3-D Computer Graphics

A Mathematical Introduction with OpenGL

SAMUEL R. BUSS
University of California, San Diego

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typefaces Times New Roman PS 10/12 pt. *System* LaTeX 2$_\varepsilon$ [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Buss, Samuel R.
3D computer graphics : a mathematical introduction with OpenGL / Samuel R. Buss.
p. cm.
Includes bibliographical references and index.
ISBN 0-521-82103-7
1. Computer graphics. 2. OpenGL. 3. Three-dimensional display systems. I. Title.
T385 .B8695 2003

006.6′93 – dc21 2002034804

ISBN 0 521 82103 7 hardback

Contents

Color art appears following page 256.

I

Introduction

This chapter discusses some of the basic concepts behind computer graphics with particular emphasis on how to get started with simple drawing in OpenGL. A major portion of the chapter explains the simplest methods of drawing in OpenGL and various rendering modes. If this is your first encounter with OpenGL, it is highly suggested that you look at the included sample code and experiment with some of the OpenGL commands while reading this chapter.

The first topic considered is the different models for graphics displays. Of particular importance for the topics covered later in the book is the idea that an arbitrary three-dimensional geometrical shape can be approximated by a set of polygons – more specifically as a set of triangles. Second, we discuss some of the basic methods for programming in OpenGL to display simple two- and three-dimensional models made from points, lines, triangles, and other polygons. We also describe how to set colors and polygonal orientations, how to enable hidden surface removal, and how to make animation work with double buffering. The included sample OpenGL code illustrates all these capabilities. Later chapters will discuss how to use transformations, how to set the viewpoint, how to add lighting and shading, how to add textures, and other topics.

I.1 Display Models

We start by describing three models for graphics display modes: (1) drawing points, (2) drawing lines, and (3) drawing triangles and other polygonal patches. These three modes correspond to different hardware architectures for graphics display. Drawing points corresponds roughly to the model of a graphics image as a rectangular array of pixels. Drawing lines corresponds to vector graphics displays. Drawing triangles and polygons corresponds to the methods used by modern graphics display hardware for displaying three-dimensional images.

I.1.1 Rectangular Arrays of Pixels

The most common low-level model is to treat a graphics image as a rectangular array of pixels in which, each pixel can be independently set to a different color and brightness. This is the display model used for cathode ray tubes (CRTs) and televisions, for instance. If the pixels are small enough, they cannot be seen individually by the human viewer, and the image, although composed of points, can appear as a single smooth image. This technique is used in art as well – notably in mosaics and, even more so, in pointillism, where pictures are composed of small

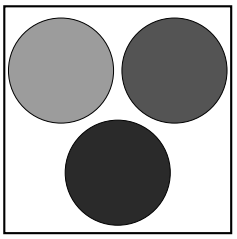


Figure I.1. A pixel is formed from subregions or subpixels, each of which displays one of three colors. See Color Plate 1.

patches of solid color but appear to form a continuous image when viewed from a sufficient distance.

Keep in mind, however, that the model of graphics images as a rectangular array of pixels is only a convenient abstraction and is not entirely accurate. For instance, on a CRT or television screen, each pixel actually consists of three separate points (or dots of phosphor): each dot corresponds to one of the three primary colors (red, blue, and green) and can be independently set to a brightness value. Thus, each pixel is actually formed from three colored dots. With a magnifying glass, you can see the colors in the pixel as separate colors (see Figure I.1). (It is best to try this with a low-resolution device such as a television; depending on the physical design of the screen, you may see the separate colors in individual dots or in stripes.)

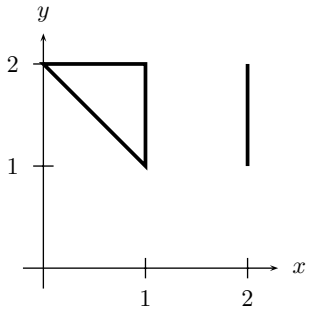
A second aspect of rectangular array model inaccuracy is the occasional use of subpixel image addressing. For instance, laser printers and ink jet printers reduce aliasing problems, such as jagged edges on lines and symbols, by micropositioning toner or ink dots. More recently, some handheld computers (i.e., palmtops) are able to display text at a higher resolution than would otherwise be possible by treating each pixel as three independently addressable subpixels. In this way, the device is able to position text at the subpixel level and achieve a higher level of detail and better character formation.

In this book however, issues of subpixels will never be examined; instead, we will always model a pixel as a single rectangular point that can be set to a desired color and brightness. Sometimes the pixel basis of a computer graphics image will be important to us. In Section II.4, we discuss the problem of approximating a straight sloping line with pixels. Also, when using texture maps and ray tracing, one must take care to avoid the aliasing problems that can arise with sampling a continuous or high-resolution image into a set of pixels.

