Chapter 1  Introduction

takes ints as arguments. We shall use the notation glVertex*() to refer to all the
forms of the vertex function.

Many OpenGL functions have parameters whose values are chosen from small
discrete sets of integers. To avoid the use of “magic numbers,” OpenGL defines
macros for these values that are specified in the include files gl.h and glu.h. The
prefixes GL_ and GLU_ define from which include file the macro comes, for
example GL_LINES and GL_LIGHT0.

OpenGL uses the basic C data types: floats, doubles, ints, chars. However, to
allow an implementation to redefine its basic types, the include file gl.h defines
the basic types GLint, GLfloat, GLdouble, and GLubyte. Although these types
are generally what you would expect, they are used in the definitions of the
OpenGL functions. For example, the function

```c
void glVertex3f(GLfloat x, GLfloat y, GLfloat z)
```

defines a vertex in three-dimensional space using three GLfloats. In practice, there
is a line in gl.h for each type of the form

typedef GLfloat float

so that these values are what we would expect. Occasionally, you will also see less
familiar types such as GLclampf (for floats in the range from 0.0 and 1.0) and
GLsizei (for as large nonnegative integers as are available on the implementa-
tion).

OpenGL also provides an alternative function to many functions that require
parameters to allow specification through pointers to arrays. These functions have
a v before the argument list. Hence, there is a function glVertex3fv() that we
might use as

```c
GLfloat point[3] = {1.0, -2.5, 0.5};
glVertex3fv(point);
```

In our description of functions, we will use braces and the word TYPE to allow
us to specify the many forms of a function. For example, we can list all the forms
of glVertex*() as

```c
glVertex(234)[sifdl](TYPE coords,...);
glVertex(234)[sifdlv](TYPE *coords);
```

We have to select a dimension (two, three, or four) and a data type (short [s],
integer [i], float [f], or double [d]). TYPE matches the chosen data type. For the
direct form, there must be the same number of parameters as dimensions. For the
pointer form, we need give only a pointer of the correct type. Thus, the following
are all valid invocations of vertex functions:
GLint ix, iy;
GLfloat x, y, z, point[3];
/** set values */
glVertex2i(ix, iy);
glVertex2f(x, y);
glVertex3f(x, y, z);
glVertex3fv(point);

In the final form, point is assumed to be an array of the form
float point[3];

The details of each function will be in a description box of the form

void glFunctionName(GLboolean param)

This function does not exist, but the value of param can be GL_TRUE or GL_FALSE.

that will appear near where the function is first introduced. There will also be
some alerts or warnings that also will be in description boxes, such as:

Alert!
Read the contents of comments like this one.

1.7 Compiling

Generally, compiling an OpenGL program should not be a problem. For UNIX
systems, a typical compile line looks like

cc myapp.c -o myapp -lglut -lGLU -lGL -lX11 -lm

For Linux, you can use almost the same line but you may have to specify where the
X libraries are located. Usually adding a flag indicating a directory, such as

-L/usr/X11R6/lib

works. You can make a simple makefile that usually works for UNIX and Linux
based on the four lines

CC = cc
LDLIBS = -lglut -lGL -lGLU -lX11 -lm -L/usr/X11R6/lib
.c:
$(CC) $@.c $(LDLIBS) -o $@
A makefile with these lines will allow you to compile a single file via the command line

```
make progname
```

which will start with `progname.c` and result in the binary `progname`. Occasionally, you may need to add `-lxmu` for the X miscellaneous utility library to the loader line.

You may have to add a `-L` flag if your X libraries are in a nonstandard place or a `-I` flag if your OpenGL include files are in a nonstandard place.

Under Windows with Visual C++, you want to build a console application. The `OpenGL32.dll` and `glut32.dll` files should already be in the system folders. The corresponding `lib` files should be in `..\VC\lib`. The include files should be in `..\VC\include\GL`. A similar process works with other compilers such as Borland. You will have to get the GLUT files from the Web. You can get the files in a precompiled form or you can build them yourself from Nate Robins's GLUT site for Windows: www.xmission.com/~nate/glut.html. These files (`glut.h`, `glut32.lib`, and `glut32.dll`) go in the same places as the other OpenGL files.

