Getting Started with OpenGL

Fall 1997

Contents

List of Figures

List of Tables

Getting Started with OpenGL

Fall 1997

1 Introduction

OpenGL consists of three libraries: the Graphics Library (GL); the Graphics Library Utilities (GLU); and the Graphics Library Utilities Toolkit (GLUT). Function names begin with `gl`, `glu`, or `glut`, depending on which library they belong to.

All three libraries are independent of a particular platform: an OpenGL program written for SGI machines should compile and run under Windows 95, although the images may not be precisely the same.

The main sources for these notes are listed below. Errors are the responsibility of the author.

1. *OpenGL Programming Guide*. Mason Woo, Jackie Neider, and Tom Davis. Second Edition, Addison-Wesley, 1997. The “official guide to learning OpenGL”.
2. *The OpenGL Utility Toolkit (GLUT) Programming Interface*. Mark J. Kilgard. Silicon Graphics, 1997. Available *via* WWW.
3. *The OpenGL Graphics System: a Specification (Version 1.1)*. Mark Segal and Kurt Akeley. Silicon Graphics, 1997. Available *via* WWW.

1.1 Compiling and Executing an OpenGL Program

Use the following steps to create and run a graphics program.

1. Write a “make file”. This file should be called `Makefile` and should contain the following text:

```
CC = cc

LDLIBS = -lglut  -lGL -lGLU -lXmu -lX11 -lm

INCLUDE =

executable:
    $(CC) -g $(INCLUDE) $(PROG).c \
        $(LDFLAGS) $(LDLIBS) -o $(PROG)
```

2. Write a C program. We assume here that this program is called `myprog.c`.
3. The program must contain the directive `#include <GL/glut.h>`. It may also contain directives to include header files from other libraries.

4. To compile the program, enter the following command:

```
make PROG=myprog
```

5. To execute the program, enter the following command:

```
myprog
```

These steps apply to a program that consists of a single C source file. If you write a program with several source files, you will have to write a suitable make file for it.

A Simple Program Figure 1 shows a simple OpenGL program. Although this program does nothing useful, it defines a pattern that is used by most OpenGL programs. The line numbers are not part of the program, but are used in the following explanation.

```
 1     #include <GL/glut.h>
 2
 3     void display (void)
 4     {
 5         glClear(GL_COLOR_BUFFER_BIT);
 6     }
 7
 8     void init (void)
 9     {
10     }
11
12     int main (int argc, char *argv[])
13     {
14         glutInit(&argc, argv);
15         glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
16         glutInitWindowSize(800, 600);
17         glutInitWindowPosition(100, 50);
18         glutCreateWindow("My first openGL program");
19         init();
20         glutDisplayFunc(display);
21         glutMainLoop();
22         return 0;
23      }
```

Figure 1: The basic OpenGL program

Line 1. Every OpenGL program should include `GL/glut.h`. The file `glut.h` `#include`s `glu.h`, `gl.h` and other header files required by GLUT.

Line 3. You must write a function that displays the graphical objects in your model. The function shown here clears the screen but does not display anything. Your program does not call this function explicitly, but it will be called at appropriate times by OpenGL.

Line 5. It is usually a good idea to clear various buffers before starting to draw new objects. The colour buffer, cleared by this call, is where the image of your model is actually stored.

Line 8. You must write a function that initializes OpenGL state variables and your own data structures. This function is called explicitly (at line 18): it is a convention to put "standard initialization" (the `glut...` calls) in the `main` program and other initialization in a function called `init()` or `myinit()`.

Line 14. The function `glutInit()` initializes the OpenGL library. It is conventional to pass the command line arguments to this function because it may use some of them. You will probably not need this feature.

Line 15. The function `glutInitDisplayMode()` initializes the display (surprise!). Its argument is a bit string.

Deleting this line would not affect the execution of the program, because the arguments shown are the default values.

Line 16. This call requests a graphics window 800 pixels wide and 600 pixels high.

Line 17. This call says that the left side of the graphics window should be 100 pixels from the left of the screen and the top of the graphics window should be 50 pixels below the top of the screen. Note that the Y value is measured from the **top** of the screen here.

Line 18. This call creates the window using the settings given previously. The text in the argument becomes the title of the window. The window does not actually appear on the screen until `glutMainLoop()` has been called.

Line 19. This is a good place to perform any additional initialization that is needed. Some initialization, such as calls to `glEnable()`, **must** come after `glutCreateWindow()`.

Line 20. This call **registers** the callback function `display()`. After executing the call, OpenGL knows what function to call to display your model in the window.

Line 21. The last statement of an OpenGL program always calls `glutMainLoop()`. This function processes events until the user terminates the program by closing the graphics window.

1.2 Drawing Objects

All that remains is to describe additions to Figure 1. The first one that we will consider is an improved `display()` function. Figure 2 shows a function that displays a line. The line is bright red because the three arguments of `glColor3f()` give the intensities of red, green, and blue as numbers between 0 and 1.

The construction `glBegin(`*mode*`); ...; glEnd();` is used to display groups of primitive objects. The value of *mode* determines the kind of object: in this case, `GL_LINES` tells OpenGL to expect one or more lines given as pairs of vertices.

```
void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_LINES);
        glVertex2f(-1.0, 0.0);
        glVertex2f(1.0, 0.0);
    glEnd();
}
```

Figure 2: Displaying a bright red line

In this case, the line goes from $(-1, 0)$ to $(1, 0)$, as specified by the two calls to `glVertex2f()`. If we use the default viewing region, as in Figure 1, this line runs horizontally across the centre of the window.

The suffix "`3f`" indicates that `glColor3f()` requires three arguments of type `float`. Similarly, `glVertex2f()` requires two arguments of type `float`. Section 7 provides more details of this notational convention and explains why it is used.

The code between `glBegin()` and `glEnd()` is not restricted to `gl` calls. Figure 3 shows a display function that draws 51 vertical yellow (red + green) lines across the window. Section 3 describes some of the other primitive objects available in OpenGL.

```
void display (void)
{
    int i;
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_LINES);
        for (i = -25; i <= 25; i++)
        {
            float x = i / 25.0;
            glVertex2f(x, -1.0);
            glVertex2f(x, 1.0);
        }
    glEnd();
}
```

Figure 3: Drawing vertical yellow lines

1.3 Initialization

In this section, we consider various actions that might be performed by the function `init()`.

By default, the function `glClear()` makes the window black. Its precise action is determined by the initialization function

```
glClearColor(red, green, blue, alpha);
```

which defines the colours to be used by `glClear()`. (The argument `alpha` determines blending, an advanced feature.)

GLUT provides a **viewing volume** defined by

$$\begin{aligned} \texttt{left} &\le x \le \texttt{right}, \\ \texttt{bottom} &\le y \le \texttt{top}, \\ \texttt{near} &\le z \le \texttt{far}. \end{aligned}$$

In other words, an object at (x, y, z) will be displayed only if x, y, and z satisfy these inequalities. The default values are `left` = `bottom` = `right` = -1 and `right` = `top` = `far` = 1.

The viewing volume is determined by the current **projection matrix**. To change the projection matrix, we must: (1) announce that we are changing the projection matrix; (2) overwrite the current projection matrix with the identity matrix; and (3) multiply the identity matrix by the new matrix. Section 4 describes matrix transformations in detail. Figure 4 shows an initialization function that clears to white and defines a viewing volume. `glOrtho()` defines the simplest kind of projection: Z coordinates are simply ignored. To obtain a realistic view of a three-dimensional model, you will need a perspective transformation defined by `glFrustum()` or `gluPerspective()` described in Section 4.3.

```
void init (void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(left, right, bottom, top, near, far);
}
```

Figure 4: Initializing

1.4 OpenGL Types

OpenGL defines a number of types that are usually equivalent to C types. You can use C types but, if you want to write portable OpenGL programs, it is better to use the OpenGL types: see Table 1. The suffix column contains the letter that is used in function names; for example, the "`f`" in `glVertex3f`.

Suffix	Data type	C Type	OpenGL Type
`b`	8-bit integer	`signed char`	`GLbyte`
`s`	16-bit integer	`short`	`GLshort`
`i`	32-bit integer	`int` or `long`	`GLint`, `GLsizei`
`f`	32-bit floating point	`float`	`GLfloat`, `GLclampf`
`d`	64-bit floating point	`double`	`GLdouble`, `GLclampd`
`ub`	8-bit unsigned integer	`unsigned char`	`GLubyte`, `GLbooean`
`us`	16-bit unsigned integer	`unsigned short`	`GLushort`
`ui`	32-bit unsigned integer	`unsigned int`	`GLuint`, `GLenum`, `GLbitfield`

Table 1: OpenGL Types

2 Callbacks

GLUT handles events with **callback functions**. If you want to handle an event, such as a keystroke or mouse movement, you write a function that performs the desired action. Then you **register** your function by passing its name as an argument to a function with a name of the form `glut...Func()`, in which "`...`" indicates which callback you are registering.

Throughout this section, we will assume that our program uses the display function shown in Figure 5. This function creates a viewing matrix that rotates the origin about all three axes. The variables `x_angle`, `y_angle`, and `z_angle` are global `float`s set by callback functions.

```
void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(x_angle, 1.0, 0.0, 0.0);
    glRotatef(y_angle, 0.0, 1.0, 0.0);
    glRotatef(z_angle, 0.0, 0.0, 1.0);
    glutWireCube(1.0);
    glutSwapBuffers();
}
```

Figure 5: Displaying a Rotated Cube

The example program `cube.c` displays this cube at a fixed angle and does nothing else. In the following sections, we explain how to modify `cube.c` by adding callbacks. The programs mentioned do what they are claimed to do, but their code may be slightly more complex than shown here.

2.1 The Idle Function

OpenGL calls the **idle function** when it has nothing else to do. A common application of the idle function is to provide continuous animation. The example program `spin.c` displays a cube spinning about the X axis. The following notes explain how to derive this program from `cube.c`.

1. The main program must register the idle function. The following call registers an idle function called `spin()`:

   ```
   glutIdleFunc(spin);
   ```

2. The idle function must be declared. It has no arguments and returns no results. The following idle function increments the global variable `angle` and requests that the model be redisplayed.

```
void spin (void)
{
    x_angle += 1.0;
    glutPostRedisplay();
}
```

Notice that, after changing the angle, `spin()` does not call `display()` directly. Instead, it calls `glutPostRedisplay()`, which informs `glutMainLoop()` to redisplay the model at the next convenient opportunity.

2.2 The Keyboard Function

If the program has registered a keyboard callback function, this function is called whenever the user presses a key. The following notes explain how to derive the example program `keys.c` from `cube.c`. The program responds to the keys '`x`', '`y`', and '`z`' by rotating 10° about the corresponding axis.