We will usually not consider pixels at all but instead will work at the higher level of polygonally based modeling. In principle, one could draw any picture by directly setting the brightness levels for each pixel in the image; however, in practice this would be difficult and time consuming. Instead, in most high-level graphics programming applications, we do not have to think very much about the fact that the graphics image may be rendered using a rectangular array of pixels. One draws lines, or especially polygons, and the graphics hardware handles most of the work of translating the results into pixel brightness levels. A variety of sophisticated techniques exist for drawing polygons (or triangles) on a computer screen as an array of pixels, including methods for shading and smoothing and for applying texture maps. These will be covered later in the book.

I.1.2 Vector Graphics

In traditional vector graphics, one models the image as a set of lines. As such, one is not able to model solid objects, and instead draws two-dimensional shapes, graphs of functions,




```
penup();
moveto(2,2);
pendown();
moveto(2,1);
penup();
moveto(1,2);
pendown();
moveto(0,2);
moveto(1,1);
moveto(1,2);
```

Figure I.2. Examples of vector graphics commands.

or wireframe images of three-dimensional objects. The canonical example of vector graphics systems are pen plotters; this includes the "turtle geometry" systems. Pen plotters have a drawing pen that moves over a flat sheet of paper. The commands available include (a) *pen up*, which lifts the pen up from the surface of the paper, (b) *pen down*, which lowers the point of the pen onto the paper, and (c) *move-to*(x, y), which moves the pen in a straight line from its current position to the point with coordinates $\langle x, y \rangle$. When the pen is up, it moves without drawing; when the pen is down, it draws as it moves (see Figure I.2). In addition, there may be commands for switching to a different color pen as well as convenience commands to make it easier to draw images.

Another example of vector graphics devices is vector graphics display terminals, which traditionally are monochrome monitors that can draw arbitrary lines. On these vector graphics display terminals, the screen is a large expanse of phosphor and does not have pixels. A traditional oscilloscope is also an example of a vector graphics display device.

Vector graphics displays and pixel-based displays use very different representations of images. In pixel-based systems, the screen image will be stored as a bitmap, namely, as a table containing all the pixel colors. A vector graphics system, on the other hand, will store the image as a list of commands – for instance as a list of pen up, pen down, and move commands. Such a list of commands is called a display list.

Nowadays, pixel-based graphics hardware is very prevalent, and thus even graphics systems that are logically vector based are typically displayed on hardware that is pixel based. The disadvantage is that pixel-based hardware cannot directly draw arbitrary lines and must approximate lines with pixels. On the other hand, the advantage is that more sophisticated figures, such as filled regions, can be drawn.

Modern vector graphics systems incorporate more than just lines and include the ability to draw curves, text, polygons, and other shapes such as circles and ellipses. These systems also have the ability to fill in or shade a region with a color or a pattern. They generally are restricted to drawing two-dimensional figures. Adobe's PostScript language is a prominent example of a modern vector graphics system.

I.1.3 Polygonal Modeling

One step up, in both abstraction and sophistication, is the polygonal model of graphics images. It is very common for three-dimensional geometric shapes to be modeled first as a set of polygons and then mapped to polygonal shapes on a two-dimensional display. The basic display hardware is generally pixel based, but most computers now have special-purpose graphics hardware for processing polygons or, at the very least, triangles. Graphics hardware for rendering triangles

is also used in modern computer game systems; indeed, the usual measure of performance for graphics hardware is the number of triangles that can be rendered per second. At the time this book is being written, nominal peak performance rates of relatively cheap hardware is well above one million polygons per second!

Polygonal-based modeling is used in nearly every three-dimensional computer graphics systems. It is a central tool for the generation of interactive three-dimensional graphics and is used for photo-realistic rendering, including animation in movies.

The essential operation in a polygonal modeling system is drawing a single triangle. In addition, there are provisions for coloring and shading the triangle. Here, "shading" means varying the color across the triangle. Another important tool is the use of texture mapping, which can be used to paint images or other textures onto a polygon. It is very typical for color, shading, and texture maps to be supported by special-purpose hardware such as low-cost graphics boards on PCs.

The purpose of these techniques is to make polygonally modeled objects look more realistic. Refer to Figure III.1 on page 68. You will see six models of a teapot. Part (a) of the figure shows a wireframe teapot, as could be modeled on a vector graphics device. Part (b) shows the same shape but filled in with solid color; the result shows a silhouette with no three-dimensionality. Parts (c) through (f) show the teapot rendered with lighting effects: (c) and (e) show flat-shaded (i.e., unshaded) polygons for which the polygonal nature of the teapot is clearly evident; parts (d) and (f) incorporate shading in which the polygons are shaded with color that varies across the polygons. The shading does a fairly good job of masking the polygonal nature of the teapot and greatly increases the realism of the image.

I.2 Coordinates, Points, Lines, and Polygons

The next sections discuss some of the basic conventions of coordinate systems and of drawing points, lines, and polygons. Our emphasis will be on the conventions and commands used by OpenGL. For now, only drawing vertices at fixed positions in the xy-plane or in xyz-space is discussed. Chapter II will explain how to move vertices and geometric shapes around with rotations, translations, and other transformations.