1.8 Sources

There is a wide variety of OpenGL information available. First, there are books on OpenGL. The most important are known as the Redbook and the Bluebook:


The Redbook uses GLUT for its examples. If you want to develop code for the X Windows system, see


If you are developing applications exclusively for Windows, see


This primer assumes that you already know some computer graphics. If you need an introduction to computer graphics that uses OpenGL, then see


The standard reference in computer graphics is

2.2 GLUT

The OpenGL Utility Toolkit (GLUT) is a library of functions that are common to virtually all modern windowing systems. GLUT has been implemented on all the popular systems, so that programs written using the GLUT API for windowing and input can be compiled with the source code unchanged on all these systems. The library organization is as shown in Figure 2.2. The application program uses only the GL, GLU, and GLUT libraries. Depending on the platform, GLUT uses glX, wgl, or a gl. We shall use GLUT throughout this volume. There are some toolkits that have been built on top of GLUT that provide additional functionality. However, if you need higher performance or want to use a specific toolkit, then most likely you will have to write platform-dependent code for the interactive parts of your application.

```c
void glutInit(int *argc, char **argv)
```

initializes GLUT and should be called before other GLUT and OpenGL functions. `glutInit()` takes the arguments from `main()` and the program can use them in an implementation-dependent manner.

Starting with the first function in `main()`, `glutInit()`, we see that the name (as it is for all GLUT functions) begins with the letters glut. Although we can pass in command line arguments from `main()`, their interpretation within GLUT is implementation dependent. We shall not use implementation-dependent arguments in our examples. The function `glutCreateWindow()` puts a window on the screen in a default position (at the upper-left corner) and at a default size (300 × 300 pixels). The argument allows us to put an optional title on the top border of the window.
int glutCreateWindow(char *title)

creates a window on the screen with the title given by the argument. The function returns an integer that can be used to refer to the window in multwindow situations.

Later we shall see how to alter the defaults in GLUT.

2.3 Event Loops and Callback Functions

Most interactive programs are based on the program reacting to a variety of discrete events. Events include mouse events, such as moving the mouse or clicking a mouse button; keyboard events, such as pressing a key; and window events, such as the resizing of a window by the user or the covering up of a window by another window. The programming paradigm used to work with events is to have events handled by the window system and placed in an event queue. Events are processed sequentially from the event queue. The application programmer needs only to write a set of callback functions that define how the program should react to specific events. In GLUT, the most common events are recognized. The application program can define its own callback functions, rely on default callbacks for a few events, or do nothing, in which case events without callbacks are ignored.

Our simple program contains only a single callback, the display callback, which is invoked whenever OpenGL determines that the window has to be redrawn. One of these times is when the window is first opened. Consequently, if we put our OpenGL rendering commands in the display callback, we can be assured that they will be drawn at least once. Note that the form of the display callback, a function with no arguments, is registered by

void glutDisplayFunc(void (*func) (void))

The function func() is called each time the window needs to be redrawn.

GLUT. If we wish to pass values to the display callback function, we can use globals in our programs. After the callbacks have been registered, the program enters the event loop by executing the function glutMainLoop(). Once we have entered the loop, we cannot escape except through a callback or some outside intervention such as hitting a “kill” key. Any code after this call will never be executed.

Some compilers insist that because correct C requires the main function to return a value, OpenGL main programs should end with the code
glutMainLoop();
    return(0);
}

We will not use this form. We prefer to not insert code that cannot be reached, even when doing so is technically correct.

Alert!
The form of GLUT callback functions is fixed. Consequently, global variables may be necessary to pass values between functions.

void glutMainLoop()

causes the program to enter an event-processing loop. It should be the last statement in the main() function.

2.4 Drawing a Rectangle

Now we have to define our display callback, which we have chosen to name display. First, note that we include the file glut.h, which is usually stored in a directory named GL wherever the standard include files are stored. This file contains the prototypes for the GLUT functions and the #defines for a variety of constants that are used in OpenGL programs. This file also contains the lines

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

to include similar definitions for the OpenGL and GLU functions and constants.