1. Register a keyboard callback function.

```
glutKeyboardFunc(readkeys);
```

2. The keyboard function is declared as in Figure 6. It receives three arguments: the key that was pressed, and the current X and Y coordinates of the mouse.

2.3 Mouse Buttons

There are two ways of using the mouse and two ways of responding to mouse activity. In this example, the buttons and the X position of the mouse are used to set the rotation angles of the cube: the left button sets the X angle, and so on.

1. Register a mouse callback function.

```
glutMouseFunc(move);
```

2. The mouse callback function is passed four arguments.
   (a) The first argument specifies the button. Its value is one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`.
   (b) The second argument specifies the button state. Its value is one of `GLUT_DOWN` (the button has been pressed) or `GLUT_UP` (the button has been released).
   (c) The remaining two parameters are the X and Y coordinates of the current position of the mouse.

   Figure 7 shows a mouse function that uses the buttons and the X position of the mouse.

```
void readkeys (unsigned char key, int x, int y)
{
    switch (key)
    {
        case 'x':
            x_angle += 10.0;
            glutPostRedisplay();
            break;
        case 'y':
            y_angle += 10.0;
            glutPostRedisplay();
            break;
        case 'z':
            z_angle += 10.0;
            glutPostRedisplay();
            break;
        default:
            break;
    }
}
```

Figure 6: A keyboard event callback function

```
void mousepos (int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        x_angle = x * 360.0 / SIZE;
        glutPostRedisplay();
    }
    else if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
    {
        y_angle = x * 360.0 / SIZE;
        glutPostRedisplay();
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    {
        z_angle = x * 360.0 / SIZE;
        glutPostRedisplay();
    }
}
```

Figure 7: A mouse event callback function

2.4 Dragging the Mouse

The program can also respond to "dragging" events in which the user holds a mouse button down and moves the mouse.

1. Register a callback function for mouse movements:

   ```
   glutMotionFunc(drag);
   ```

2. The motion function is given the X and Y coordinates of the mouse. Figure 8 shows a motion function that uses the position of the mouse to set `z_angle` and `x_angle`. Since `SIZE` is the size of the (square) window, the angles vary from 0° to 360° as the mouse is moved from the left to the right or the bottom to the top of the window.

```
void drag (int x, int y)
{
    z_angle = 360.0 * x / SIZE;
    x_angle = 360.0 * y / SIZE;
    glutPostRedisplay();
}
```

Figure 8: A mouse motion function

The function `glutPassiveMotionFunc()` works in the same way but responds to mouse movement when no buttons have been pressed.

2.5 Resizing the Graphics Window

Whenever the user moves or resizes the graphics window, OpenGL informs your program, provided that you have registered the appropriate callback function. The main problem with resizing is that the user is likely to change the shape of the window as well as its size; you do not want the change to distort your image. For an example, see `reshape.c`.

1. Register a callback function that will respond to window resizing:

   ```
   glutReshapeFunc(reshape);
   ```

2. Figure 9 shows a function that responds to a resize request. The arguments are the width and height of the new window, in pixels. The function first updates the viewport so that it covers the entire window. Next, it changes the view to match the shape of the new window. The first argument of `gluPerspective()` specifies a 30° viewing angle; the second argument ensures that the aspect ratio of the view amtches that of the window.

```
void reshape (int width, int height)
{
    double aspect_ratio = (double) width / (double) height;
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, aspect_ratio, 1.0, 10.0);
}
```

Figure 9: Responding to resizing

3 Primitive Objects

The primitives provided by OpenGL include points, lines, and polygons. There are also various short-cuts for drawing the sets of related polygons (usually triangles) that approximate three-dimensional surfaces.

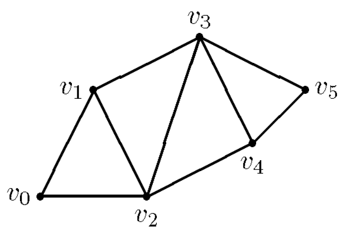
The code for drawing primitives has the general form

```
glBegin(mode);
.....
glEnd();
```

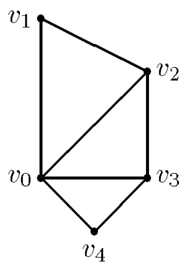
Table 2 explains what OpenGL expects for each value of the parameter `mode`. Assume, in each case, that the code specifies n vertices $v_0, v_1, v_2, \ldots, v_{n-1}$. For some modes, n should be a multiple of something: e.g., for `GL_TRIANGLES`, n will normally be a multiple of 3. It is not an error to provide extra vertices: but OpenGL will ignore them. The polygon drawn in mode `GL_POLYGON` must be convex.

Mode Value	Effect
`GL_POINTS`	Draw a point at each of the n vertices.
`GL_LINES`	Draw the unconnected line segments v_0v_1, v_2v_3, $\ldots v_{n-2}v_{n-1}$.
`GL_LINE_STRIP`	Draw the connected line segments v_0v_1, v_1v_2, $\ldots$, $v_{n-2}v_{n-1}$.
`GL_LINE_LOOP`	Draw a closed loop of lines v_0v_1, v_1, v_2, $\ldots$, $v_{n-2}v_{n-1}$, $v_{n-1}v_0$.
`GL_TRIANGLES`	Draw the triangle $v_0v_1v_2$, then the triangle $v_3v_4v_5$, and so on.
`GL_TRIANGLE_STRIP`	Draw the triangle $v_0v_1v_2$, then use v_3 to draw a second triangle, and so on (see Figure 10 (a)).
`GL_TRIANGLE_FAN`	Draw the triangle $v_0v_1v_2$, then use v_3 to draw a second triangle, and so on (see Figure 10 (b)).
`GL_QUADS`	Draw the quadrilateral $v_0v_1v_2v_3$, then the quadrilateral $v_4v_5v_6v_7$, and so on.
`GL_QUAD_STRIP`	Draw the quadrilateral $v_0v_1v_2v_3$, then the quadrilateral $v_2v_3v_5v_4$, and so on (see Figure 10 (c)).
`GL_POLYGON`	Draw a single polygon using v_0, v_1, $\ldots$, v_{n-1} as vertices ($n \geq 3$).

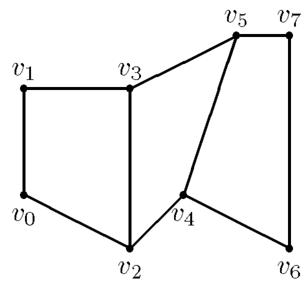
Table 2: Primitive Specifiers



(a) `GL_TRIANGLE_STRIP`



(b) `GL_TRIANGLE_FAN`



(c) `GL_QUAD_STRIP`

Figure 10: Drawing primitives

3.1 The Coordinate System

When we draw an object, where will it be? GL objects have four coordinates: x, y, z, and w. The first two, x and y must always be specified. The z coordinate defaults to 0. The w coordinate defaults to 1. (Section 4 explains the use of w.) Consequently, a call to `glVertex()` may have 2, 3, or (occasionally) 4 arguments.

Objects are visible only if they are inside the **viewing volume**. By default, the viewing volume is the 2×2 cube bounded by $-1 \le x \le 1$, $-1 \le y \le 1$, and $-1 \le z \le 1$, with the screen in the plane $z = 0$: see Figure 11. The projection is **orthographic** (no perspective), as if we were viewing the window from infinitely far away.

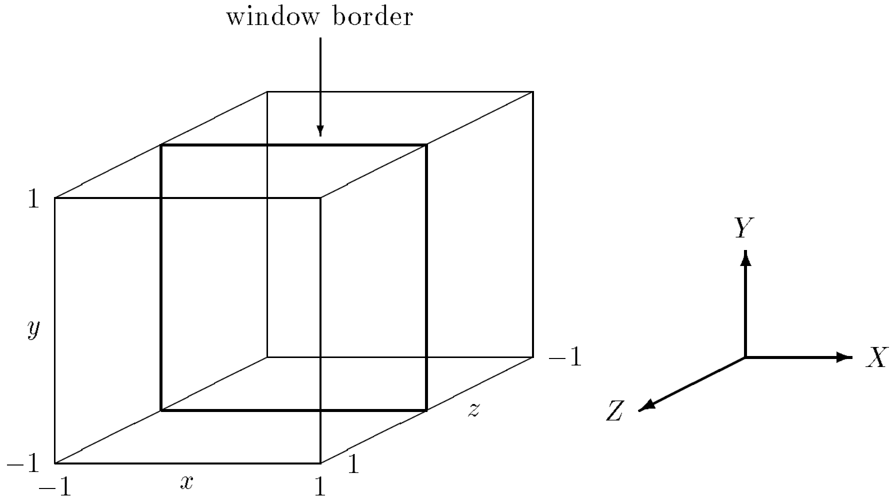


Figure 11: Default viewing volume

The following code changes the boundaries of the viewing volume but maintains an orthographic projection:

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

An object at (x, y, z) will be visible if `left` $\le x \le$ `right`, `bottom` $\le y \le$ `top`, and $-$`near` $\le z \le -$`far`. Note the sign reversal in the Z-axis: this convention ensures that the coordinate system is right-handed.

3.2 Points

The size of a point is set by `glPointSize(size)`. The default value of `size` is 1.0.

3.3 Lines

The width of a line is set by `glLineWidth(size)`. The default value of `size` is 1.0.

To draw stippled (or dashed) lines, use the following code:

```
glEnable(GL_LINE_STIPPLE);
glLineStipple(factor, pattern);
```

The `pattern` is a 16-bit number in which a 0-bit means "don't draw" and a 1-bit means "do draw". The `factor` indicates how much the pattern should be "stretched".

Example: After the following code has been executed, lines will be drawn as 10×10 pixel squares separated by 10 pixels (note that $\mathtt{A}_{16} = \mathtt{1010}_2$).

```
glEnable(GL_LINE_STIPPLE);
glLineWidth(10.0);
glLineStipple(10, 0xAAAA);
```

3.4 Polygons

The modes `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP` are straightforward. The other modes all describe polygons of one sort or another.

The way in which polygons are drawn is determined by calling `glPolygonMode(face, mode);`. Table 3 shows the possible values of `face` and `mode`.

face	mode
GL_FRONT_AND_BACK (default)	GL_FILL (default)
GL_FRONT	GL_LINE
GL_BACK	GL_POINT

Table 3: Options for drawing polygons

How do we decide which is the front of a polygon? The rule that GL uses is: if the vertices appear in a counter-clockwise order on the viewing window, we are looking at the front of the polygon. If we draw a solid object, such as a cube, the "front" of each face should be the "outside". The distinction between front and back does not matter in the default mode, in which the front and back sides of a polygon are treated in the same way.

Example: Suppose that we want to draw a unit cube centered at the origin, with vertices numbered as in Figure 12. Figure 13 shows the code required.