I.2.1 Coordinate Systems

When graphing geometric shapes, one determines the position of the shape by specifying the positions of a set of vertices. For example, the position and geometry of a triangle are specified in terms of the positions of its three vertices. Graphics programming languages, including OpenGL, allow you to set up your own coordinate systems for specifying positions of points; in OpenGL this is done by specifying a function from your coordinate system into the screen coordinates. This allows points to be positioned at locations in either 2-space ($\mathbb{R}^2$) or 3-space ($\mathbb{R}^3$) and to have OpenGL automatically map the points into the proper location in the graphics image.

In the two-dimensional xy-plane, also called $\mathbb{R}^2$, a position is set by specifying its x- and y-coordinates. The usual convention (see Figure I.3) is that the x-axis is horizontal and pointing to the right and the y-axis is vertical and pointing upwards.

In three-dimensional space $\mathbb{R}^3$, positions are specified by triples $\langle a, b, c\rangle$ giving the x-, y-, and z-coordinates of the point. However, the convention for how the three coordinate axes are positioned is different for computer graphics than is usual in mathematics. In computer graphics, the x-axis points to the right, the y-axis points upwards, and the z-axis points toward the viewer. This is different from our customary expectations. For example, in calculus, the x-,

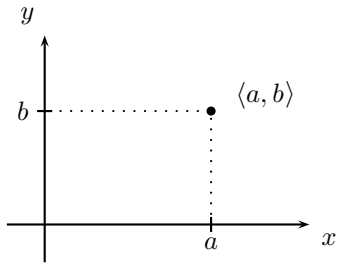


Figure I.3. The xy-plane, $\mathbb{R}^2$, and the point $\langle a, b\rangle$.

y-, and z-axes usually point forward, rightwards, and upwards (respectively). The computer graphics convention was adopted presumably because it keeps the x- and y-axes in the same position as for the xy-plane, but it has the disadvantage of taking some getting used to. Figure I.4 shows the orientation of the coordinate axes.

It is important to note that the coordinates axes used in computer graphics do form a right-handed coordinate system. This means that if you position your right hand with your thumb and index finger extended to make an L shape and place your hand so that your right thumb points along the positive x-axis and your index finger points along the positive y-axis, then your palm will be facing toward the positive z-axis. In particular, this means that the right-hand rule applies to cross products of vectors in $\mathbb{R}^3$.

I.2.2 Geometric Shapes in OpenGL

We next discuss methods for drawing points, lines, and polygons in OpenGL. We only give some of the common versions of the commands available in OpenGL. You should consult the OpenGL programming manual (Woo et al., 1999) for more complete information.

Drawing Points in OpenGL

OpenGL has several commands that define the position of a point. Two of the common ways to use these commands are[1]

```
glVertex3f(float x, float y, float z);
```

or

```
float v[3] = { x, y, z };
glVertex3fv( &v[0] );
```

The first form of the command, `glVertex3f`, specifies the point directly in terms of its x-, y-, and z-coordinates. The second form, `glVertex3fv`, takes a pointer to an array containing the coordinates. The "`v`" on the end of the function name stands for "vector." There are many other forms of the `glVertex*` command that can be used instead.[2] For instance, the "`f`,"

[1] We describe OpenGL commands with simplified prototypes (and often do not give the officially correct prototype). In this case, the specifiers "`float`" describe the types of the arguments to `glVertex3f()` but should be omitted in your `C` or `C++` code.

[2] There is no function named `glVertex*`: we use this notation to represent collectively the many variations of the `glVertex` commands.

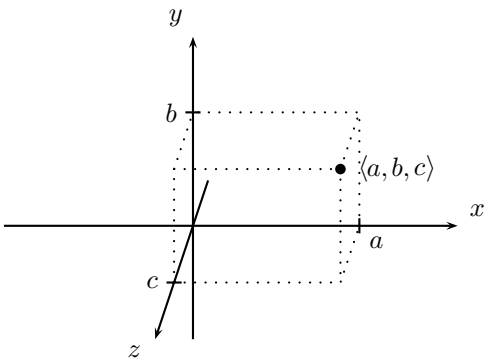


Figure I.4. The coordinate axes in $\mathbb{R}^3$ and the point $\langle a, b, c \rangle$. The z-axis is pointing toward the viewer.

which stands for "float," can be replaced by "`s`" for "short integer," by "`i`" for "integer," or by "`d`" for "double."[3]

For two-dimensional applications, OpenGL also allows you to specify points in terms of just x- and y-coordinates by using the commands

```
glVertex2f(float x, float y);
```

or

```
float v[2] = { x, y };
glVertex2fv( &v[0] );
```

`glVertex2f` is equivalent to `glVertex3f` but with $z = 0$.

All calls to `glVertex*` must be bracketed by calls to the OpenGL commands `glBegin` and `glEnd`. For example, to draw the three points shown in Figure I.5, you would use the commands

```
glBegin(GL_POINTS);
glVertex2f( 1.0, 1.0 );
glVertex2f( 2.0, 1.0 );
glVertex2f( 2.0, 2.0 );
glEnd();
```

The calls to the functions `glBegin` and `glEnd` are used to signal the start and end of drawing.

