

Lecture Notes on Computer Graphics using OpenGL

Jonathan G. Campbell
Department of Computing,
Letterkenny Institute of Technology,
Co. Donegal, Ireland.

email: jonathan dot campbell (at) gmail.com, jonathan.campbell@lyit.ie

URL:<http://www.jgcampbell.com/graphics1/cgogl.pdf>

Report No: jc/08/0005/r

Revision 4.1 (better examples in chapter 13)

Revision 4.0 (chapters 3—6 substantially revised)

22nd November 2008

Contents

1	Introduction	1
1.1	Purpose	1
1.2	Reading List	1
1.3	What is OpenGL?	4
2	Images, Displays, Animation, Colour	1
2.1	Introduction	1
2.2	3D Graphics, Images, Animation	1
2.3	Digital Images	1
2.3.1	Continuous/Analogue versus Discrete/Digital	1
2.3.2	Analogue to Digital Converters and Digital to Analogue Converters	3
2.3.3	Images and Digital Images	3
2.3.4	Anti-aliasing	9
2.3.5	Opacity, alpha	9
2.4	Displays and Factors Associated with them	9
2.4.1	Display Hardware — CRTs, LCDs, Plasmas	9
2.4.2	Flicker and Tearing	11
2.5	Visual Perception	12
2.6	A Model of a General Imaging System	14
2.6.1	Light and reflection	15
2.6.2	Motivation	15
2.6.3	Uneven Illumination	16
2.6.4	Uneven Sensor Response	16
2.6.5	Diffuse and Specular Reflection	16
2.7	Colour	17
2.7.1	Electromagnetic Waves and the Electromagnetic Spectrum	17
2.7.2	The Visible Spectrum	19
2.7.3	Sensors	21
2.7.4	Spectral Selectivity and Colour	21
2.7.5	Spectral Responsivity	22
2.7.6	Colour Display	22
2.7.7	Additive Colour	23
2.7.8	Colour Reflectance	23
2.7.9	Exercises	23
2.8	Cameras and Photographic Film	24
2.9	More on Colour Images	24

3	Introduction to OpenGL	1
3.1	What is OpenGL?	1
3.2	Your First OpenGL Program	2
3.2.1	hello.cpp	2
3.2.2	Dissection of hello.cpp	5
3.3	GLUT — GL Utility Toolkit	10
3.4	Graphic User Interfaces and Event Driven Programming	10
3.4.1	Introduction	10
3.4.2	Command Line Interface	11
3.4.3	Graphic User Interface and Events	11
3.5	C or C++?	12
3.6	Visual Studio	12
3.7	Float or Double?	13
3.8	OpenGL Types and Multiple Versions of Commands	14
3.9	Animation and Simple Interaction	14
4	More 2D Graphics	1
4.1	Points, Lines, and Polygons	1
4.1.1	Points and Homogeneous Coordinates	1
4.1.2	Specifying Vertices	1
4.1.3	Lines	2
4.1.4	Polygons	2
4.1.5	OpenGL Geometric Drawing Primitives	2
4.2	Displaying Points, Lines, and Polygons	3
4.2.1	Point Details	3
4.2.2	Line Details	3
4.3	Drawing Lines — a program example	5
4.4	Details on Polygon Rendering	8
4.4.1	Polygons as Points, lines, or Solids	8
4.4.2	Reversing and Culling Polygon Faces	8
4.5	Stippling Polygons	9
4.6	OpenGL and SDL	9
5	Introduction to 3D Graphics	1
5.1	Your first 3D program, cube.cpp, a wireframe cube	1
5.2	Mouse motion callback	8
5.3	glFrustum and gluPerspective	9
5.4	Reading the Contents of Transformation Matrices	13
5.5	Concatenating (composing) Transformations	16
5.6	A Solar System, planet.cpp	22
5.7	3-D House Example	25
5.7.1	Dissection of house3d.cpp	29
5.7.2	Assertions	30
5.7.3	Mouse Operated Menu	30
5.7.4	GLUT Timer function	31
5.8	A further example	32
5.9	Additional Clipping Planes	39

6	Lighting	1
6.1	Background Theory	1
6.1.1	Colour Sensing	2
6.1.2	Colour Reflectance	2
6.2	OpenGL Light and Material Models — Ambient, Diffuse, Specular, Emissive	3
6.3	Mathematical Description of the OpenGL Lighting Model	6
6.4	Additional Considerations	8
6.5	Your first lighting program, light.cpp	10
6.5.1	Dissection of light.cpp	14
6.6	Further example of materials and lights, material.cpp	17
6.7	Example of moving light, movelight.cpp	19
6.8	Simplifying Materials Specification using glColorMaterial	21
6.9	Normals	21
6.9.1	Example using the 3D House	24
6.10	Lighting Calculation Example.	26
7	Blending, Antialiasing, and Fog	1
7.1	Blending	1
7.2	Your first blending program, alpha2.c	1
7.3	Three-Dimensional Blending with the Depth Buffer	6
7.4	Antialiasing	6
7.5	Fog	10
8	Vertex Arrays, Vertex Buffer Objects, and Display Lists	1
8.1	Introduction	1
8.2	Display Lists	1
8.3	Vertex Arrays	4
8.3.1	Introduction	4
8.3.2	First Try — plain arrays	7
8.3.3	Second Try — plain arrays and for loop	8
8.3.4	Now Vertex-arrays	9
8.3.5	Vertex-array in a Buffer Object	12
9	Images, Font etc. in OpenGL	1
10	Texture Mapping	1
10.1	Your first texture mapping program, cubetex.c	1
11	GLU Quadrics	1
12	Interpolated Curves and Surfaces and OpenGL Evaluators	1
12.1	Interpolation	1
12.1.1	Linear Interpolation	2
12.1.2	Spline Interpolation	2
12.2	Bézier Curves	5
12.3	Your first Bézier curve program	7
12.4	Two Dimensional Evaluators	9

13 OpenGL Shading Language	1
13.1 Books and Sources	1
13.2 OpenGL Pipeline	1
13.2.1 Introduction	1
13.2.2 Shaders and GLSL	4
13.2.3 Pipeline, Detailed Review	5
13.3 GLSL Shaders — Simple Examples	9
13.3.1 Minimal Shaders	10
13.3.2 Slightly more ambitious	11
13.3.3 Diffuse lighting plus toon shading	12
13.3.4 OpenGL Program that Uses Shaders	15

Chapter 1

Introduction

1.1 Purpose

This document is an introduction to aspects of computer graphics using OpenGL. It provides course notes for the *two* Letterkenny Institute of Technology modules:

- Graphics Programming for Games 1 (year 2); this course uses Chapters 1 to 6 inclusive.
- Graphics Programming for Games 2 (year 3); this course uses Chapters 6 to 13 inclusive; we normally revise Chapter 6 (Lighting) in the second course because lighting crops up a lot in GLSL (chapter 13).

Separately, for each of the courses, we will hand out brochures giving *aims and objectives, the syllabus, assessment, policies, etc.*

1.2 Reading List

Essential

1. These notes.
2. Jonathan Campbell, *Notes on Mathematics for 2D and 3D Graphics*.
3. Jonathan Campbell, *Computer Graphics using OpenGL — Exercises*.

Recommended.

Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1*, 6th Ed. Addison Wesley, 2008.

The following, or earlier editions are equally useful for this course:

Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.0*, 5th Ed. Addison Wesley, 2005.

The programs from the book will be in my public folder.

R.S. Wright and B. Lipchak and N. Haemel, *OpenGL Superbible*, Sams, 4th ed., 2007, ISBN: 0-321-49882-8. Book website: <http://www.starstonesoftware.com/OpenGL/>.

Edward Angel, *OpenGL: a primer*, (or 2nd ed.), Addison Wesley, 2008, ISBN: 0321398114. Book website (programs only): http://www.cs.unm.edu/~angel/BOOK/PRIMER/THIRD_EDITION/.

The programs from the book will be in my public folder.

Paul Martz, *OpenGL Distilled*, Addison Wesley, 2006.

At a first glance this looks hard to follow; yes, it is terse, but once you have a good grasp of OpenGL, this book answers questions not clearly answered elsewhere.

Edward Angel, *Interactive Computer Graphics: a top-down approach using OpenGL*, 4th ed. (5th ed. now available), Addison Wesley, 2005.

OpenGL Architecture Review Board (Dave Shreiner Editor), *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*, Addison Wesley, 2000. (This is the OpenGL Blue Book. Handy if you need to track down the details of any OpenGL function.) If you were going to spend a long time programming a large OpenGL application, you would need to have both the Red Book and Blue Book by your right hand, but for this course, Angel's OpenGL primer will suffice.

Nehe Productions, *Online OpenGL Tutorials*, <http://nehe.gamedev.net/>.

Many think these tutorials are the best introduction to OpenGL.

Hearn and Baker — see the Bibliography.

Buss — see the Bibliography.

Indicative.

The long list below is roughly in order to how much I learned from each in the course of preparing the course. You too may be able to learn from them, but you need not consider buying any of them. Other books are included in the bibliography.

J.D. Foley and A. van Dam and S.K. Feiner and J.F. Hughes, *Computer Graphics: principles and practice*, 2nd ed., Addison Wesley, 1990, ISBN. 0-201-12110-7.

There is a minor revision of this called "Second Edition in C" published in 1996 (ISBN. 0-201-848406), but the difference between them is hardly worth talking about.

[We could do with a more modern book than this, but it's still the standard introductory text; pity about the archaic API and other irrelevancies. If you can get hold of a very cheap second hand copy, maybe worth having.]

J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, and R.L. Phillips, *Introduction to Computer Graphics*, Addison Wesley, 1994, ISBN: 0-201-60921-5.

Mark Segal and Kurt Ageley, *The OpenGL Graphics System, a Specification, version 2.0*. Available at: <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf> I'll have a local copy in my public folder `msc2d3d\glspec\`.

E. Lengyel, *Mathematics for 3D Programming and Computer Graphics*, Charles River Media, 2nd ed., 2004, ISBN: 1-58450-277-0.

Fletcher Dunn and Ian Parberry, *3D math primer for graphics and game development*, Wordware Publishers, 2002, ISBN: 1556229119.

John Vince, *Essential mathematics for computer graphics fast*, Springer-Verlag, 2001. ISBN: 1852333804.

Donald Hearn and M. Pauline Baker, *Computer graphics with OpenGL*, Prentice Hall, 2003, ISBN: 0131202383.

F.S. Hill(Jr.), *Computer Graphics Using OpenGL*, 3rd ed., Prentice Hall, 2008.

Dave Astle and Kevin Hawkins, *Beginning OpenGL Game Programming*, Premier Press / Thompson Course Technology, 2004.

Gene Davis, *Learning Java Bindings for OpenGL (JOGL)*, Lightning Source UK Ltd, 2005, ISBN. 42080362X. available for download purchase at: <http://www.genedavissoftware.com/books/jogl/index.html>. Mentioned just in case you ever need to use OpenGL through Java.

Andrew Watt, *3D Computer Graphics*, Addison-Wesley, 3rd ed., 2000.

Many would claim that this is the current standard on the theory of 3D computer graphics, i.e. the current replacement for the Foley and vanDam books. The book is very complete, but difficult reading — reads like it was written by an automaton, for reading by other automata.

Andrew Watt and Fabio Polcarpo, *3D Games, Volume 1*, Addison-Wesley, 2001.

Andrew Watt and Fabio Polcarpo, *3D Games, Volume 2*, Addison-Wesley, 2003. ISBN: 0201787067.

Tomas Akenine-Moeller and Eric Haines, *Real-time Rendering*, 2nd ed., A.K. Peters, 2002.

David H. Eberly, *The 3D Game Engine Architecture: Engineering Real-time Applications with Wild Magic*, Morgan Kaufmann, 2004.

Noel Llopis, *C++ for Game Programmers*, Charles River Media, 2003.

Mike McShaffrey, *Game Programming Complete*, 2nd ed., Paraglyph Press, 2005.

Samuel P. Harbinson and Guy Steele, *C: A Reference Manual*, 5th ed., Prentice Hall, 2002, ISBN: 013089592X.

Web Resources.

My course website is: <http://www.jgcampbell.com/graphics2/>. Keep an eye on that.

Nehe Productions, *Online OpenGL Tutorials*, <http://nehe.gamedev.net/>.

See above. Many think these tutorials are the best introduction to OpenGL.

My OpenGL links. Fairly random and unsorted. <http://www.jgcampbell.com/links/opengl.html>

Nate Robins' OpenGL Tutors. Superb demonstrations of key OpenGL features. A must see. <http://www.xmission.com/~nate/tutors.html>

I'll have local copies of the executables in my public folder `graphics2\progs\nate\`.

OpenGL website. <http://www.opengl.org>.

Microsoft make available a free version of their Visual C/C++ compiler and IDE:

<http://msdn.microsoft.com/vstudio/express/visualc/>

It does everything that we need; handy for those of you with your own computers.

1.3 What is OpenGL?

OpenGL is a 3D graphics application programmers interface (API). It is procedural, by which I mean to say that it is not object-oriented such as you may have become used to.

As a brief and incomplete summary, OpenGL API provides functions which allow programming of the following.

- Specification (modelling) of an arbitrarily complex set of objects in 3D space; this includes:
 - The positions of multiple objects are related by affine transformations, i.e 4-D homogeneous matrices;
 - Typically, object specifications are based on *vertices* (vertexes if you wish), i.e. *points*, for example `glVertex3f(0.25, 0.5, 0.0)`; specifies a point at $x = 0.25, y = 0.5, z = 0.0$. If your world is 2D, you can work entirely with vertexes whose $z = 0$; or, there is a set of `glVertex2*` functions. `glVertex3f` means that the function expects three (3) floats;
 - Specification of object colours, e.g. `glColor3f(1.0, 0.0, 0.0)`; specifies that the next object are to be bright red; (red = 1, green = 0, blue = 0);
 - We can specify lighting, in which case, the object's appearance when rendered will depend on its own colour, and on the colour and intensity and direction etc. of the light source.
- Specification of a virtual camera by which to view the 3D virtual world.

When the program is executed, OpenGL (i) assembles the virtual world (the scene); (ii) points the virtual camera at the scene (maybe seeing only part of the scene); (iii) projects the scene onto a projection plane (think camera film or array of image sensors in a digital camera, see Chapter 2); (iv) performs the equivalent of spatial sampling and digitisation, see Chapter 2 to produce an image that can be displayed on a computer screen — at least for the meantime, computer screens are 2D!

Outside of the true OpenGL API, we need a set of functions which interact with the windowing provide by the operating system. For this we will use GLUT (GL Utility Toolkit); such functions have the prefix `glut`.

DirectX? DirectX is MicroSoft's games programming API. The 3D graphics part of it is called Direct3D; the principles behind Direct3D are identical to the principles underlying OpenGL.

Chapter 2

Images, Displays, Animation, Colour

2.1 Introduction

This chapter discusses the nature of images, human vision, cameras and other image sensors, the nature of colour. We mention how colour images are represented in a computer.

Also, in discussing eyes and cameras, we prepare ourselves for three-dimensional (3D) graphics. When we look at a 3D world scene with our eyes, or with a camera, we are collapsing that 3D scene onto two dimensions (2D). In the 3D graphics part of a video game we create a virtual 3D world, but to display it on a 2D screen requires the computer to perform (mathematically) the same task as an eye or a camera does.

Some diagrams in this chapter are taken from Gonzalez and Woods (Gonzalez & Woods 2002), chapters 2 and 6.

2.2 3D Graphics, Images, Animation

An *image* is a picture — a two-dimensional (2D) representation of a three-dimensional *scene*. *2D Graphics* is about creating images. *Animation* is about moving images. *3D Graphics* is about creating 3D virtual scenes; the graphics engine must then manipulate the components of this scene and eventually ‘take an image of it’ (projecting) using a virtual camera. If the virtual scene is static, then the graphics engine can take its time over the manipulations and projection. However, games are dynamic. Most of the hard work in a video game is about manipulating the objects in the 3D virtual scene, then projecting to create an *image* and displaying successive images on a screen at a rate fast enough that the viewer has the sensation of continuity in time; this is *animation*.

2.3 Digital Images

2.3.1 Continuous/Analogue versus Discrete/Digital

Some aspects of our world are *continuous*, some *discrete*; roughly speaking, in a computer world, we use the terms *analogue* and *digital*. Many quantities and ‘things’ in the real world are *continuous*:

lengths, volumes, areas, masses, weights; that is until you get down to atomic sizes — as Max Planck and Albert Einstein discovered when they came across the quantum theory of physics.

If you want to *represent* real world quantities in a computer you need to convert them to *discrete* or *digital* versions; but apart from digital images taken with a digital camera, our computer game virtual worlds will be digital from the start.

An *image* is some sort of representation of some part of the real world. A painting is an image; a printed photograph is an image; both are *continuous* representations — meaning that you can magnify a part of the image larger and larger and it still looks *smooth*.

Contrast *digital* images. These are made up of little blocks called *pixels* — short for *picture elements*. An individual pixel has a fixed colour. If you keep magnifying a digital image eventually you will see that it is a joined up patchwork of *pixels*. But if the pixels are small enough — we say that the image has high *spatial resolution* — your eye sees the image as continuous.

If the discrete blocks are small enough the whole lot looks continuous. High resolution. Same for time and space.

Time is continuous. But in a clock it is represented as a digital quantity; in a normal ticking clock such as a grandfather clock, it is digitised into seconds; in an electronic watch it is digitised into whatever is the frequency of the vibrating crystal that drives it.

There are two parts to digitisation: (a) the chopping up into blocks (in the case of images) and into time samples in the case of sound and video — this is *sampling*; (b) converting brightness, in the case of images, and loudness, in the case of sounds, into numbers — this is *digitisation*. But sometimes sampling and digitisation are lumped together as *digitisation*.

Real Numbers versus Integer Numbers First of all we have the *natural* numbers $\{0, 1, 2, \dots\}$; these are the numbers we use when we count. *Integers* include the natural numbers together with their negatives, $\{-\infty, \dots, -2, -1, 0, 1, 2, \dots, +\infty\}$.

Real numbers are an entirely different matter. When we measure things, e.g. the weight of a piece of cheese, the length of a piece of string, we have a *real* number. You might say, the weight of this piece of cheese is 25 grams. Not much difference from a natural number, I hear you say. But, almost certainly, in coming up with the 25, you merely took the nearest natural number. If you were in a laboratory, using a very accurate instrument, you might have originally had 25.4124359267 grams. At home you might have got 25.4 grams. Which is correct, 25, 25.4, 25.4124359267 grams? Actually, none of them. If you wanted to be fully correct, you'd have to use thousands and millions of digits; to be exact, infinity of them. Real numbers form a continuum. Between any two real numbers, no matter how close, there are an infinite number of other real numbers. The more precision you use in your measurement, the more digits you get.

In contrast, between 3 and 6, there are just two other integer numbers, 4 and 5. You cannot be any more precise than counting the people in a classroom and stating, for example, there are 23 students present.

It is easy to represent integers in a computer; the only slight problem is that you cannot get to $+\infty$ or $-\infty$, but you can get as close as you need.

Ex. (a) What are the largest negative and positive numbers possible in a C++ `int` variable? (b) C++ `short`? (c) C++ `byte`? (d) What is the difference between `signed` and `unsigned`

2.3.2 Analogue to Digital Converters and Digital to Analogue Converters

When you have an analogue signal and need to get it into a computer, you need an Analogue to Digital Converter (ADC); an ADC performs two tasks: (a) samples in time and (b) converts to numbers.

When you have an digital signal or a set of numbers in a computer and you want to send them to, for example, a loudspeaker, you need an Digital to Analogue Converter (DAC); a DAC converts to numbers to voltage or current. Because there may be stepiness or blockiness in the analogue signal so produced, a DAC is often followed by a smoothing filter. *Anti-aliasing*, see below, is a form of smoothing out of digital effects in image displays.

Back to images.

2.3.3 Images and Digital Images

The term image or, strictly, monochrome image, refers to a two-dimensional brightness function $f(x, y)$, where x and y denote spatial coordinates, and the value of f at any point (x, y) gives the brightness (grey level or colour) at that point.

Monochrome versus colour Monochrome images are grey level images; they are sometimes called *black-and-white* — but incorrectly, for black-and-white implies that there are just two values: black, and white, and no in-between. — i.e. $f(x, y)$ is a grey level.

In a colour image $f(x, y)$ gives a colour. A colour image can be represented by three monochrome images, each representing the intensity of a primary colour (red, green, blue). Thus,

$$f_r(x, y), f_g(x, y), f_b(x, y).$$

Getting closer to programming language notation, a colour image is represented by $f(b, x, y)$, where b denotes colour (b is colour band), where $b = 0, 1, \text{ or } 2$, for red, green, or blue.

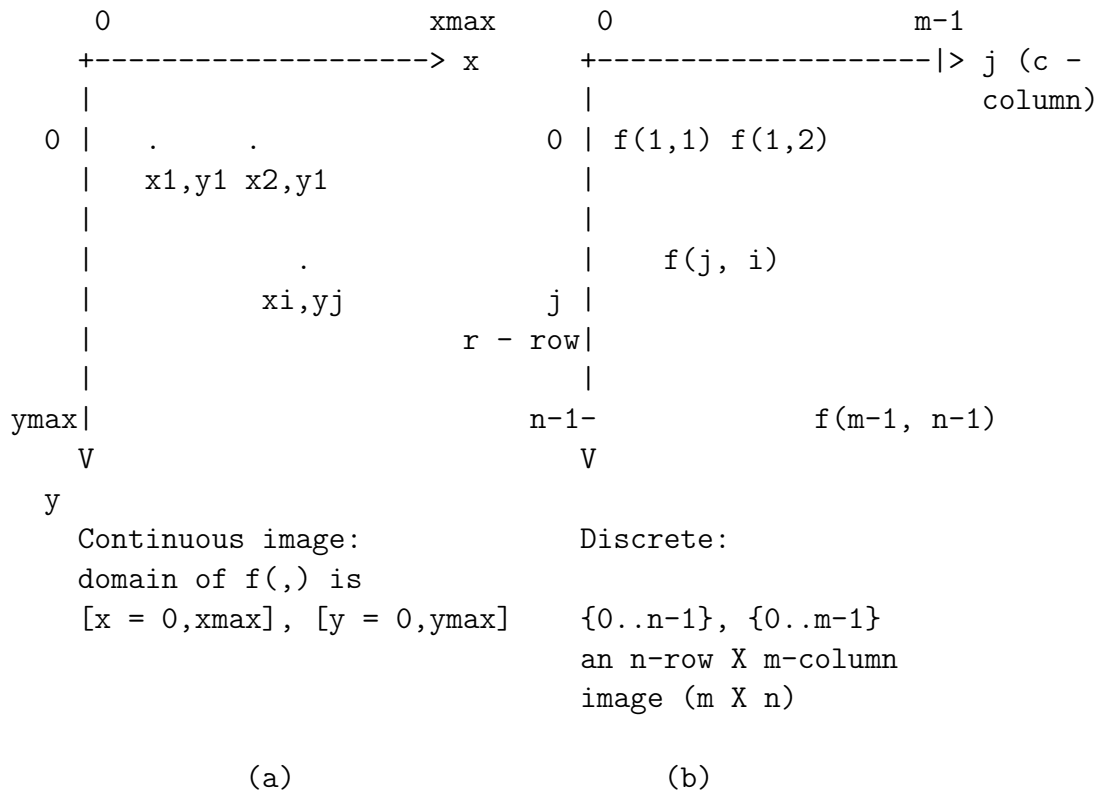
Digital The monochrome image, $f(x, y)$, mentioned above is still continuous, and in *two* senses: (i) $f(x, y)$ is a real (continuous valued) number, and, (ii) x , and y are real numbers. So, continuous valued, and spatially continuous — like a photograph.

Thus, you can achieve infinitesimally fine resolution in $f(x, y)$, and in x , and y .

In computers we must use *digital* (or *discrete*) approximations. We approximate $f(., .)$ by restricting it to a discrete set of grey levels; often an 8-bit integer $0 \dots 255$, and, we sample $f(., .)$ at a discrete lattice of points, $x_i, i = 0 \dots n - 1$, and $y_j, j = 0 \dots m - 1$; see Figure 2.1.

Thus, we arrive at a digital image, $f(r, c)$, where f can take on discrete values $0 \dots G - 1$ and $r, 0 \dots n - 1, c, 0 \dots m - 1$.

f can now be viewed as a matrix (two-dimensional array) of numbers,



There is inconsistency between (a) and (b); in the real world, x is the horizontal axis and is the first argument, and y is vertical axis and the second argument. In digital images we mostly deal with row, r , vertical, and column, c , horizontal.

Also, we have the problem that computers like r to grow $\downarrow$, rather than as y , up, in the real world.

Figure 2.1: Correspondence between continuous and discrete axes

$$f(r, c) = \begin{bmatrix} f(0, 0) & f(0, 1) & \dots & f(0, m-1) \\ f(1, 0) & f(1, 1) & \dots & f(1, m-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(n-1, 0) & f(n-1, 1) & \dots & f(n-1, m-1) \end{bmatrix}. \quad (2.1)$$

Typically, on graphics cards, we have $n = 768, 1024, \dots$, $m = 1024, 1280, \dots$. G , number of values, is typically 256, i.e. values go $0, 1, \dots, 255$. In a colour image, we have shades of red going from $0, 1, \dots, 255$; and the same for green and blue.

Ex. If we have 256 shades, i.e. 8-bits, each of red, green and blue, how many different colours are possible?

Ex. If we have 8-bits for each of red, green and blue, and the image is 1024×1024 , how many bytes of memory does the image require?

Ex. Eight bits (each) are adequate to represent colours, i.e. if you went to 12 or 16 bits there would be no perceptible improvement. Why do many graphics cards use 32-bits per pixel? There are at least two reasons.

Ex. (a) If you had to digitise a TV image broadcast by RTE or BBC, suggest suitable values of m , n ; (b) using the results of (a), and assuming 25 frames (images) per second, how much data for a one hour film ?

Other examples of digital quantities Music on tape or on vinyl LP is continuous. Music on CD is digital. CD sampling rate is 44,100 samples per second, with 12 or 16 bits per sample, 2 channels (stereo).

Ex. Verify that a 60 minute album will indeed nearly fill a 650-MB CD-ROM.

In modern telephone systems, speech is transferred digitally between major exchanges — here you can get away with 8,000 samples per second, and 8-bits per sample.

Raster scanning The image model given above corresponds to the image model used in *raster* graphics, i.e. the image is formed by regular sampling of the x -, and y -axes.

Pixel Each $f(r, c)$ in eqn 2.1 is a *pixel* (picture element).

Spatial Resolution Spatial resolution is *high* if the samples x_i , y_j are closely spaced, and is low if they are widely spaced. Clearly, the closer the spacing, the more alike the digital image will be to the original, i.e. we are always demanding higher resolution. On the other hand, the higher the resolution, the larger are m , n — more data; data volume grows as the square of the resolution.

Another reason to restrict n and m to numbers like 768, 1024 is that displays, see section 2.4, cannot handle any higher resolution.

Ex. Laser printers commonly work at 300 dots (pixels) per inch. How many pixels in an A4 page?

The effects of reducing spatial resolution are shown in Figure 2.2. The original image, upper left, is 256×256 ; the upper right image simulates the effects of reducing resolution and thereby reducing the number of pixels to 128×128 ; similarly, the lower left simulates a 64×64 pixel image and the lower right 32×32 .

These days (2008) a cheap digital camera will give you about 3000×2000 pixels and the most expensive ones about 4000×3000 .

Grey Level Resolution With proper selection of the digitisation range, it is usually possible to represent, without any humanly perceivable degradation, monochrome images using just 8-bits; the psychologists tell us that humans can perceive no more than about 160 levels at once. Also, in my experience, there appears to be some natural law that says that most image sensors cannot deliver any useful higher grey-level / colour resolution.

Ex. Palette-based image representation works a bit differently than using red, green, and blue values for each pixel Explain. See Brackeen book, or do a web search.

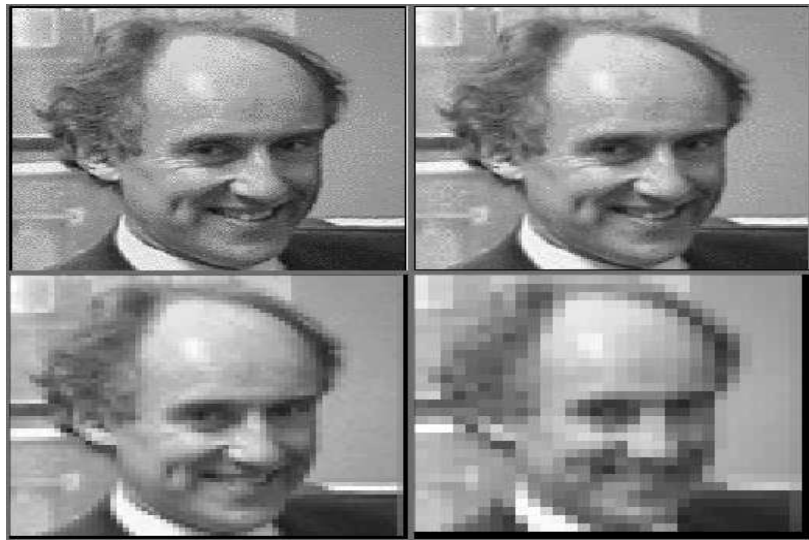


Figure 2.2: Upper left: original image. Upper right: resolution reduced by a factor of 2. Lower left and right: reduced by factors of 4 and 8.

The effects of reducing grey level resolution are shown in Figures 2.3 and 2.4. The original image is the same one that is in the upper left of Figure 2.2.



Figure 2.3: Image quantised to 16 grey levels, top; eight levels, bottom.

After printing on a laser printer and then photocopying, the effects may not be too obvious; on a computer screen, 16 levels (four bits) is indistinguishable from the original, at eight levels (three bits) you start to notice the quantisation, at four levels (two bits) the quantisation is very obvious; and then two levels (one bit). The fact that the image was *noisy* to start off reduces the effect a little.

Ex. (a) Make sure you are comfortable with the correspondences: 16 levels, 4-bits; 8 levels, 3-bits; 4 levels, 2-bits; 2 levels, 1-bits; (b) what would a 0-bit image look like?

Ex. (a) We have seen that 16 grey levels (4-bits per pixel) might be enough in some cases; how could this be used to reduce the size of an image file? (b) If the original image is $256 \times 256 \times 8$ bits, how many *bytes*? (c) How many bytes in the $256 \times 256 \times 4$ bits image?

Noise We said that the original in Figure 2.2 is *noisy*. The original was taken from one frame of a video; a video camera has to grab image very fast, i.e. spends little time on each grab, hence it has little time to smooth out random variations in the little sensors. This is noise. You get the same sort of noise if you record sounds with a very cheap microphone.

Noise is little variations in the image pixels (or sound samples) that were not there in the scene that the camera pointed at.



Figure 2.4: Image quantised to four levels, top; two levels, bottom.

2.3.4 Anti-aliasing

Pixels are little blocks of colour. When you display a line or text character or some object on top of a background, but which contrasts with the background, the blockiness can be perceivable. We can get rid of the blockiness (*aliasing*) by blending the object into its background. This spatial blending is called *anti-aliasing*. Most graphics cards will handle anti-aliasing for you.

If you look again at the bottom left image in Figure 2.2 you will see blockiness or *aliasing*; if you defocus your eyes, the blockiness will disappear and you will start to think that you see more detail in the image. The blurring caused by defocussing the eyes is the equivalent of *anti-aliasing*.

2.3.5 Opacity, alpha

In a Java program the normal way to create a colour is to give the Red, Green, Blue values:

```
//           R   G   B
Color c = new Color(255, 0, 0); // pure red

// or, you can use 0..1 (float) instead of integer 0..255

Color c = new Color(1.0, 0.0, 0.0); // pure red
```

But you can also specify the *opacity* (*transparency*) of the colour. The *opacity* is called *alpha*; so now we have Red, Green, Blue, Alpha values:

```
//           R   G   B   A
Color c = new Color(255, 0, 0, 255); // pure red, completely opaque

Color c = new Color(255, 0, 0, 0); // pure red, completely
                                transparent

Color c = new Color(255, 0, 0, 50); // is better if you want to see
                                // the colour against a background
```

2.4 Displays and Factors Associated with them

2.4.1 Display Hardware — CRTs, LCDs, Plasmas

Read pp. 106–110 of A.S. Tanenbaum, Structured Computer Organisation, Prentice-Hall, 2005; handed out.

Before you go on, please note that we now have two spatial resolutions, one in the graphics card, the other on the display hardware.

A cathode ray tube (CRT) has groups of red, green, and blue light emitting dots painted on the rear of the glass at the front of the tube, but they are so close together that the surface is treated

as *continuous*. Thus you can easily change the resolution of the images that you send to it. Of course, depending on the size and construction of the CRT, there is an upper limit, normally something like 1600×1200 for a 17-inch CRT. If, in the operating systems settings, you set the display to 640×480 , and display on the latter CRT, you might see a bit of blockiness, but the image would fill the screen.

A liquid crystal display (LCD) is truly digital; typical LCD screens are 1024×768 (15–17-inch) and 1280×1024 for 19-inch. If you set the display to 640×480 , and displayed on the one of the latter LCDs, you would probably get a little image in the middle of the screen and blank around the edges. Some LCD displays have the capability of resampling.

What follows is taken from a Usenet newsgroup sci.image.processing answer by Dave Martindale (d—@cs.ubc.ca), to a question by me.

In the case of LCD screens, there is local memory **in the LCD controller** to remember the state of every pixel in the screen. Because of this, there are two parts to "refreshing" an LCD screen: getting the data from the computer frame buffer (graphics card) to the LCD controller, and then driving the LCD panel itself.

The first part of the process is done *as if* the LCD was a raster device. The pixel data is sent from the computer to the LCD controller in row/column raster order just as if it was a CRT. If you're using an analog connection between computer and display, the graphics card actually generates analog RGB and sync signals just as if it was driving a CRT - it has no way of knowing the display is actually an LCD. Then the display controller generates local clock signals locked to the incoming analog video, and converts the signal back into digital form before storing the pixel data in the LCD controller memory. If the frame buffer pixel clock and the LCD controller pixel clock are not the same, you can end up with pixel jitter artifacts.

On the other hand, if you use a digital (DVI) connection between graphics card and display, the pixel data is transferred in digital form, with no noise introduced by D/A and A/D conversion, and without wasting time for horizontal and vertical sync and blanking periods. This is more efficient, and avoids the need for the digitizing circuitry in the display.

Either way, once the pixel information is in the LCD controller, the controller then uses it to modulate individual pixel cells in the LCD panel itself. This happens at a frequency that is determined by the controller and panel's needs, and it's not necessarily synchronized with the incoming video.

LCD screens have a *native* resolution, i.e. each screen pixel corresponds to a single indivisible element. However, many LCD monitors contain internal resampling hardware that will accept an incoming signal at a wide range of resolutions and then resample that to the actual display resolution. The image tends to lose some crispness when you do this, so it's generally better to operate at the native resolution.

Raster refresh and flicker and tearing The refresh is raster as described above.

The light output from an LCD is continuous; it doesn't come in bright pulses as the electron beam sweeps over the phosphor like a CRT. This means that LCDs don't flicker if they are updated slowly from the computer; 60 Hz refresh flickers visibly on a CRT but not LCD.

On the other hand, tearing is caused when a single displayed image on screen comes from two different points in time. To avoid this with a CRT, it's simply necessary to have the video controller swap buffers (between the previous and next rendered frame) during vertical retrace so each

displayed frame comes from a single point in time. With an LCD, there's the additional delay between display controller update and screen update which can complicate things.

If 25–30 Hz is good enough for television or movies, why the need for 70–85 Hz refresh on computer screens? With a flickering light source, we can see flicker up to about 72 Hz when the light is very bright, dropping below 50 Hz when the light is dim.

Television uses a 25 or 30 Hz complete frame rate, but the image is sent in interlaced mode: all the even scanlines are sent in the first 1/50 or 1/60 second "field", followed by all the odd scanlines in a second field. CRTs display this signal in the same way, so the screen is actually refreshed at 50 or 60 Hz; it just isn't quite the same data each time. As long as there are no drastic changes between two adjacent scanlines (and good TV is filtered so there is not), we don't see flicker.

Movies are shot at 24 FPS, but the projectors use a shutter that interrupts the light either 2 or 3 times per frame, so the actual flicker rate on screen is 48 or 72 Hz. 48 Hz is most common, and works well in most theatres where even the brightest white in the image is not that bright in absolute terms, and most of the image is much dimmer. It's common to see a little bit of flicker in the brightest portion of the image only. 72 Hz shutters are better in small theatres with particularly bright images; that pretty much eliminates visible flicker.

But computer screens aren't interlaced (anymore), and they are operated at high brightness, so you need about 72 or 75 Hz refresh rate to avoid visible flicker.

LCD versus TFT versus Plasma TFT is just one LCD panel technology; it's not a separate type of display. Plasma displays are likely very similar to LCDs when it comes to the computer-display controller connection. Driving a plasma panel takes very different voltages and current from driving an LCD panel. LCDs use very low voltage and current to change the polarization state of the liquid crystals, while the light comes from a separate backlight and two polarizing sheets actually absorb or pass light. Plasma panels emit light directly from each pixel cell.

2.4.2 Flicker and Tearing

Ex. (a) In the context of display updating, what is *flicker*; (b) how does *double buffering* solve the problem of flicker? (c) why, in double buffering, is *page flipping* preferable to copying between buffers?

Ex. (a) What is meant by saying that a CRT display has a refresh rate of 75-Hz? (b) What is *vertical retrace*?

Ex. (a) With an LCD display, you shouldn't really experience *flicker*. Why not?

Ex. (a) Explain, with the aid of a diagram, the *tearing* effect due to lack of synchronisation of *monitor refresh* and *page flipping* / *buffer copying*.

Ex. What is the difference between an *analogue* LCD display and a *digital* one? Be precise.

2.5 Visual Perception

Here, briefly, are some points about human visual perception:

- the perceived image may differ from the actual light image (i.e. the perceived *brightness* image is a considerably modified 'copy' of the physical light intensity emanating from the scene);
- there are two types of light sensors on the retina – rods and cones;
- rods are more sensitive than cones; rods are used for night (scotopic) vision; rods are largely colour insensitive (e.g. no colour evident in moonlight);
- cones are used for brighter light, cones can sense colour;
- perceived (subjective) brightness (B_s) is roughly a logarithmic function of light intensity (L): thus, if you increase L by 10, B_s increases by only 1 unit, increase L by 100 B_s increases by 2 units, 1000 increases by 3 etc.
- the visual system can handle a range of about 10^{10} (10 thousand million) in light intensity (from the threshold of scotopic vision to the glare limit). (Question: how many bits is that?)
- to handle this range, the pupil must adapt by opening and closing the pupil; opening the pupil – in darkness – lets more light in; closing it – in bright light – lets less light in;
- the eye can handle only a range of about 160 levels at any one instant, i.e. where there is no opening and closing of the pupil; of course, this explains why 8-bits (256 levels) usually suffice in a display memory.

Figure 2.5 (from (Gonzalez & Woods 2002)) shows a simplified cross section of a human eye. Note the lens and the light sensitive retina — a cameras also has a lens and light sensitive sensors or film.

Figure 2.6 (from (Gonzalez & Woods 2002)) shows image formation in a a human eye. A camera operates similarly, replacing the retina with light sensitive sensors or film.

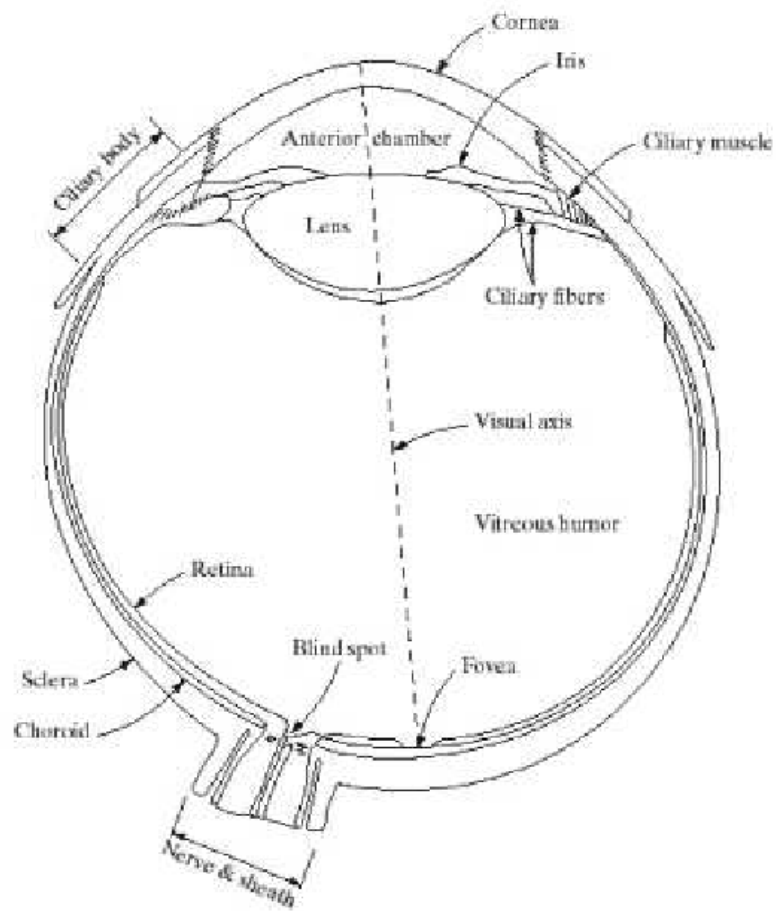


Figure 2.5: Cross section of the human eye.

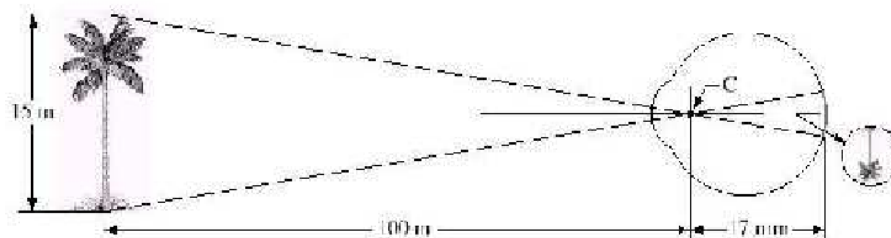


Figure 2.6: Image formation in a human eye.

2.6 A Model of a General Imaging System

Note: in this chapter we treat physical units somewhat informally — a later chapter give a little more detail.