We begin by declaring an array `pt` giving the coordinates of each vertex. Next, we declare a function `face()` that draws one face of the cube. Finally, we declare a function `cube()` that draws six faces, providing the vertices in counter-clockwise order when viewed from outside the cube.

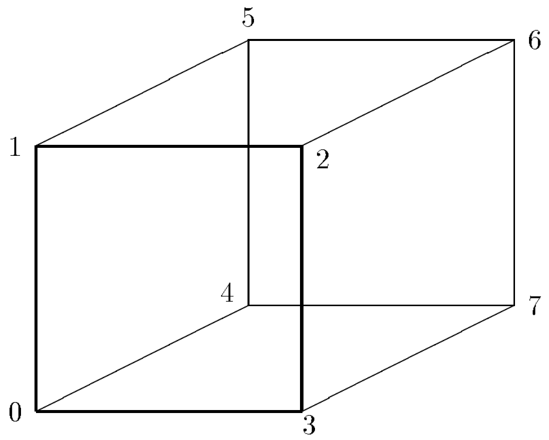


Figure 12: Labelling the corners of a cube

Polygons must be **convex**. The effect of drawing a concave or re-entrant (self-intersecting) polygon is undefined. If necessary, convert a concave polygon to two or more convex polygons by adding internal edges.

Colouring The function call `glShadeModel(mode)` determines the colouring of polygons. By default, `mode` is `GL_SMOOTH`, which means that GL will use the colours provided at each vertex and interpolated colours for all other points. For example, if you draw a rectangle with red corners and the left and blue corners at the right, the left edge will be red, the right edge will be blue, and the other points will be shades of purple.

If `mode` is `GL_FLAT`, the entire polygon has the same colour as its first vertex.

A set of three points defines a plane. If there are more than three points in a set, there may be no plane that contains them. GL will draw non-planar "polygons", but lighting and shading effects may be a bit strange.

Stippling To draw stippled polygons, execute the following code:

```
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(pattern);
```

The argument `pattern` is a pointer to a 36×36 pixel bitmap (1024 bits = 128 bytes = 256 hex characters).

Rectangles Since rectangles are used often, GL provides a special function, `glRect()` for drawing rectangles in the $z = 0$ plane. This function is defined in Section 7.

```
typedef Point GLfloat[3];
Point pt[] = {
    { -0.5, -0.5, -0.5 },
    { -0.5,  0.5, -0.5 },
    {  0.5,  0.5, -0.5 },
    {  0.5, -0.5, -0.5 },
    { -0.5, -0.5,  0.5 },
    { -0.5,  0.5,  0.5 },
    {  0.5,  0.5,  0.5 },
    {  0.5, -0.5,  0.5 } };

void face (int v0, int v1, int v2, int v3)
{
    glBegin(GL_POLYGON);
        glVertex3fv(pt[v0]);
        glVertex3fv(pt[v1]);
        glVertex3fv(pt[v2]);
        glVertex3fv(pt[v3]);
    glEnd();
}

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

Figure 13: Drawing a cube

4 Transformations

The effect of a **transformation** is to move an object, or set of objects, with respect to a coordinate system. Table 4 describes some common transformations. The least important transformation is shearing; OpenGL does not provide primitive functions for it.

Name	Effect
Translate	Move or slide along a straight line
Rotate	Rotate, spin, or twist, about an axis through the origin
Scale	Make an object larger or smaller
Shear	Turn rectangles into parallelograms
Perspective	Give illusion of depth for 3D objects

Table 4: Common Transformations

We can use transformations on:

- the model;
- parts of the model;
- the camera (or eye) position; and
- the camera "lens".

In OpenGL, the first three of these are called "model view" transformations and the last is called the "projection" transformation. Note that model transformations and camera transformations are complementary: for every transformation that changes the model, there is a corresponding inverse transformation that changes the camera and has the same effect on the screen.

The OpenGL functions that are used to effect transformations include `glMatrixMode()`, `glLoadMatrix()`, `glTranslatef()`, `glRotatef()`, and `glScalef()`. Section 7 contains descriptions of these functions.

4.1 Model View Transformations

A transformation is represented by a matrix. OpenGL maintains two current matrices: the **model view matrix** and the **projection** matrix. `glMatrixMode()` selects the current matrix; `glLoadIdentity()` initializes it; and the other functions change it. In the example below, the left column shows typical code and the right column shows the effect in pseudocode. M is the current matrix, I is the identity matrix, T is a translation matrix, R is a rotation matrix, and $\times$ denotes matrix multiplication. The final value of M is $I \times T \times R = T \times R$.

`glMatrixMode(GL_MODELVIEW);`	
`glLoadIdentity();`	$M := I$
`glTranslatef(x,y,z);`	$M := M \times T$
`glRotatef(a,x,y,z);`	$M := M \times R$

You might think it natural to draw something and then transform it. OpenGL, however, works the other way round: you define the transformations first and then you draw the objects. This seems backwards, but actually provides greater flexibility in the drawing process, because we can interleave drawing and transformation operations. Consider the following pseudocode:

```
initialize model view matrix
draw object P1
do transformation T1
draw object P2
do transformation T2
draw object P3
.....
```

The object `P1` is drawn without any transformations: it will appear exactly as defined in the original coordinate system. When the object `P2` is drawn, transformation `T1` is in effect; if `T1` is a translation, for example, `P2` will be displaced from the origin. Then `P3` is drawn with transformations `T1` and `T2` (that is, the transformation `T1` $\times$ `T2`) in effect.

In practice, we usually define a camera transformation first, before drawing any objects. This makes sense, because changing the position of the camera should change the entire scene. By default, the camera is positioned at the origin and is looking in the negative Z direction.

The easiest way to position the camera is to use the function `gluLookAt()`, which takes 9 (!) arguments, all `double`s. The first three arguments define the camera position with respect to the origin; the next three arguments are the coordinates of a point in the model towards which the camera is directed; and the last three arguments are the components of a vector pointing upwards. Here is a typical invocation of `gluLookAt()`:

```
gluLookAt ( 0.0, -10.0, -7.0,      // camera position
            0.0,   0.0,  0.0,      // camera direction
            0.0,   1.0,  0.0 );    // upward direction
```

Assume that the model is at the origin and has dimensions of the order of 5 units in each direction. Then the camera, at $(0, -10, -7)$, is outside the model and pointing towards it. In the model, "upwards" means the positive Y direction, and the vector $(0, 1, 0)$ specifies this.

Although we can think of a transformation as moving either the objects or the coordinate system, it is often simpler to think of the coordinates moving. The following example illlustrates this way of thinking.

> **Example** The program `balance.c` draws a sphere on the top of a cone which is standing on a cube. Figure 14 shows the display callback function that constructs this model (the line numbers are not part of the program).
> Line 3 clears the buffer, line 4 selects the model view matrix, and line 5 initialize the matrix. The first object drawn is a unit cube (line 6); since no transformations are in effect, the cube is at the origin. Since the cone must sit on top of the cube, line 7 translates the coordinate system up by half a unit.
> The axis of the cone provided by GLUT is the Z axis, but we need a cone with its axis in the Y direction. Consequently, we rotate the coordinate system (line 8), draw the cone (line 9), and rotate the coordinate system back again. Note that the rotation in line 9:

```
1   void display (void)
2   {
3      glClear(GL_COLOR_BUFFER_BIT);
4      glMatrixMode(GL_MODELVIEW);
5      glLoadIdentity();
6      glutWireCube(1.0);
7      glTranslatef(0.0, 0.5, 0.0);
8      glRotatef(-90.0, 1.0, 0.0, 0.0);
9      glutWireCone(0.5, 2.0, 15, 15);
10     glRotatef(90.0, 1.0, 0.0, 0.0);
11     glTranslatef(0.0, 2.5, 0.0);
12     glutWireSphere(0.5, 15, 15);
13  }
```

Figure 14: A balancing act

1. does **not** affect the cone, which has already been drawn; and
2. avoids confusion (if it it was not there, we would have a coordinate system with a horizontal Y axis and a vertical Z axis).

Line 11 moves the origin to the centre of the sphere, which is $2.0 + 0.5$ units (the height of the cone plus the radius of the sphere) above the top face of the cube. Line 12 draws the sphere.
The program `balance.c` has a transformation inserted between lines 10 and 11: this transformation does not affect the cube (which was drawn in line 6) but does affect both the cone and the sphere, which move together, as if attached.

The example demonstrates that we can start at the origin (the "centre" of the model) and move around drawing parts of the model. Sometimes, however, we have to get back to a previous coordinate system. Consider drawing a person, for instance: we could draw the body, move to the left shoulder, draw the upper arm, move to the elbow joint, draw the lower arm, move to the wrist, and draw the hand. Then what? How do we get back to the body to draw the right arm?

The solution that OpenGL provides for this problem is a **matrix stack**. There is not just one model view matrix, but a stack that can contain 32 model view matrices. (The specification of the API requires a stack depth of at least 32 matrices; implementations may provide more.) The function `glPushMatrix()` pushes the current matrix onto the stack, and the function `glPopMatrix()` pops the current matrix off the stack (and loses it). Using these functions, we could rewrite lines 8–10 of the example as:

```
glPushMatrix();
glRotatef(-90.0, 1.0, 0.0, 0.0);
glutWireCone(0.5, 2.0, 15, 15);
glPopMatrix();
```

The stack version is more efficient because it is faster to copy a matrix than it is to construct a new matrix and perform a matrix multiplication. It is always better to push and pop than to apply inverse transformations.

4.2 Projection Transformations

The usual practice in OpenGL coding is to define a projection matrix once and not to apply any transformations to it. The projection matrix can be defined during initialization, but a better place to define it is the reshape function. If the user changes the shape of the viewing window, the program should respond by changing the projection transformation so that the view is not distorted.

The following code uses the default settings:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

This code defines an orthographic projection (no perspective) and defines the **viewing volume** to be a $2 \times 2 \times 2$ unit cube centered at the origin. The six arguments are, respectively: minimum X, maximum X, minimum Y, maximum Y, minimum Z, and maximum Z.

Figure 15 shows a typical callback function that responds to changes in window size by changing the projection matrix. The **aspect ratio** of a window is its width divided by its height (television and older movies have an aspect ratio of 4/3; widescreen movies have aspect ratios as high as 2/1.) The code ensures that the viewing volume will be at least $10 \times 10 \times 10$, whatever the aspect ratio of the new window.