A sample OpenGL program, `SimpleDraw`, supplied with this text, contains the preceding code for drawing three points. If OpenGL is new to you, it is recommended that you examine the source code and try compiling and running the program. You will probably find that the points are drawn as very small, single-pixel points – perhaps so small as to be almost invisible. On most OpenGL systems, you can make points display as large, round dots by calling the following functions:

```
glPointSize(n);          // Points are n pixels in diameter
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

[3] To be completely accurate, we should remark that, to help portability and future compatibility, OpenGL uses the types `GLfloat`, `GLshort`, `GLint`, and `GLdouble`, which are generally defined to be the same as `float`, `short`, `int`, and `double`. It would certainly be better programming practice to use OpenGL's data types; however, the extra effort is not really worthwhile for casual programming.

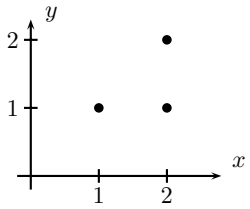


Figure I.5. Three points drawn in two dimensions.

(In the first line, a number such as 6 for n may give good results.) The `SimpleDraw` program already includes the preceding function calls, but they have been commented out. If you are lucky, executing these lines in the program before the drawing code will cause the program to draw nice round dots for points. However, the effect of these commands varies with different implementations of OpenGL, and thus you may see square dots instead of round dots or even no change at all.

The `SimpleDraw` program is set up so that the displayed graphics image is shown from the viewpoint of a viewer looking down the z-axis. In this situation, `glVertex2f` is a convenient method for two-dimensional graphing.

Drawing Lines in OpenGL

To draw a line in OpenGL, specify its endpoints. The `glBegin` and `glEnd` paradigm is still used. To draw individual lines, pass the parameter `GL_LINES` to `glBegin`. For example, to draw two lines, you could use the commands

```
glBegin( GL_LINES );
glVertex3f( x1, y1, z1 );
glVertex3f( x2, y2, z2 );
glVertex3f( x3, y3, z3 );
glVertex3f( x4, y4, z4 );
glEnd();
```

Letting $\mathbf{v}_i$ be the vertex $\langle x_i, y_i, z_i \rangle$, the commands above draw a line from $\mathbf{v}_1$ to $\mathbf{v}_2$ and another from $\mathbf{v}_3$ to $\mathbf{v}_4$. More generally, you may specify an even number, $2n$, of points, and the `GL_LINES` option will draw n lines connecting $\mathbf{v}_{2i-1}$ to $\mathbf{v}_{2i}$ for $i = 1, \ldots, n$.

You may also use `GL_LINE_STRIP` instead of `GL_LINES`: if you specify n vertices, a continuous chain of lines is drawn, namely, the lines connecting $\mathbf{v}_i$ and $\mathbf{v}_{i+1}$ for $i = 1, \ldots, n-1$. The parameter `GL_LINE_LOOP` can also be used; it draws the line strip plus the line connecting $\mathbf{v}_n$ to $\mathbf{v}_1$. Figure I.6 shows the effects of these three line-drawing modes.

The `SimpleDraw` program includes code to draw the images in Figure I.6. When the program is run, you may find that the lines look much too thin and appear jagged because they

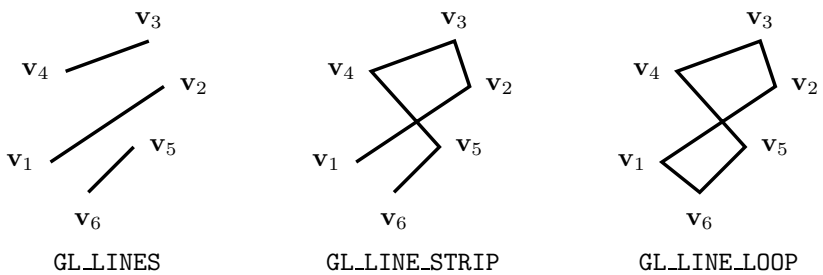


Figure I.6. The three line-drawing modes as controlled by the parameter to `glBegin`.

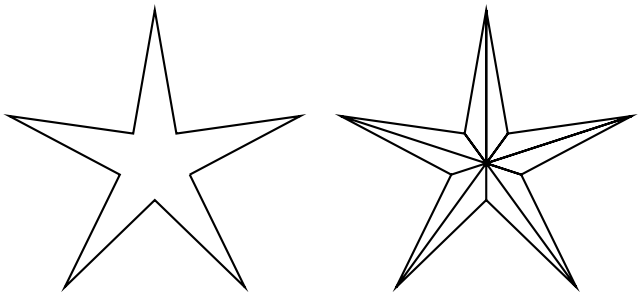


Figure I.7. Figures for Exercises I.2, I.3, and I.4.

were drawn only one pixel wide. By default, OpenGL draws thin lines, one pixel wide, and does not do any "antialiasing" to smooth out the lines. You can try making wider and smoother lines by using the following commands:

```
glLineWidth( n );      // Lines are n pixels wide
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);    // Antialias lines
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

(In the first line, a value such as 3 for n may give good results.) How well, and whether, the line-width specification and the antialiasing work will depend on your implementation of OpenGL.