A general camera-based sensing arrangement is shown in Figure 2.7 ((Gonzalez & Woods 2002)). The scene element, some distance from the camera lens, is projected onto the image plane. At the image plane there is a mosaic of light sensitive sensors, see Figure 2.8; this mosaic has the effect of transforming the two-dimensional continuous image lightness function, $f_i(y, x)$, into a discrete function, $f'[r, c]$, where r (ow) and c (olumn) are the discrete spatial coordinates. Then, the electrical voltage or current output from each sensor eventually, $f'[\cdot]$ gets digitised to yield a digital image, $f[r, c]$. Hence, we have two *digitisations*: first *spatial* — a spatial chopping (sampling) into rectangular pixels; next *amplitude* — conversion of analogue sensor output (voltage or current) into numbers. The data are then ready for transferring to computer memory.

For colour, all we need are alternating red, green and blue sensors. Normally, the sensors will all be the same, but will have red, green, or blue filters in front of them.

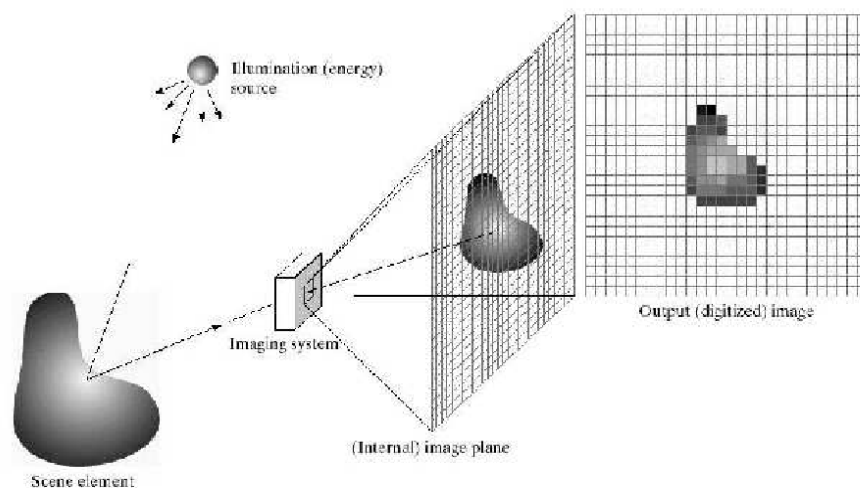


Figure 2.7: Image acquisition system (camera).



Figure 2.8: Continuous image (a) projected onto a sensor array; (b) sampled (spatially) and quantised (in amplitude values).

Thus, we arrive at a digital image: $f[r, c]$ where f can take on discrete values $[0, 1, \dots, G - 1]$ and $r \in [0, 1..nRows - 1]$, $c \in [0, 1..nCols - 1]$. And if the camera handles colour, we have three images, f_{red} , f_{green} , f_{blue} .

$$f[r, c] = \begin{bmatrix} f[0, 0] & f[0, 1] & \dots & f[0, nCols - 1] \\ f[1, 0] & f[1, 1] & \dots & f[1, nCols - 1] \\ \vdots & \vdots & \ddots & \vdots \\ f[nRows - 1, 0] & f[nRows - 1, 1] & \dots & f[nRows - 1, nCols - 1] \end{bmatrix} \quad (2.2)$$

2.6.1 Light and reflection

2.6.2 Motivation

Think *monochrome* for the moment. Sometimes we talk of a (monochrome) image as representing a two-dimensional brightness function $f(x, y)$, where x and y denote spatial coordinates, and the value of f at any point (x, y) gives the brightness (or, grey level) at that point.

For this section it would be better to talk of *light intensity* or *lightness* (instead of brightness). Correct terms: *lightness* describes the real physical light intensity, *brightness* is what we *perceive*, that is, brightness is only in the mind.

Think now of the scene as a flat two-dimensional plane – a sheet of coloured paper. Its lightness, $f(x, y)$, is the product of two factors:

- $i(x, y)$ – the illumination of the scene, i.e. the amount of light falling on the scene, at (x, y) ,
- $r(x, y)$ – the reflectance of the scene, i.e. the ratio of reflected light intensity to incident light.

$$f(x, y) = i(x, y)r(x, y) \quad (2.3)$$

Eqn. 2.3 is depicted in Figure 2.9; the amount of light falling on the surface is i ; r is the amount of light that gets reflected. r would be nearly 1 for a very white surface and nearly 0 for a black surface, but, completely white ($r = 1.0$) and completely black ($r = 0.0$) are hard to achieve.

The table below gives some naturally occurring ranges of values of i and r :

Illumination (i)	units
Sunny day at surface of earth	9000
Cloudy day	1000
Full Moon	0.01
Office lighting	100

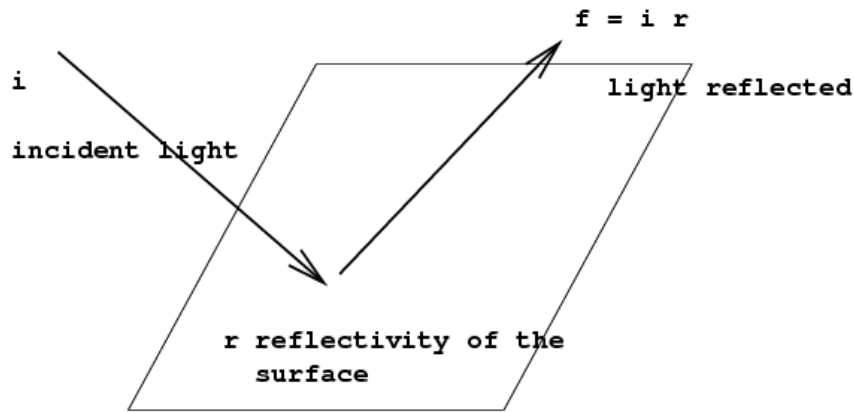


Figure 2.9: Light reflection.

Reflectance (r)	units
Snow	0.93
White paint	0.80
Stainless steel	0.65
Black velvet	0.01

2.6.3 Uneven Illumination

More often than not, when we sense a scene, we want to measure $r(x, y)$, so we assume that $i(x, y)$ is constant i_0 , so that $f(x, y) = r(x, y)i_0$. Thus except for the multiplicative constant, we have $r(x, y)$.

If illumination is *not* constant across the scene, then we have problems disentangling what variations are due to r , and what are due to i .

2.6.4 Uneven Sensor Response

Most modern electronic cameras are charge-coupled device (CCD) based. In a CCD you have a rectangular array of light sensitive devices $i = 0, 1, \dots, n-1$, $j = 0, 1, \dots, m-1$ at the image plane. The voltage given out by these is proportional to the amount of light falling on it.

Often it is assumed that an image $f(x, y)$ arriving at the cameras image plane, is converted into values (analogue or digital), $f_c(x, y)$, which are proportional to $f(x, y)$, i.e.

$$f_c(x, y) = K f(x, y) \quad (2.4)$$

If $K = K(x, y)$, i.e. it varies across the image plane, then we have non-even sensitivity and if we look very closely our image may look a little patchy.

2.6.5 Diffuse and Specular Reflection

The simple model in eqn. 2.3 and Figure 2.9 does not tell the full story. For a start there is colour, but colour can be modelled using three equations, one for red, one for green, and another for blue.

Wavelength (m)		Name	Frequency (Hz)
10^{-15}	1 femtometer (fm)	gamma rays	3×10^{23} Hz
10^{-12}	1 picameter	X-rays	3×10^{20} Hz
10^{-9}	1 nanometer	X-rays	3×10^{17} Hz
10^{-8}	10 nm	Ultraviolet	3×10^{16} Hz
10^{-7}	100 nm	U-V	
4×10^{-7}	400 nm	Visible light (violet)	
7×10^{-7}	700 nm	Visible (red)	
10^{-6}	1 micrometer	Infrared (near)	3×10^{14} Hz
10^{-5}	10 micrometers	Infrared	3×10^{13} Hz
		Infrared (heat)	
10^{-3}	1 millimeter	Infrared (heat) + microwaves	3×10^{11} Hz (300 GHz)
10^{-1}	0.1 meters	microwaves	3×10^9 (3 GHz)
1 meter		TV etc. (UHF)	3×10^8 (300 MHz)
	FM radio is ~ 100 MHz (VHF)		
10 meters		radio (shortwave)	30 MHz
100 meters		radio (shortwave)	3 MHz
200 – 600 m		radio (medium wave)	1.5 MHz to 500 KHz
1500 m (1 Km)		radio (long wave)	200 KHz

Table 2.1: The electromagnetic spectrum.

Things get more complicated when we think about the angle at which the light hits the surface and the angle of the eye or camera that is seeing the reflected light.

There are two models which give a good approximation of reflection from surfaces; these are *diffuse*, also called *Lambertian*, and *specular*.

Diffuse reflection corresponds to *matte* surfaces. Specular reflection corresponds to shiny surfaces (*specular* means *mirror-like*).

We will deal with these in Chapter 6.

2.7 Colour

2.7.1 Electromagnetic Waves and the Electromagnetic Spectrum

Light is a form of energy conveyed by waves of electromagnetic radiation. The radiation is characterised by the length of its wavelength; the range of wavelengths is called the *electromagnetic (EM) spectrum*. Visible light occupies a very small part of the spectrum.

Table 2.7.1 shows the EM spectrum: the left hand column gives the wavelength in meters, the middle gives the name of the band, and the right gives the frequency of the radiation in Hertz (cycles per second). Figure 2.10 (from (Gonzalez & Woods 2002)) shows another view of the EM spectrum.

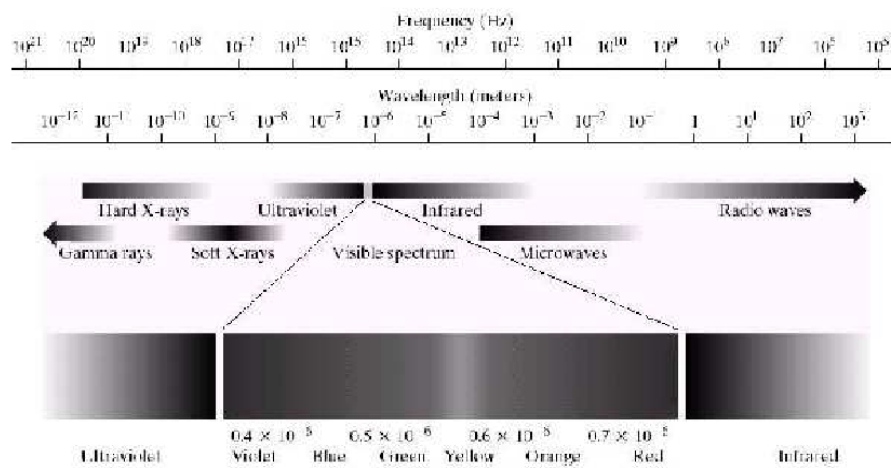


Figure 2.10: Electromagnetic spectrum.

Thus, roughly speaking, if you were to *speed-up* the frequency of vibration of a TV signal, you would get microwaves, speed-up microwaves → heat radiation, → light → UV → X-rays, etc.

If you had a very small and light magnet suspended in a vacuum and brought it near to an RTE TV transmitter, it might start to vibrate at around 600-MHz (600,000,000 times a second); that corresponds to a wavelength of 0.5-metre; for more on UHF (ultra high frequency), see http://en.wikipedia.org/wiki/Ultra_high_frequency.

Frequency, f , wavelength λ and speed of the waves (speed of light), c are related by equation 2.5,

$$f = c\lambda. \quad (2.5)$$

f is measured in Hertz (Hz); λ is measure in metres (m), and $c = 3 \times 10^8$ metres per second (ms^{-1}).

Ex. What is the frequency of yellow light? Assume an average wavelength of 600-nm. Is blue light faster or slower? Which has the smaller wavelength?

Ex. Which has the larger wavelength, a UHF TV signal (e.g. 600-MHz) or yellow light?

It is possible to use various parts of the EM spectrum for imaging: e.g. X-rays, microwaves, infrared (near), and thermal infrared. Our major interest will be in visible light.

2.7.2 The Visible Spectrum

The visible spectrum stretches from about 400-nm to 700-nm. The reason why this part of the spectrum is visible is that the rods and cones in our retinas are sensitive to these wavelengths, and insensitive to the remainder; e.g. if you look at a clothes iron in the dark, you may ‘feel’ the heat radiated from it, but your eyes will not convert that energy into a light sensation; similarly, microwaves and X-rays, they may cause damage, but you will not ‘see’ them.

The overall relative spectral sensitivity of human eyes is shown in Figure 2.11, with approximate indication of corresponding colours. The spectrum of light reaching earth from space, resulting from the blocking effects of the earth’s atmosphere, looks rather similar.

From Figure 2.11 we can see that the eye is very sensitive to radiation in the green-yellow range (peak at 550-nm), and relatively insensitive to blue, violet, and deep red; a blue light around 475-nm (relative sensitivity approx. 10%) would have to put out 10 times more power than the equivalent green-yellow light. Why did the human evolve this way? Well, the energy emitted by the sun (at least that part that reaches the earth) has an energy spectrum graph similar to Figure 2.11.

Figure 2.12 (from (Gonzalez & Woods 2002), chapter 6) shows the relative sensitivity of red, green, and blue cones in the human eye.

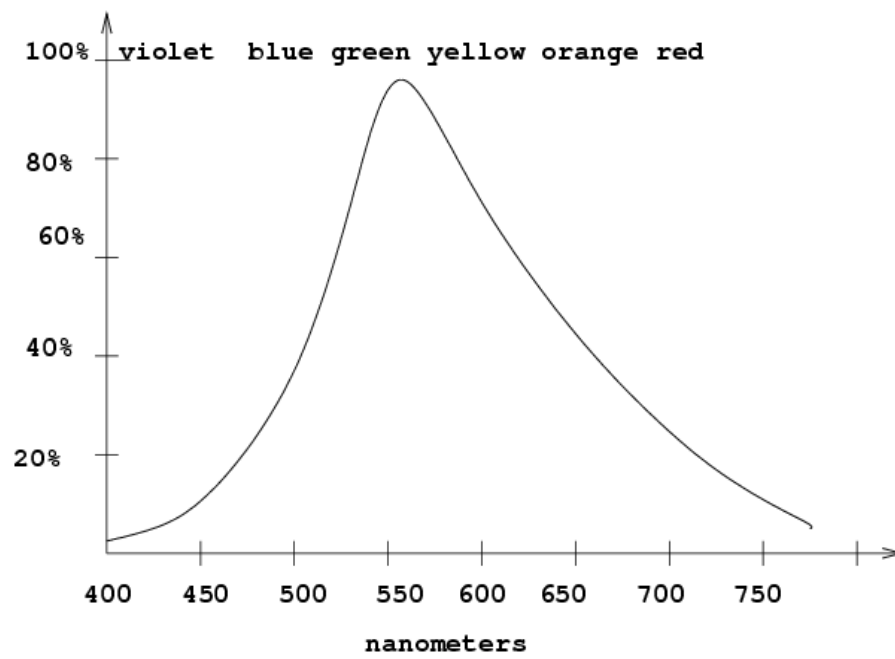


Figure 2.11: Eye overall sensitivity

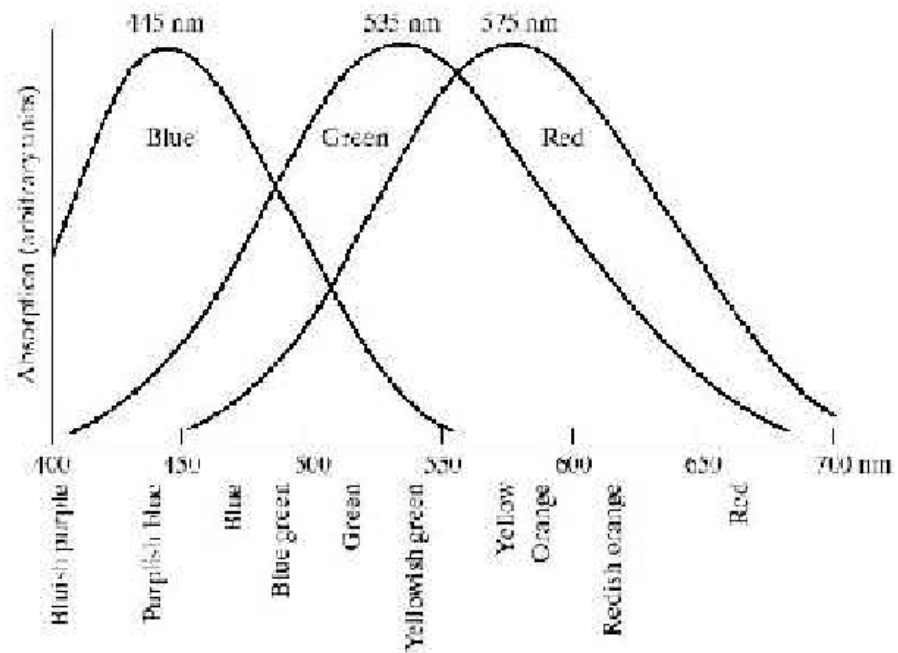


Figure 2.12: Sensitivity of red, green, and blue cones in the human eye.

2.7.3 Sensors

A light sensor is likely to have a similar spectral response curve to Figure 2.11, though usually flatter and wider – i.e. more equally sensitive to wavelengths, and sensitive to UV and to near infrared.

If Figure 2.11 was the spectral response of a sensor, then a blue light (see above), compared to a green-yellow light of the same power, would produce a sensor output of 10% of the voltage of the green-yellow.

2.7.4 Spectral Selectivity and Colour

We have already mentioned that a colour sensor (e.g. in a colour TV camera) is merely three monochrome sensors: one which senses blue, one green, and one red.

What is meant by sensing blue, green, or red? What we do is arrange for the sensor to have an effective response curve that is high in green (for example) and low elsewhere. But, we have already said that sensors have a fairly flat curve (maybe 200–1000-nm), so we must arrange somehow to block out the non-green light.

Wavelength sensitive blocking is done by a colour *filter*. A green filter allows through green light but absorbs the other; similarly blue and red. Figure 2.13 shows the transmittivity (relative amount of light energy allowed to pass through) of a green filter.

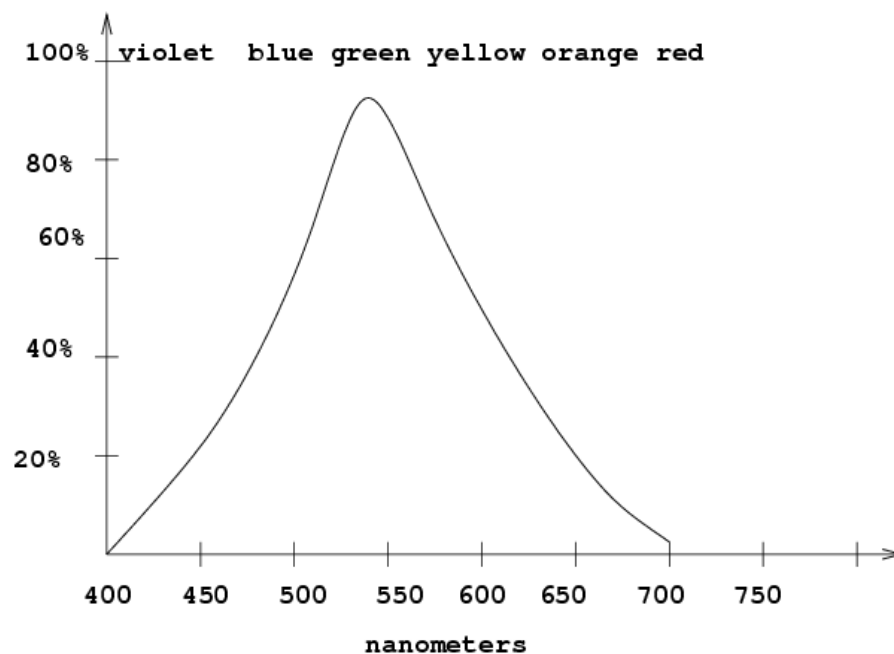


Figure 2.13: Green filter transmittivity.

So, we use three separate sensors, each with its own filter (blue, green, and red) located somewhere between the lens and the sensor.

Considering the effects of the colour selective filters in front of them, the overall sensitivity of red, green, and blue sensors would look something like that of human eye cones, see Figure 2.12.

2.7.5 Spectral Responsivity

The relative response of a sensor can be described as a function of wavelength (forget about (x, y) or (r, c) for the present): $d(\lambda)$, where λ denotes wavelength.

The light arriving through the lens can also be described as a function of λ : $g(\lambda)$, and the overall output is found by integration:

$$\text{voltage} = \int_0^{\infty} d(\lambda)g(\lambda)d\lambda \quad (2.6)$$

Obviously, the integral can be limited to (say) 100-nm to 1000-nm.

If we have a filter in front of the sensor, relative transmittance (the amount of energy it lets through), $t(\lambda)$, then the light arriving at the sensor, $g'(\lambda)$, is the product of $g()$ and $t()$:

$$g'(\lambda) = g(\lambda)t(\lambda) \quad (2.7)$$

and the equation changes to:

$$\text{voltage} = \int_0^{\infty} d(\lambda)g(\lambda)t(\lambda)d\lambda \quad (2.8)$$

or,

$$\text{voltage} = \int_0^{\infty} d(\lambda)g'(\lambda)d\lambda \quad (2.9)$$

2.7.6 Colour Display

So now we have three images stored in memory; how to display them to produce a proper sensation of colour?

Similarly to our model of a colour camera as three monochrome cameras, a colour monitor can be thought of as three monochrome monitors: one which gives out blue light, one green and one red.

A monochrome cathode ray tube display works by using an electron gun to squirt electrons at a fluorescent screen; the more electrons the brighter the image; what controls the amount of electrons is a voltage that represents brightness, say $f_v(r, c)$.

A monochrome screen is coated uniformly with phosphor that gives out white light – i.e. its energy spectrum is similar to Figure 2.11

A colour screen is coated with minute spots of colour phosphor: a blue phosphor spot, a green, a red, a blue, a green, . . . , following the raster pattern mentioned earlier. The green phosphor has a relative energy output like the curve in Figure; the blue has a curve that peaks in the blue, etc. There are three electron guns – one controlled by the blue image voltage (say, $f_b(0, r, c)$), one by the green ($f_g(r, c)$) and one by the red ($f_r(r, c)$). Between the guns and the screen, there is an intricate arrangement called a 'shadow-mask' that ensures that electrons from the blue gun reach only the blue phosphor spots, green $\rightarrow$ green spots, etc.

2.7.7 Additive Colour

If you add approximately equal measures (we are being very casual here, and not mentioning units of measure) of blue light, green light and red light, you get white light. That's what happens on a colour screen when you see bright white: each of the blue, green, and red spots are being excited a lot, and equally. Bring down the level of excitation, but keep them equal, and you get varying shades of grey.

Your intuition may lead you to think of subtractive colour; filters are subtractive: the more filters, the darker; combine blue, green and red filters and you get black. However, with additive colour, the more light added in, the brighter; the more mixture, the closer to grey – and eventually white.

2.7.8 Colour Reflectance

This subsection may be skimmed at the first reading.

All this brings a new dimension to the discussion of illumination and reflectance in section 2.6.1. Now we can think of illumination (i) and reflectance(r) as functions of λ as well as (x, y) .

Thus, the lightness function is now spectral (and therefore a function of λ), i.e.

$f(\lambda, x, y)$ is the product of two factors:

- $i(\lambda, x, y)$ – the spectral illumination of the scene, i.e. the amount of light falling on the scene, at (x, y) , at wavelength λ ,
- $r(\lambda, x, y)$ – the reflectance of the scene, i.e. the ratio of reflected light intensity to incident light

$$f(\lambda, x, y) = i(\lambda, x, y)r(\lambda, x, y) \quad (2.10)$$

Why does an object look green (assuming it is being illuminated with white light)? Simply because its $r(\lambda, ..)$ function is high for λ in the green region (500-550-nm), and low elsewhere (again, see Figure 2.13). Of course, illumination comes into the equation: a white card illuminated with green light (in this case $i(\lambda, ..)$ looks like Figure 2.11) will look green, etc.

2.7.9 Exercises

Ex. 1 A coloured card whose reflectivity is $r(\lambda, x, y)$ is illuminated with coloured light with a spectrum $i(\lambda)$ (constant over spatial coordinates (x, y) ; this is sensed with a camera whose CCD sensor has a responsivity $d(\lambda)$ (again constant over x, y); a filter with transmittance $t(\lambda)$ is used. Show that the overall voltage output is

$$v(x, y) = \int r(\lambda, x, y)i(\lambda)t(\lambda)d(\lambda)d\lambda$$

Ex. 2 A blue card is illuminated with white light; explain the relative levels of output from a colour camera for blue, green, red.

- Ex. 3** A blue card is illuminated with red light; explain the relative levels of output from a colour camera for blue, green, red.
- Ex. 4** A blue card is illuminated with blue light; explain the relative levels of output from a colour camera for blue, green, red. What, if any, will be the change from Ex. 2.5-4 ?
- Ex. 5** A white card is illuminated with yellow light; explain the relative levels of output from a colour camera for blue, green, red.
- Ex. 6** A white card is illuminated with both blue and red lights; explain the relative levels of output from a colour camera for blue, green, red.
- Ex. 7** A blue card is illuminated with both blue and red lights; explain the relative levels of output from a colour camera for blue, green, red; what, if any, will be the change from Ex. 5.

2.8 Cameras and Photographic Film

Many images start off as photographs, so film cannot be ignored. Realise that:

- just like the eye, film is limited in the range of illumination that it can handle;
- a camera adapts by opening / closing the lens diaphragm, – or, by increasing or decreasing exposure time.

2.9 More on Colour Images

We already have seen that colour displays employ red, green, and blue light emitters to create the sensation of colour. Typically, the computer representation of a colour image involves three 'grey level' images; and normally these correspond to redness, greenness and blueness images.

In a colour camera, you have the opposite of the red, green, and blue light emitters; you have red sensitive, green sensitive, and blue sensitive light sensors. Often you have three plain light sensors, one with a red filter in front of it, one with a green filter, and one with a blue filter.

Make sure to remind me to do a demonstration of this in class using a light meter and camera filters.

The figures that follow are from Gonzalez and Woods (Gonzalez & Woods 2002) p. 317.

Figure 2.14 shows a colour image — a dish of strawberries and a cup of coffee on a cream coloured table and in a cream coloured dish and cup; at least the image was colour before it was printed on a monochrome laser printer. I'll let you have full size copies of these so that you can display them and see more clearly what is meant.

Figure 2.15 shows the red, green, and blue components, i.e. what the red, green, and blue images would look like if they were displayed as separate monochrome images. Notice that the berries are quite bright in the red image and the leaves quite bright in the green image; the nearly white tablecloth and cup and dish are quite bright in red, green, and blue; that is to be expected as white contains all colours.



Figure 2.14: Colour image, displayed as colour.



Figure 2.15: Red, green, and blue components of the colour image.

Next, Figures 2.16 to 2.19 show the cyan, magenta, yellow, and black representations of Figure 2.14; the *black* image gives the darkness, i.e. negative of the lightness.

Think colour photography negatives; cyan is sort of the negative of red, magenta the negative of green, and yellow the negative of blue; and black is the equivalent of a monochrome negative.



Figure 2.16: Cyan components of the colour image.



Figure 2.17: Magenta component of the colour image.



Figure 2.18: Yellow component of the colour image.

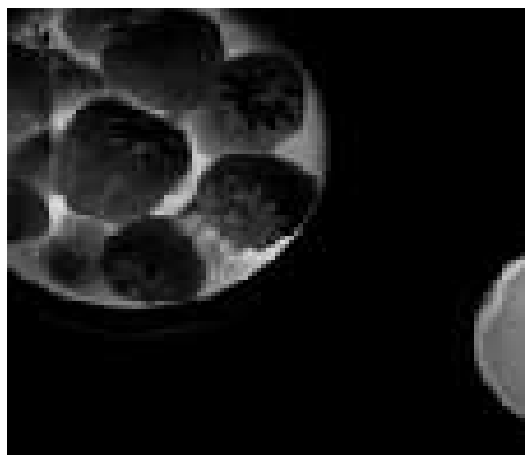


Figure 2.19: Black component of the colour image, i.e. same as a monochrome negative

Instead of explicit colour components, there is another more useful way of representing colour, again three components, Hue, Saturation, Intensity (HSI):

- Hue — a coded value which gives the colour;
- Saturation — a coded value which gives the purity of the colour;
- Intensity — a coded value which gives the overall brightness; this is the same as converting the colour to monochrome.

Figure 2.20 shows the hue, saturation, and intensity image representations of Figure 2.14. Note that the right hand image in Figure 2.20 (intensity, or lightness) is the negative of Figure 2.19 (darkness).



Figure 2.20: Hue, saturation, and intensity; intensity is the same as a monochrome positive.

HSI, or a variation on it, is used in ordinary television transmission. Of course, when you want to display on a CRT or LCD, you have to convert back to RGB.

HSI is useful when you want to do data compression. Our eyes are relatively insensitive to colour variations both spatially and grey level; hence, use high grey level resolution (say 8-bits or 256 levels), and full spatial resolution, for the Intensity part; and use much decreased resolution, e.g. four bits each in grey level, and maybe half spatial resolution, when transmitting the images or storing them to file.

Ex. (a) Take a 256×256 colour image, three bytes per colour. How many bytes? (b) using the scheme mentioned above, i.e. 8-bits for Intensity — how many bytes for the Intensity component? (c) using the scheme mentioned above for Hue, i.e. 4-bits per pixel and spatial resolution halved to yield an $N \times M$ image; what are N and M ? how many bytes for the Hue component? (d) repeat (c) for Saturation; (e) how many bytes for the HSI image? (f) what percentage is the answer to (e) of the answer to (a)?

There is a variation on HSI called YIQ that is used by the JPEG image format.

Chapter 3

Introduction to OpenGL

Most of this is from (Shreiner, Woo, Neider & Davis 2008a) Chapter 1. You would benefit by reading Chapter 1 of (Angel 2005) and Chapters 1 and 2 of (Angel 2008) and the introductory chapters of (Wright, Lipchak & Haemel 2007).

3.1 What is OpenGL?

OpenGL is a 3D graphics application programmers interface (API). It is procedural, by which I mean to say that it is not object-oriented such as you may have become used to.

Here is a brief and incomplete overview of the functions provided by the OpenGL API.

- Specification (modelling) of an arbitrarily complex set of objects in 3D space — creation of a 3D virtual world.
 - The positions of multiple objects are related by transformations (affine), i.e matrices, see (Campbell 2008a), chapters 4–7;
 - Typically, object specifications are based on *vertices* (vertexes if you wish), i.e. *points*, for example `glVertex3f(0.25, 0.5, 0.0)`; specifies a point at $x = 0.25, y = 0.5, z = 0.0$. If your world is 2D, you can work entirely with vertices whose $z = 0$; or, there is a set of `glVertex2*` functions. `glVertex3f` means that the function expects three (3) floats;
 - Colours of object may be specified, for example, `glColor3f(1.0, 0.0, 0.0)`; specifies that all following objects are to be bright red; arguments are (*red, green, blue*) and should be in the range $[0..1]$;
 - We can specify lighting, in which case, the object's appearance when rendered will depend on its own colour, and on the colour and intensity and direction etc. of the light source.
- Specification of a virtual camera by which to view the 3D virtual world.

When the program is executed, OpenGL (i) assembles the virtual world (the scene); (ii) points the virtual camera at the scene; (iii) projects the scene (the part of it that the camera can see) onto

a projection plane, see (Campbell 2008a) Chapter 8; i.e. it simulates a camera, the projection plane corresponds to a virtual version of camera film or array of image sensors in a digital camera, see Chapter 2); (iv) performs the equivalent of spatial sampling and digitisation, see Chapter 2 to produce an image that can be displayed on a computer screen — at least for the meantime, computer screens are 2D! The latter part involves *rasterisation* (converting into discrete arrays) and *clipping* (handling the situation where all of, or part of objects are outside the camera's field of view).

Outside of the true OpenGL API, we need a set of functions which interact with the windowing system provided by the operating system. For this we will use GLUT (GL Utility Toolkit); such functions have the prefix `glut`.

Then there are `glu` (GL utility) functions, which are high level functions, built from elementary OpenGL functions. There are also even higher level `glut` functions for drawing spheres, cubes etc. I have never been able to figure out why these latter functions are `glut`, rather than `glu`.

3.2 Your First OpenGL Program

Let's see an example. This is `hello.cpp` from (Shreiner, Woo, Neider & Davis 2008b) (Red Book) Chapter 1. I like the Red Book examples, they have a consistent and rational architecture (use of `init`, `display`, `reshape`, etc.). Angel's examples (Angel 2005) and (Angel 2008) are usable, but there are careless lapses in the code. I think the OpenGL Superbible code (Wright et al. 2007) is fine, and has the advantage of including Visual-C project files. The code from all these books are in my public folder (`graphics\progs\...`).

Specific examples from these notes, e.g. `hello.cpp`, are also included in (`graphics\progs\chXXcpp`), thus, `graphics\progs\ch03cpp` for this chapter.

Note. In rewriting chapters 3—6 in November 2008, I converted all my example programs to C++; chapters 7 onwards will have to wait for the next revision.

3.2.1 `hello.cpp`

The code in Figure 3.1 draws a white rectangle on a black background, Figure 3.2.

```

/* -----
 * hello.cpp, from hello.c (Red Book);
 * see source for Copyright notice. j.g.c. 2008-11-11
 * This is a simple, introductory OpenGL program.
 *-----*/
#include <GL/glut.h>
#include <cstdlib>

void display(void){
    glClear (GL_COLOR_BUFFER_BIT); // clear all pixels
    /* draw white polygon (rectangle) with opposite corners at
     * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0) */
    glColor3f (1.0, 1.0, 1.0);
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0); glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0); glVertex3f (0.25, 0.75, 0.0);
    glEnd();
    /* don't wait!  * start processing buffered OpenGL routines */
    glFlush ();
}

void reshape(int w, int h){ // w, h are dimensions of the window (see main)
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();    }

void init(void) {
    glClearColor (0.0, 0.0, 0.0, 0.0); /* select clearing color*/
}

/*
 * Declare initial window size, position, and display mode
 * (single buffer and RGB).  Open window with "hello"
 * in its title bar.  Call initialization routines.
 * Register callback function to display graphics and to reshape window.
 * Enter main loop and process events.
 */
int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow ("hello");
    init();
    // register display and reshape --- tell GLUT about them
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return EXIT_SUCCESS;  }

```

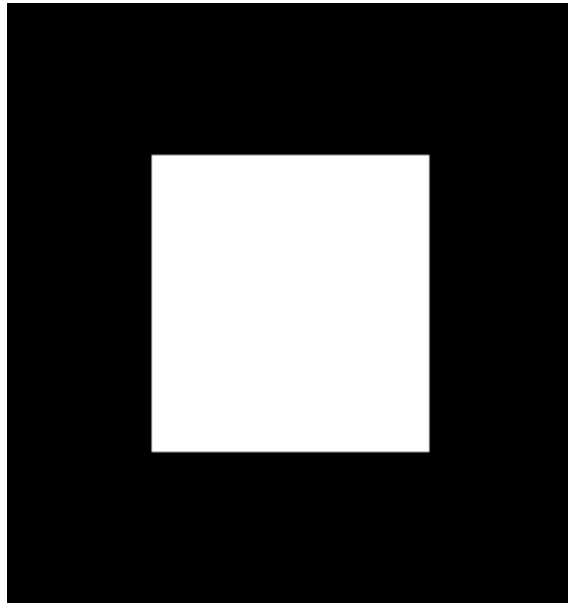


Figure 3.2: White Rectangle.

3.2.2 Dissection of hello.cpp

We start with `main`. This dissection will be much more detailed than later ones.

1. `int main(int argc, char** argv)`. It is possible to pass arguments to a C program from the operating system. `argc` is the number of arguments, `char** argv`, which is the same as `char argv[] []` is an array of *C-strings*, i.e. an array of pointers to `char`, or, equivalently in C++ declarations, *a pointer to a pointer to a char*. If this needs explanation, speak up in class.

`argc` is always greater than or equal to 1, since `argv[0]` contains the name of the program as executed. I'm not sure what happens when you execute from the Visual-C++ IDE; if you want to know, experiment.

Apart from the name of the program as executed, it seems that OpenGL can use following arguments for specification of an X-Window server and its parameters; in other words, the display is either on another machine (connected on the network) or some aspect of X-Window on the local machine needs specification. I doubt if any of this will concern our class.

2. `glutInit(&argc, argv);`. See section 3.3; if there are special windowing system parameters (but there never are in my courses), this is how to pass them to GLUT. Note the `&argc`.
3. `glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);` `GLUT_SINGLE | GLUT_RGB` is the ORing together of two bits which specify that the display is *single buffered* (later we will see use of *double buffering*, see also Chapter 2) and using an *RGB* frame buffer. The alternative to `GLUT_RGB` is `GLUT_INDEXED`, which means *palette*. Given the amount of memory in today's graphics cards, I cannot imagine that we will ever use `GLUT_INDEXED`; `GLUT_RGBA`, where *A* stands for *alpha* (opacity) is also possible.
4. `glutInitWindowSize (250, 250);` specifies a window of size (250, 250) (width, height) in raw display pixels, see section 3.3.
5. `glutInitWindowPosition (100, 100);` specifies that the window is to be placed at ($x = 100, y = 100$), again in raw screen pixel coordinates.
6. Finally, `glutCreateWindow ("hello");`, creates the window and "hello" is the name in the title bar. Quite often you will see `glutCreateWindow (argv[0]);` which takes the name of whatever the program execution name was.
7. `init ()`; simply calls the function defined earlier; we'll dissect and discuss `init` below; for now we'll stick to `main`.
8. `glutDisplayFunc(display);` registers function `display` (defined earlier, but dissected below) as the *callback* function to be called, inside `glutMainLoop`, when a display action is appropriate. The callback (`display` in this case)

Note that `display` is not a special name it could be anything, `RenderScene`, `myDisplay`, ...so long as it is registered using `glutDisplayFunc(func);` and has the signature `void displayFunction(void)`.

The signature of `glutDisplayFunc` is `glutDisplayFunc(void (*func)(void) );`. All function identifiers have the type *pointer-to-function*.

9. When will `display` be called?

- (a) First, when after control passes to `glutMainLoop` and `glutMainLoop` gets round to it.
- (b) Next, any time the window is resized or minimised / maximised, i.e. when the display needs to be repaired or changed, then `reshape` is called.

`reshape` is registered in `glutReshapeFunc(void (*func)(int width, int height));` which allows registration of a callback function to be called when the window is resized. The `glutDisplayFunc` callback is always called *after* the `glutReshapeFunc` callback. Of course, all this is done for us inside `glutMainLoop`. `reshape` will be described in more detail in a later example.

- (c) Finally, if in some situation `glutMainLoop` is hesitant to call the `glutDisplayFunc` callback as soon as is desired, you can drop a hint for `glutMainLoop` to get a move on with `glutPostRedisplay(void)`.

If you want to monitor `glutDisplayFunc` callback calls, add the following code to `hello.c`:

```
int ndisp= 0; /* **global/static variable*/

void display(void){
    cout<< "ndisp = %d\n"<< ndisp<< endl;
    glClear (GL_COLOR_BUFFER_BIT);
    ndisp++;
}
```

10. Nothing OpenGL-related has happened until now. `glutMainLoop()`; starts the GLUT *event loop* and control stays there until the event loop is exited by some special event such as killing the window, typing `ctrl-c`, or, see later chapters, a user specified quit event. See section 3.4.
11. Include files. Normally the following will do.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h> /* add this if you use \verb|printf| etc.
```

`#include <GL/glut.h>` brings in all OpenGL related stuff and anything else OpenGL needs. In spite of the fact you may be on a Windows machine keep the forward slash in `#include <GL/glut.h>`, Windows understands that as a universal, and this will allow you programs to compile on a Linux or Mac platform.

12. Now we deal with `display`. Recall that this is the *callback* function that was registered with the command: `glutDisplayFunc(display)`. This is where the main graphics action takes place.

```
void display(void){
    /* clear all pixels */
    glClear (GL_COLOR_BUFFER_BIT);

    /* draw white polygon (rectangle) with corners at
     * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0) */
}
```

```

    glColor3f (1.0, 1.0, 1.0); /* (red, green, blue) */
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0); glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0); glVertex3f (0.25, 0.75, 0.0);
    glEnd();
    /* don't wait!  * start processing buffered OpenGL routines */
    glFlush ();
}

```

13. `glClear (GL_COLOR_BUFFER_BIT)`. Clear any *color*, i.e. any drawn mark, that currently exists. Other `BUFFER_BITS` that you can clear are: `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`.

Apart from `GL_COLOR_BUFFER_BIT`, the only one that will be commonly used is `GL_DEPTH_BUFFER_BIT`. If you look at the maths. notes (Campbell 2008b) section 8.3.4, eqn. 8.17, you will note that when *projecting* a 3D scene onto a 2D window, OpenGL normally needs to be able to determine when something is behind something else and so need not be displayed. The z' in eqn. 8.17 are placed in the *depth buffer*. Later we will see this in operation. You can clear both buffers together by ORing the indicators: `glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`.

What values these are cleared to is specified in `init`.

14. `glColor3f (1.0, 1.0, 1.0); /* (red, green, blue) */`. This states that *from now on until a later glColor command is issued*, the colour to be used for all drawing is (*red* = 1.0, *green* = 1.0, *blue* = 1.0).

The 3 in the name indicates *three* parameters; a fourth parameter is possible, namely *alpha*. Alpha gives the *opacity*, 1 completely opaque, down to 0 completely transparent, see Chapter 2.

The *f* specifies float parameters; note that C, see section 3.5, does not permit function *overloading*.

As usual, there is a pageful (see (Dave Shreiner (Ed.) and OpenGL ARB 2000)) of ways to specify colour.

The signature of `glColor3f` is `void glColor3f(GLfloat r, GLfloat g, GLfloat b);`. Values below 0 will be set to 0 and values above 1 will be set to 1. If you use a `glColor*` command such as `glColor3b(GLbyte r, GLbyte g, GLbyte b)`, OpenGL will map $-128 \mapsto 0$ and $+127 \mapsto 1.0$. Unless you have good reason to use integer versions, avoid.

15. Float or double? In situations like `glColor3*`, and in `glVertex3*` below, should you use float or double? See section 3.7.

16. `glBegin(GL_POLYGON);` Start a polygon, i.e. starting with the first vertex specified, draw *lines* between successive pairs, and finally join the last with the first. Then *fill* the polygon with the current colour. If you want lines, use `glBegin(GL_LINES);`. If you want points, use `glBegin(GL_POINTS);`. Etc ...

Note that in spite of the indentation, `glBegin(GL_POLYGON);` has so special syntax status like a `for()`.

17. `glVertex3f (0.25, 0.25, 0.0);` specifies a vertex (point) at ($x = 0.25, y = 0.25, z = 0.0$). The 3 in the name indicates *three* parameters; a fourth parameter is possible, namely

w (see *homogeneous coordinates in the maths. notes* but it is rarely a good idea to specify w).

18. `glEnd(GL_POLYGON)`; indicates that the specification of vertexes is complete (for this object). As mentioned above, `glBegin` and `glEnd` are just commands; the C compiler will not worry if you forget one or either of them, but OpenGL will not be able to make sense.
19. `glFlush ()`; tells OpenGL to implement all outstanding commands; think of it as a `cout<< endl`; in C++ or `fflush` in C and to a certain extent `"\n"` in `printf` statements; these flush any outstanding output out to the display.

If you don't use `glFlush ()`; the window may be created but nothing drawn in it until some event occurs such as resizing the window. If you want to avoid scratching your head and wondering what is wrong with your program when all that is missing is `glFlush ()`;, get into the habit of always placing it at the end of the `glutDisplayFunc` callback.

20. Now reshape in which we do anything which needs to be done *every time a window is resized*.