The fundamental entity for specifying geometric objects is the vertex, a location in space. Simple geometric objects such as lines and polygons can be specified through a collection of vertices. OpenGL allows us to work in two, three, or four dimensions through variants of the function glVertex*().

void glVertex[234]{sifd}(TYPE xcoordinate, TYPE ycoordinate, ....)
void glVertex[234]{sifd}v(TYPE *coordinates)

specifies the location of a vertex in two, three, or four dimensions with one of the types short (s), int (i), float (f), or double (d). If v is present, TYPE is a pointer to the array coordinates of the type specified.
We shall use the notation `glVertex2f(x, y)` to refer to all of these variants. Thus, for example, `glVertex2f(x, y)` defines a vertex in two dimensions at the point \((x, y)\) where \(x\) and \(y\) are floats, while `glVertex2fv(p)` specifies a vertex at the first two locations of an array of floats, that is \((p[0], p[1])\).

Because vertices can define a variety of objects, we must tell OpenGL what type of object a list of vertices defines and denote the beginning and end of the list. We make these specifications through the functions `glBegin()` and `glEnd()`.

```c
void glBegin(GLenum mode)
{
 specifies the beginning of an object of type mode. Modes include GL_POINTS,
 GL_LINES, and GL_POLYGON.
}

void glEnd()
{
 specifies the end of a list of vertices.
}
```

**Alert:** Don’t forget to include the `glEnd()` after a list of vertices.

Using these three functions, we can define our rectangle

```c
glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glEnd();
```

OpenGL puts the rendered image in an area of memory called a color buffer
that usually resides on the graphics card. Color buffers are one of a number of
types of buffers that make up the frame buffer. We shall see other types of buffers
later. Before we draw the rectangle, we clear the color buffer we are using through
the function `glClear()`.

```c
void glClear(GLbitfield mask)
{
 clears all buffers whose bits are set in mask. The mask is formed by the logical OR of
 values defined in gl.h. GL_COLOR_BUFFER_BIT refers to the color buffer.
```
Thus, point at the type st. We

Figure 2.3 Output from simple.c

Because OpenGL implementations buffer commands for efficiency, we can force the render to output the results immediately by issuing a `glFlush()`.

```c
void glFlush()
```

forces previously buffered OpenGL commands to execute.

Figure 2.3 shows the output from our program. Although the program obviously produces an image, there is a variety of questions that we need to address, including:

- What do we do if we want the image to be a different size?
- What do we do if we want the image to appear in a different place on the screen?
- Why does the white rectangle occupy half the area in the window?
- Why is the background black and the rectangle white? How can we use other colors?
- Can we end the program in some manner other than by using the kill box provided by the window system?
- How do we define more complex objects?

## 2.5 Changing the GLUT Defaults

First, let's add a few GLUT functions that will give us a little finer control over the image that appears on our screen. The functions `glutInitDisplayMode()`, `glutInitWindowSize()`, and `glutInitWindowPosition()` allow us to define
what type of window we want, its size, and its position. Generally, an implementation will support a variety of properties that can be associated with a window on the screen. An application program, through the function `glutInitDisplayMode()`, requests the type of window that it requires. The most common window properties to specify are what type of color we wish to use and whether or not we need double buffering (see Section 3.4). The defaults in GLUT are RGB color and single buffering, which can be specified explicitly by the function call

```
void glutInitDisplayMode(unsigned int mode);
```

requests a display with the properties in `mode`. The values of `mode` are combined by using the logical OR of options such as color model (`GLUT_RGB`, `GLUT_INDEX`) and buffering of color buffers (`GLUT_SINGLE`, `GLUT_DOUBLE`).

The function `glutInitWindowSize()` specifies the initial size of the window on the screen, and `glutInitWindowPosition()` gives its initial position.

```
void glutInitWindowSize(int width, int height);
```

specifies the initial height and width of the window on the screen in pixels.

```
void glutInitWindowPosition(int x, int y);
```

specifies the top-left corner of the window measured in pixels from the top-left corner of the screen.