> **Exercise I.1** *The OpenGL program* `SimpleDraw` *includes code to draw the images shown in Figures I.5 and I.6, and a colorized version of Figure I.12. Run this program, and examine its source code. Learn how to compile the program and then try enabling the code for making bigger points and wider, smoother lines. (This code is already present but is commented out.) Does it work for you?*

> **Exercise I.2** *Write an OpenGL program to generate the two images of Figure I.7 as line drawings. You will probably want to modify the source code of* `SimpleDraw` *for this.*

Drawing Polygons in OpenGL

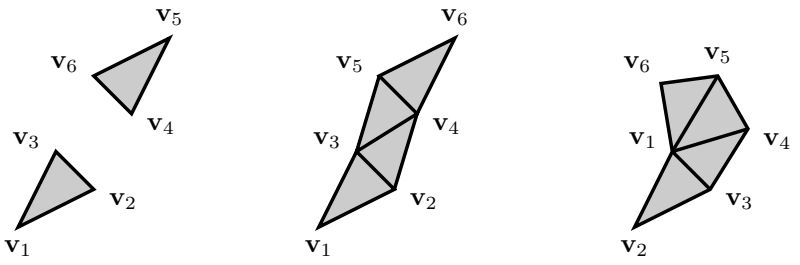
OpenGL includes commands for drawing triangles, quadrilaterals, and convex polygons. Ordinarily, these are drawn as solid, filled-in shapes. That is, OpenGL does not just draw the edges of triangles, quadrilaterals, and polygons but instead draws their interiors.

To draw a single triangle with vertices $\mathbf{v}_i = \langle x_i, y_i, z_i \rangle$, you can use the commands

```
glBegin( GL_TRIANGLES );
glVertex3f( x1, y1, z1 );
glVertex3f( x2, y2, z2 );
glVertex3f( x3, y3, z3 );
glEnd();
```

You may specify multiple triangles by a single invocation of the `glBegin(GL_TRIANGLES)` function by making $3n$ calls to `glVertex*` to draw n triangles.

Frequently, one wants to combine multiple triangles to form a continuous surface. For this, it is convenient to specify multiple triangles at once, without having to specify the same vertices repeatedly for different triangles. A "triangle strip" is drawn by invoking `glBegin`



GL_TRIANGLES GL_TRIANGLE_STRIP GL_TRIANGLE_FAN

Figure I.8. The three triangle-drawing modes. These are shown with the default front face upwards. In regard to this, note the difference in the placement of the vertices in each figure, especially of $\mathbf{v}_5$ and $\mathbf{v}_6$ in the first two figures.

with `GL_TRIANGLE_STRIP` and specifying n vertices. This has the effect of joining up the triangles as shown in Figure I.8.

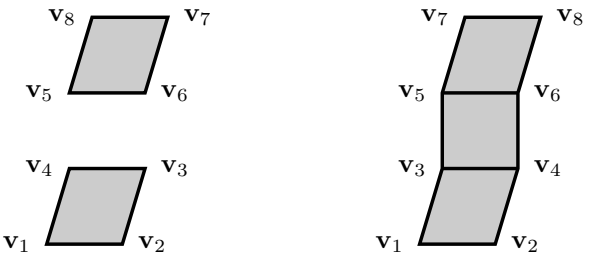
Another way to join up multiple triangles is to let them share the common vertex $\mathbf{v}_1$. This is also shown in Figure I.8 and is invoked by calling `glBegin` with `GL_TRIANGLE_FAN` and giving vertices $\mathbf{v}_1, \ldots, \mathbf{v}_n$.

OpenGL allows you to draw convex quadrilaterals, that is, convex four-sided polygons. OpenGL does not check whether the quadrilaterals are convex or even planar but instead simply breaks the quadrilateral into two triangles to draw the quadrilateral as a filled-in polygon.

Like triangles, quadrilaterals are drawn by giving `glBegin` and `glEnd` commands and between them specifying the vertices of the quadrilateral. The following commands can be used to draw one or more quadrilaterals:

```
glBegin( GL_QUADS );
glVertex3f( x1, y1, z1 );
  ...
glVertex3f( xn, yn, zn );
glEnd();
```

Here n must be a multiple of 4, and OpenGL draws the $n/4$ quadrilaterals with vertices $\mathbf{v}_{4i-3}$, $\mathbf{v}_{4i-2}$, $\mathbf{v}_{4i-1}$, and $\mathbf{v}_{4i}$, for $1 \leq i \leq n/4$. You may also use the `glBegin` parameter `GL_QUAD_STRIP` to connect the polygons in a strip. In this case, n must be even, and OpenGL draws the $n/2 - 1$ quadrilaterals with vertices $\mathbf{v}_{2i-3}$, $\mathbf{v}_{2i-2}$, $\mathbf{v}_{2i-1}$, and $\mathbf{v}_{2i}$, for $2 \leq i \leq n/2$. These are illustrated in Figure I.9.