```
void reshape(int w, int h){ // w, h are dimensions of the window (see main)
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}
```

21. `glMatrixMode(GL_PROJECTION)`. There are two *matrix modes* or two matrices: `GL_PROJECTION` which contains the projection matrix, see my *maths. notes* (Campbell 2008b) eqns. 8.18 and 8.23. Effectively, `glMatrixMode(GL_PROJECTION)`; says that we are about to set the zoom on the camera lens (well, sort of, for no camera can perform true *orthographic* projection, see (Campbell 2008b) Chapter 8. Orthographic projection is like having a very long telephoto lens at infinity.

The other matrix is `GL_MODELVIEW` which contains transformation matrices used to specify (*MODEL*) the virtual world and to *position and orient* the virtual camera, (the *VIEW*). Our program did not mention `glMatrixMode(GL_MODELVIEW)`; because we did no `MODELVIEW` transformations: our vertices are all in world coordinates and the camera is doing the equivalent of

The default pointing direction for the camera is along the negative z -axis, and oriented with its top pointing up in the positive y -direction.

22. `glLoadIdentity()`; initialises the current `GL_PROJECTION` matrix to the identity matrix; subsequent matrix commands cause the current matrix to be multiplied by some transformation matrix. Don't rely on `glMatrixMode(GL_PROJECTION)`; or `glMatrixMode(GL_MODELVIEW)`; to initialise your matrices, after all, these commands merely state: *apply any matrix command that follows to the selected matrix*.
23. `glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0)`; specifies an *orthographic* projection of the cube $x = 0.0$ to 1.0 ; $y = 0.0$ to 1.0 ; $z = -1.0$ to 1.0 , see Figure 3.3.

You can imagine everything inside the cube $x = 0.0$ to 1.0 ; $y = 0.0$ to 1.0 ; $z = -1.0$ to 1.0 in Figure 3.3 being projected onto the x - y plane at $z = +1$, or any z for that matter because the projecting rays are parallel to the z -axis.

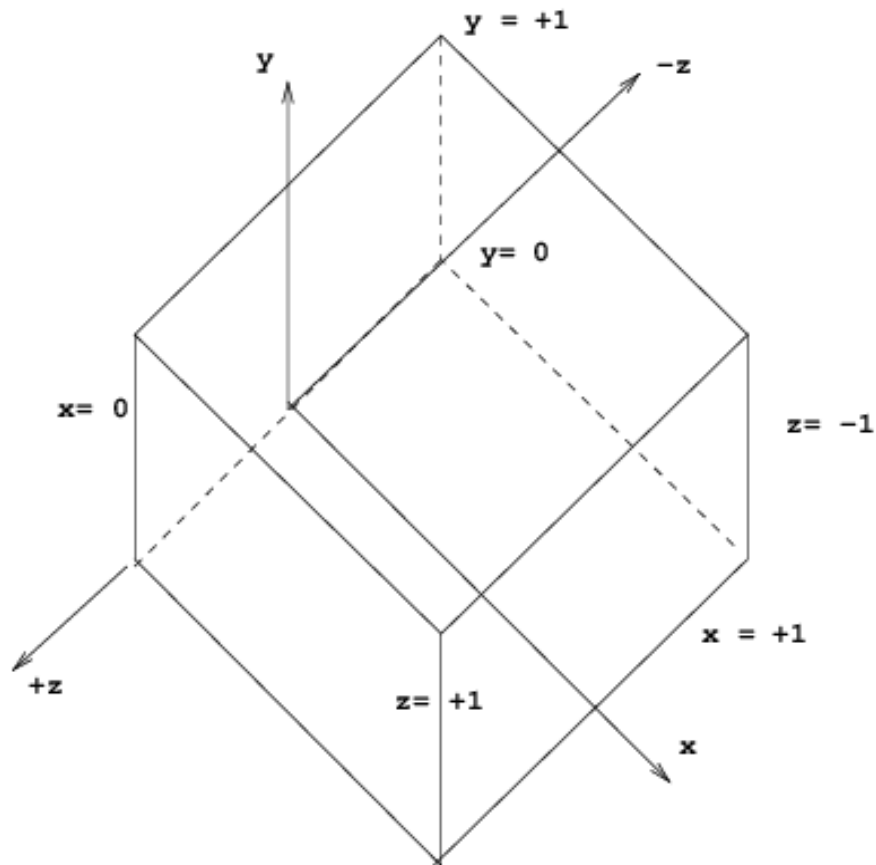


Figure 3.3: `glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0)`.

Of course, we know that the polygon is just a very thin 2D plate ($x = 0.0 - -1.0, y = 0.0 - -1.0$) at $z = 0$, so projection matters even less.

In fact, for entirely 2D graphics, GLU provides a function for 2D ‘projection’ like this:

```
void gluOrtho2D(GLdouble xLeft, GLdouble xRight,
                GLdouble yBottom, GLdouble yTop)
```

See the example in (Angel 2008), page 74.

24. Now `init`. `glClearColor (0.0, 0.0, 0.0, 0.0);`. Black, and transparent, (*red* = 0.0, *green* = 0.0, *blue* = 0.0, *alpha* = 0.0). I don’t know why transparent (*alpha* = 0.0) but that seems to be standard use.
25. What happens now? See Figure 3.4 (Figure 3-2 of (Shreiner et al. 2008a)) shows the the rendering stages.

OpenGL creates the model — the rectangle. That model gets projected onto the x-y plane at $z = +1$; if there are any parts of the rectangle outside, they will be *clipped*. Next we divide by p_z , see eqns. 8.15–8.17 of (Campbell 2008b) to get *normalised device coordinates*. But normalised device coordinates still aren’t display (pixel) coordinates; finally we need to map everything from the *world window* to *viewport window* coordinates, see (Campbell 2008b) section 5.6.9.

Unless it is told otherwise, OpenGL assumes that the *viewport* is the full window — that was created by `glutInitWindowSize (250, 250);`. Later we will see how to explicitly specify a viewport.

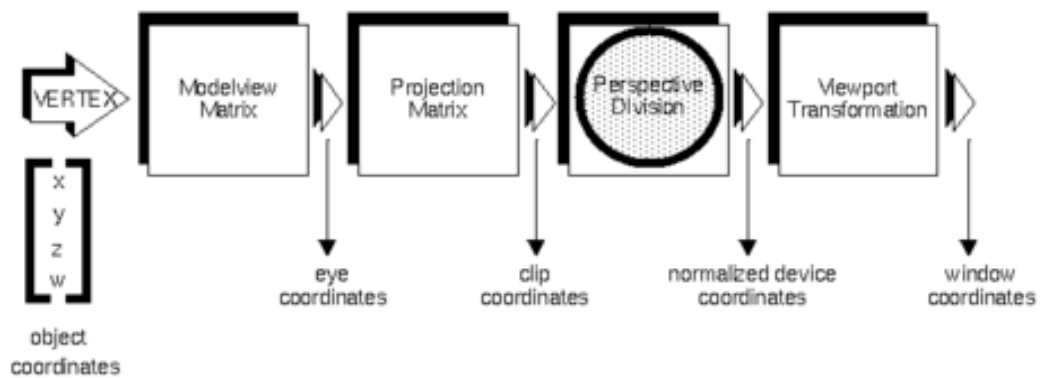


Figure 3.4: Rendering pipeline.

That was a lot of words about a small program. But already we have covered a lot of OpenGL.

3.3 GLUT — GL Utility Toolkit

GLUT provides a minimal set of functions with which to interact with the windowing system provided by the operating system. There are alternatives, but we will use GLUT; such functions have the prefix `glut`.

If you were developing a stand alone application like a game, you would interface directly with Windows. If you want to see examples, (Astle & Hawkins 2004) uses the Windows interface so called *wiggle* (WGL) from the start, and (Wright et al. 2007) introduces it in Chapter 13 and mentions Linux and Mac OS-X interfaces in Chapters 14 and 15.

To keep things simple we stick to GLUT. If you need to use the WGL interface, it's a reasonably straightforward matter to copy the examples in (Astle & Hawkins 2004) and (Wright et al. 2007).

Note that GLUT uses a *raw* raster coordinate system, i.e. raw hardware pixel coordinates, with the *x*-coordinate horizontal and pointing right, and with the *y*-coordinate vertical but pointing down. You need to take account of this when translating mouse coordinates (GLUT, raw) to graphics coordinates.

3.4 Graphic User Interfaces and Event Driven Programming

3.4.1 Introduction

You will be aware of two sorts of *user interface* provided by operating systems:

Command line As provided by MS-DOS.

Graphic User Interface (GUI) As provided by Windows.

The purpose of this section is to explain some of the differences in the implementation of the two.

3.4.2 Command Line Interface

Command line interaction, as in MS-DOS proceeds as follows:

```
begin
    Prompt user;
    Read command;      /* fetch (shell)*/
    Decode command;    /* decode (shell) */
    Do processing;     /* execute (kernel) */
    /* this may involve further prompt, input, process */
    Display result;
end;
```

And if you look at a typical program (to be executed), you will see program code of the same form. In other words, in the program, as in the operating system, everything follows a predictable sequence.

```
begin
    Prompt user;
    Read input ;
    Process input;
    Display result;
end;
```

3.4.3 Graphic User Interface and Events

A little thought will indicate that the *fetch decode execute* scheme described above will not work for a GUI:

- What do you prompt? the user can do any of a large range of actions; the user is in charge, not the program;
- What input device do you read? Mouse, keyboard, other pointing device, ...
- What do you expect to read? A number? A character string?
- etc ...

The basic sequence of a Windows application program, and likewise OpenGL's `glutMainLoop` function is as follows:

```
begin
    Initialise Desktop;
    While (not Done)
        begin
            GetNextEvent(eventMask, theEvent);
```

```

        HandleEvent(theEvent);
    end
    CleanUp;
end

```

The `HandleEvent` looks something like the following:

```

if(theEvent==0) do something;
else if(theEvent==1) do something-else;
else if(theEvent==2) do another-thing;
etc., ...

```

The important point is this: `HandleEvent` (the application part) is driven by the event, not the other way round.

This can allow an interface to be *modeless*: the program identifies the mode from the input. For example, in Microsoft Word, you may type a character, click on the mouse to move the cursor, click on a button, ... – endless possibilities.

What is happening behind the scenes is important. Via interrupts, the operating system captures *events* and maintains an event queue. These events can be requested from the OS by `GetNextEvent(eventMask, theEvent);`, and acted on by `HandleEvent(theEvent);`.

In fact, when you write a Windows program, you will be forced to start with a fixed *framework*. The main programming then is in providing functions (*callbacks*) to handle events 0, 1, 2, etc.

3.5 C or C++?

A great many OpenGL programs are programmed in C; the OpenGL API is programmed in C. However since C++ is compatible with C in the sense that C++ can use C functions, then you *can* use OpenGL in C++ programs.

Given the choice of C or C++, I choose C++ every time. Amongst other factors, C++ is object-oriented, provides stronger type checking, and has a huge and powerful *Standard Library*.

These notes used to contain just C programs; the current version has C++ programs in chapters 3—6, with many of the examples in other chapters being C.

3.6 Visual Studio

I assume that students on the course will use Microsoft Visual Studio Express Edition; Professional Edition is little different. If you want to know how to install OpenGL on a machine (header files, libraries, and DLLs), ask and I'll point you at a website.

Pay attention to the comments above in section 3.5 about `#includes`.

I'm assuming that you are comfortable with creating console applications in Visual Studio. If not, below is something I found somewhere on the web, apologies, source forgotten.

1. Start Visual Studio.
2. Select File -> New -> Project.
3. Select Win32 in Project Types and Win32 Console Application; then Empty Project in Templates view.
4. Enter your project's name.

Choose a sensible location to save in, e.g. `x:\graphics1\projects`

5. Make sure you can see the Solution Explorer, if not go to View -> Solution Explorer, and right click the source Files folder in Solution Explorer. From the menu select Add > New Item.
6. Add a new program file. Type in your code (or cut and paste it in); compile (build); execute (start without debugging).

Lee Burke's guide will be on my website.

There is a longer winded guide at: <http://csf11.acs.uwosh.edu/cs371/visualstudio/>. This is handy in that it tells you where to load OpenGL header files and DLLs and libraries. Be warned though, directories may change a little between Visual Studio 6 and 7; and probably with the new (free) Visual Studio Explorer (based on VS-8).

In cases where the project files originated in Visual Studio 6, your version VS-7 will complain, but it will happily convert.

We'll see how things go in the first practical class. As I have mentioned before, the OpenGL Superbible (Wright et al. 2007) has Visual Studio project files. (Astle & Hawkins 2004) also has Visual Studio project files, but has the distraction that it uses WGL and full Windows applications; for us, Windows programming is just too much of a distraction. I'm open to discussion however if anyone wants some help on it in readiness for a project.

3.7 Float or Double?

For *floating point* we have the choice of `float` or `double`. `float` is 32-bit floating point and is adequate for most purposes; up until ten years ago, people used `float` unless they needed the extra accuracy of `double`, which is 64-bit floating point. At that time (up to ten years ago) `double` not only took up twice the memory space, but considerably slowed arithmetic operations.

Nowadays, however, memories are larger. But most of all, many CPUs prefer to do their arithmetic in `double`. Hence, you may use `float` to speed things up, only to find that the CPU has to convert the `floats` to `doubles`, do the computation, and then convert back — all contributing to a net *slowing down*!

Thus, where I have the choice, I use `double`. However, like the examples in C, I try not to mess about with textbook examples.

3.8 OpenGL Types and Multiple Versions of Commands

Already, we have referred to the multiple versions of commands such as `glVertex*`. Not only can you have the choice of `float`, `double`, `int`, `unsigned int`, etc. but there are array versions, for example:

```
glColor3f(1.0, 0.0, 0.0); /* can be replaced by next two lines */

GLfloat arr[] = {1.0, 0.0, 0.0};
glColor3fv(arr);
```

The suffix 'v' in `glColor3fv` stands for *vector* — unfortunately, *vector* is now taken as a synonym for array.

This means that there is an almost infinite number of `glVertex` commands.

I hope I can rely on you to learn as much as is necessary on what you need, see any of the textbooks, especially (Shreiner et al. 2008b); you might wish to print out the relevant page of that.

Normally, see 3.7, when writing my own programs, I try to default to `double`.

3.9 Animation and Simple Interaction

The code in Figures 3.5 and 3.6 draws a another white rectangle on a black background, but this time can spin the rectangle about its centre. The program is `double.cpp` from (Shreiner et al. 2008b) (Red Book) Chapter 1. Compared to the previous example, `double.cpp` introduces a few new concepts:

- crude animation, including use of the `glutIdleFunc` callback registration;
- interaction via mouse events;
- interaction via keyboard key presses;
- double buffering and `glutSwapBuffers`.

```

/* -----
 * double.cpp (Red Book double.c)
 * j.g.c. 2008-11-11
 * This is a simple double buffered program.
 * Pressing the left mouse button rotates the rectangle.
 * Pressing the middle or right mouse buttons stops the rotation.
 * Keys ESCAPE, Q or q, quits.
 * -----*/

#include <GL/glut.h>
#include <cstdlib>
#include <cstdio>
#include "GLutils.h"

static GLfloat spin = 0.0;

void display(void){
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(spin, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glPopMatrix();

    glutSwapBuffers();
    printGLErrorCode(); // can often tell you where you went wrong
}

void spinDisplay(void){
    spin = spin + 2.0;
    if (spin > 360.0)
        spin = spin - 360.0;
    glutPostRedisplay();
}

void init(void) {
    glClearColor (0.5, 0.5, 0.5, 1.0);
    glShadeModel (GL_FLAT);
}

void reshape(int w, int h){
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

```

Figure 3.5: Double.cpp part 1

```

void mouse(int button, int state, int x, int y) {
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(spinDisplay);
            break;
        case GLUT_MIDDLE_BUTTON:
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(NULL);
            break;
        default:
            break;
    }
}

void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 27: /*ESCAPE*/
        case 'Q':
        case 'q':
            exit(EXIT_SUCCESS);
            break;
    }
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init();
    version();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return EXIT_SUCCESS;
}

```

Figure 3.6: Double.cpp part 2

Dissection of double.cpp We start with main and then move on to the functions.

1. The first novel term is GLUT_DOUBLE. `glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);` specifies *double buffering*, as well as RGB colour as before. Later, we will see `glutSwapBuffers();` which, after drawing to a *back buffer*, requests that the display switch to it; the *back buffer* becomes the *front buffer* and the previous front buffer become the back buffer, ready for the next drawing sequence (hidden from display).
2. `glutCreateWindow (argv[0]);` labels the display window according to the execution command;
3. `glutReshapeFunc(reshape);` registers function `reshape` as the callback for window reshape events.
4. `glutMouseFunc(mouse);` registers function `mouse` as the callback for mouse events.
5. `glutKeyboardFunc(keyboard);` registers function `keyboard` as the callback for keyboard events.

Now to the functions.

6. `static GLfloat spin = 0.0;` Since `spin` must be accessible to two functions, namely `spinDisplay` and `display`, it must be global; `static` is redundant, all global variables are `static`, since `static` means *retain the value between function calls*; but `spin` is outside any function.

Global variables are evil, but it seems that in this case they are a necessary evil.

Global means *global scope / visibility* ; *static* means *static lifetime*. Normally, local variables, e.g. `w` and `h` in `reshape` (yes, parameters are local variables, they are purely local *copies* of the arguments that were passed) are locally visible, i.e. you cannot refer to `w` outside of `reshape`. In addition, local variables have a lifetime / duration corresponding to the time that control is within the function; e.g. `w` and `h` are created when control enters `reshape` and they are deleted when control leaves `reshape`. If you want local scope, but *static* lifetime, you should declare your local variables with the `static` qualifier.

7. Now for `display`.

```
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(spin, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glPopMatrix();

    glutSwapBuffers(); }
```

8. As before, `glClear(GL_COLOR_BUFFER_BIT);` clears anything already in the frame buffer(s).
9. `glPushMatrix();`. Here, we know will be in `glMatrixMode(GL_MODELVIEW);`. You can think of these matrices as OpenGL's way of representing its current position and orientation. Now, we are about to mess around with the `GL_MODELVIEW` matrix, and it is considered correct

and good manners to return position and orientation to where it was before we modified it. `glPushMatrix()` pushes the current matrix onto a stack (hands up anyone who doesn't know what a stack is), simply a way of storing the matrix somewhere safe and where we'll be able to retrieve it.

When you use `glPushMatrix()`; you are saying, I'm going to move about a bit, let me first memorise where I started off, so that I can get back there when I'm done with my moving about (see Red Book for more of this analogy).

If you didn't do `glPushMatrix()`; in this program, you might never notice because all you are doing is rotating about the centre. But if we were translating (moving, shifting), then the translates would be cumulative and with each call to `display` you could be marching towards infinity.

Another thing you may wonder about, `spin` accumulates and wraps around at 360° . Could we use the matrix to store this rotation angle? Probably yes. But generally the *right thing* is to save the matrix on entry to the `display` callback and restore it before return. We'll deal with this in greater detail in a later chapter.

10. `glRotatef(spin, 0.0, 0.0, 1.0);`. `spin` is the angle; N.B. **in degrees**, unlike most conventions which use *radians*. The next three arguments give components of the *axis of rotation* (a vector), in this case the z-axis. The axis of rotation can be any vector, not just one of the basis vectors. More about this in later chapters.
11. `glColor3f(1.0, 1.0, 1.0);`. White.
12. `glRectf(-25.0, -25.0, 25.0, 25.0);`. A rectangle whose top left corner is at ($x = -25, y = -25$) and bottom left corner is at ($x = 25, y = 25$).
13. `glPopMatrix();`. See above. Go back to the position and orientation that we started off with.
14. `glutSwapBuffers();`. We are using double buffering, see Chapter 2; we have just drawn something, so now is the time to reveal the (back) buffer. That is, the display is on another (front) frame buffer while we are drawing.
15. `void spinDisplay(void)` is a callback function registered with `glutIdleFunc(spinDisplay);`, see the mouse callback below.

```
void spinDisplay(void){
    spin = spin + 2.0;
    if (spin > 360.0)
        spin = spin - 360.0;
    glutPostRedisplay();}
```

Essentially, anytime `glutMainLoop` finds time, it calls the `glutIdleFunc` callback, i.e. `spinDisplay`, if it is registered. As we'll see, mouse allows the `glutIdleFunc` callback to be toggled between `spinDisplay` and `NULL` (do nothing).

16. `glutPostRedisplay` gives `glutMainLoop` a nudge to call the `glutDisplayFunc (display)` as soon as possible.
17. `void init(void){`
 `glClearColor (0.0, 0.0, 0.0, 0.0); /*black and transparent*/`
 `glShadeModel (GL_FLAT);}`

18. `glShadeModel (GL_FLAT)`. This says *take the colour of one of the vertices* (the last?) and paint the rectangle all that colour. The alternative is `glShadeModel (GL_SMOOTH)` which interpolates between vertex colours. Since all the vertices are the same colour (white) both would have the same effect in this case.
19. `void reshape(int w, int h)` is the callback registered with `glutReshapeFunc`. It gets called any time we resize a window. `glutMainLoop` will see to it that the current width and height of the window are passed to `w`, `h`.

```
void reshape(int w, int h){
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();}
```

20. `glViewport (0, 0, (GLsizei) w, (GLsizei) h)`. In our first example, by default, the *viewport* was set to be the same size as the (complete) window. This does the same, just explicitly. If you do decide to specify a viewport, the `glutReshapeFunc` is the place to do it.
21. Now specify the projection details. same as last time, only we are including a larger x-y plane.
22. You might wonder what happens if the `glutReshapeFunc` callback never gets called; how will the `GL_PROJECTION` get set? in fact, *reshape* *will* get called, and before *display*, but it seems that you need a `glutPostRedisplay` or `glFlush` to ensure all this happens. I've had the case where a *display without* a `glFlush` seemed to ignore the *reshape* commands (containing a call to `glOrtho`) and this resulted in a (implicit) default of `glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)`; and a lot of head scratching by me.
23. `glMatrixMode(GL_MODELVIEW); glLoadIdentity();`. Assume that the next function will need to be in `GL_MODELVIEW` mode and initialise the matrix. Thus, *reshape* assumes they we now want a fresh start from the beginning.
24. `mouse` is the `glutMouseFunc` callback. Self evident, I hope. `int x`, `int y` are the current positions of the cursor; N.B. not important here, but `x`, `y` are in raw (pixel) window relative coordinates.

If you do not know about enums and `#defines` of constants like `GLUT_LEFT_BUTTON`, bring it up for discussion.

```
void mouse(int button, int state, int x, int y){
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(spinDisplay);
            break;
        case GLUT_MIDDLE_BUTTON:
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(NULL);
```

```

        break;
default:
    break;    }}

```

25. `keyboard` is the `glutKeyboardFunc` callback. Again, self evident, I hope. `int x`, `int y` are the current positions of the mouse (window relative coordinates); I suppose there are interactions that involve both keyboard and mouse, but offhand, I cannot think of one.

```

void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 27: /*ESCAPE*/
        case 'Q':
        case 'q':
            exit(0);
            break;    }}

```

26. `case 27: /*ESCAPE*/`. Normally I'd forbid the use of ASCII numeric codes, but I don't know of any portable way of referring to the ESCAPE key.
27. `case 'Q': case 'q':`. Many programs I use in UNIX / Linux have `q/Q` as *quit*. Gamers may object — maybe `q/Q` are used in some games? If you don't like this, just delete the two lines.

Chapter 4

More 2D Graphics

Here we look at more 2D graphics using programs and descriptions from (Shreiner et al. 2008a) Chapter 2. Refer also as necessary to (Angel 2008).

We present an example program is about drawing lines with stippling (one-dimensional stippling), i.e. dotted and dashed lines, and showing how to draw lines of width different from the default 1.0 (one pixel).

4.1 Points, Lines, and Polygons

This section more or less copied from (Shreiner et al. 2008a) Chapter 2. We have already covered points (*vertices*) in Chapter 3.

4.1.1 Points and Homogeneous Coordinates

- *Vertex* in OpenGL;
- OpenGL internally represents vertices as *homogeneous coordinates*, see Maths. notes (Campbell 2008a), Chapter 6.5, (x, y, z, w) ; we call these 4D, but more correctly they are the *homogeneous coordinates* of 3D *projective space*.
- (x, y, z, w) represents the point $(x/w, y/w, z/w)$;
- w is rarely specified in a program.

4.1.2 Specifying Vertices

Examples of `glVertex*()` from (Shreiner et al. 2008a).

```
glVertex2s(2, 3); /* 2D, so z = 0 */
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0); /* w specified, don't */
```

```
GLdouble dvect[3] = {5.0, 9.0, 1992.0}; /*array*/  
glVertex3dv(dvect);
```

The vector/array form of `glVertex*` may be more efficient because it involves passing just one pointer, instead of two or three coordinate values.

4.1.3 Lines

In OpenGL, *line* means *straight line segment*, i.e. it is between two vertices; this is in contrast to another interpretation of *line* as an infinite line.

...connected sequences of lines ... open or closed ...

4.1.4 Polygons

Polygons are the areas enclosed by single closed loops of line segments. OpenGL restricts what is a valid polygon, see Figure 4.1.



Figure 4.1: Valid Polygons.

4.1.5 OpenGL Geometric Drawing Primitives

...bracket each set of vertices between a call to `glBegin` and a call to `glEnd`. The argument passed to `glBegin()` determines the type of geometric primitive is constructed from the vertices. For example, the code in Figure 4.2 specifies the vertices for polygon shown in Figure 4.3; the right hand side of Figure 4.3 shows what would be drawn if `glBegin(GL_POLYGON)` is replaced by `glBegin(GL_POINTS)`.

Figure 4.4 shows the list of possible arguments for `glBegin` and Figure 4.5 gives a diagrammatic explanation.

```

glBegin(GL_POLYGON); /*glBegin(GL_POINTS); for right hand figure*/
    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 3.0);
    glVertex2f(3.0, 3.0);
    glVertex2f(4.0, 1.5);
    glVertex2f(3.0, 0.0);
glEnd();

```

Figure 4.2: Polygon or points.

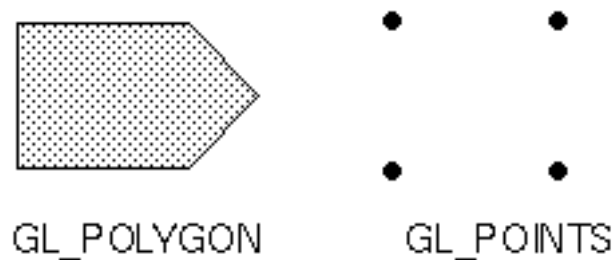


Figure 4.3: Polygon or points.

4.2 Displaying Points, Lines, and Polygons

By default, a point is drawn as a single pixel on the screen, a line is drawn solid and one pixel wide, and polygons are drawn solidly filled in. The following paragraphs discuss the details of how to change these default display modes.

4.2.1 Point Details

`glPointSize`. Size in pixels as the argument: `void glPointSize(GLfloat size);`; sets the width in pixels for rendered points; size must be greater than 0.0 and by default is 1.0.

Anti-aliasing — see Chapter 2.

4.2.2 Line Details

Lines with different widths and lines that are stippled in various ways - dotted, dashed, drawn with alternating dots and dashes, and so on. Line width can be changed using:

`void glLineWidth(GLfloat width);` sets the width in pixels for rendered lines; width must be greater than 0.0 and by default is 1.0.

Again note *anti-aliasing*.

GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_POLYGON	boundary of a simple, convex polygon
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUAD_STRIP	linked strip of quadrilaterals

Figure 4.4: Geometric Primitive Names and Meanings.

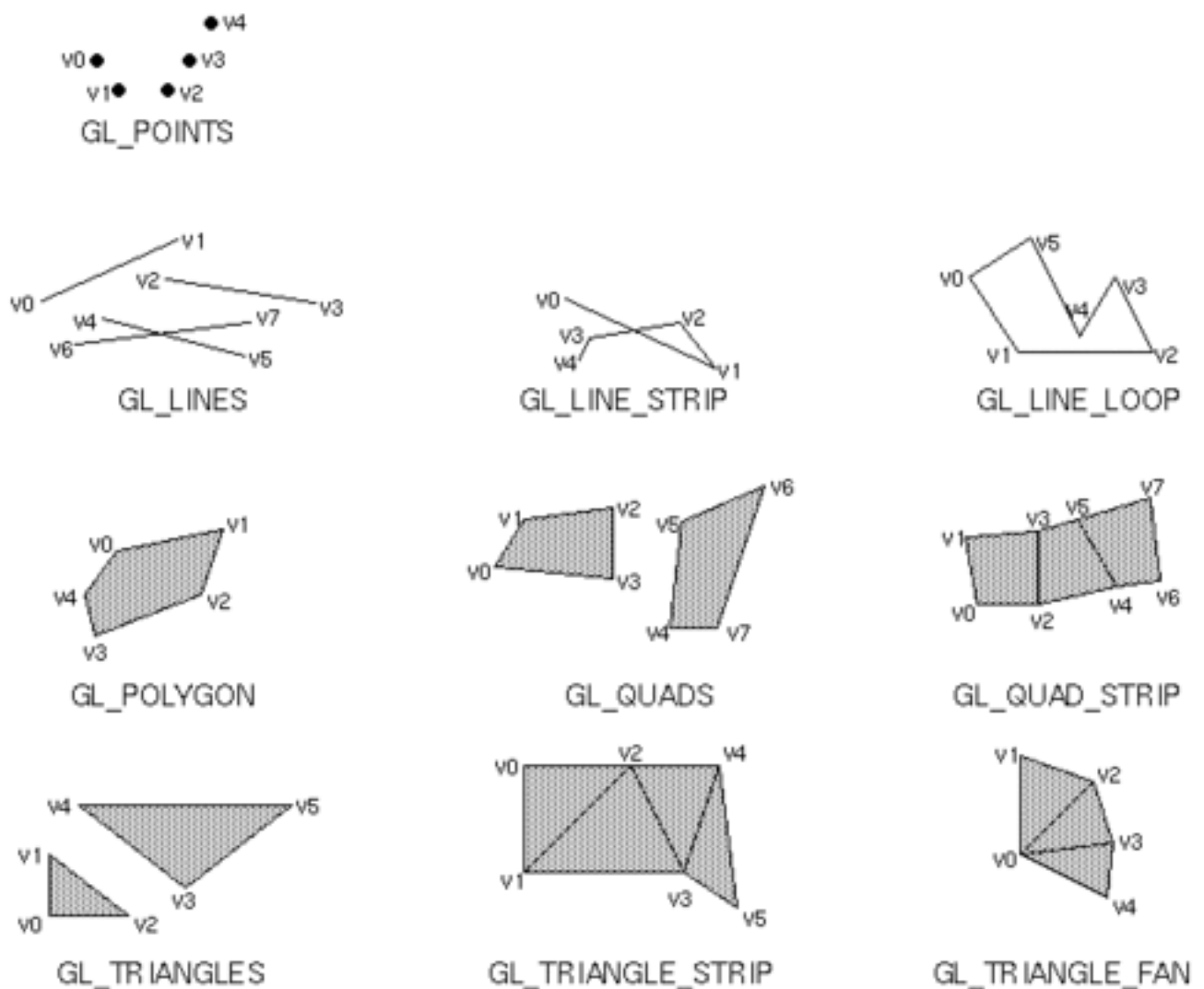


Figure 4.5: Geometric Primitive Types.

4.3 Drawing Lines — a program example

The program `lines.c` shown in Figures 4.8 and 4.9 draws the lines shown in Figure 4.6.

Make sure the that following are *outside* (enclosing) any drawing command to which the stippling refers; otherwise unpredictable results. `glEnable (GL_LINE_STIPPLE);`
`glDisable (GL_LINE_STIPPLE);`

I'm assuming that you are familiar with hexadecimal, 0x00FF etc. If not, make sure to bring it up in class.



Figure 4.6: Stippled lines from `lines.c`.

PATTERN	FACTOR	
0x00FF	1	_____
0x00FF	2	_____
0x0C0F	1	____ _
0x0C0F	3	_____
0xAAAA	1	- - - - -
0xAAAA	2	____ _
0xAAAA	3	____ _
0xAAAA	4	____ _

Figure 4.7: Stippling patterns.

```

/*
 * lines.cpp, from Red Book lines.c
 * This program demonstrates geometric primitives and
 * their attributes.
 * from OpenGL Red Book, for copyright see master directory.
 * j.g.c. 2006-01-28, 2008-11-11
 */
#include <GL/glut.h>
#include <cstdlib>

void drawOneLine(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2){
    glBegin(GL_LINES);
        glVertex2f ((x1),(y1));
        glVertex2f ((x2),(y2));
    glEnd();
}

void init(void) {
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void display(void){
    int i;

    glClear (GL_COLOR_BUFFER_BIT);
    /* select white for all lines */
    glColor3f (1.0, 1.0, 1.0);

    /* in 1st row, 3 lines, each with a different stipple */
    glEnable (GL_LINE_STIPPLE);

    glLineStipple (1, 0x0101); /* dotted */
    drawOneLine (50.0, 125.0, 150.0, 125.0);
    glLineStipple (1, 0x00FF); /* dashed */
    drawOneLine (150.0, 125.0, 250.0, 125.0);
    glLineStipple (1, 0x1C47); /* dash/dot/dash */
    drawOneLine (250.0, 125.0, 350.0, 125.0);

    /* in 2nd row, 3 wide lines, each with different stipple */
    glLineWidth (5.0);
    glLineStipple (1, 0x0101); /* dotted */
    drawOneLine (50.0, 100.0, 150.0, 100.0);
    glLineStipple (1, 0x00FF); /* dashed */
    drawOneLine (150.0, 100.0, 250.0, 100.0);
    glLineStipple (1, 0x1C47); /* dash/dot/dash */
    drawOneLine (250.0, 100.0, 350.0, 100.0);
    glLineWidth (1.0);
    // ... continued

```

Figure 4.8: Lines with stippling, lines.cpp, Part 1.

```

/* in 3rd row, 6 lines, with dash/dot/dash stipple */
/* as part of a single connected line strip */
glLineStipple (1, 0x1C47); /* dash/dot/dash */
glBegin (GL_LINE_STRIP);
for (i = 0; i < 7; i++)
    glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
glEnd ();

/* in 4th row, 6 independent lines with same stipple */
for (i = 0; i < 6; i++) {
    drawOneLine (50.0 + ((GLfloat) i * 50.0), 50.0,
        50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
}

/* in 5th row, 1 line, with dash/dot/dash stipple */
/* and a stipple repeat factor of 5 */
glLineStipple (5, 0x1C47); /* dash/dot/dash */
drawOneLine (50.0, 25.0, 350.0, 25.0);

glDisable(GL_LINE_STIPPLE);
glFlush();
}

void reshape(int w, int h){
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}

void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 27:
            exit(EXIT_SUCCESS);      break;
    }
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (400, 150);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return EXIT_SUCCESS;
}

```

4.4 Details on Polygon Rendering

Default: fill in all the pixels enclosed within the boundary. Solidly filled, or stippled with a certain pattern, see (Shreiner et al. 2008a) Chapter 2.

Also *outlined polygons*, or as points at the vertices.

If adjacent polygons share an edge or vertex, the pixels making up the edge or vertex are drawn exactly once — they're included in only one of the polygons.

4.4.1 Polygons as Points, lines, or Solids

A polygon has two sides, front and back; these may be rendered differently depending on which side is facing the viewer. ...allows cutaway views of solid objects in which there is an obvious distinction between the parts that are inside and those that are outside ...

By default, both front and back faces are drawn in the same way. To change this, or to draw only outlines or vertices, use

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

which specifies the drawing mode for a polygon's front and back faces.

face: GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK;

mode: GL_POINT, GL_LINE, or GL_FILL, which indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled.

Examples:

```
glPolygonMode(GL_FRONT, GL_FILL); /*front faces filled*/  
glPolygonMode(GL_BACK, GL_LINE); /*back faces outlined*/
```

See next subsection for the algorithm which determines *front-facing* / *back-facing*.

4.4.2 Reversing and Culling Polygon Faces

Default: polygons whose vertices appear in counterclockwise order on the screen are considered front-facing.

Change using: `void glFrontFace(GLenum mode);`,

which specifies how front-facing polygons are determined. As mentioned, by default, `mode` is `GL_CCW`, which corresponds to a counterclockwise orientation of the ordered vertices of a projected polygon *in window coordinates*. If `mode` is `GL_CW`, faces with a clockwise orientation are considered front-facing.

Front-back algorithm Front- or back-facing depends on the sign of the polygon's area computed in window coordinates. We can compute this area as:

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i \oplus 1} - x_{i \oplus 1} y_i, \quad (4.1)$$

where $i \oplus 1 = (i + 1) \bmod n$ and where x_i and y_i are the *window* coordinates of the i th vertex of the n vertex polygon. $i \oplus 1 = (i + 1) \bmod n$ simply wraps around back to zero at $i = n$, i.e. there is no $i = n$, so we go back to the first (zeroth index) point.

If GL_CCW has been specified, if $a > 0$, the polygon is considered to be front-facing; otherwise, it is back-facing. If GL_CW is specified and if $a < 0$, then the corresponding polygon is front-facing; otherwise, it's back-facing.

Discarding front or back faces during rendering To specify discarding (*culling*) front or back faces:

```
glEnable(GL_CULL_FACE);

void glCullFace(GLenum mode);
glCullFace(GL_FRONT); /*or */
glCullFace(GL_BACK);

glDisable(GL_CULL_FACE); /* to switch off*/
```

4.5 Stippling Polygons

There is a good example of filling a polygon with a stippled pattern in (Shreiner et al. 2008a) Chapter 2; this provides a sort of *poor man's* texturing. I don't think we need to cover that.

4.6 OpenGL and SDL

If you wanted to write a game using OpenGL, you might find that the GLUT user interface functions are not up to the mark. Simple DirectMedia Layer (SDL) is a (simple and open-source) games API that interacts well with OpenGL. You might let OpenGL do the 3D graphics, and use SDL for user input and for sound.

Just in case you ever need to use SDL, I include a small example here.

For links on SDL, see <http://www.jgcampbell.com/links/sdl.html>. http://lazyfoo.net/SDL_tutorials/ has a very good set of tutorials. The example here is a modified version of Lazyfoo's OpenGL-SDL example.

I will not write notes on SDL, but at least the code here will serve as a simple example and something we can discuss in class, if necessary.

I have ported Brackeen's Java platform game (first year) to C++ using SDL; let me know if you want a copy of that, but I note that while the code *works*, parts of it are *hacked* and not a pretty sight!

The example shows how to move a simple shape using the cursor keys.

Figures 4.10, 4.11 show the main program. Figures 4.12 and 4.13 show the `Shape` class. This software will be available in the chapter 4 programs on my public folder and on my website (in due course).

