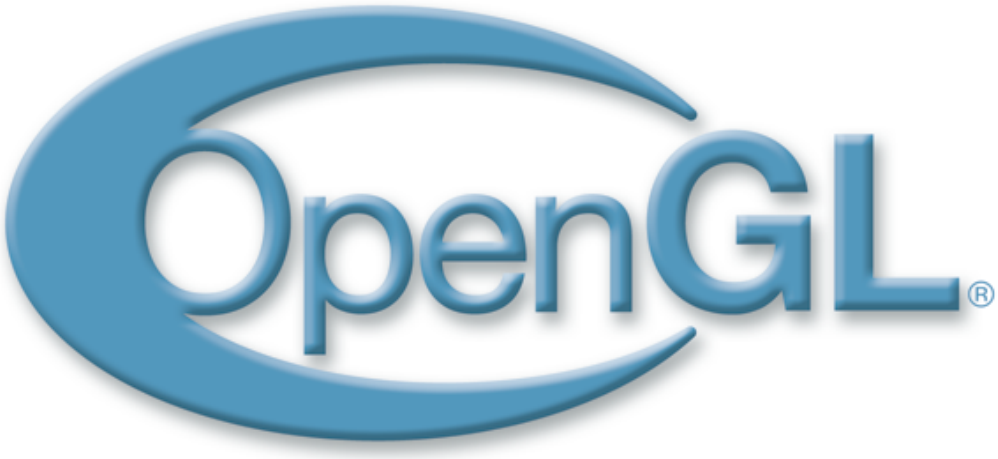
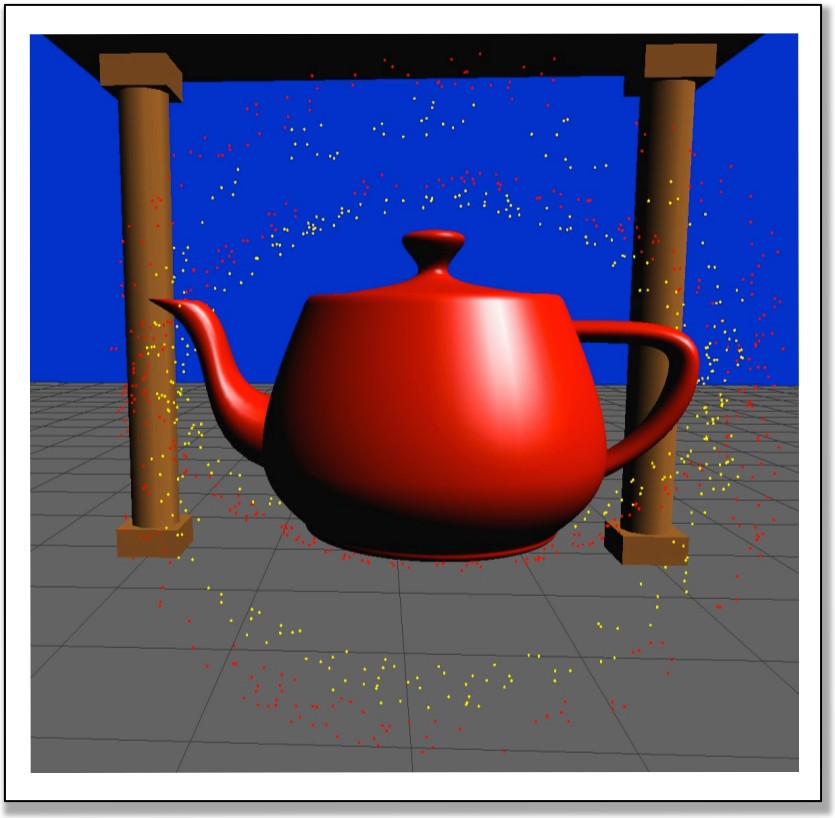
MANCHESTER 1824
The University of Manchester

School of Computer Science



Tutorial and Reference Manual

Toby Howard



Contents

Chapter 1

About this manual

Welcome to the OpenGL Tutorial and Reference Manual v4.0.

This manual is in two parts: the first (Chapters 2 to 6) is a hands-on **Tutorial**, which uses a series of example programs to illustrate some of the main features of OpenGL. The second part (Chapter 7 onwards) is a **Reference Manual**, which gives detailed descriptions of a subset of OpenGL functions.

We recommend that you first read the tutorial part of this manual first (Chapters 2–6, in order) and experiment with the example programs on-line.

Important note: this manual describes only a small part of OpenGL's entire functionality, sufficient to support our undergraduate graphics teaching. It is intended for newcomers to OpenGL. The definitive source of complete OpenGL documentation is at `www.opengl.org`. Note also that this manual describes OpenGL v1.1. Section 2.1.1 (page 6) explains why.

Comments/reports of bugs are welcome: please tell `toby.howard@manchester.ac.uk`.

1.1 Acknowledgements

It's a pleasure to thank Alan Murta and Julien Cartigny for helping with parts of this manual, and Steve Pettifer for his helpful suggestions. Brian Paul, the author of Mesa (`www.mesa3d.org`), and Mark J. Kilgard, who wrote the original GLUT, patiently answered my questions. The section on picking (pages 79–88) is reproduced from `http://www.lighthouse3d.com/opengl/picking` with the kind permission of its author, António Ramires of the University of Minho, Portugal.

1.2 Licence

Part I

OpenGL Tutorial

Chapter 2

Introduction to OpenGL

OpenGL is a worldwide standard for 3D computer graphics programming. It's very widely used: in industry, in research laboratories, in computer games – and for teaching computer graphics.

OpenGL is a powerful, professional-level system, and it would take a manual much thicker than this one to describe all its facilities completely. We have selected a **subset** of OpenGL – a portion of OpenGL's functionality which is relevant to COMP27112, and sufficient to support its programming labs. People doing 3rd Year Projects using graphics should find this manual useful too.

2.1 History and versions

OpenGL has it origins in the IRIS Graphics Language invented by Silicon Graphics Inc. in the early 1990s as an API for programming their high-performance specialised graphics workstations. In 1992 Silicon Graphics created a new system based on IRIS GL called OpenGL. Unlike IRIS GL, OpenGL was specifically designed to be **platform-independent**, so it would work across a whole range of computer hardware – not just Silicon Graphics machines. The combination of OpenGL's power and portability led to its rapid acceptance as a **standard** for computer graphics programming. Today, the development of OpenGL is managed by the OpenGL Working Group (`www.opengl.org`), which is part of the Khronos Group consortium for open standard APIs.

OpenGL itself isn't a programming language, or a software library. It's the **specification** of an Application Programming Interface (API) for computer graphics programming. In other words, OpenGL defines a set of functions for doing computer graphics. To create your graphics you need an **implementation** of OpenGL. We use a free implementation called **Mesa**, which we'll describe in Section 2.4.

Since its birth in 1992, OpenGL has been continuously developed and extended, to parallel new developments on graphics hardware. Table 2.1 shows a summary – you can find the full details at `http://en.wikipedia.org/wiki/OpenGL#Version_History`.

Version and year	Major functionality changes
OpenGL 1.0, 1992	The first version, with fixed functionality rendering pipeline.
OpenGL 1.1, 1997	Extensions including improved pixel blending, texture effects.
OpenGL 2.0, 2004	Introduction of the programmable rendering pipeline, using GLSL, the OpenGL shading language.
OpenGL 3.0, 2008	Fixed function pipeline deprecated.
OpenGL 4.0, 2010	Changes to functionality to allow OpenGL programs access to hardware features designed for Direct3D.

Table 2.1: OpenGL major versions.

2.1.1 The fixed/programmable rendering pipeline

The biggest change that's happened during OpenGL's development has been the transition from a fixed-functionality rendering pipeline to a fully programmable pipeline.

This manual describes the original fixed-pipeline version of OpenGL, which we currently use as the basis for our introductory computer graphics teaching. You may be wondering why we don't use the programmable-pipeline version of OpenGL for our teaching. Well, we certainly could do that, and we may do so in the future. However, we currently choose not to, for two main reasons: first, because of the complexity of the programming details that the programmable-pipeline brings; second, because of versioning issues – not everyone will have a graphics card that supports the version of OpenGL that we might describe.

For an introduction to computer graphics, the fixed-pipeline version of OpenGL works well. It's simple, it's easy to explain, and it maps straightforwardly onto code. And perhaps most importantly, we are trying to teach the **principles** of interactive Computer Graphics; we are not trying to teach the intricacies of OpenGL programming or shading languages.

2.2 Flavours of OpenGL

As well as standard OpenGL, several other versions of OpenGL have been developed, targeted at specific application areas.

OpenGL for Embedded Systems (OpenGL ES) is a subset of OpenGL for use in embedded systems, such as mobile phones and devices. OpenGL ES 1.0 was released in 2003. Examples of implementations include Android, iOS, Blackberry and Raspberry Pi. OpenGL ES 1.x uses fixed functionality rendering; vesions 2.x and above use programmable rendering. OpenGL ES development is managed by the Khronos group (`http://www.khronos.org/opengles`).

WebGL is a version of OpenGL ES 2.0 for the Web, using a Javascript API and the HTML5 canvas element. Development of WebGL started in 2009, and WebGL 1.0 was released in March 2011. Rendering is implemented entirely in the brower. Development is managed by the Khronos group.

2.3 A whirlwind tour of OpenGL

What exactly can OpenGL do? Here are some of its main features:

- It provides 3D geometric objects, such as lines, polygons, triangle meshes, spheres, cubes, quadric surfaces, NURBS curves and surfaces;
- It provides 3D modelling transformations, and viewing functions to create views of 3D scenes using the idea of a **virtual camera**;
- It supports high-quality rendering of scenes, including hidden-surface removal, multiple light sources, material types, transparency, textures, blending, fog;
- It provides display lists for creating graphics caches and hierarchical models. It also supports the interactive 'picking' of objects;
- It supports the manipulation of images as pixels, enabling frame-buffer effects such as anti-aliasing, motion blur, depth of field and soft shadows.

Figure 2.1 shows the relationship between an application and OpenGL in our local GNU/Linux environment. An application programmer sees OpenGL as a single library providing a set of functions for graphical input and output. In fact, it's slightly more complicated than that.

2.3.1 The support libraries: GLU and GLUT

A key feature of the design of OpenGL is the separation of **interaction** (input and windowing functions) from **rendering**. OpenGL itself is concerned only with graphics rendering. You can always identify an OpenGL function: all OpenGL function names start with **'gl'**.

Over time, two **utility libraries** have been developed which greatly extend the low-level (but very efficient) functionality of OpenGL. The first is the 'OpenGL Utility Library', or **GLU**. The second is the 'OpenGL Utility Toolkit', or **GLUT**:

- **GLU** provides functions for drawing more complex primitives than those of OpenGL, such as curves and surfaces, and also functions to help specify 3D views of scenes. All GLU function names start with **'glu'**.
- **GLUT** provides the facilities for interaction that OpenGL lacks. It provides functions for managing windows on the display screen, and handling input events from the

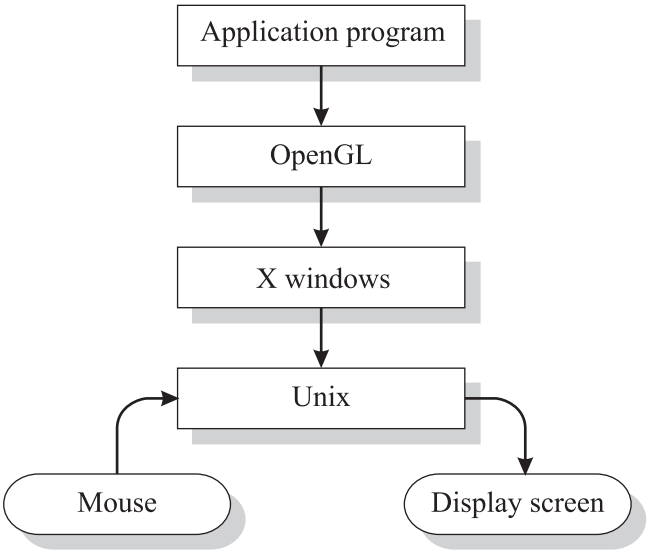


Figure 2.1: Where OpenGL fits in – a high-level view.

mouse and keyboard. It provides some rudimentary tools for creating Graphical User Interfaces (GUIs). It also includes functions for conveniently drawing 3D objects like the platonic solids, and a teapot. All GLUT function names start with **'glut'**.

Figure 2.2 shows the relationships between OpenGL, GLU, and GLUT. As you can see, it's helpful to think of 'layers' of software, where each layer calls upon the facilities of software in a lower layer.

However, somewhat confusingly, when most people say '**OpenGL**', what they really mean is '**OpenGL** plus **GLU** plus **GLUT**'. It's a slightly lazy terminology, but we'll use it too.

2.4 What is Mesa?

Mesa is a C implementation of a graphics system that implements the OpenGL specification. Whereas OpenGL is intended to run on machines which have graphics support in hardware, Mesa doesn't require the presence of any special 3D graphics acceleration hardware – although it can certainly take advantage of it if it's there. Of course, the performance of the graphics will be better with hardware acceleration, but it's still remarkably good without it on a reasonably fast PC. Find out all about Mesa at `http://www.mesa3d.org`. On Linux, OpenGL application programs link to the Mesa library.

2.5 Using OpenGL away from the School

This manual describes how to compile and run OpenGL programs on our local Linux installation. You may well want to run your Linux programs remotely, or work on Mac or Windows too, for developing at home.

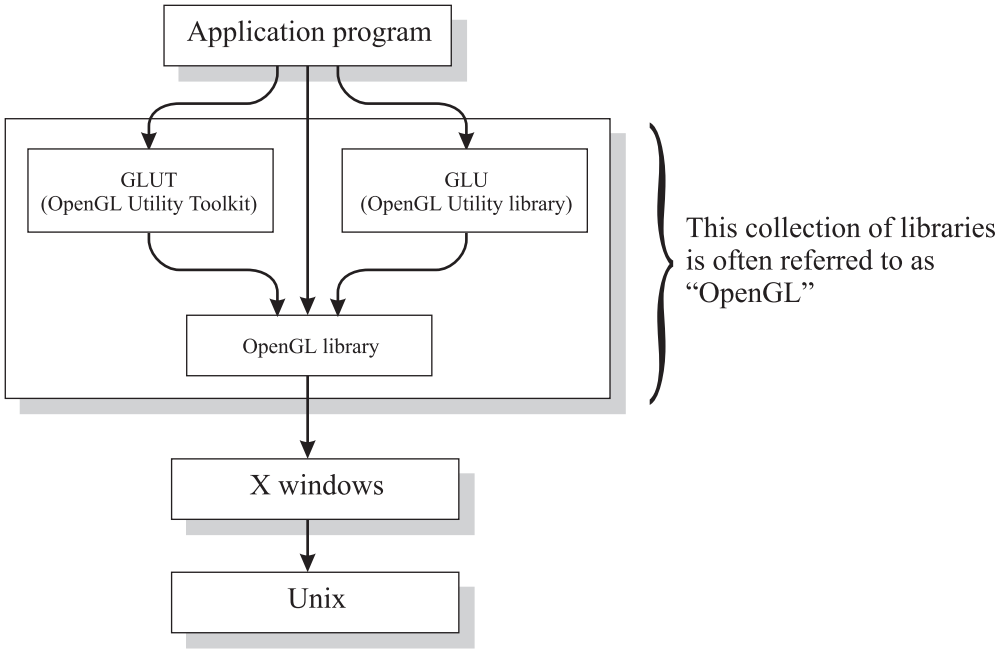


Figure 2.2: What is commonly called 'OpenGL' is actually a set of three libraries: **OpenGL** itself, and the supporting libraries **GLU** and **GLUT**.

2.5.1 Running OpenGL on a School PC, remotely

You can compile and run your OpenGL programs on a School Linux PC and view the results on a remote PC, over X. This will probably be horribly slow, but it should work, and may be handy sometimes. Here's how to do it.

1. Login from your home computer to a School PC (running Linux), making an X connection.
2. On the School PC, compile your OpenGL program.
3. On the School PC, issue the following command in your shell:

```
export LIBGL_ALWAYS_INDIRECT=1
```

4. On the School PC, run your program. You should see an X window pop up on your home computer.

2.5.2 Using OpenGL on the Mac

The good news is that Mac OS X comes with OpenGL built in, so you're ready to go. First, make sure you have Xcode and its Developers Tools installed, which will give you the gcc compiler. Then, note that the Mac `#include` files are in a different place from Linux, so if you want to have code portable between Linux/Mac/Windows, you'll need this conditional definition in your program:

```
#ifdef MACOSX
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

To compile, define the MACOSX macro, and specify the OpenGL and GLUT frameworks:

```
$ gcc -DMACOSX -framework OpenGL -framework GLUT -o myprog myprog.c
```

An alternative approach is to change the `#include` on the fly, and then compile, using a simple shell script like this:

```
#!/bin/bash
cat $1 | sed 's/<GL\/glut.h>/<GLUT\/glut.h>/' | \
gcc -x c - -o ${1%.*}  -framework OpenGL -framework GLUT
```

2.5.3 Using OpenGL on Windows

For Windows, you can develop using Cygwin (see, for example, `http://goo.gl/G8OV0`), but most people prefer to use Visual Studio – see `http://www.cs.manchester.ac.uk/ugt/COMP27112/OpenGL` for links which explain where to install the various include files and libraries you'll need.

2.6 Resources and further reading

Here are some useful resources, and suggestions for further reading, should you wish to find out more.

2.6.1 On-line resources

- You can find detailed technical OpenGL specification documents at `www.opengl.org/documentation`.
- The Moodle Graphics Programmers' forum at

  `http://moodle.cs.man.ac.uk/mod/forum/view.php?id=579`

  is the place to go for graphics queries and chat. Post your OpenGL programming queries here, and help others with theirs.
- Our local OpenGL Web pages: `http://www.cs.manchester.ac.uk/ugt/COMP27112/OpenGL`. Check here for up-to-date details of the local installation.
- Local example programs: we have a number on-line, in `/opt/info/courses/OpenGL/examples`.
- The official home of OpenGL on the Web: `www.opengl.org`. Lots of pointers to on-line information, tutorials, example programs, and downloadable software.

2.6.2 Books

There are lots of books about OpenGL, and many about computer graphics in general which use OpenGL examples. The following are two we recommend:

- **Interactive Computer Graphics: A Top-Down Approach with OpenGL** by Edward Angel. Addison-Wesley, ISBN 0-201-85571-2. General introduction to computer graphics for people new to the subject.
- **OpenGL Programming Guide, Fifth Edition: The Official Guide to Learning OpenGL, Version 1.2** by Mason Woo et al. Addison-Wesley, 0321335732. Also known as 'The Red Book', provides complete coverage of OpenGL from simple to advanced, with many code examples. Assumes familiarity with C, some maths, geometry. The coverage of this book far exceeds the material taught in COMP27112. Available free online – see `http://www.glprogramming.com/red`.

2.7 About the notation used in this manual

Experienced C programmers may wish to skip this section.

In this manual, when we introduce a new OpenGL function, we'll give its definition, followed immediately by a description of what it does. For example, here's the definition of the GLUT function which draws a sphere, which you'll meet on page 49:

void **glutWireSphere** (GLdouble *radius*,
GLint *slices*,
GLint *stacks*);

What this notation means is the following:

- The name of the function is **glutWireSphere()**;
- The result type of the function is `void`;
- The function has three arguments:
  - `radius`, of type `GLdouble`
  - `slices`, of type `GLint`
  - `stacks`, of type `GLint`

Here's an example of using this function in a program:

```
GLdouble rad= 1.0;
GLint sl= 15;
GLint st= 20;

glutWireSphere (rad, sl, st);
```

Of course, you could set the arguments directly, without declaring variables:

```
glutWireSphere (1.0, 15, 20);
```

Note that OpenGL defines its own names for data types, all of which begin with `GL`. Examples are: `GLdouble`, `GLint`, `GLfloat`. The reason it's done like this is to make the specification of OpenGL language-independent. In most cases, it'll be obvious what the data type means – `GLint`, for example, is GL's name for an integer, or an `int` in C. Where it isn't obvious, we'll tell you.

To continue with the example of **glutWireSphere()**, this is how we'd write its description:

> **glutWireSphere()** draws a sphere, of radius `radius`, centred on $(0, 0, 0)$ in object coordinates. `slices` is the number of subdivisions around the Z axis (like lines of longitude); `stacks` is the number of subdivisions along the Z axis (like lines of latitude). Solid version: **glutSolidSphere()**.

2.8 What next?

Now onto Chapter 3, which explains how to compile OpenGL programs using our local installation.

Chapter 3

Getting started with OpenGL

This chapter explains how to compile and link C programs with OpenGL using our local installation. There are two different ways to do this:

- Using the command `cogl` – this is handy for compiling single standalone OpenGL programs, and is the recommended way for compiling programs in the COMP27112 lab. `cogl` is a script in `/opt/common/bin`. It's also reproduced in Appendix A.
- Using a makefile – this is a more flexible approach, necessary for larger projects which use more than one source file. Use of a makefile is not recommended for the COMP27112 lab. See Appendix B for a sample makefile.

3.1 Compiling using `cogl`

`cogl` is a command we've written locally to make compiling single programs with OpenGL as simple as possible. We'll use the example program `thegears.c` to illustrate the use of `cogl`.

First, make sure you are running X. Then, select an appropriate directory to work in, and take your own private copy of the program `thegears.c`, as follows:

```
$ cp /opt/info/courses/OpenGL/examples/thegears.c .
```

(Don't forget that **dot (.)** as the second argument to `cp`.)

You compile and link the program as follows:

```
$ cogl thegears.c
```

This will produce an executable program called `thegears`, which you run as follows:

