself has to be allocated and filled with data, perhaps in the *init*() method:

```
rectVertices = BufferUtil.newFloatBuffer(8);
float[] data = {  -1,-1,  1,-1,  1,1,  -1,1  };
rectVertices.put(data,0,8);
rectVertices.rewind();
```

The last line, which moves the buffer's position pointer back to zero, is essential, since positions of data in the buffer are specified relative to that pointer. With this setup, we can draw the rectangle in the *display*() method like this:

```
gl.glVertexPointer(2, GL.GL_FLOAT, 0, rectVertices);
gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
gl.glNormal3f(0,0,1);
gl.glDrawArrays(GL.GL_POLYGON, 0, 4);  // Generate the polygon using 4 vertices.
gl.glDisableClientState(GL.GL_VERTEX_ARRAY);
```

The "2" that is used as the first parameter to *glVertexPointer* says that each vertex consists of two floating point values, giving the coordinates of a vertex that lies in the xy-plane, just as for *glVertex2f*.

In this example, one normal vector will work for all the vertices, and there are no texture coordinates. In general, we will need a different normal vector and possibly texture coordinates for each vertex. We just have to store these data in their own arrays, in much the same way that we did the vertex data.

The following methods are used in the same way as *glVertexPointer*, to specify the buffers that hold normal vector and texture coordinate data:

```
public void glNormalPointer(int type, int stride, Buffer buffer)
public void glTexCoordPointer(int size, int type, int stride, Buffer buffer)
```

Note that *glNormalPointer* does not have a *size* parameter. For normal vectors, you must always give three numbers for each vector, so the size would always be 3 in any case.

To tell OpenGL to use the data from the buffers, you have to enable the appropriate client state, using the method calls

```
gl.glEnableClientState(GL.GL_NORMAL_ARRAY);
gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
```

When you call *glDrawArrays*, if *GL_NORMAL_ARRAY* is enabled, then normal vectors will be retrieved from the buffer specified by *glNormalPointer*. One normal vector will be retrieved for each vertex, and the normals in the normal buffer must correspond one-to-one with the vertices in the vertex buffer. The texture coordinate buffer works in the same way.

* * *

For a more extended example, we will look at how to draw the cylinder and sphere in the picture at the beginning of this section.

The sphere consists of thousands of randomly generated points on the unit sphere (that is, the sphere of radius 1 centered at the origin), with appropriate texture coordinates and normal vectors. The entire set of points can be drawn with one call to *glDrawArrays*, using *GL_POINTS* as the primitive type. (This example emphasizes the fact that points are essentially free-floating vertices that can have their own normals and texture coordinates.) We need buffers to hold vertices, normals, and texture coordinates. However, in this particular case, the normal to the unit sphere at a given point has the same coordinates as the point itself, so in fact I use the same set of data for the normals as for the vertices, stored in the same buffer. Here is the method that is used to create the data, where *sphereVertices* and *sphereTexCoords* are *FloatBuffers* and *spherePointCount* is the number of points to be generated:

```
private void createPointSphere() {
    spherePointCloud = BufferUtil.newFloatBuffer(spherePointCount*3);
    sphereTexCoords = BufferUtil.newFloatBuffer(spherePointCount*2);
    for (int i = 0; i < spherePointCount; i++) {
       double s = Math.random();
       double t = Math.random();
       sphereTexCoords.put(2*i,(float)s);
       sphereTexCoords.put(2*i+1,(float)t);
       double u = s * Math.PI * 2;
       double z = t * 2 - 1;
       double r = Math.sqrt(1-z*z);
       double x = r * Math.cos(u);
       double y = r * Math.sin(u);
       spherePointCloud.put(3*i,(float)x);
       spherePointCloud.put(3*i+1,(float)y);
       spherePointCloud.put(3*i+2,(float)z);
    }
}
```

You don't need to understand the math, but note that each vertex requires three floats while each set of texture coordinates requires two. Here, I've used indexed *put* commands to store the data into the buffers. In the method for creating the data for the cylinder, I used relative *put*, which has the advantage that I don't need to computer the correct index for each number that I put in the buffer. I don't present that method here, but you can find it in the *source code*.

Once the data has been stored in the buffers, it's easy to draw the sphere. We have to set up pointers to the data, enable the appropriate client states, and call *glDrawArrays* to generate the points from the data. In this case, the normals and the vertices are identical, and the data for them are taken from the same buffer.

```
// Tell OpenGL where to find the data:

gl.glVertexPointer(3, GL.GL_FLOAT, 0, sphereVertices);
gl.glNormalPointer(GL.GL_FLOAT, 0, sphereVertices);
gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, sphereTexCoords);

// Tell OpenGL which arrays are being used:

gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL.GL_NORMAL_ARRAY);
gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);

// Turn on texturing, and draw the sphere:

sphereTexture.enable();
gl.glDrawArrays(GL.GL_POINTS, 0, spherePointCount); // Generate the points!

// At this point, texturing and client states could be disabled.
```

Things are just a little more complicated for the cylinder, since it requires three calls to *glDrawArrays*, one for the side of the cylinder, one for the top, and one for the bottom. We can, however, store the vertices for all three parts in the same buffer, as long as we remember the starting point for each part. The data is stored in a *FloatBuffer* named *cylinderPoints*. The vertices for the side of the cylinder start at position 0, for the top at position *cylinderTopStart*, and for the bottom at *cylinderBottomStart*. (That is, the side used *cylinderTopStart* vertices, and the first vertex for the top is at index *cylinderTopStart* in the list of vertices.) The normals for the side of the cylinder are stored in a *FloatBuffer* named *cylinderSideNormals*. For the top and bottom, the normal vector can be set by a single call to *glNormal3f*, since the same normal is used for each vertex. This means that *GL_NORMAL_ARRAY* has to be enabled while drawing the side but not while drawing the top and bottom of the cylinder. Putting all this together, the cylinder can be drawn as follows:

```
gl.glVertexPointer(3,GL.GL_FLOAT,0,cylinderPoints);
gl.glNormalPointer(GL.GL_FLOAT,0,cylinderSideNormals);

// Draw the side, using data from the vertex buffer and from the normal buffer.

gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL.GL_NORMAL_ARRAY);
gl.glDrawArrays(GL.GL_QUAD_STRIP, 0, (cylinderVertexCount+1)*2);
gl.glDisableClientState(GL.GL_NORMAL_ARRAY);  // Turn off normal array.
                                               // Leave vertex array enabled.

// Draw the top and bottom, using data from vertex buffer only.

gl.glNormal3f(0,0,1); // Normal for all vertices for the top.
gl.glDrawArrays(GL.GL_TRIANGLE_FAN, cylinderTopStart, cylinderVertexCount+2);
gl.glNormal3f(0,0,-1); // Normal for all vertices for the bottom.
gl.glDrawArrays(GL.GL_TRIANGLE_FAN, cylinderBottomStart, cylinderVertexCount+2);

gl.glDisableClientState(GL.GL_VERTEX_ARRAY);  // Turn off vertex array.
```

In addition to vertex, normal, and texture coordinate arrays, *glDrawArrays* can use several other arrays, including a color array that holds color values and generic vertex attribute arrays that hold data for use in GLSL programs.

3.4.3 Vertex Buffer Objects

Vertex arrays speed up drawing by greatly reducing the number of calls to OpenGL routines. However, the data for the routines still has to be transferred to the graphics card each time it is used, and this can still be a bottleneck on performance. OpenGL Version 1.5 introduced another technique that offers the possibility of reducing and in some cases eliminating this bottleneck. The technique is referred to as ***vertex buffer objects*** or ***VBOs***.

A vertex buffer object is a region of memory managed by OpenGL that can store the data from vertex arrays. Note that these buffers are **not** the same sort of thing as Java nio buffers. When VBOs are used, the data from the Java nio buffers specified by methods such as *glVertexPointer* and *glNormalPointer* is (at least potentially) **copied** into a VBO which can reside in memory inside the graphics card or in system memory that is more easily accessible to the graphics card.

Before using VBOs, you should be sure that the version of OpenGL is 1.5 or higher. It is a good idea to test this and store the answer in a **boolean** variable. In Jogl, you can do this with:

```
version_1_5 = gl.isExtensionAvailable("GL_VERSION_1_5");
```

VBOs are allocated by the *glGenBuffers* method, which creates one or more VBOs and returns an integer ID number for each buffer created. The ID numbers are stored in an array. For example, four VBOs can be created like this:

```
int[] bufferID = new int[4];    // Get 4 buffer IDs.
gl.glGenBuffers(4,bufferID,0);
```

The third parameter is an offset that tells the starting index in the array where the IDs should be stored; it is usually 0 (and is absent in the C API).

Now, most of the OpenGL routines that work with VBOs do not mention a VBO index. Instead, they work with the "current VBO." For use with vertex arrays, the current VBO is set by calling

```
gl.glBindBuffer(GL.GL_ARRAY_BUFFER, vboID);
```

where *vboID* is the ID number of the VBO that is being made current. Later, we'll see another possible value for the first parameter of this method. It's important to understand that this method does nothing except to say, "OK, from now on anything I do with VBOs that is relevant to vertex arrays should be applied to VBO number *vboID*." It is a switch that directs future commands to a particular VBO. You can also call *gl.glBindBuffer*(*GL.GL_ARRAY_BUFFER*,0) to direct commands away from VBOs altogether; this can be important because the meaning of some commands changes when they are used with VBOs.

To specify the data that is to be stored in a VBO, use the *glBufferData* method from the *GL* class. Note that this method supplies data for the VBO whose ID has been selected using *glBindBuffer*:

```
public void glBufferData(int target, int size, Buffer data, int usage)
```

The target, for now, should be GL.GL_ARRAY_BUFFER (and in general should match the first parameter in *glBindBuffer*). *size* is the size of the data **in bytes**, and the data is stored in *data*, which is a Java nio *Buffer*. (Note that the data should already be there; even though the data will be used by future commands, this method specifies the data itself, not just the location of the data.) The *usage* parameter is particularly interesting. It is a "hint" that tells

OpenGL how the data will be used. OpenGL will try to store the data in the optimal location for its intended use. For our purposes in this section, the possible values are

- *GL.GL_STATIC_DRAW* – the data is expected to be used many times, without further modification. It would be optimal to store the data in the graphics card, where OpenGL has direct access to it. This is the appropriate usage for this section's *VertexArrayDemo* application.
- *GL.GL_STREAM_DRAW* – the data will be used once or at most a few times. It is not so important to save it on the graphics card.
- *GL.GL_DYNAMIC_DRAW* – the data will be changed many times. The data is ideally stored in memory that can be quickly accessed by both the CPU and the graphics card. This usage would be appropriate for an animation in which the vertex data can change from one frame to the next.

The *VertexArrayDemo* application actually uses VBOs when the OpenGL version is 1.5 or higher. VBOs are used to store the vertex, normal, and texture coordinate data for the sphere and cylinder. Four VBOs are used. Here is the code that creates the VBO's and provides them with data:

```
if (version_1_5) {
    int[] bufferID = new int[4];    // Get 4 buffer IDs for the data.
    gl.glGenBuffers(4,bufferID,0);

    spherePointBufferID = bufferID[0];    // VBO for sphere vertices/normals.
    sphereTexCoordID = bufferID[1];       // VBO for sphere texture coords.
    cylinderPointBufferID = bufferID[2];  // VBO for cylinder vertices.
    cylinderNormalBufferID = bufferID[3]; // VBO for cylinder normals.

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, spherePointBufferID);
    gl.glBufferData(GL.GL_ARRAY_BUFFER, spherePointCount*3*4,
                                          sphereVertices, GL.GL_STATIC_DRAW);

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, sphereTexCoordID);
    gl.glBufferData(GL.GL_ARRAY_BUFFER, spherePointCount*2*4,
                                        sphereTexCoords, GL.GL_STATIC_DRAW);

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, cylinderPointBufferID);
    gl.glBufferData(GL.GL_ARRAY_BUFFER, ((cylinderVertexCount+1)*4+2)*3*4,
                                         cylinderPoints, GL.GL_STATIC_DRAW);

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, cylinderNormalBufferID);
    gl.glBufferData(GL.GL_ARRAY_BUFFER, (cylinderVertexCount+1)*2*3*4,
                                   cylinderSideNormals, GL.GL_STATIC_DRAW);

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, 0);  // Leave no buffer ID bound.
}
```

The *if* statement tests whether the version of OpenGL is high enough to support vertex buffer objects, using the variable *version_1_5* which was discussed above. Note how *glBindBuffer* is used to select the VBO to which the following *glBufferData* method will apply. Also, note that the size specified in the second parameter to *glBufferData* is given in bytes. Since a **float** value takes up four bytes, the size is obtained by multiplying the number of floats by 4.

* * *

Of course, we also have to *use* the data from the VBOs! The VBOs are holding vertex array data. The *glDrawArrays* and *glEnableClientState* commands are used in exactly the same way whether or not we are using VBOs. However, the command for telling OpenGL where to find that data is a little different when using VBOs. When the data is stored in VBOs, alternative forms of the *glVertexPointer*, *glNormalPointer*, and *glTexCoordPointer* methods are used:

```
public void glVertexPointer(int size, int type, int stride, long vboOffset)
public void glNormalPointer(int type, int stride, long vboOffset)
public void glTexCoordPointer(int size, int type, int stride, long vboOffset)
```

The difference is the last parameter, which is now an integer instead of a *Buffer*. (In the C API, there is only one version of each command, with a pointer as the fourth parameter, but that parameter is interpreted differently depending on whether VBOs are being used or not.) The *vboOffset* gives the starting position of the data within the VBO. This offset is given as the number **of bytes** from the beginning of the VBO; the value is often zero. The VBO in question is not mentioned. As usual, it is the VBO that has been most recently specified by a call to *gl.glBindBuffer*(*GL.GL_ARRAY_BUFFER*,*vboID*).

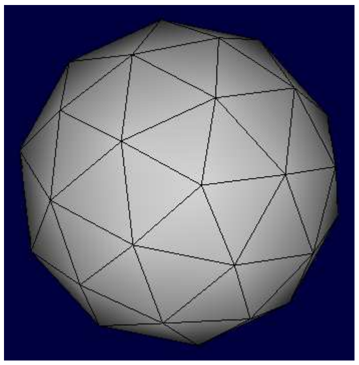
We can now look at the complete sphere-drawing method from *VertexArrayDemo*, which uses VBOs when the OpenGL version is 1.5 or higher. Note that this only makes a difference when telling OpenGL the location of the data. Once that's done, the code for drawing the sphere is identical whether VBOs are used or not:

```
private void drawPointSphere(GL gl, int pointCt) {
    if (version_1_5) {
          // Use glBindBuffer to say what VBO to work on, then use
          // glVertexPointer to set the position where the data starts
          // in the buffer (in this case, at the start of the buffer).
        gl.glBindBuffer(GL.GL_ARRAY_BUFFER, spherePointBufferID);
        gl.glVertexPointer(3, GL.GL_FLOAT, 0, 0);
          // Use the same buffer for the normal vectors.
        gl.glNormalPointer(GL.GL_FLOAT, 0, 0);
          // Now, set up the texture coordinate pointer in the same way.
        gl.glBindBuffer(GL.GL_ARRAY_BUFFER, sphereTexCoordID);
        gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, 0);
        gl.glBindBuffer(GL.GL_ARRAY_BUFFER,0); // Leave no VBO bound.
    }
    else {
           // Use glVertexPointer, etc., to set the buffers from which
           // the various kinds of data will be read.
        gl.glVertexPointer(3,GL.GL_FLOAT,0,sphereVertices);
        gl.glNormalPointer(GL.GL_FLOAT,0,sphereVertices);
        gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, sphereTexCoords);
    }
    gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL.GL_NORMAL_ARRAY);
    gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
    sphereTexture.enable();
    gl.glDrawArrays(GL.GL_POINTS, 0, spherePointCount); // Generate the points!
    gl.glDisableClientState(GL.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL.GL_NORMAL_ARRAY);
    gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
    sphereTexture.disable();
 }
```

Obviously, vertex buffer arrays are non-trivial to use. The reward is the possibility of better performance when rendering complex scenes.

3.4.4 Drawing with Array Indices

The *glDrawArrays* method is great for drawing primitives given as a list of vertex coordinates. However, it can't be used for data in indexed face set (IFS) format. For that, there is the *glDrawElements* method. For an indexed face set, a face is specified by a sequence of integers representing indices into a list of vertices. *glDrawElements* can work directly with data in this format. To give us an example to work with, we consider an "icosphere," a polygonal approximation of a sphere that is obtained by subdividing the faces of an icosahedron. You get a better approximation by subdividing the faces more times. Here is a picture of an icosphere created by one subdivision:



You can find an applet that draws icospheres in the on-line version of these notes. The source code for the program can be found in *IcosphereIFS.java*.