GL_QUADS GL_QUAD_STRIP

Figure I.9. The two quadrilateral-drawing modes. It is important to note that the order of the vertices is different in the two modes!

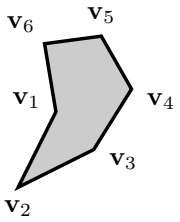


Figure I.10. A polygon with five vertices. This looks similar to the triangle fan of Figure I.8 but can give different results because the OpenGL standards do not specify how the polygon will be triangulated.

The vertices for `GL_QUADS` and for `GL_QUAD_STRIP` are specified in different orders. For `GL_QUADS`, vertices are given in counterclockwise order. For `GL_QUAD_STRIP`, they are given in pairs in left-to-right order suggesting the action of mounting a ladder.

OpenGL also allows you to draw polygons with an arbitrary number of sides. You should note that OpenGL assumes the polygon is planar, convex, and simple. (A polygon is *simple* if its edges do not cross each other.) Although OpenGL makes these assumptions, it does not check them in any way. In particular, it is quite acceptable to use nonplanar polygons (just as it is quite acceptable to use nonplanar quadrilaterals) as long as the polygon does not deviate too far from being simple, convex, and planar. What OpenGL does is to triangulate the polygon and render the resulting triangles.

To draw a polygon, you call `glBegin` with the parameter `GL_POLYGON` and then give the n vertices of the polygon. An example is shown in Figure I.10.

Polygons can be combined to generate complex surfaces. For example, Figure I.11 shows two different ways of drawing a torus as a set of polygons. The first torus is generated by using quad strips that wrap around the torus; 16 such strips are combined to make the entire torus. The second torus is generated by using a single long quadrilateral strip that wraps around the torus like a ribbon.

> **Exercise I.3** *Draw the five-pointed star of Figure I.7 as a solid, filled-in region. Use a single triangle fan with the initial point of the triangle fan at the center of the star. (Save your program to modify for Exercise I.4.)*

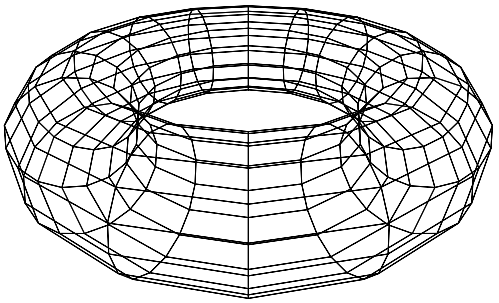
Colors

OpenGL allows you to set the color of vertices, and thereby the color of lines and polygons, with the `glColor*` commands. The most common syntax for this command is

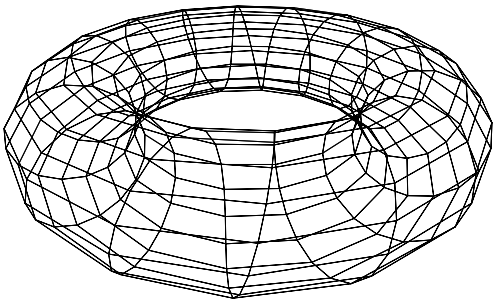
```
glColor3f( float r, float g, float b );
```

The numbers r, g, b specify respectively the brightness of the red, green, and blue components of the color. If these three values all equal 0, then the color is black. If they all equal 1, then the color is white. Other colors can be generated by mixing red, green, and blue. For instance, here are some ways to specify some common colors:

```
glColor3f( 1, 0, 0 );              // Red
glColor3f( 0, 1, 0 );              // Green
glColor3f( 0, 0, 1 );              // Blue
glColor3f( 1, 1, 0 );              // Yellow
glColor3f( 1, 0, 1 );              // Magenta
glColor3f( 0, 1, 1 );              // Cyan
```



(a) Torus as multiple quad strips.



(b) Torus as a single quad strip.

Figure I.11. Two different methods of generating wireframe tori. The second torus is created with the supplied OpenGL program `WrapTorus`. In the second torus, the quadrilaterals are not quite planar.

The brightness levels may also be set to fractional values between 0 and 1 (and in some cases values outside the range [0, 1] can be used to advantage, although they do not correspond to actual displayable colors). These red, green, and blue color settings are used also by many painting and drawing programs and even many word processors on PCs. Many of these programs have color palettes that let you choose colors in terms of red, green, and blue values. OpenGL uses the same RGB system for representing color.

The `glColor*` command may be given inside the scope of `glBegin` and `glEnd` commands. Once a color is set by `glColor*`, that color will be assigned to all subsequent vertices until another color is specified. If all the vertices of a line or polygon have the same color, then the entire line or polygon is drawn with this color. On the other hand, it is possible for different vertices of line or polygon to have different colors. In this case, the interior of the line or polygon is drawn by blending colors; points in the interior of the line or polygon will be assigned a color by averaging colors of the vertices in such a way that the colors of nearby vertices will have more weight than the colors of distant vertices. This process is called *shading* and blends colors smoothly across a polygon or along a line.