```
$ ./thegears
```

You should see a square OpenGL window appear on your display, looking like Figure 3.1. Move your mouse into the OpenGL window, and press `h` on the keyboard to bring up the help screen, and have a play.

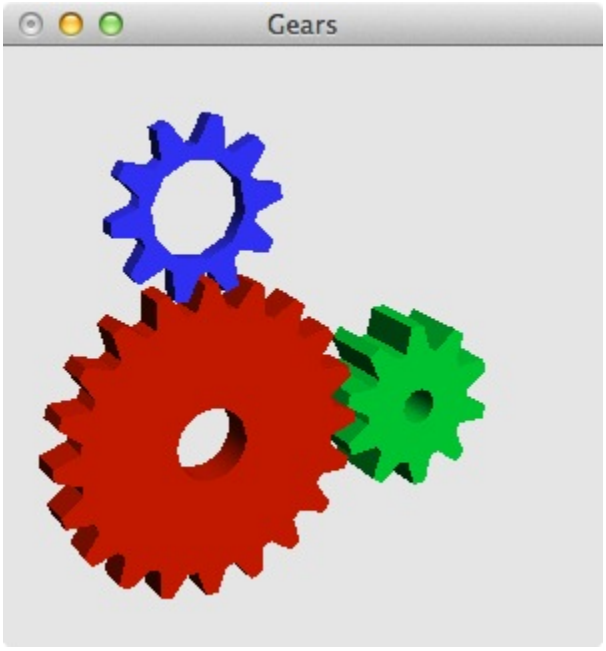


Figure 3.1: thegears.c

3.2 Try some other examples

Try compiling and the following from `/opt/info/courses/OpenGL/examples/`.

- `tori.c`: some doughnuts. Move the mouse slowly.
- `teapots.c`: draws our teapot collection.
- `morph3d.c`: morphiness.
- `reflect.c`: reflective texture mapping. Try the arrow keys.
- `pointblast.c`: a simple particle system. Try the mouse buttons.
- `star`.c: moving starfield. Hit `t` to warp.
- `lorenz.c`: organised chaos.

The following are part of the `xscreensaver` collection, and are already compiled, so just `cd` to the above directory, and run the programs. You'll have to type `control-c` in your shell to stop them running:

- `moebius`: ants crawl inexplicably around a moebius strip.
- `sproingies`: multi-coloured bouncy things tumble down an infinite staircase, and occasionally explode.

- `superquadrics`: 3D shapes morph into each other, based on the 'superquadric' objects developed by researcher **Alan Barr**[w].
- `cage`: be amazed as OpenGL draws an impossible object.

3.3 What next?

Now onto Chapter 4, which introduces the structure of an OpenGL program.

Chapter 4

Beginning OpenGL programming

In this and the next two chapters, we introduce the basic ideas of OpenGL using a series of example programs.

4.1 The OpenGL model

Figure 4.1 shows the relationships between an application program, the graphics system, input and output devices, and the user.

The **application program** has its own internal **model** of what it's doing – its own interpretation of what the graphics it's manipulating actually **means**. It draws the graphics using the facilities of the **graphics system** – in our case, OpenGL. The user views the graphics, and uses **input devices**, such as a mouse, to **interact**. Information about the user's interactions are sent back to the application, which decides what action to take. Typically, it will make changes to its internal model, which will cause the graphics to be updated, and so another **loop** in the interaction cycle begins.

4.2 Platform- and device-independence

As we saw in Chapter 2, OpenGL is designed to be platform-independent and device-independent, so it isn't concerned with the exact makes and models of graphics display and interaction hardware it uses. Instead, OpenGL functions refer to **windows** and **events**:

- An OpenGL **window** is a rectangular area on a physical display screen into which OpenGL draws graphics. Usually, an OpenGL window corresponds exactly to a window managed by the window manager', such as X. (It's also possible to have multiple OpenGL windows simultaneously active on a single display – see Section 9.4.)
- An OpenGL **event** occurs when the user operates an input device. In order to respond to the input event, the application must provide a C function – known as a **callback function** – to handle the event; OpenGL automatically calls the application's function, passing it the event data.

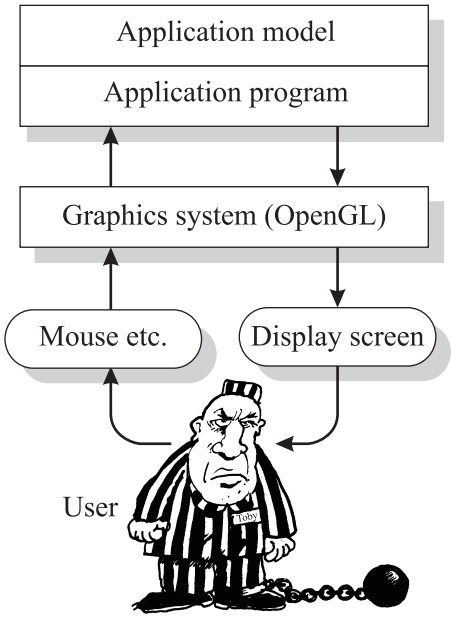


Figure 4.1: The graphical interaction loop.

In fact, OpenGL doesn't draw its graphics directly to the window. It actually draws into a data structure (an array of pixels) inside OpenGL called the **frame-buffer**, often just called the **buffer**. Periodically, OpenGL copies the pixels in the frame buffer into the window. More on this in Section 6.2.

4.3 Example 1: a bare-bones program

We'll begin with the simplest possible OpenGL program. It's `ex1.c` in the `examples` directory. Take a copy of this program, and compile it with `cogl`:

```
$ cp /opt/info/courses/OpenGL/examples/ex1.c .
$ cogl ex1.c
```

When you run `ex1`, you should see an OpenGL window appear. To stop the program running, place your mouse inside the shell window from which you ran the program, and hit `control-c`.

Here's the code for `ex1.c`:

```
1 #include <GL/glut.h>
2
3 void display (void) {
4 /* Called when OpenGL needs to update the display */
5   glClearColor(0.9,0.9,0.9,0.0); /* Set grey background */
6   glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
7   glFlush();                     /* Force update of screen */
8 }
9
```

```
10 int main (int argc, char **argv) {
11   glutInit (&argc, argv);     /* Initialise OpenGL */
12   glutCreateWindow ("ex1");   /* Create the window */
13   glutDisplayFunc (display); /* Register the "display" function */
14   glutMainLoop ();            /* Enter the OpenGL main loop */
15   return 0;
16 }
```

The program begins with `#include <GL/glut.h>`, and all OpenGL programs must start with this line, which accesses all the OpenGL include files: it pulls in all the function prototypes and other definitions used by OpenGL. Miss it out, and `cogl` will flatly refuse to compile your program. (Mac users: your include line will be slightly different – see Section 2.5.2 on page 9).

`ex1.c` contains two functions: `display()`, and `main()`. The execution of all C programs starts at `main()`, so we'll start there too.

We first call the **glutInit()** function:

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

glutInit() initializes the GLUT library, and it must be called before any other GLUT function. `argc` and `argv` should be the arguments of the application's `main()` – **glutInit()** understands several command-line options – see the GLUT manual for details.

Next, we call **glutCreateWindow()**:

int **glutCreateWindow** (char **name*);

glutCreateWindow() creates an OpenGL window for rendering and interaction, with `name` displayed in its titlebar. GLUT assigns this window an integer identifier, returned as the result of the function. The window identifier is used when writing OpenGL programs which use multiple windows (described in Section 9.4). By default, the window has a size of $(300, 300)$ pixels, and its position is up to the window manager to choose. If the functions **glutInitWindowSize()** or **glutInitWindowPosition()** (page 23) have already been called, their arguments will control the size and position of the window.

Next comes a call to **glutDisplayFunc()**, and this is a bit more interesting. It's an example of one of the cornerstones of OpenGL programming, which we'll need to look at in detail – the use of **callback functions**.

4.4 Callback functions

A callback function, more often just called a **callback**, is a C function, written by the application programmer. In program `ex1.c`, `display()` is the only callback function we define. But there's one important difference between a callback function and an ordinary C function: **the application never calls the callback function directly**. Instead, the callback function **is called by OpenGL**, whenever OpenGL decides it needs to be called.

In `ex1.c`, we use the most basic callback of all – a function that draws the graphics that we want OpenGL to display. We use **glutDisplayFunc()** to tell OpenGL which application function it should call whenever it needs to refresh the window to draw graphics:

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

glutDisplayFunc() registers the name of the callback function to be invoked when OpenGL needs to redisplay (or display for the first time) the contents of the window. The application **must** register a display function – it isn't optional.

The argument of **glutDisplayFunc()** is rather cryptic, and worth a closer look:

```
void (*func)(void)
```

This says that `func()` must be a function which returns `void`, and has no arguments. In other words, a function like `display()`:

```
void display (void) {
/* Called when OpenGL needs to update the display */
  glClearColor(0.9,0.9,0.9,0.0); /* Set grey backbround */
  glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
  glFlush();                     /* Force update of screen */
```

So to summarise, in our example the line:

```
 glutDisplayFunc (display); /* Register the "display" function */
```

tells OpenGL to call the application's function `display()` function whenever it needs to redraw the graphics.

It's up to the application to define what the `display()` function does – who else could know? In `ex1.c`, the `display()` function doesn't do much: it simply calls **glClear()**:

void **glClear** (GLbitfield *mask*);

glClear() clears one or more of OpenGL's buffers, specified by `mask`. In this manual, we'll only be concerned with one buffer, the **frame buffer**, which holds the pixels which will be copied to the window. This has the special name `GL_COLOR_BUFFER_BIT`. When **glClear()** is called, each pixel in the buffer is set to the **current clear colour**, which is black by default. Here, we first set the colour to a light grey using the function **glClearColor()** (see page 90).

Now we have a call to **glFlush()**:

void **glFlush** (void);

The purpose of this function is to instruct OpenGL to make sure the screen is up to date – it causes the contents of any internal OpenGL buffers are 'flushed' to the screen. Note that you only ever need to call **glFlush()** when you're not using **double-buffering** (which we'll meet in Chapter 6). In practice, most OpenGL programs will use double-buffering – to stop screen flicker – but for now in these simple examples we're not using it just yet.

What would happen if we didn't call **glFlush()** at the end of `display()`? Then, we couldn't guarantee that the screen will show the up-to-date picture. And that's clearly not desirable

for a real-time interactive graphics program!

4.5 The main event loop

glutMainLoop() starts the GLUT 'event processing' loop:

void **glutMainLoop** (void);

Once started, this loop will carry on for as long as the program is running. Each time around the loop, GLUT checks to see if anything has changed since last time, and calls the appropriate callback functions.

In pseudocode, the action of **glutMainLoop()** is this:

```
while (1) { /* loop forever */
  if (the application has changed the graphics) {
     call the DISPLAY callback function;
     }

  if (the window has been moved or resized) {
     call the RESHAPE callback function;
     }

  if (any keyboard and/or mouse events have happened) {
     call the KEYBOARD and/or MOUSE callback function;
     }

  call the IDLE callback function;

} /* while */
```

We'll ignore the `reshape()` function for now, returning to it in Section 5.3. And we'll look at the `idle()` function in Section 6.1.

4.6 Example 2: a keyboard event callback

As we saw above, quitting `ex1.c` must be done from the command-line, which isn't very nice from a user-interface point of view. Here's how we can do it better, using a callback, in program `ex2.c`:

```
1 #include <GL/glut.h>
2 #include <stdio.h>
3
4 void display (void) {
5 /* Called when OpenGL needs to update the display */
6   glClearColor(0.9,0.9,0.9,0.0); /* Set grey background */
7   glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
8   glFlush();                     /* Force update of screen */
9 }
10
11 void keyboard (unsigned char key, int x, int y) {
12 /* Called when a key is pressed */
```

```
13   if (key == 27) exit (0);    /* 27 is the Escape key */
14   else printf ("You␣pressed␣%c\n", key);
15 }
16
17 int main(int argc, char **argv) {
18   glutInit (&argc, argv);      /* Initialise OpenGL */
19   glutCreateWindow ("ex2");    /* Create the window */
20   glutDisplayFunc (display);   /* Register the "display" function */
21   glutKeyboardFunc (keyboard); /* Register the "keyboard" function */
22   glutMainLoop ();             /* Enter the OpenGL main loop */
23   return 0;
24 }
```

Try `ex2.c` out.

The addition we've made is to tell OpenGL what to do when it detects a keyboard event. We tell it to call the function `keyboard()` using **glutKeyboardFunc()**:

void **glutKeyboardFunc** (*void (*func)(unsigned char key, int x, int y)*);

glutKeyboardFunc() registers the application function to call when OpenGL detects a key press generating an ASCII character. This can only occur when the mouse focus is inside the OpenGL window.

Again, the specification of the argument type is a bit cryptic. It says that it expects a function `func()` which returns `void`, and has the three arguments `key`, `x` and `y`. So, it's a function like this:

```
void keyboard (unsigned char key, int x, int y) {
  /* Called when a key is pressed */
}
```

Three values are passed to the callback function: `key` is the ASCII code of the key pressed; `x` and `y` give the pixel position of the mouse at the time.

Back to `ex2.c` – inside the `keyboard()` callback, we look at the value of `key`. If it's `27` (the ASCII code for the `escape` key) we call the standard C function `exit()` to terminate the program cleanly; otherwise, we print (in the shell window) a message saying which key was pressed. Note that `ex2.c` needs also `#include <stdio.h>`, because we're using the `printf()` function.

Note: **glutKeyboardFunc()** only responds to pressed keys which have single ASCII codes. For other keys, such as the arrow or function keys, use the **glutSpecialFunc()** function (page 73).

4.7 Example 3: customizing the window

In `ex3.c` we add a few new functions to give us better control over the drawing window:

```
1 #include <GL/glut.h>
2
3 void display (void) {
4 /* Called when OpenGL needs to update the display */
```

```
5     glClearColor(0.9,0.9,0.9,0.0); /* Set grey backbround */
6     glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
7     glFlush();                     /* Force update of screen */
8  }
9
10 void keyboard (unsigned char key, int x, int y) {
11 /* Called when a key is pressed */
12   if (key == 27) exit (0);    /* 27 is the Escape key */
13 }
14
15 int main(int argc, char **argv) {
16   glutInit (&argc, argv);      /* Initialise OpenGL */
17   glutInitWindowSize (500, 500);     /* Set the window size */
18   glutInitWindowPosition (100, 100); /* Set the window position */
19   glutCreateWindow ("ex3");    /* Create the window */
20   glutDisplayFunc (display);   /* Register the "display" function */
21   glutKeyboardFunc (keyboard); /* Register the "keyboard" function */
22   glutMainLoop ();             /* Enter the OpenGL main loop */
23   return 0;
24 }
```

Try `ex3.c` out.

First, we specify a size and position for the window using **glutInitWindowSize()**:

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

glutInitWindowSize() sets the value of GLUT's **initial window size** to the size specified by `width` and `height`, measured in pixels.

Similarly, **glutInitWindowPosition()** sets the value of GLUT's **initial window position**:

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

`x` and `y` give the position of the top left corner of the window measured in pixels from the **top left corner** of the X display.

4.8 What next?

Now onto Chapter 5, which looks at 2D and 3D graphics.

Chapter 5

2D and 3D graphics

In this chapter we start doing some graphics. We'll begin by extending `ex3.c` to do some 2D drawing – just a triangle, but it'll serve to illustrate how drawing works in OpenGL.

5.1 Example 4: drawing a 2D triangle

`ex4.c` draws a triangle, using the coordinates shown in Figure 5.1.

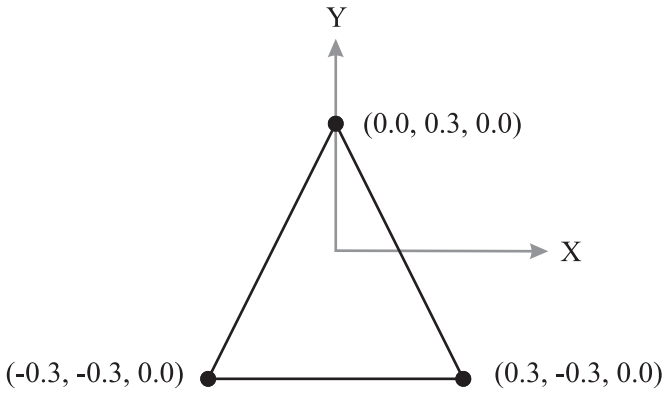


Figure 5.1: The triangle from example `ex4.c`. It's defined on the $Z = 0$ plane. The Z axis comes out of the page towards you.

Here's the code for ex4.c:

```
 1 #include <GL/glut.h>
 2
 3 void display (void) {
 4 /* Called when OpenGL needs to update the display */
 5    glClearColor(0.9,0.9,0.9,0.0); /* Set grey backbround */
 6    glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
 7    glLoadIdentity ();
 8    gluLookAt (0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
 9    glColor3f(1.0,0.0,0.0);
10    glBegin (GL_LINE_LOOP); /* Draw a triangle */
```

```
11       glVertex3f(-0.3, -0.3, 0.0);
12       glVertex3f(0.0, 0.3, 0.0);
13       glVertex3f(0.3, -0.3, 0.0);
14    glEnd();
15    glFlush(); /* Force update of screen */
16 }
17
18 void keyboard (unsigned char key, int x, int y) {
19 /* Called when a key is pressed */
20   if (key == 27) exit (0); /* 27 is the Escape key */
21 }
22
23 void reshape (int width, int height)
24 { /* Called when the window is created, moved or resized */
25    glViewport (0, 0, (GLsizei) width, (GLsizei) height);
26    glMatrixMode (GL_PROJECTION); /* Select the projection matrix */
27    glLoadIdentity ();            /* Initialise it */
28    glOrtho(-1.0,1.0, -1.0,1.0, -1.0,1.0); /* The unit cube */
29    glMatrixMode (GL_MODELVIEW);  /* Select the modelview matrix */
30 }
31
32  int main(int argc, char **argv) {
33   glutInit (&argc, argv);      /* Initialise OpenGL */
34   glutInitWindowSize (500, 500);     /* Set the window size */
35   glutInitWindowPosition (100, 100); /* Set the window position */
36   glutCreateWindow ("ex4");    /* Create the window */
37   glutDisplayFunc (display);   /* Register the "display" function */
38   glutReshapeFunc (reshape);   /* Register the "reshape" function */
39   glutKeyboardFunc (keyboard); /* Register the "keyboard" function */
40   glutMainLoop ();             /* Enter the OpenGL main loop */
41   return 0;
42 }
```

Try `ex4.c` out. You should see a red triangle on a grey background, as in Figure 5.2.

Although this is a simple example, it illustrates one of the most crucial aspects of OpenGL–**viewing**. OpenGL is a system for drawing 3D graphics. But display screens are 2D – they're flat. Figure 5.3 shows the situation.

In example `eg4.c`, we draw the triangle on the $Z = 0$ plane. But this is still 3D graphics!

5.2 Viewing using the camera

The idea of creating a 2D view of a 3D scene is simple: we 'take a picture' of the scene using a **camera**, and display the camera's picture in the window on the display screen. For convenience, OpenGL splits the process into three separate steps:

- **Step one**: First, we specify the position and orientation of the camera, using the function **gluLookAt()**;
- **Step two**: Second, we decide what kind of projection we'd like the camera to create. We can choose an **orthographic** projection (also known as a **parallel projection**) using the function **glOrtho()** (page 62); or a **perspective** projection using the function **gluPerspective()** (page 62);

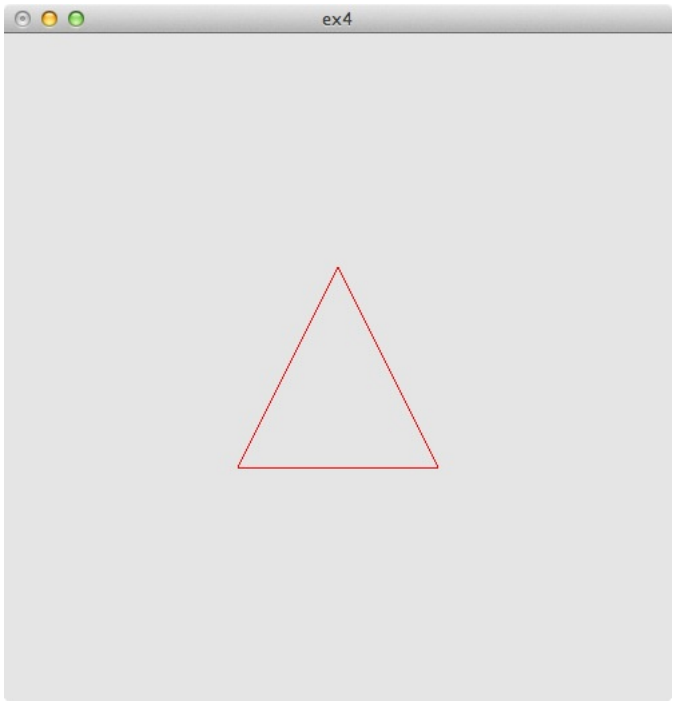
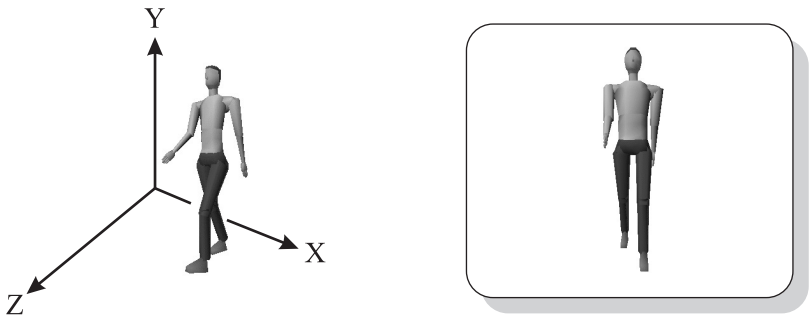


Figure 5.2: ex4.c.



OpenGL's 3D graphics "world" The flat display screen

Figure 5.3: OpenGL's 3D 'world', and the 2D display screen.

- **Step three**: Finally, we specify the size and shape of the camera's image we wish to see in the window, using **glViewport()** (page 64). This last step is optional – by default the camera's image is displayed using the whole window.

In OpenGL, the camera model described above is always active – you can't switch it off. It's implemented using **transformation matrices**, and we describe this in detail in Chapter 9. For now, here's a brief description of the process.

OpenGL keeps two transformation matrices: the **modelview** matrix, M; and the **projection matrix**, P. The modelview matrix holds a transformation which composes the scene in world coordinates, and then takes a view of the scene using the camera (step one, above). The projection matrix applies the camera projection (step two, above).

Whenever the application program specifies a coordinate c for drawing, OpenGL transforms the coordinate in two stages, as follows, to give a new coordinate c'. First it transforms the coordinate c by the matrix M, and then by the matrix P, as follows:

$$c' = P \cdot M \cdot c$$

When an OpenGL application starts up, P and M are unit matrices – they apply the **identity transformation** to coordinates, which has no effect on the coordinates. It's **entirely up to the application** to ensure that the M and P matrices always have suitable values. Normally, an application will set M in its `display()` function, and P in its `reshape()` function, as we shall now describe.

5.3 The window reshape function

After creating the window, and registering the display and keyboard callbacks, we now register a new function, the `reshape()` callback:

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

glutReshapeFunc() registers the application callback to call when the window is first created, and also if the window manager subsequently informs OpenGL that the user has reshaped the window. The new height and width of the window, in pixels, are passed to the callback. Typically, the callback will use these values to define the way that OpenGL's virtual camera projects its image onto the window, as we see in the next section.

5.3.1 Specifying the projection

We usually specify the projection in the `reshape()` callback function, because the projection will often need to be adjusted if the user changes the shape of the window. In example `ex4.c` we use an orthographic (also known as 'parallel') projection:

```
void reshape (int width, int height)
 { /* Called when the window is created, moved or resized */
   glViewport (0, 0, (GLsizei) width, (GLsizei) height);
```

```
    glMatrixMode (GL_PROJECTION); /* Select the projection matrix */
    glLoadIdentity ();
    glOrtho(-1.0,1.0, -1.0,1.0, -1.0,1.0); /* The unit cube */
    glMatrixMode (GL_MODELVIEW);  /* Select the modelview matrix */
}
```

We begin by setting the **viewport** using **glViewport()**, which specifies a rectangular portion of the window in which to display the camera's image. As in this example, it's common to use the the whole of the window, so we set the viewport to be a rectangle of equal dimensions to the window. We'll look at **glViewport()** in detail in Section 9.3.

Next, we set up an orthographic projection. **glMatrixMode()** (page 53) selects which matrix subsequent functions will affect – in this case we select the projection matrix (P). Then we initialise it to the unit transformation with **glLoadIdentity()** (page 54). This is very important, as we shall see in a moment. Then, we select the orthographic projection using **glOrtho()** (page 62). The projection we've chosen maps a unit cube, centred on the origin, onto the viewport.

glOrtho() actually does two things: first it creates a new temporary matrix (let's call it T) to implement the projection, and then it multiplies P with T, as follows:

$$P = P \cdot T$$

That's why we need to make sure P is initialised to the unit transformation first.

Note that the `reshape()` function ends with another call to **glMatrixMode()**, which this time selects the modelview matrix (M) for subsequent modification, for when we position the camera in the `display()` function.

5.3.2 Positioning the camera

This is usually done in the application's `display()` function, using the function **gluLookAt()**. We'll describe this function in detail in Section 9.1. In `ex4.c`, we use it to position the camera on the Z axis at $(0.0, 0.0, 0.5)$, looking towards the origin:

```
glLoadIdentity ();          /* start with a unit modelview matrix */
gluLookAt (0.0, 0.0, 0.5,   /* position of camera */
           0.0, 0.0, 0.0,   /* point at which camera looks */
           0.0, 1.0, 0.0); /* "up" direction of camera */
```

Again, because **gluLookAt()** creates a new transformation and multiplies it into the current matrix (M in this case), we need to ensure that M is first initialised using **glLoadIdentity()**.

5.4 Example 5: a 3D cube with perspective projection

We now turn to 3D drawing, and `ex5.c` draws a cube, centred on the origin (see Figure 5.4.