```
void reshape (GLsizei width, GLsizei height)
{
    float hvvw;       // half width of viewing volume
    float hvvh;       // half height of viewing volume
    if (width >= height)
    {
        hvvh = 5.0;
        hvvw = 5.0 * (float) width / (float) height;
    }
    else
    {
        hvvh = 5.0 * (float) height / (float) width;
        hvvw = 5.0;
    }
    glViewport(0, 0, width, height);   // use the entire window
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-hvvw, hvvw, -hvvh, hvvh, -5.0, 5.0);
}
```

Figure 15: Adjusting the projection to match the window shape

4.3 Perspective Transformations

It is not surprising that distant objects appear to be smaller than close objects: if an object is further from our eyes, it will subtend a smaller angle at the eye and hence create a smaller image on the retina. A projection that captures this phenomenon in a two-dimensional view is called a **perspective projection**.

The difference between an orthographic projection and a perspective projection lies in the shape of the viewing volume. As we have seen, the viewing volume for an orthographic projection is a cuboid (rectangular box). The viewing volume for a perspective projection is a **frustrum**, or truncated pyramid. The apex of the pyramid is at the eye; and the top and bottom of the frustrum determine the distances of the nearest and furthest points that are visible. The sides of the pyramid determine to visible objects in the other two dimensions.

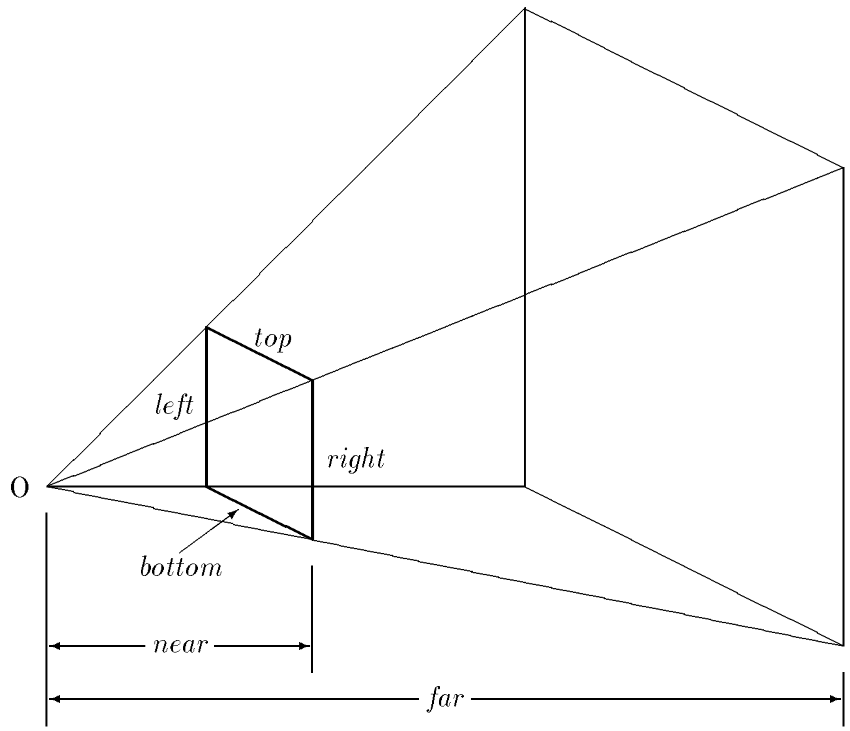


Figure 16: `glFrustum(left, right, bottom, top, near, far)`

OpenGL provides the function `glFrustum()` to define the frustrum directly, as shown in Figure 16. The eye and the apex of the pyramid are at the point labelled O at the left of the diagram. We can think of the closer vertical plane as the screen: the closest objects lie in the plane of the screen and other objects lie behind it. The first four arguments determine the range of X and Y values as follows: `left` corresponds to minimum X; `right` corresponds to maximum X; `bottom` corresponds to minimum Y; and `top` corresponds to maximum Y.

Note that `near` and `far` do **not** correspond directly to Z values. Both `near` and `far` must be positive, and visible Z values lie in the range $-\texttt{far} \le Z \le -\texttt{near}$.

The function `gluPerspective()` provides another way of defining the frustrum that is more intuitive and sometimes more convenient. The first argument defines the vertical angle that the viewport subtends at the eye in degrees. This angle is called α in Figure 17; its value must satisfy $0° < \alpha < 180°$. The height of the window is $h = 2 \cdot \texttt{near} \cdot \tan(\alpha/2)$. The second argument, `ar` in Figure 17, is the **aspect ratio**, which is the width of the viewport divided by its height. Thus $w = \texttt{ar} \cdot h = 2 \cdot \texttt{ar} \cdot \texttt{near} \cdot \tan(\alpha/2)$. The last two arguments, `near` and `far`, determine the nearest and furthest points visible.

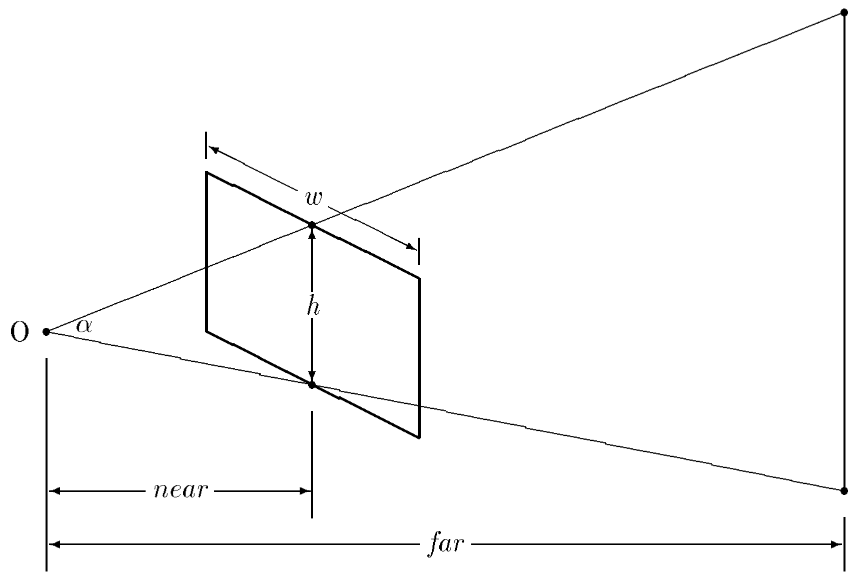


Figure 17: `gluPerspective(alpha, ar, near, far)`

The model view transformation and the projection transformation must work together in such a way that at least part of the model lies in the viewing volume. If it doesn't, we won't see anything.

Both projection transformations assume that our camera (or eye) is situated at the origin and is looking in the negative Z direction. (Why negative Z, rather than positive Z? By convention, the screen coordinates have X pointing right and Y pointing up. If we want a right-handed coordinate system, Z must point towards the viewer.) Consequently, we should position our model somewhere on the negative Z axis.

Figure 18 demonstrates how to use `gluLookAt()` and `gluPerspective()` together. The two important variables are `alpha` and `dist`.

The value of `alpha` determines the vertical angle subtended by the viewport at the user's eye. Figure 19 shows the effect of a small value of `alpha`. The model is a wire cube and its image is half as large as the viewport. The effect is that of viewing the cube from a large distance: the back of the cube is only slightly smaller than the front.

Figure 20 shows the effect of a large value of `alpha`. Again, the model is a wire cube and its image occupies half of the viewport. The effect is of viewing the cube from a small distance: the back of the cube is much smaller than the front.

Making `alpha` smaller without moving the model simulates "zooming in": the image gets larger in the viewport. Making `alpha` larger simulates "zooming out": the image gets smaller.

```
const int SIZE = 500;

float alpha = 60.0;
float dist = 5.0;

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

void init (void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(alpha, 1.0, dist - 1.0, dist + 1.0);
}

int main (int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitWindowSize(SIZE, SIZE);
    glutInitWindowPosition(100, 50);
    glutCreateWindow("A Perspective View");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Figure 18: Using `gluLookAt()` and `gluPerspective()`

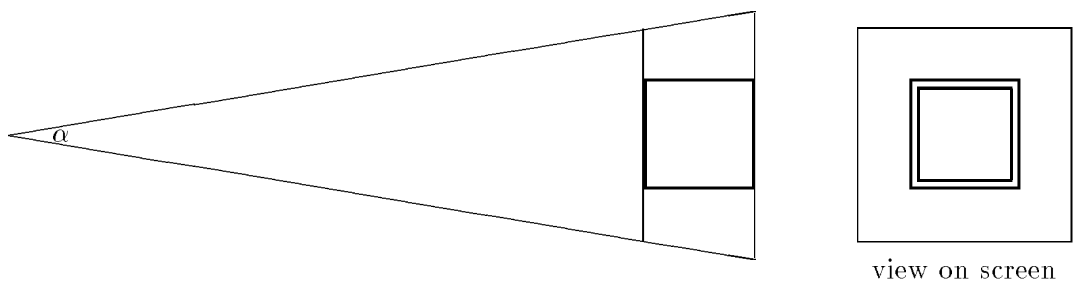


Figure 19: `gluPerspective()` with a small value of α

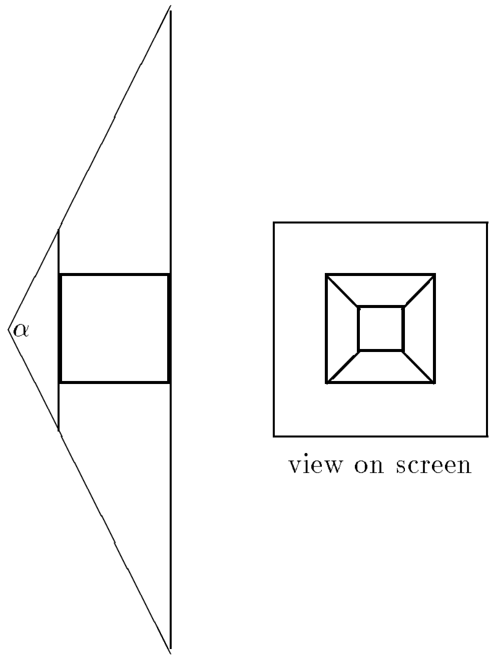


Figure 20: `gluPerspective()` with a large value of α

Although we can use `alpha` to manipulate the image in any way we please, there is a correct value for it. Suppose that the user is sitting at a distance d from a screen of height h: the appropriate value of `alpha` is given by

$$\alpha = 2\tan^{-1}\left(\frac{h}{2\,d}\right).$$

For example, if $h = 12\,\mathrm{in}$ and $d = 24\,\mathrm{in}$, then

$$\begin{aligned} \alpha &= 2\tan^{-1}\left(\frac{12}{2\times 24}\right) \\ &\approx 28^\circ. \end{aligned}$$

Photographers should note that a 35 mm camera with a normal 50 mm lens corresponds to $\alpha = 26^\circ$. A 24 mm wide-angle lens corresponds to $\alpha = 45^\circ$.

Changing the value of `dist` in Figure 18 also changes the size of the image in the viewport, but in a different way. The perspective does not change and, consequently, the shape (or "distortion") of the image does not change. The value of `dist` corresponds to the distance between the viewer and the object: increasing it has the same effect as walking away from the model.