## 2.6 Color in OpenGL

OpenGL supports two basic color models: RGB (or RGBA) mode and color-index mode. In RGB mode, each color is a triplet of red, green, and blue values. The eye adds or blends these primary colors (or color components), forming the color that we see. This additive model is appropriate for monitors and projective systems. The print industry uses a subtractive model, which is discussed in most graphics texts.

If we use real numbers to specify colors, then 0.0 is none of a component and 1.0 is the maximum amount of that component. Thus, the RGB triplet of (1.0, 0.0, 0.0) is a bright red, (0.5, 0.5, 0.0) is a dark yellow, (1.0, 1.0, 1.0) is white, and (0.0, 0.0, 0.0) is black. In RGBA mode, we use a fourth color component, A or alpha, which is an opacity. An opacity of 1.0 means the color is opaque and
cannot be "seen through" while a value of 0.0 means that a color is transparent. We will not need opacity until much later. If we use an integer type to specify a color, the range is from 0 to the maximum value of the type. If, for example, we use unsigned bytes, the color values range from 0 to 255. Note that specifying a color to a given precision does not guarantee that the physical display can support the specified precision. There is no reason that we cannot specify each component of a color as a real number even if the display can show only two colors, a foreground color and a background color.

The color precision of the display may be less than that used to specify a color by glColor*().

In color-index mode, colors are specified as indices into a table of red, green, and blue values. In this mode, we form a table of the allowed colors, usually with 256 possible colors. Historically, color-index mode was common on graphics systems. Now with inexpensive memory, this mode is not used often. In addition, color-index mode requires more detailed interaction with the windowing system than does RGB color. Hence, we shall always use RGB or RGBA color.

### 2.6.1 Setting Colors

In our simple program, we used the default values for our colors. The default color for clearing the screen was black and the default drawing color, the color that was used to fill the polygon, was white. These can be changed by the functions glColor*() and glClearColor().

```c
void glColor3b(int r, int g, int b)
void glColor3b(int r, int g, int b, int a)
void glColor4b(int r, int g, int b, int a)
void glColor4b(int r, int g, int b, int a, int a)

specifies RGBA colors using the standard types. If the a is present, the color is in an array pointed to by color in which each component is of type TYPE.

void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a)

specifies the RGBA color used when clearing the color buffer.
```

The clear color must be specified as an RGBA color.
2.10  Coordinate Systems and Transformations

At this point we have seen two coordinate systems in our functions. The first is called **object coordinates** (or **world coordinates**\(^2\)). It is the application coordinate system that programmers use to write their programs. Each application programmer can decide what units she prefers, then specify values in these units in OpenGL functions such as `glVertex*()`. Thus, she might use microns for problems in VLSI design or light years for astronomical problems. The second coordinate system is called **window coordinates** (or **screen coordinates**) and uses units measured in pixels. The allowable range of window coordinates is determined by properties of the physical display and what part of that display is selected by the application program.

OpenGL automatically makes a coordinate transformation from object to window coordinates as part of the rendering process.\(^3\) The only information required is the size of the display window on the screen and how much of the object space the user wishes to display. The former is determined by `glutInitWindowSize()` (and possibly modified by later interactions), while the latter is set by `gluOrtho2D()`.

The required coordinate system transformations in OpenGL are determined by two matrices, the **model-view matrix** and the **projection matrix**, that are part of OpenGL's state. We shall study these matrices in detail in Chapter 5. However, we need to use a simple projection matrix in even the most basic programs. The function `gluOrtho2D()` is used to specify a projection matrix for two-dimensional applications. The typical sequence to set either of the matrices requires that we perform three steps:

1. Identify which matrix we wish to alter.
2. Set the matrix to an identity matrix.
3. Alter the matrix to form the desired matrix.

The second step is not required if we want to alter an existing matrix incrementally. Thus, if we want to set up a two-dimensional clipping window whose lower-left corner is at \((-1.0, -1.0)\) and whose upper-right corner is at \((1.0, 1.0)\), which are the default values we used in OpenGL, we execute the functions

```c
glMatrixMode(GL_PROJECTION);
gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
```

---

2. Technically, in OpenGL objects are specified in object coordinates and can be transformed by OpenGL transformations into world coordinates.