For use with *glDrawElements*, you can set up buffers to hold vertex coordinates, normal coordinates, and texture coordinates, exactly as you would for *glDrawArrays* (including the use of vertex buffer objects, if desired). Now, however, the elements in the arrays are stored in arbitrary order, not the order in which they will be used to generate primitives. The order needed for generating primitives will be specified by a separate list of indices into the arrays. Note, however, that the order of elements in the various arrays must correspond. That is, the first vector in the normal array and the first set of texture coordinates in the texture array must correspond to the first vertex in the vertex array, the second item in the normal and texture arrays must correspond to the second vertex, and so on.

For the icosphere example, the geometry consists of a large nubmer of triangles, and the whole thing can be drawn with a single use of the *GL_TRIANGLES* primitive. Let's assume that the vertex coordinates have been stored in a Java nio FloatBuffer named *icosphereVertexBuffer*. Because the points are points on the unit sphere, we can use the same set of coordinates as unit normal vectors. And let's say that the face data (the list of vertex indices for each triangle) is in an IntBuffer named *icosphereIndexBuffer*. We could actually draw the icosphere directly using glBegin/glEnd: Assuming *indexCount* is the number of integers in *icosphereIndexBuffer*, the following code would work:

```
gl.glBegin(GL.GL_TRIANGLES);
   for (int i = 0; i < indexCount; i++) {
      int vertexIndex = icosphereIndexBuffer.get(i); // Index of i-th vertex.
      float vx = icosphereVertexBuffer.get(vertexIndex); // Get vertex coords.
```

```
        float vy = icosphereVertexBuffer.get(vertexIndex);
        float vz = icosphereVertexBuffer.get(vertexIndex);
        gl.glNormal3f(vx,vy,vz);  // Use vertex coords as normal vector.
        gl.glVertex3f(vx,vy,vz);  // Generate the i-th vertex.
    }
gl.glEnd();
```

But *glDrawElements* is meant to replace precisely this sort of code. Before using it, we must use *glVertexPointer* and *glNormalPointer* to set up a vertex pointer and normal pointer, as for *glDrawArrays.* How we do this depends on whether we are using vertex buffer objects, but it is done exactly as above.

VBOs can also be used to store the face index data for use with *glDrawElements*, and how we use *glDrawElements* also depends on whether or not a vertex buffer object is used. Unfortunately, the use of VBOs for this purpose is not exactly parallel to their use for vertex and normal data. If we do **not** use a VBO for the face data, then the face data is passed directly to glDrawArrays, as follows:

```
gl.glDrawElements(GL.GL_TRIANGLES, indexCount,
                                GL.GL_UNSIGNED_INT, icosphereIndexBuffer);
```

The first parameter is the type of primitive that is being drawn. The second is the number of vertices that will be generated. The third is the type of data in the buffer. And the fourth is the nio *Buffer* that holds the face data. The face data consists of one integer for each vertex, giving an index into the data that has already been set up by *glVertexPointer* and related methods. Note that there is no indication of the position within the buffer where the data begins. The data begins at the buffer's current internal position pointer. If necessary, you can set that pointer by calling the buffer's *position* method before calling *glDrawElements.* (This is one place where are are forced to work with the internal buffer pointer, if you want to store data for more than one call to *glDrawElements* in the same buffer.)

Now, let's look at what happens when a VBO is used to hold the face data. In this case, the data must be loaded into the VBO before it can be used. For this application, the first parameter to *glBindBuffer* and *glBufferData* must be *GL.GL_ELEMENT_ARRAY_BUFFER*:

```
gl.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, icopsphereIndexID);
gl.glBufferData(GL.GL_ELEMENT_ARRAY_BUFFER, indexCount*4,
                             icosphereIndexBuffer, GL.GL_STATIC_DRAW);
```

At the time when *glDrawElements* is called, the same buffer ID must be bound, and the fourth parameter to *glDrawElements* is replaced by an integer giving the position of the data within the VBO, given as the number of bytes from the start of the VBO:

```
gl.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, icopsphereIndexID);
gl.glDrawElements(GL.GL_TRIANGLES, indexCount, GL.GL_UNSIGNED_INT, 0);
```

See the source code, *IcosphereIFS.java*, to see how all this is used in the context of a full program.

* * *

This has been an admittedly short introduction to *glDrawElements*, but hopefully it is similar enough to *glDrawArrays* that the information given here is enough to get you started using it. Let's look at another, simpler, example that shows how you might use *glDrawElements* to draw an IFS representing a simple polyhedron. We will use the data for the same pyramid that was used as an example in the previous section:

```
float[][] vertexList = {  {1,0,1}, {1,0,-1}, {-1,0,-1}, {-1,0,1}, {0,1,0}  };
int[][] faceList   = {  {4,3,0},  {4,0,1},  {4,1,2},  {4,2,3},  {0,3,2,1}  };
```

Since this is such a small object, we will not attempt to use vertex buffer objects. We will draw one face at a time, using a single normal vector for the face; I will specify the appropriate normal directly. Here is the code for drawing the pyramid (which would actually be scattered in several parts of a program):

```
// Declare variables to hold the data, probably as instance variables:

FloatBuffer vertexBuffer;
IntBuffer faceBuffer;

// Create the Java nio buffers, and fill them with data, perhaps in init():

vertexBuffer = BufferUtil.newFloatBuffer(15);
faceBuffer = BufferUtil.newIntBuffer(16);

float[] vertexList = {  1,0,1, 1,0,-1, -1,0,-1, -1,0,1, 0,1,0  };
int[] faceList   = {  4,3,0,  4,0,1,  4,1,2,  4,2,3,  0,3,2,1  };

vertexBuffer.put(vertexList, 0, 15);
vertexBuffer.rewind();

faceBuffer.put(faceList, 0, 16); // No need to rewind; position is set later.

// Set up for using a vertex array, in init() or display():

gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
gl.glVertexPointer(3, GL.GL_FLOAT, 0, vertexBuffer);

// Do the actual drawing, one face at a time.  Set the position in
// the faceBuffer to indicate the start of the data in each case:

faceBuffer.position(0);
gl.glNormal3f(0, 0.707f, 0.707f);
gl.glDrawElements(GL.GL_POLYGON, 3, GL.GL_UNSIGNED_INT, faceBuffer);

faceBuffer.position(3);
gl.glNormal3f(0.707f, 0.707f, 0);
gl.glDrawElements(GL.GL_POLYGON, 3, GL.GL_UNSIGNED_INT, faceBuffer);

faceBuffer.position(6);
gl.glNormal3f(0, 0.707f, -0.707f);
gl.glDrawElements(GL.GL_POLYGON, 3, GL.GL_UNSIGNED_INT, faceBuffer);

faceBuffer.position(9);
gl.glNormal3f(-0.707f, 0.707f, 0);
gl.glDrawElements(GL.GL_POLYGON, 3, GL.GL_UNSIGNED_INT, faceBuffer);

faceBuffer.position(12);
gl.glNormal3f(0, -1, 0);
gl.glDrawElements(GL.GL_POLYGON, 4, GL.GL_UNSIGNED_INT, faceBuffer);
```

* * *

This has not been a complete survey of the OpenGL routines that can be used for drawing primitives. For example, there is a *glDrawRangeElements* routine, introduced in OpenGL version 1.2, that is similar to *glDrawElements* but can be more efficient. And *glMultiDrawArrays*, from version 1.4, can be used to do the job of multiple calls to *glDrawArrays* at once. Interested readers can consult an OpenGL reference.

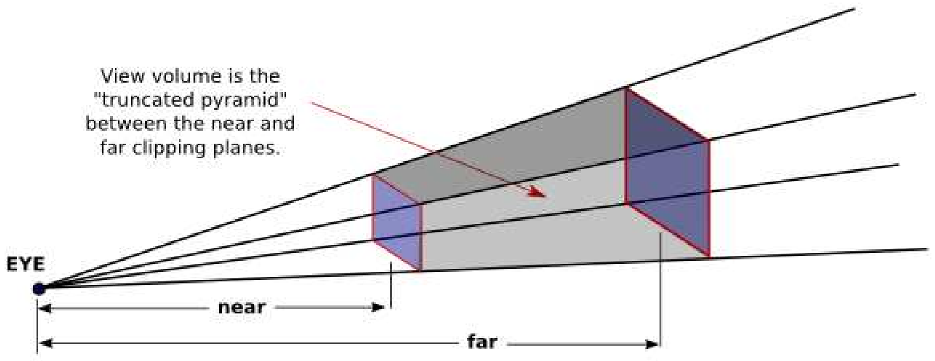
3.5 Viewing and Projection

This chapter on geometry finishes with a more complete discussion of projection and viewing transformations.

3.5.1 Perspective Projection

There are two general types of projection, ***perspective projection*** and ***orthographic projection***. Perspective projection gives a realistic view. 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 a 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, 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 buffer 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 are shown in the image that OpenGL produces. That range of distances is specified by two values, ***near*** and ***far***. 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***:


View volume is the "truncated pyramid" between the near and far clipping planes.
EYE
near
far

The view volume is bounded by six planes—the sides, top, and bottom of the pyramid. These planes are called ***clipping planes*** because anything that lies on the wrong side of each plane is clipped away.

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***, and a perspective transformation can be set up with the *glFrustum* command

```
gl.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.

(These are “eye coordinates” in OpenGL.) So, the near clipping plane is at $z = -near$, and the far clipping plane is at $z = -far$. 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 ($xmin$,$ymax$,$-near$). Note that although *xmin* is usually equal to the negative of *xmax* and *ymin* is usually equal to the negative of *ymax*, 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.

When the *glFrustum* method is used to set up the projection transform, the matrix mode should be set to *GL_PROJECTION*. Furthermore, the identity matrix should be loaded before calling *glFrustum* (since *glFrustum* modifies the existing projection matrix rather than replacing it, and you don’t even want to try to think about what would happen if you combine several projection matrices into one). So, a use of *glFrustum* generally looks like this, leaving the matrix mode set to *GL_MODELVIEW* at the end:

```
gl.glMatrixMode(GL.GL_PROJECTION);
gl.glLoadIdentity();
gl.glFrustum(xmin,xmax,ymin,ymax,near,far);
gl.glMatrixMode(GL.GL_MODELVIEW);
```

This might be used in the *init*() method, at the beginning of the *display*() method, or possibly in the *reshape*() method, if you want to take the aspect ratio of the viewport into account.

The *glFrustum* method is not particularly easy to use. The GLU library includes the method *gluPerspective* as an easier way to set up a perspective projection. If *glu* is an object of type *GLU* then

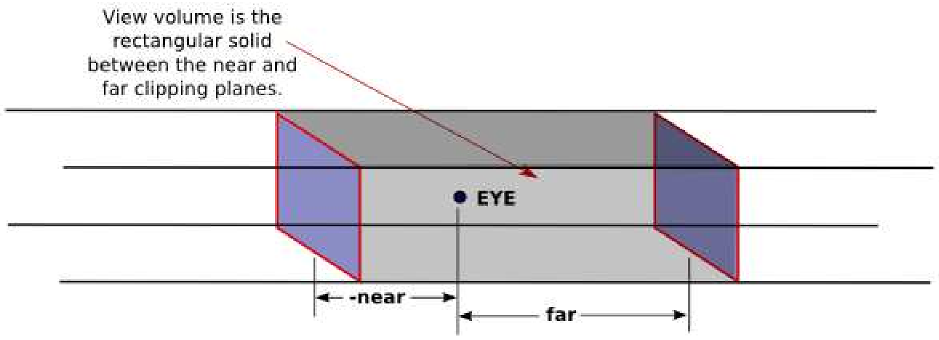
```
glu.gluPerspective(fieldOfViewAngle, aspect, near, far);
```

can be used instead of *glFrustum*. The *fieldOfViewAngle* is the vertical angle between the top of the view volume pyramid and the bottom. The *aspect* is the aspect ratio of the view, that is, the width of the pyramid at a given distance from the eye, divided by the height at the same distance. The value of *aspect* should generally be set to the aspect ratio of the viewport. The *near* and *far* parameters have the same meaning as for *glFrustum*.

3.5.2 Orthographic Projection

Orthographic projections are comparatively easy to understand: A 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 are visible in the image. In fact, it’s not really clear what it means to say that there is a viewer in the case of orthographic projection. Orthographic projections are still useful, however, especially in interactive modeling programs where it is useful to see true sizes and angles, unmodified by perspective.

Nevertheless, in OpenGL there is a viewer, which 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 it is in this illustration:


View volume is the rectangular solid between the near and far clipping planes.
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 generally called like this:

```
gl.glMatrixMode(GL.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrtho(xmin,xmax,ymin,ymax,near,far);
gl.glMatrixMode(GL.GL_MODELVIEW);
```

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!

3.5.3 The Viewing Transform

To determine what a viewer will actually see in a 3D world, you have to do more than specify the projection. You also have to position the viewer in the world. That is done with the viewing transformation. Remember that the projection transformation is specified in eye coordinates, which have the viewer at the origin, facing down the negative direction of the z-axis. The viewing transformation says where the viewer really is, in terms of the world coordinate system, and where the viewer is really facing. The projection transformation can be compared to choosing which camera and lens to use to take a picture. The viewing transformation places the camera in the world and points it.

Recall that OpenGL has no viewing transformation as such. It has a modelview transformation, which combines the viewing transform with the modeling transform. While the viewing transformation moves the viewer, the modeling transformation moves the objects in the 3D world. The point here is that these are really equivalent operations, if not logically, then at least in terms of the image that is produced when the camera finally snaps a picture.

Suppose, for example, that we would like to move the camera from its default location at the origin back along the positive z-axis from the origin 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 camera ends up in exactly the same position relative to the objects. It follows that both operations are implemented by the same OpenGL command, *gl.glTranslatef* (0,0,-20). More generally, applying any transformation to the camera is equivalent to applying the ***inverse***, or opposite, of the transformation to the world. Rotating the camera to left has the same effect as rotating the world to the right. This even works for scaling: Imagine yourself sitting inside the camera looking out. If the camera *shrinks* (and you along with it) it will look to you like the world outside is *growing*—and what you see doesn't tell which is really happening. Suppose that we use the commands

```
gl.glRotatef(90,0,1,0);
gl.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. An object that was originally at the origin ends up on the negative z-axis; the object is then directly in front of the viewer, at a distance of ten units. If we consider the same transformation as a viewing transform, the effect on the viewer is the inverse of the effect on the world. That is, the transform commands first rotate the viewer **−90** degrees about the y-axis, then translate the viewer 10 units in the **negative** x-direction. This leaves the viewer on the negative x-axis at (−10,0,0), looking at the origin. An object at the origin will then be directly in front of the viewer, at a distance of 10 units. Under both interpretations of the transformation, the relationship of the viewer to the object is the same in the end.

Since this can be confusing, the GLU library provides a convenient method for setting up the viewing transformation:

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

This method places the camera at the point (*eyeX*,*eyeY*,*eyeZ*), looking in the direction of the point (*refX*,*refY*,*refZ*). The camera is oriented so that the vector (*upX*,*upY*,*upZ*) points upwards in the camera's view. This method is meant to be called at the beginning of the *display*() method to establish the viewing transformation, and any further transformations that are applied after that are considered to be part of the modeling transformation.