```

/* ----- oglsdl1.cpp -----
   j.g.c. 2008-10-30
   based on Lazy Foo' Productions Lesson36
----- */
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>
#include <GL/glut.h>
#include <string>
#include <iostream>
#include "Shape.h"

using namespace std;

void handleGLError(){
    GLenum errCode;
    const GLubyte *errString;

    if((errCode = glGetError()) != GL_NO_ERROR){
        errString = gluErrorString(errCode);
        cerr<< "OpenGL error: "<< errString<< endl;
        exit(1);
    }
}

void GLinit(int sw, int sh){
    glClearColor( 0, 0, 0, 0 );

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho( 0, sw, sh, 0, -1, 1 );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    handleGLError(); // exits if any error
}

void SDLUtilsErr(std::string message){
    std::cerr<< "Error: "<< message<< ' '<< SDL_GetError()<< std::endl;
    exit(1);
}

void SDLUtilsStatus(std::string message){
    std::cout<< message<< " SDL_GetError: "<< SDL_GetError()<< std::endl;
} // ... continued

```

Figure 4.10: SDL-OpenGL main program, oglsdl1.cpp

```

void SDLInit(int screenWidth, int screenHeight, int screenBpp,
             std::string progName){
    if(SDL_Init(SDL_INIT_EVERYTHING) == -1)SDLUtilsErr("SDL_init");
    if((SDL_SetVideoMode(screenWidth, screenHeight, screenBpp,
        SDL_OPENGL)) == NULL)SDLUtilsErr("SDL_SetVideoMode");
    SDL_WM_SetCaption(progName.c_str(), NULL);
    GLinit(screenWidth, screenHeight);
    atexit(SDL_Quit);
}

int main(int argc, char *argv[]){
    // desired frame rate
    const int fps = 50;
    const int framePeriod = int(1000.0/fps); // desired millisec per frame
    int sw = 800, sh= 600, sbpp = 32;
    SDLInit(sw, sh, sbpp, "First SDL-OpenGL program");

    int w = 20, h = 20;
    Shape s(sw, sh, w, h);

    SDL_Event event;
    bool quit = false;  bool esc = false;
    while(!quit){
        int frameStart = SDL_GetTicks();
        while( SDL_PollEvent( &event ) ){
            //Handle key presses
            s.handleInput(event);
            if(event.type == SDL_KEYDOWN){
                esc = (event.key.keysym.sym == SDLK_ESCAPE);
            }

            quit = esc || (event.type == SDL_QUIT);
        }
        s.move();
        glClear( GL_COLOR_BUFFER_BIT );

        //Cap frame rate, i.e. hang around until we've used up a frame period
        while(true){
            int timeNow = SDL_GetTicks();
            int timeUsed = timeNow - frameStart;
            if(timeUsed> framePeriod) break;
        }

        s.display();
        SDL_GL_SwapBuffers();

    }
    SDL_Quit();  return 0;
}

```

Figure 4.11: SDL-OpenGL main program, oglsdl1.cpp

```

/* ----- Shape.h -----
   shape for moving in SDL/OpenGL
   based on Lazy Foo' Productions Lesson 36  j.g.c. 2008-10-30
   ----- */
#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>

class Shape{
private:
    int sw_, sh_; // window dimensions
    int x, y;      //shape position
    int w_, h_;    //shape dimensions
    int dx_, dy_; //velocity

public:
    Shape(int sw, int sh, int w = 0, int h = 0);
    void handleInput(SDL_Event &event);
    void move();
    void display();
};

Shape::Shape(int sw, int sh, int w, int h)
    : sw_(sw), sh_(sh), x(0), y(0), w_(w), h_(h),
      dx_(0), dy_(0){}

// adjusts velocity based on cursor keys
void Shape::handleInput(SDL_Event &event){

    if(event.type == SDL_KEYDOWN){
        switch( event.key.keysym.sym ){
            case SDLK_UP: dy_ -= h_ / 2; break;
            case SDLK_DOWN: dy_ += h_ / 2; break;
            case SDLK_LEFT: dx_ -= w_ / 2; break;
            case SDLK_RIGHT: dx_ += w_ / 2; break;
            default: break;
        }
    }
    else if( event.type == SDL_KEYUP ){
        switch( event.key.keysym.sym ){
            case SDLK_UP: dy_ += h_ / 2; break;
            case SDLK_DOWN: dy_ -= h_ / 2; break;
            case SDLK_LEFT: dx_ += w_ / 2; break;
            case SDLK_RIGHT: dx_ -= w_ / 2; break;
            default: break;
        }
    }
} // ... continued

```

Figure 4.12: Shape class, Shape.h, part 1

```

void Shape::move(){
    x += dx_;
    if((x < 0 ) || ( x + w_ > sw_)){
        x -= dx_;
    }

    y += dy_;
    if((y < 0 ) || ( y + h_ > sh_)){
        y -= dy_;
    }
}

void Shape::display(){
    glPushMatrix();
    //Move to position
    glTranslatef((GLfloat)x, (GLfloat)y, 0 );

    glBegin(GL_QUADS);
        glColor4f(1.0, 1.0, 1.0, 1.0);
        // draw shape
        glVertex3i(0, 0, 0 );
        glVertex3i(w_, 0, 0 );
        glVertex3i(w_, h_, 0 );
        glVertex3i(0, h_, 0 );
    glEnd();

    glPopMatrix();
}

```

Figure 4.13: Shape class, Shape.h, part 2

Chapter 5

Introduction to 3D Graphics

Here we take a first look 3D graphics using programs and descriptions from (Shreiner et al. 2008a) Chapter 3. Refer also as necessary to (Angel 2008) Chapter 4. At the end of the chapter we introduce some additional new things: (a) how to write and use a menu callback function; (b) how to write and use a timer callback function; (c) how to check for OpenGL errors; (d) how to check which version of OpenGL you are running; (d) (a) how to write and use a mouse motion callback function.

5.1 Your first 3D program, cube.cpp, a wireframe cube

The program `cube.cpp`, parts of which are shown in Figures 5.2 and 5.3 draws the wireframe cube shown in Figure 5.1.

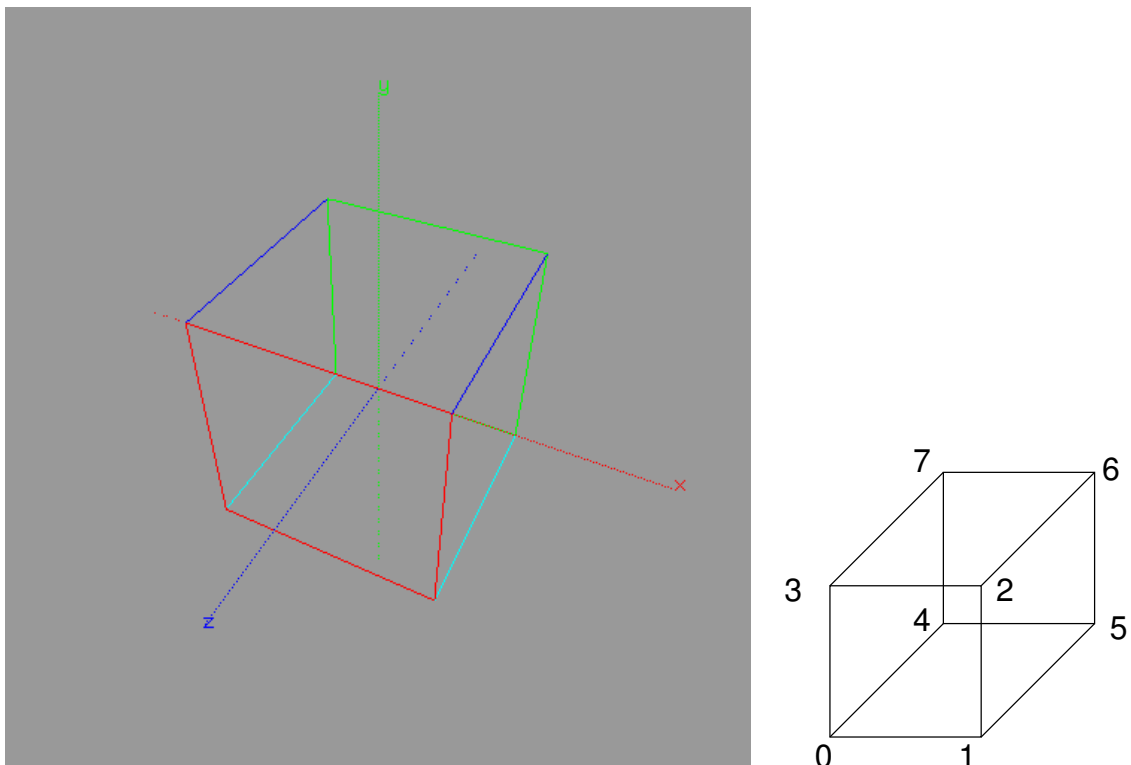


Figure 5.1: (a) Wireframe cube; (b) Vertex numbering used in the program.

```

void wireCube(GLint vi[][3], GLfloat s){
    GLfloat vf[8][3];
    GLfloat s2 = s*0.5f;  // each side is 1.0
    // scale to size, could use glScale, but this is clearer here
    for(int v = 0; v < 8; v++){
        for(int d = 0; d < 3; d++){
            vf[v][d] = GLfloat(vi[v][d])*s2;
        }
    }

    glBegin(GL_LINE_LOOP);
    // front
    glColor3f(1.0, 0.0, 0.0);
    glVertex3fv(vf[0]);
    glVertex3fv(vf[1]);
    glVertex3fv(vf[2]);
    glVertex3fv(vf[3]);
    glEnd();
    glBegin(GL_LINE_LOOP);
    // back
    glColor3f(0.0, 1.0, 0.0);
    glVertex3fv(vf[4]);
    glVertex3fv(vf[7]);
    glVertex3fv(vf[6]);
    glVertex3fv(vf[5]);
    glEnd();

    glBegin(GL_LINES);
    // top
    glColor3f(0.0, 0.0, 1.0);
    glVertex3fv(vf[3]);
    glVertex3fv(vf[7]);
    glVertex3fv(vf[2]);
    glVertex3fv(vf[6]);
    // bottom
    glColor3f(0.0, 1.0, 1.0);
    glVertex3fv(vf[0]);
    glVertex3fv(vf[4]);
    glVertex3fv(vf[1]);
    glVertex3fv(vf[5]);
    glEnd();
}

```

Figure 5.2: Wireframe cube, cube.cpp, cube drawing part.

```

void display(void){
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(fov, (GLfloat)ww/ (GLfloat)hh, 1.0, 20.0);
    //glFrustum (-0.7, 0.7, -0.7, 0.7, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();

    //eye[0] = 1.0; eye[1] = 2.0; eye[2] = 2.0;
    gluLookAt(eye[0], eye[1], eye[2], // camera centre
              0.0, 0.0, 0.0,          // pointing AT
              0.0, 1.0, 0.0);        // up vector

    double xNeg= -1.2, xPos= 1.2, yNeg = -1.2, yPos = 1.2,
           zNeg = -1.2, zPos = 1.2;
    drawAxes(xNeg, xPos, yNeg, yPos, zNeg, zPos);

    glPushMatrix();
    glRotatef((GLfloat)roty,0.,1.,0.);
    glRotatef((GLfloat)rotx,1.,0.,0.);
    //glScalef(1.0, 2.0, 1.0);
    solidCube(vv, 1.0);
    //wireCube(vv, 1.0);
    glPopMatrix();

    glutSwapBuffers();
    printGLErrorCode();
}

```

Figure 5.3: Wireframe cube, cube.cpp, display.

Dissection of `cube.cpp`

1. First have a look at Figure 5.4 which explains the concepts of modelling, viewing, projection, and viewport using a camera analogy. Maybe not so good if you don't already know about cameras and projectors; I'll bring a camera to the class and see if I can help people in that position.

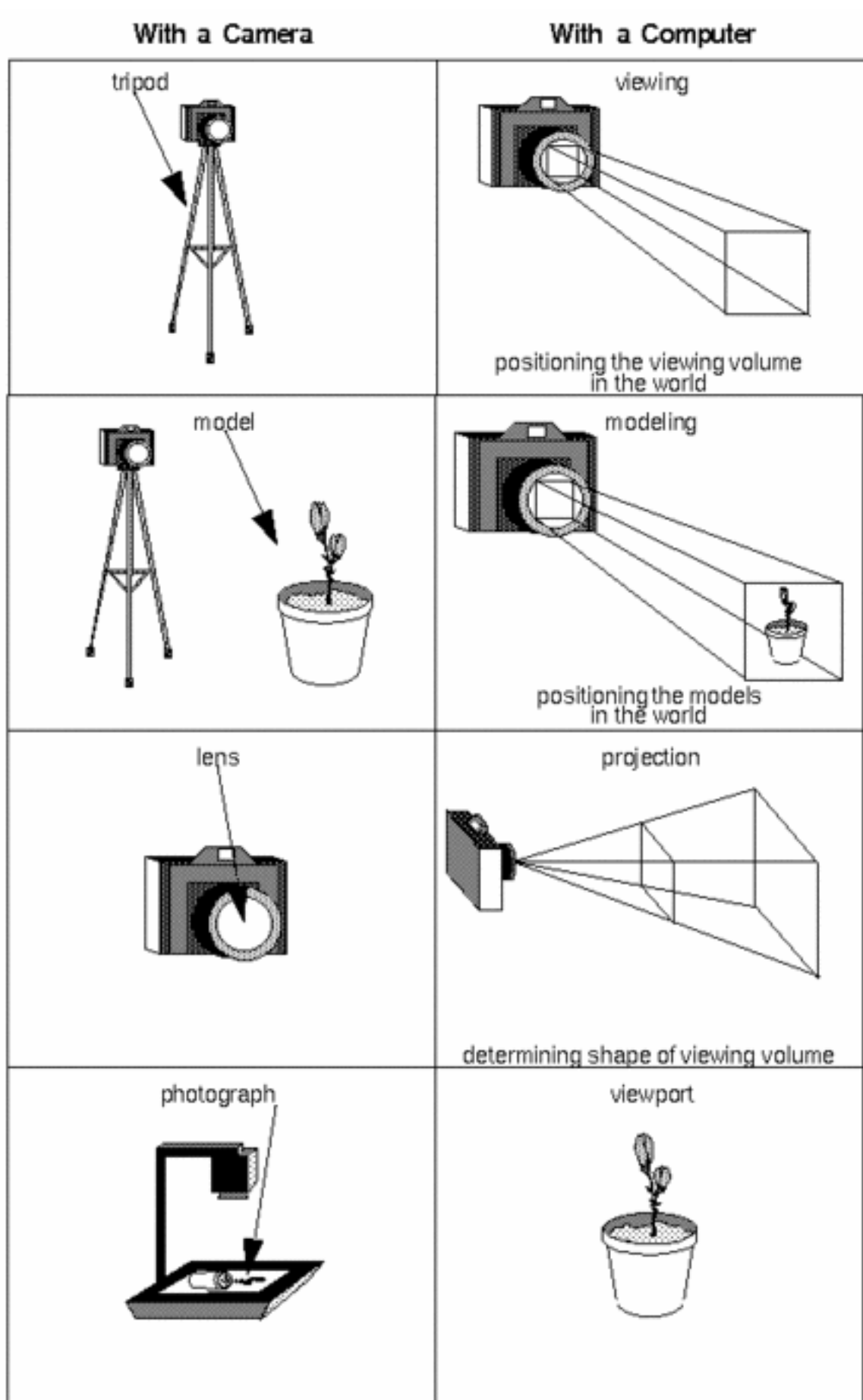


Figure 5.4: Modelling, viewing, projection and viewport.

- Typically, projection takes place in `reshape`; however in this program we want to be able to alter the projection *field-of-view* (fov) in `display` — so that the effect of the change can be made via `glutPostRedisplay`.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity ();
gluPerspective(fov, (GLfloat)ww/ (GLfloat)hh, 1.0, 20.0);
// glFrustum is an alternative to gluPerspective
//glFrustum (-0.7, 0.7, -0.7, 0.7, 1.0, 20.0);
```

`glFrustum` and `gluPerspective` are fully described below in section 5.3.

I warn again. **Do not confuse viewing with projection!** Projection is concerned with choosing the camera lens and with additional specification of *clipping* planes.

- fov gives the *vertical* (y) field-of-view of the camera, i.e. it gives how wide the equivalent leans angle is.
- The second argument of `gluPerspective` is *aspect ratio*; this gives the multiplier used to calculate the horizontal (x) field-of-view. Ordinarily aspect ratio is 1.0, but `(GLfloat)ww/ (GLfloat)hh` is a trick to ensure that when the viewport/window is not square this is compensated for so that graphics objects do not appear distorted by becoming elongated along the vertical or horizontal axes. As an exercise, replace the `(GLfloat)ww/ (GLfloat)hh` with 1.0 and see what happens when you reshape the window.
- This is not cast in stone, but typically modelling and viewing take place in `display`.

```
void display(void){
    // ... code removed
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();

    //eye[0] = 1.0; eye[1] = 2.0; eye[2] = 2.0;
    gluLookAt(eye[0], eye[1], eye[2], // camera centre
              0.0, 0.0, 0.0,          // pointing AT
              0.0, 1.0, 0.0);         // up vector

    double xNeg= -1.2, xPos= 1.2, yNeg = -1.2, yPos = 1.2,
           zNeg = -1.2, zPos = 1.2;
    drawAxes(xNeg, xPos, yNeg, yPos, zNeg, zPos);

    glPushMatrix();
    glRotatef((GLfloat)roty,0.,1.,0.);
    glRotatef((GLfloat)rotx,1.,0.,0.);
    //glScalef(1.0, 2.0, 1.0);
    //solidCube(vv, 1.0);
    wireCube(vv, 1.0);
    glPopMatrix();
}
```

- Viewing* is positioning, pointing and orienting the virtual camera. `gluLookAt` does that.

```
gluLookAt (eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
```

eyeX, eyeY, eyeZ give the location of the camera / eye. Default, that is if you do not call gluLookAt, is the origin (0,0,0). **Ex.** What would happen if in cube.cpp we placed eyeX, eyeY, eyeZ at (0,0,0)? Where is the centre of the cube?

atX, atY, atZ give the pointing direction of the camera; default direction is pointing down the negative z-axis; default eyeX, eyeY, eyeZ is the origin; default at parameters for atX, atY, atZ are (0,0,-1).

upX, upY, upZ give the orientation. (0.0,1.0,0.0) means camera is oriented with *up* along the y-axis — the default.

Yet again, do not confuse viewing with projection!

7. *Modelling* is constructing the 3D model (objects); modelling transformations are concerned with positioning (translation), orientation (rotation) and scaling of the objects in the virtual world and with their relative positions, orientations and sizes (scale).
8. glScalef (1.0, 2.0, 1.0); scales the cube by (1,2,1) so that it becomes twice as high (y) and wide (x) and deep (z); experiment with this on cube.cpp.
9. The following lines rotate the cube first about the y-axis (0.,1.,0.) and then about the x-axis (1.,0.,0.)

```
glRotatef((GLfloat)roty,0.,1.,0.);  
glRotatef((GLfloat)rotx,1.,0.,0.);
```

10. Why is the matrix mode called GL_MODELVIEW? Because modelling and viewing are just two sides of the same coin. Example. Assume that in gluLookAt we have positioned the camera at (0,0,5). This is five units back from the centre of the cube. If we had set eyeX, eyeY, eyeZ at (0,0,0) (the default), and then added the modelling transformation

```
glTranslatef(0.0, 0.0, -5.0);
```

we would have achieved the same effect; that is, moving the camera back five units is the same as moving the object forward five units (don't forget the camera is pointing down the *negative z-axis*).

The same equivalence goes for rotation and camera orientation; we'll do some experiments in practicals.

11. wireCube(vv, 1.0); is where we call the wire cube drawing function. There is GLUT function (glutWireCube) that draws a wireframe cube centered at the origin and has sides of length one unit; however it draws in one colour, making it difficult to see in detail what is going on.
12. Why glPushMatrix() and glPopMatrix()?

```
glPushMatrix();  
glRotatef((GLfloat)roty,0.,1.,0.);  
glRotatef((GLfloat)rotx,1.,0.,0.);  
wireCube(vv, 1.0);  
glPopMatrix();
```

OpenGL stores its matrices on *stacks*; if you do not know what a stack is, ask me in class, or use Google. The current matrix of each type (e.g. modelview and projection) is at the top of the relevant stack.

`glPushMatrix()` makes a copy of the current top matrix and pushes it onto the stack. This means that there are now two copies of that matrix, one at the top of the stack, another next one down. `glPushMatrix()` is therefore a way to *save* the current matrix ready for *restoring* later.

When we now apply a modelling transformation such as `glRotatef`, the current matrix (top of the stack) is modified; if we had not used `glPushMatrix()` we would not be able to get back to the original. When we have finished with the code for which we want the modified matrix (two rotations applied, about x-axis and y-axis) to be used, we use `glPopMatrix()` to discard the modified matrix and restore the original to the top of the stack, and hence make it current.

The true use of this will become apparent when we examine the `planets.cpp` program later. In this program because, at the beginning of each call to `display` we always call `glLoadIdentity` and because we do nothing after we call `wireCube`, then `glPushMatrix()` and `glPopMatrix()` have no real effect. **Ex.** Make sure you can explain this. However, we include them because it is good practice to use `glPushMatrix()` and `glPopMatrix()` and it is easy to forget them — one cause of not seeing what you want, or of a blank screen.

13. Now go and run Nate Robins' projection and modelview tutorials.

5.2 Mouse motion callback

The following function registers mouse motion when a button is pressed ('dragging'). It is registered as a callback in `main` using `glutMotionFunc(motion)`. As you will see when using `cube.cpp` it gives a quite intuitive (trackerball?) method of rotating the cube.

```
void motion(int x, int y){
    // the next block stops it jumping when mouse first moved
    static bool first = true;
    if(first){
        beginx = x;
        beginy = y;
        first = false;
    }
    if(rotate){
        rotx = rotx + (y - beginy);
        roty = roty + (x - beginx);
        beginx = x;
        beginy = y;
        glutPostRedisplay();
    }
    return;
}
```

5.3 glFrustum and gluPerspective

The primary method (if not the most intuitive) of defining a perspective projection in OpenGL is to define a *frustum* of a pyramid using `glFrustum`, see Figure 5.5. `glFrustum` takes six arguments:

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,  
              GLdouble near, GLdouble far).
```

In Figure 5.5 these are abbreviated l, r, b, t, n, f .

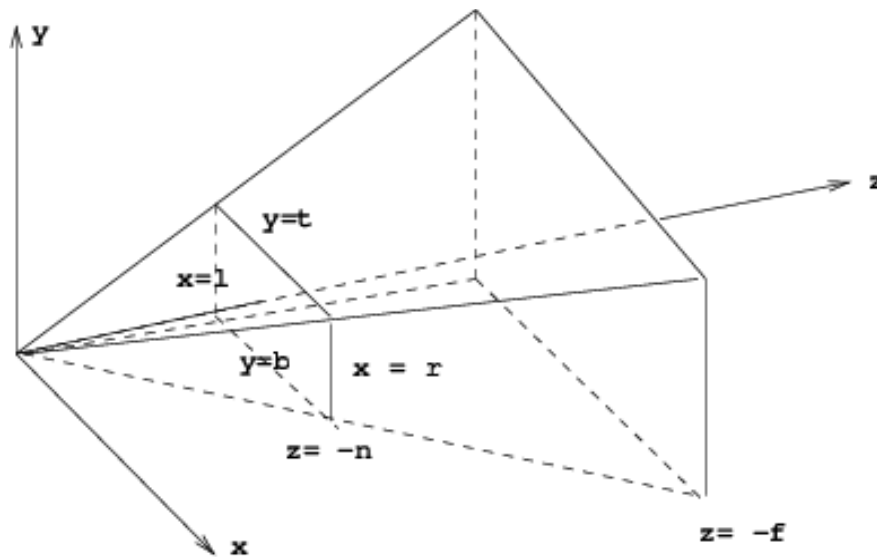


Figure 5.5: Perspective view frustum volume defined by `glFrustum`.

Respectively, these denote the *left* and *right* (x-axis) extent, and the *bottom* and *top* (y-axis) extent of the field of view; $-near$ is the z position of the projection plane. Rendering is limited to the frustum defined by $left, right, bottom, top, near, far$; anything outside is not rendered and these form a 3D *clipping region*. OpenGL maps the frustum given by l, r, b, t, n, f to the homogeneous clip space cube shown in Figure 5.6. It maps: $l \mapsto -1, r \mapsto +1, b \mapsto -1, t \mapsto +1$. For use in hidden surface removal, mapped z coordinates are retained; z coordinates are mapped thus: $n \mapsto -1, f \mapsto +1$; notice that this reverses the direction of the z axis.

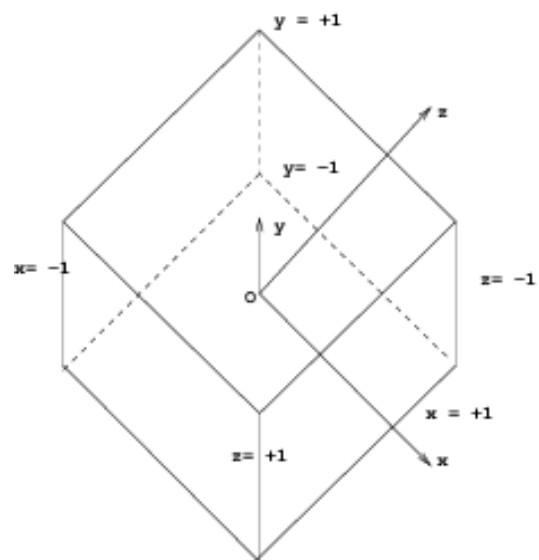


Figure 5.6: Homogeneous clip space cube.

`glFrustum` may be rather unintuitive to use, so `gluPerspective` has been provided:

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

Example (from `house3d.cpp`)

```
void reshape (int w, int h){
    double theta = 50.0;
    double n = 1.0;
    double tanBy2 = tan(theta*toRadians/2.0);
    //cout<< tanBy2<< endl;
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    // glFrustum (-n*tanBy2, n*tanBy2, -n*tanBy2*aspectRatio,
    //           n*tanBy2*aspectRatio, n, 20.0);
    gluPerspective(theta, (double)w/(double)h, n, 20.0);
    glMatrixMode (GL_MODELVIEW);
    assert(glGetError() == GL_NO_ERROR);
}
```

The call to `glFrustum` (commented out) is exactly equivalent to the call to `gluPerspective`; notice the trick to keep aspect the same no matter the shape of the window (width `w`, height `h`). Notice also that the `tan` function take *radians* rather than *degrees*.

Ex. Show that these two are equivalent:

```
// glFrustum (-n*tanBy2, n*tanBy2, -n*tanBy2*aspectRatio,
//           n*tanBy2*aspectRatio, n, 20.0);
gluPerspective(theta, (double)w/(double)h, n, 20.0);
```

Figure 5.7 shows what the arguments mean. Compare with the equivalent `glFrustum` diagram, Figure 5.8.

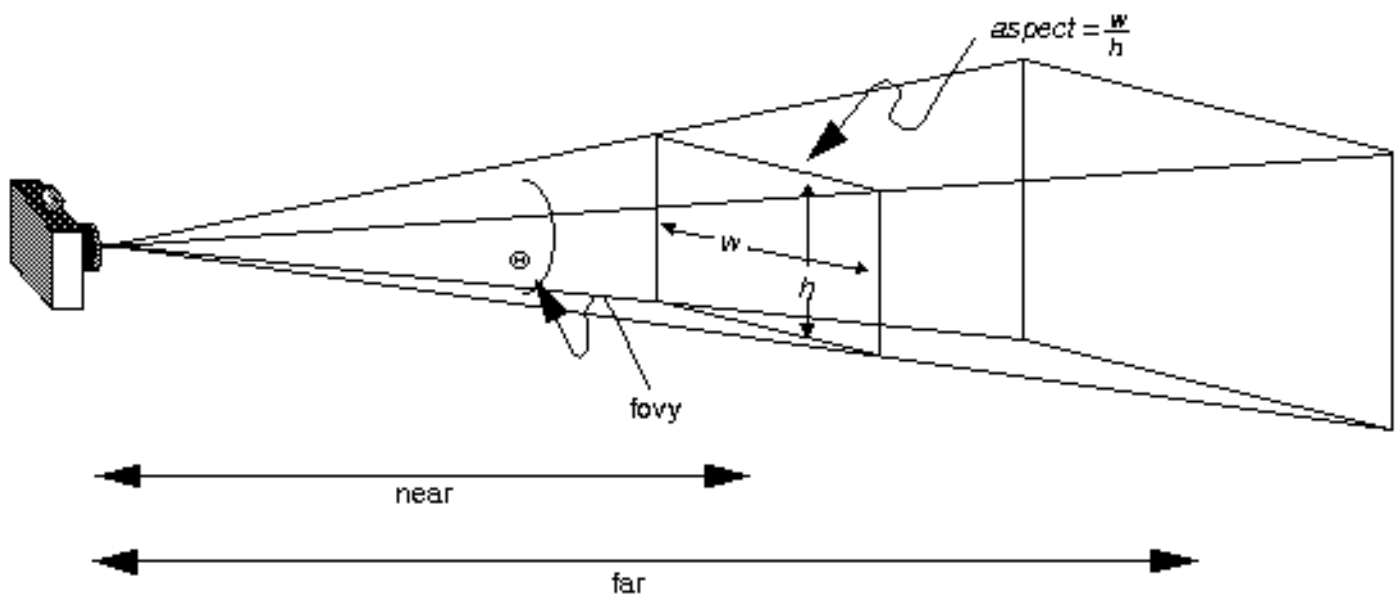


Figure 5.7: gluPerspective.

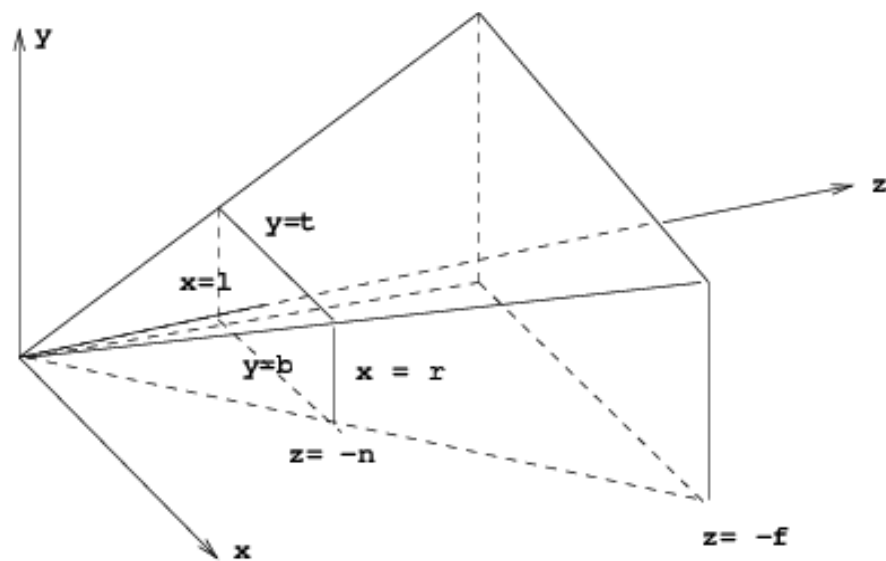


Figure 5.8: Perspective view frustum volume defined by glFrustum.

5.4 Reading the Contents of Transformation Matrices

As described in (Campbell 2008b) Chapters 7 and 8, OpenGL *modelview* and *projection* matrices (separate matrices) are stored in four-by-four homogeneous transformation matrices. See also (Campbell 2008b) Chapters 5.8 for a description of OpenGL's memory layout of matrices.

`glGetFloatv` allows us to retrieve the contents of these matrices. `matPrint`, see Figure 5.9, is a utility I use to read and print them; it is contained in `GLutils.h`, `.cpp`.

```
void matPrint(GLenum nmat, const char *msg){
    double mat[16];
    int r, c; int nc= 4, nr= 4, i;

    glGetDoublev(nmat, mat);
    printGLErrorCode();
    assert(glGetError() == GL_NO_ERROR);
    cout<< msg<< endl;
    cout<< "[";
    streamsize prec = cout.precision(4);
    streamsize width = cout.width();
    double d;
    for(r= 0; r< nr; r++){
        cout<< "(";
        for(c= 0; c< nc; c++){
            i= r + c*nr;
            d= mat[i];
            if(fabs(d)< 1.0e-04)d=0.0;
            cout<< setw(6)<< dec<< d; // dangerous!
            if(c == nc-1) cout<< ")"<< endl;
            else cout<< ", ";
        }
    }
    cout<<"]"<< endl;
    cout<< setprecision(prec) << endl;
    cout<< setw(width);
    assert(glGetError() == GL_NO_ERROR);
}
```

Figure 5.9: Fetch and Print Transformation Matrices (in `GLutils.h`, `.cpp`).

Figure 5.10 shows our inclusion of `matPrint` in the display of `cube.cpp`; we have modified the code a little from what is above to make the output more understandable.

When we now execute `cube.cpp` we get the output shown in Figure 5.11.

```

void display(void){
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    matPrint(GL_PROJECTION_MATRIX, "GL_PROJECTION_MATRIX 0");
    //gluPerspective(fov, (GLfloat)ww/ (GLfloat)hh, 1.0, 20.0);
    glFrustum (-0.7, 0.7, -0.7, 0.7, 1.0, 20.0);
    matPrint(GL_PROJECTION_MATRIX, "GL_PROJECTION_MATRIX 1");
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 0");
    eye[0] = 0.0; eye[1] = 0.0; eye[2] = 10.0;
    gluLookAt(eye[0], eye[1], eye[2], // camera centre
              0.0, 0.0, 0.0,          // pointing AT
              0.0, 1.0, 0.0);         // up vector
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 1");

    double xNeg= -1.2, xPos= 1.2, yNeg = -1.2, yPos = 1.2,
           zNeg = -1.2, zPos = 1.2;
    drawAxes(xNeg, xPos, yNeg, yPos, zNeg, zPos);

    glPushMatrix();
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 2");
    roty = 30;
    glRotatef((GLfloat)roty,0.,1.,0.);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 3");
    rotx = 60;
    glRotatef((GLfloat)rotx,1.,0.,0.);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 4");
    //glScalef(1.0, 2.0, 1.0);
    solidCube(vv, 1.0);
    //wireCube(vv, 1.0);
    glPopMatrix();
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 5");

    glutSwapBuffers();
    printGLErrorCode();
}

```

Figure 5.10: Printing the projection and modelview matrices from cube.cpp

```

GL_PROJECTION_MATRIX 0
[( 1, 0, 0, 0)
 ( 0, 1, 0, 0)
 ( 0, 0, 1, 0)
 ( 0, 0, 0, 1)
]
GL_PROJECTION_MATRIX 1
[( 1.429, 0, 0, 0)
 ( 0, 1.429, 0, 0)
 ( 0, 0, -1.105, -2.105)
 ( 0, 0, -1, 0)
]
GL_MODELVIEW_MATRIX 0
[( 1, 0, 0, 0)
 ( 0, 1, 0, 0)
 ( 0, 0, 1, 0)
 ( 0, 0, 0, 1)
]
GL_MODELVIEW_MATRIX 1
[( 1, 0, 0, 0)
 ( 0, 1, 0, 0)
 ( 0, 0, 1, -10)
 ( 0, 0, 0, 1)
]
GL_MODELVIEW_MATRIX 2
[( 1, 0, 0, 0)
 ( 0, 1, 0, 0)
 ( 0, 0, 1, -10)
 ( 0, 0, 0, 1)
]
GL_MODELVIEW_MATRIX 3
[( 0.866, 0, 0.5, 0)
 ( 0, 1, 0, 0)
 ( -0.5, 0, 0.866, -10)
 ( 0, 0, 0, 1)
]
GL_MODELVIEW_MATRIX 4
[( 0.866, 0.433, 0.25, 0)
 ( 0, 0.5, -0.866, 0)
 ( -0.5, 0.75, 0.433, -10)
 ( 0, 0, 0, 1)
]
GL_MODELVIEW_MATRIX 5
[( 1, 0, 0, 0)
 ( 0, 1, 0, 0)
 ( 0, 0, 1, -10)
 ( 0, 0, 0, 1)
]

```

Figure 5.11: Projection and Modelview matrices from cube.cpp

Exercise. Explain the two projection matrices in Figure 5.11 in precise detail; see (Campbell 2008b).

Exercise. Explain the five modelview matrices in Figure 5.11 in precise detail; see Chapter 7 of (Campbell 2008b). Notice that matrix 5 is the same as matrix 1, i.e. `glPushMatrix`, `glPopMatrix` saved and restored as we required.

Note, the examples in the next section are easier to understand as the rotations are about the z-axis.

5.5 Concatenating (composing) Transformations

It may help read the *Examples of Composing Several Transformations* section of (Shreiner et al. 2008a) Chapter 3. Figure 5.12 shows the output from `model1.cpp` (and extended version of `model1.c` from (Shreiner et al. 2008a); the relevant parts of `model1.cpp` are shown in Figures 5.13 and 5.14.

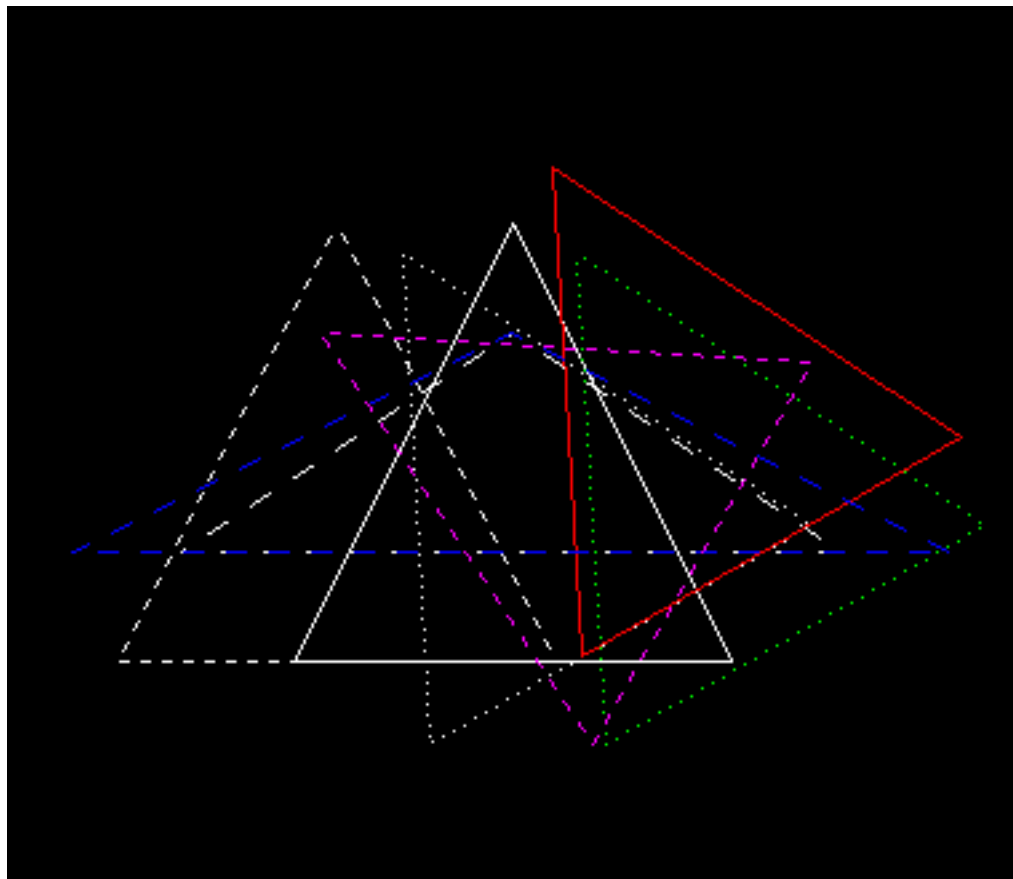


Figure 5.12: Composing transformations: output from `model1.cpp`

```

void draw_triangle(void){
    glBegin (GL_LINE_LOOP);
        glVertex2f(0.0, 25.0);
        glVertex2f(25.0, -25.0);
        glVertex2f(-25.0, -25.0);
    glEnd();
}

void display(void){
    double mat1[16]= {
        2.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.5000, 0.0000, 0.0000,
        0.0000, 0.0000, 1.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 1.0000};

    double mat2[16]= {
        0.5000, 0.866, 0.0000, 0.0000,
        -0.866, 0.5000, 0.0000, 0.0000,
        0.0000, 0.0000, 1.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 1.0000};

    glClear (GL_COLOR_BUFFER_BIT);    glColor3f (1.0, 1.0, 1.0);

    glLoadIdentity ();
    glColor3f (1.0, 1.0, 1.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 1");
    draw_triangle ();

    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0xF0F0);
    glLoadIdentity ();
    glTranslatef (-20.0, 0.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 2");
    draw_triangle ();

    glLineStipple (1, 0xF00F);
    glLoadIdentity ();    glScalef (1.5, 0.5, 1.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 3");
    draw_triangle ();

    glLineStipple (1, 0x8888);
    glLoadIdentity ();
    glRotatef (30.0, 0.0, 0.0, 1.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 4");
    draw_triangle ();    /* continued ...*/
}

```

Figure 5.13: Concatenating transformations (model1.cpp), Part 1.

```

    /* ... continued*/
    glColor3f(1.0, 0.0, 0.0);
    glLineStipple (1, 0xffff);
    glLoadIdentity ();
    glRotatef (30.0, 0.0, 0.0, 1.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 5");
    glTranslatef (20.0, 0.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 6");
    draw_triangle ();

    glColor3f(0.0, 1.0, 0.0);
    glLineStipple (1, 0x8888);
    glLoadIdentity ();
    glTranslatef (20.0, 0.0, 0.0);
    glRotatef (30.0, 0.0, 0.0, 1.0);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 7");
    draw_triangle ();

    glColor3f(0.0, 0.0, 1.0);
    glLineStipple (1, 0xF00F);
    glLoadIdentity ();
    glMultMatrixd(mat1);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 8");
    draw_triangle ();

    glColor3f(1.0, 0.0, 1.0);
    glLineStipple (1, 0xF0F0);
    glLoadIdentity ();
    glMultMatrixd(mat2);
    matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 9");
    draw_triangle ();

    glDisable (GL_LINE_STIPPLE);
    glFlush ();

```

Figure 5.14: Concatenating transformations (model1.cpp), Part 2.

Dissection of model1.cpp Now we dissect model1.cpp together with snapshots of the MODELVIEW_MATRIX that have been output using matPrint. The basic figure is an equilateral triangle with it's apex at $(x = 0, y = 25)$ and its base at points at $(x = -25, y = -25)$ and $(x = 25, y = -25)$; thus, before any transformations, the origin is at $(0, 0)$.

First we recall eqns. 7.2 and 7.3 of (Campbell 2008b); eqn. 5.1 shows a translation, and eqn. 5.2 shows a rotation about the z-axis.

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \\ u_w \end{bmatrix}. \quad (5.1)$$

$$R_z(b) = \begin{bmatrix} \cos b & -\sin b & 0 & 0 \\ \sin b & \cos b & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.2)$$

1. The basic triangle is drawn,

```
glLoadIdentity ();
glColor3f (1.0, 1.0, 1.0);
matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 1");
draw_triangle ();
```

and the modelview matrix is the identity matrix:

```
GL_MODELVIEW_MATRIX 1
[(1.0000, 0.0000, 0.0000, 0.0000)
(0.0000, 1.0000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

2. We translate $t_x = -20$ and draw a second triangle.

```
GL_MODELVIEW_MATRIX 2
[(1.0000, 0.0000, 0.0000, -20.0000)
(0.0000, 1.0000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

3. Now glScalef (1.5, 0.5, 1.0);.

```
GL_MODELVIEW_MATRIX 3
[(1.5000, 0.0000, 0.0000, 0.0000)
(0.0000, 0.5000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

4. Next `glRotatef (30.0, 0.0, 0.0, 1.0);`; this rotates 30° about the z-axis; note that OpenGL works in degrees; $\cos 30^\circ = 0.866$ and $\sin 30^\circ = 0.5$.

```
GL_MODELVIEW_MATRIX 4
[(0.8660, -0.5000, 0.0000, 0.0000)
(0.5000, 0.8660, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

5. Now we compose transformations and analysis becomes less trivial.

```
glLoadIdentity (); //I
glRotatef (30.0, 0.0, 0.0, 1.0); //R
matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 5");
glTranslatef (20.0, 0.0, 0.0); //T
matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 6");
```

The first matrix is obvious enough:

```
GL_MODELVIEW_MATRIX 5
[(0.8660, -0.5000, 0.0000, 0.0000)
(0.5000, 0.8660, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)],
```

but it is the order of combination of the matrices that is crucial to getting to grips with this aspect of OpenGL. The modelview matrix successively contains I , $IR = R$ and finally RT . Thus if we consider transformation of a vertex $\mathbf{u}$, we effectively have $RT\mathbf{u} = R(T\mathbf{u})$. That is, effectively, $\mathbf{u}$ is first translated and then the translated vertex rotated. Of course we know that in fact the matrices are composed $P = IRT = RT$ and then P applied $P\mathbf{u}$; the composite matrix is shown below:

```
GL_MODELVIEW_MATRIX 6
[(0.8660, -0.5000, 0.0000, 17.3205)
(0.5000, 0.8660, 0.0000, 10.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)].
```

Ex. Multiply RT , where R is a rotation of 30° about the z-axis as above (GL_MODELVIEW_MATRIX 5) and T represents a translation of 20 along the x-axis:

```
[(1.0000, 0.0000, 0.0000, 20.0000)
(0.0000, 1.0000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

Ex. Draw a diagram which shows the effect of RT , i.e. a translation *followed* by a rotation *about the origin*.