```
1 #include <GL/glut.h>
2
3 void display (void) {
4 /* Called when OpenGL needs to update the display */
```

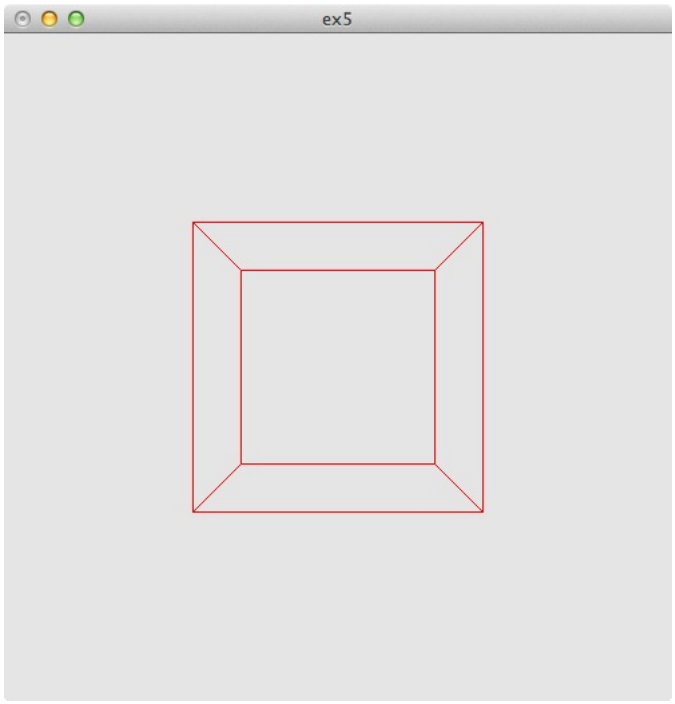


Figure 5.4: ex5.c.

```
5   glClearColor(0.9,0.9,0.9,0.0); /* Set grey background */
6   glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
7   glLoadIdentity ();
8   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
9   glColor3f(1.0,0.0,0.0);
10  glutWireCube(2.0);
11  glFlush();                    /* Force update of screen */
12 }
13
14 void keyboard (unsigned char key, int x, int y) {
15 /* Called when a key is pressed */
16   if (key == 27) exit (0);       /* 27 is the Escape key */
17 }
18
19 void reshape (int w, int h) {
20 /* Called if the window is moved or resized */
21    glViewport (0, 0, (GLsizei)w, (GLsizei)h);
22    glMatrixMode (GL_PROJECTION);
23    glLoadIdentity ();
24    gluPerspective (60, (GLfloat)w / (GLfloat)h, 1.0, 100.0);
25    glMatrixMode (GL_MODELVIEW);
26 }
27
28 int main(int argc, char **argv) {
29   glutInit (&argc, argv);            /* Initialise OpenGL */
30   glutInitWindowSize (500, 500);     /* Set the window size */
31   glutInitWindowPosition (100, 100); /* Set the window position */
32   glutCreateWindow ("ex5");          /* Create the window */
33   glutDisplayFunc (display);         /* Register the "display" function */
```

```
34   glutKeyboardFunc (keyboard);       /* Register the "keyboard" function */
35   glutReshapeFunc (reshape);         /* Register the "reshape" function */
36   glutMainLoop ();                   /* Enter the OpenGL main loop */
37   return 0;
38 }
```

Try `ex5.c` out.

In `display()`, we call **glutWireCube()**, which draws a wire-frame cube (see page 48). This time, however, we view it using a **perspective** projection as specified in our `reshape()` function:

```
gluPerspective (60, /* field of view in degrees */
                (GLfloat)w / (GLfloat)h, /* aspect ratio of view */
                1.0, 100.0); /* near and far clipping planes */
```

gluPerspective() sets a perspective projection, so we see the kind of view a camera would normally give, where lines further away from the viewer appear smaller. Here, we specify a field of view of 60 degrees, and an aspect (width-to-height) ratio for the view which exactly matches the aspect ratio of the window. We'll explain the use of clipping planes in Chapter 9.

5.5 What next?

Now onto Chapter 6, which looks at the use of **double buffering** for achieving smooth animation.

Chapter 6

Animated graphics

Computer graphics really comes to life when we draw images that **move**.

6.1 Example 6: a rotating cube

In this next example – `ex6.c` – we'll make OpenGL spin the cube about its centre. Have a look at the code, then take a copy of the program, and compile and run it (see Figure 6.1).

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

GLfloat angle= 0.0;

void spin (void) {
   angle+= 1.0;
   glutPostRedisplay();
}

void display(void) {
   glClearColor(0.9,0.9,0.9,0.0); /* Set grey backbround */
   glClear (GL_COLOR_BUFFER_BIT);
   glLoadIdentity ();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glRotatef(angle, 1, 0, 0);
   glRotatef(angle, 0, 1, 0);
   glRotatef(angle, 0, 0, 1);
   glColor3f(1.0,0.0,0.0);
   glutWireCube(2.0);
   glFlush();            /* Force update of screen */
}

void reshape (int w, int h) {
   glViewport (0, 0, (GLsizei)w, (GLsizei)h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (60, (GLfloat) w / (GLfloat) h, 1.0, 100.0);
   glMatrixMode (GL_MODELVIEW);
}

void keyboard(unsigned char key, int x, int y) {
```

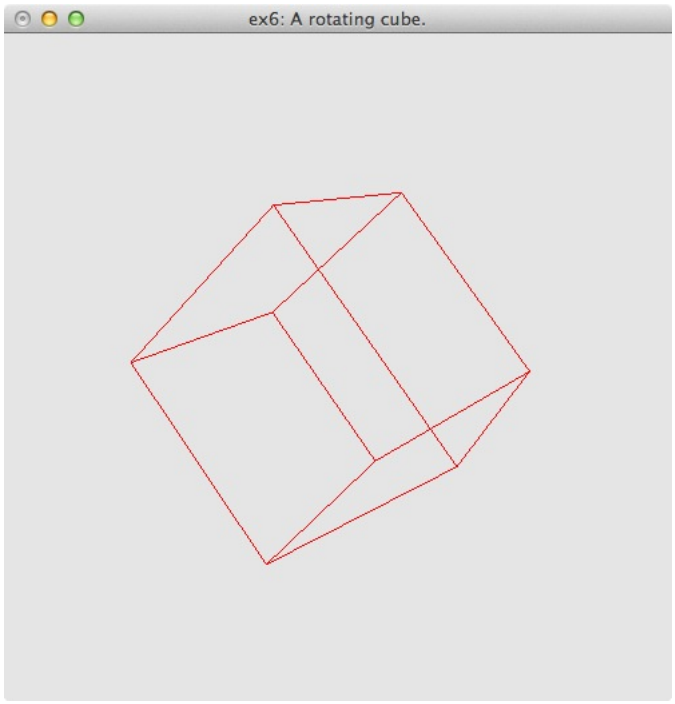


Figure 6.1: ex6.c.

```
32   if (key == 27)  exit (0);                 /* escape key */
33 }
34
35 int main(int argc, char **argv) {
36    glutInit(&argc, argv);
37    glutInitWindowSize (500, 500);
38    glutInitWindowPosition (100, 100);
39    glutCreateWindow ("ex6: A rotating cube.");
40    glutDisplayFunc(display);
41    glutReshapeFunc(reshape);
42    glutKeyboardFunc(keyboard);
43    glutIdleFunc(spin);          /* Register the "idle" function */
44    glutMainLoop();
45    return 0;
46 }
```

You should see the cube rotating, but in a rather horrible broken-up sort of way. We'll come back to that in a moment.

The engine behind the animation is the event loop. Using **glutIdleFunc()**, we register an application callback function that gets called each time around the **glutMainLoop()**:

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

glutIdleFunc() registers a callback which will be automatically be called by OpenGL in **each cycle** of the event loop, **after** OpenGL has checked for any events and called the relevant callbacks.

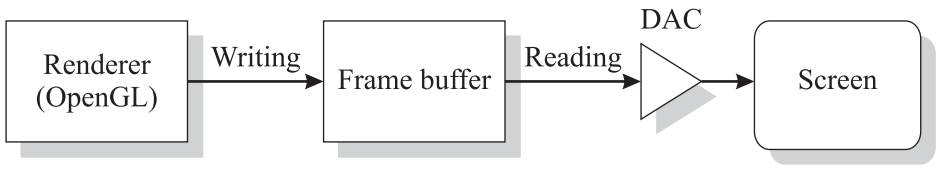


Figure 6.2: Single buffering.

In `ex6.c`, the idle function we've registered is called `spin()`:

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

First `spin()` increments the global variable `angle`. Then, it calls **glutPostRedisplay()**, which tells OpenGL that the window needs redrawing:

void **glutPostRedisplay** (void);

glutPostRedisplay() tells OpenGL that the application is asking for the display to be refreshed. OpenGL will call the application's `display()` callback at the next opportunity, which will be during the next cycle of the event loop.

Note: While OpenGL is processing a single cycle of the event loop, several callbacks may call **glutPostRedisplay()**. Nevertheless, OpenGL won't actually call the display callback until all outstanding events have been dealt with. And, within one cycle of the event loop, a succession of outstanding calls to **glutPostRedisplay()** will be treated as a single call to **glutPostRedisplay()**, so display callbacks will only be executed once – which is probably what you want.

6.2 Double-buffering and animation

As we saw, the rotating cube looks horrible. Why?

The problem is that OpenGL is operating **asynchronously** with the refreshing of the display. OpenGL is pumping out frames too fast: it's writing (into the frame-buffer) a new image of the cube in a slightly rotated position, **before** the previous image has been completely displayed.

Recall the architecture of raster displays: as shown in Figure 6.2, the pixel data is stored in the frame buffer, which is repeatedly read (typically at 60 Hz) by the digital-to-analogue converter (DAC) to control the intensity of the electron beam as it sweeps across the screen, one scan-line at a time. With a single frame-buffer, the renderer (OpenGL) is writing pixel information into the buffer **at the same time** the DAC is reading the information out. If the writer and the reader are out of sync, the reader can never be guaranteed to read and display a complete frame – so the viewer always sees images which comprise part of one frame and part of another. This is very disturbing to the eye – and destroys any possibility of seeing

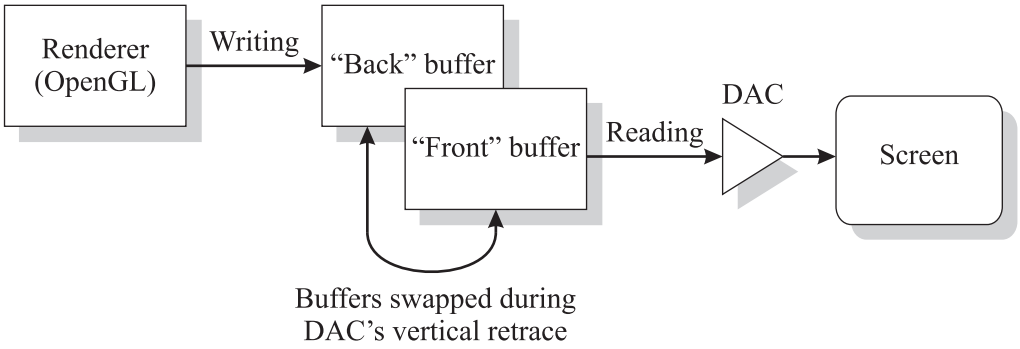


Figure 6.3: Double buffering.

smooth animation.

One solution is to use an additional buffer, as shown in Figure 6.3. The idea here is that one buffer, called the 'back buffer' is only ever **written to** by the renderer. The other buffer – the 'front buffer' – is only ever **read by** the DAC. The renderer writes its new frame into the back buffer, and when that's done, it then requests that the back and front buffers be swapped over. The **trick** is to perform the swapping while the DAC is performing its **vertical retrace**, which is when it's finished a complete sweep of its buffer, and is resetting to begin again. There's enough slack time here to swap the contents of the two buffers over. This method will ensure that the DAC only ever reads and displays a complete frame.

By default, OpenGL works in **single-buffer** mode, and so we get the fragmented animation seen in `ex6.c`. But we can tell OpenGL to use double-buffering, using **glutInitDisplayMode()**:

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

glutInitDisplayMode() sets the **current display mode**, which which will be used for a window created using **glutCreateWindow()**. `mode` is:

- `GLUT_SINGLE`: selects a single-buffered window – which is the default if **glutInitDisplayMode** isn't called;
- `GLUT_DOUBLE`: selects a double-buffered window;

(There are more display modes, beyond the scope of this manual. For a full description, see the GLUT manual or the Red Book.)

For example, to select a double-buffered window you would call:

```
glutInitDisplayMode (GLUT_DOUBLE);
glutCreateWindow ("my window");
```

Once we're using double-buffering, we can tell OpenGL that a frame is complete, and that the buffers should be swapped using **glutSwapBuffers()**:

void **glutSwapBuffers** (void);

glutSwapBuffers() swaps the back buffer with the front buffer, at the next opportunity, which is normally the next vertical retrace of the monitor. The contents of the new back buffer (which was the old front buffer) are undefined.

Note: Swapping the buffers doesn't have the side effect of **clearing** any buffers. Clearing a buffer must be done explicitly by the application, by calling **glClear()**. Note again that now we're using double-buffering, it's no longer necessary to use **glFlush()**.

6.3 Exercise: smooth the cube

Edit your copy of `ex6.c` as follows:

- In `main()`, after the call to **glutInit()**, insert a call to **glutInitDisplayMode()** to select a double-buffered window;
- In `display()`, after the call to **glutWireCube()**, insert a call to **glutSwapBuffers()**.
- Also, **remove** the call to **glFlush()**. We don't need that anymore, since it gets called internally by **glutSwapBuffers()**. And if we leave **glFlush()** in the code, not only will its effect be redundant, but it'll also slow the program down.

See the difference? Smooth animation!

6.4 Example 7: rotating objects following the mouse

Finally, we now extend `ex6.c` to display a few different objects, and to follow the mouse around.

We won't describe the code here – have a look at `ex7.c` on-line for yourself. And try running it. Cycle between the various objects by pressing the `space` key.

The main new functions we use are **glutPassiveMotionFunc()** (page 74) and **gluUnProject()** (page 65).

6.5 What next?

This is the end of the Tutorial section of the manual. The remaining chapters form the OpenGL Reference Manual.

Part II

OpenGL Reference Manual

Chapter 7

Graphics primitives

In this chapter we describe the coordinate system OpenGL uses, and some of the OpenGL graphics primitives.

7.1 Coordinate systems

OpenGL uses right-handed Cartesian coordinate systems, as shown in Figure 7.1.

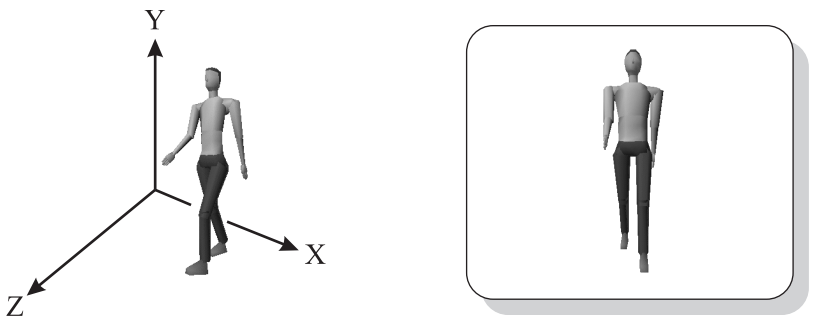


Figure 7.1: A right-handed coordinate system. The positive Z axis comes out of the page.

By convention, we draw the positive X axis heading rightwards, the positive Y axis heading vertically, with the positive Z axis heading out of the page towards you.

All the OpenGL functions which create graphical primitives such as lines and polygons work in **object coordinates**. OpenGL automatically transforms object coordinates, first by the **modelview matrix** (M) and then by the **projection matrix** (P). We describe the modelview matrix and the projection matrix in Chapters 8 and 9.

7.2 Defining a vertex

The basic building block for creating graphics with OpenGL is a **point in 3D space**. To describe a shape, you specify the set of points that together make up the shape. In OpenGL terminology, a point in 3D space is called a **vertex**.

You define a single vertex using the function **glVertex3f()**:

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

Here, the '3f' part of the function name means that the function takes three arguments, each of which is a `GLfloat`. As we described in Section 2.7, GL uses its own data types. `GLfloat` is equivalent to the C type `float`.

So, for example, to define a vertex at (10,8,5), you would call:

```
glVertex3f (10.0, 8.0, 5.0);
```

7.3 OpenGL function flavours

Many OpenGL functions come in several flavours. For example, suppose you only ever want to do 2D drawing, so you're only concerned with specifying vertices in the XY plane, and all vertices will have a Z coordinate of 0. To make life easier, OpenGL offers a variant form of the **glVertex3f()** function, called **glVertex2f()**:

```
void glVertex2f ( GLfloat x,
                  GLfloat y );
```

Internally, this function still creates a 3D vertex, but it sets its Z coordinate to 0.0 for you, to save you the bother. But in this manual, we will always use the 3D form of functions – the less functions we have to remember, the better!

7.4 Defining shapes: primitives

A vertex on its own isn't very interesting. Now we look at how to group vertices together into **vertex lists**, which define geometrical shapes. The grouping of vertices is done with the **glBegin()** and **glEnd()** functions:

```
void glBegin ( GLenum mode );
```

glBegin() defines the start of a vertex list. `mode` determines the kind of shape the vertices describe, which can be:

- A set of unconnected points (`GL_POINTS`);

- Lines (GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP);
- The boundary of a single convex polygon (GL_POLYGON);
- A collection of triangles (GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN);
- A collection of quadrilaterals (GL_QUADS, GL_QUAD_STRIP).

void **glEnd** (void);

glEnd() defines the end of a vertex list.

7.4.1 A common mistake to avoid!

OpenGL places restrictions on what can and can't be included in a **glBegin()**–**glEnd()** block. Only a small number of commands are permitted, of which the folowing are the most common: **glVertex3f()**, **glNormal3f()**, **glColor3f()**, **glTexCoord()**.

Calling other functions – in particular transformation functions like **glRotatef()**, or attribute setting functions like **glLineWidth()** may result in incorrect/invalid results.

For a complete list of valid functions, see http://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml.

7.5 Drawing points

We use the following mode in **glBegin()** to draw points:

- GL_POINTS: each vertex represents a point.

```
glBegin (GL_POINTS);
   glVertex3f (0.0, 6.0, 4.0);
   glVertex3f (0.0, 8.0, 0.0);
   glVertex3f (8.0, 6.0, 0.0);
   glVertex3f (8.0, 3.0, 0.0);
   glVertex3f (6.0, 0.0, 5.0);
   glVertex3f (2.0, 0.0, 5.0);
glEnd ();
```

7.6 Drawing lines

In the function **glBegin()**, the values of mode which interpret vertices as points to connect with lines are:

- GL_LINES: each pair of vertices is drawn as a separate line.

- `GL_LINE_STRIP`: all the vertices are joined up with lines.
- `GL_LINE_LOOP`: all the vertices are joined up with lines, and an extra line is drawn from the last vertex to the first.

Figure 7.2 illustrates how the same set of vertices can be drawn as lines in different ways according to `mode`:

```
glBegin (GL_LINES);  /* or GL_LINE_STRIP or GL_LINE_LOOP */
   glVertex3f (0.0, 6.0, 4.0);
   glVertex3f (0.0, 8.0, 0.0);
   glVertex3f (8.0, 6.0, 0.0);
   glVertex3f (8.0, 3.0, 0.0);
   glVertex3f (6.0, 0.0, 5.0);
   glVertex3f (2.0, 0.0, 5.0);
glEnd ();
```

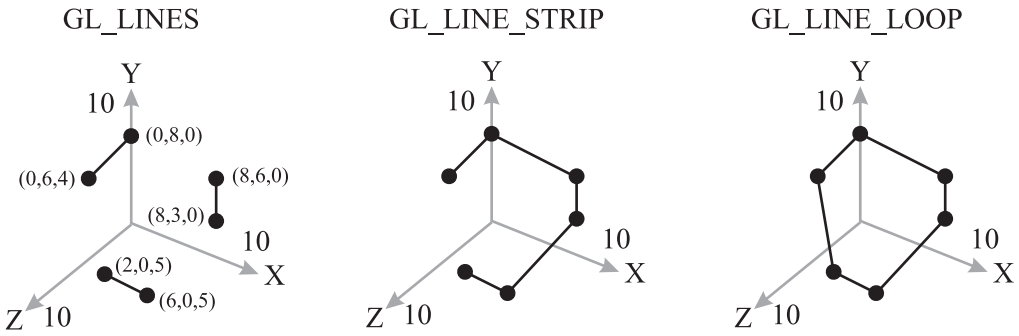


Figure 7.2: The same set of vertices drawn using different line styles.

As well as geometry, primitives also have **attributes**, which control ther visual style.