* * *

The *Camera* class that I wrote for the *glutil* package combines the functions of the projection and viewing transformations. For an object of type *Camera*, the method

```
camera.apply(gl);
```

is meant to be called at the beginning of the *display* method to set both the projection and the view. The viewing transform will be established with a call to *glu.gluLookAt*, and the projection will be set with a call to either *glOrtho* or *glFrustum*, depending on whether the camera is set to use orthographic or perspective projection. The parameters that will be used in these methods must be set before the call to *camera.apply* by calling other methods in the *Camera* object. The method

```
camera.setView(eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ );
```

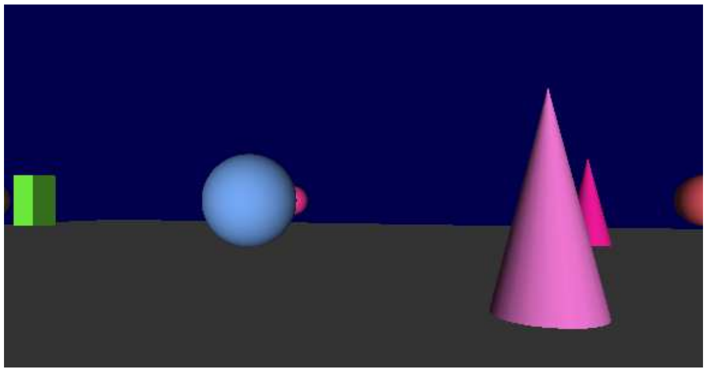
sets the parameters that will be used in a call to *glu.gluLookAt*. In the default settings, the eye is at (0,0,30), the reference point is the origin (0,0,0), and the up vector is the y-axis. The method

```
camera.setLimits(xmin,xmax,ymin,ymax,zmin,zmax)
```

is used to specify the projection. The parameters do not correspond directly to the parameters to *glOrtho* or *glFrustum*. They are specified relative to a coordinate system in which the reference point (*refX*,*refY*,*refZ*) has been moved to the origin, the up vector (*upX*,*upY*,*upZ*) has been rotated onto the positive y-axis, and the viewer has been placed on the positive z-axis, at a distance from the origin equal to the distance between (*eyeX*,*eyeY*,*eyeZ*) and (*refX*,*refY*,*refZ*). (These are almost standard eye coordinates, except that the viewer has been moved some distance backwards along the positive z-axis.) In these coordinates, *zmin* and *zmax* specify the minimum and maximum z-values for the view volume, and *xmin*, *xmax*, *ymin*, and *ymax* specify the left-to-right and bottom-to-top limits of the view volume on the xy-plane, that is, at $z = 0$. (The x and y limits might be adjusted, depending on the configuration of the camera, to match the aspect ratio of the viewport.) Basically, you use the *camera.setLimits* command to establish a box around the reference point (*refX*,*refY*,*refZ*) that you would like to be in view.

3.5.4 A Simple Avatar

In all of our sample programs so far, the viewer has stood apart from the world, observing it from a distance. In many applications, such as 3D games, the viewer is a part of the world and gets to move around in it. With the right viewing transformation and the right user controls, this is not hard to implement. The sample program *WalkThroughDemo.java* is a simple example where the world consists of some random shapes scattered around a plane, and the user can move among them using the keyboard's arrow keys. You can find an applet version of the program in the on-line version of this section. Here is a snapshot:



The viewer in this program is represented by an object of the class *SimpleAvatar*, which I have added to the *glutil* package. A *SimpleAvatar* represents a point of view that can be rotated and moved. The rotation is about the viewer's vertical axis and is controlled in the applet by the left and right arrow keys. Pressing the left-arrow key rotates the viewer through a positive angle, which the viewer perceives as turning towards the left. Similarly, the right-arrow key rotates the viewer towards the right. The up-arrow and down-arrow keys move the viewer forward or backward in the direction that the viewer is currently facing. The motion is parallel to the xz-plane; the viewer's height above this plane does not change.

In terms of the programming, the viewer is subject to a rotation about the y-axis and a translation. These transformations are applied to the viewer in that order (not the reverse—a rotation following a translation to the point (x,y,z) would move the viewer away from that point). However, this viewing transform is equivalent to applying the inverse transform to the world and is implemented in the code as

```
    gl.glLoadIdentity();
    gl.glRotated(-angle,0,1,0);
    gl.glTranslated(-x,-y,-z);
```

where *angle* is the rotation applied to the viewer and (x,y,z) is the point to which the viewer is translated. This code can be found in the *apply*() method in the *SimpleAvatar* class, and that method is meant to be called in the *display* method, before anything is drawn, to establish the appropriate projection and viewing transformations to show the world as seen from the avatar's point of view.

3.5.5 Viewer Nodes in Scene Graphs

For another example of the same idea, we can return to the idea of scene graphs, which were introduced in Subsection 2.1.5. A scene graph is a data structure that represents the contents of a scene. But if we truly want to make our viewer part of the scene, then there should be a way for the viewer to be part of the data for the scene, that is part of the scene graph. We would like to be able to represent the viewer as a node in a scene graph, in the same way that a cube or sphere can be represented by a node. This would mean that the viewer could be subjected to transformations just like any other node in the scene graph. And it means that a viewer can be part of a complex, hierarchical model. The viewer might be the driver in a moving car or a rider on an amusement park ride.

A scene is a hierarchical structure in which complex objects can be built up out of simpler objects, and transformations can be applied to objects on any level of the hierarchy. The overall transformation that is finally applied to an object consists of the product of all the transformations from the root of the scene graph to the object. If we place a viewer into a scene graph, then the viewer should be subject to transformation in exactly the same way. For example, if the viewer is part of a complex object representing a car, then the viewer should be subject to exactly the same transformation as the car and this should allow the viewer to turn and move along with the car.

However, the viewer is not quite the same as other objects in the scene graph. First of all, the viewer is not a visible object. It could be "attached" to a visible object, but the viewer we are talking about is really a point of view, represented by a projection and viewing transformation. A second point, which is crucial, is that the viewer's projection and viewing transformation have to be established before anything is drawn. This means that we can't simply traverse the scene graph and implement the viewer node when we come to it—the viewer node has to be applied before we even start traversing the scene graph. And while there can be several viewer nodes in a scene graph, there can only be one view at a time. There has to be one **active** viewer whose view is shown in the rendered image. Of course, it's possible to switch from one viewer to another and to redraw the image from the new point of view.

The package *simplescenegraph3d* contains a very simple implementation of scene graphs for 3D worlds. One of the classes in this package is *AvatarNode*, which represents exactly the type of viewer node that I have been discussing. An *AvatarNode* can be added to a scene graph and transformed just like any other node. It can be part of a complex object, and it will be carried along with that object when the object is transformed.

Remember that to apply a transformation to a viewer, you have to apply the inverse of that transformation to the world. When the viewer is represented by an *AvatarNode* in a scene graph, the transformation that we want to apply to the viewer is the product of all the transformations applied to nodes along a path from the root of the scene graph to the *AvatarNode*. To apply the inverse of this transformation, we need to apply the inverses of these transformations, in the opposite order. To do this, we can start at the *AvatarNode* and walk **up** the path in the scene graph, from child node to parent node until we reach the top. Along the way, we apply the inverse of the transformation for each node. To make it possible to navigate a scene graph in this way, each node has a *parent* pointer that points from a child node to its parent node. There is also a method *applyInverseTransform* that applies the inverse of the transform in the node. So, the code in the *AvatarNode* class for setting up the viewing transformation is:

```
SceneNode3D node = this;
while (node != null) {
   node.applyInverseTransform(gl);
   node = node.parent;
}
```

An *AvatarNode* has an *apply* method that should be called at the beginning of the *display* method to set up the projection and viewing transformations that are needed to show the world from the point of view of that avatar. The code for setting up the viewing transformation is in the *apply* method.

In the on-line version of this section, you will find an applet that uses this technique to show you a moving scene from several possible viewpoints. A pop-up menu below the scene allows the user to select one of three possible points of view, including two views that are represented by objects of type *AvatarNode* in a scene graph. (The third view is the familiar global view in which the user can rotate the scene using the mouse.) The source code for the program is *MovingCameraDemo.java*.

Chapter 4

Light and Material

In order to create realistic scenes, we need to simulate the way that lights interacts with objects in the world, and we need to consider how that world is perceived by people's visual systems. OpenGL uses a model of light and vision that is a rather crude approximation for reality. It is far from good enough to fool anyone into thinking that they are looking at reality, or even at a photograph, but when used well, it is good enough to let the viewer understand the 3D world that is being represented.

Much more realism is possible in computer graphics, but the techniques that are required to achieve such realism are not built into standard OpenGL. Some of them can be added to OpenGL using the GL Shading Language, which makes it possible to reprogram parts of the OpenGL processing pipeline. But some of them require more more computational power than is currently available in a desktop computer. The goal of OpenGL is to produce a good-enough approximation of realism in a reasonable amount of time—preferably fast enough for interactive graphics at 30 to 60 frames per second.

This chapter will look at the OpenGL approach to simulating light and materials. So far, we have used only the default setup for lighting, and we have used only simple color for materials. It's time to look at the full range of options and at a bit of the theory behind them.

4.1 Vision and Color

Most people are familiar with some basic facts about the human visual system. They know that the full spectrum of colors can be made from varying amounts red, blue, and green. And they might know that this is because the human eye has three different kinds of "cone cells" for color vision, one kind that sees red light, one that sees green, and one that sees blue. These facts are mostly wrong. Or, at least, they are only approximations.

The detailed physics of light is not part of our story here, but for our purposes, light is electromagnetic radiation, a kind of wave that propagates thorough space. The physical property of light that corresponds to the perception of color is ***wavelength***. Electromagnetic radiation that can be detected by the human visual system has a wavelength between about 400 and 700 nanometers. All the colors of the rainbow, from red to violet, are found in this range. This is a very narrow band of wavelengths; outside this band are other kinds of electromagnetic radiation including ultraviolet and infrared, just next to the visible band, as well as X-rays, gamma rays, microwave, and radio waves.

To fully describe a light source, we have to say how much of each visible wavelength it contains. Light from the sun is a combination of all of the visible wavelengths of light (as well as wavelengths outside the visible range), with a characteristic amount of each wavelength. Light from other sources will have a different distribution of wavelengths. When light from some source strikes a surface, some of it is absorbed and some is reflected (and some might pass into and through the object). The appearance of an object depends on the nature of the light that illuminates it and on how the surface of the object interacts with different wavelengths of light. An object that reflects most of the red light that hits it and absorbs most of the other colors will appear red—if the light that illuminates it includes some red for it to reflect.

The visual system in most people does indeed have three kinds of cone cells that are used for color vision (as well as "rod cells" that do not see in color). However, the three types of cone cell do not detect exclusively red, green, and blue light. Each type of cell responds to a range of wavelengths, and the wavelength bands for the three types of cell have a large overlap. It is true that red, green, and blue light can be combined to duplicate many of the colors that can be seen. That is, to the extent that our experience of a color boils down to a certain level of excitation of each of the three types of cone cells, it is possible in many cases to find a combination of red, green, and blue that will produce the same levels of excitation.

Many but not all. "Red", "green", and "blue" here are inexact terms, but they can be taken to mean three wavelengths or combination of wavelengths that are used as ***primary colors***. But no matter what three primary colors are used, it is impossible to duplicate all visible colors with combinations of the three primary colors. Display devices such as computer screens and projectors do, in fact, produce color by combining three primary colors. The range of colors that can be shown by a given device is called the ***color gamut*** of that device. No device has a color gamut that includes all possible colors.

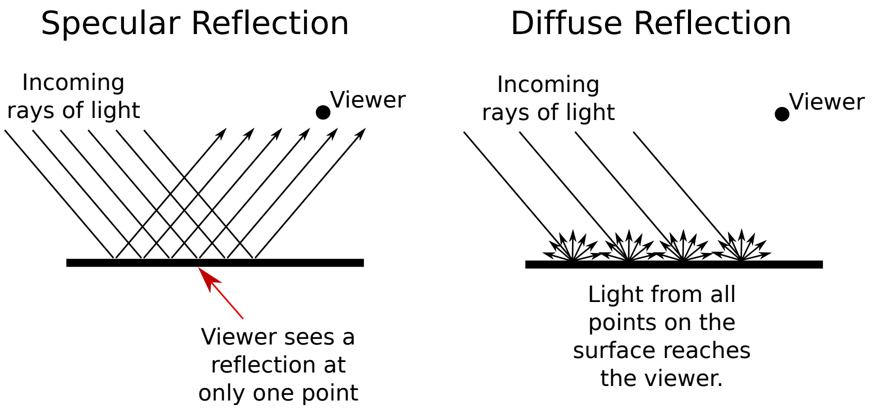
We have been discussing an RGB (red/green/blue) color system, which is appropriate for devices that make color by combining light of different wavelengths. Not all devices do this. A printer, for example, combines differently colored dyes or inks, and the light that will produce the perceived color has to come from an outside source. In this case, the color depends on which colors are absorbed by the ink and which are reflected by the inks. For example, a yellow ink absorbs blue light and reflects red and green. A magenta ink absorbs green while reflecting red and blue. A combination of yellow ink and magenta ink will absorb both blue and green to some extent, leaving mostly red light to be reflected. So in this system, red is a combination of yellow and magenta. By adding cyan ink to the yellow and magenta, a wide range of colors can be produced. Cyan, magenta, and yellow make up the CMY color system. This is a "subtractive" color system, since each color of ink subtracts some wavelengths from the light that strikes the ink. In practice, black ink is often added to the mix, giving the CMYK color system, where the "K" stands for black. The color gamut of a CMYK printer is different from and smaller than the color gamut of a computer screen. The color gamut of a printer can be improved by using additional inks, but it is difficult to match the colors from a printer to the colors on the screen (let alone to those in real life).

RGB and CMYK are only two of many color systems that are used to describe colors. Users of Java are probably familiar with the HSB (hue/saturation/brightness) color system. OpenGL uses RGB colors, so I will not discuss the other possibilities further here.

A substantial number of people, by the way, are color blind to some extent because they lack one or more of the three types of cone cells. For people with the most common type of color blindness, red and green look the same. Programmers should avoid coding essential information only in the colors used, especially when the essential distinction is between red and green.

* * *

In the rest of this chapter, we'll look at the OpenGL approach to light and material. There are a few general ideas about the interaction of light and materials that you need to understand before we begin. When light strikes a surface, some of it will be reflected. Exactly how it reflects depends in a complicated way on the nature of the surface, what I am calling the material properties of the surface. In OpenGL (and in many other computer graphics systems), the complexity is approximated by two general types of reflection, ***specular reflection*** and ***diffuse reflection***.


Specular Reflection
Incoming rays of light
Viewer
Viewer sees a reflection at only one point
Diffuse Reflection
Incoming rays of light
Viewer
Light from all points on the surface reaches the viewer.

In perfect specular ("mirror-like") reflection, an incoming ray of light is reflected from the surface intact. The reflected ray makes the same angle with the surface as the incoming ray. A viewer can see the reflected ray only if the viewer is in the right position, somewhere along the path of the reflected ray. Even if the entire surface is illuminated by the light source, the viewer will only see the reflection of the light source at those points on the surface where the geometry is right. Such reflections are referred to as ***specular highlights***. In practice, we think of a ray of light as being reflected not as a single perfect ray, but as a cone of light, which can be more or less narrow. Specular reflection from a very shiny surface produces very narrow cones of reflected light; specular highlights on such a material are small and sharp. A duller surface will produce wider reflected light cones and bigger, fuzzier specular highlights. In OpenGL, the material property that determines the size and sharpness of specular highlights is called ***shininess***.

In pure diffuse reflection, an incoming ray of light is scattered in all directions equally. A viewer would see see reflected light from all points on the surface, and the surface would appear to be evenly illuminated.

When a light strikes a surface, some wavelengths of light can be absorbed, some can be reflected diffusely, and some can be reflected specularly. The degree to which a material reflects light of different wavelengths is what constitutes the color of the material. We now see that a material can have two different colors—a ***diffuse color*** that tells how the material reflects light diffusely and a ***specular color*** that tells how it reflects light specularly. The diffuse color is the basic color of the object. The specular color determines the color of specular highlights.

In fact, OpenGL goes even further. There are in fact four colors associated with a material. The third color is the ***ambient color*** of the material. Ambient light refers to a general level of illumination that does not come directly from a light source. It consists of light that has been reflected and re-reflected so many times that it is no longer coming from any particular source. Ambient light is why shadows are not absolutely black. In fact, ambient light is only a crude approximation for the reality of multiply reflected light, but it is better than ignoring multiple reflections entirely. The ambient color of a material determines how it will reflect various wavelengths of ambient light. Ambient color is generally set to be the same as the diffuse color.

The fourth color associated with a material is an ***emission color***. The emission color is color that does not come from any external source, and therefore seems to be emitted by the material itself. This does not mean that the object is giving off light that will illuminate other objects, but it does mean that the object can be seen even if there is no source of light (not even ambient light). In the presence of light, the object will be brighter than can be accounted for by the light that illuminates it, and in that sense it might appear to glow. The emission color is used only rarely.

In the next section, we will look at how to use materials in OpenGL. The section after that will cover working with light sources, which have their own set of properties.

4.2 OpenGL Materials

OpenGL uses the term ***material*** to refer to the properties of an object that determine how it interacts with light. In standard OpenGL processing (when it has not been overridden by the GL Shading Language), material properties are assigned to vertices. Lighting calculations are done for vertices only, and the results of those calculations are interpolated to other points of the object. The calculations involved are summarized in a mathematical formula known as the lighting equation, which will be covered in the next section. But you should know that the whole calculation is a rather loose approximation for physical reality, and the use of interpolation is another entire level of approximation that can introduce serious distortion, especially when using large polygons. The system is designed for speed of calculation rather than perfect realism.