6. Now we do apply the transformations in reverse order.

```
glLoadIdentity ();
glTranslatef (20.0, 0.0, 0.0);
glRotatef (30.0, 0.0, 0.0, 1.0);
matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 7");
```

In this case the result is more transparent:

```
GL_MODELVIEW_MATRIX 7
[(0.8660, -0.5000, 0.0000, 20.0000)
(0.5000, 0.8660, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

Ex. Multiply TR and verify the result in GL_MODELVIEW_MATRIX 7.

Ex. Draw a diagram which shows the effect of TR , i.e. a rotation about the origin *followed* by a translation.

Ex. Draw a diagram which shows the effect of the sequence below.

```
glLoadIdentity ();
glTranslatef (20.0, 0.0, 0.0); //T
glRotatef (30.0, 0.0, 0.0, 1.0); //R1
glRotatef (-30.0, 0.0, 0.0, 1.0); //R2
```

Ex. What will be the modelview matrix after T, R1, R2?

7. Now we show how to load your own modelview matrix, in this case mat1:

```
double mat1[16]= {
2.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.5000, 0.0000, 0.0000,
0.0000, 0.0000, 1.0000, 0.0000,
0.0000, 0.0000, 0.0000, 1.0000};
```

which is a scaling matrix $s_x = 2, s_y = 0.5$.

```
glLoadIdentity ();
glMultMatrixd(mat1);
matPrint(GL_MODELVIEW_MATRIX, "GL_MODELVIEW_MATRIX 8");
```

And the output to confirm:

```
GL_MODELVIEW_MATRIX 8
[(2.0000, 0.0000, 0.0000, 0.0000)
(0.0000, 0.5000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

8. Next we load a rotation matrix (60°), `mat2`.

```
double mat2[16]= {
0.5000, 0.866, 0.0000, 0.0000,
-0.866, 0.5000, 0.0000, 0.0000,
0.0000, 0.0000, 1.0000, 0.0000,
0.0000, 0.0000, 0.0000, 1.0000};
```

And the output of `matPrint`:

```
GL_MODELVIEW_MATRIX 9
[(0.5000, -0.8660, 0.0000, 0.0000)
(0.8660, 0.5000, 0.0000, 0.0000)
(0.0000, 0.0000, 1.0000, 0.0000)
(0.0000, 0.0000, 0.0000, 1.0000)]
```

Notice anything wrong? The output appears to be transposed? No, it's the input that is transposed; as described in (Campbell 2008b), Section 5.8, matrix data are stored in memory as `m[0][0]`, `m[1][0]`, `m[2][0]`, `m[3][0]`, `m[0][1]`, ... and not `m[0][0]`, `m[0][1]`, ... as might naively be expected.

5.6 A Solar System, `planet.cpp`

To further enforce the lesson on composing transformations, we present another example from (Shreiner et al. 2008a), Chapter 3, namely `planet.cpp`. Figure 5.15 shows the static graphic with a *sun* and a smaller *planet*. In `planet.c`, one can increment year and day angles using the keyboard. The display part of the program is shown in Figure 5.16.

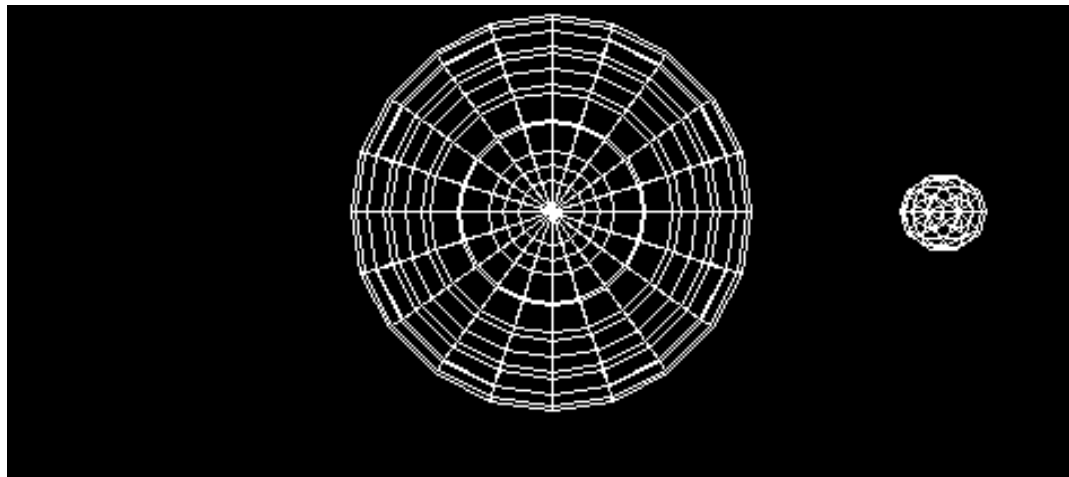


Figure 5.15: Solar system, `planet.cpp`.

```

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);

    glPushMatrix();
        glutWireSphere(1.0, 20, 16);    /* draw sun */
        glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
        glTranslatef (2.0, 0.0, 0.0);
        glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
        glutWireSphere(0.2, 10, 8);    /* draw planet */
    glPopMatrix();
    glutSwapBuffers();
}

```

Figure 5.16: Concatenating transformations (planet.c).

Comments on planet.cpp In planet.cpp, we first draw the *sun* at the origin, appropriately enough `glutWireSphere(1.0, 20, 16);`. The arguments are `glutWireSphere(GLdouble radius, GLint slices, GLint stacks);` `slices` corresponds to lines of longitude (east-west) and `stacks` corresponds to lines of latitude (north-south).

Then three modelling transformations, followed by drawing the planet:

```
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0); // P
glTranslatef (2.0, 0.0, 0.0); // T
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0); // R
glutWireSphere(0.2, 10, 8); /*planet*/
```

Following the analysis of the previous section, note carefully what is happening, for any planet vertex $\mathbf{u}$, we have $PT\mathbf{R}\mathbf{u} = P(T(R\mathbf{u}))$, in other words, transformations occur in the following sequence:

1. `glRotatef ((GLfloat) day, ...` The planet spins about *its* y-axis, 1° every day angle unit;
2. `glTranslatef (2.0, 0.0, 0.0);` The planet's initial centre / origin is 2 units along the x-axis;
3. `glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);` The planet orbits about *the sun's* y-axis, 1° every year angle unit.

Ex. Add a moon which orbits about the planet; orbit over the poles, i.e. rotate about the x-axis; add appropriate code to keyboard to handle month angle.

Ex. Take your version of planet.c with the moon added and add a `glutIdleFunc`; callback that increments a `dayTotal` variable by 5. Assign `dayTotal` to `day` and wrap `day` around at 360 as before. Then derive `month` and `year` from `dayTotal` so that the solar system animation proceeds without any keyboard interaction.

In addition to these experiments on planet.c, we will do some practical experiments with robot.cpp also from (Shreiner et al. 2008a) Chapter 3; these programs will be in my public folder's `\progs\ch05cpp\`. See also section 5.10 of (Angel 2008).

5.7 3-D House Example

Program `house3d.cpp` introduces a 3-D wireframe house drawing. In this we will see how to (a) write and use a menu callback function; (b) write and use a timer callback function; (c) check for OpenGL errors; (d) check which version of OpenGL you are running.

```
/* ----- house3d.cpp -----
 * j.g.c. 2005-03-22, 2006-12-31
 * wireframe 3D house
 -----*/
#include <GL/glut.h> #include <iostream> #include <cmath>
#include <cassert> #include <sstream>

using std::cout; using std::endl;
const int DT = 100; unsigned long t = 0;
const double pi = 4.0*atan(1.0); const double toRadians = pi/180.0;
const int QUIT_VALUE( 99 );
const int RED = 0, GREEN = 1, BLUE = 2; int xCol = RED;
const float col[4][3]={1.0, 0.0, 0.0},
                    {0.0, 1.0, 0.0},
                    {0.0, 0.0, 1.0},
                    {1.0, 1.0, 1.0}};

void timer(int value);
```

```

void house() {
    float a = 1.0;
    /*front*/
    glColor3f (1.0, 0.0, 0.0);
    glBegin(GL_LINES);
        /*top*/      glVertex3f(+a, +a, +a); glVertex3f(-a, +a, +a);
        /*left*/     glVertex3f(-a, +a, +a); glVertex3f(-a, -a, +a);
        /*bottom*/   glVertex3f(-a, -a, +a); glVertex3f(+a, -a, +a);
        /*right*/    glVertex3f(+a, -a, +a); glVertex3f(+a, +a, +a);
        /*roof*/
        glVertex3f(-a, +a, +a); glVertex3f(+a, +a, +a);
        glVertex3f(+a, +a, +a); glVertex3f(0.0, 1.3, +a);
        glVertex3f(0.0, 1.3, +a); glVertex3f(-a, +a, +a);
    glEnd();
    /* door */
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_LINES);
        /*top */ glVertex3f(+0.15, +0., +a); glVertex3f(-0.15, +0., +a);
        /*left*/ glVertex3f(-0.15, +0., +a); glVertex3f(-0.15, -a, +a);
        /*bott*/ glVertex3f(-0.15, -a, +a); glVertex3f(+0.15, -a, +a);
        /*righ*/ glVertex3f(+0.15, -a, +a); glVertex3f(+0.15, +0., +a);
    glEnd();
    /*joins front-rear*/
    glColor3f(0.0, a, 0.0);
    glBegin(GL_LINES);
        /*top left*/
        glVertex3f(-a, +a, -a); glVertex3f(-a, +a, +a);
        /*top right*/
        glVertex3f(+a, +a, +a); glVertex3f(+a, +a, -a);
        /*bottom left*/
        glVertex3f(-a, -a, +a); glVertex3f(-a, -a, -a);
        /*bottom right*/
        glVertex3f(+a, -a, +a); glVertex3f(+a, -a, -a);
        /*roof apex*/
        glVertex3f(0.0, 1.3, +a); glVertex3f(0.0, 1.3, -a);
    glEnd();
    /*rear*/
    glColor3f (0.0, 0.0, a);
    glBegin(GL_LINES);
        /*top*/      glVertex3f(+a, +a, -a); glVertex3f(-a, +a, -a);
        /*left*/     glVertex3f(-a, +a, -a); glVertex3f(-a, -a, -a);
        /*bott*/     glVertex3f(-a, -a, -a); glVertex3f(+a, -a, -a);
        /*right*/    glVertex3f(+a, -a, -a); glVertex3f(+a, +a, -a);
        /*roof*/
        glVertex3f(-a, +a, -a); glVertex3f(+a, +a, -a);
        glVertex3f(+a, +a, -a); glVertex3f(0.0, 1.3, -a);
        glVertex3f(0.0, 1.3, -a); glVertex3f(-a, +a, -a);
    glEnd();
    assert(glGetError() == GL_NO_ERROR); }

```

```

void display(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    //gluLookAt(3.0, 4.0, 12.0, 0.0, 0.0, -10000.0, 0.0, 1.0, 0.0);
    glTranslatef(-3.0f, -4.0f, -12.0f);

    axes(10., -10., 10., -10., 10., -10.);
    house();
    glFlush();
    assert(glGetError() == GL_NO_ERROR);
}

void reshape (int w, int h){
    double theta = 50.0;
    double n = 1.0;
    double tanBy2 = tan(theta*toRadians/2.0);
    //cout<< tanBy2<< endl;
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    // glFrustum (-n*tanBy2, n*tanBy2, -n*tanBy2*aspectRatio,
                n*tanBy2*aspectRatio, n, 20.0);
    gluPerspective(theta, (double)w/(double)h, n, 20.0);
    glMatrixMode (GL_MODELVIEW);
    assert(glGetError() == GL_NO_ERROR);
}

void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 27:
        case 'q':
        case 'Q':
            exit(0);    break;
    }
}

static void mainMenuCB( int value ){
    switch (value){
        case QUIT_VALUE:{
            exit( 0 );
        }
        case RED:{
            xCol= RED;
            glutPostRedisplay();
            break;
        }
        case GREEN:{
            xCol= GREEN;

```

```

        glutPostRedisplay();
        break;
    }
    case BLUE:{
        xCol= BLUE;
        glutPostRedisplay();
        break;
    }
    default:
        break;
    }
}

void init(void) {
    version();
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);

    glutCreateMenu( mainMenuCB );
    glutAddMenuEntry( "X-Axis Red", RED );
    glutAddMenuEntry( "X-Axis Green", GREEN );
    glutAddMenuEntry( "X-Axis Blue", BLUE );
    glutAddMenuEntry( "Quit", QUIT_VALUE );
    glutAttachMenu( GLUT_RIGHT_BUTTON );
    glutTimerFunc(DT, timer, 1);
}

```

```

void timer(int val){
    t += DT;
    if(t%1000==0)cout<< "t = "<< t/1000<< "s."<< endl;
    glutPostRedisplay();
    glutTimerFunc(DT, timer, 1);
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutMainLoop();
    return 0;
}

```

5.7.1 Dissection of house3d.cpp

1. Here is a portable method of computing π .

```
const double pi = 4.0*atan(1.0); const double toRadians = pi/180.0;
```

2. We draw the house centred on the origin.
3. Later in chapter 8 (*display lists, vertex arrays and buffer objects*) we will argue that all those `GL_LINES` calls are rather inefficient, but efficiency is no concern here.
4. Notice how to find out if OpenGL has encountered an error.

```
assert(glGetError() == GL_NO_ERROR);
```

5. The above shows how to use assertions; `assert` takes a `bool` argument; if the argument is false, the program is halted and a message printed giving the offending line number.
6. version in `GLutils.h`, `.cpp` shows how to get the OpenGL version number; for example, to check if certain features (like *buffer objects*, see chapter 8) are available.

Notice how to get the version string; then how to use `stringstream` to decode it. An `istringstream` is an input `stringstream` and you can read bits out of it like a proper input stream.

```

void version(){
    std::string ver((const char*) glGetString(GL_VERSION));
    assert(!ver.empty());
    std::istringstream verStream(ver);
    cout<< ver<< endl;
    int major, minor;

```

```

    char dummySep;
    verStream >> major>> dummySep>> minor;
    cout<< "OpenGL version "<< major<< "."<< minor<< endl;
    assert(glGetError() == GL_NO_ERROR);
}

```

7. For clarity, we show the x-, y-, and z-axes. The axes code is in GLutils.h, .cpp.
8. Menus and timers are discussed in separate subsections.

5.7.2 Assertions

C++ provides a macro function, `assert`, which evaluates a Boolean expression and if it evaluates to *false* generates a run-time error message and halts the program. `assert` is defined in `cassert`.

The following shows an example use of `assert` to check that OpenGL has *not* encountered any error.

```
assert(glGetError() == GL_NO_ERROR);
```

`assert` takes a `bool` argument; if the argument is *false*, the program is halted and a message printed giving the offending line number.

For performance or other reasons, the assertion can be disabled either by including the compiler switch `-DNDEBUG`; I'm not sure how Visual Studio handles this. Alternatively insert the following pre-processor directive in the source file:

```
#define NDEBUG
```

Use of *assertions* like this is an example of *defensive programming* and can be very useful in testing and can greatly simplify debugging.

5.7.3 Mouse Operated Menu

All the menu does is allow you to select the colour of the x-axis. First we define the menu *callback function*; this gives the actions that should result from *menu selections*

```

static void mainMenuCB( int value ){
    switch (value){
        case QUIT_VALUE:{
            exit( 0 );
        }
        case RED:{
            xCol= RED;
            glutPostRedisplay();
        }
    }
}

```

```

        break;
    }
    case GREEN:{
        xCol= GREEN;
        glutPostRedisplay();
        break;
    }
    case BLUE:{
        xCol= BLUE;
        glutPostRedisplay();
        break;
    }
    default:
        break;
    }
}

```

Next, in `init`, we create the menu.

```

glutCreateMenu( mainMenuCB );
glutAddMenuEntry( "X-Axis Red", RED );
glutAddMenuEntry( "X-Axis Green", GREEN );
glutAddMenuEntry( "X-Axis Blue", BLUE );
glutAddMenuEntry( "Quit", QUIT_VALUE );
glutAttachMenu( GLUT_RIGHT_BUTTON );

```

5.7.4 GLUT Timer function

In running an animation, you *could*, like in the rotating rectangle in chapter 5, let `glutIdleFunc` handle it; but in `glutIdleFunc` you have no control over the speed of the animation.

This is where the `glutTimerFunc` callback is useful.

```

void timer(int val){ //expects an in argument, ignored here
    t += DT; // DT is 100
    if(t%1000==0)cout<< "t = "<< t/1000<< "s."<< endl;
    glutPostRedisplay();
    glutTimerFunc(DT, timer, 1);
}

```

`glutTimerFunc` has signature:

```

glutTimerFunc(unsigned int millisecs, void (*func)(int val1), int val2);

```

- `millisecs` is the number of *millisecs* to wait before the callback function is called.

- `func` is the callback function.
- `val2` is the value (`val1`) that is passed to the callback function.

The `glutIdleFunc` is called only once; if we require a timing sequence, then the callback must initiate another countdown before it returns.

In the very simple example, the time is called every 100 ms. ($DT = 100$); if a new second has passed, we print out the time in seconds. [5]

5.8 A further example

`house3d10.cpp` gives a more substantial example which has a larger menu and which allows the user to animate the last discrete action under control of the timer. We will do a lot of work with this example in practical classes.

```
/* ----- house3d10.cpp -----
 * j.g.c. 2005-03-22, 2006-12-31, 2007-01-12
 * wireframe 3D house
 *-----*/
#include <GL/glut.h>
#include <iostream>
#include <cmath>
#include <cassert>
#include <sstream>

using std::cout;
using std::endl;
const int DT = 100;
unsigned long t = 0;
const double pi = 4.0*atan(1.0);
const double toRadians = pi/180.0;
const int QUIT_VALUE( 99 );
const int RED = 0, GREEN = 1, BLUE = 2;
int xCol = RED;
const float col[4][3]={1.0, 0.0, 0.0},
                {0.0, 1.0, 0.0},
                {0.0, 0.0, 1.0},
                {1.0, 1.0, 1.0}};
float axes[3][3]={1.0, 0.0, 0.0},
                {0.0, 1.0, 0.0},
                {0.0, 0.0, 1.0}};

//float * axis3v;
enum modeEnum {EYE, AT, UP, FRUSTUM, ROTATE, SCALE, TRANSLATE};
enum axisEnum {X, Y, Z};
enum modeEnum mode = EYE;
```

```

enum axisEnum axis= X;

float ff[]= {1.0, 1.0, 1.0};
float eye[]= {0.0, 0.0, 5.0};
float at[]= {0.0, 1.0, 0.0};
float up[]= {0.0, 1.0, 0.0};
float rot[]= {0.0, 0.0, 1.0};
float trans[]= {0.0, 0.0, 0.0};
float scale[]={1.0, 1.0,1.0};
float sgn= 1.0;
float ang= 0.0;
bool animate = false;

int nDisp=0, nResh=0;
int ww, hh;

void modify(float sign){
    //axis3v = &axes[axis][0];
    sgn = sign;
    switch(mode){
        case FRUSTUM:
            ff[axis]+=0.1*sign;
            break;
        case ROTATE:
            ang+=1.0*sign;
            break;
        case EYE:
            eye[axis]+=0.1*sign;
            break;
        case AT:
            at[axis]+=0.1*sign;
            break;
        case UP:
            up[axis]+=0.1*sign;
            break;
        case TRANSLATE:
            trans[axis]+= 1.0*sign;
            break;
        case SCALE:
            scale[axis]+= 1.0*sign;
            break;
        default:
            printf("impossible in modify\n");
            break;
    }
}

void timer(int value);

```

```

void drawAxes(double xPos, double xNeg, double yPos, double yNeg,
             double zPos, double zNeg){
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0xAAAA);
    glBegin(GL_LINES);
        glColor3f(1.0, 0.0, 0.0);
        //glColor3fv(col[xCol]);
        glVertex3d(xPos, 0.0, 0.0);    glVertex3d(0.0, 0.0, 0.0);
        glColor3f(0.0, 1.0, 0.0);
        glVertex3d(0.0, yPos, 0.0);    glVertex3d(0.0, 0.0, 0.0);
        glColor3f(0.0, 0.0, 1.0);
        glVertex3d(0.0, 0.0, zPos);    glVertex3d(0.0, 0.0, 0.0);
    glEnd();

    glLineStipple (1, 0xA0A0);
    glBegin(GL_LINES);
        glColor3f(1.0, 0.0, 0.0);
        //glColor3fv(col[xCol]);
        glVertex3d(0.0, 0.0, 0.0);    glVertex3d(xNeg, 0.0, 0.0);
        glColor3f(0.0, 1.0, 0.0);
        glVertex3d(0.0, 0.0, 0.0);    glVertex3d(0.0, yNeg, 0.0);
        glColor3f(0.0, 0.0, 1.0);
        glVertex3d(0.0, 0.0, 0.0);    glVertex3d(0.0, 0.0, zNeg);
    glEnd();
    glDisable (GL_LINE_STIPPLE);
    //cout << glGetError() << endl;
    assert(glGetError() == GL_NO_ERROR);
}

void house() {
    float a = 1.0;
    glPushMatrix();
    glScalef(s, s, s);

    /*front*/
    glColor3f (1.0, 0.0, 0.0);
    glBegin(GL_LINES);
        /*top*/    glVertex3f(+a, +a, +a); glVertex3f(-a, +a, +a);
        /*left*/   glVertex3f(-a, +a, +a); glVertex3f(-a, -a, +a);
        /*bottom*/ glVertex3f(-a, -a, +a); glVertex3f(+a, -a, +a);
        /*right*/  glVertex3f(+a, -a, +a); glVertex3f(+a, +a, +a);
        /*roof*/
        glVertex3f(-a, +a, +a); glVertex3f(+a, +a, +a);
        glVertex3f(+a, +a, +a); glVertex3f(0.0, 1.3, +a);
        glVertex3f(0.0, 1.3, +a); glVertex3f(-a, +a, +a);
    glEnd();

    /* door */
    glColor3f(1.0, 1.0, 0.0);

```

```

glBegin(GL_LINES);
    /*top */ glVertex3f(+0.15, +0., +a); glVertex3f(-0.15, +0., +a);
    /*left*/ glVertex3f(-0.15, +0., +a); glVertex3f(-0.15, -a, +a);
    /*bott*/ glVertex3f(-0.15, -a, +a); glVertex3f(+0.15, -a, +a);
    /*righ*/ glVertex3f(+0.15, -a, +a); glVertex3f(+0.15, +0., +a);
glEnd();

/*joins front-rear*/
glColor3f(0.0, a, 0.0);
glBegin(GL_LINES);
    /*top left*/
    glVertex3f(-a, +a, -a); glVertex3f(-a, +a, +a);
    /*top right*/
    glVertex3f(+a, +a, +a); glVertex3f(+a, +a, -a);
    /*bottom left*/
    glVertex3f(-a, -a, +a); glVertex3f(-a, -a, -a);
    /*bottom right*/
    glVertex3f(+a, -a, +a); glVertex3f(+a, -a, -a);
    /*roof apex*/
    glVertex3f(0.0, 1.3, +a); glVertex3f(0.0, 1.3, -a);
glEnd();

/*rear*/
glColor3f (0.0, 0.0, a);
glBegin(GL_LINES);
    /*top*/    glVertex3f(+a, +a, -a); glVertex3f(-a, +a, -a);
    /*left*/   glVertex3f(-a, +a, -a); glVertex3f(-a, -a, -a);
    /*bott */  glVertex3f(-a, -a, -a); glVertex3f(+a, -a, -a);
    /*right*/  glVertex3f(+a, -a, -a); glVertex3f(+a, +a, -a);
    /*roof*/
    glVertex3f(-a, +a, -a); glVertex3f(+a, +a, -a);
    glVertex3f(+a, +a, -a); glVertex3f(0.0, 1.3, -a);
    glVertex3f(0.0, 1.3, -a); glVertex3f(-a, +a, -a);
glEnd();
glPopMatrix();
assert(glGetError() == GL_NO_ERROR);
}

void reshape (int w, int h){
    nResh++;
    cout<< "reshape ... "<< "nResh= "<< nResh<< endl;
    ww = w; hh = h;
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glFrustum (-1.0*ff[0], 1.0*ff[0], -1.0*ff[1], 1.0*ff[1],
                1.5*ff[2], 20.0*ff[2]);
    //gluPerspective(theta, (double)w/(double)h, n, 20.0);
    glMatrixMode (GL_MODELVIEW);

```

```

    assert(glGetError() == GL_NO_ERROR);
}

void printArray(const char * name, float a[], unsigned int len){
    cout<< name<< " ";
    for(unsigned int i = 0; i< len; i++){
        cout<< a[i]<< " ";
    }
    cout<< endl;
}

void display(void){
    reshape(ww, hh);
    nDisp++;
    cout<< "display ... "<< "nDisp= "<< nDisp<< endl;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    //gluLookAt(3.0, 4.0, 12.0, 0.0, 0.0, -10000.0, 0.0, 1.0, 0.0);
    gluLookAt (eye[0], eye[1], eye[2], at[0], at[1], at[2],
               up[0], up[1], up[2]);

    //glTranslatef(-3.0f, -4.0f, -12.0f);

    drawAxes(10., -10., 10., -10., 10., -10.);

    printArray("scale", scale, 3);
    printArray("trans", trans, 3);
    printArray("rot axis", axes[axis], 3);
    cout<< "ang = "<< ang<< " axis "<< axis<< endl;

    glScalef(scale[0], scale[1], scale[2]);
    glRotatef(ang, axes[axis][0], axes[axis][1], axes[axis][2]);
    glTranslatef(trans[0], trans[1], trans[2]);
    house();
    glFlush();
    assert(glGetError() == GL_NO_ERROR);
}

void version(){
    std::string ver((const char*) glGetString(GL_VERSION));
    assert(!ver.empty());
    std::istringstream verStream(ver);

    cout<< ver<< endl;

    int major, minor;
    char dummySep;
    verStream >> major>> dummySep>> minor;

```

```

    cout<< "OpenGL version "<< major<< "."<< minor<< endl;
    assert(glGetError() == GL_NO_ERROR);
}

void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 27:
            exit(0);
            break;
        case 'x':
        case 'X':
            axis = X;
            break;
        case 'y':
        case 'Y':
            axis = Y;
            break;
        case 'z':
        case 'Z':
            axis = Z;
            break;
        case 'a':
        case 'A':
            animate = !animate;
            break;
        case '+':
            modify(+1.0);
            glutPostRedisplay();
            break;
        case '-':
            modify(-1.0);
            glutPostRedisplay();
            break;
    }
}

static void mainMenuCB( int value ){
    if(value == QUIT_VALUE){
        exit( 0 );
    }
    else{
        mode = (modeEnum)value;
    }
}

void timer(int val){
    t += DT;
    if(t%1000==0)cout<< "t = "<< t/1000<< "s."<< endl;
    if(animate){

```

```

        modify(sgn);
        glutPostRedisplay();
    }
    glutTimerFunc(DT, timer, 1);
}

void init(void) {
    version();
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    /*glShadeModel (GL_SMOOTH);*/

    /*next two lines for anti-aliasing
    glHint has no effect without glEnable(GL_LINE_SMOOTH)
    */
    /*glHint(GL_NICEST, GL_LINE_SMOOTH_HINT);
    */
    /*glEnable(GL_LINE_SMOOTH);
    */
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);

    glutCreateMenu( mainMenuCB );
    glutAddMenuEntry( "Frustum", FRUSTUM );
    glutAddMenuEntry( "Rotate", ROTATE );
    glutAddMenuEntry( "Translate", TRANSLATE );
    glutAddMenuEntry( "Scale", SCALE );
    glutAddMenuEntry( "Eye", EYE );
    glutAddMenuEntry( "At", AT );
    glutAddMenuEntry( "Up", UP );
    glutAddMenuEntry( "Quit", QUIT_VALUE );
    glutAttachMenu( GLUT_RIGHT_BUTTON );
    glutTimerFunc(DT, timer, 1);
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutMainLoop();
    return 0;
}

```

5.9 Additional Clipping Planes

We know that `glFrustum` defines six clipping planes:

```
void glFrustum(GLdouble left, GLdouble right,  
               GLdouble bottom, GLdouble top,  
               GLdouble near, GLdouble far).
```

We can define additional clipping planes using `glClipPlane`. Figure 5.17 shows the clipping performed by display from `clip.c` shown in Figure 5.18; the left hand figure of Figure 5.17 is what we see when the clip plane commands are commented out.

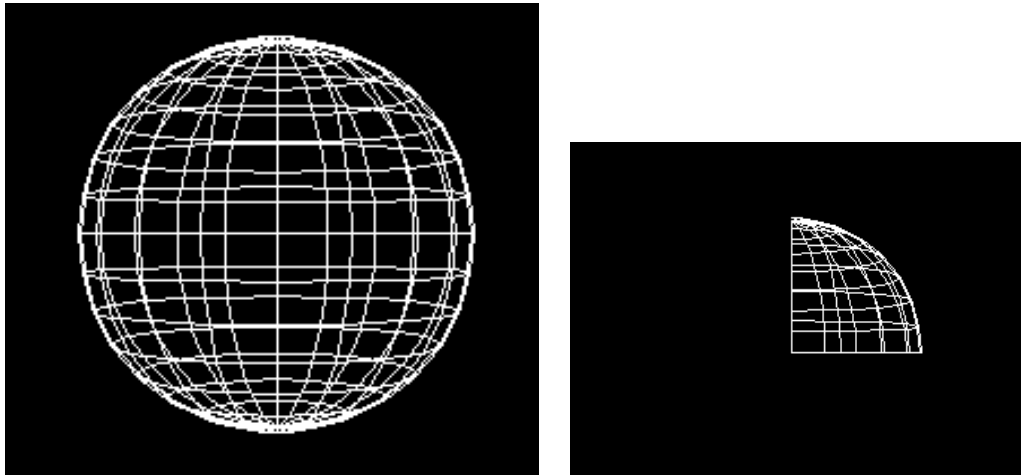


Figure 5.17: Additional clipping planes. L: wireframe sphere, not clipped; R: clipped.

```

void display(void){
    GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
    GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};

    glClear(GL_COLOR_BUFFER_BIT); glColor3f (1.0, 1.0, 1.0);

    glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);

    /*    clip lower half -- y < 0            */
    glClipPlane (GL_CLIP_PLANE0, eqn);
    glEnable (GL_CLIP_PLANE0);
    /*    clip left half -- x < 0            */
    glClipPlane (GL_CLIP_PLANE1, eqn2);
    glEnable (GL_CLIP_PLANE1);

    glRotatef (90.0, 1.0, 0.0, 0.0);
    glutWireSphere(1.0, 20, 16);
    glPopMatrix();
    glFlush ();}

```

Figure 5.18: Using clipping planes (clip.c).

Chapter 6

Lighting

We now show how to add realism to our graphics by including lighting. We take a lot of these notes from (Shreiner et al. 2008a) Chapter 5; that source may be a little thin on the background physics, so it may be worthwhile consulting (Hearn & Baker 2003) Chapter 10 (but no code examples), (Wright et al. 2007) Chapter 5 for good code examples, and (Akenine-Moeller & Haines 2002) Chapter 6 for good theoretical coverage of the *bidirectional reflectance distribution function* (BRDF).

6.1 Background Theory

Up to now we have set the colours of objects and that this is how the objects appear; granted, we have a little variation with the choice of

```
glShadeModel (GL_FLAT); // or
//glShadeModel (GL_SMOOTH);
```

but we are still a long way from how real objects appear in real scenes with directional and coloured lighting, and when the objects differ from being made of matte reflecting materials or shiny materials.

We now give a little theory on light wavelength (λ) dependent effects and colour, repeated from Chapter 2; first spectral (colour/wavelength) responsivity of sensors and transmittivity of colour filters; second spectral reflectance effects; finally we show how OpenGL simulates the infinity of wavelengths with just three colours — something that is quite familiar to use from discussions on colour images and displays in Chapter 2.

After that we discuss angular effects. Look at a light shining on a shiny table — the lightness, and indeed colour, that you see, is dependent on (a) the position of the light source; this position may be given by the direction vector of the light, or by its angles if we are working in angular coordinates, and (b) the position of the viewer, again given by position vector or angles. Each of the positions (a) (light) and (b) viewer, are with respect to the normal vector of the surface, or some other reference in which the surface normal is known. Thus in addition to wavelength, λ , and spatial position, we have a whole raft of angles to take into account. We cover angular effects in a separate section.

6.1.1 Colour Sensing

The relative response of a sensor can be described as a function of wavelength (forget about (x, y) or (r, c) for the present): $d(\lambda)$, where λ denotes wavelength. The light arriving falling on a sensor can also be described as a function of λ , $g(\lambda)$, and the overall output voltage, v , of the sensor is found by integrating the product $d(\lambda)g(\lambda)$ overall wavelengths.

$$v = \int_0^\infty d(\lambda)g(\lambda)d\lambda \quad (6.1)$$

If you are uncomfortable with integrals, think summations, $d(\lambda_i)g(\lambda_i)$ gives the voltage contribution at λ_i , and so we sum the contributions for all λ_i .

If we have a filter in front of the sensor, relative transmittance, $t(\lambda)$, then the light arriving at the sensor, $g'(\lambda)$, is the product of $g(\lambda)$ and $t(\lambda)$:

$$g'(\lambda) = g(\lambda)t(\lambda), \quad (6.2)$$

and eqn.6.1 changes to:

$$v = \int_0^\infty d(\lambda)g(\lambda)t(\lambda)d\lambda. \quad (6.3)$$

If we now jump quickly to a three colour sensor system, or a general three colour model, or the particular OpenGL three colour model, we have,

$$v_i = \int_0^\infty d(\lambda)g(\lambda)t_i(\lambda)d\lambda. \quad (6.4)$$

where $i = 1, 2, 3$, corresponding to red, green, and blue, and $t_i(\lambda)$ is a corresponding red, green, or blue filter.

We can now describe the colour of a pixel using only three numbers, v_r, v_g, v_b : *redness, greenness, blueness*. This approximation can be extremely accurate.

And when we want to display a pixel, we use three appropriately coloured sources, b_r, b_g, b_b , activated according to v_r, v_g, v_b . The three sources b_r, b_g, b_b correspond in colour output to the transmittivities t_r, t_g, t_b used in the sensor system which originally sensed the pixel.

6.1.2 Colour Reflectance

Think monochrome for a second. When you look at a surface, it's *lightness* (lightness is the proper term, rather than brightness) depends on (a) it's reflectance r , and (b) the illumination i falling on it. If it is a good reflector, r is high, and there's a lot of light falling on it, i is high, then the surface appears very bright. On the other hand, r high, low i , the surface appears darkish; and similarly r low (e.g. dark grey), and high i , the surface again appears darkish; and if both r and i are low then the surface appears close to black.

But illumination (i) and reflectance(r) are also functions of λ .

Thus, the lightness function is now spectral, and therefore a function of λ , i.e. the lightness of a spot on the surface at spatial coordinates (x, y) is $f(\lambda, x, y)$ and is the product of two factors:

$i(\lambda, x, y)$, the spectral illumination of the scene, i.e. the amount of light falling on the scene, at (x, y) , at wavelength λ ; $r(\lambda, x, y)$, the reflectance of the scene, i.e. the ratio of light reflected light to incident light at λ .

$$f(\lambda, x, y) = i(\lambda, x, y)r(\lambda, x, y). \quad (6.5)$$

As with sensing and display in the previous subsection, we can again model the full range of λ using the three colours, red, green, and blue. Thus,

$$f_i(x, y) = i_i(x, y)r_i(x, y), \quad \text{for } i = r, g, b. \quad (6.6)$$

Note that we have no cross terms, for example $i_b.r_g$ in eqn. 6.6. In this *model* we have three separate colours and a pure green reflector ($r_r = r_b = 0$) reflects no red or blue, no matter how high i_r, i_b .

6.2 OpenGL Light and Material Models — Ambient, Diffuse, Specular, Emissive

For total realism we would need to exactly model light sources and the reflecting capabilities of materials given by the *Bidirectional reflectance distribution function* (BRDF), $\rho'(\theta_v, \phi_v, \theta_L, \phi_L)$, where θ_v, ϕ_v give the angles that specify the viewer direction and θ_L, ϕ_L are the angles that specify the light direction, see Figure 6.1.

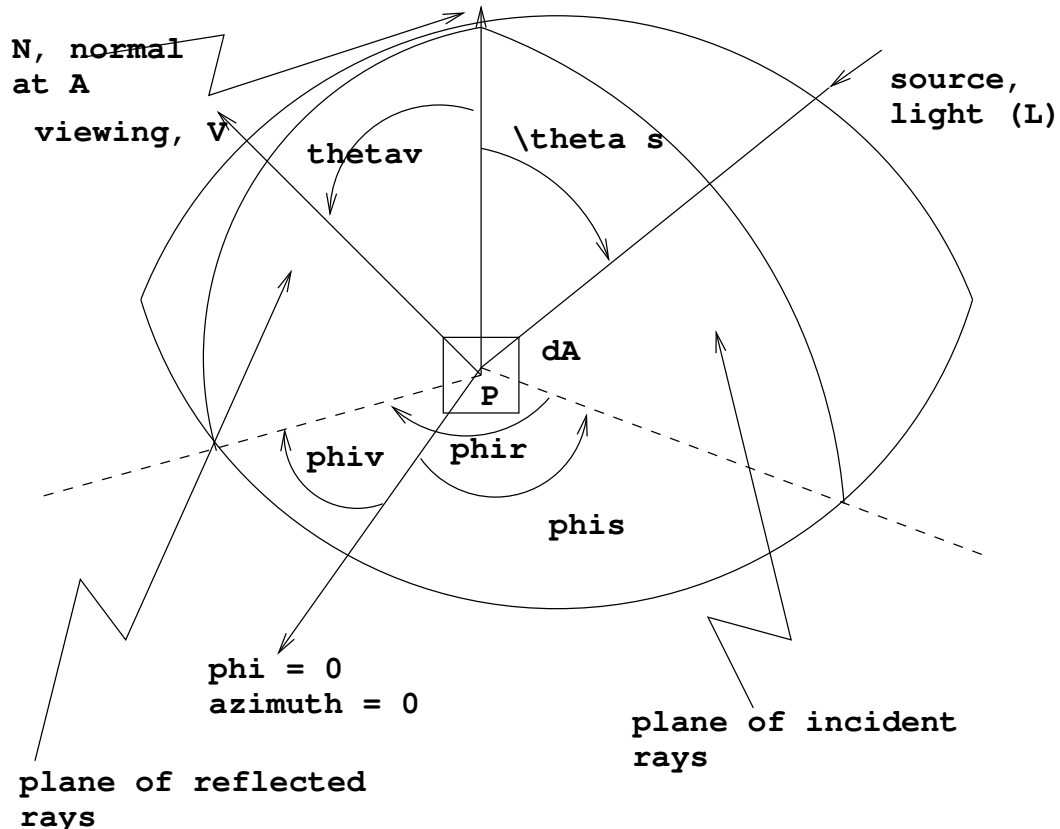


Figure 6.1: Viewing geometry.

In Figure 6.1, θ is the *zenith angle*, i.e. the angle between an incident light ray and the vertical (z-axis) at P or dA ; the *elevation angle* $= 90^\circ - \theta$. θ is written `theta` in the diagram; `thetas` is the `theta` of the light source, `thetav` is the `theta` of the viewer.

The *azimuth angle* of a ray incident at a point P or area element dA is the angle between plane containing the ray and the reference plane of zero azimuth at P or dA , i.e. local North at dA ; the symbol is ϕ (`phi` in the diagram; `phis` is the `phi` of the light source, `phiv` is the `phi` of the viewer and `phir` is the azimuth angle separating the source and the viewer).

BRDF is clearly a very general description of reflectance — but at a great cost, we need *four* angle parameters, plus the wavelength λ parameter, making five parameters.

Such complexity is clearly out of the question for a graphics system such as OpenGL — at least currently.

But as we have said, a world of just colour and no lighting is a flat and uninteresting world. just colour and no lighting or reflected light.

There is a simple lighting-reflectance model called *Lambertian* (called *diffuse* in OpenGL).

Lambertian (diffuse) model In the Lambertian model light is directional but when it is reflected it is reflected equally in all directions in the upper hemisphere. The amount of light incident on a fixed area of the surface depends on the angle at which the light strikes the surface. If θ is the angle of the light source with respect to the *normal* to the surface (i.e. the relative *zenith angle*), then the lightness of the surface varies as $\cos \theta$; hence it is 1 (maximum) when the light source is perpendicular ($\theta = 0$) to the surface and it is 0 when the light is at $\theta = 90^\circ$.

Could we get away with purely diffuse (Lambertian) reflectors with their simplicity of the single parameter k_d (reflectance)? Granted, we would need to allow k_d to vary with colour (k_d^r, k_d^g, k_d^b) for red green and blue — recall that we can model λ variations by considering just three colours. The answer unfortunately is no. Diffuse reflectors are *matte* so you see none of the highlights and shininess that give us sensations of depth and position.

OpenGL simulates reality by allowing four light models and corresponding material reflectivity characteristics: *ambient*, *diffuse*, *specular*, and *emissive*. A light or a surface may have a mixture of these. Emissive is a special case and applies only to a surface which is emitting its own light.

First we give a description in English; later we include some mathematics.

Ambient Ambient light is totally non-directional; think of up-lighting in a room, or light that has entered a room from a window; each of these would have a high ambient content.

We can specify the ambient colour distribution of a light, i.e. the amount of red, green and blue.

We can also specify the *ambient colour* of a material, i.e. it's reflectivity in red, green and blue. In OpenGL, ambient light interacts only with ambient material colour.

Think of ambient light and materials as being like the colours we have been using up to now, and that all scenes were lit with a pure white, $r = 1, g = 1, b = 1$, ambient light. In other words, the reflectivity model is the simple model we used in Chapter 2 and in section 6.1.2 above: you have illumination i (coloured) and a reflecting surface r (coloured) and to get

the light reflected you multiply $i \times r$, for each wavelength, or for each colour, red, green, blue, in the three colour model.

In case you are confused, the term *ambient* applies more naturally to light only — but OpenGL's model needs ambient material colours.

Diffuse The term *diffuse* applies more naturally to the reflectivity of materials than to lights, but as with *ambient*, OpenGL's model needs diffuse lighting to go with diffuse materials.

Diffuse light is directional but when it is reflected it is reflected equally in all directions in the upper hemisphere, this is called *Lambertian reflection*.

We can specify the colour distribution of diffuse lights and likewise the *diffuse colour* of a material, i.e. (k_d^r, k_d^g, k_d^b) .

Specular The term *specular* also applies more naturally to the reflectivity of materials than to lights, but as with *ambient* and *diffuse*, OpenGL's model needs specular lighting to go with specular materials.

Specular light is directional and when it is reflected it is reflected in a preferred direction; think of a shiny surface.

We can specify the colour distribution of specular lights and materials. In OpenGL, specular light interacts only with specular material colour.

Emissive Emissive refers only to materials. Materials may emit their own light. Think of a glowing object; emissive light adds to the lightness of the object. Of course, the glowing object may also be reflecting ambient, diffuse, and specular light. Emissive light *does not act as a light source*, i.e. it does not contribute to the overall lighting of a scene.