7.6.1 Line attributes

void **glLineWidth** (GLfloat *width*);

glLineWidth() sets the curent line width, measured in pixels. The default value is 1.0.

void **glLineStipple** (GLint *factor*,
GLushort *pattern*);

glLineStipple() sets the stippling pattern for lines, which enables lines to be drawn in a flexible variety of dot/dash patterns. By default, stippling is switched off (see Section 7.6.2), and must be enabled by calling:

```
glEnable(GL_LINE_STIPPLE);
```

Line stippling works on a pixel-by-pixel basis, as the line is rendered into the frame buffer. `pattern` is a 16-bit series of 0s and 1s. When OpenGL renders a line, for each pixel it is

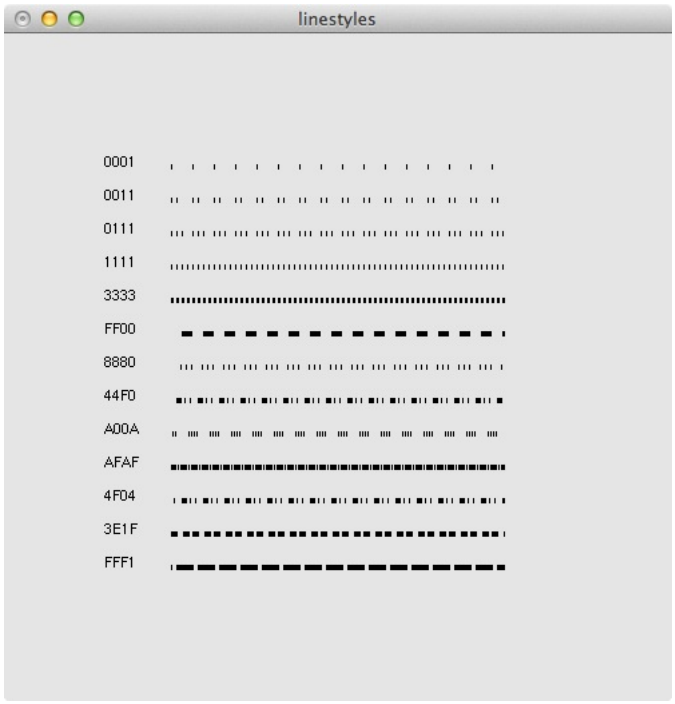


Figure 7.3: linestyles.c.

about to write, it first consults the next bit in `pattern`, starting at the **low-order bit**, If this bit is a 1, the pixel is written, in the current drawing colour. If the bit is a 0, the pixel is not written.

For example, suppose the pattern specified was (to choose a random example) `0x3E1F`. In binary this is `0011 1110 0001 1111`.

So, when drawing a line, OpenGL would draw the first 5 pixels on, the next 4 off, then one on, the next five on, and the next 2 off. For the next pixel, OpenGL would return to the low-order bit of the pattern, and repeat.

`factor` is a way of elongating the pattern – it multiplies each sub-sequence of consecutive 0s and 1s. For example, if factor=3, then if the bit series `0110` appeared in the pattern, it would be 'stretched' to be `01111110`. Figure 7.3 shows some examples of different patterns, from the example program `linestyles.c`. Here, `factor` is set to 1.0. Try changing the patterns, and the factor.

7.6.2 Enabling OpenGL capabilities

void **glEnable** (GLenum *capability*);

void **glDisable** (GLenum *capability*);

OpenGL has a number of capabilities which by default are not active – for reasons of effi-

ciency. These include lighting, texturing, hidden surface removal and line stippling. To use one of these capabilities, it must be explicitly 'enabled' by the application, using **glEnable()**. The capability may be subsequently disabled using **glDisable()**. Some of the valid values of `capability` are:

- `GL_LINE_STIPPLE`
- `GL_LIGHTING`
- `GL_FOG`
- `GL_DEPTH_TEST`

7.7 Drawing triangles

The different values of `mode` in **glBegin()** to create triangles are:

- `GL_TRIANGLES`: each triplet of points is drawn as a separate triangle. If the number of vertices is not an exact multiple of 3, the final one or two vertices are ignored.
- `GL_TRIANGLE_STRIP`: constructs a set of triangles with the vertices v0, v1, v2 then v2, v1, v3 then v2, v3, v4 and so on. The ordering is to ensure that the triangles are all drawn correctly form part of surface.
- `GL_TRIANGLE_FAN`: draws a set of triangles with the vertices v0, v1, v2 then v0, v2, v3 then v0, v3, v4 and so on.

```
glBegin (GL_TRIANGLES);
   glVertex3f (0.0, 6.0, 4.0);
   glVertex3f (0.0, 8.0, 0.0);
   glVertex3f (8.0, 6.0, 0.0);
   glVertex3f (8.0, 3.0, 0.0);
   glVertex3f (6.0, 0.0, 5.0);
   glVertex3f (2.0, 0.0, 5.0);
glEnd ();
```

7.8 Drawing quadrilaterals

We can use two values for `mode` in **glBegin()** to create quadrilaterals.

- `GL_QUADS`: each set of four vertices is drawn as a separate quadrilaterals. If the number of vertices is not an exact multiple of 4, the final one, two or three vertices are ignored.
- `GL_QUAD_STRIP`: constructs a set of quadrilaterals with the vertices v0, v1, v3, v2 then v2, v3, v5, v4 then v4, v5, v7, v6 and so on.

```
glBegin (GL_QUADS); /* or GL_QUAD_STRIP */
   glVertex3f (0.0, 6.0, 4.0);
   glVertex3f (0.0, 8.0, 0.0);
   glVertex3f (8.0, 6.0, 0.0);
   glVertex3f (8.0, 3.0, 0.0);
   glVertex3f (6.0, 0.0, 5.0);
   glVertex3f (2.0, 0.0, 5.0);
glEnd ();
```

7.9 Drawing polygons

We draw a polygon using the following `mode` in **glBegin()**:

- `GL_POLYGON`: the vertices define the boundary of a single convex polygon.

The polygon specified must not intersect itself and must be convex. Figure 7.4 shows a polygon with 5 vertices, drawn with the following code:

```
glBegin (GL_POLYGON)
   glVertex3f (0.0, 6.0, 0.0);
   glVertex3f (0.0, 6.0, 6.0);
   glVertex3f (6.0, 6.0, 6.0);
   glVertex3f (9.0, 6.0, 2.0);
   glVertex3f (9.0, 6.0, 0.0);
glEnd ();
```

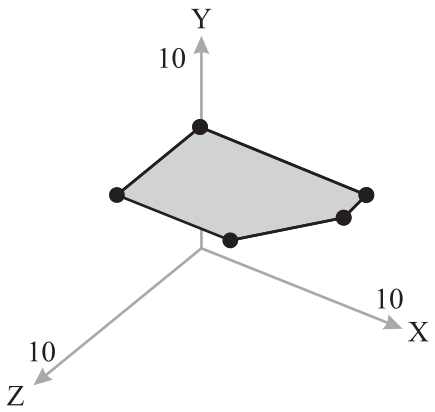


Figure 7.4: A simple polygon with 5 vertices.

For efficiency and simplicity, OpenGL only guarantees to draw a polygon **correctly** if it's **convex**. A polygon is convex if, taking any pair of points inside the polygon and drawing a straight line between them, all points along the line are also inside the polygon. Figure 7.5 shows a few examples of convex polygons (on the left) and non-convex polygons (on the right).

Note: to draw a non-convex polgyon in OpenGL, it must first be broken into a set of convex polygons, each of which is then drawn separately. This process is called **tesselation**, and

non-convex polygons can be broken down this way. GLU provides a set of functions for doing this – see the Red Book, Chapter 11.

Polygons must also be **planar** (completely flat) if they are to be rendered correctly.

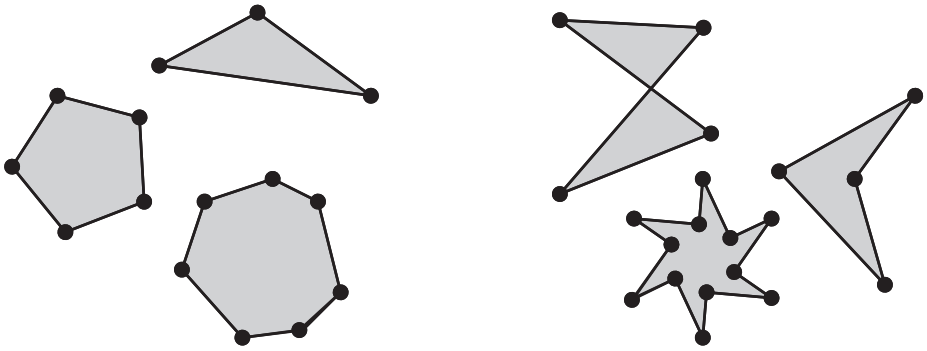


Figure 7.5: Convex polygons (left) and non-convex polygons (right).

7.9.1 Polygon attributes

```
void glPolygonMode ( GLenum face,
                     GLenum mode );
```

glPolygonMode() sets the drawing mode for polygons.

`face` can be `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`. `mode` can be `GL_FILL`, or `GL_LINE`.

7.10 GLUT's primitives

GLUT provides a number of functions for easily drawing more complicated objects. Each comes in two versions: **wire** and **solid**. The wire forms are drawn using lines (`GL_LINE` or `GL_LINE_LOOP`); the solid forms use polygons (with surface normals, suitable for creating lit, shaded images). Note that these objects do not use display lists (see Chapter 14).

7.10.1 Cube

```
void glutWireCube ( GLdouble size );
```

glutWireCube() draws a cube, with edge length `size`, centred on $(0, 0, 0)$ in object coordinates. Solid version: **glutSolidCube()**.

7.10.2 Sphere

```
void glutWireSphere ( GLdouble radius,
                      GLint slices,
                      GLint stacks );
```

glutWireSphere() draws a sphere, of radius `radius`, centred on $(0, 0, 0)$ in object coordinates. `slices` is the number of subdivisions around the Z axis (like lines of longitude); `stacks` is the number of subdivisions along the Z axis (like lines of latitude). Solid version: **glutSolidSphere()**.

7.10.3 Cone

```
void glutWireCone ( GLdouble base,
                    GLdouble height,
                    GLint slices,
                    GLint stacks );
```

glutWireCone() draws a cone, with base radius `radius`, and height `height`. The cone is oriented along the Z axis, with the base placed at $Z = 0$, and the apex at $Z =$ height. `slices` is the number of subdivisions around the Z axis; `stacks` is the number of subdivisions along the Z axis. Solid version: **glutSolidCone()**.

```
void glutWireTorus ( GLdouble innerRadius,
                     GLdouble outerRadius,
                     GLint nsides,
                     GLint rings );
```

glutWireTorus() draws a torus centred on $(0, 0, 0)$ in object coordinates. The axis of the torus is aligned with the Z axis. `innerRadius` and `outerRadius` give the inner and outer radii of the torus respectively; `nsides` is the number of sides in each radial section, and `rings` is the number of radial sections. Solid version: **glutSolidTorus()**.

7.10.4 Platonic solids

```
void glutWireTetrahedron ( void );
```

glutWireTetrahedron() draws a tetrahedron (4-sided regular object) of radius $\sqrt{3}$ centred on $(0, 0, 0)$ in object coordinates. Solid version: **glutSolidTetrahedron()**.

```
void glutWireOctahedron ( void );
```

glutWireOctahedron() draws an octahedron (8-sided regular object) of radius 1 centred on

$(0, 0, 0)$ in object coordinates. Solid version: **glutSolidOctahedron()**.

void **glutWireDodecahedron** (void);

glutWireDodecahedron() draws a dodecahedron (12-sided regular object) of radius $\sqrt{3}$ centred on $(0, 0, 0)$ in object coordinates. Solid version: **glutSolidDodecahedron()**.

void **glutWireIcosahedron** (void);

glutWireIcosahedron() draws an icosahedron (20-sided regular object) of radius 1 centred on $(0, 0, 0)$ in object coordinates. Solid version: **glutSolidIcosahedron()**.

7.10.5 Teapot

void **glutWireTeapot** (GLdouble *scale*);

glutWireTeapot() draws a teapot, scaled by `scale`. Solid version: **glutSolidTeapot()**.

Chapter 8

Modelling using transformations

This chapter is about modelling: we explain how to use transformations to assemble 3D scenes. Chapter 9 explains how to create a view of the scene using the camera model.

8.1 Vectors and matrices

We saw in Section 7.1 that a 3D vertex – a point in space – is represented as x, y, z, mathematically, we write a 3D point as a **column vector**: If we have a point p, we write it as:

$$p \leftarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

You'll notice the extra '1' at the bottom of the vector – this is known as a **homogeneous** representation. To cut a long story short, the use of such vector representations is a mathematical trick which allows all common transformation types to be expressed in a consistent manner using 4×4 matrices.

Warning: beware that some Computer Graphics textbooks represent coordinates as **row vectors**. Using row vectors doesn't change the basic methods used for matrix transformations, but the order in which matrices appear, and their rows and columns, are reversed. Trying to think in terms of both column and row vectors is a recipe for disaster. Stick to column vectors always.

In OpenGL all coordinate transformations are specified using 4×4 matrices. If we transform a point p with a matrix M (for example, a scale by sx, sy, sz), we get a transformed point p', as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \leftarrow \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or, more succinctly:

$$p' \leftarrow M \cdot p$$

If we subsequently transform p' by another matrix N, to give p'', we have:

$$p'' \leftarrow N \cdot p'$$

so expressing the entire transformation we have:

$$p'' \leftarrow N \cdot M \cdot p$$

8.2 A note about matrix ordering

Notice that the order in which the matrices are written, reading from left to right, is the **reverse** of the order in which their transformations are applied. In the above example, the first transformation applied to p is M, and the transformed point is then subsequently transformed by N.

In general, matrix multiplication is **not commutative**. So, with two matrices M and N,

$$M \cdot N \neq N \cdot M$$

In other words, the **order** in which transformation matrices are applied is crucial.

One of the most common problems in computer graphics, **blank screen syndrome**, is often due to incorrectly ordered matrix transformations. Your image has been lovingly computed, but it is being displayed several miles to the West of your display screen; or that tiny blob in the left-hand corner of your screen is your image, compressed into a few pixels.

8.3 Selecting the current matrix

OpenGL maintains two separate 4×4 transformation matrices:

- the **modelview** matrix; this is used to accumulate modelling transformations, and also to specify the position and orientation of the camera;
- the **projection** matrix; this is used to specify how the 3D OpenGL world is transformed into a 2D camera image for subsequent display. The projection matrix performs either a perspective or othographic (parallel) projection.

At any time, one or the other of these matrices is selected for use, and is called the **current matrix**, or sometimes C for short. Most of the OpenGL functions for managing transformations affect the contents of C.

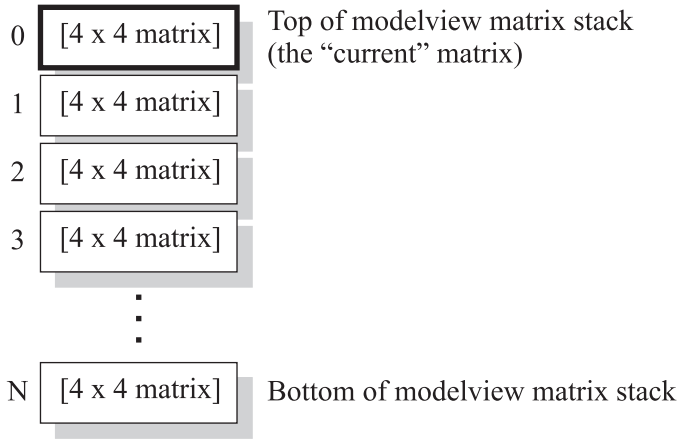


Figure 8.1: An OpenGL modelview matrix stack. The top element of the **selected** stack is often referred to as the 'current matrix' C.

8.4 Setting the current matrix

If our first transformation M represents a scale by (sx, sy, sz), and the second transformation N a translation by (tx, ty, tz), we would code this in OpenGL as follows:

```
glMatrixMode (GL_MODELVIEW); /* Select the modelview matrix */
glLoadIdentity ();           /* Set the current matrix to identity */
glTranslatef (tx, ty, tz);   /* Post-multiply by (tx,ty,tz) */
glScalef (sx, sy, sz);       /* Post-multiply by (sx,sy,sz) */
glVertex3f(x, y, z);         /* Define the vertex */
```

Note that all the OpenGL functions which affect the current matrix C do so by **post-multiplication**. This means that we write the sequence of OpenGL transformation functions in the **reverse order** to the effect they actually have on vertices. This can take a bit of getting used to.

In fact, the modelview matrix isn't a single matrix stored somewhere inside OpenGL – it's actually the top matrix on a **stack** of modelview matrices. This is shown in Figure 8.1. Similarly, the projection matrix is the top matrix on a **stack** of projection matrices. We'll see later why OpenGL uses stacks of matrices.

Figure 8.2 shows how the modelview and projection matrices on the top of their respective stacks affect a vertex specified by the application.

void **glMatrixMode** (GLenum *mode*);

glMatrixMode() selects the matrix stack, and makes the top matrix on the stack the 'current matrix' (C). `mode` selects the matrix stack, as follows: `GL_MODELVIEW`: selects the modelview matrix stack; `GL_PROJECTION`: selects the projection matrix stack.

Once a current matrix has been selected using **glMatrixMode()**, all subsequent matrix functions (such as **glRotatef()**, etc.) affect the current matrix. For example, to load a translation by (x, y, z) into the modelview matrix, the code would be:

```
glMatrixMode (GL_MODELVIEW);
```

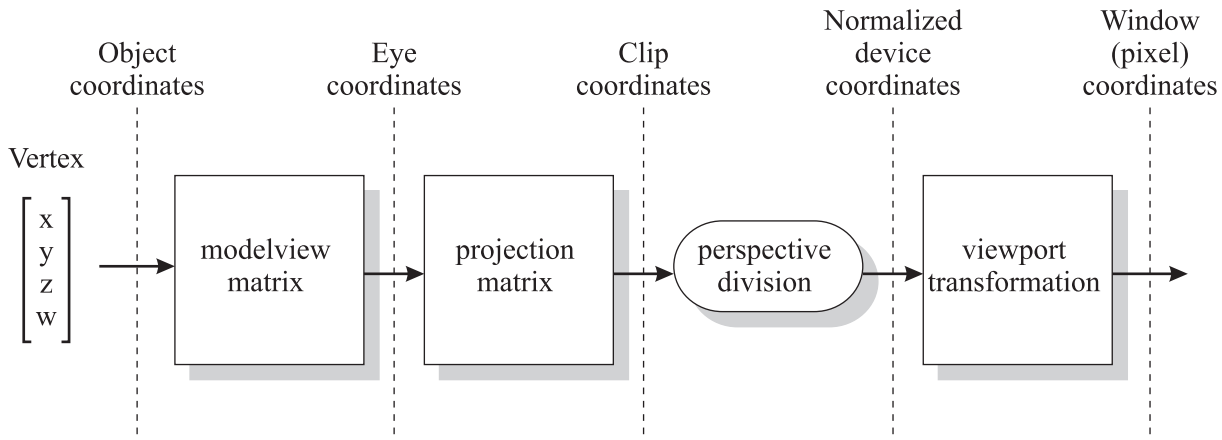


Figure 8.2: The OpenGL viewing pipeline, showing the sequence of transformations and operations applied to a 3D vertex.

```
glLoadIdentity ();
glTranslatef (x, y, z);
```

Subsequent matrix operations will continue to affect the current modelview matrix, until **glMatrixMode()** is called again to select a different matrix.

8.5 Operating on the current matrix

There are a number of utility functions for changing the current matrix.

8.5.1 Setting to identity

void **glLoadIdentity** (void);

glLoadIdentity() sets the current matrix C to be the identity matrix I:

$$C \leftarrow I$$

where

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

8.5.2 Translation

```
void glTranslatef ( GLfloat x,
                    GLfloat y,
                    GLfloat z );
```

glTranslatef() creates a matrix M which performs a translation by (x, y, z), and then post-multiplies the current matrix by M as follows:

$$C \leftarrow C \cdot M$$

8.5.3 Scaling

```
void glScalef ( GLfloat x,
                GLfloat y,
                GLfloat z );
```

glScalef() creates a matrix M which performs a scale by (x, y, z), and then post-multiplies the current matrix by M as follows:

$$C \leftarrow C \cdot M$$

8.5.4 Rotation

```
void glRotatef ( GLfloat angle,
                 GLfloat x,
                 GLfloat y,
                 GLfloat z );
```

glRotatef creates a matrix M which performs a counter-clockwise rotation of `angle` degrees. The axis about which the rotation occurs is the vector from the origin $(0, 0, 0)$ to the point (x, y, z), and then post-multiplies the current matrix by M as follows:

$$C \leftarrow C \cdot M$$

8.6 Using the matrix stacks

Because all the OpenGL transformation functions (like **glTranslate()**) always change the current matrix by post-multiplying with the new transformation, sometimes it can be awkward to easily get the correct sequence of transformations. This is where the matrix stacks come in. There are two separate matrix stacks: one for the modelview matrix and one for the projection matrix. Only one matrix stack is current at a particular time, and this is selected by

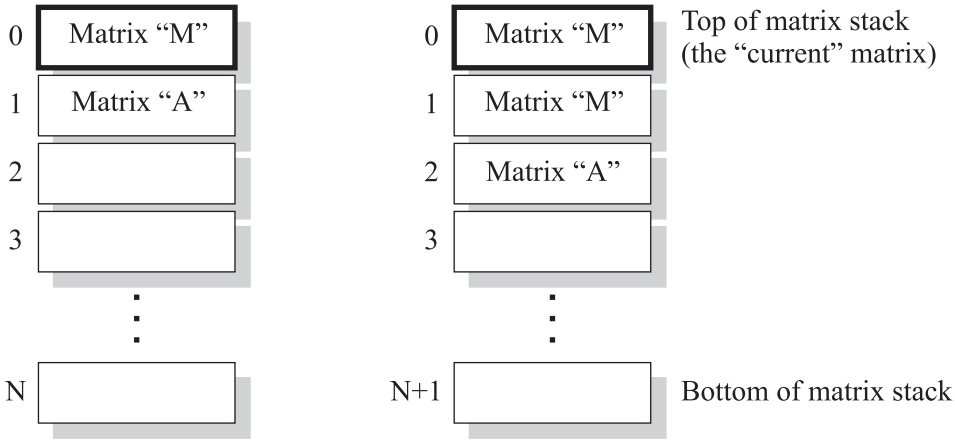


Figure 8.3: The effect of calling **glPushMatrix()** on the current OpenGL matrix stack. The figure shows the stack before (left) and after (right) **glPushMatrix()** is called.

calling **glMatrixMode()**. There are two functions which operate on the current matrix stack: **glPushMatrix()** and **glPopMatrix()**. They behave as you might expect:

```
void glPushMatrix ( void );
```

glPushMatrix() Pushes the current matrix stack down one level. The matrix on the top of the stack is copied into the next-to-top position, as shown in Figure 8.3. The current matrix stack is determined by the most recent call to **glMatrixMode()**. C is not changed. It is an error if **glPushMatrix()** is called when the stack is full.

Correspondingly, there's a function to pop a matrix off the stack:

```
void glPopMatrix ( void );
```

glPopMatrix() pops the current matrix stack, moving each matrix in the stack one position towards the top of the stack, as shown in Figure 8.4. The current matrix stack is determined by the most recent call to **glMatrixMode()**. C becomes the matrix previously at the second-to-top of the stack. It is an error if **glPopMatrix()** is called when the stack contains only one matrix.

8.7 Creating arbitrary matrices

You can usually create the matrices you need by using the simple matrix manipulation functions **glLoadIdentity()**, **glTranslate()**, **glScale()** and **glRotate()**, but sometimes – and this is an advanced topic – you need to provide arbitrary 4×4 matrices of your own. See Appendix C for details of how to do this.