3. OpenGL uses a series of other coordinate systems internally as part of the rendering pipeline. Some of these are not visible to the application program, while others, such as camera coordinates, will be needed when we discuss three-dimensional applications in Chapter 5.
void init()
{
    /* set clear color to black */
    glColor3f(0.0, 0.0, 0.0);
    /* set fill color to white */
    glColor3f(1.0, 1.0, 1.0);
    /* set up standard orthogonal view with clipping */
    /* box as cube of side 2 centered at origin */
    /* This is default view and these statements could be removed. */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
}

int main(int argc, char** argv)
{
    /* initialize mode and open a window in upper-left corner of
     * screen */
    /* window title is name of program (arg[0]) */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("simple");
    glutDisplayFunc(display);
    init();
    glutMainLoop();
}

Program 2 illustrates the organization that we shall use for almost all our
programs. Our programs will consist of four major parts:

1. A main() function that initializes GLUT, puts a window on the screen,
   identifies the callback functions, and enters the main loop
2. A init() function that sets state variables to their initial values
3. A display callback, display(), that contains the code describing our
   objects
4. Other callbacks that deal with input and window events

Although other structures are possible, this organization has some advantages.
The main() function is almost the same from program to program. Differences are
usually due to which callbacks and menus are used in a particular application. Using
init() allows us to place a lot of detailed state information and desired parameters in
one place, separate from the geometry (which is in the display callback) and from the
dynamics of animated and interactive programs (which usually are in the callbacks).
void glMatrixMode(GLenum mode)

specifies which matrix will be affected by subsequent transformation functions. The
type is usually GL_MODELVIEW or GL_PROJECTION.

void glLoadIdentity()

initializes the current matrix to an identity matrix.

Because these matrices are part of the OpenGL state, OpenGL will use their
current values whenever a primitive is specified. These matrices can be changed
virtually anywhere in an application program. For our simple example, where
there is no user interaction, we can set the matrices once as part of the initialization
phase of the program. In Chapter 3, we will change transformation matrices
in response to user events, such as the resizing of the screen window.

2.11 Simple.c, Second Version

We can now incorporate all these changes into our program. The resulting pro-
gram will behave the same as the first program, but its structure is more general
and characterizes more complex two-dimensional applications.

/* simple.c second version */
/* This program draws a white rectangle on a black background. */
#include <GL/glut.h>  /* glut.h includes gl.h and glu.h */
void display()
{
    /* clear window */
    glClear(GL_COLOR_BUFFER_BIT);
    /* draw unit square polygon */
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    /* flush GL buffers */
    glFlush();
}
glPointSize(2.0);
glBegin(GL_POINTS);
  glColor3f(1.0, 1.0, 1.0);
  glVertex2f(-0.5, -0.5);
  glColor3f(1.0, 0.0, 0.0);
  glVertex2f(-0.5, 0.5);
  glColor3f(0.0, 0.0, 1.0);
  glVertex2f(0.5, 0.5);
  glColor3f(0.0, 1.0, 0.0);
  glVertex2f(0.5, -0.5);
glEnd();

Note that glPointSize() is one of the functions that cannot go between a 
glBegin() and a glEnd().

2.12.2 Lines
There are three choices (see Figure 2.5) for type that we can use to define one or 
more line segments between a glBegin() and a glEnd():

GL_LINES: Each successive pair of vertices between glBegin() and glEnd() 
defines a line segment. Thus, the code

```
glBegin(GL_LINES);
  glVertex2f(-0.5, -0.5);
  glVertex2f(-0.5, 0.5);
  glVertex2f(0.5, 0.5);
  glVertex2f(0.5, -0.5);
glEnd();
```

defines two line segments, the first from (-0.5, -0.5) to (-0.5, 0.5) and the 
second from (0.5, 0.5) to (0.5, -0.5).

![Diagram of line types](image)

Figure 2.5 Point and line types
set the drawing color to yellow, define lines as two pixels wide, and define a
dashed stipple pattern in which groups of six pixels are not colored and the follow-
six pixels are rendered in yellow.