It is possible to change `alpha` and `dist` together in such a way that the size of a key object in the model stays the same while the perspective changes. This is a rather simple trick in OpenGL, but it is an expensive effect in movies or television because the zoom control of the lens must be coupled to the tracking motion of the camera. Hitchcock used this trick to good effect in his movies *Vertigo* and *Marnie*.

5 Menus

GLUT provides menus. The format of menus is restricted but the functions are easy to use. Follow the following steps to define a menu without submenus.

1. Call `glutCreateMenu(func)` to create the menu. `func` is the name of a function that will handle menu-selection events (see item 4 below).
2. Call `glutAddMenuEntry(string,num)` for each menu item. `string` is the string that will be displayed in the menu. `num` is a constant that will be returned by the callback function when the user selects this item. You must ensure that all selections have different values.
3. Call `glutAttachMenu(button)` to "attach" the menu to a mouse button. `button` should be one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`.
4. Write the callback function for the menu. This function has one integer argument and returns nothing. The value passed to it will be one of the constants that you chose in step 2.

Figure 21 shows an example of the code required for a simple menu. It is not much more complicated to create a menu with submenus.

1. To create a submenu, call `glutCreateMenu()` as before. This time, however, you should keep a copy of the integer that it returns. Suppose, for example, that you write

   ```
   int sub1 = glutCreateMenu(get_key);
   ```

2. Complete the submenu with calls to `glutAddMenuEntry()`, but do not attach it to a button.
3. Write code for the main menu **after** the submenus. (That is, build the menu "tree" bottom-up.) To add a submenu to the main menu, call `glutAddSubMenu(string,num)`. `string` is the text that appears in the main menu to identify the submenu and `num` is the value returned by `glutCreateMenu()` in the first step.

Note that the main menu can contain both simple entries (added with `glutAddMenuEntry()`) and submenus (added with `glutAddSubMenu()`). The program `menu.c` provides a simple example of nested menus.

```
/* Constants that identify menu selections.*/
#define M_RED       20
#define M_GREEN     21
#define M_BLUE      22

/* Menu callback function.
void choose_colour (int sel)
{
    switch (sel)
    {
        case M_RED:
            /* Action for red selection. */
            break;
        case M_GREEN:
            /* Action for green selection. */
            break;
        case M_BLUE:
            /* Action for blue selection. */
            break;
        default:
            break;
    }
}

/* This function sets up the menu.
 * It must be called during initialization.
 */
void initialize_menu (void)
{
    glutCreateMenu(choose_colour);
    glutAddMenuEntry("Red", M_RED);
    glutAddMenuEntry("Green", M_GREEN);
    glutAddMenuEntry("Blue", M_BLUE);
    glutAddMenuEntry("Quit", M_QUIT);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

Figure 21: Code for a simple menu

6 Lighting

We talk about "three dimensional" graphics but, with current technology, the best we can do is to create the illusion of three-dimensional objects on a two-dimensional screen. Perspective is one kind of illusion, but it is not enough. A perspective projection of a uniformly-coloured cube, for example, looks like an irregular hexagon. We can paint the faces of the cube in different colours, but this does not help us to display a realistic image of a cube of one colour.

We can improve the illusion of depth by simulating the lighting of a scene. If a cube is illuminated by a single light source, its faces have different brightnesses, and this strengthens the illusion that it is three dimensional. This section describes some of the features that OpenGL provides for lighting a scene.

To light a model, we must provide OpenGL with information about the objects in the scene and the lights that illuminate them. To obtain realistic lighting effects, we must define:

1. normal vectors for each vertex of each object;
2. material properties for each object;
3. the positions and other properties of the light sources; and
4. a lighting model.

Since OpenGL provides reasonable defaults for many properties, we have to add only a few function calls to a program to obtain simple lighting effects. For full realism, however, there is quite a lot of work to do.

6.1 A Simple Example

Suppose that we have a program that displays a sphere. We must eliminate hidden surfaces by passing `GLUT_DEPTH` to `glutInitDisplayMode()` and by clearing `GL_DEPTH_BUFFER_BIT` in the display function. The following steps would be necessary to display an illuminated sphere.

1. Define normal vectors. If we use the GLUT function `glutSolidSphere()`, this step is unnecessary, because normals are already defined for the sphere and other GLUT solid objects.
2. Define material properties. We can use the default properties.
3. Define light sources. The following code should be executed during initialization:

   ```
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   ```

4. Define a lighting model. We can use the default model.

This is the simplest case, in which we add just two lines of code. The following sections describe more of the details.

6.2 Normal Vectors

For any plane surface, there is a *normal vector* at right angles to it. A pencil standing vertically on a horizontal desktop provides an appropriate mental image.

OpenGL associates normals with vertices rather than surfaces. This seems surprising at first, but it has an important advantage. Often, we are using flat polygons as approximations to curved surfaces. For example, a sphere is constructed of many small rectangles whose edges are lines of latitude and longitude. If we provide a single normal for each rectangle, OpenGL will illuminate each rectangle as a flat surface. If, instead, we provide a normal at each vertex, OpenGL will illuminate the rectangles as if they were curved, and provide a better illusion of a sphere.

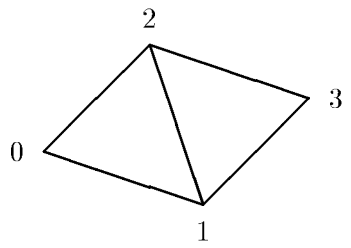


Figure 22: A simple object

Consequently, there are two ways of specifying normals. Consider the very simple object shown in Figure 22 consisting of two triangles. We could provide a single normal for each triangle using the following code:

```
glBegin(GL_TRIANGLES);
  glNormal3f(u0,v0,w0);  /* Normal for triangle 012.*/
  glVertex3f(x0,y0,z0);
  glVertex3f(x1,y1,z1);
  glVertex3f(x2,y2,z2);
  glNormal3f(u1,v1,w1);  /* Normal for triangle 132. */
  glVertex3f(x1,y1,z1);
  glVertex3f(x3,y3,z3);
  glVertex3f(x2,y2,z2);
glEnd();
```

The function `glNormal()` defines a normal vector for any vertex definitions that follow it. Note that it is possible for two normals to be associated with the same vertex: in the code above, vertices 1 and 2 each have two normals. As far as OpenGL is concerned, the code defines *six* vertices, each with its own normal.

The other way of describing this object requires a normal for each vertex. The normals at vertices 0 and 3 are perpendicular to the corresponding triangles. The normals at vertices 1 and 2 are computed by averaging the normals for the triangles. In the following code, each vertex definition is preceded by the definition of the corresponding normal.

```
glBegin(GL_TRIANGLES);
  glNormal3f(u0,v0,w0);   /* Normal to triangle 012 */
  glVertex3f(x0,y0,z0);
  glNormal3f(ua,va,wa);   /* Average */
  glVertex3f(x1,y1,z1);
  glNormal3f(ua,va,wa);   /* Average */
  glVertex3f(x2,y2,z2);

  glNormal3f(ua,va,wa);   /* Average */
  glVertex3f(x1,y1,z1);
  glNormal3f(u3,v3,w3);   /* Normal to triangle 132 */
  glVertex3f(x3,y3,z3);
  glNormal3f(ua,va,wa);   /* Average */
  glVertex3f(x2,y2,z2);
glEnd();
```

We do not usually see code like this in OpenGL programs, because it is usually more convenient to store the data in arrays. The following example is more typical; each component of the arrays `norm` and `vert` is an array of three `float`s.

```
glBegin{GL_TRIANGLES}
  for (i = 0; i < MAX; i++)
  {
    glNormal3fv(norm[i]);
    glVertex3fv(vert[i]);
  }
glEnd();
```

Each normal vector should be normalized (!) in the sense that its length should be 1. Normalizing the vector (x, y, z) gives the vector $(x/s, y/s, z/s)$, where $s = \sqrt{x^2 + y^2 + z^2}$.

If you include the statement `glEnable(GL_NORMALIZE)` in your initialization code, OpenGL will normalize vectors for you. However, it is usually more efficient to normalize vectors yourself.

Computing Normal Vectors The vector normal to a triangle with vertices (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) is (a, b, c), where

$$\begin{aligned} a &= + \begin{vmatrix} y_2 - y_1 & z_2 - z_1 \\ y_3 - y_1 & z_3 - z_1 \end{vmatrix} \\ b &= - \begin{vmatrix} x_2 - x_1 & z_2 - z_1 \\ x_3 - x_1 & z_3 - z_1 \end{vmatrix} \\ c &= + \begin{vmatrix} x_2 - x_1 & y_2 - z_1 \\ x_3 - x_1 & y_3 - z_1 \end{vmatrix} \end{aligned}$$

To compute the average of n vectors, simply add them up and divide by n. In practice, the division by n is not usually necessary, because we can simply add the normal vectors at a vertex and then normalize the results.

Formally, the normalized average vector of a set of vectors $\{ (x_i, y_i, z_i) \mid i = 1, 2, \ldots, n \}$ is $\left(\frac{X}{S}, \frac{Y}{S}, \frac{Z}{S}\right)$, where $X = \sum_{i=1}^{i=n} x_i$, $Y = \sum_{i=1}^{i=n} y_i$, $Z = \sum_{i=1}^{i=n} z_i$, and $S = \sqrt{X^2 + Y^2 + Z^2}$.

Figure 23 shows a simple function that computes vector normal for a plane defined by three points `p`, `q`, and `r`, chose from an array of points.

```
enum { X, Y, Z };
typedef float Point[3];
typedef float Vector[3];

Point points[MAX_POINTS];

void find_normal (int p, int q, int r, Vector v) {
    float x1 = points[p][X];
    float y1 = points[p][Y];
    float z1 = points[p][Z];
    float x2 = points[q][X];
    float y2 = points[q][Y];
    float z2 = points[q][Z];
    float x3 = points[r][X];
    float y3 = points[r][Y];
    float z3 = points[r][Z];
    v[X] = + (y2-y1)*(z3-z1) - (z2-z1)*(y3-y1);
    v[Y] = - (x2-x1)*(z3-z1) + (z2-z1)*(x3-x1);
    v[Z] = + (x2-x1)*(y2-y1) - (y2-y1)*(x3-x1);
    }
```

Figure 23: Computing normals

6.3 Defining Material Properties

When lighting is enabled, OpenGL ignores `glColor()` calls. Instead, it computes the colour of each point using information about the lights, the objects, and their relative positions. The function that defines the effect of lighting an object is `glMaterialfv()` and it requires three arguments.