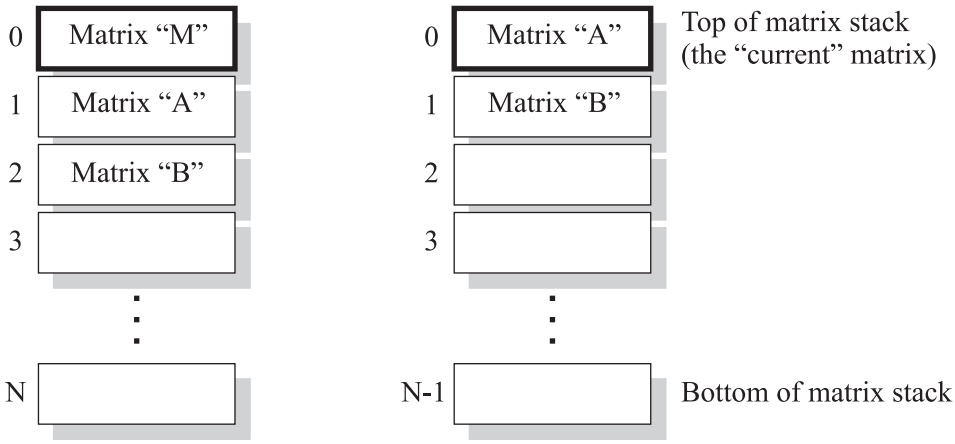


Figure 8.4: The effect of calling **glPopMatrix()** on an OpenGL matrix stack. The figure shows the stack before (left) and after (right) **glPopMatrix()** is called.

Chapter 9

Viewing

In this chapter we look at how to use the OpenGL **viewing model**. The idea is simple: we create a 3D scene using modelling transformations. We then 'take a picture' of the scene using a **camera**, and display the camera's picture on the display screen. For convenience, OpenGL splits the process into three separate parts:

- First, we specify the position and orientation of the camera, using **gluLookAt()**.
- Second, we decide what kind of picture we'd like the camera to create. Usually, for 2D graphics we'll use an orthographic (also known as 'parallel') view using **glOrtho()**). For 3D viewing, we'll usually want a perspective view, using **gluPerspective()**.
- Finally, we describe how to map the camera's image onto the display screen, using **glViewport()**.

9.1 Controlling the camera

Let's look again at the OpenGL viewing pipeline, in Figure 9.1.

We set the position and orientation of the OpenGL camera, as shown in Figure 9.2, using **gluLookAt()**:

```
void gluLookAt ( GLdouble eyex,
                 GLdouble eyey,
                 GLdouble eyez,
                 GLdouble centerx,
                 GLdouble centery,
                 GLdouble centerz,
                 GLdouble upx,
                 GLdouble upy,
                 GLdouble upz );
```

The position of the camera in space – sometimes also called the **eyepoint** – is given by `(eyex, eyey, eyez)`. `(centerx, centery, centerz)` specifies a **look point** for the

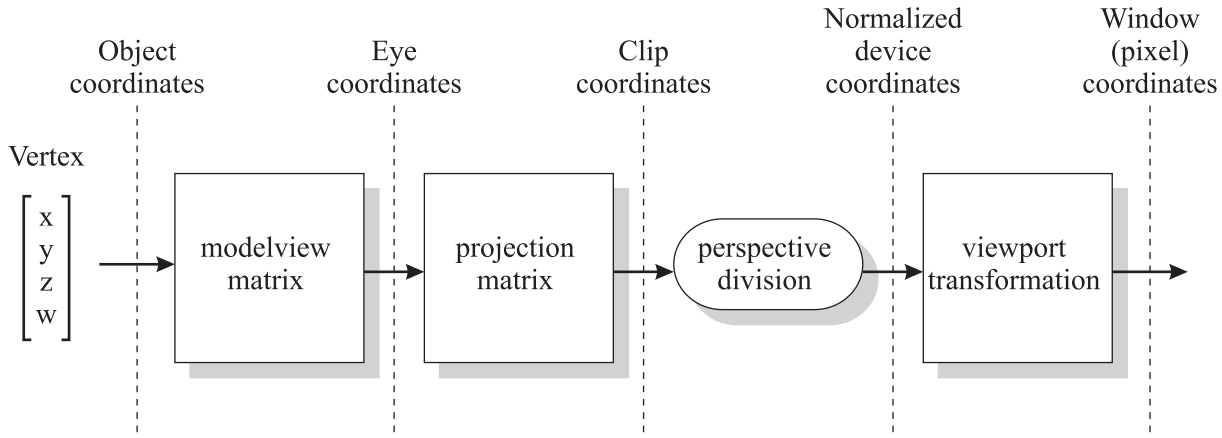


Figure 9.1: The OpenGL viewing pipeline, showing the sequence of transformations and operations applied to a 3D vertex.

camera to 'look at', and a good choice for this would a point of interest in the scene, and often the center of the scene is used. Together, the points `(eyex, eyey, eyez)` and `(centerx, centery, centerz)` define a **view vector**. The last set of **gluLookAt()**'s arguments specify the **up vector** of the camera. This defines the camera's orientation at the eyepoint.

There is no need for the view vector and the up vector to be defined at right angles to each other (although if they're parallel, weird views may result). Often the up vector is set to a fixed direction in the scene, e.g. pointing up the world Y axis. In the general case, OpenGL twists the camera around the view vector axis until the top of the camera matches the specified up direction as closely as possible.

What **gluLookAt()** actually does is to create a transformation matrix which encapsulates all the specified camera parameters. This is called the 'viewing matrix', or V. **gluLookAt()** then post-multiplies the current modelview matrix (C) by V:

$$C \leftarrow C \cdot V$$

If you don't call **gluLookAt()**, the OpenGL camera is given some default settings:

- it's located at the origin, $(0, 0, 0)$;
- it looks down the negative Z axis;
- its 'up' direction is parallel to the Y axis.

This is the same as if you had called **gluLookAt()** as follows:

```
gluLookAt (0.0, 0.0, 0.0,  /* camera position */
           0.0, 0.0, -1.0, /* point of interest */
           0.0, 1.0, 0.0); /* up direction */
```

Note that the value -1.0 could be any negative value, because this is just specifying the direction 'down the negative Z axis'.

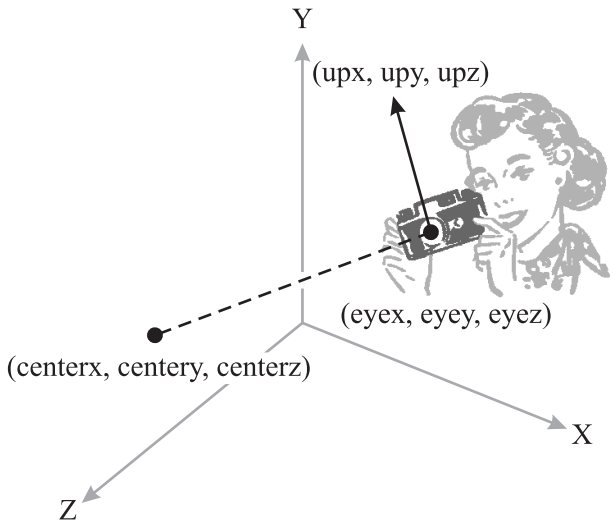


Figure 9.2: The OpenGL camera.

9.2 Projections

Now that we've positioned and pointed the OpenGL camera, the next step is to specify what kind of image we want. This is done using the **projection matrix**, P. OpenGL applies the projection transformation **after** it has applied the modelview transformation.

9.2.1 The view volume

Consider the real world camera analogy, in which we choose the lens type (wide-angle, telephoto etc.). The choice of lens affects the field of view, and selects what portion of the 3D world will appear within the bounds of final image. The volume of space which eventually appears in the image is known as the **view volume** (or **view frustum**). As well as discarding objects which lie outside the image 'frame' OpenGL also imposes limits on how far away objects must be from the camera in order to appear in the final picture.

The actual 3D shape of the view volume depends on what kind of projection is used. For orthographic (parallel) projections the view volume is box-shaped, whereas perspective projections have a view volume shaped like a truncated pyramid. The facets encasing the view volume effectively define six **clipping planes**, which partition the frustum interior from the unseen outside world.

9.2.2 Orthographic projection

glOrtho() creates a matrix for an orthographic projection, and post-multiplies the current matrix (which is normally the projection matrix) by it:

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

Figure 9.3 illustrates how the arguments are interpreted. The values define a box-shaped view volume. It is important to set the values such that `left` < `right`, `bottom` < `top` and `near` < `far`. The contents of the view volume are projected onto a rectangular region in the XY plane, with an aspect ratio (`right - left`) / (`top - bottom`).

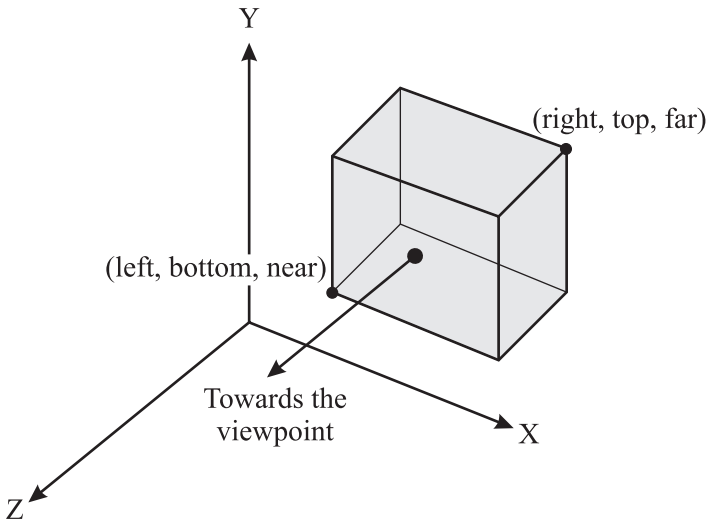


Figure 9.3: The orthographic viewing voulme specified by **glOrtho**.

9.2.3 Perspective projection

gluPerspective() creates a matrix for a perspctive projection, and post-multiplies the current matrix (which will normally be the projection matrix) by it:

```
void gluPerspective ( GLdouble fovy,
                      GLdouble aspect,
                      GLdoublea near,
                      GLdouble far );
```

Figure 9.4 illustrates how the arguments are interpreted. `fovy` is the angle (in degrees) of the

image's vertical field of view; `aspect` is the aspect ratio of the frustum – its width divided by its height; and `near` and `far` respectively specify the positions of the near and far clipping planes, measured as their distances from the **centre of projection** (eyepoint). `near` and `far` must have positive values.

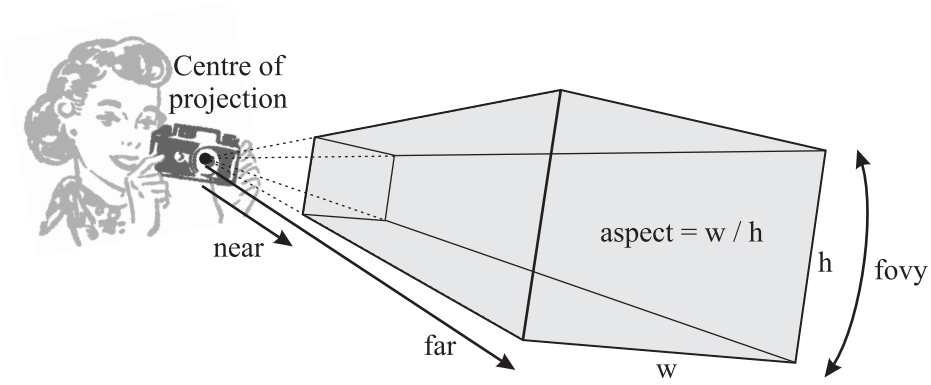


Figure 9.4: The perspective viewing frustum specified by **gluPerspective**.

9.3 Setting the viewport

glViewport() sets the position and size of the **viewport** – the rectangular area in the display window in which the final image is drawn, as shown in Figure 9.5:

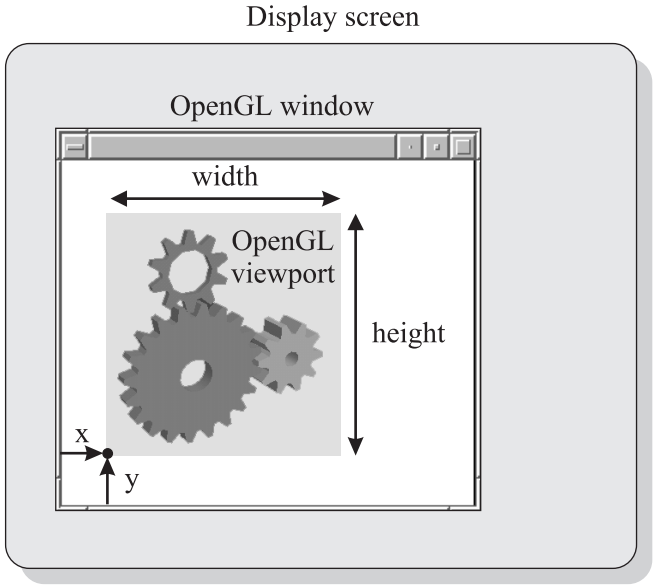


Figure 9.5: How the viewport is defined.

```
void glViewport ( GLint x,
                  GLint y,
                  GLsizei width,
                  GLsizei height );
```

`x` and `y` specify the lower-left corner of the viewport, and `width` and `height` specify its width and height. If a viewport is not set explicitly it defaults to fill the entire OpenGL window. This means that if the window's aspect ratio does not match that defined in **gluPersepctive()** or **glOrtho()** (e.g. after a window resize) the displayed image will appear distorted. **glViewport()** may also be used to draw several separate images within a single OpenGL window.

9.4 Using multiple windows

Most OpenGL programs use a single drawing window. However GLUT does support the use of multiple windows simultaneously. During execution of an OpenGL program, all rendering appears on the **current window**. By default, the current window is always the most recently created window (by **glutCreateWindow()**). If you want to use multiple windows, first create each window and note the window identifier returned by each call to **glutCreateWindow()**. Then, select a window to render using **glutSetWindow()**:

```
void glutSetWindow ( int window );
```

To find out which window is currently selected, call **glutSetWindow()**:

```
int glutGetWindow ( void );
```

You can also destroy windows, using:

```
void glutDestroyWindow ( int window );
```

Obviously, you can't refer to the window identifier for a window which has been destroyed.

9.5 Reversing the viewing pipeline

Sometimes you'll want to click a pixel point in the window and find out what point in your original object coordinates it corresponds to. This is easy to work out – all you have to do is to invert the viewport, projection and modelview transformations as follows:

Given an object coordinate P_o, its corresponding pixel coordinate P_p is given by:

$$P_p = M_{\text{viewport}} \cdot M_{\text{projection}} \cdot M_{\text{modelview}} \cdot P_o$$

So, if we know P_p, we can obtain P_o by applying the **inverse** of each of the transformations:

$$P_o = M^{-1}_{\text{modelview}} \cdot M^{-1}_{\text{projection}} \cdot M^{-1}_{\text{viewport}} \cdot P_p$$

But there's a problem with doing this. Because a screen pixel position is 2D, and our original object coordinates were 3D, every point along a vector in object coordinates can project to the same 2D screen position. This means it isn't possible to perform an **unambiguous** reverse projection from screen to world. So, the application must choose a z value for the pixel too, which lies between the near and far clipping planes.

int **gluUnProject** (GLdouble *winx*,
GLdouble *winy*,
GLdouble *winz*,
const GLdouble *modelMatrix [16]*,
const GLdouble *projMatrix [16]*,
const GLint *viewport [4]*,
GLdouble **objx*,
GLdouble **objy*,
GLdouble **objz*);

gluUnProject() maps the window coordinates `winx`, `winy`, `winz` into object coordinates `objx`, `objy`, `objz`.

The following code, taken from the example program `unproject.c`, shows how **gluUnproject()** is typically used:

```
GLdouble projmatrix[16], mvmatrix[16];
GLint viewport[4];

glGetIntegerv (GL_VIEWPORT, viewport);
glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);
/*  note viewport[3] is height of window in pixels  */
realy = viewport[3] - (GLint) y - 1;
printf ("Coordinates␣at␣cursor␣are␣(%4d,␣%4d)\n", x, realy);
gluUnProject ((GLdouble) x, (GLdouble) realy, 0.0,
              mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
printf ("World␣coords␣at␣z=0.0␣are␣(%f,␣%f,␣%f)\n",
        wx, wy, wz);
gluUnProject ((GLdouble) x, (GLdouble) realy, 1.0,
              mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
printf ("World␣coords␣at␣z=1.0␣are␣(%f,␣%f,␣%f)\n",
         wx, wy, wz);
```

See Section 15.1 for descriptions of the functions **glGetIntegerv()** and **glGetDoublev()**.

Chapter 10

Drawing pixels and images

Sometimes, for image processing applications, you need access pixels directly. Often the most convenient way to do this is to set up a view which gives a one-to-one mapping between object coordinates and pixel coordinates.

10.1 Using object coordinates as pixel coordinates

To do this, you define an orthographic projection, where the width and height of the viewing volume **exactly match** the width and height of the viewport. Normally you will draw on the $z = 0$ plane, so we set the near and far clipping planes to -1.0 and 1.0 respectively.

To begin with, let's assume your program's `main` has created an OpenGLwindow of an appropriate size:

```
glutInitWindowSize (360, 335);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Pixel world");
```

It's usual to place the projection specification in the `reshape` function:

```
void reshape (int width, int height)
{
   glViewport (0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   glOrtho (0.0, (GLfloat) width, 0.0, (GLfloat) height, -1.0, 1.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}
```

then the object coordinate point $(60.0, 40.0, 0.0)$ would map to the OpenGLwindow pixel at $(60, 40)$.

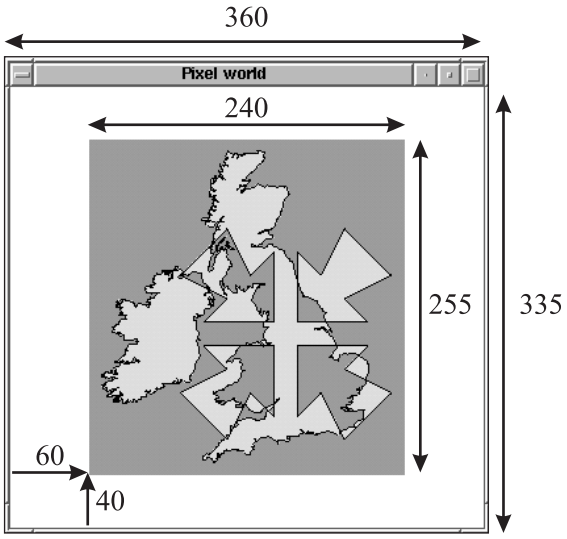


Figure 10.1: The pixel rectangle drawn by **glDrawPixels()** at the current raster position.

10.2 Setting the pixel drawing position

The function **glRasterPos3f()** sets the **current raster position** – the pixel position at which the next pixel rectangle specified using **glDrawPixels()** will be drawn:

```
void glRasterPos3f ( GLfloat x,
                     GLfloat y,
                     GLfloat z );
```

The position (x, y, z) is expressed in object coordinates, and is transformed in the normal way by the modelview and projection matrices.

10.3 Drawing pixels

glDrawPixels draws a rectangle of pixels, with `width` pixels horizontally, and `height` pixels vertically.

```
void glDrawPixels ( GLsizei width,
                    GLsizei height,
                    GLenum format,
                    GLenum type,
                    const GLvoid *pixels );
```

The bottom left-hand corner of the pixel rectangle is positioned at the **current raster position**.

`pixels` is a pointer to an array containing the actual pixel data. Because pixel data can be encoded in several different ways, the type of `pixels` is a (`void *`) pointer. `format` and

`type` specify the pixel data encoding: normally `format` will be `GL_RGB`, which states that each pixel is described by three sequential values giving the red, green and blue componets; `type` specifies the data type used for each of the R, G and B components, and will normally be `GL_FLOAT`, with each of the R, G and B values in the range [0.0,1.0].

For example, used in conjunction with the viewing code in Section 10.1, the following code defines and draws a pixel rectangle of size 240 by 255, positioned at (60,40). The result is shown in Figure 10.1:

```
#define WIDTH  240
#define HEIGHT 255

GLfloat image[WIDTH][HEIGHT][3]; /* pixel data, R,G,B */

/* code omitted to write pixel values into 'image' */

void display (void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glRasterPos3f(60.0, 40.0, 0.0);
  glDrawPixels(WIDTH, HEIGHT, GL_RGB, GL_FLOAT, image);
}
```

Chapter 11

Displaying text

Unlike many graphics systems, OpenGL doesn't directly support the specification and rendering of text. It's left up to the application programmer to draw text using one of two approaches:

- Define the shape of a character as collection of pixels in a bitmap. Here, the shape of a character is not geometric, and so it isn't affected at all by the modelview and projection matrices.
- Draw the shape of each character using OpenGL primitives, most commonly lines. With this approach, each character is a little geometrical object, and can be transformed using the modelview and projection matrices like ordinary OpenGL primitives.

Clearly, both of these methods mean that the application programmer would have to do quite a lot of work to draw text! Fortunately, however, the GLUT library comes to the rescue.

GLUT provides a number of font definitions in both the bitmap and line (also known as 'stroke') forms. Here, we only describe GLUT's bitmap text, which is most commonly used. For details of GLUT's stroke text, see the GLUT manual.

11.1 GLUT's bitmap fonts

GLUT defines three groups of bitmap fonts, based on standard X-windows fonts:

- A fixed-width font, where each character occupies a pixel rectangle of fixed size, either 9×15 or 8×13: `GLUT_BITMAP_9_BY_15`, `GLUT_BITMAP_8_BY_13`;
- A proportionally-spaced Times-Roman font, at 10 or 24 points: `GLUT_BITMAP_TIMES_ROMAN_10`, `GLUT_BITMAP_TIMES_ROMAN_24`;
- A proportionally-spaced Helvetica font, at 10, 12 or 24 points: `GLUT_BITMAP_HELVETICA_10`, `GLUT_BITMAP_HELVETICA_12`, `GLUT_BITMAP_HELVETICA_18`.

You can see these fonts (and a stroke font) demonstrated in the example program `font.c` (You'll need to copy an extra file `tkmap.c` from /opt/info/courses/OpenGL/examples/ into the same folder you have `font.c`). Use the arrow keys to rotate the fonts – you'll see that the stroke font is properly transformed, but the only the start points of the bitmap fonts move.

11.2 Drawing a single character

void **glutBitmapCharacter** (void **font*,
int *char*);

glutBitmapCharacter() draws the single character whose ASCII code is `char`, from font `font`. The position at which the character's bitmap is drawn is the **current raster position**, set by **glRasterPos3f()**.