2.12.3 Enabling OpenGL Features

Stippling is one of many OpenGL features that have to be enabled specifically. The
renderer has many capabilities such as lighting, hidden-surface removal, and tex-
ture mapping it can perform, although generally each feature will slow down the
rendering processing. A program can turn on—enable—or turn off—disable—each
of these features individually with the application program. Some features such as
lighting may be required in one part of a program but not in others, so an OpenGL
program may be more efficient if that feature is disabled when it is no longer
needed.

```c
void glEnable(GLenum feature)

void glDisable(GLenum feature)
```

turns on or off the OpenGL option feature.

Line stippling is enabled by

```c
glEnable(GL_LINE_STIPPLE);
```

Don't forget to enable features you want to use. Setting the parameters is not sufficient
if the feature has not yet been enabled.

One of the characteristics of high performance graphics hardware is that many
features such as lighting and texture mapping are carried out in hardware rather
than software. Consequently, enabling these features may not incur a significant
performance penalty.

2.12.4 Filled Primitives

The polygon primitive with which we started is one example of a filled primitive:
a primitive with an interior that can be filled with a color or a pattern when it is
displayed. There are six filled primitives with the type parameters, as shown in
Figure 2.6:

- GL_POLYGON: Defines a polygon by a sequence of `glVertex*(*)` calls
  between a `glBegin()` and `glEnd()`.
else
{
    first = !first;
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON):
    glVertex2i(xx, yy);
    glVertex2i(xx, hh - y);
    glVertex2i(x, hh - y);
    glVertex2i(x, yy);
    glEnd();
}

There are a few subtleties in this code. The most important is the necessity of inverting the y value returned by the mouse callback. This inversion is required because the values returned to the mouse callback are given in screen coordinates whose origin is at the upper-left corner. The values used for the clipping window and specification of the geometry are in world coordinates where the origin is at the lower-right corner. However, to carry out the inversion, we need the height of the screen window, a value that can change during the execution of the program if the user resizes the window. We handle this problem using the global value of the window height (hh), which is updated automatically by the reshape callback.

**Alert!**

Keep the window size as globals so that you can convert the mouse position to world coordinates correctly.

The vertices specified in our example use the integer form of glVertex2i(). The values used for the locations of the vertices are those obtained from the mouse callback and thus are integers. A little thought will show that the clipping window that we defined in the reshape callback is too small. A better choice for this example would be to use the reshape callback to have the clipping window match the screen window, as in the code:

```c
int ww, hh; /* globals for viewport height and width */
void myReshape(GLsizei w, GLsizei h)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLfloat)w, 0.0, (GLfloat)h);
    glMatrixMode(GL_MODELVIEW);
    glViewport(0, 0, w, h);
    ww = w;
    hh = h;
}
As we have developed the code, there is no display callback; all the work is done in the mouse callback. As a practical matter, GLUT insists that every program have a display callback. We could just put in the dummy display callback

```c
void mydisplay()
{

}
```

but, although this display callback will work, many application programmers would object to this style.

A more general strategy is to place drawing functions in the display callback and use the other callbacks for state changes. Although in many situations we may not always be able to do this, we can do it easily for this example through the two callbacks

```c
GLint x1, y1, x2, y2;
int hh;

void mymouse(int x, int y, int button, int state)
{
    static bool first = true;
    int xx, yy;
    if(state == GLUT_DOWN&&button == GLUT_LEFT_BUTTON)
        exit();
    if(state == GLUT_DOWN&&button == GLUT_RIGHT_BUTTON)
        if(first)
            { 
                x1 = x;
                y1 = hh - y;
                first = !first;
            }
        else
            { 
                first = !first;
                x2 = x;
                y2 = hh-y;
            }
    glutPostRedisplay();
}
```