4.2.1 Setting Material Properties

Material properties are set using the *glMaterial** family of commands, principly *glMaterialfv* and *glMateriali*. (There are no versions of this command that take parameters of type **double**.) These commands are defined in the *GL* class and are specified by

```
public void glMateriali(int face, int propertyName, int propertyValue)
public void glMaterialfv(int face, int propertyName, float[] propertyValue, int offset)
```

The first parameter tells which face of a polygon the command applies to. It must be one of the constants *GL.GL_FRONT_AND_BACK*, *GL.GL_FRONT* or *GL.GL_BACK*. This reflects the fact that different material properties can be assigned to the front and the back faces of polygons, since it sometimes desirable for the front and the back faces to have a different appearance. When the *face* parameter is *GL.GL_FRONT_AND_BACK*, the command sets the value of the material property for both sides simultaneously. The front material is also used for point and line primitives. Note that the back material is ignored completely unless two-sided lighting has been turned on by calling

```
gl.glLightModeli(GL.GL_LIGHT_MODEL_TWO_SIDE, GL.GL_TRUE)
```

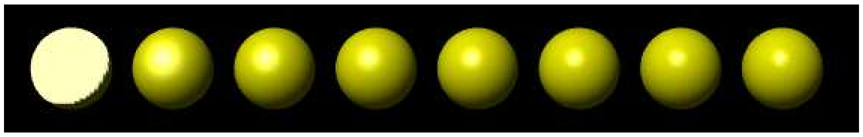
The second parameter for *glMaterial* is *propertyName*, which tells which material property is being set. For *glMateriali*, the only legal property name is *GL.GL_SHININESS*, and the property value must be an integer in the range from 0 to 128, inclusive. This property determines the size and sharpness of specular highlights. Larger values produce smaller, sharper highlights. The default value is zero, which gives very large highlights that are almost never desirable. Try values close to 10 for a larger, fuzzier highlight and values of 100 or more for a small, sharp highlight. (It's worth noting that specular highlights are one area where polygon size can have a significant impact. Since lighting calculations are only done at vertices, OpenGL will entirely miss any specular highlight that should occur in the middle of a polygon but not at the vertices. And when a highlight does occur at a vertex, the interpolation process can smear out the highlight to points that should not show any specular highlight at all. Using very small polygons will alleviate the problem, but will require more computation.)

For *glMaterialfv*, the *propertyName* parameter can be *GL.GL_AMBIENT*, *GL.GL_DIFFUSE*, *GL.GL_AMBIENT_AND_DIFFUSE*, *GL.GL_SPECULAR*, or *GL.GL_EMISSION*. The names refer to the four different types of material color supported by OpenGL, as discussed in the previous section. The property name *GL.GL_AMBIENT_AND_DIFFUSE* allows both the ambient and the diffuse material colors to be set simultaneously to the same value. The third parameter for *glMaterialfv* is an array of type **float**`[]`, and the fourth parameter specifies an index in the array. The index is often zero and is absent in the C API. Four numbers in the array, starting at the specified index, specify the red, green, blue, and alpha components of a color. The alpha component is used for blending, or transparency; for the time being, we will set alpha equal to 1.

In the case of the red, blue, and green components of the ambient, diffuse, or specular color, the term "color" really means reflectivity. That is, the red component of a color gives the proportion of red light hitting the surface that is reflected by that surface, and similarly for green and blue. There are three different types of reflective color because there are three different types of light in the environment, and a material can have a different reflectivity for each type of light. (More about that in the next section.)

The red, green, and blue component values are generally between 0 and 1, but they are not clamped to that range. That is, it is perfectly legal to have a negative color component or a color component that is greater than 1. For example, setting the red component to 1.5 would mean that the surface reflects 50% more red light than hits it—a physical impossibility, but something that you might want to do for effect.

The default material has ambient color (0.2,0.2,0.2,1) and diffuse color (0.8,0.8,0.8,1). Specular and emission color are both black, that is, (0,0,0,1). It's not surprising that materials, by default, do not emit extra color. However, it is a little surprising that materials, by default, have no specular reflection. This means that the objects that you have seen in all our examples so far exhibit ambient and diffuse color only, with no specular highlights. Here is an image to change that:



This image shows eight spheres that differ only in the value of the *GL_SHININESS* material property. The ambient and diffuse material colors are set to (0.75,0.75,0,1), for a general yellow appearance. The specular color is (0.75,0.75,0.75,1), which adds some little blue to the specular highlight, making it appear whiter as well as brighter than the rest of the sphere. For the sphere on the left, the shininess is 0, which leads to an ugly specular "highlight" that covers an entire hemisphere. Going from left to right, the shininess increases by 16 from one sphere to the next. The material colors for this image were specified using

```
    gl.glMaterialfv(GL.GL_FRONT, GL.GL_SPECULAR,
                              new float[] { 0.75f, 0.75f, 0.75f, 1 },  0);
gl.glMaterialfv(GL.GL_FRONT, GL.GL_AMBIENT_AND_DIFFUSE,
                              new float[] { 0.75f, 0.75f, 0, 1 },  0);
```

and the shininess for sphere number i was set using

```
gl.glMateriali(GL.GL_FRONT, GL.GL_SHININESS, i*16);
```

4.2.2 Color Material

Materials are used in lighting calculations and are ignored when lighting is not enabled. Similarly, the current color as set by *glColor** is ignored, by default, when lighting is enabled. However, as we have seen, calling *gl.glEnable*(*GL.GL_COLOR_MATERIAL*) will cause OpenGL to take the current color into account even while lighting is enabled.

More specifically, enabling *GL_COLOR_MATERIAL* tells OpenGL to substitute the current color for the ambient and for the diffuse material color when doing lighting computations. The specular and emission material colors are not affected. However, you can change this default behavior of color material by using the method

```
public void glColorMaterial(int face, int propertyName)
```

to say which material properties should be tied to *glColor*. The parameters in this method correspond to the first two parameters of *glMaterialfv*. That is, *face* can be *GL.GL_FRONT_AND_BACK*, *GL.GL_FRONT*, or *GL.GL_BACK*, and *propertyName* can be *GL_AMBIENT_AND_DIFFUSE*, *GL_DIFFUSE*, *GL_AMBIENT*, *GL_SPECULAR*, or *GL_EMISSION*. The most likely use of this method is to call

```
gl.glColorMaterial(GL.GL_FRONT_AND_BACK, GL.GL_DIFFUSE)
```

so that *GL_COLOR_MATERIAL* will affect only the diffuse material color and not the ambient color.

* * *

In Section 3.4, we saw how to use the *glDrawArrays* and *glDrawElements* methods with vertex arrays, normal arrays, and texture coordinate arrays. It is also possible to specify a color array to be used with these methods. (More exactly, in Jogl, the color data must actually be stored in a Java nio *Buffer* rather than an array.) Use of a color array must be enabled by calling

```
gl.glEnableClientState(GL.GL_COLOR_ARRAY)
```

and *gl.glColorPointer* must be called to specify the location of the data. Exactly how to do this depends on whether a vertex buffer object is used, but the format is exactly parallel to that used with *gl.glVertexPointer* for specifying the location of vertex data.

Furthermore, it's important to note that using *GL_COLOR_ARRAY* is equivalent to calling *glColor** for each vertex that is generated. If lighting is enabled, these colors are ignored, unless *GL_COLOR_MATERIAL* has been enabled. That is, if you are using a color array while lighting is enabled, you must call *gl.glEnable(GL.GL_COLOR_MATERIAL)* before calling *gl.glDrawArrays* or *gl.glDrawElements*. Otherwise, the color data will not have any effect. You could also, optionally, call *gl.glColorMaterial* to say which material property the color data will affect.

4.3 OpenGL Lighting

Our 3D worlds so far have been illuminated by just one light—a white light shining from the direction of the viewer onto the scene. You get this light just by turning on lighting with *gl.glEnable*(*GL.GL_LIGHTING*) and enabling light number zero with *gl.glEnable*(*GL.GL_LIGHT0*). This "viewpoint light" shines on everything that the viewer can see, and it is sufficient for some purposes. However, in OpenGL, it is possible to define multiple lights. They can shine from different directions and can have various colors. Every OpenGL implementation is required to support at least eight lights, which are identified by the constants *GL_LIGHT0*, *GL_LIGHT1*, ..., *GL_LIGHT7*. (These constants are consecutive integers.) Each light can be separately enabled and disabled; they are all off by default. Only the lights that are enabled while a vertex is being rendered can have any effect on that vertex.

4.3.1 Light Color and Position

Light number zero, as we have seen, is white by default. All the other lights, however, are black. That is, they provide no illumination at all even if they are enabled. The color and other properties of a light can be set with the *glLight** family of commands, most notably

```
public void glLightfv(int light, int propName, float[] propValue, int offset)
```

The first parameter, *light*, specifies the light whose property is being set. It must be one of the constants *GL.GL_LIGHT0*, *GL.GL_LIGHT1*, and so on. The second parameter, *propName*, specifies which property of the light is being set. Commonly used properties are *GL.GL_POSITION*, *GL.GL_AMBIENT*, *GL.GL_DIFFUSE*, and *GL.GL_SPECULAR*. Some other properties are discussed in the next subsection. The third parameter of *glLightfv* is an array that contains the new value for the property. The fourth parameter is, as usual, the starting index of that value in the array.

A light can have color. In fact, each light in OpenGL has an ambient, a diffuse, and a specular color, which are set using *glLightfv* with property names *GL.GL_AMBIENT*, *GL.GL_DIFFUSE*, and *GL.GL_SPECULAR* respectively. Just as the color of a material is more properly referred to as reflectivity, color of a light is more properly referred to as ***intensity*** or energy. A light color in OpenGL is specified by four numbers giving the red, green, blue, and alpha intensity values for the light. These values are often between 0 and 1 but are not clamped to that range. (I have not, however, been able to figure out how the alpha component of a light color is used, or even if it is used; it should be set to 1.)

The diffuse intensity of a light is the aspect of the light that interacts with diffuse material color, and the specular intensity of a light is what interacts with specular material color. It is common for the diffuse and specular light intensities to be the same. For example, we could make a somewhat blue light with

```
float[] bluish = { 0.3f, 0.3f, 0.7f, 1 };
gl.glLightfv(GL.GL_LIGHT1, GL.GL_DIFFUSE, bluish, 0);
gl.glLightfv(GL.GL_LIGHT1, GL.GL_SPECULAR, bluish, 0);
```

The ambient intensity of a light works a little differently. Recall that ambient light is light that is not directly traceable to any light source. Still, it has to come from somewhere and we can imagine that turning on a light should increase the general level of ambient light in the environment. The ambient intensity of a light in OpenGL is added to the general level of ambient light. This ambient light interacts with the ambient color of a material, and this interaction has no dependence on the position of any light source. So, a light doesn't have to shine on an object for the object's ambient color to be affected by the light source; the light source just has to be turned on. Since ambient light should never be too intense, the ambient intensity of a light source should always be rather small. For example, we might want our blue light to add a slight bluish tint to the ambient light. We could do this by calling

```
gl.glLightfv(GL.GL_LIGHT1, GL.GL_AMBIENT, new float[] { 0, 0, 0.1f, 0}, 0);
```

I should emphasize again that this is all just an approximation, and in this case not one that has a basis in the physics of the real world. Real light sources do not have separate ambient, diffuse, and specular colors.

* * *

The other major property of a light is its position. There are two types of lights, ***positional*** and ***directional***. A positional light represents a light source at some point in 3D space. Light is emitted from that point—in all directions by default, or, if the light is made into a spotlight, in a cone whose vertex is at that point. A directional light, on the other hand, shines in parallel rays from some set direction. A directional light imitates light from the sun, whose rays are essentially parallel by the time they reach the earth.

The type and position or direction of a light are set using *glLightfv* with property name equal to *GL.GL_POSITION*. The property value is an array of four numbers (x,y,z,w), of which at least one must be non-zero. When the fourth number, w, is zero, then the light is directional and the point (x,y,z) specifies the direction of the light: The light rays shine in the direction of the line from the point (x,y,z) to the origin. This is related to homogeneous coordinates: The source of the light can be considered to be a point at infinity in the direction of (x,y,z). (See Subsection 3.1.3.) On the other hand, if the fourth number, w, is 1, then the light is positional and is located at the point (x,y,z). Again, this is really homogeneous coordinates: Any non-zero value for w specifies a positional light at the point $(x/w,y/w,z/w)$. The default position for all lights is (0,0,1,0), representing a directional light shining from the positive direction of the z-axis (and towards the negative direction of the z-axis).

The position specified for a light is transformed by the modelview matrix that is in effect at the time the position is set using *glLightfv*. Thus, lights are treated in the same way as other objects in OpenGL in that they are subject to the same transformations. For example, setting the position of light number 1 with

```
gl.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION, new float[] { 1,2,3,1 }, 0);
```

puts the light in the same place as

```
gl.glTranslatef(1,2,3);
gl.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION, new float[] { 0,0,0,1 }, 0);
```

The *Camera* class in package *glutil* allows the user to rotate the 3D world using the mouse. If *cam* is a Camera, then *cam.apply*(*gl*) can be called at the beginning of *display*() to set up the projection and viewing transformations. If a light position is set up after calling *cam.apply*, then that light will rotate along with the rest of the scene.

Note that the default light position is, in effect, set before any transformation has been applied and is therefore given directly in eye coordinates. That is, when we say that the default light shines towards the negative direction of the z-axis, we mean the z-axis in eye coordinates, in which the viewer is at the origin, looking along the negative z-axis. So, the default position makes a light into a viewpoint light that shines in the direction the viewer is looking.

* * *

The sample program *LightDemo.java* demonstrates various aspects of light and material. The online version of this section includes this program as an applet. I encourage you to try the applet or the corresponding application.

In the *LightDemo* example, a wave-like surface is illuminated by three lights: a red light that shines from above, a blue light that shines from below, and a white light that shines down the x-axis. (In this example, the z-axis is pointing up, and the x-axis is pointing out of the screen.) The surface itself is gray, although it looks colored under the colored lights. Both the surface material and the lights have a non-zero specular component to their color. A set of axes and a grid of lines on the surface are also shown, at the user's option; these features are drawn with lighting turned off.

The user can use the mouse to rotate the scene. A control below the display determines whether the lights are fixed with respect to the viewer or with respect to the world. In the first case, as the user rotates the scene, the surface rotates but the lights don't move; so, for example, the part of the surface that is illuminated by the red light is whatever part happens to be facing upwards in the display. In the second case, the lights rotate along with the surface; so, for example, the red light always illuminates the same part of the surface, no matter how the scene has been rotated. Note that even in the second case, the positions of specular highlights on the surface **do** change as the scene is rotated, since specular highlights depend on the position of the viewer relative to the surface, as well as on the position of the lights relative to the surface.

The program also has controls that let the user turn the red, blue, and white lights, as well as ambient light, on and off. If all the lights are turned off, the surface disappears entirely. If only ambient light is on, the surface appears a a flat patch of gray. Ambient light by itself does not give any 3D appearance to the surface, since it does not depend in any way on the orientation of the surface or the location of any light source. The user should try other combinations of settings and try to understand what is going on. Also, take a look at the *source code.*

4.3.2 Other Properties of Lights

In addition to color and position, lights have six other properties that are more rarely used. These properties have to do with ***spotlights*** and ***attenuation***. Five of the six properties are numbers rather than arrays and are set using either of the following forms of *glLight**

```
public void glLighti(int light, int propName, int propValue)
public void glLightf(int light, int propName, float propValue)
```

It is possible to turn a positional light into a spotlight, which emits light in a cone of directions, rather than in all directions. (For directional lights, spotlight settings are ignored.) A positional light is a spotlight if the value of its *GL_SPOT_CUTOFF* property is set to a number in the range 0 to 90. The value of this property determines the size of the cone of light emitted by the spotlight; it gives the angle, measured in degrees, between the axis of the cone and the side of the cone.

The default value of *GL_SPOT_CUTOFF* is a special value, 180, that indicates an ordinary light rather than a spotlight. The only legal values for *GL_SPOT_CUTOFF* are 180 and numbers in the range 0 to 90.