You can turn off shading of lines and polygons by using the command

```
glShadeModel( GL_FLAT );
```

and turn it back on with

```
glShadeModel( GL_SMOOTH );
```

In the flat shading mode, an entire region gets the color of one of its vertices. The color of a line, triangle, or quadrilateral is determined by the color of the *last* specified vertex. The color of a general polygon, however, is set by the color of its first vertex.

The background color of the graphics window defaults to black but can be changed with the `glClearColor` command. One usually starts drawing an image by first calling the `glClear` command with the `GL_COLOR_BUFFER_BIT` set in its parameter; this initializes the color to black or whatever color has been set by the `glClearColor` command.

Later in the book we will see that shading is an important tool for creating realistic images, particularly when combined with lighting models that compute colors from material properties and light properties, rather than using colors that are explicitly set by the programmer.

> **Exercise I.4** *Modify the program you wrote for Exercise I.3, which drew a five-pointed star as a single triangle fan. Draw the star in the same way, but now make the triangles alternate between two colors.*

Hidden Surfaces

When we draw points in three dimensions, objects that are closer to the viewpoint may occlude, or hide, objects that are farther from the viewer. OpenGL uses a depth buffer that holds a distance or depth value for each pixel. The depth buffer lets OpenGL do hidden surface computations by the simple expedient of drawing into a pixel only if the new distance will be less than the old distance. The typical use of the depth buffer is as follows: When an object, such as a triangle, is rendered, OpenGL determines which pixels need to be drawn and computes a measure of the distance from the viewer to each pixel image. That distance is compared with the distance associated with the former contents of the pixel. The lesser of these two distances determines which pixel value is saved, because the closer object is presumed to occlude the farther object.

To better appreciate the elegance and simplicity of the depth buffer approach to hidden surfaces, we consider some alternative hidden surface methods. One such method, called the *painter's algorithm*, sorts the polygons from most distant to closest and renders them in back-to-front order, letting subsequent polygons overwrite earlier ones. The painter's algorithm is easy but not completely reliable; in fact, it is not always possible to sort polygons consistently according to their distance from the viewer (cf. Figure I.12). In addition, the painter's algorithm cannot handle interpenetrating polygons. Another hidden surface method is to work out all the information geometrically about how the polygons occlude each other and to render only the visible portions of each polygon. This, however, is quite difficult to design and implement robustly. The depth buffer method, in contrast, is very simple and requires only an extra depth, or distance, value to be stored per pixel. Furthermore, this method allows polygons to be rendered independently and in any order.

The depth buffer is not activated by default. To enable the use of the depth buffer, you must have a rendering context with a depth buffer. If you are using the OpenGL Utility Toolkit (as in the code supplied with this book), this is done by initializing your graphics window with a command such as

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

which initializes the graphics display to use a window with RGB buffers for color and with a depth buffer. You must also turn on depth testing with the command

```
glEnable( GL_DEPTH_TEST );
```

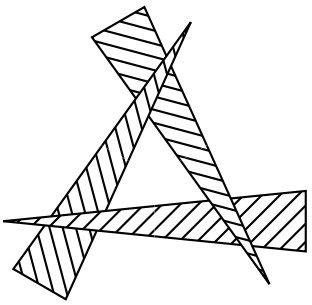


Figure I.12. Three triangles. The triangles are turned obliquely to the viewer so that the top portion of each triangle is in front of the base portion of another.

It is also important to clear the depth buffer each time you render an image. This is typically done with a command such as

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

which both clears the color (i.e., initializes the entire image to the default color) and clears the depth values.

The `SimpleDraw` program illustrates the use of depth buffering for hidden surfaces. It shows three triangles, each of which partially hides another, as in Figure I.12. This example shows why ordering polygons from back to front is not a reliable means of performing hidden surface computation.

Polygon Face Orientations

OpenGL keeps track of whether polygons are facing toward or away from the viewer, that is, OpenGL assigns each polygon a front face and a back face. In some situations, it is desirable for only the front faces of polygons to be viewable, whereas at other times you may want both the front and back faces to be visible. If we set the back faces to be invisible, then any polygon whose back face would ordinarily be seen is not drawn at all and, in effect, becomes transparent. (By default, both faces are visible.)

OpenGL determines which face of a polygon is the front face by the default convention that vertices on a polygon are specified in counterclockwise order (with some exceptions for triangle strips and quadrilateral strips). The polygons in Figures I.8, I.9, and I.10 are all shown with their front faces visible.