```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
    glVertex2i(x1, y1);
    glVertex2i(x1, hh - y2);
    glVertex2i(x2, hh - y2);
    glVertex2i(x2, y1);
    glEnd();
}
```
The matrix determined by `gluLookAt()` is applied to the existing matrix. Hence, we must first make sure we are in the desired matrix mode, usually the model-view mode, and have initialized this matrix.

Suppose that we want an isometric view of the cube, which is centered at the origin and aligned with the axes in world coordinates. An isometric view (Figure 4.6) is symmetric with respect to the vertices of the cube. A simple way to obtain such a view is to place the camera on a line passing through the origin and the point (1, 1, 1). Thus, we can set up our viewing by the OpenGL code

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
 gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

The implementation of the function `gluLookAt()` consists of the required translations and rotations to set up the view.

### 4.4.1 A Cube Display Program

Putting everything together, we get a minimal three-dimensional program.

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

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glutWireCube(0.5);
    glutSwapBuffers();
}

void reshape(int w, int h)
```


glViewport(0, 0, w, h);
gMatrixMode(GL_PROJECTION);
LoadIdentity();
glOrtho(-4.0, 4.0, -4.0, 4.0, -4.0, 4.0);

void init()
{
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glColor3f(0.0, 0.0, 0.0);
}

int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(0, 0);
  glutCreateWindow("cube");
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  init();
  glutMainLoop();

  Note that we are using the default colors so we could have left out the color functions in init(). The program is not interactive, so double buffering is not required and we could have located the camera in init(). However, our choices are probably better as we want to expand this structure to interactive three-dimensional applications.

  Although we could have obtained the same isometric view of the cube by placing the eye point anywhere along the line from the origin through the point (1, 1, 1), if we changed the eye point we would have also had to change the near and far distances in glOrtho() because these distances are measured from the origin in eye space.

  ![Alert!]

  The clipping volume set in glOrtho() is measured from the origin in eye space. Thus, the near distance should be less than the far distance.

4.5 Building Objects

Let's redo the example, building our own cube. This time, we shall use polygons and give a different color to each face. We will center the cube at the origin and let it have sides of length 2. We could start with code something like
```c
void cube()
{
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
        glVertex3f(-1.0, -1.0, -1.0);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 1.0, 1.0);
        glVertex3f(-1.0, -1.0, 1.0);
    glEnd();

    /* the other faces */
    .
    .
    .
}

void display()
{
    glClear(COLOR_BUFFER_BIT);
    cube();
}
```

Note that we will be filling the polygons, we should be sure that we specify the vertices in a counterclockwise manner when each face is viewed from the outside so that we will have the correct interpretation of its orientation (front or back facing).

This code requires 42 function calls within `cube()` to define the cube. We could do slightly better if we use the type `GL_QUADS` rather than `GL_POLYGON` in the first `glBegin()`. We could then eliminate all subsequent uses of `glBegin()` and omit all the calls to `glEnd()` except the last because each consecutive group of four vertices would determine a quad.

### 4.5.1 Using Arrays

We can obtain a better structure for the code by putting the colors and the vertices in arrays. This structure will not only make the code clearer but will also prove more flexible for interactive applications.

We number the vertices as in Figure 4.7. The necessary arrays are then

```c
GLfloat vertices[][3] = {{1.0, -1.0, -1.0}, {-1.0, 1.0, 1.0},
        {1.0, 1.0, -1.0}, {-1.0, -1.0, 1.0},
        {1.0, 1.0, 1.0}, {1.0, -1.0, -1.0}};

GLfloat colors[][3] = {{0.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
        {0.0, 1.0, 0.0}, {1.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
        {1.0, 0.0, 1.0}};
```
We could use the previous code simply by substituting `glColor3fv()` for `glColor3f()` and `glVertex3fv()` for `glVertex3f()`, but that would not be much of a gain. Instead we add a simple function that draws a single polygon in terms of the indices of the vertices.

```c
void polygon(int a, int b, int c, int d)
{
    glColor3fv(colors[a]);
    glBegin(GL_POLYGON);
    glVertex3fv(vertices[a]);
    glVertex3fv(vertices[b]);
    glVertex3fv(vertices[c]);
    glVertex3fv(vertices[d]);
    glEnd();
}
```

This function assigns a color to the polygon from the first index, but we could change this choice easily. We can now replace the cube function by

```c
void cube()
{
    polygon(0, 3, 2, 1);
    polygon(2, 3, 7, 6);
    polygon(3, 0, 4, 7);
    polygon(1, 2, 6, 5);
    polygon(4, 5, 6, 7);
    polygon(5, 4, 0, 1);
}
```

Although the execution of this code does not save us any function calls, it is a lot cleaner than our previous example. One of its advantages is that the location of each vertex appears only once. In an interactive program, where vertices might be
changed by the user during program execution, making changes to vertices is particularly simple.