1. The first argument determines which face we are defining. The possible values are `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

   It is usually best to use `GL_FRONT` because, in most situations, only front faces should be visible. (Recall that a face is at the "front" if its vertices are given in counterclockwise order as seen by the viewer.) You can specify different properties for front and back faces by calling `glMaterialfv()` twice, once with `GL_FRONT` and again with `GL_BACK`.

2. The second argument is a parameter name. Table 5 shows its possible values.

3. The third argument provides the value of the corresponding parameter: it is a pointer to an array.

The first column of Table 5 shows the parameter names accepted by `glMaterialfv()`; the second column describes the meaning of the parameter; and the third column shows the value that will be provided if you do not call `glMaterialfv()`.

Parameter	Meaning	Default
GL_DIFFUSE	diffuse colour of material	$(0.8, 0.8, 0.8, 1.0)$
GL_AMBIENT	ambient colour of material	$(0.2, 0.2, 0.2, 1.0)$
GL_AMBIENT_AND_DIFFUSE	ambient and diffuse	
GL_SPECULAR	specular colour of material	$(0.0, 0.0, 0.0, 1.0)$
GL_SHININESS	specular exponent	0.0
GL_EMISSION	emissive colour of material	$(0.0.0, 0.0, 1.0)$

Table 5: Parameters for `glMaterialfv()`

The diffuse colour of an object is its colour when it is directly illuminated; it is what we normally mean when we say, for example, "the book is blue". The ambient colour of an object is its colour when it is illuminated by the ambient (or "background") light in the scene.

For most objects, the diffuse and ambient colours are almost the same. For this reason, OpenGL provides the parameter name `GL_AMBIENT_AND_DIFFUSE` which sets the ambient and diffuse colours to the same value.

The specular colour of an object produces highlights when the object is directly illuminated. Smooth, polished surfaces give specular reflections but dull, matte objects do not. Specular highlights usually have the colour of the light source, rather than the object; consequently, if a specular colour is defined, it is usually white.

The specular exponent determines the smoothness of a specular surface. Its value must be between 0.0 and 128.0. A low value gives a large, dull highlight, and a high value gives a small, bright highlight.

Define an emissive colour for an object if it is emitting light. For example, a light bulb in your model would have an emissive colour of white or almost white. Defining an emissive colour does **not** make the object a source of light: you must put a light source at the same position in the scene to provide the light.

Figure 24 shows some examples of the use of `glMaterialfv()`. Properties do not automatically revert to their defaults: if you want to restore a default value, you must do so explicitly. For example, the shininess of the dark blue sphere must be turned off before drawing the dull red cube.

6.4 Defining the Lights

To use lighting, you must first enable it by calling `glEnable(GL_LIGHTING)`. You must also enable each light that you want. OpenGL provides eight lights, named `GL_LIGHT0` through `GL_LIGHT7`. To enable light n, call `glEnable(GL_LIGHT`n`)`.

The function `glLightfv()` specifies the properties of a light and is used in the same way as `glMaterialfv()`. There are three arguments: the first specifies a light, the second is a

```
/* Data declarations */
GLfloat off[] = { 0.0, 0.0, 0.0, 0.0 };
GLfloat white[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat red[] = { 1.0, 0.0, 0.0, 1.0 };
GLfloat deep_blue[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat shiny[] = { 50.0 };
GLfloat dull[] = { 0.0; }

/* Draw a small, dark blue sphere with shiny highlights */
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, deep_blue);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);
glMaterialfv(GL_FRONT, GL_SHININESS, shiny);
glutSolidSphere(0.2, 10, 10);

/* Draw a large, red cube made of non-reflective material */
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red);
glMaterialfv(GL_FRONT, GL_SPECULAR, off);
glMaterialfv(GL_FRONT, GL_SHININESS, dull);
glutSolidCube(10.0);

/* Draw a white, glowing sphere */
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, off);
glMaterialfv(GL_FRONT, GL_SPECULAR, off);
glMaterialfv(GL_FRONT, GL_SHININESS, dull);
glMaterialfv{GL_FRONT, GL_EMISSION, white);
glutSolidCube(10.0, 20, 20);
```

Figure 24: Using `glMaterial()`

parameter name, and the third is a value for that parameter. Table 6 shows the parameter names and their default values.

Parameter	Meaning	Default
`GL_DIFFUSE`	diffuse colour	$(1.0, 1.0, 1.0, 1.0)$
`GL_AMBIENT`	ambient colour	$(0.0, 0.0, 0.0, 1.0)$
`GL_SPECULAR`	specular colour	$(1.0, 1.0, 1.0, 1.0)$
`GL_POSITION`	position	$(0.0, 0.0, 1.0, 0.0)$
`GL_CONSTANT ATTENUATION`	constant attenuation	1.0
`GL_LINEAR_ATTENUATION`	linear attenuation	0.0
`GL_QUADRATIC_ATTENUATION`	quadratic attenuation	0.0
`GL_SPOT_DIRECTION`	direction of spotlight	$(0.0, 0.0, -1.0)$
`GL_SPOT_CUTOFF`	cutoff angle of spotlight	180.0
`GL_SPOT_EXPONENT`	exponent of spotlight	0.0

Table 6: Parameters for `glLightfv()`

The diffuse colour is the colour of the directional light that appears to come from the light itself. It is the colour that we normally associate with the light. The default is bright white.

The ambient colour is the contribution that the light makes to the ambient colour of the scene. Since ambient light pervades the scene, without appearing to come from any particular direction, the default value is black.

The specular colour is the colour given to highlights in specular objects that are illuminated by this light. It us usually the same as the diffuse colour. For example, a brown bottle illuminated by a white light has white highlights.

The position of the light can be specified in two ways: if the fourth component of the vector is 0.0, the light is called *directional* and the effect is that of a light source infinitely far way. If the fourth component is 1.0, the light source is called *positional* and the effect is that of a light at the corresponding point.

For example, the vector $(-1.0, 1.0, 0.0, 0.0)$ is a directional light source to the left of the scene ($x = -1.0$), above it ($y = 1.0$), and infinitely far away ($w = 0.0$). This position vector would be suitable for sunlight.

The vector $(0.0, 1.0, -5.0, 1.0)$ is a positional light source at the top centre ($x = 0.0$, $y = 1.0$) of the scene, and behind it ($z = -5.0$).

Directional lighting requires less computation than positional lighting because there are fewer angles to calculate.

In the real world, the intensity of a light decreases with distance. OpenGL calls this decrease "attenuation". There is no attenuation for a directional light because it is infinitely far away. OpenGL attenuates a light source by multiplying the intensity of the light by

$$\frac{1}{C + L\,d + Q\,d^2}$$

where C is the constant attenuation factor, L is the linear attenuation factor, Q is the quadratic attenuation factor, and d is the distance between the light and the object. The default values are $C = 1.0$ and $L = Q = 0.0$. Including linear and/or quadratic attenuation factors increases the realism of the scene at the expense of extra computation.

By default, a light radiates in all directions. A *spotlight* radiates a cone of light in a particular direction. Obviously, a directional light cannot be a spotlight. To turn a positional light into a spotlight, provide the following values:

- `GL_SPOT_DIRECTION` determines the direction in which the spotlight is pointing.
- `GL_SPOT_CUTOFF` determines the "spread" of the light. It is the angle between the axis of the cone and the edge of the light beam. The default value, 180°, corresponds to an omni-directional light (i.e. not a spotlight). If the angle is not 180°, it must be between 0° and 90°. The angle 5° gives a highly-focused searchlight effect; the angle 90° illuminates anything in front of the light but nothing behind it.
- `GL_SPOT_EXPONENT` determines how the light varies from the axis to the edges of the cone. The default value, 0.0, gives uniform light across the cone; a higher value focuses the light towards the centre of the cone.

6.5 Defining a Lighting Model

The function `glLightModel{if}[v]()` defines a lighting model. The first argument is a parameter name and the second argument is a value for that parameter. Table 7 shows the possible parameter names, their meanings, and their default values.

Parameter	Meaning	Default
`GL_LIGHT_MODEL_AMBIENT`	ambient light intensity	$(0.2, 0.2, 0.2, 1.0)$
`GL_LIGHT_MODEL_LOCAL_VIEWER`	simulate a close viewpoint	`GL_FALSE`
`GL_LIGHT_MODEL_TWO_SIDE`	select two-sided lighting	`GL_FALSE`

Table 7: Parameters for `glLightModel()`

The ambient light intensity determines the colour of the light that pervades the scene, independently of any actual lights. Its default value, dark grey, ensures that you will see something even if your lights are not switched on or are pointing in the wrong direction. In reality, most scenes do have a certain amount of ambient light. For example, a sunlit scene is illuminated not only by the sun (directional and bright yellow) but also by the sky (light blue and ambient). Dark greenish-blue ambient light would be suitable for an underwater scene. For special applications, such as deep space scenes, set the ambient light to black.

By default, OpenGL calculates specular reflections with the assumption that the scene is being viewed from infinitely far away. Specifying a local viewer by giving the value `GL_TRUE` provides more realistic results but requires a considerable amount of additional computation.

By default, OpenGL illuminates only the front sides of polygons. (This is true even if you have defined material properties for both sides of your polygons.) In many cases, the back sides of polygons are invisible anyway. For example, the back sides of the polygons of a sphere face the inside of the sphere. Specifying two-sided illumination by giving the value `GL_TRUE` tells OpenGL to illuminate any face of a polygon that is visible, at the expense of additional computation.

6.6 Putting It All Together

It is usually best to build a lit scene incrementally. Start by enabling lighting and turning on a light:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

At this point, you are using default values for the properties of the objects in your model and the lights. Next, add statements that provide explicit values when the defaults are not suitable. You will probably want to start by positioning your light (or lights) and then defining the surface properties of your objects. Some interactions may be surprising, although they are realistic. For example, a green object will appear black (and therefore invisible) in red light.

Points to note:

- Remember that OpenGL is a state machine. Once a call has been executed, its effects remain until undone. For example, if you have a number of objects and you make the first one red, all the others will be red unless you explicitly say otherwise.
- If your lights do not move, you can set them up during initialization. If you want moving lights, the calls that set position must be in your `display()` callback function. Usually, you will set light positions after choosing the viewpoint (e.g. with `gluLookAt()`) and before making any other transformations.
- An object which is a light source must be lit twice: once to make it visible and once to make it act as a light source. Figure 25 is an extract from the example program `planet.c` that shows how to make the sun both visible and a source of illumination.
- OpenGL has a rather simple view of illumination. It does not compute shadows and it does not recognize objects as obstacles in the light paths. In `planet.c`, the planet and its moon are illuminated by the sun, but the moon does not disappear when the planet is between it and the sun (lunar eclipse) and it does not throw a shadow on the planet (solar eclipse).

```
GLfloat dir[] = { 0.0, 0.0, 1.0, 0.0 };
GLfloat pos[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat sun_col[] = { 1.0, 1.0, 0.6, 0.0 };

.....

void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    /* Put a directional light on the positive Z axis to
       illuminate the sun. */
    glLightfv(GL_LIGHT0, GL_POSITION, dir);

    /* Choose viewpoint */
    glTranslatef(0.0, 0.0, -dist);

    /* Define solar properties and draw the sun */
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, sun_col);
    glutSolidSphere(0.4, 40, 40);

    /* Move the light to the centre of the sun so that it
       can illuminate the planets. */
    glLightfv(GL_LIGHT0, GL_POSITION, pos);

    /* Draw planet */
    .....
}
```

Figure 25: Illuminating the sun

7 Reference

Each function of OpenGL is specified by a prototype and a brief explanation. The explanations are brief: you may find the suggested examples help you to understand them. The specifications follow some conventions, listed below.