Colours are Independent In OpenGL, lights are allowed to have independent colour components for ambient, diffuse, and specular. Likewise materials can have independent colour components for ambient, diffuse, specular, and emissive. If you think this odd, take the example from (Shreiner et al. 2008a): think of the example of a red snooker ball under white light — the snooker ball looks mostly red, but the (specular) highlights are white.

Colours are Combined Independently Let us say we have a light with ambient components I_a^r, I_a^g, I_a^b (red, green, blue), diffuse components I_d^r, I_d^g, I_d^b , and specular components I_s^r, I_s^g, I_s^b .

Now let us have a material with ambient components m_a^r, m_a^g, m_a^b (red, green, blue), diffuse components m_d^r, m_d^g, m_d^b , and specular components m_s^r, m_s^g, m_s^b .

The viewed red lightness of that material will be

$$r^r = I_a^r m_a^r + I_d^r m_d^r + I_s^r m_s^r + e^r, \quad (6.7)$$

where e^r is any emissive component. And similarly for green (r^g) and blue (r^b). If any of the sums r^r, r^g, r^b is greater than 1.0, it is *clamped* (the OpenGL term) to 1.0.

6.3 Mathematical Description of the OpenGL Lighting Model

This section and its figures are derived from

<http://www.delphi3d.net/articles/viewarticle.php?article=phong.htm>.

The OpenGL lighting model is based on the lighting model published by Bui-Tuong Phong in 1973.

The geometry involved is shown in Figure 6.2.

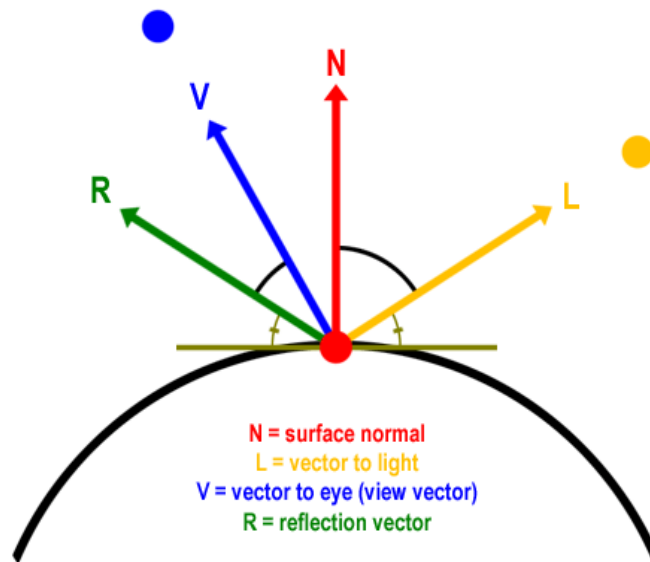


Figure 6.2: Phong lighting model; **L** is the direction of the light; **N** is the normal to the surface, **V** is the direction of the viewer (eye); **R** is the direction where reflection angle is equal to the incidence (lighting) angle, i.e. the angle where all the light would go if we had a perfectly mirror surface.

As in the previous section, we need to specify:

- Ambient light and ambient material reflectivity: I_a, m_a ; note, most books use k_a for ambient reflectivity, but we stick with m .
- Diffuse light and diffuse material reflectivity: I_d, m_d ;
- Specular light and reflectivity: I_s, m_s ;
- Material emission: m_e .

Each of these has colour, so, for example, $m_a = (m_a^r, m_a^g, m_a^b, m_a^a)$, for red, green, blue and alpha, but, to keep the notation simple we normally exclude the superscripts because we know that everything has three colours plus a possible alpha.

We now describe the viewed radiance (I) due to each light and material type.

Ambient

$$I_a = I_a m_a. \quad (6.8)$$

Diffuse $I_d = I_d m_d \mathbf{N} \cdot \mathbf{L}$, where $\mathbf{N}$ is the *normal* vector and $\mathbf{L}$ is the direction vector of the light source. If $\mathbf{N} \cdot \mathbf{L} \leq 0$ it is set to 0; and 0 means no reflection.

Recall the scalar product: $\mathbf{N} \cdot \mathbf{L} = \|\mathbf{N}\| \|\mathbf{L}\| \cos \theta$; since the vectors are normalised $\|\mathbf{N}\| = 1$ and $\|\mathbf{L}\| = 1$; this is the Lambertian (diffuse) formula.

$\mathbf{N} \cdot \mathbf{L} \leq 0$ means that the light is *below the horizon*.

In summary,

$$I_d = I_d m_d \max(\mathbf{N} \cdot \mathbf{L}, 0). \quad (6.9)$$

Specular $I_s = I_s m_s (\mathbf{V} \cdot \mathbf{R})^s$, where $\mathbf{V}$ is the viewing direction and $\mathbf{R}$ is the *perfect* reflecting angle; $\mathbf{R}$ is calculated as $\mathbf{R} = 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}$.

As for *diffuse*, if the scalar product $(\mathbf{V} \cdot \mathbf{R}) \leq 0$, it is set to 0.

In summary,

$$I_d = I_s m_s \max((\mathbf{V} \cdot \mathbf{R}), 0)^s, \quad (6.10)$$

where,

$$\mathbf{R} = 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}. \quad (6.11)$$

Figure 6.3 gives a graphical depiction of typical values of $(\mathbf{V} \cdot \mathbf{R})^s$, for angles θ between $\mathbf{V}$ and $\mathbf{R}$ between $-90^\circ, 0, +90^\circ$ and for $s = 1$ (rather like diffuse, but not exactly), to $s = 50$, very shiny, with a very peaky angular reflectivity.

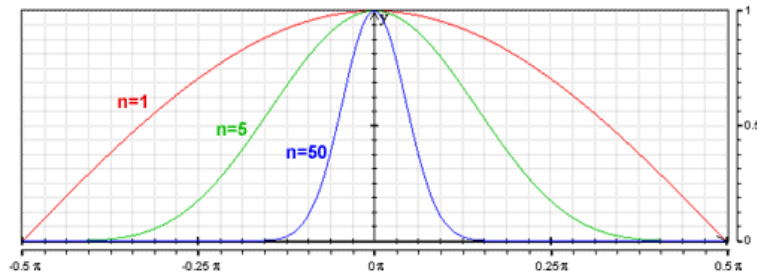


Figure 6.3: Phong specular lighting model; n in the figure is our shininess factor, s

Emissive

$$I_e = m_e. \quad (6.12)$$

Summary — Total Lighting Equation We can now summarise lighting by gathering together the contributions from eqns. 6.8 to 6.12 and summing for N_l lights,

$$I_{total} = m_e + \sum_{i=1}^{N_l} [I_a^i m_a + I_d^i m_d \max(\mathbf{N} \cdot \mathbf{L}^i, 0) + I_s^i m_s \max((\mathbf{V} \cdot \mathbf{R}^i), 0)^s]. \quad (6.13)$$

The superscript i for *light-index* is untidy, but we cannot have another subscript. The final s superscript stands for *to-the-power-of shininess*.

6.4 Additional Considerations

We now discuss additional factors and considerations that can make light simulations more realistic.

Colour Typically, the perceived colour of an object is associated with the *ambient* and *diffuse* colours. If you think about the colour of *specular* reflections, they often closely resemble the colour of the light source; in other words, the specular colour of the material should be *white*.

Attenuation The intensity of a real light decreases according to the distance from it. OpenGL models this using eqn. 6.14

$$att = \frac{1}{k_c + k_l d + k_q d^2}, \quad (6.14)$$

where

d = distance from the light to the shaded vertex,

k_c = GL_CONSTANT_ATTENUATION,

k_l = GL_LINEAR_ATTENUATION,

k_q = GL_QUADRATIC_ATTENUATION.

By default, $k_c = 1, k_l = k_q = 0$, i.e. $att = 1$, no attenuation. Below are examples of setting the attenuation coefficients:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);  
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);  
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Ambient, diffuse, and specular contributions are all attenuated; the emission and *global ambient* values are not.

If the light is a *directional* one, see below, the attenuation factor is always 1.

Directional Light Directional light simulates a light like the sun. Recall *homogeneous coordinates* and the added w component that is always 1 (or definitely non-zero) for points and 0 for vectors; specifying directional light is one occasion where you actually specify the fourth, w , coordinate, for example

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

Spotlights Spotlights emit light in a directional cone, see Figure 6.4, where c is the spotlight cutoff half-angle and A is the direction of the vertex in question.

By default $c = 180^\circ$ so that the light covers the full 360° .

The following line sets the cutoff parameter to 45 degrees and the next line specifies the spot exponent, t , the default of which is 0:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);  
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 1.0); // default 0
```

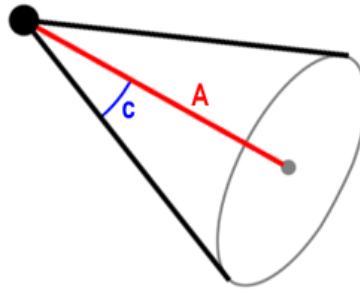


Figure 6.4: Spotlight.

You also need to specify a spotlight's direction, A in Figure 6.4:

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

The direction is specified in object coordinates. By default, the direction is $(0.0, 0.0, -1.0)$, so if you don't explicitly set the value of `GL_SPOT_DIRECTION`, the light points down the negative z-axis. Also, keep in mind that a spotlight's direction is transformed by the modelview matrix just as though it were a normal vector, and the result is stored in eye coordinates.

We can write an equation for the spotlight attenuation, sa ,

$$sa = (\mathbf{D} \cdot \mathbf{A})^t, \text{ if } \mathbf{D} \cdot \mathbf{A} \geq 0, \quad (6.15)$$

$$= 0, \text{ otherwise.}$$

where the light direction to the vertex is $\mathbf{D} = (\mathbf{q} - \mathbf{p}) / \|\mathbf{q} - \mathbf{p}\|$, $\mathbf{p}$ is the light position and $\mathbf{q}$ is the position of the vertex in question; $\mathbf{A}$ is the direction of the cone axis, see Figure 6.4.

Global Ambient Light As noted above, each light source can have an ambient component. Because of the nature of ambient light, it is often handy to have an ambient lighting that exists independently of any specific light. That is called *global ambient light* and is specified as follows:

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

As with emissive light, global ambient is not attenuated.

Total Lighting Equation Revised to Include Spotlight and Attenuation Factors We can now revise eqn. 6.13 to include spotlight and attenuation Factors

$$I_{total} = m_e + \sum_{i=1}^{N_l} sa^i att^i [I_a^i m_a + I_d^i m_d \max(\mathbf{N} \cdot \mathbf{L}^i, 0) + I_s^i m_s \max((\mathbf{V} \cdot \mathbf{R}^i), 0)^s]. \quad (6.16)$$

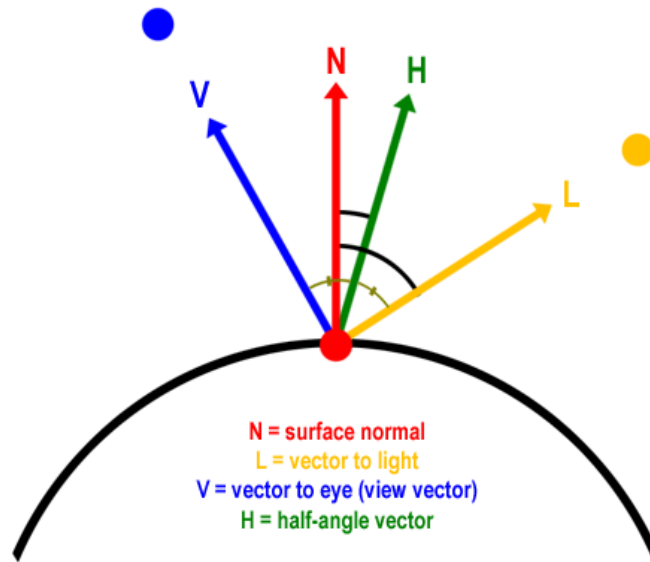


Figure 6.5: Blinn specular lighting model.

Blinn's Specular Lighting Eqn. 6.11 is expensive to compute, so Blinn introduced an approximation that is much easier to compute, see Figure 6.5. This is not used in OpenGL, but is used in some software rendering systems.

The Blinn specular term is,

$$I_d = I_s m_s \max((\mathbf{N} \cdot \mathbf{H}), 0)^s, \quad (6.17)$$

where $\mathbf{H}$ is the so-called *half-angle* (halfway between light direction $\mathbf{L}$ and viewing direction $\mathbf{V}$,

$$\mathbf{H} = (\mathbf{L} + \mathbf{V})/2. \quad (6.18)$$

6.5 Your first lighting program, light.cpp

The program `light.cpp` shown in Figures 6.8 and 6.9 displays the spheres shown in Figures 6.6 and 6.7. Figure 6.6 (a) demonstrates just *diffuse* lighting; the light is positioned out to the left on the x-axis. Figure 6.6 (b) demonstrates *diffuse* lighting and *specular* lighting together. In both figures we have included a small amount of *emissive* light, but you wouldn't notice it unless you switched off the others.

Figure 6.7 (a) demonstrates *diffuse* lighting, *specular* lighting, and ambient lighting together; finally, Figure 6.7 (b) does the same, but with the light in line with the camera and 20 units along the x-axis.

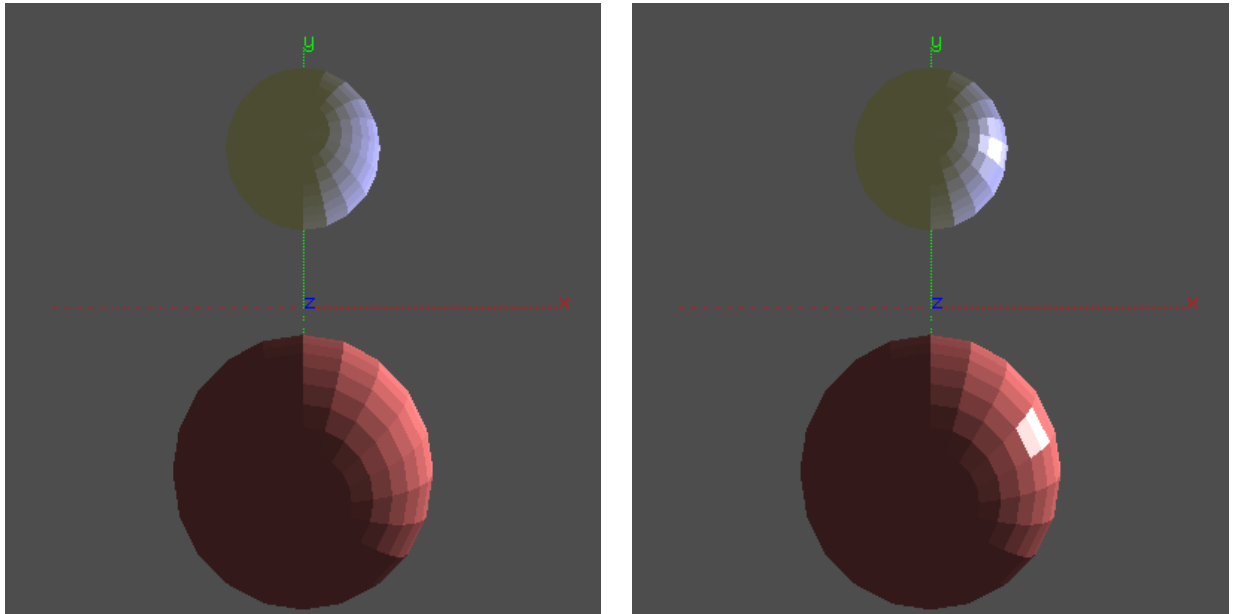


Figure 6.6: (a) Spheres, diffuse lighting; (b) diffuse and specular lighting.

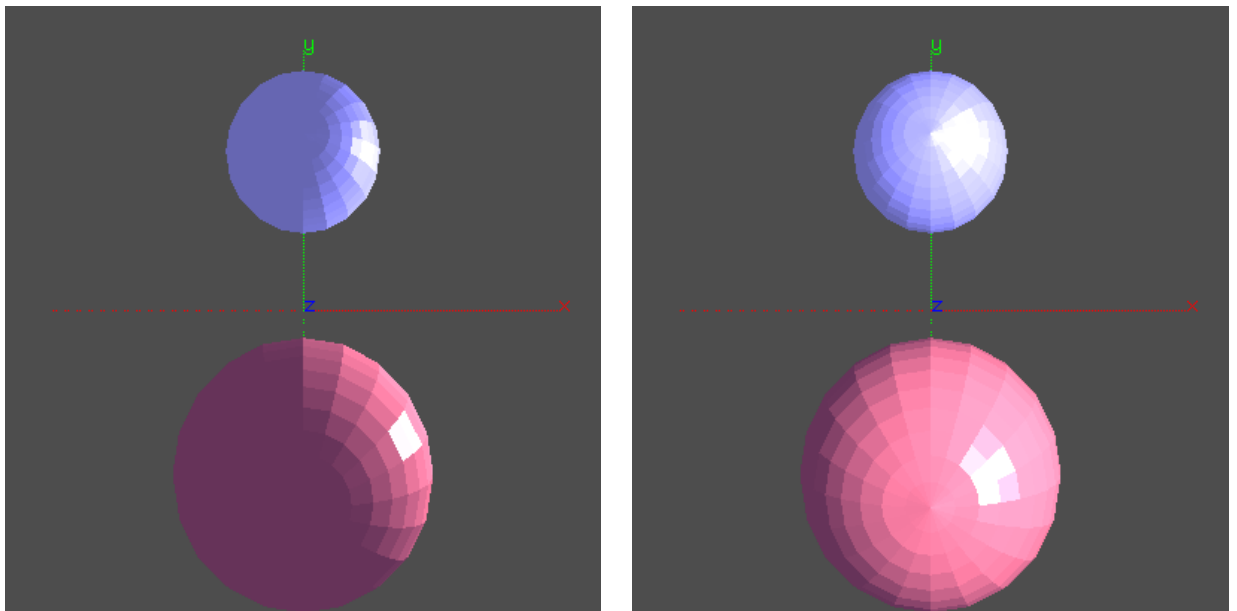


Figure 6.7: (a) Spheres, diffuse, specular and ambient lighting. (b) same with light near the camera.

```

GLfloat none[] = {0.0,0.0,0.0,1.0}; // zero light or colour

void initLights(){
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    GLfloat l0[] = {1.0,1.0,1.0,1.0}; // light is white
    GLfloat l0a[] = {0.2,0.2,0.5,1.0}; // ambient light is bluish

    glLightfv(GL_LIGHT0, GL_AMBIENT, l0a);
    //glLightfv(GL_LIGHT0, GL_AMBIENT, none);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, l0);
    //glLightfv(GL_LIGHT0, GL_DIFFUSE, none);
    glLightfv(GL_LIGHT0, GL_SPECULAR, l0);
    //glLightfv(GL_LIGHT0, GL_SPECULAR, none);

    //glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1);
    //glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05);
    glEnable(GL_LIGHT0);
}

```

Figure 6.8: Spheres with lighting, light.cpp (part 1).

```

void display(){
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    gluLookAt(0.,0.,25., // eye (camera position)
              0.,0.,0.,  // at
              0.,1.,0.); // up
    GLfloat l0Pos[] = {20.,0.,0.,1.};
    glLightfv(GL_LIGHT0, GL_POSITION, l0Pos);

    glDisable(GL_LIGHTING);
    double xNeg= -10., xPos= 10., yNeg = -10.,
           yPos = 10., zNeg = -10., zPos = 10.;
    drawAxes(xNeg, xPos, yNeg, yPos, zNeg, zPos);
    glEnable(GL_LIGHTING);

    glRotatef((GLfloat)roty,0.,1.,0.);
    glRotatef((GLfloat)rotx,1.,0.,0.);
    glPushMatrix();
    glTranslatef(0.,6.,0.); //move to where we want to put object
    GLfloat me1[] = { 0.2, 0.2, 0.5, 1.0 }; // emissive, bluish
    GLfloat ma1[] = { 0.5, 0.5, 1.0, 1.0 }; // ambient, bluish
    GLfloat md1[] = { 0.5, 0.5, 1.0, 1.0 }; // diffuse, bluish
    GLfloat ms1[] = { 1.0, 1.0, 1.0, 1.0 }; // specular, white
    GLfloat msh1[] = {50.0 }; // shininess
    glMaterialfv(GL_FRONT, GL_EMISSION, me1);
    glMaterialfv(GL_FRONT, GL_AMBIENT, ma1);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, md1);
    glMaterialfv(GL_FRONT, GL_SPECULAR, ms1);
    glMaterialfv(GL_FRONT, GL_SHININESS, msh1);
    glutSolidSphere(3.,20,20);
    glPopMatrix();
    glPushMatrix();
    glTranslatef(0.,-6.,0.);
    GLfloat ma2[] = { 1.0, 0.5, 0.5, 1.0 }; // ambient, redish
    GLfloat md2[] = { 1.0, 0.5, 0.5, 1.0 }; // diffuse, redish
    GLfloat ms2[] = { 1.0, 1.0, 1.0, 1.0 }; // specular, white
    GLfloat msh2[] = {128.0 }; // shininess
    glMaterialfv(GL_FRONT, GL_EMISSION, none);
    glMaterialfv(GL_FRONT, GL_AMBIENT, ma2);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, md2);
    glMaterialfv(GL_FRONT, GL_SPECULAR, ms2);
    glMaterialfv(GL_FRONT, GL_SHININESS, msh2);
    glutSolidSphere(5.,20,20);

    glPopMatrix();
    glPopMatrix ();  glutSwapBuffers();
}

```

Figure 6.9: Spheres with lighting, light.cpp (part 2).

6.5.1 Dissection of light.cpp

1. Lighting must be enabled: `glEnable(GL_LIGHTING);`
2. It can be disabled, e.g. temporarily, and then enabled again, as in the example of while axes are being drawn

```
glDisable(GL_LIGHTING);
double xNeg= -10., xPos= 10., yNeg = -10.,
      yPos = 10., zNeg = -10., zPos = 10.;
drawAxes(xNeg, xPos, yNeg, yPos, zNeg, zPos);
glEnable(GL_LIGHTING);
```

3. OpenGL allows up to eight lights to be in operation at any one time. Each must be enabled individually (and can be disabled individually):

```
glEnable(GL_LIGHT0); // light 0

glEnable(GL_LIGHT1);
...
glEnable(GL_LIGHT7);
```

4. Light colour parameters may be set as follows; here we make the *main* (diffuse and specular) light colour white; ambient light is made bluish. Note: although ambient light is associated with LIGHT0 just as diffuse and specular light,

```
GLfloat none[] = {0.0,0.0,0.0,1.0}; // zero light or colour
GLfloat l0[] = {1.0,1.0,1.0,1.0}; // light is white
GLfloat l0a[] = {0.2,0.2,0.5,1.0}; // ambient light is bluish

glLightfv(GL_LIGHT0, GL_AMBIENT, l0a);
//glLightfv(GL_LIGHT0, GL_AMBIENT, none);
glLightfv(GL_LIGHT0, GL_DIFFUSE, l0);
//glLightfv(GL_LIGHT0, GL_DIFFUSE, none);
glLightfv(GL_LIGHT0, GL_SPECULAR, l0);
//glLightfv(GL_LIGHT0, GL_SPECULAR, none);
```

5. Other light parameters may be set, for example:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05);
```

We could give loads of examples of other parameters, but it will be easier to look at examples in tutorials and practical classes and in coursework assignments.

6. If the light parameters mentioned above are fixed (for the duration of the program execution), then it makes sense to initialise them only once, for example in a (typical) `init` function that gets called once.

But light position is a different matter.

7. Light position. Here we set light position; in the current state of `light.cpp`, this is in `display` and just after `gluLookAt` is called:

```
gluLookAt(0.,0.,25., // eye (camera position)
          0.,0.,0., // at
          0.,1.,0.); // up
GLfloat l0Pos[] = {20.,0.,0.,1.};
glLightfv(GL_LIGHT0, GL_POSITION, l0Pos);
```

From earlier chapters you will be aware that `gluLookAt` alters the modelview matrix and that every vertex afterwards is positioned with respect to that modelview matrix (plus any other modifications that we make to the modelview matrix).

I will not complicate the story here by reiterating what `gluLookAt` does (**Ex.** What is the effect of the above call to `gluLookAt`?) but suffice to say that the *centre of the world* is moved and any every vertex afterwards is positioned with respect to that centre.

The same is the case for light position. In

```
GLfloat l0Pos[] = {20.,0.,0.,1.};
glLightfv(GL_LIGHT0, GL_POSITION, l0Pos);
```

the light is positioned 20 units along the x-axis.

8. Light positioned before `gluLookAt` called.

If we had positioned the light in `initLights`—, which is called in `init`, i.e. before we have made any alteration to the modelview matrix (which at that stage would be the identity matrix), then the light would have been positioned with respect to the *centre of the world* **then**; we will experiment with this, but we can say that the light would have been 20 x-axis units away from the camera centre (eye), and probably not what we wanted, see Figure 6.7 (b).

9. Light on camera? If we want to simulate a light on the camera, e.g. as in a *first person shooter* with a torch, then we set the light position at (0, 0, 0, 1) and do this *before* `gluLookAt` is called.

If you need more on this, see *Controlling a Light's Position and Direction* in

<http://www.glprogramming.com/red/chapter05.html>

See also the relevant section of the OpenGL FAQ at:

<http://www.opengl.org/resources/faq/technical/lights.htm>

10. Directional light versus positional light. You can see in Figures 6.6 and 6.7 that the light is shining up at the top sphere and down at the lower one, i.e. it is 20 units along the x-axis away from the centre point between them. If you want the light to appear *directional*, i.e. infinitely far away like the sun, then you would use:

```
GLfloat l0Pos[] = {20.,0.,0.,0.};
// or
GLfloat l0Pos[] = {1.,0.,0.,0.}; // would have the same effect
```

When the w (fourth) coordinate is 1, we have positional light; when the w (fourth) coordinate is 0, we have directional light. Recall from (Campbell 2008a) that in *homogeneous* coordinates, $w = 1$ means a *point* (position) while $w = 0$ means a vector (thus direction).

In section 6.6 we give a further example of setting materials and light parameters.

There is a nice table (Table 3.1, pages 51-52) in (McReynolds & Blythe 2005) giving parameters of common materials; you may be able to find a version of the table on the web; or, ask me for a photocopy.

In section 6.7 we give an example of a light moving around a scene.

Default values for lights If you do not explicitly set light parameters, and you enable lighting and one or more lights, the default parameters shown in Figure 6.10 are used.

Parameter Name	Default Value	Meaning
-----	-----	-----
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light *
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light *
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SHININESS	0.0	specular shininess
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

* The default values listed for GL_DIFFUSE and GL_SPECULAR in apply only to GL_LIGHT0. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR. In other words, if you specify no light parameters, but yet enable the light, for GL_LIGHT0 you will have bright white diffuse and specular lighting and no ambient lighting. For GL_LIGHT1 ... GL_LIGHT7 all lighting defaults to zero.

Figure 6.10: Default values for lights

6.6 Further example of materials and lights, material.cpp

The program shown partially in Figure 6.12 (code for first row only), `material.cpp`, gives further examples of different materials and lights, see Figure 6.11.

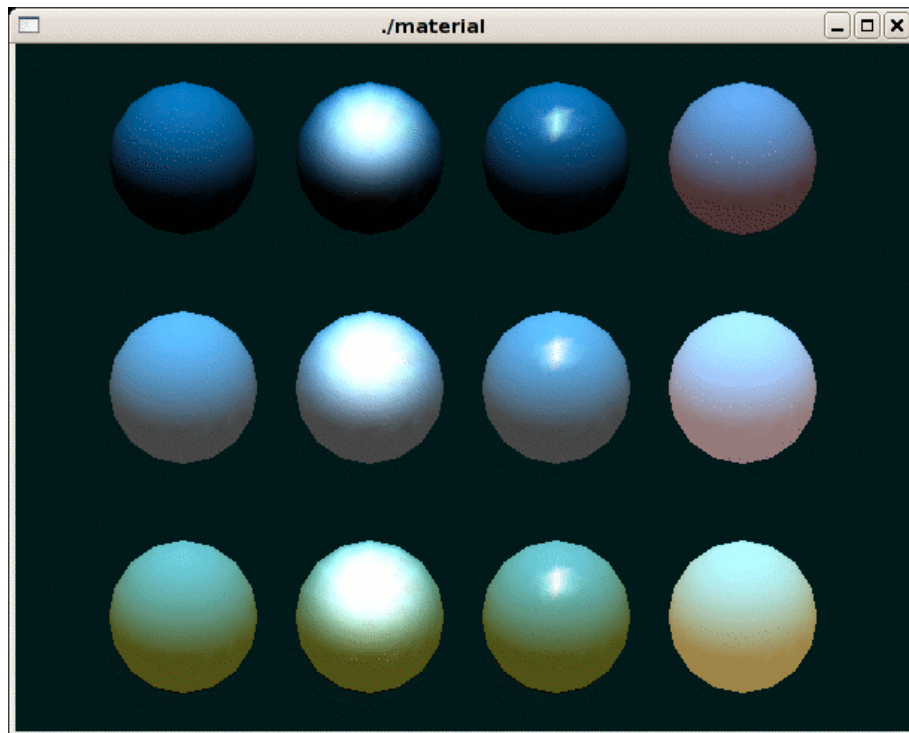


Figure 6.11: Different materials and lights.

```

GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat no shininess[] = { 0.0 }; GLfloat low shininess[] = { 5.0 };
GLfloat high shininess[] = { 100.0 };
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//diffuse reflection only; no ambient or specular
glPushMatrix();   glTranslatef (-3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();

//diffuse and specular reflection; low shininess; no ambient
glPushMatrix();   glTranslatef (-1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();

//diffuse and specular reflection; high shininess; no ambient
glPushMatrix();   glTranslatef (1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, high shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();

//diffuse reflection; emission; no ambient or specular refl.
glPushMatrix();   glTranslatef (3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();

```

6.7 Example of moving light, movelight.cpp

The program shown partially in Figure 6.14 `movelight.cpp`, gives an example of a light moving around a scene, see Figure 6.13.

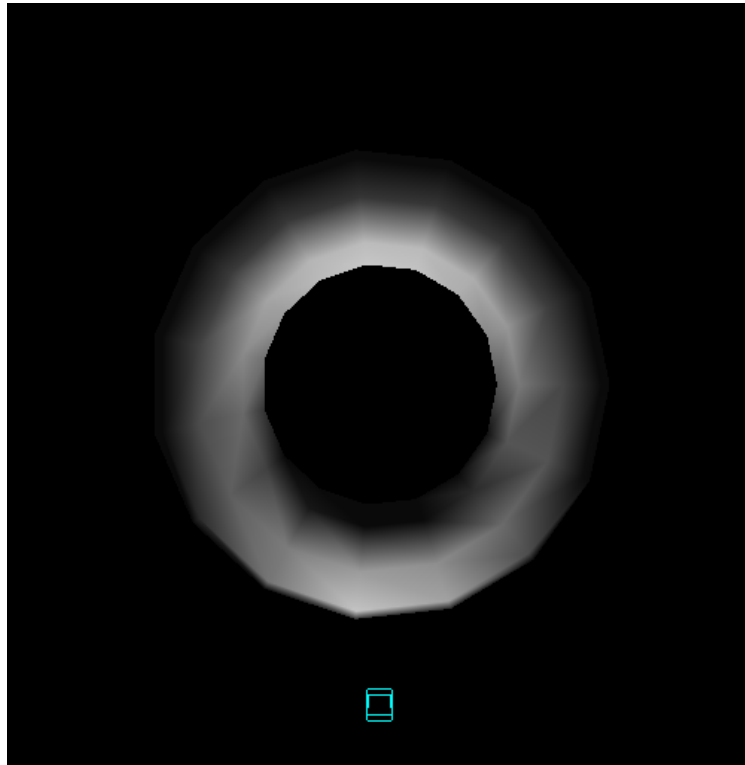


Figure 6.13: Moving light.

```

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

/* Here is where the light position is reset after the modeling
 * transformation (glRotated) is called. This places the
 * light at a new position in world coordinates. The cube
 * represents the position of the light.
 */
void display(void){
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    glPushMatrix ();
    glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);

    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
    glLightfv (GL_LIGHT0, GL_POSITION, position);

    glTranslated (0.0, 0.0, 1.5);
    glDisable (GL_LIGHTING);
    glColor3f (0.0, 1.0, 1.0);
    glutWireCube (0.1);
    glEnable (GL_LIGHTING);
    glPopMatrix ();
    // torus at centre of 'world'
    glutSolidTorus (0.275, 0.85, 8, 15);
    glPopMatrix ();
    glutSwapBuffers();
}

void mouse(int button, int state, int x, int y){
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {
                spin = (spin + 30) % 360;
                glutPostRedisplay();
            }
            break;
    }
}

```

Figure 6.14: Moving light, movelight.cpp.

6.8 Simplifying Materials Specification using glColorMaterial

It is said that `glMaterialfv` consumes a lot of computing power; hence, there was a search for more efficient methods of setting material reflectivity properties. `glColorMaterial` allows you to set material colours using `glColor3*`.

There is good coverage of this topic in:

http://www.sjbaker.org/steve/omniv/opengl_lighting.html

It is used as follows:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* now glColor* changes diffuse reflection */

/* OR, even more useful */
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
/* now glColor* changes ambient and diffuse reflection */

glColor3f(0.2, 0.5, 0.8);
/* draw some objects here */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* glColor* no longer changes ambient / diffuse reflection */
/* now glColor* changes specular reflection */
glColor3f(0.9, 0.0, 0.2);
/* draw other objects here */
glDisable(GL_COLOR_MATERIAL);
```

We note that, in the absence of `glEnable(GL_COLOR_MATERIAL)`, in scenes with lighting, `glColor3*` would have **no** effect.

6.9 Normals

In the examples given, we have always used OpenGL supplied objects such as `glutSolidSphere`; these ensure that a *normal vector* is computed for each vertex; a quick look at eqn. 6.13, and common sense, indicates why we need surface normals any time we want to use diffuse or specular lighting.

An appendix to chapter 2 of the Red Book tells you all you need to know about computing normals.

<http://www.glprogramming.com/red/appendixe.html>

Note also the problem with normals that they can be distorted when non-uniform scaling is applied (i.e. a scaling in which the scale factor for x, y, and z are not the same), see (Foley, van Dam, Feiner & Hughes 1990), chapter 5.6.

If you need revision on use of the *vector product* (cross product) to compute normals, see Chapter 3 of our Maths. notes and mention it in class. If we have two points, $\mathbf{p}_1$ and $\mathbf{p}_2$ on a polygon face, then, treating them as vectors, the normal to the face is given by the vector product,

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_2. \quad (6.19)$$

OpenGL **requires** that normals be *normalised* (term not related), i.e. reduced to unit length / magnitude; otherwise you will get silly results.

When using OpenGL *evaluators* (see chapter 12) you can ask OpenGL to automatically generate normals:

```
glEnable(GL_AUTO_NORMAL);
```

Here is a quick and dirty vector (cross) product that I've added to my Vector4D class (you should have an earlier version of Vector4D in the mathematics notes I gave you.

```
Vector4D Vector4D::cross(Vector4D v){
    Vector4D v1= Vector4D();
    v1.d_[0]= d_[1]* v.d_[2] - d_[2]* v.d_[1];
    v1.d_[1]= d_[2]* v.d_[1] - d_[1]* v.d_[2];
    v1.d_[2]= d_[0]* v.d_[1] - d_[1]* v.d_[0];
    v1.d_[3]= 0.0;
    double mag = v1.length();
    assert(mag > 1.0e-10);
    Vector4D v2 = v1.scale(1.0/mag);
    return v2;
}

Vector4D Vector4D::scale(double s){
    Vector4D v1= Vector4D();
    for(size_t i= 0; i< n_; i++)v1.d_[i] = s*d_[i];
    return v1;
}

double Vector4D::length(){
    double len= 0.0;
    for(size_t i= 0; i< n_; i++)len+= d_[i]*d_[i];
    return sqrt(len);
}
```

And here is how it is used to compute a normal in the house3d program.

```

void house() {
    Vector4D v1, v2, v3;
    float vf[5][3] = {{+1.0, +1.0, +1.0},
                      {-1.0, +1.0, +1.0},
                      {-1.0, -1.0, +1.0},
                      {+1.0, -1.0, +1.0},
                      {0.0, 2.0, 1.0}    };

    float vr[5][3];
    // rear = front with z = -1
    for(int i = 0; i < 5; i++){
        vr[i][0] = vf[i][0];    vr[i][1] = vf[i][1];    vr[i][2] = -1.0;
    }
    float nFront[3] = {0.0, 0.0, 1.0};
    float nRear[3] = {0.0, 0.0, -1.0};
    float nRight[3] = {1.0, 0.0, 0.0};
    float nLeft[3] = {-1.0, 0.0, 0.0};
    float nRightRoof[3] = {0.5, 0.5, 0.0};
    float nLeftRoof[3] = {-0.5, 0.5, 0.0};
    /*front*/
    glColor3f (1.0, 0.0, 0.0);
    glBegin(GL_QUADS);
        v1 = Vector4D(vf[0][0], vf[0][1], vf[0][3], 1.0);
        v2 = Vector4D(vf[1][0], vf[1][1], vf[1][3], 1.0);
        v3 = v1.cross(v2);
        cout<< "v1 = "<< v1<< endl;
        cout<< "v2 = "<< v2<< endl;
        cout<< "v3 = "<< v3<< endl;
        for(int j = 0; j < 3; j++)nFront[j] = v3.d_[j];
        printArray("nFront", nFront, 3);
        glNormal3fv(nFront);
        for(int i = 0; i < 4; i++)glVertex3fv(vf[i]);
    glEnd();
    // etc. ...
}

```

The output from that fragment is:

```
v1 = ( 1.000, 1.000, -1.000, 1.000)
v2 = (-1.000, 1.000, -1.000, 1.000)
v3 = ( 0.000, 0.000, 1.000, 0.000)
nFront 0 0 1
```

The front of the house *does* point in the positive z-direction.

We note the warning in (Hill 2008), page 292, on the possible inaccuracy of this simple method of computing the vector product.

6.9.1 Example using the 3D House

```
void display(void){
    glLoadIdentity ();
    //gluLookAt(eyeX, eyeY, eyeZ, 0.0, 0.0, atZ, 0.0, 1.0, 0.0);
    glTranslated(-2.0, -3.0, -12.0);

    glDisable(GL_LIGHTING);
    glDisable(GL_COLOR_MATERIAL);
    drawAxes(-4., 2.8, -4., 2.8, -4., 4.5);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    GLfloat l_amb[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat l_dif[] = { 0.2, 1.0, 0.2, 1.0 };
    GLfloat l_spc[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat m_spc[] = { 1.0, 0.8, 0.8, 1.0 };
    GLfloat m_shn[] = { 5.0 };

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, l_amb);
    GLfloat l_pos[] = { 20.0, 20.0, 20.0, 0.0 };
    glLightfv (GL_LIGHT0, GL_POSITION, l_pos);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, l_dif);
    glLightfv (GL_LIGHT0, GL_SPECULAR, l_spc);

    glMaterialfv(GL_FRONT, GL_SPECULAR, m_spc);
    glMaterialfv(GL_FRONT, GL_SHININESS, m_shn);

    house();

    glPushMatrix();
    glTranslated(4.0, 5.0, 0.0);
    glRotated(50.0, 1.0, 0.0, 0.0);
    //house();
}
```

```

glutSolidTeapot(1.0);
glPopMatrix();

glPushMatrix();
glTranslated(5.0, 0.0, 0.0);
glRotated(30.0, 0.0, 1.0, 0.0);
house();
glPopMatrix();

glPushMatrix();
glTranslated(0.0, 5.0, 0.0);
glScaled(1.0, 1.0, 1.0);
house();
glPopMatrix();

```

And the output is shown in Figure 6.15.

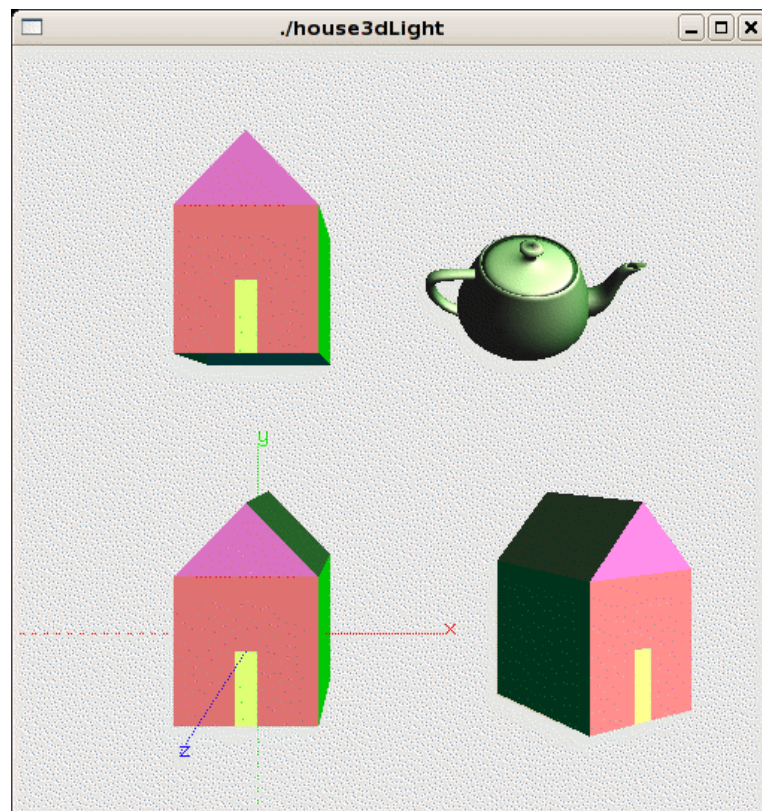


Figure 6.15: 3D House with Lighting.

6.10 Lighting Calculation Example.

This is a model answer to an exercise I sometimes set on examinations or in coursework.

1. Consider the brass teapot and the light; no emissive light; assume that the light is directional and that there is no attenuation whatsoever.

```
// Brass, see McReynolds p. 51
GLfloat m_am2[] = { 0.329, 0.223, 0.027, 1.0 };
GLfloat m_di2[] = { 0.78, 0.57, 0.114, 1.0 };
GLfloat m_sp2[] = { 0.99, 0.94, 0.81, 1.0 };
GLfloat m_sh2[] = { 27.89 };
```

Light.

```
GLfloat l_am[] = { 0.4, 0.4, 0.4, 1.0 };
GLfloat l_di[] = { 0.8, 0.8, 0.8, 1.0 };
GLfloat l_sp[] = { 0.8, 0.8, 0.8, 1.0 };
```

- (a) Compute the ambient colour.