4.5.2 Vertex Arrays

OpenGL provides a facility called vertex arrays that extends the use of arrays in a way that avoids most of the function calls to draw the cube. The main idea is that information stored in arrays can be stored on the clients (the application programs) and accessed by a single function call. We can store the information in a way that retains the structuring that we defined above, such as the order in which vertices are called to draw the cube.

OpenGL provides support for six types of arrays: vertex, color, color index, normal, texture coordinate, and edge flag. We shall see some of the other types later. There are three steps in using vertex arrays. First, like other OpenGL features, we must enable their functionality. Second, we must specify the format of the arrays. These two steps are usually part of the initialization phase of our programs. Finally, we use the arrays to render the scene.

In our example, we need only color and vertex arrays. We enable them by

```c
 glEnableClientState(GL_COLOR_ARRAY);
 glEnableClientState(GL_VERTEX_ARRAY);
```

The form of the arrays is given by

```c
 glVertexPointer(3, GL_FLOAT, 0, vertices);
 glColorPointer(3, GL_FLOAT, 0, colors);
```

```c
 void glEnableClientState(GLenum array)
 void glDisableClientState(GLenum array)
```

enables and disables arrays of types GL_VERTEX_ARRAY, GL_COLOR_ARRAY,
GL_INDEX_ARRAY, GL_NORMAL_ARRAY,
GL_TEXTURE_COORD_ARRAY, or GL_EDGE_FLAG_ARRAY.

```c
 void glVertexPointer(GLint dim, GLenum type, GLsizei stride,
 GLvoid *array)
 void glColorPointer(GLint dim, GLenum type, GLsizei stride,
 GLvoid *array)
```

provides the information on arrays. The data are in array, dim is the dimension of the data (two, three, or four), type denotes how the data are stored (GL_SHORT,
GL_INT, GL_FLOAT, or GL_DOUBLE), and stride is the number of bytes between consecutive data values (0 means the data are packed in the array).
The first value (3) denotes that the data are three-dimensional. The second and third parameters indicate that the data are floats packed in the arrays given by the fourth parameter.

We need a new array that stores the indices in the order in which we want to use them. The following array contains the necessary information

```c
GLuint cubeIndices[] = {10, 3, 2, 1, 2, 3, 7, 6, 0, 4, 7, 3, 1, 2, 6, 5, 4, 5, 6, 7, 0, 1, 5, 4};
```

Now we can draw the cube through the function `glDrawElements()`.

```c
void glDrawElements(GLenum mode, GLsizei n, GLenum type, void *indices)
```

draws elements of type `mode` using `n` indices from the array `indices` for each. The array is of type `{GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT}`.

If we render each face individually, we can use the loop

```c
for(i = 0; i < 6; i++) glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, cubeIndices);
```

However, there is a simpler way if we recall that when we use the type `GL_QUADS`, each successive group of four vertices determines a new quad. Thus, a single function call suffices:

```c
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
```

There is a subtle problem here. When we execute `glDrawElements()`, all the enabled arrays are rendered. Thus, the rendering is equivalent to what we would see from code of the form

```c
glColor3fv(color(cubeIndices[0]));
glVertex3fv(vertex(cubeIndices[0]));
glColor3fv(color(cubeIndices[1]));
glVertex3fv(vertex(cubeIndices[1]));
/* etc */
```

Thus, the code is as if there is a color change before each vertex. We avoided one potential problem by making sure we had the same number of colors as vertices. But we saw in Chapter 2 that the default shading model is smooth. If we assign a