A spotlight is not completely specified until we know what direction it is shining. The direction of a spotlight is given by its *GL_SPOT_DIRECTION* property, an array of three numbers that can be set using the *glLightfv* method. Like the light's position, the spotlight direction is subject to the modelview transformation that is in effect at the time when the spotlight direction is set. The default value is (0,0,−1), giving a spotlight that points in the negative direction of the z-axis. Since the default is set before the modelview transformation has been changed from the identity, this means the viewer's z-axis. That is, the default spotlight direction makes it point directly away from the viewer, into the screen.

For example, to turn light number 1 into a spotlight with a cone angle of 30 degrees, positioned at the point (10,15,5) and pointing toward the origin, you can use these commands:

```
gl.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION, new float[] {10,15,5}, 0);
gl.glLighti(GL.GL_LIGHT1, GL.GL_SPOT_CUTOFF, 30);
gl.glLightfv(GL.GL_LIGHT1, GL.GL_SPOT_DIRECTION, new float[] {-10,-15,-5}, 0);
```

By default, everything in the cone of a spotlight is illuminated evenly. It is also possible to make the illumination decrease as the angle away from the axis of the cone increases. The value of the *GL_SPOT_EXPONENT* property of a light determines the rate of falloff. This property must have a value in the range 0 to 128. The default value, 0, gives even illumination. Other values cause the illumination to fall off with increasing angle, with larger values giving a more rapid falloff.

Since lighting calculations are only done at vertices, spotlights really only work well when the polygons that they illuminate are very small. When using larger polygons, don't expect to see a nice circle of illumination.

* * *

In real-world physics, the level of illumination from a light source is proportional to the reciprocal of the square of the distance from the light source. We say that the light "attenuates" with distance. OpenGL lights do not follow this model because it does not, in practice, produce nice pictures. By default in OpenGL, the level of illumination from a light does not depend at all on distance from the light. However, it is possible to turn on attenuation for a positional light source. For an vertex at distance r from a light source, the intensity of the light on that vertex is computed as

$$I * (1 / (a + b*r + c*r^2))$$

where I is the intrinsic intensity of the light (that is, its color level—this calculation is done for each of the red, green, and blue components of the color). The numbers a, b, and c are values of the properties *GL_CONSTANT_ATTENUATION*, *GL_LINEAR_ATTENUATION*, and *GL_QUADRATIC_ATTENUATION*. By default, a is 1 while b and c are 0. With these default values, the intensity of the light at the vertex is equal to I, the intrinsic intensity of the light, and there is no attenuation.

Attenuation can sometimes be useful to localize the effect of a light, especially when there are several lights in a scene. Note that attenuation applies only to positional lights. The attenuation properties are ignored for directional lights.

4.3.3 The Global Light Model

There are a few properties of the OpenGL lighting system that are "global" in the sense that they are not properties of individual lights. These properties can be set using the *glLightModeli* and *glLightModelfv* methods:

```
public void glLightModelfv(int propertyName, float[] propertyValue, int offset)
public void glLightModeli(int propertyName, int propertyValue)
```

There is only one property that can be set with *glLightModelfv*, the global ambient light intensity, using property name *GL.GL_LIGHT_MODEL_AMBIENT*. Global ambient light is ambient light that is present in the environment even when no light sources are turned on. The default value is (0.2,0.2,0.2,1). For a yellowish ambient light, for example, you could say

```
gl.glLightModelfv(GL.GL_LIGHT_MODEL_AMBIENT,
                                          new float[] { 0.2f, 0.2f, 0, 1 }, 0);
```

There are three properties that can be set using *glLightModeli*. We have already encountered one of them. The command

```
gl.glLightModeli(GL.GL_LIGHT_MODEL_TWO_SIDE, GL.GL_TRUE);
```

turns on two-sided lighting, which tells OpenGL to compute normal vectors for the back faces of polygons by reversing the normal vectors that were specified for the front faces. This also enables the front and the back faces of polygons to have different material properties. The second parameter can be *GL.GL_TRUE* or *GL.GL_FALSE*, but these are actually just symbolic constants for the numbers 1 and 0.

By default, when computing specular highlights, OpenGL assumes that the direction to the viewer is in the positive direction of the z-axis. Essentially, this assumes, for the purpose of specular light calculations, that the viewer is "at infinity." This makes the computation easier but does not produce completely accurate results. To make OpenGL do the correct calculation, you can call

```
gl.glLightModeli(GL.GL_LIGHT_MODEL_LOCAL_VIEWER, GL.GL_TRUE);
```

In practice, this rarely makes a significant difference, but it can be noticeable if the viewer is fairly close to the illuminated objects.

Finally, there is an option that is available only if the OpenGL version is 1.2 or higher, *GL.GL_LIGHT_MODEL_COLOR_CONTROL*. The value for this property must be either *GL.GL_SEPARATE_SPECULAR_COLOR* or *GL_GL_SINGLE_COLOR*. The latter is the default. This default yields poor specular highlights on objects to which a texture has been applied. The alternative will produce better specular highlight on such surfaces by applying the specular highlight after the texture has been applied. To make sure that the option is actually available, you can use the following code:

```
if (gl.isExtensionAvailable("GL_VERSION_1_2") {
   gl.glLightModeli(GL.GL_LIGHT_MODEL_COLOR_CONTROL,
                                      GL.GL_SEPARATE_SPECULAR_COLOR);
}
```

4.3.4 The Lighting Equation

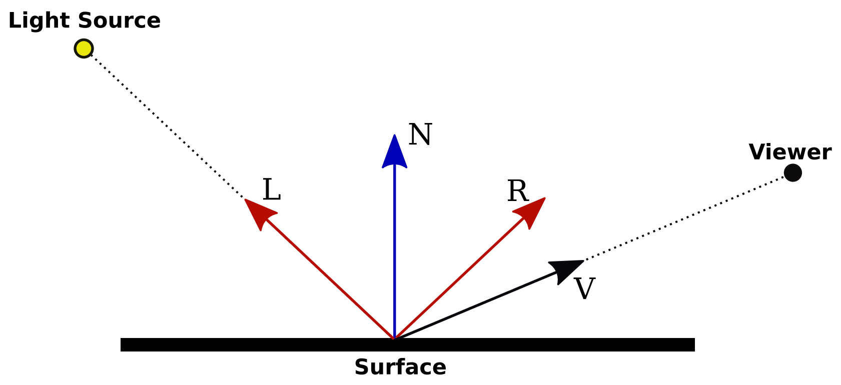
What does it actually mean to say that OpenGL performs "lighting calculations"? The goal of the calculation is produce a color, (r,g,b,a), for a vertex. The alpha component, a, is easy—it's simply the alpha component of the diffuse material color at that vertex. But the calculations of r, g, and b are fairly complex.

Ignoring alpha components, let's assume that the ambient, diffuse, specular, and emission colors of the material have RGB components (ma_r,ma_g,ma_b), (md_r,md_g,md_b), (ms_r,ms_g,ms_b), and (me_r,me_g,me_b), respectively. Suppose that the global ambient intensity is (ga_r,ga_g,ga_b). Then the red component of the vertex color will be

$$\mathtt{r = me}_r \mathtt{ + ga}_r\mathtt{*ma}_r \mathtt{ + I}_{0r} \mathtt{ + I}_{1r} \mathtt{ + I}_{2r} \mathtt{ + \ ...}$$

where I_{ir} is the contribution to the color that comes from the i-th light. A similar equation holds for the green and blue components of the color. This equation says that the emission color is simply added to any other contributions to the color. The contribution of global ambient light is obtained by multiplying the global ambient intensity by the material ambient color. This is the mathematical way of saying that the material ambient color is the proportion of the ambient light that is reflected by the surface.

The contributions from the light sources are much more complicated. Note first of all that if a light source is disabled, then the contribution from that light source is zero. For an enabled light source, we have to look at the geometry as well as the colors:



In this illustration, N is the normal vector at the point whose color we want to compute. L is a vector that points towards the light source, and V is a vector that points towards the viewer. (Both the viewer and the light source can be "at infinity", but the direction is still well-defined.) R is the direction of the reflected ray, that is, the direction in which a light ray from the source will be reflected when it strikes the surface at the point in question. The angle between N and L is the same as the angle between N and R. All of the vectors are unit vectors, with length 1. Recall that for unit vectors A and B, the inner product $A \cdot B$ is equal to the cosine of the angle between the two vectors. Inner products occur at several points in the lighting equation.

Now, let's say that the light has ambient, diffuse, and specular color components (la_r,la_g,la_b), (ld_r,ld_g,ld_b), and (ls_r,ls_g,ls_b). Also. let mh be the value of the shininess property (*GL_SHININESS*) of the material. Then the contribution of this light source to the red component of the vertex color can be computed as

$$I_r = la_r*ma_r + f*att*spot*(\ ld_r*md_r*(L\cdot N) + ls_r*ms_r*max(0,V\cdot R)^{mh}\ )$$

with similar equations for the green and blue components. Here, f is 0 if the surface is facing away from the light and is 1 otherwise. f is 1 when $L{\cdot}N$ is greater than 0, that is, when the angle between L and N is less than 90 degrees. When f is zero, there is no diffuse or specular contribution from the light to the color of the vertex. Note that even when f is 0, the ambient component of the light can still affect the vertex color.

In the equation, *att* is the attenuation factor, which represents attenuation of the light intensity due to distance from the light. The value of *att* is 1 if the light source is directional. If the light is positional, then *att* is computed as $1/(a+b*r+c*r^2)$, where a, b, and c are the attenuation constants for the light and r is the distance from the light source to the vertex. And *spot* accounts for spotlights. For directional lights and regular positional lights, *spot* is 1. For a spotlight, *spot* is zero when the angle between N and L exceeds the cutoff angle of the spotlight. Otherwise, *spot* is given by $(N{\cdot}L)^e$, where e is the value of the falloff property (*GL_SPOT_EXPONENT*) of the light.

The diffuse component of the color, before adjustment by *f*, *att*, and *spot*, is given by `ld`$_r$`*md`$_r$`*(L·N)` This represents the diffuse intensity of the light times the diffuse reflectivity of the material, multiplied by the cosine of the angle between *L* and *N*. The angle is involved because for a larger angle, the same amount of energy from the light is spread out over a greater area. As the angle increases from 0 to 90 degrees, the cosine of the angle decreases from 1 to 0, so the larger the angle, the smaller the diffuse color contribution. The specular component, `ls`$_r$`*ms`$_r$`*max(0,V·R)`^{mh}, is similar, but here the angle involved is the angle between the reflected ray and the viewer, and the cosine of this angle is raised to the exponent *mh*. The exponent is the material's shininess property. When this property is 0, there is no dependence on the angle (as long as the angle is greater than 0), and the result is the sort of huge and undesirable specular highlight that we have seen in this case. For positive values of shininess, the specular contribution is maximal when this angle is zero and it decreases as the angle increases. The larger the shininess value, the faster the rate of decrease. The result is that larger shininess values give smaller, sharper specular highlights.

Remember that the same calculation is repeated for every enabled light and that the results are combined to give the final vertex color. It's easy, especially when using several lights, to end up with color components larger than one. In the end, before the color is used to color a pixel on the screen, the color components must be clamped to the range zero to one. Values greater than one are replaced by one. It's easy, when using a lot of light, to produce ugly pictures in which large areas are a uniform white because all the color values in those areas exceeded one. All the information that was supposed to be conveyed by the lighting has been lost. The effect is similar to an over-exposed photograph. It can take some work to find appropriate lighting levels to avoid this kind of over-exposure.

4.4 Lights and Materials in Scenes

In this section, we turn to some of the practicalities of using lights and materials in a scene and, in particular, in a scene graph.

4.4.1 The Attribute Stack

OpenGL is a state machine with dozens or hundreds of state variables. It can be easy for a programmer to lose track of the current state. In the case of transformation matrices, the OpenGL commands *glPushMatrix* and *glPopMatrix* offer some help for managing state. Typically, these methods are used when temporary changes are made to the transformation matrix. Using a stack with push and pop is a neat way to save and restore state. OpenGL extends this idea beyond the matrix stacks.

Material and light properties are examples of *attributes*. OpenGL has an ***attribute stack*** that can be used for saving and restoring attribute values. The push and pop methods for the attribute stack are defined in class *GL*:

```
public void glPushAttrib(int mask)
public void glPopAttrib()
```

There are many attributes that can be stored on the attribute stack. The attributes are divided into attribute groups, and one or more groups of attributes can be pushed onto the stack with a single call to *glPushAttrib*. The *mask* parameter tells which group or groups of attributes are to be pushed. Each call to *glPushAttrib* must be matched by a later call to *glPopAttrib*. Note that *glPopAttrib* has no parameters, and it is not necessary to tell it which attributes to pop—it will simply restore the values of all attributes that were saved onto the stack by the matching call to *glPushAttrib*.

The attribute that most concerns us here is the one identified by the constant *GL.GL_LIGHTING_BIT*. A call to

```
gl.glPushAttrib( GL.GL_LIGHTING_BIT );
```

will push onto the attribute stack all the OpenGL state variables associated with lights and materials. The command saves material properties, properties of lights, settings for the global light model, the enable state for lighting and for each individual light, the settings for *GL_COLOR_MATERIAL*, and the shading model (*GL_FLAT* or *GL_SMOOTH*). All the saved values can be restored later by the single matching call to *gl.glPopAttrib*().

The parameter to *glPushAttrib* is a bit mask, which means that you can "or" together several attribute group names in order to save the values in several groups in one step. For example,

```
gl.glPushAttrib( GL.GL_LIGHTING_BIT | GL.GL_CURRENT_BIT );
```

will save values in the *GL_CURRENT_BIT* group as well as in the *GL_LIGHTING_BIT* group. You might want to do this since the current color is not included in the "lighting" group but is included in the "current" group. It is better to combine groups in this way rather than to use separate calls to *glPushAttrib*, since the attribute stack has a limited size. (The size is guaranteed to be at least 16, which should be sufficient for most purposes. However, you might have to be careful when rendering very complex scenes.)

There are other useful attribute groups. *GL.GL_POLYGON_BIT*, *GL.GL_LINE_BIT*, and *GL.GL_POINT_BIT* can be used to save attributes relevant to the rendering of polygons, lines, and points, such as the point size, the line width, the settings for line stippling, and the settings relevant to polygon offset. *GL.GL_TEXTURE_BIT* can be used to save many settings that control how textures are processed (although not the texture images). *GL.GL_ENABLE_BIT* will save the values of **boolean** properties that can be enabled and disabled with *gl.glEnable* and *gl.glDisable*.

Since *glPushAttrib* can be used to push large groups of attribute values, you might think that it would be more efficient to use the *glGet* family of commands to read the values of just those properties that you are planning to modify, and to save the old values in variables in your program so that you can restore them later. (See Subsection 3.1.4.) But in fact, a *glGet* command can require your program to communicate with the graphics card—and wait for the response, which is the kind of thing that can hurt performance. In contrast, calls to *glPushAttrib* and *glPopAttrib* can be queued with other OpenGL commands and sent to the graphics card in batches, where they can be executed efficiently by the graphics hardware. In fact, you should always prefer using *glPushAttrib*/*glPopAttrib* instead of a *glGet* command when you can.

I should note that there is another stack, for "client attributes." These are attributes that are stored on the client side (that is, in your computer's main memory) rather than in the graphics card. There is really only one group of client attributes that concerns us:

```
gl.glPushClientAttrib(GL.GL_CLIENT_VERTEX_ARRAY_BIT);
```

will store the values of settings relevant to using vertex arrays, such as the values for *GL_VERTEX_POINTER* and *GL_NORMAL_POINTER* and the enabled states for the various arrays. This can be useful when drawing primitives using *glDrawArrays* and *glDrawElements.* (Subsection 3.4.2.) The values saved on the stack can be restored with

```
gl.glPopClientAttrib();
```

* * *

Now, how can we use the attribute stack in practice? It can be used any time complex scenes are drawn when you want to limit state changes to just the part of the program where they are needed. When working with complex scenes, you need to keep careful control over the state. It's not a good idea for any part of the program to simply change some state variable and leave it that way, because any such change affects all the rendering that is done down the road. The typical pattern is to set state variables in the *init*() method to their most common values. Other parts of the program should not make permanent changes to the state. The easiest way to do this is to enclose the code that changes the state between push and pop operations.

Complex scenes are often represented as scene graphs. A scene graph is a data structure that represents the contents of a scene. We have seen how a screen graph can contain basic nodes representing geometric primitives, complex nodes representing groups of sub-nodes, and nodes representing viewers. In our scene graphs so far, represented by the package *simplescenegraph3d*, a node can have an associated color and if that color is null then the color is inherited from the parent of the node in the scene graph or from the OpenGL environment if it has no parent.