```
GLfloat m_am2[] = { 0.329, 0.223, 0.027, 1.0 };
GLfloat l_am[] = { 0.4, 0.4, 0.4, 1.0 };
-----
ambient colour    0.1316  0.0892  0.0108
```

- (b) Assuming that the light is in line with the normal, compute the diffuse colour. Draw a diagram.

$\mathbf{L} \cdot \mathbf{N} = |\mathbf{L}| |\mathbf{N}| \cos \theta$; $\theta = 0^\circ$; since $\mathbf{L}$ and $\mathbf{N}$ are normalised (see Maths notes, chapter 3), $|\mathbf{L}| = 1$ and $|\mathbf{N}| = 1$, so $\mathbf{L} \cdot \mathbf{N} = \cos \theta = 1$.

```
GLfloat m_di2[] = { 0.78, 0.57, 0.114, 1.0 };
GLfloat l_di[] = { 0.8, 0.8, 0.8, 1.0 };
-----
                0.624  0.456  0.0912
angle correction = 1
```

- (c) Assuming that the light vector is 30° away from the normal, compute the diffuse colour. Draw a diagram.

See above, angle correction = $\cos 30^\circ$.

```
                0.624  0.456  0.0912    x 0.866 =
cos 30 = 0.866    0.540  0.395  0.0790
```

- (d) Assuming that the light vector is 85° away from the normal, compute the diffuse colour. Draw a diagram.

```
                0.624  0.456  0.0912    x 0.866 =
cos 85 = 0.087    0.054  0.040  0.0079
```

- (e) Assuming that the viewing angle is in line with the mirror reflection angle, compute the specular colour. Draw a diagram.

$\mathbf{V} \cdot \mathbf{R} = |\mathbf{V}| |\mathbf{R}| \cos \theta$; $\theta = 0^\circ$; since $\mathbf{V}$ and $\mathbf{R}$ are normalised (see Maths notes, chapter 3), $|\mathbf{V}| = 1$ and $|\mathbf{R}| = 1$, so $\mathbf{V} \cdot \mathbf{R} = \cos \theta = 1$. Correction is $S = 1^{28} = 1$.

```
GLfloat m_sp2[] = { 0.99, 0.94, 0.81, 1.0 };
                                   GLfloat m_sh2[] = { 27.89 };
GLfloat l_sp[] = { 0.8, 0.8, 0.8, 1.0 };
-----
                0.792 0.752 0.648
```

correction = 1

- (f) Assuming that the viewing angle is 60° away from the mirror reflection angle, compute the specular colour. Draw a diagram. I changed the original 45° to 60° , but if anyone had not heard, the 45° answer is fine. With $\cos 60^\circ = 0.5$, you can work out the power in your head, provided you know that $2^{32} \approx 4.2 \times 10^9$, and anyone working with computers should know that.

```
GLfloat m_sp2[] = { 0.99, 0.94, 0.81, 1.0 };
                                   GLfloat m_sh2[] = { 27.89 };
GLfloat l_sp[] = { 0.8, 0.8, 0.8, 1.0 };
-----
                0.792 0.752 0.648
```

$\cos 60 = 0.5$, so correction = $0.5^{28} = (1/2)^{28} = 1/(2^{28})$.

$2^{32} \approx 4.2 \times 10^9$, so $2^{28} = 4.2 \times 10^9 / 16 = 0.26 \times 10^9$

so $0.5^{28} = (1/2)^{28} = 1/(2^{28}) = 1 \times 10^{(-9)}/0.26 \approx 4 \times 10^{(-9)}$

so corrected specular colour ≈ 0 .

- (g) Using your results from (a), (b) and (e), what would be the total colour as seen by the viewer?

```
(a)          0.132 0.089 0.0108
(b)          0.624 0.456 0.0912
(e)          0.792 0.752 0.648
-----      add
          1.548 1.297 0.750
```

colour values are *clamped* to 0..1 (you cannot have any colour greater than 1, or less than 0, so final colour is

```
1.0  1.0  0.75
```

Chapter 7

Blending, Antialiasing, and Fog

This chapter covers additional details on: materials, namely blending for transparent materials; antialiasing (which uses blending), which we've mentioned before; and fog, which can be used in its own right, or to give additional impression of depth.

We take a lot of these notes from from (Shreiner et al. 2008a) Chapter 6. Chapter 6 of (Wright et al. 2007) is also good and has some fine examples; note that all examples from (Wright et al. 2007) are on my public folder.

7.1 Blending

When *fragments* corresponding to two or more opaque scene objects (i.e. objects in the 3D model), when projected, are potential occupants of a pixel on the screen, the choice is made as follows: (i) the default, whatever is drawn last takes over; (ii) if depth testing is enabled (recall `glEnable(GL_DEPTH_TEST);`), new colours replace the current colour only if, according to the depth buffer (z-buffer) they are closer to the viewer.

That is up to now. But now we cover blending, in which we allow for transparent / translucent materials. Blending is enabled with `glEnable(GL_BLENDING);` and blending of two colours typically depends on the fourth colour factor, *alpha*. One can select from a whole raft of combination functions using `glBlendFunc`. Typically, if *alpha* = 1 the colour is opaque and its fragment will clobber anything behind it (depth testing) or before it (no depth testing), but `glBlendFunc` gives an array of choices to alter this.

7.2 Your first blending program, alpha2.c

The program `alpha2.c` shown in Figure 7.2 displays the triangles and rectangles shown in Figure 7.1.

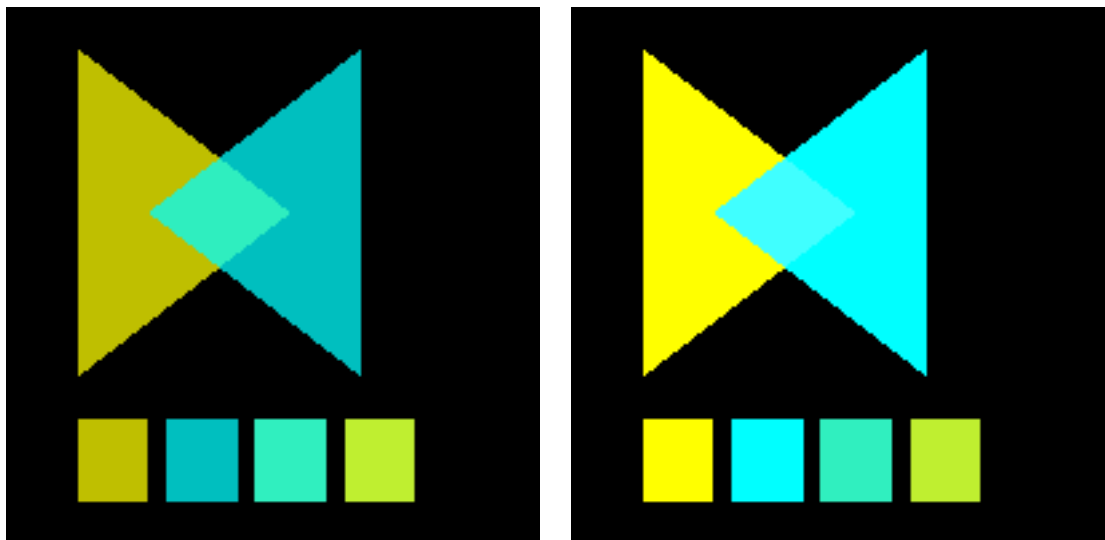


Figure 7.1: (a) Yellow triangle (left) and cyan triangle; drawing order affects overlap. Rectangles for comparison: yellow; cyan; yellow then cyan; cyan then yellow. (b) blending effectively disabled by choosing source = *GL_ONE*.

```

static int leftFirst = GL_TRUE;
/* Initialize alpha blending function.
*/
static void init(void){
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    //glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
    glShadeModel (GL_FLAT);
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

static void drawLeftTriangle(void){
    glBegin (GL_TRIANGLES);
    glColor4f(1.0, 1.0, 0.0, 0.75); //yellow
    glVertex3f(0.1, 0.9, 0.0);
    glVertex3f(0.1, 0.1, 0.0);    glVertex3f(0.7, 0.5, 0.0);
    glEnd();
}

static void drawRightTriangle(void){
    glBegin (GL_TRIANGLES);
    glColor4f(0.0, 1.0, 1.0, 0.75); //cyan
    glVertex3f(0.9, 0.9, 0.0);
    glVertex3f(0.3, 0.5, 0.0);    glVertex3f(0.9, 0.1, 0.0);
    glEnd();
}

void display(void){
    glClear(GL_COLOR_BUFFER_BIT);
    if (leftFirst) {
        drawLeftTriangle();    drawRightTriangle();
    }
    else {
        drawRightTriangle();    drawLeftTriangle();
    }
    //yellow
    glColor4f(1.0, 1.0, 0.0, 0.75); glRectf(0.1, -0.2, 0.3, 0.0);
    //cyan
    glColor4f(0.0, 1.0, 1.0, 0.75); glRectf(0.35, -0.2, 0.55, 0.0);

    //background + yellow + cyan
    glColor4f(0.25*0.75, 0.75+0.25*0.75, 0.75, 1.0);
    glRectf(0.6, -0.2, 0.8, 0.0);

    //background + cyan + yellow
    glColor4f(0.75, 0.75+0.25*0.75, 0.25*0.75, 1.0);
    glRectf(0.85, -0.2, 1.05, 0.0);
}

```

Figure 7.2: Blending colours, alpha2.c.

Dissection of alpha2.c

1. Figure 7.2 shows just `init`, `display` and the supporting triangle drawing functions.
2. The program displays a yellow triangle on the left and cyan triangle on the right with some overlap; drawing order affects the colour of the overlap. We use

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
//          source weight S destination weight D
//glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
glShadeModel (GL_FLAT);
glClearColor (0.0, 0.0, 0.0, 0.0);
```

3. Blending is enable using `glEnable (GL_BLEND)`;
4. Rectangles are shown for comparison. From the left: yellow; cyan; yellow then cyan; cyan then yellow.
5. The right hand graphic in Figure 7.2 shows the effect of using

```
glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

This effectively stops blending, as will be explained shortly.

6. When running the program, pressing `t/T` changes the order of display and redisplay.

How is blending done? OpenGL blends by accumulating colour, from what it calls the *source* (C_s), into the current value in display buffer, which it calls the *destination* (C_d). It adds C_s to C_d and assigns the result to C_d , i.e. $C_d = C_sS + C_dD$, where S and D are blending factors chosen by `glBlendFunc`.

Simple examples are easy to explain. Let us first take the example from (Wright et al. 2007), p. 231, and assume that we use a common choice of blending function:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The latter means that the source blending weight $S = A_s$, the alpha of the source colour and that $D = 1 - A_s$.

Assume that the destination (frame buffer) currently contains $C_d = (R_d = 1, G_d = 0, B_d = 0, A_d = 0)$ and $C_s = (R_s = 0, G_s = 0, B_s = 1, A_s = 0.5)$. So, $S = A_s = 0.5$ and $D = 1 - A_s = 0.5$. Now, blending, we have

$$C_d = (R_d, G_d, B_d, A_d) = (R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a), \quad (7.1)$$

where we allow the most general situation, i.e. that blending factors are different for each colour.

In the case of the example we have

$$\begin{aligned} C_d &= (R_d, G_d, B_d, A_d) \\ &= (R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a) \\ &= (0 * 0.5 + 1 * 0.5, 0 * 0.5 + 0 * 0.5, 1 * 0.5 + 0 * 0.5, 0.5 * 0.5 + 0.5 * 0.5) \\ &= (0.5, 0, 0.5, 0.5). \end{aligned} \quad (7.2)$$

Note: $A_s = 0.5$ makes for an easy explanation, but some details are swept aside.

A more complex example In the case of the example in Figure 7.2, we have, after the buffer has been cleared,

$$C_d = (R_d = 0, G_d = 0, B_d = 0, A_d = 0).$$

Now let us blend in

$$C_s = (R_s = 1, G_s = 1, B_s = 0, A_s = 0.75)$$

with blending weights

$$S = 0.75, D = 1 - 0.75 = 0.25.$$

We get

$$C_d = (R_d = 0.75, G_d = 0.75, B_d = 0, A_d = 0.75 * 0.75).$$

Next, we blend in

$$C_s = (R_s = 0, G_s = 1, B_s = 1, A_s = 0.75).$$

We get

$$C_d = (R_d = 0.25 * 0.75, G_d = 0.75 + 0.25 * 0.75, B_d = 0.75, A_d = \dots).$$

The rectangle

```
//background + yellow + cyan
glColor4f(0.25*0.75, 0.75+0.25*0.75, 0.75, 1.0);
glRectf(0.6, -0.2, 0.8, 0.0);
```

allows us to see that this is correct. Recall that this is yellow first, then blend in cyan.

On the other hand, if we display in the opposite order, cyan first, then blend in yellow, we get

$$C_d = (R_d = 0.75, G_d = 0.75 + 0.25 * 0.75, B_d = 0.25 * 0.75, A_d = \dots).$$

The rectangle

```
//background + cyan + yellow
glColor4f(0.75, 0.75+0.25*0.75, 0.25*0.75, 1.0);
glRectf(0.85, -0.2, 1.05, 0.0);
```

allows us to see that this is correct.

Ex. Show that

```
glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
           source      destination
```

effectively removes blending. Hint. What is S ? What is $D = 1 - S$?

7.3 Three-Dimensional Blending with the Depth Buffer

Read this section in (Shreiner et al. 2008a) Chapter 6 and examine and execute the example `alpha3D.c`.

Order of blending is important. Depth buffer needed. But depth buffer can get confused by transparent / translucent objects.

Draw opaque objects first. Then make depth buffer *read-only* and draw translucent objects.

7.4 Antialiasing

See (Shreiner et al. 2008a); diagrams and code and most of the text taken from there.

Figure 7.3(a) shows the jagged effect of *aliasing*, i.e. the effects of *rasterisation* becoming visible. Figure 7.3(b) shows how *antialiasing* removes or reduces the visual effect; although close up views like this are unimpressive, antialiasing does work as you will see in the example program.

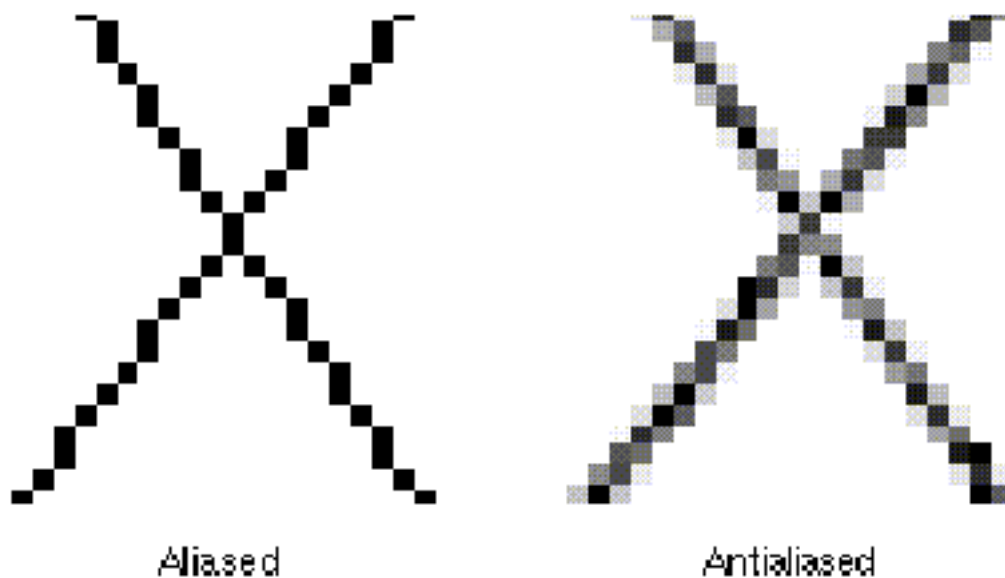


Figure 7.3: (a) Aliasing; (b) Antialiasing.

Antialiasing works by colouring pixels according to the amount of coverage by the line, see Figure 7.4. When using RGBA mode, OpenGL calculates the coverage value for each fragment, and then multiplies the fragment's alpha value by the coverage value.

When RGBA colours are used, we must enable blending, see section 7.1.

Figure 7.5 gives a demonstration of RGBA antialiasing.

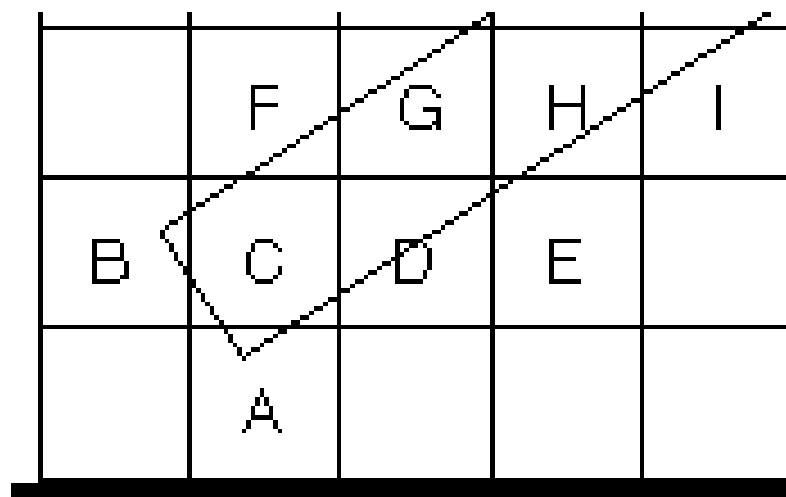


Figure 7.4: Antialiasing by coverage.

```

void init(void){
    GLfloat values[2];
    glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
    printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n", values[0]);

    glGetFloatv (GL_LINE_WIDTH_RANGE, values);
    printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
        values[0], values[1]);

    glEnable (GL_LINE_SMOOTH);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth (1.5);

    glClearColor(0.0, 0.0, 0.0, 0.0);
}

/* Draw 2 diagonal lines to form an X
*/
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f (0.0, 1.0, 0.0);
    glPushMatrix();
    glRotatef(-rotAngle, 0.0, 0.0, 0.1);
    glBegin (GL_LINES);
        glVertex2f (-0.5, 0.5);
        glVertex2f (0.5, -0.5);
    glEnd ();
    glPopMatrix();

    glColor3f (0.0, 0.0, 1.0);
    glPushMatrix();
    glRotatef(rotAngle, 0.0, 0.0, 0.1);
    glBegin (GL_LINES);
        glVertex2f (0.5, 0.5);
        glVertex2f (-0.5, -0.5);
    glEnd ();
    glPopMatrix();

    glFlush();
}

```

Figure 7.5: Antialiasing, aargb.c.

Dissection of aargb.c

1. The following prints implementation detail connected with line drawing.

```
GLfloat values[2];
glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n", values[0]);

glGetFloatv (GL_LINE_WIDTH_RANGE, values);
printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
        values[0], values[1]);
```

My implementation gives:

```
GL_LINE_WIDTH_GRANULARITY value is 0.1
GL_LINE_WIDTH_RANGE values are 1.0 10.0
```

In other words, one can specify line width using `glLineWidth` for values between 1 and 10 in steps of 0.1.

2. `glEnable (GL_LINE_SMOOTH);` enables antialiasing.
3. Next, we enable blending, see section 7.1; and we use the normal blending function that was used in section 7.1.

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

4. `glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);`

Figure 7.6 shows the values for Use with `glHint` and their meanings.

GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT
Specify the desired sampling quality of points, lines, or polygons
during antialiasing operations.

GL_FOG_HINT Specifies whether fog calculations are done per pixel
(GL_NICEST) or per vertex (GL_FASTEST).

GL_PERSPECTIVE_CORRECTION_HINT Specifies the desired quality of
colour and texture-coordinate interpolation.

Figure 7.6: Values used in `glHint`

7.5 Fog

...depth ...simulation of real fog, haze, pollution ...

...objects fade into fog colour depending on distance from viewpoints ...

...density of fog determines the rate of fade ...

...fog applied *after* matrix transformations, lighting, texturing (see Chapter 10) ...

...can improve speed performance, because no need to render highly fogged objects ...

See Nate Robin's Fog example in his Tutors.

Figure 7.7 shows the use of fog.

```

static GLint fogMode;
/* Initialize depth buffer, fog, light source,
 * material property, and lighting model.
 */
static void init(void){
    GLfloat position[] = { 0.5, 0.5, 3.0, 0.0 };

    glEnable(GL_DEPTH_TEST);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    {
        GLfloat mat[3] = {0.1745, 0.01175, 0.01175};
        glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
        mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
        glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
        mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
        glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
        glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
    }
    glEnable(GL_FOG);
    {
        GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

        fogMode = GL_EXP;
        glFogi (GL_FOG_MODE, fogMode);
        glFogfv (GL_FOG_COLOR, fogColor);
        glFogf (GL_FOG_DENSITY, 0.35);
        glHint (GL_FOG_HINT, GL_DONT_CARE);
        glFogf (GL_FOG_START, 1.0);
        glFogf (GL_FOG_END, 5.0);
    }
    glClearColor(0.5, 0.5, 0.5, 1.0); /* fog color */
}

static void renderSphere (GLfloat x, GLfloat y, GLfloat z){
    glPushMatrix();
    glTranslatef (x, y, z);
    glutSolidSphere(0.4, 16, 16);
    glPopMatrix();
}

/* display() draws 5 spheres at different z positions.*/
void display(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    renderSphere (-2., -0.5, -1.0); renderSphere (-1., -0.5, -2.0);
    renderSphere (0., -0.5, -3.0); renderSphere (1., -0.5, -4.0);
    renderSphere (2., -0.5, -5.0); glFlush();
}

```

Figure 7.7: Antialiasing, fog.c.

Dissection of fog.c

1. The fog related part of `fog.c` is contained in the following fragment of code.
2. Note the use of the block `{ ... }` to isolate the declaration of `fogColor`; in C, unlike C++, variables can be declared only at the beginning of a block; variable declared at the beginning of a block are *visible* only in that block; their *lifetime* begins when control enters that block and ends when control leaves the block.

```
glEnable(GL_FOG);
{
    GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

    fogMode = GL_EXP;
    glFogi (GL_FOG_MODE, fogMode);
    glFogfv (GL_FOG_COLOR, fogColor);
    glFogf (GL_FOG_DENSITY, 0.35);
    glHint (GL_FOG_HINT, GL_DONT_CARE);
    glFogf (GL_FOG_START, 1.0);
    glFogf (GL_FOG_END, 5.0);
}
glClearColor(0.5, 0.5, 0.5, 1.0); /* fog color */
}
```

3. `glFogi(GL_FOG_MODE, fogMode)`; specifies *fog mode*, in this case `GL_EXP`; the other possible *fog modes* are: `GL_EXP2` and `GL_LINEAR`.
4. Fog computes a fog weight f based on (a) the distance z from the viewing origin (eye) to the fragment in question, and (b) *fog mode*:

- `GL_LINEAR`.

$$f = \frac{end - z}{end - start}. \quad (7.3)$$

- `GL_EXP`.

$$f = e^{-density \cdot z}. \quad (7.4)$$

- `GL_EXP2`.

$$f = e^{-(density \cdot z)^2}. \quad (7.5)$$

f is always clamped to the range $[0, 1]$. In some contexts f is called the *extinction coefficient*. `fog.c` allows you to cycle between the *fog modes* by typing `f`.

5. *end* and *start* are specified using

```
glFogf (GL_FOG_START, 1.0);
glFogf (GL_FOG_END, 5.0);
```

6. Figure 7.8 shows graphs of example fog density equations.

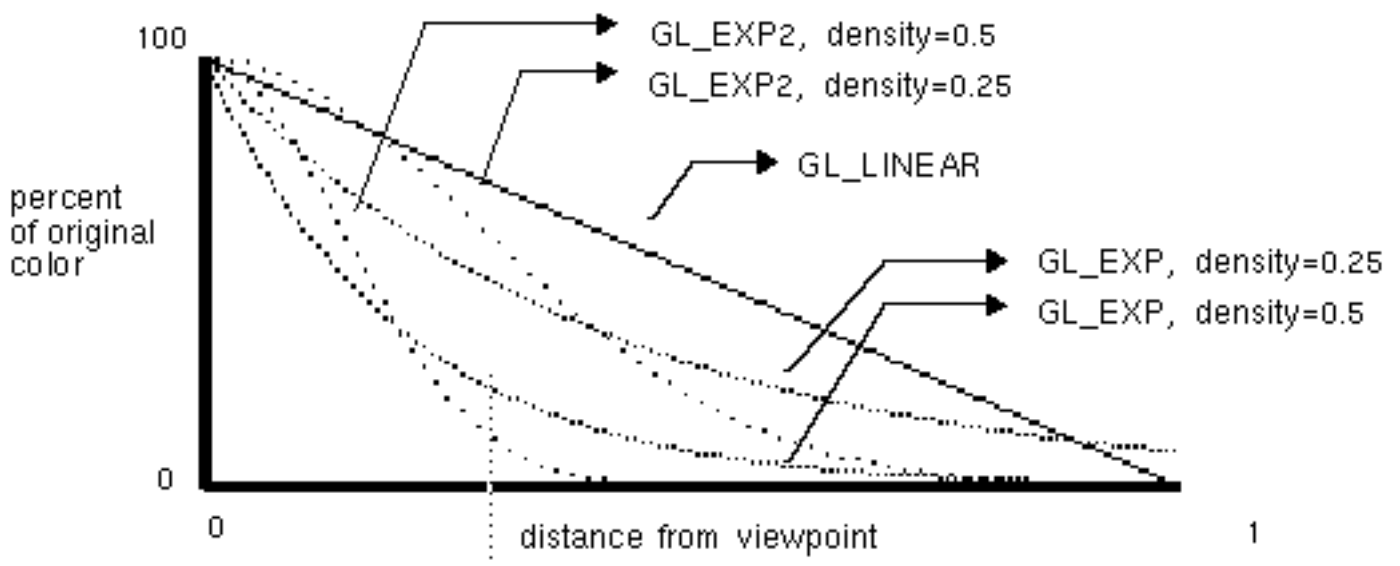


Figure 7.8: Fog density equations.

7. In RGBA mode, the fragment's colours, $C_i, i = r, g, b$, are computed by

$$C'_i = fC_i + (1 - f)C_f, \quad (7.6)$$

where C'_i is the new colour, C_i is the colour before fogging, and C_f is the *fog colour*. The fragment's alpha is not affected.

8. *fog colour* is specified using

```
GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};
glFogfv (GL_FOG_COLOR, fogColor);
```

9. See Figure 7.6 for the meaning of `glHint` (`GL_FOG_HINT`, `GL_DONT_CARE`);. Possible values are `GL_DONT_CARE` or `GL_NICEST` (fog computed per fragment / pixel) or per vertex `GL_FASTEST`.

Chapter 8

Vertex Arrays, Vertex Buffer Objects, and Display Lists

8.1 Introduction

A *Vertex array* provides a method of collecting vertex data (and colour data) together in a block (array) and then rendering from that block. This can lead to efficiency gains in two ways: (i) cut down the number of OpenGL API calls; (ii) by grouping the data together, transfer to the GPU may be done in one burst.

The next optimisation is to use a *vertex buffer object*; here we specify that the vertex array is to reside on GPU VRAM. You will be aware that, as GPUs get faster, the performance bottleneck become the rate at which data may be transferred to (and from, which is even slower) the GPU.

Display list is an earlier method of allowing display data and instructions to be transferred to the GPU just once. The drawback is that once a display list is transferred, it cannot be modified.

We discuss display lists first.

8.2 Display Lists

These brief notes are from (Shreiner et al. 2008a) Chapter 6. See also (Angel 2008) Chapter 3.12 for a nice brief description.

The books make the distinction between *immediate mode* and *retained mode* graphics. Essentially, the distinction is:

Immediate mode Instructions (*primitives*) are passed to the graphics *server* as soon as they are executed on the *client*; the primitives enter the graphics pipeline and eventually result in a rendered display. This cycle happens any time there is a redisplay (typically by calling the display callback). The server makes no attempt to store the primitives.

Retained mode Instructions (*primitives*) are collected together as a *display list* (the OpenGL term). A display list is somewhat like a subprogram. Some form of the display list (or multiple display lists) is/are passed to the server and stored there. Then a *call* executed on the client causes execution.

There are two notable advantages: (a) potential saving of costly transfers between client and server; (b) provision of a *sort of* object-oriented way of creating graphics objects.

The (partial) program `torus.c` shown in Figure 8.2 gives an example use of a display list; the output is shown in Figure 8.1.

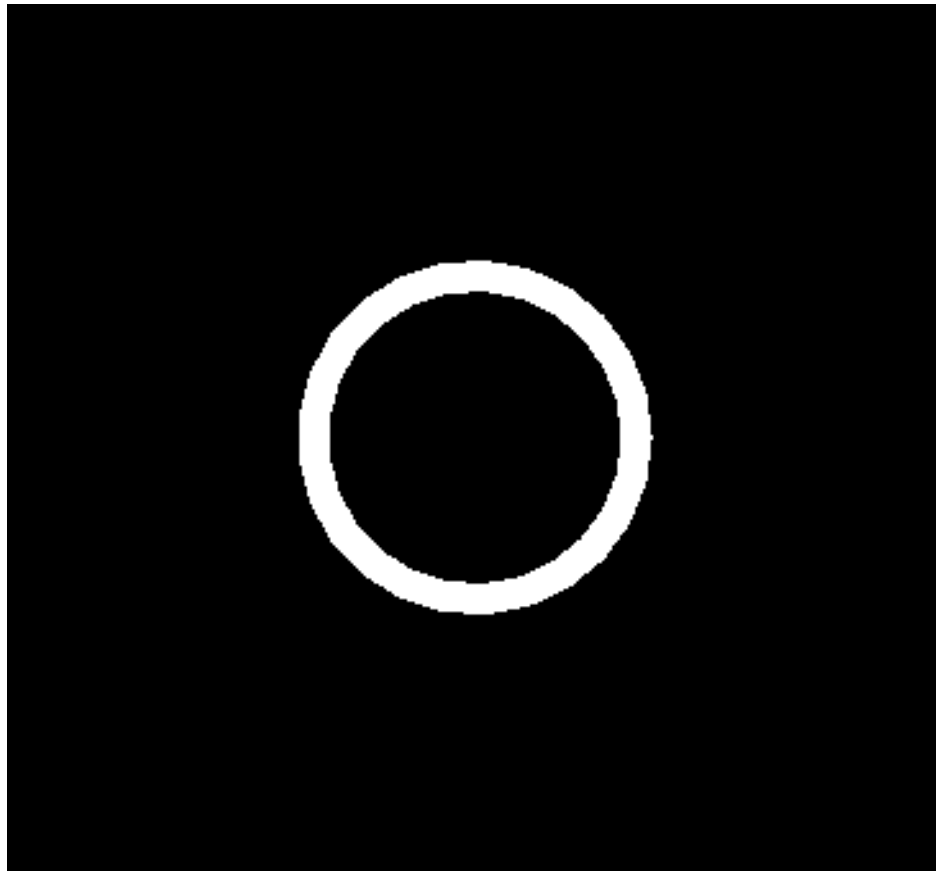


Figure 8.1: Torus.c output.

```

GLuint theTorus;
/* Draw a torus */
static void torus(int numc, int numt)
{
    int i, j, k;
    double s, t, x, y, z, twopi;

    twopi = 2 * PI_;
    for (i = 0; i < numc; i++) {
        glBegin(GL_QUAD_STRIP);
        for (j = 0; j <= numt; j++) {
            for (k = 1; k >= 0; k--) {
                s = (i + k) % numc + 0.5;
                t = j % numt;

                x = (1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
                y = (1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
                z = .1 * sin(s * twopi / numc);
                glVertex3f(x, y, z);
            }
        }
        glEnd();
    }
}

/* Create display list with Torus and initialize state */
static void init(void)
{
    theTorus = glGenLists (1);
    glNewList(theTorus, GL_COMPILE);
    torus(8, 25);
    glEndList();

    glShadeModel(GL_FLAT);
    glClearColor(0.0, 0.0, 0.0, 0.0);
}

/* Clear window and draw torus */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glCallList(theTorus);
    glFlush();
}

```

Figure 8.2: Torus using display list, torus.c.

Dissection of torus.c

1. Function `torus` generates a torus using a *quad strip*. There is nothing unusual there.
2. Now `init`. First we ask for one (`glGenLists(1)`) display list identifier — but that is just an integer. From now on we reference that display list using the (integer) identifier stored in `theTorus`. If we had another display list that displayed a robot (say), we would ask for another identifier as:

```
GLuint theRobot;  
theRobot = glGenLists(1);
```

3. Now we build the list.

```
glNewList(theTorus, GL_COMPILE);  
torus(8, 25);  
glEndList();
```

`glNewList(theTorus, GL_COMPILE);` tells OpenGL to start the list identified by `theTorus`. `GL_COMPILE` tells it to just build the list and when complete (`glEndList()`) send it to the server; do not execute yet. `GL_COMPILE_AND_EXECUTE` is the alternative.

Then `torus(8, 25);` *pseudo-executes*, but instead of sending the commands to the server and into the graphics pipeline, they are grabbed and passed into the display list.

`glEndList();` tells OpenGL that the display list is complete. Presumably the display list is then sent to the server.

4. Now `display`. The first two lines are as we have done before. `glCallList(theTorus);` sends an instruction to the server to execute the display list identified by `theTorus`.

```
glClear(GL_COLOR_BUFFER_BIT);  
glColor3f (1.0, 1.0, 1.0);  
glCallList(theTorus);
```

5. Just to show that there is no end display significance of using a display list, we note that replacing the `glCallList` call by plain execution of the OpenGL primitives (`torus(8, 25);`), see below, gives the same display.

```
//glCallList(theTorus);  
torus(8, 25);
```

8.3 Vertex Arrays

8.3.1 Introduction

Vertex arrays are best approached via an example that does *not* use them, and then modifying the example to use them. In the next section we show how to extend a similar example to use vertex buffer objects. The interesting bits of the program `varray0.cpp` are shown in Figure 8.4; the output is shown in Figure 8.3. We want to cut down the number of `glVertex*` and `glColor*` calls.



Figure 8.3: Varray0.cpp output.

```

/* ----- varray0.cpp -----
 * varray0.cpp, from varray.c, varray.cpp j.g.c. 2007-01-04
 * demonstrates what vertex arrays replace
 *----- */
#include <GL/glut.h> #include <iostream> #include <sstream> #include <cassert>

void init() {
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
    glutCreateMenu( mainMenuCB );
    glutAddMenuEntry( "Quit", QUIT_VALUE );
    glutAttachMenu( GLUT_RIGHT_BUTTON );
}

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    glBegin (GL_TRIANGLES);
        glColor3f(1.0, 0.2, 0.2);    glVertex2i(25, 25);
        glColor3f(0.2, 0.2, 1.0);    glVertex2i(100, 325);
        glColor3f(0.8, 1.0, 0.2);    glVertex2i(175, 25);
        glColor3f(0.75, 0.75, 0.75); glVertex2i(175, 325);
        glColor3f(0.35, 0.35, 0.35); glVertex2i(250, 25);
        glColor3f(0.5, 0.5, 0.5);    glVertex2i(325, 325);
    glEnd ();
    glFlush ();
}

```

Figure 8.4: Varray0.cpp

8.3.2 First Try — plain arrays

See Figure 8.5; no real advantage. Note the use of `glColor*v` and `glVertex*v`.

```
/* ----- varray1.cpp -----
 * varray1.cpp, from varray0.cpp, varray.cpp j.g.c. 2007-01-04
 * demonstrates what vertex arrays replace
 *----- */
// vertexes and colour placed in arrays
const GLint verts[6][2] = {{25, 25},
                           {100, 325},
                           {175, 25},
                           {175, 325},
                           {250, 25},
                           {325, 325}};
const GLfloat cols[6][3] = {{1.0, 0.2, 0.2},
                             {0.2, 0.2, 1.0},
                             {0.8, 1.0, 0.2},
                             {0.75, 0.75, 0.75},
                             {0.35, 0.35, 0.35},
                             {0.5, 0.5, 0.5}};

void init() {
    glClearColor (0.0, 0.0, 0.0, 0.0); glShadeModel (GL_SMOOTH);
    glutCreateMenu( mainMenuCB ); glutAddMenuEntry( "Quit", QUIT_VALUE );
    glutAttachMenu( GLUT_RIGHT_BUTTON );
}

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    glBegin (GL_TRIANGLES);
        glColor3fv(cols[0]);    glVertex2iv(verts[0]);
        glColor3fv(cols[1]);    glVertex2iv(verts[1]);
        glColor3fv(cols[2]);    glVertex2iv(verts[2]);
        glColor3fv(cols[3]);    glVertex2iv(verts[3]);
        glColor3fv(cols[4]);    glVertex2iv(verts[4]);
        glColor3fv(cols[5]);    glVertex2iv(verts[5]);
    glEnd ();
}
```

Figure 8.5: First try, varray1.cpp

8.3.3 Second Try — plain arrays and for loop

See Figure 8.6; again, no real advantage.

```
/* ----- varray2.cpp -----
 * from varray1.cpp, varray.cpp j.g.c. 2007-01-04
 * demonstrates what vertex arrays replace
 *----- */

const GLint verts[6][2] = {{25, 25},
                           {100, 325},
                           {175, 25},
                           {175, 325},
                           {250, 25},
                           {325, 325}};

const GLfloat cols[6][3] = {{1.0, 0.2, 0.2},
                             {0.2, 0.2, 1.0},
                             {0.8, 1.0, 0.2},
                             {0.75, 0.75, 0.75},
                             {0.35, 0.35, 0.35},
                             {0.5, 0.5, 0.5}};

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    glBegin (GL_TRIANGLES);
        for(int i = 0; i < 6; i++){
            glColor3fv(cols[i]);
            glVertex2iv(verts[i]);
        }
    glEnd ();
    glFlush ();
    assert(glGetError() == GL_NO_ERROR);
}
```

Figure 8.6: Second try, varray2.cpp

8.3.4 Now Vertex-arrays

See Figures 8.7, 8.8 and 8.9. You should read about vertex-arrays in the Red Book and on the web. We will discuss this program in more detail when we can execute the program in a practical laboratory.

```
/* ----- varray.cpp -----
 * varray.cpp, from varray.c j.g.c. 2007-01-04
 * This program demonstrates vertex arrays (OpenGL Red Book).
 *----- */
#define POINTER 1      #define INTERLEAVED 2
#define DRAWARRAY 1   #define ARRAYELEMENT 2   #define DRAWELEMENTS 3
int setupMethod = POINTER; int derefMethod = DRAWARRAY;

void setupPointers(void) {
    static GLint vertices[] = {25, 25,
                               100, 325,
                               175, 25,
                               175, 325,
                               250, 25,
                               325, 325};
    static GLfloat colors[] = {1.0, 0.2, 0.2,
                               0.2, 0.2, 1.0,
                               0.8, 1.0, 0.2,
                               0.75, 0.75, 0.75,
                               0.35, 0.35, 0.35,
                               0.5, 0.5, 0.5};
    glEnableClientState (GL_VERTEX_ARRAY);
    glEnableClientState (GL_COLOR_ARRAY);
    glVertexPointer (2, GL_INT, 0, vertices);
    glColorPointer (3, GL_FLOAT, 0, colors);
}

void setupInterleave(void){
    static GLfloat intertwined[] =
        {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
         1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
         1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
         0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
         0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
         0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
    glInterleavedArrays (GL_C3F_V3F, 0, intertwined); } ... continued ...
```

Figure 8.7: Vertex-arrays, varray.cpp, part 1

```

... varray.cpp ...
void init(void) {
    version();
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
    setupPointers ();
    glutCreateMenu( mainMenuCB );
    glutAddMenuEntry( "Quit", QUIT_VALUE );
    glutAttachMenu( GLUT_RIGHT_BUTTON );
    assert(glGetError() == GL_NO_ERROR);
}

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    cout<< "POINTER 1"<< endl;
    cout<< "INTERLEAVED 2"<< endl;
    cout<< "setupMethod = "<< setupMethod<< endl;

    cout<< "DRAWARRAY 1"<< endl;
    cout<< "ARRAYELEMENT 2"<< endl;
    cout<< "DRAWELEMENTS 3"<< endl;
    cout<< "derefMethod = "<< derefMethod<< endl;

    if (derefMethod == DRAWARRAY)
        glDrawArrays (GL_TRIANGLES, 0, 6);
        /*above same as
        glBegin(GL_TRIANGLES);
            for(int i = 0; i< 6; i++){
                glArrayElement(i);
            }
        glEnd();
        */
    else if (derefMethod == ARRAYELEMENT) {
        glBegin (GL_TRIANGLES);
        glArrayElement (2);
        glArrayElement (3);
        glArrayElement (5);
        glEnd ();
    }
    else if (derefMethod == DRAWELEMENTS) {
        GLuint indices[4] = {0, 1, 3, 4};
        glDrawElements (GL_POLYGON, 4, GL_UNSIGNED_INT, indices);
    }
    glFlush ();
    assert(glGetError() == GL_NO_ERROR);
}
... continued ...

```

Figure 8.8: Vertex-arrays, varray.cpp, part 2

```

... varray.cpp ...
void mouse (int button, int state, int x, int y){
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {
                if (setupMethod == POINTER) {
                    setupMethod = INTERLEAVED;
                    setupInterleave();
                }
                else if (setupMethod == INTERLEAVED) {
                    setupMethod = POINTER;
                    setupPointers();
                }
                glutPostRedisplay();
            }
            break;
        case GLUT_MIDDLE_BUTTON:
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN) {
                if (derefMethod == DRAWARRAY)
                    derefMethod = ARRAYELEMENT;
                else if (derefMethod == ARRAYELEMENT)
                    derefMethod = DRAWELEMENTS;
                else if (derefMethod == DRAWELEMENTS)
                    derefMethod = DRAWARRAY;
                glutPostRedisplay();
            }
            break;
        default:
            break;
    }
}

```

Figure 8.9: Vertex-arrays, varray.cpp, part 3

8.3.5 Vertex-array in a Buffer Object

Figure 8.10 shows a simple use of *buffer objects*. You should read about *buffer objects* in the Red Book and on the web. We will discuss this program in more detail when we can execute the program in a practical laboratory.