- Function descriptions are in alphabetical sequence by name. The prefix (`gl-`, `glu-`, or `glut-`) is not part of the name.
- If a function has many arguments of the same type, the argument types are omitted from the prototype but specified in the explanation that follows it.
- A name of the form `glname{234}{sifd}[v]` describes several functions. The name of a particular function is obtained by taking one character from each "set" `{...}` and optionally adding the character enclosed in brackets `[...]`.

  The digit gives the number of arguments and the letter gives the type. The final "`v`", if present, says that there will be a single argument that is a pointer to an array.

  When the function definition uses this form, the argument types are omitted because they must conform to the function name.
- The best way to understand these functions is to study the way in which they are used in the example programs, as indicated by the **Example** subheadings below. No examples are given for functions that appear in almost all programs.

Function Definitions

`void glutAddMenuEntry (char *label, int selection);`

Add an entry to the current menu. The first argument is the text that the user will see and the second argument is the integer that will be passed to the callback function when the user selects this entry.

See also

`glutCreateMenu()`.

`void glutAddSubMenu (char *label, int menu);`

Add a sub-menu to the current menu. The first argument is the text that the user will see and the second argument is an integer returned by `glutCreateMenu()`.

See also

`glutCreateMenu()`.

`void glutAttachMenu (int button);`

Associate the current menu with a mouse button. The value of the argument will be one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`.

See also

`glutCreateMenu()`.

void glBegin (GLenum mode);

Marks the beginning of a vertex data list. The permitted values of `mode` are given in Table 2 on page 12.

See also

`glEnd()`.

void glutBitmapCharacter (void *font, int character);

Displays one bitmapped character at the current raster position and moves the raster position right by the width of the character. Table 8 lists the permitted values of `font`.

See also

`glRasterPos()`, `glutStrokeCharacter()`.

int glutBitmapWidth(void *font, int character);

Returns the width of the character. Table 8 lists the permitted values of `font`.

Font name	Description
GLUT_BITMAP_8_BY_13	fixed width, 8×13 pixels
GLUT_BITMAP_9_BY_15	fixed width, 9×15 pixels
GLUT_BITMAP_TIMES_ROMAN_10	proportional width, 10 points
GLUT_BITMAP_TIMES_ROMAN_24	proportional width, 24 points
GLUT_BITMAP_HELVETICA_10	proportional width, 10 points
GLUT_BITMAP_HELVETICA_12	proportional width, 12 points
GLUT_BITMAP_HELVETICA_18	proportional width, 18 points

Table 8: Fonts for `glutBitmapCharacter()`

void glClear (GLbitfield mask);

Clears the specified buffers. Table 9 shows the names of the mask bits. Using multiple calls of `glClear()` may be less efficient than using `|` to "OR" mask bits.

void glClearColor (red, green, blue, alpha);

Set the clearing colour used for clearing the colour buffers in RGBA mode. The arguments are all of type `float` and they are clamped to the range $[0, 1]$.

Buffer	Mask bit
Colour	GL_COLOR_BUFFER_BIT
Depth	GL_DEPTH_BUFFER_BIT
Accumulation	GL_ACCUM_BUFFER_BIT
Stencil	GL_STENCIL_BUFFER_BIT

Table 9: Mask bits for clearing

Default

```
void glClearColor (0.0, 0.0, 0.0, 0.0);
```

void glColor{34}f[v] (...);

Set the current colour. The first three values give the intensities for red, green, and blue, respectively. The fourth argument, if present, gives the value of alpha to be used for blending. If `v` is present, there is one argument that points to an array of the appropriate size (3 or 4). The values of all arguments should be in the range $[0, 1]$. Integer arguments can be used, but the conversion rules are not given here.

Examples

`axes.c`.

int glutCreateMenu (void (*func)(int selection));

Create a menu and register a callback function that will be invoked when the user makes a selection from the menu. The value returned by `glutCreatemenu()` is an integer that identifies the menu. It is used as an argument to `glutCreateSubMenu()`, telling the submenu which menu it belongs to. The value passed to the callback function is an integer corresponding to the item selected from the menu by the user.

See also

`glutAddMenuEntry()`, `glutAddSubMenu()`, `glutAttachMenu()`.

Examples

`menu.c`.

int glutCreateSubWindow (win, x, y, width, height);

All arguments are integers. `win` is the number of the new window's parent. `(x,y)` is the location of the new window relative to the parent window's origin. `width` and `height` are the width and height of the new window.

int glutCreateWindow (char *title);

Creates a window with features that have been previusly defined. The string `title` will appear at the top of the window. The window is not actually displayed on the screen until `glutMainLoop()` has been entered.

The value returned is a unique identifier for the window. It is used in programs that support several graphics windows.

See also

`glutInitDisplayMode()`, `glutInitWindowPosition()`, `glutInitWindowSize()`.

void glutDestroyWindow(int win);

Destroy the window with number `win`.

void glDisable (GLenum capability);

Turns off one of GL's capabilities.

See also

`glEnable()`.

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

Register a callback function that displays the model in the graphics window. The display function has no arguments and does not return a value.

void glEnable (GLenum capability);

Turns on one of GL's capabilities. There are many capabilities, including `GL_DEPTH_TEST`, `GL_FOG`, and so on.

See also

`glDisable()`.

Examples

`planet.c`.

void glEnd();

Marks the end of a vertex data list introduced by `glBegin();`.

char *gluErrorString (int code);

Return a string describing the error indicated by `code`.

See also

`glGetError();`

void glFlush (void);

Forces previously issued OpenGL commands to begin execution. It is not usually necessary to call `glFlush()` for programs run on a single-user system, but it is important for programs that run on a network.

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

Defines a projection matrix and multiplies the current projection matrix by it. The matrix defines a "viewing volume" for a perspective view in the negative Z direction.

All arguments should be positive `double`s. The first four arguments determine the size of the viewing "window". Only objects whose distance from the viewpoint is between `near` and `far` will be drawn.

See Also

`gluPerspective()`, `glOrtho()`.

int glGetError (void);

Return the value of the error flag. If the result is `GL_NO_ERROR`, then no error has occurred. Otherwise, you can obtain a string describing the error by passing the result to `gluErrorString()`.

void glutIdleFunc (void (*func)(void));

Register a callback function that will be called when no other events are being processed.

Examples

`spin.c`.

void glutInit (int argc, char *argv[]);

Initialize the GLUT library. This function should be called before any other `glut-` functions. The arguments have the same type of the arguments of `main()` and are, by convention, the same as the arguments passed to `main()`.

void glutInitDisplayMode (unsigned int mode);

Specify a display mode. The argument is a bitfield constructed from constants chosen from Table 10 and OR'd together.

Default

```
glutInitDisplayMode (GLUT_RGBA | GLUT_SINGLE | GLUT_DEPTH);
```

Mode bit	Description
GLUT_RGB	Use RGB colour values (default)
GLUT_RGBA	Use RGB colour values (same as GLUT_RGB)
GLUT_INDEX	Use colour indexes
GLUT_SINGLE	Use a single colour buffer (default)
GLUT_DOUBLE	Use two colour buffers
GLUT_DEPTH	Use a depth buffer
GLUT_STENCIL	Use a stencil buffer
GLUT_ALPHA	Use a buffer with an alpha component
GLUT_STEREO	Use a stereo pair of windows
GLUT_ACCUM	Use an accumulation buffer

Table 10: Display mode bits

void glutInitWindowPosition (int x, int y);

Sets the initial position of the graphics window to be created by `glutCreateWindow()`. The arguments give the position of the top left corner of the window in screen coordinates ($Y = 0$ at the top of the screen). If the arguments are negative, the underlying window system chooses the window position.

Default

```
glutWindowPosition (-1, -1);
```

void glutInitWindowSize (int width, int height);

Sets the initial size of the graphics window that to be created by `glutCreateWindow()`. The arguments give the size of the window in pixels.

Default

```
glutInitWindowSize (300, 300);
```

void glutKeyboardFunc (void (*func)(key, x, y));

Register a callback function that handles keyboard events. The arguments passed to the function are the `unsigned char` code of the key that was pressed, and the current position of the mouse.

See also

`glutSpecialFunc()`.

Examples

`keys.c`.

void glLight{if}[v] (GLenum light, GLenum name, float [*]value);

Creates the specified light if it has not already been created and defines some of its properties. The first argument must be GL_LIGHTn, where $n = 0, 1, 2, \ldots, 7$. The second argument is a parameter name, and should be one of:

GL_AMBIENT	GL_SPOT_DIRECTION	GL_CONSTANT_ATTENUATION
GL_DIFFUSE	GL_SPOT_EXPONENT	GL_LINEAR_ATTENUATION
GL_SPECULAR	GL_SPOT_CUTOFF	GL_QUADRATIC_ATTENUATION
GL_POSITION		

The third argument gives the value of the parameter. The most common form of call is `glLightfv`, in which case the third argument is a pointer to an array of four `float`s or `double`s. See Table 6 on page 34 for an explanation of the parameter names and values.

Default

```
float black[] = { 0.0, 0.0, 0.0, 1.0 };
float white[] = { 1.0, 1.0, 1.0, 1.0 };
float pos[]   = { 0.0, 0.0, 1.0, 0.0 };
float dir[]   = { 0.0, 0.0, -1.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, black);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white);
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECION, dir);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 0.0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 180.0);
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.0);
```

void glLightModel{if}[v] (GLenum name, float [*] value);

Define properties of the lighting model. The first argument is a parameter name, and it should be one of:

```
GL_LIGHT_MODEL_AMBIENT
GL_LIGHT_MODEL_LOCAL_VIEW
GL_LIGHT_MODEL_TWO_SIDE
```

See Table 7 on page 35 for an explanation of the meanings of these parameters and their values.

Default

```
float amb[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE);
glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
```

void gluLookAt (ex, ey, ez, cx, cy, cz, ux, uy, uz);

Defines a viewing matrix and multiplies it to the right of the current `GL_MODELVIEW` matrix. All of the arguments are `double`.