11.3 Drawing a text string

An application will often wish to display text strings. Here's a simple function to do that (taken from `thegears.c`):

```
void drawString (void *font, float x, float y, char *str) {
/* Draws string 'str' in font 'font', at world (x,y,0) */
   char *ch;
   glRasterPos3f(x, y, 0.0);
   for (ch= str; *ch; ch++)
       glutBitmapCharacter(font, (int)*ch);
}
```

We might call this function as follows, to draw a string at world $(-7.0, 0.0, 0.0)$:

```
glColor3f(1.0, 1.0, 1.0); /* Select white */
drawString (GLUT_BITMAP_HELVETICA_18, -7.0, 0.0, "Press␣Esc␣to␣quit");
```

Note: make sure the Z-coordinate of your bitmap text remains inside the frustum. If it doesn't, OpenGL will clip the entire text out. (Note also that if you're doing lighting (see Chapter 16) – you'll have to ensure lighting is disabled while you're drawing bitmap text.)

Chapter 12

Interaction

The basic OpenGL library has no facilities for interaction – it's only concerned with rendering. This was a design decision made in the interests of efficiency and portability.

The GLUT library provides some rudimentary facilities for creating graphical user interfaces (GUIs). Specifically, the **glutMainLoop()** function traps events, and allows an application to deal with them in three ways:

- **Mouse events** are triggered when a mouse button is pressed, and also when the mouse changes position;
- **Keyboard events** are triggered when the user presses an ASCII key or a cursor movement/function key;
- **Menu events** are triggered when the application has defined GLUT pop-up menus and assigned them to mouse buttons.

GLUT also provides a timing mechanism, which allows the programmer to specify that an arbitrary function is automatically called at a specific time interval.

12.1 Keyboard events

For keyboard events, GLUT calls the application callback function registered by **glutKeyboardFunc()** or **glutSpecialFunc()**:

void **glutKeyboardFunc** (*void (*func)(unsigned char key, int x, int y)*);

glutKeyboardFunc() registers the application function to call when OpenGL detects a key press generating an ASCII character. This can only occur when the mouse focus is inside the OpenGL window.

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

glutSpecialFunc() registers the application callback to call when OpenGL detects a that key

press generating a non-ASCII character has occurred. This can only occur when the mouse focus is inside the OpenGL window. Three values are passed to the callback: `key` is an integer code for the key pressed; `x` and `y` give the pixel position of the mouse. Some useful codes are:

- `GLUT_KEY_LEFT` Left arrow key
- `GLUT_KEY_RIGHT` Right arrow key
- `GLUT_KEY_UP` Up arrow key
- `GLUT_KEY_DOWN` Down arrow key
- `GLUT_KEY_F1` F1 function key (and similarly F2–F12)

12.2 Mouse events

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

glutMouseFunc() registers an application callback function which GLUT will call when the user presses a mouse button within the window. The following values are passed to the callback function:

- `button` records which button was pressed, and can be `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` or `GLUT_RIGHT_BUTTON`.
- `state` records whether the event was generated by pressing the button (`GLUT_DOWN`), or releasing it (`GLUT_UP`).
- `x`, `y` give the current mouse position in pixels. Note: when using OpenGL with X, the mouse y position is measured from the **top** of the screen window.

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

glutMotionFunc() registers an application callback function which GLUT will call when the mouse moves within the window while one of its buttons is pressed. The current mouse position `x`, `y` is passed to the callback function.

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

glutPassiveMotionFunc() has the same job as **glutMotionFunc()**, but no buttons need to be pressed for an event to be generated.

12.3 Controlling the mouse cursor

You can set the position of the cursor using **glutWarpPointer()**:

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

where (x,y) is in pixels relative to the window's origin (top-left).

To change the shape of the mouse cursor, use **glutSetCursor()**:

```
void glutSetCursor ( int cursor );
```

Where `cursor` is one of the following:

```
GLUT_CURSOR_NONE        // Turns the cursor off
GLUT_CURSOR_RIGHT_ARROW // Basic arrows
GLUT_CURSOR_LEFT_ARROW
GLUT_CURSOR_INFO        // Symbolic cursor shapes
GLUT_CURSOR_DESTROY
GLUT_CURSOR_HELP
GLUT_CURSOR_CYCLE
GLUT_CURSOR_SPRAY
GLUT_CURSOR_WAIT
GLUT_CURSOR_TEXT
GLUT_CURSOR_CROSSHAIR
GLUT_CURSOR_UP_DOWN     // Directional cursors
GLUT_CURSOR_LEFT_RIGHT
GLUT_CURSOR_TOP_SIDE    // Sizing cursors
GLUT_CURSOR_BOTTOM_SIDE
GLUT_CURSOR_LEFT_SIDE
GLUT_CURSOR_RIGHT_SIDE
GLUT_CURSOR_TOP_LEFT_CORNER
GLUT_CURSOR_TOP_RIGHT_CORNER
GLUT_CURSOR_BOTTOM_RIGHT_CORNER
GLUT_CURSOR_BOTTOM_LEFT_CORNER
GLUT_CURSOR_INHERIT     // Inherit from parent window
GLUT_CURSOR_FULL_CROSSHAIR // Fullscreen crosshair (if available)
```

12.4 Menu events

GLUT menus are very straightforward to use. Once defined by the application and attached to a specified mouse button, they pop up on the window at the position of the mouse when the appropriate button is pressed. The user then makes a selection from the items in the menu, and GLUT calls an application callback function for the menu, passing as an argument the number of the selected item. GLUT menus can have items which invoke pop-up sub-menus.

The following example program, `menu.c` shows how to create two menus, one attached to the right mouse button, and one to the middle mouse button. Try running the program.

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

```
 2 #include <stdio.h>
 3
 4 void display (void)
 5 { /* Callback called when OpenGL needs to update the display */
 6   glClear (GL_COLOR_BUFFER_BIT); /* Clear the window */
 7 }
 8
 9 void keyboard (unsigned char key, int x, int y)
10 { /* Callback called when a key is pressed */
11   if (key == 27) { exit (0); }       /* 27 is the Escape key */
12 }
13
14 void tobys_bistro (int item)
15 { /* Callback called when the user clicks the right mouse button */
16   printf ("Toby's bistro: you clicked item %d\n", item);
17 }
18
19 void steves_chippy (int item)
20 { /* Callback called when the user clicks the middle mouse button */
21   printf ("Steve's chippy: you clicked item %d\n", item);
22 }
23
24 int main (int argc, char** argv)
25 {
26   glutInit (&argc, argv);         /* Initialise OpenGL */
27   glutCreateWindow ("Menus");     /* Create the window */
28   glutDisplayFunc (display);      /* Register the "display" function */
29   glutKeyboardFunc (keyboard);    /* Register the "keyboard" function */
30
31   glutCreateMenu (tobys_bistro); /* Create the first menu & add items */
32   glutAddMenuEntry ("Petto di Tacchino alla Napoletana", 1);
33   glutAddMenuEntry ("Bruschetta al Pomodoro e Olive", 2);
34   glutAddMenuEntry ("Chianti Classico", 3);
35   glutAttachMenu (GLUT_RIGHT_BUTTON); /* Attach it to the right button */
36
37   glutCreateMenu (steves_chippy); /* Create the second menu & add items */
38   glutAddMenuEntry ("Rissoles", 1);
39   glutAddMenuEntry ("Curry sauce", 2);
40   glutAddMenuEntry ("Vimto", 3);
41   glutAttachMenu (GLUT_MIDDLE_BUTTON); /* Attach it to the middle button */
42
43   glutMainLoop ();                   /* Enter the OpenGL main loop */
44   return 0;
45 }
```

12.4.1 Defining menus

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

glutCreateMenu() creates a new pop-up menu, which becomes the **current menu**. The argument is the name of the application's callback function which is called when an item in the menu is selected by the user. **glutCreateMenu()** allocates the new menu a unique `int`

identifier number, which it returns.

```
void glutAddMenuEntry ( char *name,
                        int value );
```

glutAddMenuEntry() adds new item to the end of the current menu. `name` is the text to display in the item. `value` is the value passed to the application's callback if this item is selected by the user.

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

glutAddSubMenu() adds a new sub-menu to the end of the current menu. `name` is the text to display in the item in the current menu which, when pressed, will display the sub-menu. `menu` is the identifer of the sub-menu, which is created separately with a call to **glutCreateMenu()**.

```
void glutAttachMenu ( int button );
```

glutAttachMenu() attaches the current menu to mouse button `button`. Whenever this button is subsequently pressed, the menu will pop up. `button` must be one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` or `GLUT_RIGHT_BUTTON`.

```
void glutSetMenu ( int menu );
```

glutSetMenu() sets the **current menu** to the menu whose identifier is `menu`.

12.4.2 Changing menus dynamically

It's also possible to change the items in a menu as the program runs.

```
void glutChangeToMenuEntry ( int entry,
                             char * name,
                             int value );
```

glutChangeToMenuEntry() changes an entry in the current menu.

12.5 GLUT timing mechanisms

Sometimes we want to keep track of elapsed time, and trigger things to happen at times we specify. GLUT provides two mechanisms for this.

12.5.1 Time-triggered user callback functions

The first mechanism is to tell GLUT to call user callback functions after specified elapsed times, using **glutTimerFunc()**:

```
void glutTimerFunc ( unsigned int msecs,
                     (*userFunc)(int valuePassed) func,
                     int value );
```

glutTimerFunc() registers the user's callback function `userFunc()`, which GLUT will call after `msecs` milliseconds have elapsed since **glutTimerFunc()** was called. When `userFunc()` gets called, its `valuePassed` argument will be set to `value`. Note that `msecs` is a lower bound; for example, specifying 30ms for `msecs` means that GLUT will not call `userFunc()` until at least 30ms have elapsed. The actual call may take place later, depending on what else GLUT has to do, and how busy the CPU is. You can call **glutTimerFunc()** to register multiple callbacks, and GLUT will process each in turn. Note also that after your callback function has been called, it will be de-registered, so you will need to call **glutTimerFunc()** again. Example program `ex6a.c` demonstrates this, rotating the cube using a timer function, rather than the `idle()` function we used on page 34.

```
 1 #include <GL/glut.h>
 2 #include <stdlib.h>
 3
 4 int interval= 500; // timer interval in ms.
 5
 6 GLfloat angle= 0.0;
 7
 8 void myTimerFunction (int value) {
 9    angle+= 1.0;
10    glutPostRedisplay();
11    /* Now register the "timer" function again */
12    glutTimerFunc(interval, myTimerFunction, 0);
13 }
14
15 void display(void) {
16    glClearColor(0.9,0.9,0.9,0.0); /* Set grey backbround */
17    glClear (GL_COLOR_BUFFER_BIT);
18    glLoadIdentity ();
19    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
20    glRotatef(angle, 1, 0, 0);
21    glRotatef(angle, 0, 1, 0);
22    glRotatef(angle, 0, 0, 1);
23    glColor3f(1.0,0.0,0.0);
24    glutWireCube(2.0);
25    glFlush();            /* Force update of screen */
26 }
27
28 void reshape (int w, int h) {
29    glViewport (0, 0, (GLsizei)w, (GLsizei)h);
30    glMatrixMode (GL_PROJECTION);
31    glLoadIdentity ();
32    gluPerspective (60, (GLfloat) w / (GLfloat) h, 1.0, 100.0);
33    glMatrixMode (GL_MODELVIEW);
34 }
```

```
35
36 void keyboard(unsigned char key, int x, int y) {
37   if (key == 27)  exit (0);                 /* escape key */
38 }
39
40 int main(int argc, char **argv) {
41    glutInit(&argc, argv);
42    glutInitWindowSize (500, 500);
43    glutInitWindowPosition (100, 100);
44    glutCreateWindow ("ex6a: A rotating cube using glutTimerFunc()");
45    glutDisplayFunc(display);
46    glutReshapeFunc(reshape);
47    glutKeyboardFunc(keyboard);
48    /* Register the "timer" function */
49    glutTimerFunc(interval, myTimerFunction, 0);
50    glutMainLoop();
51    return 0;
52 }
```

12.5.2 Querying elapsed time

The second mechanism is for the application to keep track of elapsed time itself. It can query the number of milliseconds elapsed since **glutInit()** was called, using **glutGet()** (Section 15.1.3, page 96) with the argument `GLUT_ELAPSED_TIME`.

```
elapsedSinceStart= glutGet(GLUT_ELAPSED_TIME);
```

12.6 Picking

One of the most common requirements in interactive graphics is for the user to click on an object on the display, and have the object's 'name' returned to the application. This is called **picking**. OpenGL provides a mechanism for picking objects in a 3D scene, and this section will show you how to detect which objects are under the mouse pointer or in a square region of the OpenGL window. The steps involved in detecting which objects are at the location where the mouse was clicked are:

1. Get the window coordinates of the mouse cursor.
2. Enter selection mode.
3. Redefine the viewing volume so that only a small area of the window around the cursor is rendered.
4. Render the scene, either using all primitives or only those relevant to the picking operation.
5. Exit selection mode and identify the objects which were rendered on that small part of the screen.

In order to identify the rendered objects you must name all relevant objects in your scene. The OpenGL API allows you to give names to primitives, or sets of primitives (objects). When in selection mode, a special rendering mode, no objects are actually rendered in the framebuffer. Instead the names of the objects (plus depth information) are collected in an array. For unnamed objects, only depth information is collected.

Using OpenGL terminology, a 'hit' occurs whenever a primitive is rendered in selection mode. Hit records are stored in the selection buffer. Upon exiting the selection mode OpenGL returns the selection buffer with the set of hit records. Since OpenGL provides depth information for each hit the application can then easily detect which object is closer to the user.

12.6.1 Introducing the Name Stack

As the title suggests, the names you assign to objects are stored in a stack. Actually you don't give names to objects, as in a text string. Instead you number objects. Nevertheless, since in OpenGL the term 'name' is used, we will also use the term 'name' instead of 'number'.

When an object is rendered, if it intersects the viewing volume, a hit record is created. The hit record contains the names currently on the name stack plus the minimum and maximum depth for the object. Note that a hit record is created even if the name stack is empty, in which case it only contains depth information. If more objects are rendered before the name stack is altered or the application leaves the selection mode, then the depth values stored on the hit record are altered accordingly.

A hit record is stored on the selection buffer only when the current contents of the name stack are altered or when the application leaves the selection mode.

The rendering function for the selection mode therefore is responsible for the contents of the name stack as well as the rendering of primitives.

OpenGL provides the following functions to manipulate the Name Stack:

void **glInitNames** (void);

This function creates an empty name stack. You are required to call this function to initialize the stack prior to pushing names.

void **glPushName** (GLuint *name*);

Adds `name` to the top of the stack. The stack's maximum depth is implementation-dependent, however it will contain at least 64 names, which should prove to be more than enough for the vast majority of applications. Nevertheless if you want to be sure you may query the state variable `GL_NAME_STACK_DEPTH` (use **glGetIntegerv()**). Pushing values onto the stack beyond its capacity causes an overflow error `GL_STACK_OVERFLOW`.

void **glPopName** (void);

Removes the name from top of the stack. Popping a value from an empty stack causes an

underflow, error GL_STACK_UNDERFLOW.

void **glLoadName** (GLuint *name*);

This function replaces the top of the name stack with name. It is the same as calling

```
glPopName();
glPushName(name);
```

This function is basically a shortcut for the above snippet of code. Loading a name on an empty stack causes the error GL_INVALID_OPERATION.

Note: Calls to the above functions are ignored when not in selection mode. This means that you may have a single rendering function with all the name stack functions inside it. When in the normal rendering mode the functions are ignored and when in selection mode the hit records will be collected. Note: You can't place these functions inside a **glBegin()**–**glEnd()** construction, which is sometimes awkward, since that will require a new rendering function for the selection mode in some cases. For instance if you have a set of points inside a **glBegin()**–**glEnd()** and you want to name them in such a way that you can tell them apart, then you must create one **glBegin()**–**glEnd()** block for each name.

12.6.2 Rendering for the Selection Mode

An example of a rendering function is now presented – picking.c. Try it out. Let's look at part of the code:

```
1 #define BODY 1
2 #define HEAD 2
3
4 void renderInSelectionMode() {
5   glInitNames();
6
7   glPushName(BODY);
8   drawBody();
9   glPopName();
10
11  glPushName(HEAD);
12  drawHead();
13  drawEyes();
14  glPopName();
15
16  drawGround();
17 }
```

Let's go line by line.

5: glInitNames() – This function creates an empty stack. This is required before any other operation on the stack such as Load, Push or Pop.

7: glPushName(BODY) – A name is pushed onto the stack. The stack now contains a single name.

8: drawBody() – A function which calls OpenGL primitives to draw something. If any of the primitives called in here intersects the viewing volume a hit record is created. The

contents of the hit record will be the name currently on the name stack, BODY, plus the minimum and maximum depth values for those primitives that intersect the viewing volume.

9: `glPopName()` – Removes the name of the top of the stack. Since the stack had a single item, it will now be empty. The name stack has been altered so if a hit record was created the name will be saved in the selection buffer.

11: `glPushName(HEAD)` – A name is pushed onto the stack. The stack now contains a single name again. The stack has been altered, but there is no hit record so nothing goes into the selection buffer.

12: `drawHead()` – Another function that renders OpenGL primitives. Again if any of the primitives intersects the viewing volume a hit record is created.

13: `drawEyes()` – Yet another function which renders OpenGL primitives. If any of the primitives in here intersects the viewing volume and a hit record already exists, the hit record is updated accordingly. The names currently in the hit record are kept, but if any of the primitives in `drawEyes()` has a smaller minimum, or a larger maximum depth, then these new values are stored in the hit record. If the primitives in `drawEyes()` do intersect the viewing volume but there was no hit record from drawHead then a new one is created.

14: `glPopName()` – The name on the top of the stack is removed. Since the stack had a single name the stack is now empty. Again the stack has been altered so if a hit record was created it will be stored in the selection buffer

16: `drawGround()` – If any if the primitives called in here intersects the viewing volume a hit record is created. The stack is empty, so no names will be stored in the hit record, only depth information. If no alteration to the name stack occurs after this point, then the hit record created will only be stored in the selection buffer when the application leaves the selection mode. This will be covered later on the tutorial.

Note: lines 9 and 11 could have been replaced by `glLoadName(HEAD)`. Note that you can push a dummy name (some unused value) right at the start, and afterwards just use `glLoadName()`, instead of Pushing and Popping names. However it is faster to disregard objects that don't have a name than to check if the name is the dummy name. On the other hand it is probably faster to call glLoadName than it is to Pop followed by Push.

12.6.3 Using multiple names for an object

There is no rule that says that an object must have a single name. You can give multiple names to an object. Suppose you have a number of snowmen disposed on a grid. Instead of naming them as $1, 2, 3, \ldots$ you could name each of them with the row and column where they are placed. In this case each snowman will have two names describing its position: the row and column of the grid. The following two functions show the two different approaches. First using a single name for each snowman.

```
for(int i = -3; i < 3; i++)
  for(int j = -3; j < 3; j++) {
    glPushMatrix();
    glPushName(i*6+j);
    glTranslatef(i*10.0,0,j * 10.0);
    glCallList(snowManDL);
    glPopName();
```

```
    glPopMatrix();
  }
```

The following function gives two names to each snowman. In this latter function if a hit occurs the hit record will have two names.

```
for(int i = -3; i < 3; i++) {
  glPushName(i);
  for(int j = -3; j < 3; j++) {
    glPushMatrix();
    glPushName(j);
    glTranslatef(i*10.0,0,j * 10.0);
    glCallList(snowManDL);
    glPopName();
    glPopMatrix();
    }
  glPopName();
}
```

12.6.4 Hierarchical Naming

This is a natural extension to multiple naming. It may be of interest to find out where in a particular object you have clicked. For instance you may want to know not only in which snowman you clicked but also if you clicked on the head or the body. The following function provides this information.

```
for(int i = -3; i < 3; i++) {
  glPushName(i);
  for(int j = -3; j < 3; j++) {
    glPushMatrix();
    glPushName(j);
    glTranslatef(i*10.0,0,j * 10.0);
    glPushName(HEAD);
    glCallList(snowManHeadDL);
    glLoadName(BODY);
    glCallList(snowManBodyDL);
    glPopName();
    glPopName();
    glPopMatrix();
    }
  glPopName();
}
```

In this case you'll have three names when you click on either the body or head of a snowman: the row, the column and the value BODY or HEAD respectively.

12.6.5 A Note about Rendering in Selection Mode

As mentioned before, all calls to stack functions are ignored when not in selection mode – that is, when rendering to the frame buffer. This means that you can have a single rendering function for both modes. However this can result in a serious waste of time. An application may have only a small number of pickable objects. Using the same function for both

modes will require to render a lot of non-pickable objects when in selection mode. Not only the rendering time will be longer than required as will the time required to process the hit records.

Therefore it makes sense to consider writing a special rendering function for the selection mode in these cases. However you should take extra care when doing so. Consider for instance an application where you have several rooms, each room with a potential set of pickable objects. You must prevent the user from selecting objects through walls, i.e. objects that are in other rooms and are not visible. If you use the same rendering function then the walls will cause a hit, and therefore you can use depth information to disregard objects that are behind the wall. However if you decide to build a function just with the pickable objects then there is no way to tell if an object is in the current room, or behind a wall.

Therefore, when building a special rendering function for the selection mode you should include not only the pickable objects, but also all the objects that can cause occlusions, such as walls. For highly interactive real time applications it is probably a good option to build simplified representations of the objects that may cause occlusions. For instance a single polygon may replace a complex wall, or a box may replace a table.

So far the OpenGL naming scheme has been presented. This section will show you how to enter the selection mode for picking purposes. The first step is to tell OpenGL where to store the hit records. This is accomplished with the following function:

void **glSelectBuffer** (GLsizei *size*,
GLuint **buffer*);

`buffer` is an array of unsigned integers. This array is where OpenGL will store the hit records. `size` is the size of the array. You are required to call this function before entering the selection mode. Next you enter the selection mode with a call to:

void **glRenderMode** (GLenum *mode*);

`mode` – use `GL_SELECT` to enter the rendering mode and `GL_RENDER` to return to normal rendering. This later value is the default.

Now comes the tricky part. The application must redefine the viewing volume so that it renders only a small area around the place where the mouse was clicked.

In order to do that it is necessary to set the matrix mode to `GL_PROJECTION`. Afterwards, the application should push the current matrix to save the normal rendering mode settings. Next initialise the matrix, as follows:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
```

We now have a clean projection matrix. All that is required now is to define the viewing volume so that rendering is done only in a small area around the cursor. This can be accom-

plished using the following function:

```
void gluPickMatrix ( GLdouble x,
                     GLdouble y,
                     GLdouble width,
                     GLdouble height,
                     GLint viewport[4] );
```

`x`, `y` – these two values represent the cursor location. They define the centre of the picking area. This value is specified in window coordinates. However note that window coordinates in OpenGL have the origin at the bottom left corner of the viewport, whereas for the Operating system is the upper left corner. `width`, `height` – the size of the picking region, too small and you may be hard pressed to pick small objects, too large and you may end up with too many hits. `viewport` – the current viewport.

Before you call the above function you have to get the current viewport. This can be done querying OpenGL for the state variable `GL_VIEWPORT` (use the function **glGetIntegerv()**). Then you call **gluPickMatrix()**, and finally set your projection (perspective or orthogonal) just as you did for the normal rendering mode. Finally get back to the modelview matrix mode, and initialise the Name Stack to start rendering. The following code shows the sequence of function calls to do just that using a perspective projection.