A more realistic package for scene graphs, *scenegraph3d*, adds material properties to the nodes and adds a node type for representing lights. All the nodes in a scene graph are represented by sub-classes of the class *SceneNode3D* from the *scenegraph3d* package. This base class defines and implements both the transforms and the material properties for the nodes in the graph. Now, instead of just the current drawing color, there are instance variables to represent all the material properties—ambient color, diffuse color, specular color, emission color, and shininess. The *SceneNode3D* class has a *draw*() method that manages the transforms and the material properties (and calls another method, *basicDraw*() to do the actual drawing). Here is the method:

```
final public void draw(GL gl) {
    boolean hasColor = (color != null) || (specularColor != null)
              || ambientColor != null || diffuseColor != null
              || emissionColor != null || shininess >= 0;
    if (hasColor) {
       gl.glPushAttrib(GL.GL_LIGHTING_BIT | GL.GL_CURRENT_BIT);
       if (color != null)
          gl.glGetFloatv(GL.GL_CURRENT_COLOR, color, 0);
       if (ambientColor != null)
          gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_AMBIENT, ambientColor, 0);
       if (diffuseColor != null)
          gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_DIFFUSE, diffuseColor, 0);
       if (specularColor != null)
          gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_SPECULAR, specularColor, 0);
       if (emissionColor != null)
          gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_EMISSION, emissionColor, 0);
       if (shininess >= 0)
          gl.glMateriali(GL.GL_FRONT_AND_BACK, GL.GL_SHININESS, shininess);
    }
    gl.glPushMatrix();
    gl.glTranslated(translateX, translateY, translateZ);
    gl.glRotated(rotationAngle, rotationAxisX, rotationAxisY, rotationAxisZ);
    gl.glScaled(scaleX, scaleY, scaleZ);
    drawBasic(gl);  // Render the part of the scene represented by this node.
    gl.glPopMatrix();
    if (hasColor)
```

```
        gl.glPopAttrib();
    }
```

This method uses *glPushAttrib* and *glPopAttrib* to make sure that any changes that are made to the color and material properties are limited just to this node. The variable *hasColor* is used to test whether any such changes will actually be made, in order to avoid doing an unnecessary push and pop. The method also uses *glPushMatrix* and *glPopMatrix* to limit changes made to the transformation matrix. The effect is that the material properties and transform are guaranteed to have the same values when the method ends as they did when it began.

4.4.2 Lights in Scene Graphs

We have seen that the position of a light is transformed by the modelview matrix that is in effect when the position is set, and a similar statement holds for the direction of a spotlight. In this way, lights are like other objects in a scene. The contents of a scene are often represented by a scene graph, and it would be nice to be able to add lights to a scene graph in the same way that we add geometric objects. We should be able to animate the position of a light in a scene graph by applying transformations to the graph node that represents the light, in exactly the same way that we animate geometric objects.

We could, for example, place a light at the same location as a sun or a streetlight, and the illumination from the light would appear to come from the visible object that is associated with the light. If the sun moves during an animation, the light can move right along with it.

There is one way that lights differ from geometric objects: Geometric objects can be rendered in any order, but lights have to be set up and enabled before any of the geometry that they illuminate is rendered. So we have to work in two stages: Set up the lighting, then render the geometry. When lights are represented by nodes in a scene graph, we can do this by traversing the scene graph twice. During the first traversal, the properties and positions of all the lights in the scene graph are set, and geometry is ignored. During the second traversal, the geometry is rendered and the light nodes are ignored.

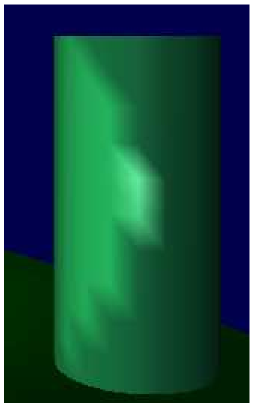
This is the approach taken in the scene graph package, *scenegraph3d*. This package contains a *LightNode* class. A *LightNode* represents an OpenGL light. The *draw*() method for a scene graph, shown above, does a traversal of the graph for the purpose of rendering geometry. There is also a *turnOnLights*() method that does a traversal of the scene graph for the purpose of turning on lights and setting their properties, while making sure that they are subject to all the transformations that are applied in the scene graph. The on-line version of this section includes an applet that demonstrates the use of *LightNodes* in scene graphs.

There are four lights in the applet, one inside a sun that rotates around the "world," one in a moon that does likewise, and two spotlights in the headlights of a cart. The headlights come on only at night (when the sun is on the other side of the world). There is also a light that illuminates the scene from the position of the viewer and that can be turned on and off using a control beneath the display area. The viewpoint light is not part of the scene graph. The source code can be found in *MovingLightDemo.java*.

One issue when working with lights in OpenGL is managing the limited number of lights. In fact, the scene graphs defined by the package *scenegraph3d* can handle no more than eight lights; any lights in the scene graph beyond that number are ignored. Lights in the scene graph are assigned one of the light constants (*GL_LIGHT0*, *GL_LIGHT1*, and so on) automatically, as the graph is traversed, so that the user of the scene graph does not have to worry about managing the light constants. One interesting point is that the light constants are not simply assigned to *LightNodes*, and indeed it would be impossible to do so since the same *LightNode* can be encountered more than once during a single traversal of the graph. For example, in the *MovingLightDemo* program, the two headlights of the cart are represented by just one *LightNode*. There are two lights because the node is encountered twice during a graph traversal; the lights are in different locations because different transformations are in effect during the two encounters. A different light constant is used for each encounter, and each adds a new OpenGL light to the scene.

For more details about how lights are managed, see the methods *turnOnLights* and *turnOnLightsBasic* in *scenegraph3d/SceneNode3D.java* and the implementation of *turnOnLightsBasic* in *scenegraph3d/LightNode.java*. It's also worth looking at how scene graphs are used in the sample program *MovingLightDemo.java*.

By the way, that example also demonstrates some of the limitations of OpenGL lighting. One big limitation is that lights in OpenGL do not cast shadows. This means that light simply shines through objects. This is why is is possible to place a light **inside** a sphere and still have that light illuminate other objects: The light simply shines through the surface of the sphere. That's not such a bad thing, but when the sun in the applet is below the world, it shines up through the ground and illuminates objects in the scene from below. The effect is not all that obvious. You have to look for it, but even if you don't notice it, it makes things look subtly wrong. Problems with spotlights are also apparent in the example. As mentioned previously, you don't get a neat circle of light from a spotlight unless the polygons that it illuminates are very small. As a result, the illumination from the headlights flickers a bit as the cart moves. And if you look at the boundary of the circle of illumination from a headlight on an object made up of large polygons, you'll see that the boundary can be very jagged indeed. For example, consider this screenshot of a cylinder whose left side is illuminated by one of the spotlights in the program:



4.5 Textures

TEXTURES WERE INTRODUCED in Subsection 2.4.2. In that section, we looked at Java's *Texture* class, which makes it fairly easy to use image textures. However, this class is not a standard part of OpenGL. This section covers parts of OpenGL's own texture API (plus more detail on the *Texture* class). Textures are a complex subject. This section does not attempt to cover all the details.

4.5.1 Texture Targets

So far, we have only considered texture images, which are two-dimensional textures. However, OpenGL also supports one-dimensional textures and three-dimensional textures. OpenGL has three ***texture targets*** corresponding to the three possible dimensions: *GL_TEXTURE_1D*, *GL_TEXTURE_2D*, and *GL_TEXTURE_3D*. (There are also several other targets that are used for aspects of texture mapping that I won't discuss here.) OpenGL maintains some separate state information for each target. Texturing can be turned on and off for each target with

```
gl.glEnable(GL.GL_TEXTURE_1D);     /     gl.glDisable(GL.GL_TEXTURE_1D);
gl.glEnable(GL.GL_TEXTURE_2D);     /     gl.glDisable(GL.GL_TEXTURE_2D);
gl.glEnable(GL.GL_TEXTURE_3D);     /     gl.glDisable(GL.GL_TEXTURE_3D);
```

At most one texture target will be used when a surface is rendered. If several targets are enabled, 3D textures have precedence over 2D textures, and 2D textures have precedence over 1D. Except for one example later in this section, we will work only with 2D textures.

Many commands for working with textures take a texture target as their first parameter, to specify which target's state is being changed. For example,

```
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
```

tells OpenGL to repeat two-dimensional textures in the *s* direction. To make one-dimensional textures repeat in the *s* direction, you would use *GL.GL_TEXTURE_1D* as the first parameter. Texture wrapping was mentioned at the end of Subsection 2.4.2, where repeat in the *s* direction was set using an object *tex* of type *Texture* with

```
tex.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
```

More generally, for a *Texture*, *tex*, *tex.setTexParameteri*(*prop*,*val*) is simply an abbreviation for *gl.glSetTexParameteri*(*tex.getTarget*(),*prop*,*val*), where *tex.getTarget*() returns the target—most likely *GL_TEXTURE_2D*—for the texture object. The *Texture* method simply provides access to the more basic OpenGL command. Similarly, *tex.enable*() is equivalent to *gl.glEnable*(*tex.getTarget*()).

Texture targets are also used when setting the ***texture environment***, which determines how the colors from the texture are combined with the color of the surface to which the texture is being applied. The combination mode for a given texture target is set by calling

```
gl.glTexEnvi( target, GL.GL_TEXTURE_ENV_MODE, mode );
```

where *target* is the texture target, such as *GL.GL_TEXTURE_2D*, for which the combination mode is being changed. The default value of *mode* is *GL.GL_MODULATE*, which means that the color components from the surface are multiplied by the color components from the texture. This is commonly used with a white surface material. The surface material is first used in the lighting computation to produce a basic color for each surface pixel, before combination with the texture color. A white surface material means that what you end up with is basically the texture color, modified by lighting effects. This is usually what you want, but there are other texture combination modes for special purposes. For example, the *GL.GL_REPLACE* mode will completely replace the surface color with the texture color. In fact, the texture environment offers many options and a great deal of control over how texture colors are used. However, I will not cover them here.

4.5.2 Mipmaps and Filtering

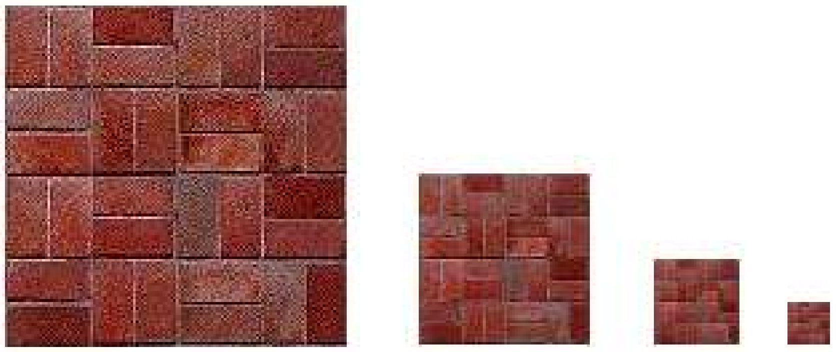
When a texture is applied to a surface, the pixels in the texture do not usually match up one-to-one with pixels on the surface, and in general, the texture must be stretched or shrunk as it is being mapped onto the surface. Sometimes, several pixels in the texture will be mapped to the same pixel on the surface. In this case, the color that is applied to the surface pixel must somehow be computed from the colors of all the texture pixels that map to it. This is an example of ***filtering***; in particular, it is "minification filtering" because the texture is being shrunk. When one pixel from the texture covers more than one pixel on the surface, the texture has to be magnified, and we have an example of "magnification filtering."

One bit of terminology before we proceed: The pixels in a texture are referred to as ***texels***, short for texture pixels, and I will use that term from now on.

When deciding how to apply a texture to a point on a surface, OpenGL has the texture coordinates for that point. Those texture coordinates correspond to one point in the texture, and that point lies in one of the texture's texels. The easiest thing to do is to apply the color of that texel to the point on the surface. This is called ***nearest neighbor filtering***. It is very fast, but it does not usually give good results. It doesn't take into account the difference in size between the pixels on the surface and the texels. An improvement on nearest neighbor filtering is ***linear filtering***, which can take an average of several texel colors to compute the color that will be applied to the surface.

The problem with linear filtering is that it will be very inefficient when a large texture is applied to a much smaller surface area. In this case, many texels map to one pixel, and computing the average of so many texels becomes very inefficient. OpenGL has a neat solution for this: ***mipmaps***.

A mipmap for a texture is a scaled-down version of that texture. A complete set of mipmaps consists of the full-size texture, a half-size version in which each dimension is divided by two, a quarter-sized version, a one-eighth-sized version, and so on. If one dimension shrinks to a single pixel, it is not reduced further, but the other dimension will continue to be cut in half until it too reaches one pixel. In any case, the final mipmap consists of a single pixel. Here are the first few images in the set of mipmaps for a brick texture:



You'll notice that the mipmaps become small very quickly. The total memory used by a set of mipmaps is only about one-third more than the memory used for the original texture, so the additional memory requirement is not a big issue when using mipmaps.

Mipmaps are used only for minification filtering. They are essentially a way of pre-computing the bulk of the averaging that is required when shrinking a texture to fit a surface. To texture a pixel, OpenGL can first select the mipmap whose texels most closely match the size of the pixel. It can then do linear filtering on that mipmap to compute a color, and it will have to average at most a few texels in order to do so.

Starting with OpenGL Version 1.4, it is possible to get OpenGL to create and manage mipmaps automatically. For automatic generation of mipmaps for 2D textures, you just have to say

```
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_GENERATE_MIPMAP, GL.GL_TRUE);
```

and then forget about 2D mipmaps! Of course, you should check the OpenGL version before doing this. In earlier versions, if you want to use mipmaps, you must either load each mipmap individually, or you must generate them yourself. (The GLU library has a method, *gluBuild2DMipmaps* that can be used to generate a set of mipmaps for a 2D texture, with similar functions for 1D and 3D textures.) The best news, perhaps, is that when you are using Java *Texture* objects to represent textures, the *Texture* will manage mipmaps for you without any action on your part except to ask for mipmaps when you create the object. (The methods for creating *Textures* have a parameter for that purpose.)

* * *

OpenGL supports several different filtering techniques for minification and magnification. The filters that can be used can be set with *glTexParameteri.* For the 2D texture target, for example, you would call

```
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, magFilter);
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, minFilter);
```

where *magFilter* and *minFilter* are constants that specify the filtering algorithm. For the *magFilter*, the only options are *GL.GL_NEAREST* and *GL.GL_LINEAR*, giving nearest neighbor and linear filtering. The default for the MAG filter is *GL_LINEAR*, and there is rarely any need to change it. For *minFilter*, in addition to *GL.GL_NEAREST* and *GL.GL_LINEAR*, there are four options that use mipmaps for more efficient filtering. The default MIN filter is *GL.GL_NEAREST_MIPMAP_LINEAR* which does averaging between mipmaps and nearest neighbor filtering within each mipmap. For even better results, at the cost of greater inefficiency, you can use *GL.GL_LINEAR_MIPMAP_LINEAR*, which does averaging both between and within mipmaps. (You can research the remaining two options on your own if you are curious.)

One very important note: If you are **not** using mipmaps for a texture, it is imperative that you change the minification filter for that texture to *GL_NEAREST* or, more likely, *GL_LINEAR.* The default MIN filter **requires** mipmaps, and if mipmaps are not available, then the texture is considered to be improperly formed, and OpenGL ignores it!

4.5.3 Texture Transformations

Recall that textures are applied to objects using texture coordinates. The texture coordinates for a vertex determine which point in a texture is mapped to that vertex. Texture coordinates can be specified using the *glTexCoord** families of methods. Textures are most often images, which are two-dimensional, and the two coordinates on a texture image are referred to as s and t. Since OpenGL also supports one-dimensional textures and three-dimensional textures, texture coordinates cannot be restricted to two coordinates. In fact, a set of texture coordinates in OpenGL is represented internally as homogeneous coordinates (see Subsection 3.1.4), which are referred to as (s,t,r,q). We have used *glTexCoord2d* to specify texture s and t coordinates, but a call to *gl.glTexCoord2d*(s,t) is really just shorthand for *gl.glTexCoord4d*$(s,t,0,1)$.