The matrix establishes a point of view for looking at the scene. Your "eye" is at (e_x, e_y, e_z) and you are looking towards the point (c_x, c_y, c_z). The vector (u_x, u_y, u_z) is pointing upwards.

Examples

`planet.c`.

void glLoadIdentity (void);

Erases any current matrix and sets the current matrix to be the identity matrix.

void glLineWidth (GLfloat width);

Sets the width of lines. The default values of `width` is 1.0, and its value must be greater than 0.0.

void glutMainLoop (void);

Enters the GLUT processing loop. Within this loop, events are processed and the corresponding callback functions are invoked. This function never returns: the program is terminated by an event that GLUT does not handle, such as closing the graphics window.

glMaterial{if}[v] (GLenum face, GLenum name, float[*] value);

Defines a material property for use in lighting. The first argument specifies which face of each polygon is being defined. The allowed values are `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`. You are allowed to call `glMaterial()` twice, with different values for front and back polygons. Note that back faces will not be rendereed unless you use `glLightModel()` to specify `GL_LIGHT_MODEL_TWO_SIDE`.

The second argument is the name of a parameter and the third argument is the corresponding value. When `v` is included in the function name, the third argument should be a pointer to an array. See Table 5 on page 32 for an explanation of the parameter names and their meanings.

Default

```
float dim[] = { 0.2, 0.2, 0.2, 0.8 };
float bright[] = { 0.8, 0.8, 0.8, 1.0 };
float black[] = { 0.0, 0.0, 0.0, 1.0 };
float shine[] = { 0.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, dim);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, bright);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, black);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, shine);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, black);
```

void glMatrixMode (GLenum mode);

Sets the matrix mode to one of GL_MODELVIEW (default), GL_PROJECTION, or GL_TEXTURE.

void glutMotionFunc (void (*func)(int x, int y));

Register a callback function for mouse movements. The function is called when the mouse is dragged with one of its buttons pressed, and it is given the current mouse coordinates.

See also

glutPassiveMotionFunc().

Examples

drag.c.

void glutMouseFunc (void (*func)(button, state, x, y));

Register a callback function that handles mouse events. The function accepts four arguments, all of type int: a mouse button (one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON); the state of the button (either GLUT_UP or GLUT_DOWN); and the current position of the mouse.

Examples

move.c.

void glNormal3{bsidf}(nx, ny, nz);
void glNormal3{bsidf}v(p);

Defines a normal vector for the current vertex.

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

Defines an orthographic projection matrix with viewing volume left $\le x \le$ right, bottom $\le y \le$ top, and near $\le -z \le$ far. The arguments are of type double.

void glOrtho2D (left, right, bottom, top);

This function is used mainly for two-dimensional graphics. The call

```
glOrtho2D (left, right, bottom, top);
```

is equivalent to

```
glOrtho (left, right, bottom, top, -1.0, 1.0);
```

void glutPassiveMotionFunc (void (*func)(int x, int y));

Register a callback function for mouse movements. The function is called when the mouse is dragged without any buttons pressed, and it is given the current mouse coordinates.

void gluPerspective (alpha, aspect, near, far);

Defines a projection matrix and multiplies the current projection matrix by it. All arguments are `double`s. The camera is situated at the origin and is looking in the negative Z direction. `alpha` is the vertical angle of the field of view in degrees: it must be positive and less than $180°$. A typical value is $28°$, corresponding to a 50mm lens on a 35mm camera. `aspect` is the aspect ratio (width/height) of the image. Only objects whose distance from the viewpoint is between `near` and `far` ($\texttt{near} \le -z \le \texttt{far}$) will be drawn.

See also

`glFrustum()`, `glOrtho()`.

Examples

`planet.c`.

void glPointSize (GLfloat size);

Sets the size of a point. The default value of `size` is 1.0 and its value must be greater than 0.

void glPolygonMode (GLenum face, GLenum mode);

Controls the drawing mode for a polygon's front and back faces. The value of `face` may be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. The value of `mode` may be `GL_POINT` (draw vertexes), `GL_LINES` (draw edges), or `GL_FILL` (draw filled polygon).

Default

```
glPolygonMode (FRONT_AND_BACK, GL_FILL);
```

void PopMatrix (void);

Pops the top matrix of the current matrix stack. The value of the popped matrix is lost.

void glutPostRedisplay (void);

Requests that the graphics window be redrawn at the next opportunity. (`glutMainLoop()` will eventually invoke the callback function registered by `glutDisplayFunc()`.)

void glPushMatrix (void);

Pushes down the matrixes in the current matrix stack.

Examples

`figure.c`, `robot.c`.

void glRasterPos{234}{sifd}[v] (x, y, z, w);

Set the current raster position. The arguments are used in the same way as the coordinates of a vertex (that is, transformations are applied). This function is typically used before displaying a bitmap or bitmapped character.

See also

`glutBitmapCharacter()`.

void glRect{sifd}[v] (x0, y0, x1, y1);

Draw a rectangle with top left corner at `(x0, y0)` and bottom right corner `(x1, y1)`. The rectangle is in the plane $z = 0$ and is drawn according to the current polygon drawing style.

See also

`glPolygonMode()`.

void glutReshapeFunc (void (*func)(int width, int height));

Register a callback function that adjusts the display after the graphics window has been moved or resized. The reshape function must accept two arguments, the new width and new height of the window in pixels, and does not return a value.

If no reshape function is specified, GLUT calls `glViewport(0, 0, width, height)` to adjust the view after the user has resized the window.

Examples

`reshape.c`.

glRotate{fd} (angle, x, y, z);

Multiples the current matrix by a matrix that rotates counterclockwise through the given angle. The axis of rotation is a line from the origin $(0, 0, 0)$ to the point (x, y, z).

Examples

`spin.c`.

void glScale{fd} (x, y, z);

Multiples the current matrix by a matrix that stretches or shrinks an object along each axis. The object will be distorted if the arguments are not equal. If an argument is -1, the object is reflected in the corresponding axis.

void glutSetCursor (int style);

Change the cursor image in the current window. The value of `style` determines the shape of the cursor; Table 11 shows some of the possibilities.

Identifier	Description
GLUT_CURSOR_LEFT_ARROW	left arrow
GLUT_CURSOR_RIGHT_ARROW	right arrow
GLUT_CURSOR_TOP_SIDE	arrow pointing to top of window
GLUT_CURSOR_BOTTOM_SIDE	arrow pointing to bottom of window
GLUT_CURSOR_LEFT_SIDE	arrow pointing to left side of window
GLUT_CURSOR_RIGHT_SIDE	arrow pointing to right side of window
GLUT_CURSOR_CURSOR_INFO	pointing hand
GLUT_CURSOR_DESTROY	skull and crossbones
GLUT_CURSOR_HELP	question mark
GLUT_CURSOR_CYCLE	rotating arrows
GLUT_CURSOR_SPRAY	spray can
GLUT_CURSOR_WAIT	wrist watch
GLUT_CURSOR_TEXT	text insertion marker
GLUT_CURSOR_CROSSHAIR	small cross hair
GLUT_CURSOR_FULL_CROSS_HAIR	full window cross hair
GLUT_CURSOR_NONE	invisible cursor

Table 11: Cursor codes

`void glShadeModel (GLenum mode);`

Sets the shading model to either `GL_SMOOTH` (default) or `GL_FLAT`.

`void glutSpecialFunc (void (*func)(int key, int x, int y));`

Registers a callback function for special key codes, such as arrows, `F` keys, and so on. The callback parameters `x` and `y` give the position of the mouse when the key was pressed. Note that the keys ESC, BACKSPACE, and DELETE have regular ASCII codes and are recognized by `glutKeyboardFunc()`.

The keys are identified by constants with the prefix `GLUT_KEY_`. The remainder of the name is one of:

F1	F2	F3	F4	F5	F6
F7	F8	F9	F10	F11	F12
LEFT	UP	RIGHT	DOWN	PAGE_UP	
PAGE_DOWN	HOME	END	INSERT		

See also

`glutKeyboardFunc()`.

void glutStrokeCharacter (void *font, int character);

Draws a stroked character at the origin of the current model view and advance the model view matrix by the width of the character. Stroked characters may be (and usually are) scaled. There are two fonts available:

`GLUT_STROKE_ROMAN`. A proportionally spaced Roman font for ASCII characters 32 through 127. All characters sit on a base line and reach up to 119.05 units above and 33.33 units below the line.

`GLUT_MONO_ROMAN`. A fixed-space Roman font for ASCII characters 32 through 127. All characters sit on a base line and reach up to 119.05 units above and 33.33 units below the line. Each character is 104.76 units wide.

See also

`glutBitmapCharacter()`, `glutStrokeWidth()`.

int glutStrokeWidth (void *font, int character);

Return the width of the given character. The font must be a stroked font.

void glutSwapBuffers (void);

Exchange the "front" colour buffer and the "back" colour buffer. OpenGL displays the front buffer while the program is drawing into the back buffer. This works only if the mode `GLUT_DOUBLE` was selected in the call to `glutInitDisplayMode()`.

Examples

`spin.c`.

void glTranslate{fd} (x, y, z);

Multiples the current matrix by a matrix that translates (moves) an object from (X, Y, Z) to $(X + x, Y + y, Z + z)$.

void glVertex{234}{sifd}[v] (TYPE coords);

Specifies a vertex. Up to four coordinates (x, y, z, and w) may be provided. z defaults to 0 and w defaults to 1.

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

Defines a pixel rectabgle in the viewing window into which the final image is to be mapped. The arguments are integers. The lower-left corner of the viewport is at (x, y) and the viewport has the indicated width and height. The default viewport is the entire viewing window.

Special Objects

The GLUT library provides several objects that are rather tedious to build. These objects are fully defined, with normals, and they are useful for experimenting with lighting. Their prototypes are listed below.

For a sphere, "slices" are lines of longitude and "stacks" are lines of latitude. Cones are analogous. For a torus, "sides" run around the torus and "rings" run around the "tube".

```
void glutWireSphere (double radius, int slices, int stacks);
void glutSolidSphere (double radius, int slices, int stacks);
void glutWireCube (double size);
void glutSolidCube (double size);
void glutWireTorus (double inner, double outer, int nsides, int rings);
void glutSolidTorus (double inner, double outer, int nsides, int rings);
void glutWireIcosahedron (void);
void glutSolidIcosahedron (void);
void glutWireOctahedron (void);
void glutSolidOctahedron (void);
void glutWireTetrahedron (void);
void glutSolidTetrahedron (void);
void glutWireDodecahedron (void);
void glutSolidDodecahedron (void);
void glutWireCone (double radius, double height, int slices, int stacks);
void glutSolidCone (double radius, double height, int slices, int stacks);
void glutWireTeapot (double size);
void glutSolidTeapot (double size);
```