```
glGetIntegerv(GL_VIEWPORT,viewport);
gluPickMatrix(cursorX,viewport[3]-cursorY, 5,5,viewport);
gluPerspective(45,ratio,0.1,1000); glMatrixMode(GL_MODELVIEW);
glInitNames();
```

Note the second parameter of **gluPickMatrix()**. As mentioned before, OpenGL has a different origin for its window coordinates than the operation system. The second parameter provides for the conversion between the two systems, i.e. it transforms the origin from the upper left corner, as provided by GLUT for example, into the bottom left corner.

The picking region in this case is a 5x5 window. You may find that it is not appropriate for your application. Do some tests to find an appropriate value if you're finding it hard to pick the right objects. The following function does all operations required to enter the selection mode and start picking, assuming that `cursorX` and `cursorY` are the operating system window coordinates for the location of the mouse click.

```
#define BUFSIZE 512
GLuint selectBuf[BUFSIZE]

...

void startPicking(int cursorX, int cursorY) {

  GLint viewport[4];

  glSelectBuffer(BUFSIZE,selectBuf);
  glRenderMode(GL_SELECT);

  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glLoadIdentity();
```

```
  glGetIntegerv(GL_VIEWPORT,viewport);
  gluPickMatrix(cursorX,viewport[3]-cursorY,
  5,5,viewport);
  gluPerspective(45,ratio,0.1,1000);
  glMatrixMode(GL_MODELVIEW);
  glInitNames();
}
```

You may copy and paste this function to your application, with the appropriate modifications regarding your projection. If you do that then just call this function when entering the rendering mode and before rendering any graphic primitives.

In order to process the hit records the application must first return to the normal rendering mode. This is done calling **glRenderMode()** with `GL_RENDER`. This function returns the number of hit records that were created during rendering in the selection mode. After this step the application can process the selection buffer. Note that before calling glRender with `GL_RENDER` there is no guarantee that the hit records have been saved into the selection buffer.

Furthermore, it is necessary to restore the original projection matrix. Since this matrix was saved when entering the selection mode with glPushMatrix, all that is required is to pop the matrix. The following excerpt of code shows the required steps:

```
void stopPicking() {
  int hits;

  // restoring the original projection matrix
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glMatrixMode(GL_MODELVIEW);
  glFlush();

  // returning to normal rendering mode
  hits = glRenderMode(GL_RENDER);

  // if there are hits process them
  if (hits != 0)
  processHits(hits,selectBuf);
}
```

So the final issue is related to the selection buffer structure. The selection buffer stores the hit records sequentially by the order they occurred, i.e. by the order that the primitives were drawn. Note that primitives that wouldn't be drawn due to Z buffer depth culling still produce hit records. The hit records are potentially variable size records due to the number of names they contain.

- The first field of the hit record is the number of names it contains.
- The second and third fields represent the minimum and maximum depth of the hit. Note that only the vertices of the primitives that intersect the viewing volume are taken into account. For polygons that get clipped OpenGL reconstructs the vertices. So basically you get the maximum and minimum depths of the rendered segment of the primitives that intersects the viewing volume, not the minimum and maximum

depths of the entire primitive. The depth is taken from the Z buffer (where it lies in the range $[0, 1]$), it gets multiplied by $2^{32} - 1$ and it is rounded to the nearest integer. Note that the depths you get are not linearly proportional to the distance to the viewpoint due to the nonlinear nature of the z buffer.

- The following fields are the sequence of names recorded for the hit. These names are the contents of the name stack when the hit was recorded. Note that since the number of names can be zero this sequence can be the empty sequence.

An example of a selection buffer with 3 hit records is presented next:

Hit Record Contents	Description
0	No names have been stored for the first hit
4.2822e+009	Minimum depth for first hit
4.28436e+009	Maximum depth for first hit
1	Number of names for the second hit
4.2732e+009	Minimum depth for second hit
4.27334e+009	Maximum depth for second hit
6	A single name for the second hit
2	Number of names for the third hit
4.27138e+009	Minimum depth for third hit
4.27155e+009	Maximum depth for third hit
2	First name for third hit
5	Second name for third hit

In order to detect which object was closest to the viewpoint you use the depth information. For instance you can select the object with the smalest minimum depth has the one which the user intended to click on. In the above example the relevant hit is the third one. The following function, adapted from the one in the OpenGL Red Book (`http://www.glprogramming.com/red/`), prints out the names for the closest object.

```
void processHits2 (GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint names, *ptr, minZ,*ptrNames, numberOfNames;

   printf (''hits = %d\n'', hits);
   ptr = (GLuint *) buffer;
   minZ = 0xffffffff;
   for (i = 0; i < hits; i++) {
      names = *ptr;
        ptr++;
          if (*ptr < minZ) {
              numberOfNames = names;
                minZ = *ptr;
                  ptrNames = ptr+2;
                    }

            ptr += names+2;
            }
  printf (''The closest hit names are ``);
```

```
  ptr = ptrNames;
  for (j = 0; j < numberOfNames; j++,ptr++) {
     printf (''%d '', *ptr);
  }
  printf (''\n'');
}
```

Chapter 13

Colour

We express colour using a **colour model**, which gives us a way of assigning numerical values to colours. A common simple model is the Red-Green-Blue (RGB) model, where a colour is represented by a mixture of the three primary colours red, green and blue. This is illustrated in Figure 13.1.

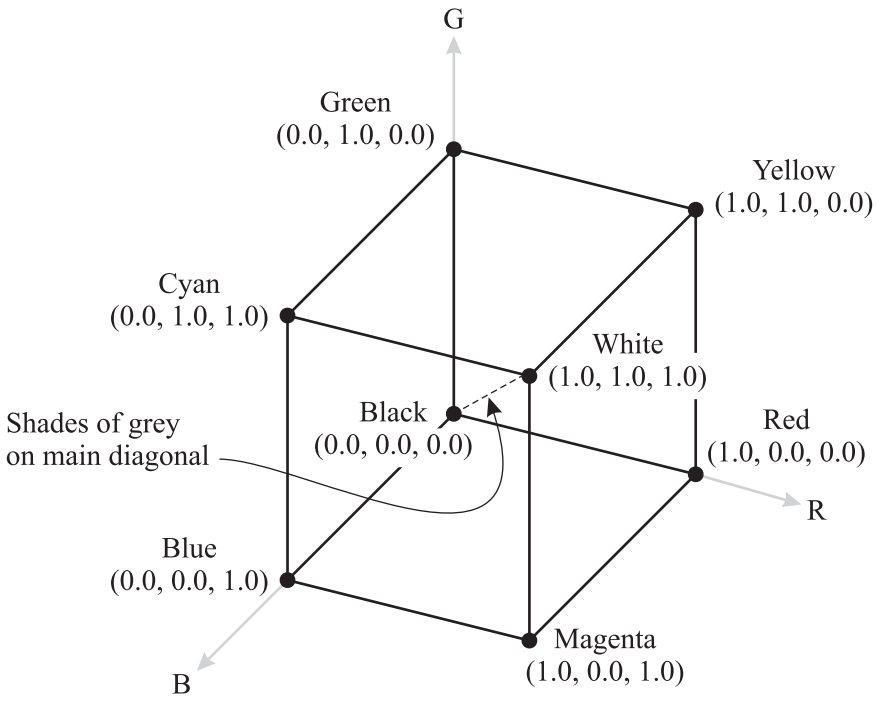


Figure 13.1: The RGB colour model.

13.1 RGB colour in OpenGL

OpenGL supports the RGB colour model, in a slightly extended form. OpenGL adds a fourth component to the colour, called **alpha**, and the revised model is called the **RGBA** model.

Alpha represents the opacity (or, equivalently the transparency) of a colour, and is used when blending colours together. We don't discuss the use of alpha further in this manual.

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

glClearColor() sets the current clearing colour to be used when clearing a buffer using **glClear()**. `red`, `green` and `blue` are the RGB components of the colour. The `GLclampf` datatype limits these values to floats in the range $[0.0, 1.0]$. Set `alpha` to 0.0.

```
void glColor3f ( GLclampf red,
                 GLclampf green,
                 GLclampf blue );
```

glColor3f() sets the current drawing colour, using a triple of RGB values in the range $[0.0, 1.0]$.

Chapter 14

Retained data

Graphics systems typically act in two ways:

- **Immediate mode**: whenever an application **defines** a primitive, it is **drawn** immediately.
- **Retained mode**: the actions of **defining** a primitive and **drawing** a primitive are treated quite separately. When an application defines a primitive, the graphics system keeps a record of the definition as a **display list**, but the primitive isn't drawn. Subsequently, the application requests that the stored primitive be drawn.

OpenGL provides both ways of working.

14.1 Immediate mode vs retained mode

By default, OpenGL works in **immediate mode**: whenever a primitive is defined, OpenGL draws it immediately. Once it has been drawn and rendered as pixels, OpenGL forgets all about the original primitive.

For example, if we execute the code:

```
glBegin(GL_TRIANGLES)
   glVertex(1.0, 3.0, 0.0);
   glVertex(5.0, 3.0, 0.0);
   glVertex(3.0, 4.0, 0.0);
glEnd();
```

OpenGL will draw the triangle defined by the three vertices, once they have been transformed by the graphics pipeline, as pixels in the display buffer. But OpenGL does not keep any internal record of the original definition of the vertices in object coordinates.

The use of immediate mode has a very important consequence for the application programmer: to ensure that the contents of display are up-to-date, the application must execute all the code that defines primitives (including setting transformations and rendering parameters).

There's another way to use OpenGL, called **retained mode**, which is quite different from **immediate mode**.

14.2 Retained mode

Here, the graphical shapes which are to be drawn are specified within a display list. The OpenGL display list mechanism is best thought of as a **cache** for graphics. It isn't a full-fledged data structure which the application can manipulate. Once created, a display list:

- cannot be edited – its data is execute-only.
- cannot be queried – an application cannot 'read back' the data stored in a display list.

If display lists sound very restricted in their functionality, that's exactly the intention. The OpenGL display list is designed for efficiency, not versatility.

14.3 Using display lists

glNewList() creates and opens a new display list named `list`:

```
void glNewList ( GLuint list,
                 GLenum mode );
```

All subsequent OpenGL commands will be stored in the display list. The `list` argument is a positive integer which identifies the display list being created. It is up to the programmer to allocate unique `list` values for each display list. `mode` determines what happens while the list is being created. There are two options:

- `GL_COMPILE`: the commands are not executed as they are stored in the display list. This means the contents of the display list will not be drawn until the display list is called using **glCallList()**.
- `GL_COMPILE_AND_EXECUTE`: the commands are executed as soon as they are stored in the display list.

Only one display list can be open for writing at a time: OpenGL will report an error if another display list is already open. If a display list with the name `list` already exists when, **glNewList()** is called, OpenGL automatically empties the existing display list, and overwrites it with the new definition.

```
void glEndList ( void );
```

glEndList() closes the currently open display list, marking the end of its definition.

The following example creates a simple display list which draws a green triangle:

```
GLuint TRI= 1;

glNewList (TRI, GL_COMPILE);
   glBegin (GL_TRIANGLES);
   glColor3f (0.0, 1.0, 0.0);  /* Green */
   glVertex3f (1.0, 3.0, 0.0);
   glVertex3f (5.0, 3.0, 0.0);
   glVertex3f (3.0, 4.0, 0.0);
   glEnd ();
glEndList ();
```

Note that here we've used a symbolic constant `TRI` for the name of the display list, rather than the raw number `1`. This helps readability.

Once a display list has been created, it can be instanced repeatedly, using **glCallList()**:

void **glCallList** (GLuint *list*);

The effect of calling **glCallList()** on a display list named `list` is to execute again all the OpenGL commands stored in the list. Any drawing specified by the commands will happen, as will any changes to the OpenGL context – such as matrices and attributes.

Here's how we could instance the `TRI` display list:

```
for (i= 0; i < 5; i++) {    /* Instance a display list */
   glTranslatef (0.1, 0.0, 0.0);
   glCallList (TRI);
}
```

This code will display five instances of `TRI`, each instance shifted in x by 0.1 units.

14.4 Mixing immediate mode with retained mode

It's perfectly acceptable to use immediate mode and retained mode at the same time. Consider the following code fragment:

```
glColor3f (1.0, 0.0, 0.0);  /* Red */

glBegin (GL_LINES);         /* Draw a line (immediate mode) */
   glVertex3f (0.0, 0.0, 0.0);
   glVertex3f (0.2, 0.5, 0.0);
glEnd ();

for (i= 0; i < 5; i++) {    /* Instance a display list */
   glTranslatef (0.1, 0.0, 0.0);
   glCallList (TRI);
}

glBegin (GL_LINES);         /* Draw another line (immediate mode) */
   glVertex3f (0.0, 0.0, 0.0);
   glVertex3f (0.5, 0.5, 0.0);
glEnd ();
```

Here, we first set the current colour to red, then draw a line in immediate mode. Next, as in the previous example, we draw five instances of the green triangle – and note how

the **glColor3f()** call (green) stored in the `TRI` display list overwrites the effect of the colour currently in effect when the display list is called (red). Finally, we draw another line in immediate mode. **Question:** what colour will the second line be? **Answer:** green, because green was the most recent colour selected (when the `TRI` display list was called).

Chapter 15

State

OpenGL is a **state machine**: calling OpenGL functions change the state of the machine. This is a fancy way of saying that inside the OpenGL system are a bunch of global variables which the application can set and query. The current values of these variables control the way OpenGL behaves.

For example, calling **glColor3f()** sets the **current drawing colour**, which will be used for drawing primitives.

15.1 State enquiries

Both OpenGL and GLUT keep 'state' – a set of variables which describe the current configuration.

An application can query the values of state variables using a simple 'keyword and value' model.

15.1.1 OpenGL state

For basic OpenGL state, there's a separate enquiry function for each **type** of state variable. For example, for integers:

int **glGetIntegerv** (GLenum *pname*,
GLint **params*);

glGetIntegerv() enquires the integer state variable specified by `pname`. For example:

```
GLint col[1];

glGetIntegerv(GL_CURRENT_COLOR, col);
```

The value of the current drawing colour will be returned in `col`. Note that this is an **array** variable.

Similarly, the function **glGetDoublev()** enquires the current value of a `GLdouble` state variable:

int **glGetDoublev** (GLenum *pname*,
GLdouble **params*);

There are similar functions for enquiring other types of state variable. And there are a huge number of state variables – see Appendix B of the Red Book for a complete list.

15.1.2 Enquiring the OpenGL viewing state

Sometimes it's necessary to enquire the current values of the viewing state. This can be done as follows:

```
GLdouble projection[16], modelview[16];
GLint viewport[4];

glGetIntegerv (GL_VIEWPORT, viewport);
glGetDoublev (GL_MODELVIEW_MATRIX, modelview);
glGetDoublev (GL_PROJECTION_MATRIX, projection);
```

15.1.3 Enquiring GLUT state

There's a separate function for enqiring GLUT state:

int **glutGet** (GLenum *stateVariable*);

All the different GLUT state variables are listed at `http://www.opengl.org/documentation/specs/glut/spec3/node70.html`. Here are a few examples:

```
currentWindowWidth= glutGet(GLUT_WINDOW_WIDTH);
elapsedTime= glutGet(GLUT_ELAPSED_TIME);
currentCursorType= glutGet(GLUT_WINDOW_CURSOR);
```

Chapter 16

Lighting

This chapter describes the facilities OpenGL provides for lighting and shading 3D scenes, so that the objects in them look solid and (somewhat) realistic. Our intention here is to provide enough background and details to get you going, creating lit and shaded scenes. As always, for further information refer to the Red Book, in particular Chapter 5.

16.1 The OpenGL lighting model

OpenGL provides a 'local' lighting model, which computes the illumination of a single polygon with respect to one or more light sources. It needs the following information in order to do this:

- The position of each light source
- The colour of each light source
- How the intensity of each light source decreases with distance
- The type of each light source: ambient, diffuse or specular
- The geometry of the polygon
- The colour of the polygon

In the real world, the interaction between light and matter is incredibly complicated. Much research has been undertaken into techniques for simulating these interactions using mathematics and computer graphics, and there are several sophisticated techniques which can create very realistic images. Two of the best known are ray tracing and radiosity. These are called 'global' models, because they consider not only the interactions of one light and one object, but also the interactions between all lights and all objects in the scene. Such interactions are responsible, for example, for reflections and shadows.

The basic OpenGL lighting model cannot compute reflections and shadows, because it considers each polygon in isolation from all others. This may come as a surprise, since many

OpenGL-based games clearly display sophisticated illumination. In many cases they do this by computing their own illumination, and rendering it using OpenGL textures.

Nevertheless, OpenGL's lighting model is useful, robust, and fast.

16.2 Hidden surface removal

If 3D scenes are to look plausible, we need to worry about 'hidden surface removal'. If we have a scene containing several objects, and we view the scene from a certain position, one object might obscure another. In order to display such a situation realistically, we must ensure that obscured parts of objects do not get drawn. By default, OpenGL simply draws objects in the order specified by the programmer, taking no account of whether one object would obscure another for a given viewpoint.

OpenGL implements hidden-surface removal using a simple technique called depth-buffering (also known as Z-buffering). This takes place during rasterization, using a 'depth buffer' – an array which records a depth value corresponding to each pixel in the window. Initially, each depth value is set to be a very large number. Whenever a new pixel is generated, for example during the scan-conversion of a polygon P_1, the pixel's Z value is compared with the corresponding value in the depth-buffer. If the pixel's depth is less than that in the buffer, the pixel is drawn and its depth recorded in the depth buffer, over-writing the previous value. Otherwise, the pixel is not drawn and the depth buffer is not updated.

Now, suppose subsequently that during the scan-conversion of polygon P_2, the same pixel is generated, because P_1 and P_2 overlap in the scene. If the depth value of P_2's pixel is greater than that stored in the depth buffer, then P_2 is further away from the eye than P_1, and so is obscured by P_1.

To tell OpenGL to perform hidden-surface removal using a depth buffer, you need to do three things.

First, in the call to **glutInitDisplayMode()**, instruct GLUT to create a depth buffer, specifying `GLUT_DEPTH` in addition to any other flags you're using:

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

Second, enable the depth test, which is switched off by default, using **glEnable()**:

```
glEnable (GL_DEPTH_TEST);
```

Finally, you need to explictly clear the depth buffer (in other words, re-load it with large depth values) each time around the rendering loop:

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/ * all your display code */
```

Here's `ex8.c`, which draws a tumbling green cube orbiting a stationary red cube.

```
1 #include <GL/glut.h>
2
3 float r= 0.0;
4 int hidden= 0;
5
6 void init(void) {
```

```
 7    glClearColor (0.0, 0.0, 0.0, 0.0);
 8 }
 9
10 void spin (void) {
11    r+= 0.5;
12    glutPostRedisplay();
13 }
14
15 void display(void) {
16    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
17    if (hidden) glEnable(GL_DEPTH_TEST);
18    else glDisable(GL_DEPTH_TEST);
19    glLoadIdentity ();
20    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
21
22    glColor3f (1.0, 0.0, 0.0);
23    glutSolidCube(1.0);          /* Red cube */
24    glRotatef(r*2.0, 0, 1, 0);   /* Orbit angle */
25    glTranslatef(0.0, 0.0, 1.0); /* Orbit radius */
26    glRotatef(r, 1, 0, 0);       /* Tumble in x,y,z */
27    glRotatef(r, 0, 1, 0);
28    glRotatef(r, 0, 0, 1);
29    glColor3f (0.0, 1.0, 0.0);
30    glutSolidCube(0.5);          /* Green cube */
31    glutSwapBuffers();
32 }
33
34 void reshape (int w, int h) {
35    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
36    glMatrixMode (GL_PROJECTION);
37    glLoadIdentity ();
38    gluPerspective (60, (GLfloat) w / (GLfloat) h, 1.0, 100.0);
39    glMatrixMode (GL_MODELVIEW);
40 }
41  void keyboard(unsigned char key, int x, int y) {
42   if (key == 27) { exit (0); }                 /* escape key */
43   if (key == 'h')
44      hidden= !hidden;
45 }
46
47 int main(int argc, char **argv) {
48    glutInit(&argc, argv);
49    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
50    glutInitWindowSize (500, 500);
51    glutInitWindowPosition (100, 100);
52    glutCreateWindow ("ex8:␣Press␣'h'␣to␣toggle␣hidden␣surface␣removal.");
53    init ();
54    glutDisplayFunc (display);
55    glutReshapeFunc (reshape);
56    glutKeyboardFunc (keyboard);
57    glutIdleFunc (spin);
58    glutMainLoop ();
59    return 0;
60 }
```

By default, hidden-surface removal is off, so the cubes are drawn in the order they're coded in `display()` – that's the stationary red cube first, then the rotating green cube. Where they

overlap, the green cube's pixels will always over-write the red cube's pixels.

Press 'h' to switch on hidden-surface removal, and you can now see when the green cube orbits **behind** the red cube, and is therefore obscured by it.

Notice that the green cube is drawn in a single colour – and doesn't look at all 'solid'. We'll see how to address that in subsequent sections.