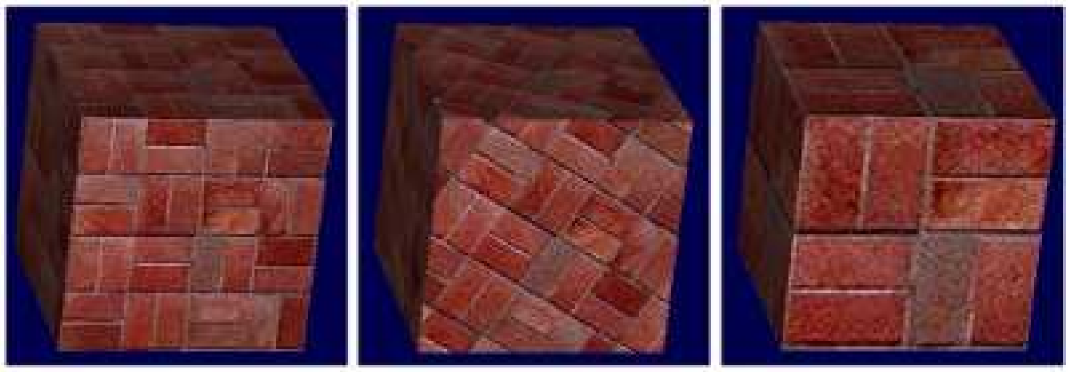
Since texture coordinates are no different from vertex coordinates, they can be transformed in exactly the same way. OpenGL maintains a ***texture transformation matrix*** as part of its state, along with the modelview matrix and projection matrix. When a texture is applied to an object, the texture coordinates that were specified for its vertices are transformed by the texture matrix. The transformed texture coordinates are then used to pick out a point in the texture. Of course, the default texture transform is the identity, which has no effect.

The texture matrix can represent scaling, rotation, translation and combinations of these basic transforms. To specify a texture transform, you have to use *glMatrixMode* to set the matrix mode to *GL_TEXTURE*. With this mode in effect, calls to methods such as *glRotated*, *glScalef*, and *glLoadIdentity* are applied to the texture matrix. For example to install a texture transform that scales texture coordinates by a factor of two in each direction, you could say:

```
gl.glMatrixMode(GL.GL_TEXTURE);
gl.glLoadIdentity(); // Make sure we are starting from the identity matrix.
gl.glScaled(2,2,2);
gl.glMatrixMode(GL.GL_MODELVIEW); // Leave matrix mode set to GL_MODELVIEW.
```

Now, what does this actually mean for the appearance of the texture on a surface? This scaling transforms multiplies each texture coordinate by 2. For example, if a vertex was assigned 2D texture coordinates (0.4,0.1), then that vertex will be mapped, after the texture transform is applied, to the point $(s,t) = (0.8,0.2)$ in the texture. The texture coordinates vary *twice as fast* on the surface as they would without the scaling transform. A region on the surface that would map to a 1-by-1 square in the texture image without the transform will instead map to a 2-by-2 square in the image—so that a larger piece of the image will be seen inside the region. In other words, the texture image will be *shrunk* by a factor of two on the surface! More generally, the effect of a texture transformation on the appearance of the texture is the **inverse** of its effect on the texture coordinates. (This is exactly analogous to the inverse relationship between a viewing transformation and a modeling transformation.) If the texture transform is translation to the right, then the texture moves to the left on the surface. If the texture transform is a counterclockwise rotation, then the texture rotates clockwise on the surface.

The following image shows a cube with no texture transform, with a texture transform given by a rotation about the center of the texture, and with a texture transform that scales by a factor of 0.5:



These pictures are from the sample program *TextureAnimation.java*. You can find an applet version of that program on-line.

4.5.4 Creating Textures with OpenGL

Texture images for use in an OpenGL program usually come from an external source, most often an image file. However, OpenGL is itself a powerful engine for creating images. Sometimes, instead of loading an image file, it's convenient to have OpenGL create the image internally, by rendering it. This is possible because OpenGL can read texture data from its own color buffer, where it does its drawing. To create a texture image using OpenGL, you just have to draw the image using standard OpenGL drawing commands and then load that image as a texture using the method

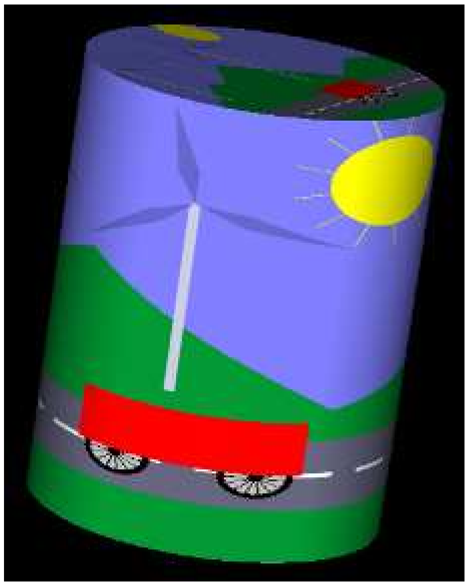
```
gl.glCopyTexImage2D( target, mipmapLevel, internalFormat,
                                     x, y, width, height, border );
```

In this method, *target* will be *GL.GL_TEXTURE_2D* except for advanced applications; *mipmapLevel*, which is used when you are constructing each mipmap in a set of mipmaps by hand, should be zero; the *internalFormat*, which specifies how the texture data should be stored, will ordinarily be *GL.GL_RGB* or *GL.GL_RGBA*, depending on whether you want to store an alpha component for each texel; *x* and *y* specify the lower left corner of the rectangle in the color buffer from which the texture will be read and are usually 0; *width* and *height* are the size of that rectangle; and *border*, which makes it possible to include a border around the texture image for certain special purposes, will ordinarily be 0. That is, a call to *glCopyTexImage2D* will typically look like

```
gl.glCopyTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB, 0, 0, width, height, 0);
```

As usual with textures, the *width* and *height* should ordinarily be powers of two, although non-power-of-two textures are supported if the OpenGL version is 2.0 or higher.

As an example, the sample program *TextureFromColorBuffer.java* uses this technique to produce a texture. The texture image in this case is a copy of the two-dimensional hierarchical graphics example from Subsection 2.1.4. Here is what this image looks like when the program uses it as a texture on a cylinder:



The texture image in this program can be animated. For each frame of the animation, the program draws the current frame of the 2D animation, then grabs that image for use as a texture. It does this in the *display*() method, even though the 2D image that is draws is not shown. After drawing the image and grabbing the texture, the program erases the image and draws a 3D textured object, which is the only thing that the user gets to see in the end. It's worth looking at that display method, since it requires some care to use a power-of-two texture size and to set up lighting only for the 3D part of the rendering process:

```
public void display(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();

    int[] viewPort = new int[4];  // The current viewport; x and y will be 0.
    gl.glGetIntegerv(GL.GL_VIEWPORT, viewPort, 0);
    int textureWidth = viewPort[2];  // The width of the texture.
    int textureHeight = viewPort[3]; // The height of the texture.

    /* First, draw the 2D scene into the color buffer. */

    if (version_2_0) {
           // Non-power-of-two textures are supported.  Use the entire
           // view area for drawing the 2D scene.
        draw2DFrame(gl); // Draws the animated 2D scene.
    }
    else {
           // Use a power-of-two texture image. Reset the viewport
           // while drawing the image to a power-of-two-size,
           // and use that size for the texture.
        gl.glClear(GL.GL_COLOR_BUFFER_BIT);
        textureWidth = 1024;
        while (textureWidth > viewPort[2])
            textureWidth /= 2; // Use a power of two that fits in the viewport.
        textureHeight = 512;
        while (textureWidth > viewPort[3])
            textureHeight /= 2; // Use a power of two that fits in the viewport.
        gl.glViewport(0,0,textureWidth,textureHeight);
        draw2DFrame(gl);  // Draws the animated 2D scene.
        gl.glViewport(0, 0, viewPort[2], viewPort[3]);  // Restore full viewport.
    }

    /* Grab the image from the color buffer for use as a 2D texture. */

    gl.glCopyTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGBA,
            0, 0, textureWidth, textureHeight, 0);

    /* Set up 3D viewing, enable 2D texture,
       and draw the object selected by the user. */

    gl.glPushAttrib(GL.GL_LIGHTING_BIT | GL.GL_TEXTURE_BIT);

    gl.glEnable(GL.GL_LIGHTING);
    gl.glEnable(GL.GL_LIGHT0);
    float[] dimwhite = { 0.4f, 0.4f, 0.4f };
    gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPECULAR, dimwhite, 0);
    gl.glEnable(GL.GL_DEPTH_TEST);
    gl.glShadeModel(GL.GL_SMOOTH);
    if (version_1_2)
        gl.glLightModeli(GL.GL_LIGHT_MODEL_COLOR_CONTROL,
                                            GL.GL_SEPARATE_SPECULAR_COLOR);
    gl.glLightModeli(GL.GL_LIGHT_MODEL_LOCAL_VIEWER, GL.GL_TRUE);
```

```
        gl.glClearColor(0,0,0,1);
        gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
        camera.apply(gl);

        /* Since we don't have mipmaps, we MUST set the MIN filter
         * to a non-mipmapped version; leaving the value at its default
         * will produce no texturing at all! */
        gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR);

        gl.glEnable(GL.GL_TEXTURE_2D);

        float[] white = { 1, 1, 1, 1 }; // Use white material for texturing.
        gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_AMBIENT_AND_DIFFUSE, white, 0);
        gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_SPECULAR, white, 0);
        gl.glMateriali(GL.GL_FRONT_AND_BACK, GL.GL_SHININESS, 128);
        int selectedObject = objectSelect.getSelectedIndex();
               // selectedObject Tells which of several object to draw.
        gl.glRotated(15,3,2,0); // Apply some viewing transforms to the object.
        gl.glRotated(90,-1,0,0);
        if (selectedObject == 1 || selectedObject == 3)
            gl.glTranslated(0,0,-1.25);
        objects[selectedObject].render(gl);

        gl.glPopAttrib();

    }
```

4.5.5 Loading Data into a Texture

Although OpenGL can draw its own textures, most textures come from external sources. The data can be loaded from an image file, it can be taken from a *BufferedImage*, or it can even be computed by your program on-the-fly. Using Java's *Texture* class is certainly the easiest way to load existing images. However, it's good to also know how to load texture data using only basic OpenGL commands.

To load external data into a texture, you have to store the color data for that texture into a Java nio *Buffer* (or, if using the C API, into an array). The data must specify color values for each texel in the texture. Several formats are possible, but the most common are *GL.GL_RGB*, which requires a red, a blue, and a green component value for each texel, and *GL.GL_RGBA*, which adds an alpha component for each texel. You need one number for each component, for every texel. When using *GL.GL_RGB* to specify a texture with n texels, you need a total of $3*n$ numbers. Each number is typically an unsigned byte, with a value in the range 0 to 255, although other types of data can be used as well.

(The use of unsigned bytes is somewhat problematic in Java, since Java's **byte** data typed is signed, with values in the range -128 to 127. Essentially, the negative numbers are reinterpreted as positive numbers. Usually, the safest approach is to use an **int** or **short** value and type-cast it to **byte**.)

Once you have the data in a *Buffer*, you can load that data into a 2D texture using the *glTexImage2D* method:

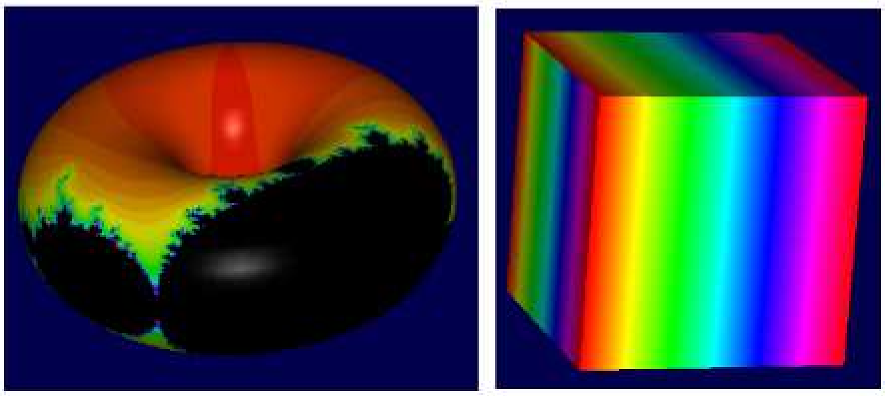
```
    gl.glTexImage2D(target, mipmapLevel, internalFormat, width, height, border,
                                format, dataType, buffer);
```

The first six parameters are similar to parameters in the *glCopyTexImage2D* method, as discussed in the previous subsection. The other three parameters specify the data. The format is *GL.GL_RGB* if you are providing RGB data and is *GL.GL_RGBA* for RGBA data. Other formats are also possible. Note that the *format* and the *internalFormat* are often the same, although they don't have to be. The *dataType* tells what type of data is in the buffer and is usually *GL.GL_UNSIGNED_BYTE*. Given this data type, the buffer should be of type *ByteBuffer*. The number of bytes in the buffer must be 3**width***height* for RGB data and 4**width***height* for RGBA data.

It is also possible to load a one-dimensional texture in a similar way. The *glTexImage1D* method simply omits the *height* parameter. As an example, here is some code that creates a one-dimensional texture consisting of 256 texels that vary in color through a full spectrum of color:

```
ByteBuffer textureData1D = BufferUtil.newByteBuffer(3*256);
for (int i = 0; i < 256; i++) {
    Color c = Color.getHSBColor(1.0f/256 * i, 1, 1); // A color of the spectrum.
    textureData1D.put((byte)c.getRed());  // Add color components to the buffer.
    textureData1D.put((byte)c.getGreen());
    textureData1D.put((byte)c.getBlue());
}
textureData1D.rewind();
gl.glTexImage1D(GL.GL_TEXTURE_1D, 0, GL.GL_RGB, 256, 0,
                          GL.GL_RGB, GL.GL_UNSIGNED_BYTE, textureData1D);
```

This code is from the sample program *TextureLoading.java*, which also includes an example of a two-dimensional texture created by computing the individual texel colors. The two dimensional texture is the famous Mandelbrot set. Here are two images from that program, one showing the one-dimensional spectrum texture on a cube and the other showing the two-dimensional Mandelbrot texture on a torus. (Note, by the way, how the strong specular highlight on the black part of the Mandelbrot set adds to the three-dimensional appearance of this image.) You can find an applet version of the program in the on-line version of this section.



4.5.6 Texture Coordinate Generation

Texture coordinates are typically specified using the *glTexCoord** family of methods or by using texture coordinate arrays with *glDrawArrays* and *glDrawElements*. However, computing texture coordinates can be tedious. OpenGL is capable of generating certain types of texture coordinates on its own. This is especially useful for so-called "reflection maps" or "environment maps," where texturing is used to imitate the effect of an object that reflects its environment. OpenGL can generate the texture coordinates that are needed for this effect. However, environment mapping is an advanced topic that I will not cover here. Instead, we look at a simple case: object-linear coordinates.

With object-linear texture coordinate generation, OpenGL uses texture coordinates that are computed as linear functions of object coordinates. Object coordinates are just the actual coordinates specified for vertices, with *glVertex** or in a vertex array. The default when object-linear coordinate generation is turned on is to make the object coordinates equal to the texture coordinates. For two-dimensional textures, for example,

```
gl.glVertex3f(x,y,z);
```

would be equivalent to

```
gl.glTexCoord2f(x,y);
gl.glVertex3f(x,y,z);
```

However, it is possible to compute the texture coordinates as arbitrary linear combinations of the vertex coordinates x, y, z, and w. Thus, *gl.glVertex4f(x,y,z,w)* becomes equivalent to

```
gl.glTexCoord2f(a*x + b*y + c*z + d*w, e*x + f*y + g*z + h*w);
gl.glVertex4f(x,y,z,w);
```

where (a,b,c,d) and (e,f,g,h) are arbitrary arrays.