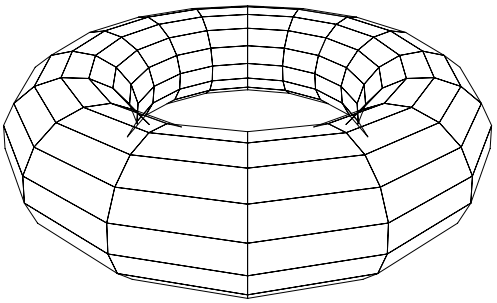
You can change the convention for which face is the front face by using the `glFrontFace` command. This command has the format

```
glFrontFace( { GL_CW  } );
             { GL_CCW }
```

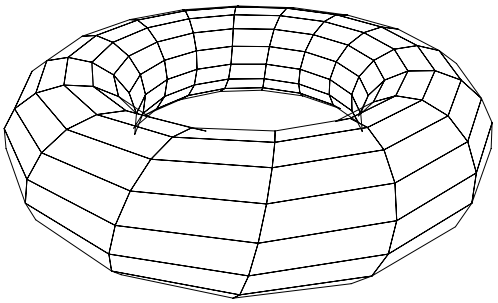
where "CW" and "CCW" stand for clockwise and counterclockwise; `GL_CCW` is the default. Using `GL_CW` causes the conventions for front and back faces to be reversed on subsequent polygons.

To make front or back faces invisible, or to do both, you must use the commands

```
                { GL_FRONT          }
glCullFace( {   GL_BACK           } );
                { GL_FRONT_AND_BACK }
glEnable( GL_CULL_FACE );
```



(a) Torus as multiple quad strips.



(b) Torus as a single quad strip.

Figure I.13. Two wireframe tori with back faces culled. Compare with Figure I.11.

You must explicitly turn on the face culling with the call to `glEnable`. Face culling can be turned off with the corresponding `glDisable` command. If both front and back faces are culled, then other objects such as points and lines are still drawn.

The two wireframe tori of Figure I.11 are shown again in Figure I.13 with back faces culled. Note that hidden surfaces are not being removed in either figure; only back faces have been culled.

Toggling Wireframe Mode

By default, OpenGL draws polygons as solid and filled in. It is possible to change this by using the `glPolygonMode` function, which determines whether to draw solid polygons, wireframe polygons, or just the vertices of polygons. (Here, "polygon" means also triangles and quadrilaterals.) This makes it easy for a program to switch between the wireframe and nonwireframe mode. The syntax for the `glPolygonMode` command is

```
glPolygonMode( { GL_FRONT | GL_BACK | GL_FRONT_AND_BACK }, { GL_FILL | GL_LINE | GL_POINT } );
```

The first parameter to `glPolygonMode` specifies whether the mode applies to front or back faces or to both. The second parameter sets whether polygons are drawn filled in, as lines, or as just vertices.

> **Exercise I.5** *Write an OpenGL program that renders a cube with six faces of different colors. Form the cube from six quadrilaterals, making sure that the front faces are facing*

outwards. If you already know how to perform rotations, let your program include the ability to spin the cube around. (Refer to Chapter II and see the `WrapTorus` *program for code that does this.)*

If you rendered the cube using triangles instead, how many triangles would be needed?

Exercise I.6 *Repeat Exercise I.5 but render the cube using two quad strips, each containing three quadrilaterals.*

Exercise I.7 *Repeat Exercise I.5 but render the cube using two triangle fans.*

I.3 Double Buffering for Animation

The term "animation" refers to drawing moving objects or scenes. The movement is only a visual illusion, however; in practice, animation is achieved by drawing a succession of still scenes, called frames, each showing a static snapshot at an instant in time. The illusion of motion is obtained by rapidly displaying successive frames. This technique is used for movies, television, and computer displays. Movies typically have a frame rate of 24 frames per second. The frame rates in computer graphics can vary with the power of the computer and the complexity of the graphics rendering, but typically one attempts to get close to 30 frames per second and more ideally 60 frames per second. These frame rates are quite adequate to give smooth motion on a screen. For head-mounted displays, where the view changes with the position of the viewer's head, much higher frame rates are needed to obtain good effects.

Double buffering can be used to generate successive frames cleanly. While one image is displayed on the screen, the next frame is being created in another part of the memory. When the next frame is ready to be displayed, the new frame replaces the old frame on the screen instantaneously (or rather, the next time the screen is redrawn, the new image is used). A region of memory where an image is being created or stored is called a buffer. The image being displayed is stored in the *front buffer*, and the *back buffer* holds the next frame as it is being created. When the buffers are swapped, the new image replaces the old one on the screen. Note that swapping buffers does not generally require copying from one buffer to the other; instead, one can just update pointers to switch the identities of the front and back buffers.

A simple example of animation using double buffering in OpenGL is shown in the program `SimpleAnim` that accompanies this book. To use double buffering, you should include the following items in your OpenGL program: First, you need to have a graphics context that supports double buffering. This is obtained by initializing your graphics window by a function call such as

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

In `SimpleAnim`, the function `updateScene` is used to draw a single frame. It works by drawing into the back buffer and at the very end gives the following commands to complete the drawing and swap the front and back buffers:

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

It is also necessary to make sure that `updateScene` is called repeatedly to draw the next frame. There are two ways to do this. The first way is to have the `updateScene` routine call `glutPostRedisplay()`. This will tell the operating system that the current window needs rerendering, and this will in turn cause the operating system to call the routine specified by `glutDisplayFunc`. The second method, which is used in `SimpleAnim`, is to use `glutIdleFunc` to request the operating system to call `updateScene` whenever the CPU is