16.3 Defining lights

By default, lighting is off. It's enabled as follows:

```
glEnable (GL_LIGHTING);
```

OpenGL provides at least eight lights, named `GL_LIGHT0` through `GL_LIGHT7`. By default, each light is switched off, so a light must be enabled if it is to have any effect. For example, to use light 0:

```
glEnable (GL_LIGHT0);
```

As well as enabling a light, you need to set its position, colour and other attributes, but if you don't, the light has handy default values. In particular, its colour is white, and its located at the position $(0, 0, 1)$. We can use these defaults to add a light to the previous example, which we altered slightly so that the orbiting object is now a sphere. Here's `ex9.c`:

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

float r= 0.0;
int hidden= 1, flat= 1;

void init(void) {
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glEnable (GL_LIGHTING);
   glEnable (GL_LIGHT0);
}

void spin (void) {
   r+= 1.0;
   glutPostRedisplay();
}

void display(void) {
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   if (hidden) glEnable(GL_DEPTH_TEST);
   else glDisable(GL_DEPTH_TEST);
   if (flat) glShadeModel (GL_FLAT);
   else glShadeModel (GL_SMOOTH);
   glLoadIdentity ();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   glColor3f (1.0, 0.0, 0.0);
   glutSolidCube(1.0);          /* Red cube */
   glRotatef(r*2.0, 0, 1, 0);   /* Orbit angle */
   glTranslatef(0.0, 0.0, 1.0); /* Orbit radius */
   glRotatef(r, 1, 0, 0);       /* Tumble in x,y,z */
```

```
31    glRotatef(r, 0, 1, 0);
32    glRotatef(r, 0, 0, 1);
33    glColor3f (0.0, 1.0, 0.0);
34    glutSolidSphere(0.5, 20, 15); /* Green sphere */
35    glutSwapBuffers();
36 }
37
38 void reshape (int w, int h) {
39    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
40    glMatrixMode (GL_PROJECTION);
41    glLoadIdentity ();
42    gluPerspective (60, (GLfloat) w / (GLfloat) h, 1.0, 100.0);
43    glMatrixMode (GL_MODELVIEW);
44 }
45  void keyboard(unsigned char key, int x, int y) {
46   if (key == 27) { exit (0); } /* escape key */
47   if (key == 'h') hidden= !hidden;
48   if (key == 's') flat= !flat;
49 }
50
51 int main(int argc, char **argv) {
52    glutInit(&argc, argv);
53    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
54    glutInitWindowSize (500, 500);
55    glutInitWindowPosition (100, 100);
56    glutCreateWindow ("ex9");
57    init ();
58    glutDisplayFunc (display);
59    glutReshapeFunc (reshape);
60    glutKeyboardFunc (keyboard);
61    glutIdleFunc (spin);
62    glutMainLoop ();
63    return 0;
64 }
```

Running `ex9`, you'll see the orbiting sphere lit by `LIGHT0`, and now its different faces are shaded according to how they're oriented with respect to the light source.

Note that you can still use 'h' to toggle hidden surface removal. Try toggling it and observe the incorrect results when it's switched off. For lighting to work correctly, the depth buffer must be **enabled**.

16.4 Defining the shading model

In `ex9` we make use of a new function, **glShadeModel()**:

void **glShadeModel** (GLenum *mode*);

glShadeModel() specifies how OpenGL renders primitives. For example, when rendering a polygon, if `mode` is `GL_FLAT`, OpenGL chooses one vertex of the polygon, computes its colour, and assigns this colour to all pixels in the polygon. If `mode` is `GL_SMOOTH`, OpenGL computes a colour for each vertex, and the interior pixels of the polygon are coloured by interpolating between the vertex colours. If **glShadeModel()** is not called, the default be-

haviour is GL_SMOOTH.

In ex9, we initially select GL_FLAT, and the 's' key toggles between this and GL_SMOOTH.

16.5 Defining materials

There's another difference between the visual appearances of ex8 and ex9. In ex8, we use **glColor3f()** to set the colour of the two objects, so they're drawn red and green. In ex9, these colour settings are still there, but now **they don't have any effect;** instead, the objects take their colour only from the light source. When lighting is enabled, we need to specify the intrinsic colours of objects in a more sophisticated way. (For a helpful description of how colours are computed, see http://www.sjbaker.org/steve/omniv/opengl_lighting.html.)

OpenGL uses the concept of **material properties**. An object is considered to have a surface made of a material with several properties, which determine how it interacts with light. These are:

- ambient colour: how well the material reflects ambient light
- diffuse colour: how well the material reflects diffuse light
- specular colour: how well the material reflects specular light
- emissiveness: whether the material emits light itself
- shininess: how glossy the material is

There's a single function for setting material properties:

```
void glMaterialfv ( GLenum face,
                    GLenum paramName,
                    TYPE *param );
```

face specifies which face of a primitive the material property should effect, and may be GL_FRONT, GL_BACK or GL_FRONT_AND_BACK.

paramName selects which material property to change, as follows:

- GL_AMBIENT sets the ambient colour (default is (0.2, 0.2, 0.2, 1.0));
- GL_DIFFUSE sets diffuse colour (default is (0.8, 0.8, 0.8, 1.0));
- GL_SPECULAR sets the specular colour (default is (0.0, 0.0, 0.0, 1.0));
- GL_EMISSION sets the emissive colour (default (0.0, 0.0, 0.0, 1.0));
- GL_SHININESS sets the specular exponent (default 0.0);

`param` is the value to set. For `GL_SHININESS`, the type of this argument is `GLfloat`; for all other values of `mode`, it's `GLfloat *`.

Example program `ex10.c` illustrates the use of **glMaterialfv()**, to set the diffuse colours of the cube and sphere. The following extract shows the relevant code:

```
GLfloat redDiffuseMaterial[] = {1.0, 0.0, 0.0, 0.0};
GLfloat greenDiffuseMaterial[] = {0.0, 1.0, 0.0, 0.0};

/* code omitted */

glMaterialfv(GL_FRONT, GL_DIFFUSE, redDiffuseMaterial);
glutSolidCube(1.0);           /* Red cube */

/* code omitted */

glMaterialfv(GL_FRONT, GL_DIFFUSE, greenDiffuseMaterial);
glutSolidSphere(0.5, 20, 15); /* Green sphere */
```

16.6 Defining lights

The properties of a light are defined in a similar way to those of materials, using **glLightfv()**:

void **glLightfv** (GLenum *light*,
GLenum *paramName*,
TYPE **param*);

`paramName` selects which light property to change, as follows:

- `GL_AMBIENT` sets the ambient light colour (default (0.0, 0.0, 0.0, 1.0))
- `GL_DIFFUSE` sets the diffuse light colour (default (1.0, 1.0, 1.0, 1.0))
- `GL_SPECULAR` sets the specular light colour (default (1.0, 1.0, 1.0, 1.0))
- `GL_POSITION` sets the position of the light (default (0.0, 0.0, 1.0, 0.0)). If the w coordinate of the position is 0.0, the light is considered to be at $+\infty$, and the (x, y, z) components of its position give the direction the light shines in.
- `GL_SPOT_DIRECTION` sets the direction of a spotlight (default (0.0, 0.0, -1.0))
- `GL_SPOT_EXPONENT` sets the exponent of a spotlight (default 0.0)
- `GL_SPOT_CUTOFF` sets the spotlight cutoff angle. The default value is 180.0, which indicates the light is not a spotlight. Any other value indicates the light is a spotlight.
- `GL_CONSTANT_ATTENUATION` sets the constant attenuation factor (default 1.0)
- `GL_LINEAR_ATTENUATION` sets the linear attenuation factor (default 0.0)

16.7 The lighting equation

OpenGL uses the following lighting equation to compute the colour V of a vertex:

$$V = M_e + (I_g * M_a) + \sum_{i=0}^{n-1} A_i * S_i * (\text{amb}_i + \text{diff}_i + \text{spec}_i)$$

where:

- M_e is the material's emission
- I_g is the scaled global ambient light
- M_a is the material's ambient reflectivity
- A_i is the attentuation factor for light i: $\frac{1}{k_{c_i}+k_{l_i}d_i+k_{q_i}d_i^2}$
- where k_{c_i} is the value of `GL_CONSTANT_ATTENUATION`
- k_{l_i} is `GL_LINEAR_ATTENUATION`
- k_{q_i} is `GL_QUADRATIC_ATTENUATION`
- d_i is the distance from light source i to the vertex.
- S_i is the spotlight effect of light i. If the light isn't a spotlight, $S_i = 1.0$; if the light is a spotlight, but V is outside the light's cone of illumination, $S_i = 0.0$; otherwise, $S_i = \max(\hat{v}_i \cdot d_i, 0)^{n_s}$, where $\hat{v}$ is the normalised vector from the position of light i to vertex V, d_i is the direction of light i, and n_s is GL_SPOT_EXPONENT.
- amb_i is the ambient reflection component: $I_{a_i} * M_a$
- diff_i is the diffuse reflection component: $\max(\hat{L}_i \cdot \hat{N}, 0) * I_{d_i} * M_d$
- spec_i is the specular reflection component: $\max(\hat{S}_i \cdot \hat{N}, 0)^n * I_{s_i} * M_s)$
- M_d is the material's diffuse reflectivity
- M_s is the material's specular reflectivity
- I_{a_i} is the ambient component of light i
- I_{d_i} is the diffuse component of light i
- I_{s_i} is the specular component of light i
- $\hat{L}_i$ is the normalised vector from V to the position of light source i
- $\hat{N}$ is the unit normal vector for V
- $\hat{S}_i$ is the normalised vector sum of $\hat{L}_i$ and the normalised vector pointing from V to the viewpoint.

Chapter 17

Textures

In this chapter we take an introductory look at how to add textures to polygons. There is a lot to this topic, and here we cover the basics only, and give an example program to get you started.

A 2D texture in OpenGL is a 2D array of colours, and each element in the array is called a 'texel'. We tell OpenGL how to take texel colours fron the array during rendering, and use them to influence the pixel colours.

17.1 Creating textures

How can we define a texture? There are two ways: first, we can hard-code the texel values into the texture array – a tedious process which is only really suitable for regular patterns; second, we can use an existing image (a JPG, PNG, GIF, BMP, etc), and read its pixels into the texture array. This is by far the most common way of define textures, but OpenGL doesn't provide any functions at all for decoding and reading image files – the programmer has to take care of all that themselves. There are, however, several libraries for doing this, and you can find a list of the most popular ones at `http://www.opengl.org/wiki/Image_Libraries`.

17.2 An example program

We'll use `ex11.c` to illustrate the basics of texturing, in conjunction with a simple image loader for BMP images, in the files bitmap.c, and bitmap.h which `ex11.c` includes. The loader does low-level file-reading and binary decoding, and we don't need to be concerned with the details here. Let's look at `ex11`, and you can see its output in Figure 17.1.

```
1 #include <GL/glut.h>
2 #include "ex11-bitmap.c"
3
4 GLfloat white_light[]=     { 1.0, 1.0, 1.0, 1.0 };
5 GLfloat light_position0[]= { 5.0, 5.0, 5.0, 0.0 };
6 GLfloat matSpecular[]=     { 1.0, 1.0, 1.0, 1.0 };
7 GLfloat matShininess[]=    { 50.0 };
```

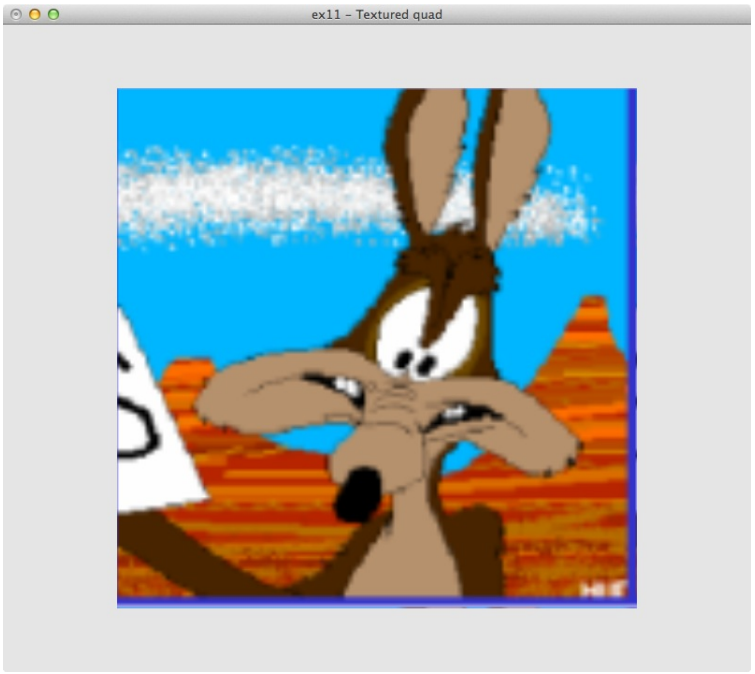


Figure 17.1: ex11.c.

```
 8 GLfloat matSurface[]=      { 0.0, 1.0, 0.0, 0.0 };
 9
10 BITMAPINFO *TexInfo; // Texture bitmap information
11 GLubyte    *TexBits; // Texture bitmap pixel bits
12 int texName;
13
14 void initialise(void) {
15    TexBits = LoadDIBitmap("coyote.bmp", &TexInfo);
16    glGenTextures(1, &texName);
17    glBindTexture(GL_TEXTURE_2D, texName);
18
19    glLightfv(GL_LIGHT0, GL_POSITION, light_position0);
20    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
21    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
22    glEnable(GL_LIGHTING);
23    glEnable(GL_LIGHT0);
24    glEnable(GL_DEPTH_TEST);
25    glShadeModel(GL_SMOOTH);
26 }
27
28 void display(void) {
29   glClearColor(0.9, 0.9, 0.9, 0.0);
30    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
31
32    // define material properties
33    glMaterialfv(GL_FRONT, GL_SPECULAR,  matSpecular);
34    glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
35    glMaterialfv(GL_FRONT, GL_AMBIENT,   matSurface);
36
```

```
37    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
38    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
39    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TexInfo->bmiHeader.biWidth,
40                 TexInfo->bmiHeader.biHeight, 0, GL_RGB,
41                 GL_UNSIGNED_BYTE, TexBits);
42    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
43    glEnable(GL_TEXTURE_2D);
44
45    float sz= 1.2;
46    glBegin(GL_QUADS);
47    glTexCoord2f(0.0, 1.0); glVertex3f(-sz,-sz,0.0);
48    glTexCoord2f(1.0, 1.0); glVertex3f(sz,-sz,0.0);
49    glTexCoord2f(1.0, 0.0); glVertex3f(sz,sz,0.0);
50    glTexCoord2f(0.0, 0.0); glVertex3f(-sz,sz,0.0);
51    glEnd();
52
53    glutSwapBuffers();
54 }
55
56 void reshape(int w, int h) {
57    glViewport(0, 0,(GLsizei) w,(GLsizei) h);
58    glMatrixMode(GL_PROJECTION);
59    glLoadIdentity();
60    gluPerspective(17.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
61    glMatrixMode(GL_MODELVIEW);
62    glLoadIdentity();
63    gluLookAt(0.0,0.0,10.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
64 }
65
66 int main(int argc, char** argv) {
67    glutInit(&argc, argv);
68    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
69    glutInitWindowSize(800, 800);
70    glutCreateWindow("ex11␣-␣Textured␣quad");
71    glutDisplayFunc(display);
72    glutReshapeFunc(reshape);
73    initialise();
74    glutMainLoop();
75    return 0;
76 }
```

The program defines lights and material properties as we saw in Chapter 16 (lines 8–12, 19-23 and 33-35) and enables z-buffering (line 24) and smooth shading (line 25).

To perform texturing, we first need to read the texture image in. The loader (in `bitmap.c` defines a new type `BITMAPINFO` and a function `LoadDIBitmap()` which opens and decodes the file splodge.bmp, and loads its pixels into `TexBits`, and records other other information, such as the image size, in `TexInfo`:

```
BITMAPINFO *TexInfo;   // Texture bitmap information
GLubyte *TexBits;         // Texture bitmap pixel bits

TexBits= LoadDIBitmap("coyote.bmp", &TexInfo);
```

We now need to create an OpenGL texture object to work with, using **glGenTextures()**:

```
void glGenTextures ( GLsizei n,
                     GLuint *textureNames );
```

glGenTextures() creates `n` texture objects, and returns an array of their names. In `ex11.c` we generate just 1 texture.

We then need to select a texture object to work with. This is known as 'binding' a texture, and we use **glBindTexture()**:

```
void glBindTexture ( GLenum target,
                     GLuint textureName );
```

For simple 2D textures we specify a target of `GL_TEXTURE_2D`, but there are several other types of target – see `http://www.opengl.org/sdk/docs/man/xhtml/glTexImage2D.xml` for details.

Next we need to specify how we want the texture to be applied to our polygons, which we do using **glTexParameteri**:

```
void glTexParameteri ( GLenum target,
                       GLenum parameterName,
                       TYPE parameterValue );
```

Again, for simple textures we use a target of `GL_TEXTURE_2D`, but other targets are possible (see the above link). For setting the texture mapping behaviour we want, we use `parameterName` and `parameterValue`. Again, there are many parameters we can set, and here we mention the most common parameters, as used in `ex11.c`:
GL_TEXTURE_MAG_FILTER sets the magnification filter, used when the texel resolution of the texture is less than the pixel resolution being mapped to. This can be be `GL_NEAREST` or `GL_LINEAR`; **GL_TEXTURE_MIN_FILTER** sets the minification filter, used when the texel resolution of the texture exceeds the pixel resolution being mapped to. This can be be `GL_NEAREST` or `GL_LINEAR`.

Now, we set the texels in the texture using:

```
void glTexImage2D ( GLenum target,
                    GLint level,
                    GLint internalFormat,
                    GLsizei width,
                    GLsizei height,
                    GLint border,
                    GLenum format,
                    GLenum type,
                    const GLvoid *data );
```

Lots of arguments here – for the complete story see `http://www.opengl.org/sdk/docs/man/xhtml/glTexImage2D.xml`. In brief:

- `target`: the type of texture to use. For simple 2D textures, this is `GL_TEXTURE_2D`.
- `level`: this specifies the level-of-detail, and is used when we specifiy a set of mip-maps. For simple textures, set this to 0.
- `internalFormat` specifies how to interpret the bits of the texels. In `ex11.c` we use `GL_RGB`.
- `width` and `height` give the width/height of the texel array. `border` is always 0. `format` specifies the pixel data with which the texture will be combined. In `ex11.c` we use `GL_RGB`. `type` specifies the data type of the pixel data. In `ex11.c` we use `GL_RGB`. `data` is a pointer to the start of the texel data.

No to decide how the texture data should interact with the rendered pixel data.

```
void glTexEnvf ( GLenum target,
                 GLenum pname,
                 TYPE param );
```

Again, `target` will usually be `GL_TEXTURE_2D`. `pname` has three possible values, but here we only mention `GL_TEXTURE_ENV_MODE`, which specifies how the texture data is combined with the pixel data. Possible values are `GL_ADD`, `GL_MODULATE`, `GL_DECAL`, `GL_BLEND`, `GL_REPLACE`, and `GL_COMBINE`. In `ex11.c` we use `GL_REPLACE`.

Then, we enable texture mapping, with:

```
glEnable(GL_TEXTURE_2D);
```

The final piece of the jigsaw is to specify how the texture should be aligned relative to the geometry to which it's to be applied. We do this by using **glTexCoord2f()** to associatw parts of the the texture array, which has the coordinate range $[0, 1]$ in s and t, with specific vertices

during the definition of the geometry.

void **glTexCoord2f** (GLfloat *s*,
GLfloat *t*);

In `ex11.c` we do this as follows:

```
glBegin(GL_QUADS);
   glTexCoord2f(0.0, 1.0); glVertex3f(-s,-s,0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(s,-s,0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(s,s,0.0);
   glTexCoord2f(0.0, 0.0); glVertex3f(-s,s,0.0);
   glEnd();
```

Appendix A

The cogl script

`cogl` is handy for compiling a single OpenGL program, which is normally sufficient for simple applications. You can find cogl on-line at `/opt/common/bin/cogl`.

```
#!/bin/bash
#
# NOTE: Specific to the OpenGL installation in the
# School of Computer Science at The University of Manchester
#
# Authors: Karl Sutt and Toby Howard
# Last edited: 2 Jan 2013
# License: GPLv2

# Prints the usage message
function usage
{
echo "␣Usage:␣cogl␣myOpenGLProgram.c␣[outfile]"
echo "␣If␣you␣omit␣\"outfile\"␣the␣output␣will␣be␣\"myOpenGLProgram\""
}

# INFILE is the source file. OUTFILE is the target file.
INFILE=$1
OUTFILE=${INFILE%.*} # strip off . and following characters.

# If we have no arguments, then print usage message and exit
if [ $# -eq 0 ]; then
   usage
   exit 1
else
   if [ "$2" != "" ]; then # If the second argument is provided, use that
      OUTFILE=$2
   fi
   echo "␣␣cogl:␣compiling␣\"$INFILE\",␣output␣will␣be␣\"$OUTFILE\""
   gcc -O3 -fomit-frame-pointer -march=i686 -m32 -Wall -pipe \
          -DFX -DXMESA -lGL -lGLU -lglut -lX11 -lm -g -o $OUTFILE $INFILE
   exit
fi
```

Appendix B

Using a makefile

`cogl` is handy for compling a single OpenGL program, which is normally sufficient for simple applications. For more complex projects, however, which split functions across several files, it's better to use a makefile.

We won't discuss here the general principles of makefiles – that's a whole topic in itself – but here's a sample makefile for accessing the Mesa libraries on the Linux teaching system. You can find this makefile on-line at `/opt/info/courses/OpenGL/Makefile`.

```
INCDIR = /usr/include
LIBDIR = /usr/lib
XLIBS = -L/usr/X11/lib -L/usr/X11R6/lib -lX11 -lXext -lXmu -lXt -lXi
GL_LIBS = -L$(LIBDIR) -lglut -lGLU -lGL -lm $(XLIBS)
LIB_DEP = $(LIBDIR)/$(GL_LIB) $(LIBDIR)/$(GLU_LIB) $(LIBDIR)/$(GLUT_LIB)

CC = gcc

CFLAGS = -I${INCDIR} -I/usr/X11R6/include -O3 -fomit-frame-pointer \
                 -march=i486 -Wall -pipe -DFX -DXMESA
thegears:       thegears.o
        ${CC} ${CFLAGS}  thegears.o -o thegears ${GL_LIBS}
```

Appendix C

Advanced matrix operations

You can usually create the matrices you want by using the simple matrix manipulation functions **glLoadIdentity()**, **glTranslate()**, **glScale()** and **glRotate()**, but sometimes you need to provide arbitrary 4×4 matrices of your own. The functions described in this section enable you to do this. Refer to Section C.1 for details of how OpenGL interprets the sequence of elements in an arbitrary matrix.

void **glLoadMatrixf** (const GLfloat *m);

glLoadMatrixf() takes a matrix `m` (a pointer to a sequence of 16 floats) and sets the current matrix C to this matrix:

$$C \leftarrow m$$

glMultMatrixf() takes a matrix `m` (a pointer to a sequence of 16 floats) and post-multiplies it with the current matrix C, as follows:

$$C \leftarrow C \cdot m$$

void **glMultMatrixf** (const GLfloat *m);

C.1 How an OpenGL matrix is stored

By using the utility functions such as **glRotatef()**, **glMultMatrixf()**, and so on, it's simple to create and manipulate matrices. Some applications, however, may wish to create their own matrices, and pass them to OpenGL.

In order to do this correctly, it's necessary to know how OpenGL stores its matrices internally.

Suppose you wanted to create your own matrix and pass it to OpenGL. We'll take the simple

example of a matrix to perform a translation by (x, y, z), which has the mathematical form:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We would normally declare such a matrix in C as follows:

```
/* assume x, y and z are already declared */

float M[4][4]= { 1, 0, 0, x,
                 0, 1, 0, y,
                 0, 0, 1, z,
                 0, 0, 0, 1 };
```

C stores multi-dimensional arrays in **row-major** format, so `M` is actually this sequence of 16 floats (decimal points omitted for clarity):

```
{ 1, 0, 0, x, 0, 1, 0, y, 0, 0, 1, z, 0, 0, 0, 1 }
```

OpenGL, however expects matrices to be in **column-major** format, where an ordered sequence of elements e_1 through e_{16} defines the following matrix:

$$\begin{bmatrix} e_1 & e_5 & e_9 & e_{13} \\ e_2 & e_6 & e_{10} & e_{14} \\ e_3 & e_7 & e_{11} & e_{15} \\ e_4 & e_8 & e_{12} & e_{16} \end{bmatrix}$$

This is the transpose of row-major format. So, if we pass the matrix `M` to OpenGL, as the argument to **glLoadMatrixf()** or **glMultMatrixf()**, we won't get the result we expect. OpenGL would access the 16 elements of `M` 'column-wise' and create the following OpenGL matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

This is a matrix for three-point perspective – not translation! The results will be spectacular, but spectacularly wrong.

The safest thing to do is to stick to OpenGL's functions for manipulating matrices – then you need never worry about the way they're stored. But if you do need to compute exotic matrices and pass them to OpenGL, be very careful with the row/column ordering.

Index

The names of OpenGL functions are printed in **bold**, and a bold page number indicates the main description of the function.