To use texture generation, you have to enable and configure it for each texture coordinate separately. For two-dimensional textures, you want to enable generation of the s and t texture coordinates:

```
gl.glEnable(GL.GL_TEXTURE_GEN_S);
gl.glEnable(GL.GL_TEXTURE_GEN_T);
```

To say that you want to use object-linear coordinate generation, you can use the method *glTexGeni* to set the texture generation "mode" to object-linear for both s and t:

```
gl.glTexGeni(GL.GL_S, GL.GL_TEXTURE_GEN_MODE, GL.GL_OBJECT_LINEAR);
gl.glTexGeni(GL.GL_T, GL.GL_TEXTURE_GEN_MODE, GL.GL_OBJECT_LINEAR);
```

If you accept the default behavior, the effect will be to project the texture onto the surface from the xy-plane (in the coordinate system in which the coordinates are specified, before any transformation is applied). If you want to change the equations that are used, you can specify the coordinates using *glTexGenfv*. For example, to use coefficients (a,b,c,d) and (e,f,g,h) in the equations:

```
gl.glTexGenfv(GL.GL_S, GL.GL_OBJECT_PLANE, new float[] { a,b,c,d }, 0);
gl.glTexGenfv(GL.GL_T, GL.GL_OBJECT_PLANE,  new float[] { e,f,g,h }, 0);
```

The sample program *TextureCoordinateGeneration.java* demonstrates the use of texture coordinate generation. It allows the user to enter the coefficients for the linear equations that are used to generate the texture coordinates. The same program also demonstrates "eye-linear" texture coordinate generation, which is similar to the object-linear version but uses eye coordinates instead of object coordinates in the equations; I won't discuss it further here. As usual, you can find an applet version on-line.

4.5.7 Texture Objects

For our final word on textures, we look briefly at ***texture objects***. Texture objects are used when you need to work with several textures in the same program. The usual method for loading textures, *glTexImage**, transfers data from your program into the graphics card. This is an expensive operation, and switching among multiple textures by using this method can seriously degrade a program's performance. Texture objects offer the possibility of storing texture data for multiple textures on the graphics card and to switch from one texture object to another with a single, fast OpenGL command. (Of course, the graphics card has only a limited amount of memory for storing textures, and texture objects that don't fit in the graphics card's memory are no more efficient than ordinary textures.)

Note that if you are using Java's *Texture* class to represent your textures, you won't need to worry about texture objects, since the *Texture* class handles them automatically. If *tex* is of type *Texture*, the associated texture is actually stored as a texture object. The method *tex.bind()* tells OpenGL to start using that texture object. (It is equivalent to *gl.glBindTexture(tex.getTarget(),tex.getTextureObject())*, where *glBindTexture* is a method that is discussed below.) The rest of this section tells you how to work with texture objects by hand.

Texture objects are similar in their use to vertex buffer objects, which were covered in Subsection 3.4.3. Like a vertex buffer object, a texture object is identified by an integer ID number. Texture object IDs are managed by OpenGL, and to obtain a batch of valid textures IDs, you can call the method

```
gl.glGenTextures(n, idList, 0);
```

where *n* is the number of texture IDs that you want, *idList* is an array of length at least *n* and of type **int**[] that will hold the texture IDs, and the 0 indicates the starting index in the array where the IDs are to be placed. When you are done with the texture objects, you can delete them by calling *gl.glDeleteTextures(n,idList,0)*.

Every texture object has its own state, which includes the values of texture parameters such as *GL_TEXTURE_WRAP_S* as well as the color data for the texture itself. To work with the texture object that has ID equal to *texID*, you have to call

```
gl.glBindTexture(target, texID)
```

where *target* is a texture target such as *GL.GL_TEXTURE_1D* or *GL.GL_TEXTURE_2D*. After this call, any use of *glTexParameter**, *glTexImage**, or *glCopyTexImage** with the same texture target will be applied to the texture object with ID *texID*. Furthermore, if the texture target is enabled and some geometry is rendered, then the texture that is applied to the geometry is the one associated with that texture ID. A texture binding for a given target remains in effect until another texture object is bound to the same target. To switch from one texture to another, you simply have to call *glBindTexture* with a different texture object ID.

Chapter 5

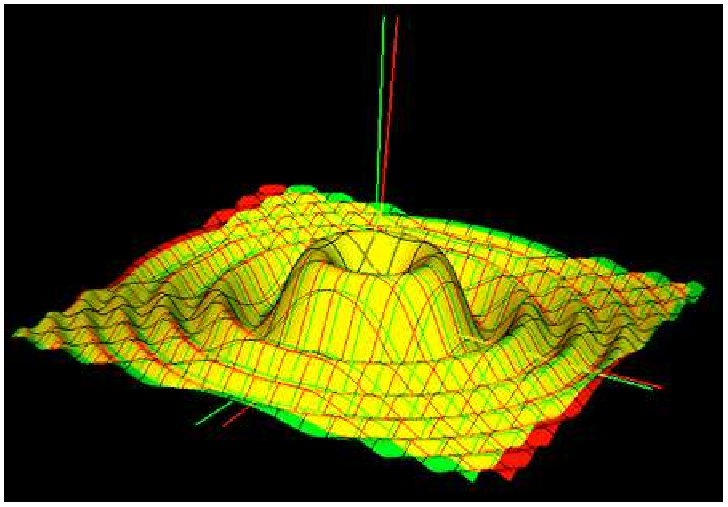
Some Topics Not Covered

This page contains a few examples that demonstrate topics not covered in Chapters 1 through 4. The source code for the examples might be useful to people who want to learn more. The next version of this book should cover these topics—and more— in detail.

* * *

In ***stereoscopic viewing***, a slightly different image is presented to each eye. The images for the left eye and the right eye are rendered from slightly different viewing directions, imitating the way that a two-eyed viewer sees the real world. For many people, stereoscopic views can be visually fused to create a convincing illusion of 3D depth. One way to do stereoscopic viewing on a computer screen is to combine the view from the left eye, drawn in red, and the view from the right eye, drawn in green. To see the 3D effect, the image must be viewed with red/green (or red/blue or red/cyan) glasses designed for such 3D viewing. This type of 3D viewing is referred to as ***anaglyph*** (search for it on Google Images).

The sample program *Sterographer.java*, in the package *stereoGraph3d*, can render an anaglyph stereo view of the graph of a function. The program uses modified versions of the *Classname* and *TrackBall* classes that can be found in the same package. In order to combine the red and green images, the program uses the method *glColorMask* method. (The program is also a nice example of using vector buffer objects and *glDrawElements* for drawing primitives.) An applet version of the program can be found on-line; here is a screenshot:



* * *

Mouse interaction with a 3D scene is complicated by the fact that the mouse coordinates are given in device (pixel) coordinates, while the objects in the scene are created in object coordinates. This makes it difficult to tell which object the user is clicking on. OpenGL can help with this "selection" problem. It offers the *GL_SELECT* render mode to make selection easier. Unfortunately, it's still fairly complicated and I won't describe it here.

The sample program *SelectionDemo.java* in the package *selection* demonstrates the use of the *GL_SELECT* render mode. The program is a version of *WalkThroughDemo.java* where the user can click on objects to select it. The size of the selected object is animated to show that it is selected, but otherwise nothing special can be done with it. An applet version can be found on-line.

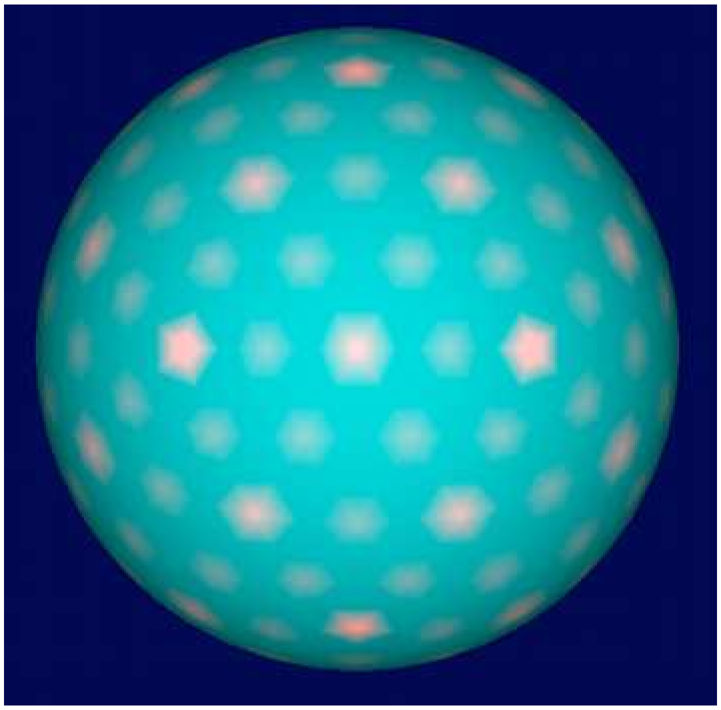
* * *

Throughout the text, we have been talking about the standard OpenGL "rendering pipeline." The operations performed by this pipeline are are actually rather limited when compared to the full range of graphics operations that are commonly used in modern computer graphics. And new techniques are constantly being discovered. It would be impossible to make every new technique a standard part of OpenGL. To help with this problem, OpenGL 2.0 introduced the ***OpenGL Shading Language*** (GLSL). Parts of the OpenGL rendering pipeline can be replaced with programs written in GLSL. GLSL programs are referred to as ***shaders***. A ***vertex shader*** written in OpenGL can replace the part of the pipeline that does lighting, transformation and other operations on each vertex. A ***fragment shader*** can replace the part that operates on each pixel in a primitive. GLSL is a complete programming language, based on C, which makes GLSL shaders very versitile. In the newest versions of OpenGL, shaders are preferred over the standard OpenGL processing.

Since the OpenGL API for working with shaders is rather complicated, I wrote the class *GLSLProgram* to represent a GLSL program (source code *GLSLProgram.java* in package *glsl*). The sample programs *MovingLightDemoGLSL.java* and *IcosphereIFS_GLSL.java*, both in package *glsl*, demonstrate the use of simple GLSL programs with the *GLSLProgram* class. These examples are not meant as a demonstration of what GLSL can do; they just show how to use GLSL with Java. The GLSL programs can only be used if the version of OpenGL is 2.0 or higher; the program will run with lower version numbers, but the GLSL programs will not be applied.

The first sample program, *MovingLightDemoGLSL.java* is a version of *MovingLight-Demo.java* that allows the user to turn a GLSL program on and off. The GLSL program consists of a fragment shader that converts all pixel colors to gray scale. (Note, by the way, that the conversion applies only to pixels in primitives, not to the background color that is used by *glClear*). An applet version can be found on-line.

The second sample program, *IcosphereIFS_GLSL.java*, uses both a fragment shader and a vertex shader. This example is a version of *IcosphereIFS.java*, which draws polyhdral approximations for a sphere by subdividing an icosahedron. The vertices of the icosphere are generated by a recursive algorithm, and the colors in the GLSL version are meant to show the level of the recursion at which each vertex is generated. (The effect is not as interesting as I had hoped.) As usual, an applet version is on-line; here is a screenshot:



Appendix: Source Files

This is a list of source code files for examples in *Fundamentals of Computer Graphics with Java, OpenGL, and Jogl*.

- *QuadraticBezierEdit.java* and *QuadraticBezierEditApplet.java*, from Subsection 1.2.2. A program that demonstrates quadratic editing curves and allows the user to edit them.
- *CubicBezierEdit.java* and *CubicBezierEditApplet.java*, from Subsection 1.2.2. A program that demonstrates quadratic editing curves and allows the user to edit them.
- *HierarchicalModeling2D.java* and *HierarchicalModeling2DApplet.java*, from Subsection 1.3.2. A program that shows an animation constructed using hierarchical modeling with *Graphics2D* transforms.
- *BasicJoglApp2D.java*, from Subsection 2.1.1. An OpenGL program that just draws a triangle, showing how to use a *GLJPanel* and *GLEventListener*.
- *BasicJoglAnimation2D.java* and *BasicJoglAnimation2DApplet.java*, from Subsection 2.1.3. A very simple 2D OpenGL animation, using a rotation transform to rotate a triangle.
- *JoglHierarchicalModeling2D.java* and *JoglHierarchicalModeling2DApplet.java*, from Subsection 2.1.3. An animated 2D scene using hierarchical modeling OpenGL. This program is pretty much a port of *HierarchicalModeling2D.java*, which used Java *Graphics2D* instead of OpenGL.
- *JoglHMWithSceneGraph2D.java*, from Subsection 2.1.5. Another version of the hierarchical modeling animation, this one using a scene graph to represent the scene. The scene graph is built using classes from the source directory *scenegraph2D*.
- *Axes3D.java*, used for an illustration in Section 2.2. A very simple OpenGL 3D program that draws a set of axes. The program uses GLUT to draw the cones and cylinders that represent the axes.
- *LitAndUnlitSpheres.java*, used for an illustration in Subsection 2.2.2. A very simple OpenGL 3D program that draws four spheres with different lighting settings.
- *glutil*, introduced in Section 2.3 is a package that contains several utility classes including *glutil/Camera.java* for working with the projection and view transforms; *glutil/TrackBall.java*, for implementing mouse dragging to rotate the view; and *glutil/UVSphere.java*, *glutil/UVCone.java*, and *glutil/UVCylinder.java* for drawing some basic 3D shapes.
- *PaddleWheels.java* and *PaddleWheelsApplet.java*, from Subsection 2.3.1. A first example of modeling in 3D. A simple animation of three rotating "paddle wheels." The user can rotate the image by dragging the mouse (as will be true for most examples from now on). This example depends on several classes from the package *glutil*
- *ColorCubeOfSpheres.java* and *ColorCubeOfSpheresApplet.java*, from Subsection 2.3.2. Draws a lot of spheres of different colors, arranged in a cube. The point is to do a

lot of drawing, and to see how much the drawing can be sped up by using a display list. The user can turn the display list on and off and see the effect on the rendering time. This example depends on several classes from the package *glutil*.

- *TextureDemo.java* and *TextureDemoApplet.java*, from Subsection 2.4.2. Shows six textured objects, with various shapes and textures. The user can rotate each object individually. This example depends on several classes from the package *glutil* and on textures from *textures*.
- *PrimitiveTypes.java* and *PrimitiveTypesApplet.java*, from Subsection 3.2.5. A 2D program that lets the user experiment with the ten OpenGL primitive types and various options that affect the way they are drawn.
- *VertexArrayDemo.java* and *VertexArrayDemoApplet.java*, from Section 3.4. Uses vertex arrays and, in OpenGL 1.5 or higher, vertex buffer objects to draw a cylinder inside a sphere, where the sphere is represented as a random cloud of points.
- *IcosphereIFS.java* and *IcosphereIFSApplet.java*, from Subsection 3.4.4. Demonstrates the use of *glDrawElements*, with or without vertex buffer objects to draw indexed face sets.
- *WalkThroughDemo.java* and *WalkThroughDemoApplet.java*, from Subsection 3.5.4. The user navigates through a simple 3D world using the arrow keys. The user's point of view is represented by an object of type *SimpleAvatar*, from the *glutil* package. The program uses several shape classes from the same package, as well as a basic implementation of 3D scene graphs found in the package *simplescenegraph3d*.
- *MovingCameraDemo.java* and *MovingCameraDemoApplet.java*, from Subsection 3.5.5. The program uses the *simplescenegraph3d* package to implement a scene graph that includes two *AvatarNodes*. These nodes represent the view of the scene from two viewers that are embedded in the scene as part of the scene graph. The program also uses several classes from the *glutil* package.
- *LightDemo.java* and *LightDemoApplet.java*, from Subsection 4.3.1. Demonstrates some light and material properties and setting light positions. Requires *Camera* and *TrackBall* from the *glutil* package.
- *MovingLightDemo.java* and *MovingLightDemoApplet.java*, from Subsection 4.4.2. The program uses the *scenegraph3d* package, a more advanced version of *scenegraph3d*, to implement a scene graph that uses two *LightNodes* to implement moving lights. The program also uses several classes from the *glutil* package.
- There are four examples in Section 4.5 that deal with textures: *TextureAnimation.java*, *TextureFromColorBuffer.java*, *TextureLoading.java*, and *TextureCoordinateGeneration.java* (and their associated applet classes). All of these examples use the *glutil* package.
- There are four examples in Chapter 5, a section at the end of the book that describes a few topices that not covered in the book proper. *StereoGrapher.java* demonstrates a form of stereo rendering that requires red/green 3D glasses; this program uses modified versions of the Trackball and Camera classes, which can be found in the *stereoGraph3D* package. *SelectionDemo.java* demonstrates using OpenGL's *GL_SELECT* render mode to let the user "pick" or "select" items in a 3D scene using the mouse; this demo is a modification of *WalkThroughDemo.java* and requires the packages *glutil* and *simplescenegraph3d*. Finally, *MovingLightDemoGLSL.java* and *IcosphereIFS_GLSL.java* demonstrate the use of "shaders" written in the OpenGL Shading Language (GLSL); these examples

are modifications of *MovingLightDemo.java* and *IcosphereIFS.java*. The GLSL programs in these examples are very simple. The examples use a class, *GLSLProgram.java*, that is meant to coordinate the use of GLSL programs. *MovingLightDemoGLSL* requires the packages *glutil* and *scenegraph3d*.