```

/* ----- varrayb.cpp -----
 * from varray.cpp j.g.c. 2007-01-04
 * This program demonstrates vertex-arrays in
 * buffer objects (OpenGL Red Book). No colour.
 *----- */

// had to do this to get to compile 2007-01-04; see
//http://www.gamedev.net/community/forums/topic.asp?topic_id=422358

#define GL_GLEXT_PROTOTYPES
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <GL/glut.h>
#include <GL/glext.h>

#include <iostream>#include <sstream> #include <cassert>
const int QUIT_VALUE( 99 );

/**
GLvoid* bufferObjectPtr(unsigned int i){
    return (GLvoid*)( ((char*)NULL) + i);
}

void setupPointers(void){
    GLuint buff; glGenBuffers(1, &buff);
    static GLint vertices[] = {25, 25,
                               100, 325,
                               175, 25,
                               175, 325,
                               250, 25,
                               325, 325};

    glBindBuffer(GL_ARRAY_BUFFER, buff);
    glBufferData(GL_ARRAY_BUFFER, 2*6*sizeof(GLint), vertices,
                 GL_STATIC_DRAW);
    glVertexPointer (2, GL_INT, 0, bufferObjectPtr(0));
    glEnableClientState(GL_VERTEX_ARRAY);
}

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    glDrawArrays (GL_TRIANGLES, 0, 6);
    /* above the same as
    glBegin(GL_TRIANGLES);
        for(int i = 0; i< 6; i++){ glArrayElement(i);}
    glEnd(); */
    glFlush ();}

```

Figure 8.10: Vertex-arrays in a buffer object, varrayb.cpp

Chapter 9

Images, Font etc. in OpenGL

Chapter 10

Texture Mapping

For a start, as noted by (Watt 2000), texture mapping would be far better called *colour mapping*. That is what it is, you take a *colour* pattern, for example read in from an image file, and you *map* that image to surfaces (polygons) within the scene. By *map*, we mean that we take the 2D image and considering it as a rubber sheet, we drape it over polygons, expanding and compressing as necessary to get the (rectangular) image to fit on the polygon.

As usual with OpenGL there's a pile of details, and a whole raft of different ways of doing things, but that's it in a nutshell.

The coverage in (Shreiner et al. 2008a) is not bad, but not up to their usual high standard; they recommend (Watt 2000) (Chapter 8) and that is good advice. (Angel 2008) has a rather good introduction, and our first example is a modified example (`cubetex.c`) from that. See also (Wright et al. 2007) which has a comprehensive coverage.

Why texture mapping? Well, it's a nice cheap and easy, and effective, way of adding realism to graphics. Think of all the polygons it would take to create a realistic image of grass, or wood, or sand. With texture mapping you make a plain surface and paint a picture of grass, or wood, or sand onto it. You could even map a picture of your face onto some shape like a sphere; or map a satellite picture or atlas picture of the earth onto a sphere (recall the `planets.c` program.)

10.1 Your first texture mapping program, `cubetex.c`

The program `cubetex.c` shown in Figures 10.2 and 10.3 displays the textured cubes shown in Figure 10.1.



Figure 10.1: (a) Checkerboard textured cube; (a) Jelly bean textured cube.

```

#define IW 64
#define IH 64
int nRows= IH, nCols= IW;
static GLubyte image[IH][IW][4];

void makeCheckImage(void){
    int r, c, dat;

    for (r = 0; r < IH; r++) {
        for (c = 0; c < IW; c++) {
            dat = (((r&0x8)==0)^((c&0x8)==0))*255;
            //printf("r,c, dat= %d, %d, %d\n", r,c, dat);
            image[r][c][0] = (GLubyte) dat;
            image[r][c][1] = (GLubyte) dat;
            image[r][c][2] = (GLubyte) dat;
            image[r][c][3] = (GLubyte) 255;
        }
    }
}

void init(void){
    glClearColor(1.0,1.0,1.0,1.0);
    //glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    //makeCheckImage();
    readImage();
    glEnable(GL_TEXTURE_2D);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, nRows, nCols,
                0, GL_RGBA, GL_UNSIGNED_BYTE, image);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
}

```

Figure 10.2: Textured cube, cubetex.c, part 1.

```

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

GLfloat colors[][4] = {{0.0,0.0,0.0,0.5},{1.0,0.0,0.0,0.5},
    {1.0,1.0,0.0,0.5}, {0.0,1.0,0.0,0.5}, {0.0,0.0,1.0,0.5},
    {1.0,0.0,1.0,0.5}, {1.0,1.0,1.0,0.5}, {0.0,1.0,1.0,0.5}};

GLfloat colors2[][4] = {{1.0,1.0,1.0,0.5},{1.0,1.0,1.0,0.5},
    {1.0,1.0,1.0,0.5}, {1.0,1.0,1.0,0.5}, {1.0,1.0,1.0,0.5},
    {1.0,1.0,1.0,0.5}, {1.0,1.0,1.0,0.5}, {1.0,1.0,1.0,0.5}};

void polygon(int i1, int i2, int i3, int i4){
    glBegin(GL_POLYGON);
        glColor4fv(colors2[i1]);
        glTexCoord2f(0.0,0.0);
        glVertex3fv(vertices[i1]);
        glColor4fv(colors2[i2]);
        glTexCoord2f(0.0,1.0);
        glVertex3fv(vertices[i2]);
        glColor4fv(colors2[i3]);
        glTexCoord2f(1.0,1.0);
        glVertex3fv(vertices[i3]);
        glColor4fv(colors2[i4]);
        glTexCoord2f(1.0,0.0);
        glVertex3fv(vertices[i4]);
    glEnd();
}

void colorcube(void){
    /* map vertices to faces */
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}

```

Figure 10.3: Textured cube, cubetex.c, part 2.

Dissection of cubetex.c

1. First of all we create an image, whose pixels are unsigned char. Though I disapprove of *smart* code that is hard to decipher, I have left the Red Book's smart way of generating a checkerboard (squares a 8×8). $(r \& 0x8) == 0$ gives 1 (boolean true) for 0 to 7, 16 to 23, ..., and likewise for c . Then $\wedge$ XORs the two conditions. So we end up with alternate black (0, 0, 0, $\alpha = 255$) and white (255, 255, 255, $\alpha = 255$) pixels.

```
#define IW 64
#define IH 64
int nRows= IH, nCols= IW;
static GLubyte image[IH][IW][4];

void makeCheckImage(void){
    int r, c, dat;

    for (r = 0; r < IH; r++) {
        for (c = 0; c < IW; c++) {
            dat = (((r&0x8)==0)^((c&0x8)==0))*255;
            //printf("r,c, dat= %d, %d, %d\n", r,c, dat);
            image[r][c][0] = (GLubyte) dat;
            image[r][c][1] = (GLubyte) dat;
            image[r][c][2] = (GLubyte) dat;
            image[r][c][3] = (GLubyte) 255;
        }
    }
}
```

2. When you examine cubetex.c, you will see that I have included the possibility of reading an image from a file. The file jbl.ppm is an image of some jelly beans. I have had to hack severely to be able to read a 256×256 image. In addition, the program can read only ASCII PPM format. If you look at (Wright & Lipchak 2004), you will find an alternative file reader. We will do more on this later, and I'll see to it that we have a way of handling most image file formats.
3. Next `init()`; `glClearColor` and enable depth test as before.

```
void init(void){
    glClearColor(1.0,1.0,1.0,1.0);
    glEnable(GL_DEPTH_TEST);
    makeCheckImage();
    //readImage();
    glEnable(GL_TEXTURE_2D);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, nRows, nCols,
                 0, GL_RGBA, GL_UNSIGNED_BYTE, image);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
}
```

4. Then call `makeCheckImage()`; to create the image data; note the alternative `readImage()`; commented out.
5. Now `glEnable(GL_TEXTURE_2D)`; enables texturing; and specifically *2D texturing*. We can have 1D (for lines) and 3D texturing (for volumes).
6. Next associate the image data (`image`) with the 2D texturing and tell it all about the data layout and sizes.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, nRows, nCols,
             0, GL_RGBA, GL_UNSIGNED_BYTE, image);
```

7. And then fill in some details about how to interpolate etc.
8. Now look at Figure 10.3 to see how we create the cube, and more importantly, how we tell OpenGL how to drape the texture (the image) onto the cube.
`colors2` is just an array of arrays (3) containing RGB values, in this case all white; if you want a coloured cube, substitute `colors`

```
void polygon(int i1, int i2, int i3, int i4){
    glBegin(GL_POLYGON);
        glColor4fv(colors2[i1]);
        glTexCoord2f(0.0,0.0);
        glVertex3fv(vertices[i1]);
        glColor4fv(colors2[i2]);
        glTexCoord2f(0.0,1.0);
        glVertex3fv(vertices[i2]);
        glColor4fv(colors2[i3]);
        glTexCoord2f(1.0,1.0);
        glVertex3fv(vertices[i3]);
        glColor4fv(colors2[i4]);
        glTexCoord2f(1.0,0.0);
        glVertex3fv(vertices[i4]);
    glEnd();
}
```

9. `polygon` creates one face of the cube.
`glVertex3fv(vertices[i1]);` etc. sets the vertices for that face.
10. `glTexCoord2f(0.0,0.0);` says *link texture coordinate (0.0,0.0) with the next vertex specified*.
 Texture coordinate (0.0,0.0) corresponds to image pixel (*row* = 0, *col* = 0). Texture coordinate (0.0,1.0) corresponds to image pixel (*row* = 0, *col* = *nCols*), etc.
 So we end up with that (polygon) face of the cube having the image stretched over it and with the corners of the image pinned to appropriate corners of the polygon.
11. `colorcube` then creates all the faces and does all the linking.
12. Please refer to (Angel 2008) pages 169–170 to see how this association is done.

Chapter 11

GLU Quadrics

These brief notes are from (Shreiner et al. 2008a) Chapter 6. See also (Angel 2008) Chapter 4.7.1 for a nice brief description.

At this point, it is worth noting that (Angel 2008) Chapter 4.7.2 gives a brief outline of the high-level objects provided by GLUT. Note: (1) All GLUT functions generate normals as a matter of course — this for lighting, see Chapter 6. (b) However, none of them *except the teapot* generate texture coordinates — for texture mapping, see Chapter 7.

For a general discussion on modelling of surfaces using *implicit* equations, *explicit* equations, and *parametric representations*, see any of: (Foley, van Dam, Feiner, Hughes & Phillips 1994) Chapter 9, (Foley et al. 1990) Chapter 11, (Watt 2000) Chapters 1–3, (Hearn & Baker 2003) Chapter 8, and (Wright et al. 2007) Chapter 10.

One way of modelling curves and surfaces is as solutions to an *implicit* equation of the general form

$$f(x, y, z) = 0. \quad (11.1)$$

The points of the surface are those points (x, y, z) where the equation is satisfied.

Examples in two-dimensions (x, y)

$$x - 1 = 0 \quad (11.2)$$

is a vertical plane (parallel to the y -axis) through $x = 1$.

$$x + y = 0 \quad (11.3)$$

is a 45° diagonal line, top left to bottom right, through the origin.

$$x + y - 1 = 0 \quad (11.4)$$

is a 45° diagonal line, top left to bottom right, that cuts the y -axis at $y = 1$ and the x -axis at $x = 1$.

$$x^2 - y = 0 \quad (11.5)$$

is a parabola (cup shape with open side facing up) sitting on the origin. Note the two x values for every one y value.

Quadric surfaces are defined by the general *quadratic* equation,

$$q(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fzx + 2gx + 2hy + 2jz + k = 0, \quad (11.6)$$

where we say *quadric* and *quadratic* because the highest power or combined power in a term is *squared*.

Clearly, one can obtain eqns. 11.2 to 11.5 by appropriate choice of a, b etc., with many of them zero. If $a = b = c = 1$, $k = -1$, and the remaining coefficients are zero, we have a unit radius sphere centred on the origin.

Quadrics are handy and efficient for a number of reasons (Foley et al. 1994) (Foley et al. 1990):

- it is easy to compute the surface normal at any point; simply:

$$\mathbf{n} = \left(\frac{\partial q}{\partial x}, \frac{\partial q}{\partial y}, \frac{\partial q}{\partial z} \right). \quad (11.7)$$

- to test whether a general point, (x', y', z') is on the surface is a simple matter of substituting (x', y', z') into eqn. 11.5;
- if we have x, y we easily compute z — especially for hidden surface / depth testing;
- computing intersection curves between two surfaces is easy.

Eqn. 11.6 can be expressed in matrix form as,

$$\mathbf{u}^T \cdot \mathbf{Q} \cdot \mathbf{u} = 0, \quad (11.8)$$

where $\mathbf{u} = \begin{bmatrix} x \\ y \\ z \\ w = 1 \end{bmatrix}$, and $\mathbf{Q} = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix}$.

If we do a 3D transformation using a 4×4 matrix, M , such as we have in (Campbell 2008b) Chapter 7, then $\mathbf{Q}$ in eqn. 11.8 transforms to

$$(\mathbf{M}^{-1})^T \cdot \mathbf{Q} \cdot \mathbf{M}^{-1}. \quad (11.9)$$

The (partial) program `quadric.c` shown in Figures 11.2 and 11.3 gives an example use of quadrics, see Figure 11.1.

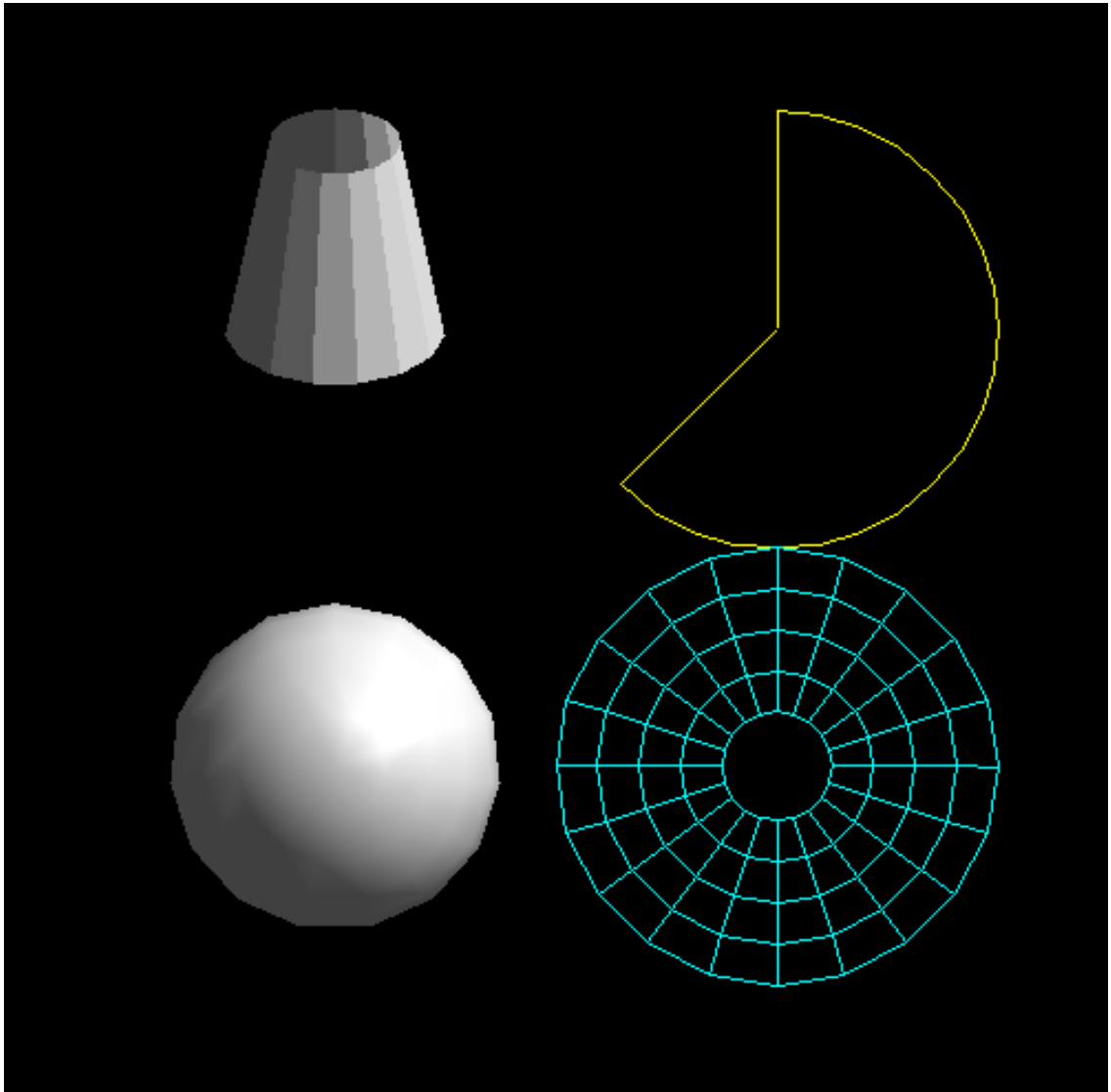


Figure 11.1: Quadric objects.

```

#ifndef CALLBACK
#define CALLBACK
#endif

GLuint startList;

void CALLBACK errorCallback(GLenum errorCode){
    const GLubyte *estring;
    estrings = gluErrorString(errorCode);
    fprintf(stderr, "Quadric Error: %s\n", estrings);
    exit(0);
}

void init(void) {
    GLUQuadricObj *qobj;
    GLfloat mat_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat model_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);

    glEnable(GL_LIGHTING); glEnable(GL_LIGHT0); glEnable(GL_DEPTH_TEST);

    /* Create 4 display lists, each with a different quadric object.
     * Different drawing styles and surface normal specifications
     * are demonstrated.
     */
    startList = glGenLists(4);
    qobj = gluNewQuadric();
    gluQuadricCallback(qobj, GLU_ERROR, errorCallback);

    gluQuadricDrawStyle(qobj, GLU_FILL); /* filled figure */
    gluQuadricNormals(qobj, GLU_SMOOTH); /* smooth shading */
    glNewList(startList, GL_COMPILE);
        gluSphere(qobj, 0.75, 15, 10);
    glEndList();

    gluQuadricDrawStyle(qobj, GLU_FILL);
    gluQuadricNormals(qobj, GLU_FLAT); /* flat shading, one normal per
                                         polygon */
    glNewList(startList+1, GL_COMPILE);
        gluCylinder(qobj, 0.5, 0.3, 1.0, 15, 5);
    glEndList(); // continued ...

```

Figure 11.2: Quadric objects, quadric.c, part 1

```

// ... continued
gluQuadricDrawStyle(qobj, GLU_LINE); /* all polygons wireframe */
gluQuadricNormals(qobj, GLU_NONE);
glNewList(startList+2, GL_COMPILE);
    gluDisk(qobj, 0.25, 1.0, 20, 4);
glEndList();

gluQuadricDrawStyle(qobj, GLU_SILHOUETTE); /* boundary only */
gluQuadricNormals(qobj, GLU_NONE);
glNewList(startList+3, GL_COMPILE);
    gluPartialDisk(qobj, 0.0, 1.0, 20, 4, 0.0, 225.0);
glEndList();
}

void display(void){
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();

    glEnable(GL_LIGHTING);
    glShadeModel (GL_SMOOTH);
    glTranslatef(-1.0, -1.0, 0.0);
    glCallList(startList);

    glShadeModel (GL_FLAT);
    glTranslatef(0.0, 2.0, 0.0);
    glPushMatrix();
    glRotatef(300.0, 1.0, 0.0, 0.0);
    glCallList(startList+1);
    glPopMatrix();

    glDisable(GL_LIGHTING);
    glColor3f(0.0, 1.0, 1.0);
    glTranslatef(2.0, -2.0, 0.0);
    glCallList(startList+2);

    glColor3f(1.0, 1.0, 0.0);
    glTranslatef(0.0, 2.0, 0.0);
    glCallList(startList+3);

    glPopMatrix();
    glFlush();
}

```

Figure 11.3: Quadric objects, quadric.c, part 2

Dissection of quadric.c

1. First we see the following callback function; it is registered later using `gluQuadricCallback(qobj, GLU_ERROR, errorCallback);`, where `qobj` is the pointer to the quadric object being created.

```
void CALLBACK errorCallback(GLenum errorCode)
//void errorCallback(GLenum errorCode)
{
    const GLubyte *estring;
    estring = gluErrorString(errorCode);
    fprintf(stderr, "Quadric Error: %s\n", estring);
    exit(0);
}
```

Up to now, we have been careless about error reporting. When you are writing your own programs, you can expect errors. Many OpenGL *run-time* (as opposed to *compile-time*) errors happen silently, i.e. nothing is displayed, or what you expect is not displayed. It is better always to request error codes and report them; for more details, see the Red Book (Shreiner et al. 2008a) Chapter 14.

2. `CALLBACK` is just something that Visual Studio may want. If I leave it out, GNU C is quite happy.
3. `GLUquadricObj *qobj`; You need such a pointer to reference quadric objects.
4. Next `init` creates lights and materials; see Chapter 6.
5. Now `init` requests four display list identifiers; see Chapter 8. If no display list identifiers had been requested before, it is likely that `startList` would be 1; and if you requested more identifiers later, you would get 5.

```
startList = glGenLists(4);
qobj = gluNewQuadric();
gluQuadricCallback(qobj, GLU_ERROR, errorCallback);
```

6. `qobj = gluNewQuadric();` a new empty quadric is created and associated with `qobj`.
7. And the error callback is registered — and associated with `qobj`; i.e. the callback is called to report any errors that happen during any quadric operation associated with `qobj`.
8. Now we create a smooth filled (`GLU_FILL`) sphere of radius 0.75 and with 15 longitude slices and 10 latitude slices.

```
gluQuadricDrawStyle(qobj, GLU_FILL); /* smooth shaded */
gluQuadricNormals(qobj, GLU_SMOOTH);
glNewList(startList, GL_COMPILE);
    gluSphere(qobj, 0.75, 15, 10);
glEndList();
```

9. `gluQuadricNormals(qobj, GLU_SMOOTH);` tells OpenGL to compute one normal per vertex; two other possibilities are `GLU_NONE`, no normals, i.e. anticipating no diffuse or specular lighting, and `GLU_FLAT`, one normal per polygon.
10. Next a cylinder. Note the use of `startList+1`; if the display list for the sphere had identifier N , the cylinder's display list would have identifier $N + 1$.
11. And then a disk; wireframe.
12. And finally a partial disk, wireframe, with the outer boundary only.
13. Now display. Display requests execution of the display lists in turn using `glCallList(startList);`, `glCallList(startList+1);` etc.
14. Note the use of lighting for the first two lists (sphere and cylinder) only.
15. And note the different effect of `glShadeModel(GL_SMOOTH);` (sphere) and `glShadeModel(GL_FLAT);` (cylinder).

Chapter 12

Interpolated Curves and Surfaces and OpenGL Evaluators

These brief notes cover the Bézier curve and surface programs given in (Shreiner et al. 2008a) Chapter 12. See also (Angel 2008) Chapter 9 for a nice brief description. Also, I'll be handing you out a few pages from a book called Numerical Recipes in C (very useful for mathematical functions). And some pages from (Vince 2001).

See also (Foley et al. 1994) Chapter 9, (Foley et al. 1990) Chapter 11, (Watt 2000) Chapters 1–3, (Hearn & Baker 2003) Chapter 8, and (Wright et al. 2007).

What we are covering here comes under the category of *parametric* representation of curves and surfaces. Typically what we have is

$$\mathbf{c}(u) = (x(u), y(u), z(u)), \quad (12.1)$$

for *curves* (curved lines) and,

$$\mathbf{s}(u, v) = (x(u, v), y(u, v), z(u, v)), \quad (12.2)$$

for *surfaces*, i.e. 2D surfaces.

In eqns. 12.1 and 12.2, u and v are the *parameters*. Typically an instance of one of these equations to represent a small part of an overall curve, or of an overall surface. In the case of Bézier curves and surfaces, the parametric functions $x(u)$ etc and $x(u, v)$ are polynomials; usually cubic polynomials are used. But quadratic Bézier curves are possible; and fourth power and above. The *parameters* (coefficients of the terms u, u^2 etc.) are derived from *control points*; control points are specified points which are used to guide the curve or surface generation — this will become clear later.

Usually, u varies in the range $[0, 1]$; however, if we require it to vary between $[u_0, u_1]$, we can scale and shift the parameter using $u' = (u - u_0)/(u_1 - u_0)$; u' varies in the range $[0, 1]$. **Ex.** Verify that last statement.

But first we look at plain interpolation.

12.1 Interpolation

If you were given a series of points of a curve y_i, x_i and you needed the y value for some intermediate point, you might use interpolation. In Figure 12.1, the large dark points show points that you *know*

and the lighter points are the results of interpolation. The known points are points from a sine curve.

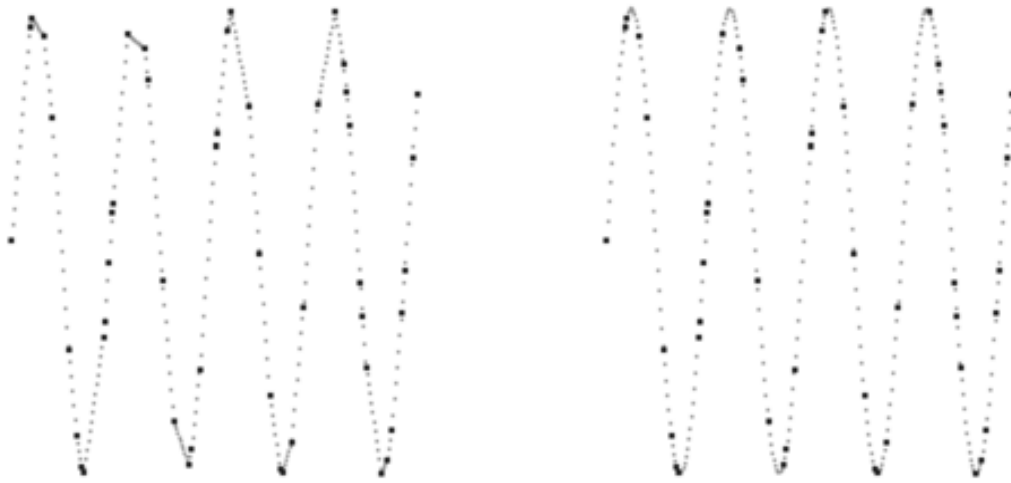


Figure 12.1: Interpolation. Linear (left) and cubic spline (right).

12.1.1 Linear Interpolation

The left hand side shows the results of *linear interpolation*. What this means is that you perform the mathematical equivalent of drawing straight lines between the known points (*control points*). We often do this when drawing curves — by hand or on a computer. If the control points are close together, or if the underlying curve is slowly varying (smooth), then the approximation is good and the interpolated curve looks satisfactory. However, if the points are far apart and linear segments poorly represent the underlying curve, the curve looks odd.

Note that in the earlier chapters we have been using a lot of linear interpolation or its equivalent; representing shapes by meshes of polygons etc.

Notice also that we can think of linear interpolation as a set of (linear) parametric curves, one for each pair of control points. More of this later when we discuss Bézier curves.

12.1.2 Spline Interpolation

Instead of linear interpolation, draughtspeople often used french curves or plastic flexible curves for drawing curves through a set of points. In the aircraft and ship industries, slivers of wood or metal called *splines* were often used. Hence the term *spline interpolation*.

In linear interpolation, the only constraint is that the parametric curves pass through the control points; i.e. continuity of the overall curve. But if we were using French curves or plastic flexible curves or drawing freehand, we would apply other constraints, namely that the slopes would also be continuous. See Numerical Recipes in C, pages 113–115.

The (partial) program `cubsplcurve.c` shown in Figures 12.2 and 12.3 gives an example use of linear and spline interpolation.

```

#define NPTS 50 float xx[NPTS], yy[NPTS], yya[NPTS] ;
int npts= NPTS; int ncycles= 5; int interp= 1;

//call this ONCE to create the parameters in y2
void spline(float x[], float y[], int n, float yp1, float ypn,
            float y2[]){

    int i,k; float p,qn,sig,un, u[NPTS];

    if (yp1 > 0.99e30) y2[1]=u[1]=0.0;
    else {
        y2[1] = -0.5;
        u[1]=(3.0/(x[2]-x[1]))*((y[2]-y[1])/(x[2]-x[1])-yp1);
    }
    for (i=2;i<=n-1;i++) {
        sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
        p=sig*y2[i-1]+2.0;
        y2[i]=(sig-1.0)/p;
        u[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
        u[i]=(6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
    }
    if (ypn > 0.99e30) qn=un=0.0;
    else {
        qn=0.5;
        un=(3.0/(x[n]-x[n-1]))*(ypn-(y[n]-y[n-1])/(x[n]-x[n-1]));
    }
    y2[n]=(un-qn*u[n-1])/(qn*y2[n-1]+1.0);
    for (k=n-1;k>=1;k--)
        y2[k]=y2[k]*y2[k+1]+u[k];
}

// call this for any value of the parameter x
float splint(float xa[], float ya[], float y2a[], int n, float x){
    int klo,khi,k; float h,b,a, y;

    klo=1;
    khi=n;
    while (khi-klo > 1) {
        k=(khi+klo) >> 1;
        if (xa[k] > x) khi=k;
        else klo=k;
    }
    h=xa[khi]-xa[klo];
    if (h == 0.0) nrerror("Bad xa input to routine splint");
    a=(xa[khi]-x)/h;
    b=(x-xa[klo])/h;
    y=a*ya[klo]+b*ya[khi]+
        ((a*a*a-a)*y2a[klo]+(b*b*b-b)*y2a[khi])*(h*h)/6.0;
    return y;
} // ... continued

```

Figure 12.2: Linear and spline interpolation, cubsplcurve.c, part1

```

float lint(float xa[], float ya[], int n, float x){
    int klo,khi,k;  float h,b,a, y;

    klo=1;  khi=n;
    while (khi-klo > 1) {
        k=(khi+klo) >> 1;
        if (xa[k] > x) khi=k;
        else klo=k;
    }
    h=xa[khi]-xa[klo];
    if (h == 0.0) nrerror("Bad xa input to routine lint");
    a=(xa[khi]-x)/h;
    b=(x-xa[klo])/h;
    y=a*ya[klo]+b*ya[khi];
    return y;}

void init(void){
    float fac, pi= 3.14159, x=0.0;  float yp1, ypn;  int i;
    glClearColor(0.0, 0.0, 0.0, 0.0);  glShadeModel(GL_FLAT);

    srand(131);  printf("RAND_MAX= %d\n", RAND_MAX);
    fac= (float)ncycles*(2.0*pi/(float)npts)*2.0/(float)RAND_MAX;

    for(i= 1; i< npts; i++){
        xx[i]= x;  yy[i]= sin(x);  x+= (float)rand()*fac;
    }
    yp1= 1.0; ypn= 1.0;
    spline(xx, yy, npts, yp1, ypn, yya);
}

void display(void){
    int i;  float x, y;
    glClear(GL_COLOR_BUFFER_BIT);  glColor3f(1.0, 1.0, 1.0);

    glPointSize(3.0);  glColor3f(1.0, 0.0, 1.0);
    glBegin(GL_POINTS);
        for (i = 0; i <= 300; i++){
            x= xx[npts-1]*(float) i/300.0;
            if(interp==1)y= splint(xx, yy, yya, npts, x);
            else y= lint(xx, yy, npts, x);
            glVertex2f(x, y);
        }
    glEnd();
    /* The following code displays the control points as dots. */
    glPointSize(5.0);  glColor3f(1.0, 1.0, 0.0);

    glBegin(GL_POINTS);
        for (i = 0; i < npts; i++) glVertex2f(xx[i], yy[i]);
    glEnd();}

```

Figure 12.3: Linear and spline interpolation, cubsplcurve.c, part2

12.2 Bézier Curves

Bézier curves were devised by an engineer Paul Bézier at Renault sometime in the early 1960s. Oddly enough, they had been earlier invented by Paul de Casteljau at Citroën in 1959, but de Casteljau's report was kept secret and surfaced only in 1975. Bézier curves use the notion of *blending functions*, namely Bernstein polynomials; these were described by S. Bernstein in 1912.

Bernstein polynomials can be defined as follows:

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}, \quad (12.3)$$

where the value of the *binomial* coefficient $\binom{n}{i}$ is given by

$$\binom{n}{i} = {}_nC_i = \frac{n!}{i!(n-i)!}. \quad (12.4)$$

Cubic Bézier curves are common. The four Bernstein polynomial blending functions, B_0^3 , B_1^3 , B_2^3 , and B_3^3 upon which Cubic Bézier curves are based are shown in Figure 12.4.

$$\begin{aligned} B_0^3(u) &= u^3, \\ B_1^3(u) &= 3u^2(1-u), \\ B_2^3(u) &= 3u(1-u)^2, \\ B_3^3(u) &= (1-u)^3. \end{aligned} \quad (12.5)$$

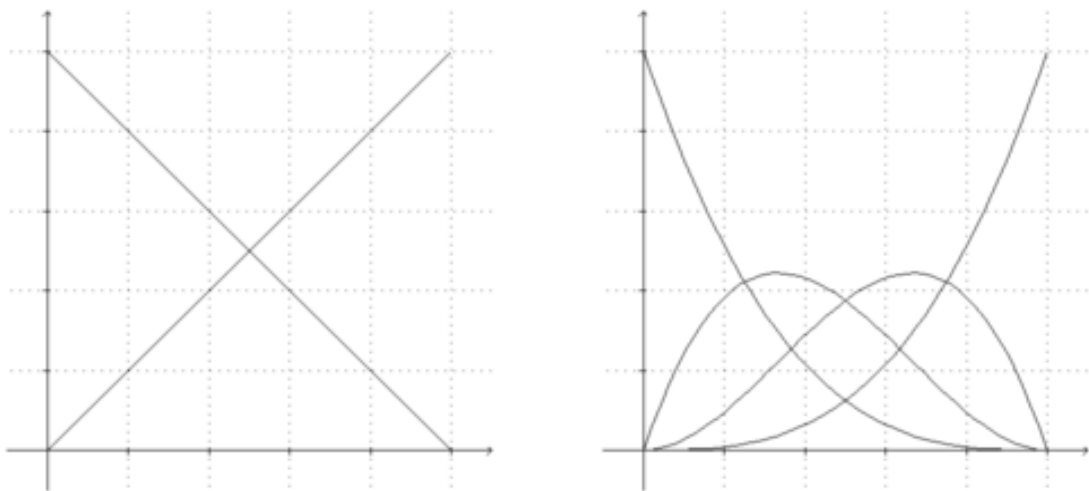


Figure 12.4: Blending functions. Linear (left) and cubic Bernstein (Bézier) (right).

Blending Functions To see how blending functions work, let us return to linear interpolation. One way of thinking about linear interpolation is that the interpolated value is as *blending* using blending functions a and b as in the code below. Here we are interpolating y for values of parameter x in $[xLo, xHi]$, where the y values at the end points are yLo and yHi .

```

a= (xHi - x)/(xHi -xLo);
b= (x- xLo)/(xHi - xLo);
y= a*yLo + b*yHi;

```

The blending functions (for parameter $u = 0 \cdots 1$) are shown in the right hand side of Figure 12.4; the blending function equivalent of a grows from 0 to 1, while the corresponding equivalent of b decreases from 1 to 0. What this means is that y_{Lo} dominates near $u = 0$ and y_{Hi} dominates near $u = 1$ and there is a *linear* transfer of dominance in between.

In fact, the linear blending functions turn out to be first-order Bernstein polynomials:

$$\begin{aligned} B_0^1(u) &= u, \\ B_1^1(u) &= (1 - u). \end{aligned} \tag{12.6}$$

So where do Bézier curves fit in? When describing a curve using a third order (cubic) Bézier curve, we specify four control points: the two end points, p_0, p_3 through which the Bézier curve will pass, and two other points, p_1, p_2 which control the shape of the curve. The following parametric equation is used:

$$p(u) = \sum_{i=0}^{n(=3)} B_i^3(u) p_i, \tag{12.7}$$

$$= B_3^3(u) p_0 + B_1^3(u) p_1 + B_2^3(u) p_2 + B_0^3(u) p_3 \tag{12.8}$$

$$= (1 - u)^3 p_0 + 3u(1 - u)^2 p_1 + 3u^2(1 - u) p_2 + u^3 p_3. \tag{12.9}$$

Eqn. 12.9 is slight abuse of notation, what we mean is that there are separate (independent) equations for the x , y , and z values.

It is easy to see that the curve passes through the end points p_0 , where $u = 0$, and p_3 , where $u = 1$. **Ex.** Show this.

12.3 Your first Bézier curve program

The (partial) program `bezcurve.c` shown in Figure 12.6 gives an example of Bézier curves in OpenGL. The result is shown in Figure 12.5.

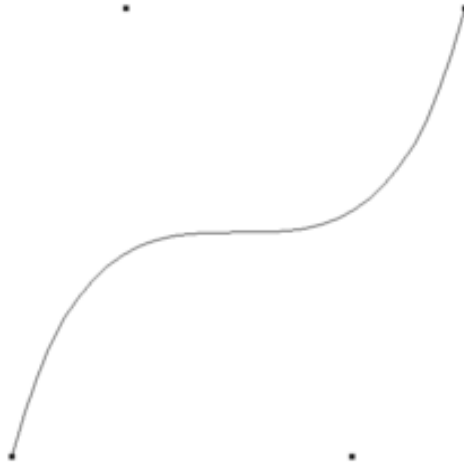


Figure 12.5: Cubic Bézier curve (four control points).

```

GLfloat ctrlpoints[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void init(void){
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
}

void display(void){
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord1f((GLfloat) i/30.0);
    glEnd();
    /* The following code displays the control points as dots. */
    glPointSize(5.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        for (i = 0; i < 4; i++)
            glVertex3fv(&ctrlpoints[i][0]);
    glEnd();
    glFlush();
}

```

Figure 12.6: Bézier curve, bezcurve.c

Dissection of `bezcurve.c`

1. There are four 3D control points.

```
GLfloat ctrlpoints[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    { 2.0, -4.0, 0.0}, { 4.0, 4.0, 0.0}};
```

2. We inject the control points into the Bézier curve evaluator system.

```
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
```

The arguments for this command are as follows, copied verbatim from (Shreiner et al. 2008a):

- `GL_MAP1_VERTEX_3`; three-dimensional vertices are used;
 - `0.0`. Low value of parameter u ;
 - `1.0`. High value of parameter u ;
 - `3`. The number of floating-point values to advance in the data between one control point and the next. Sometime called *stride*; note that 2D arrays are stored as a 1D array (1D and 2D in a different sense to the graphics sense);
 - `4`. The order of the spline, which is the degree+1; in this case, the degree is 3 (since the curve is a cubic).
 - `&ctrlpoints[0][0]`. Pointer to the first control point's data.
3. The second and third arguments control the parameterization of the curve — as the variable u ranges from 0 to 1, the curve goes from one end to the other. The call to `glEnable` enables the one-dimensional evaluator for two-dimensional vertices.
 4. The curve is drawn in `display` between the `glBegin` and `glEnd` calls. Since the evaluator is enabled (`glEnable(GL_MAP1_VERTEX_3)`), the command `glEvalCoord1f()` becomes like issuing a `glVertex` command with coordinates that are the coordinates of a vertex on the curve corresponding to the input parameter u .

12.4 Two Dimensional Evaluators

Whereas one dimensional Bézier curves can be described by

$$p(u) = \sum_{i=0}^n B_i^n(u) p_i, \quad (12.10)$$

see eqn. 12.9, two dimensional Bézier *surfaces* are described by the two dimensional analogue,

$$s(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(u) B_j^m(v) p_{ij}, \quad (12.11)$$

i.e. a multiplication of *curve*, where n is the order of the curves used in the u dimension, m is the order of the curves used in the v dimension, and p_{ij} are the control points. Typically $n = m = 3$ (cubic), and we need $4 \times 4 = 16$ control points.

The (partial) program `bezsurf.c` shown in Figure 12.8 gives an example of Bézier surfaces in OpenGL. The result is shown in Figure 12.7.

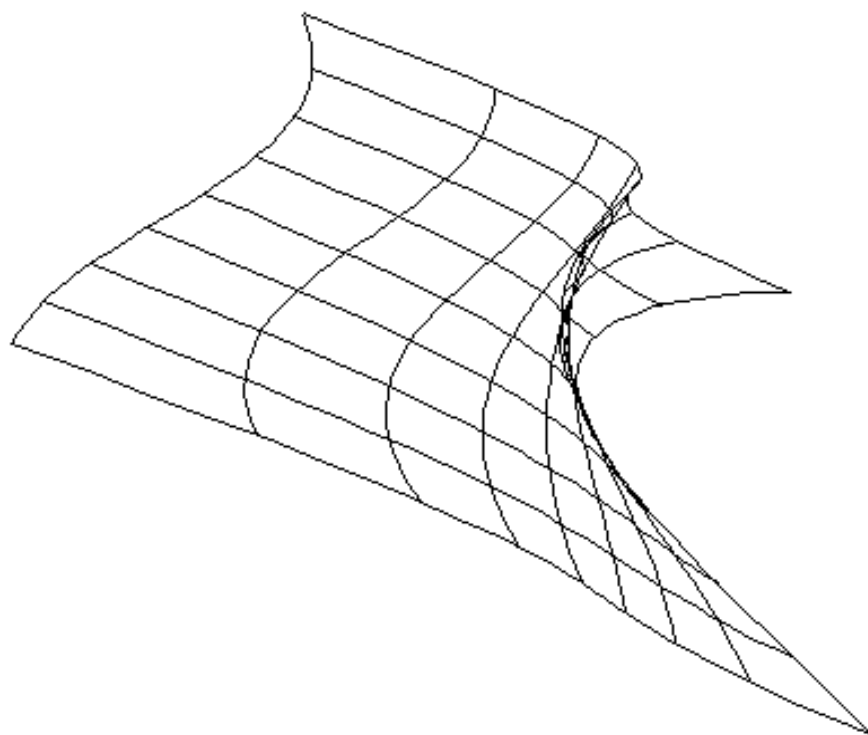


Figure 12.7: Cubic Bézier surface (16 control points).

```

GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void init(void){
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
            0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void display(void){
    int i, j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    for (j = 0; j <= 8; j++) {
        glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
        glEnd();
        glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
        glEnd();
    }
    glPopMatrix ();
    glFlush();
}

```

Figure 12.8: Bézier surface, bezsurf.c

Chapter 13

OpenGL Shading Language

13.1 Books and Sources

These notes are based on three main sources:

- *Randi J. Rost, OpenGL Shading Language, Addison Wesley, 2nd ed., 2006 (The Orange Book)* (<http://www.3dshaders.com/>) (Rost 2006);
- *Dave Shreiner et al, OpenGL Programming Guide (The Red Book), Addison Wesley, 6th ed. (or 5th will do), 2008, Chapter 15* (Shreiner et al. 2008b);
- Lighthouse3D's tutorials at: <http://www.lighthouse3d.com/opengl/glsl/> (Fernandes 2007).

See also:

- *R.S. Wright and B. Lipchak and N. Haemel, OpenGL Superbible, Addison-Wesley, 4th, 2007* (Wright et al. 2007);
- <http://nehe.gamedev.net/data/articles/article.asp?article=21> (Rudolf 2007);
- <http://www.clockworkcoders.com/oglsl/tutorials.html>.

13.2 OpenGL Pipeline

13.2.1 Introduction

As an introduction and overview and preview of the role of *shading languages*, we return to the *OpenGL pipeline* for a closer and more detailed look. This is also useful as a review of Chapters 3 to 8.

We have already seen Figure 13.1, but that shows only the vertex transformation part of the pipeline.

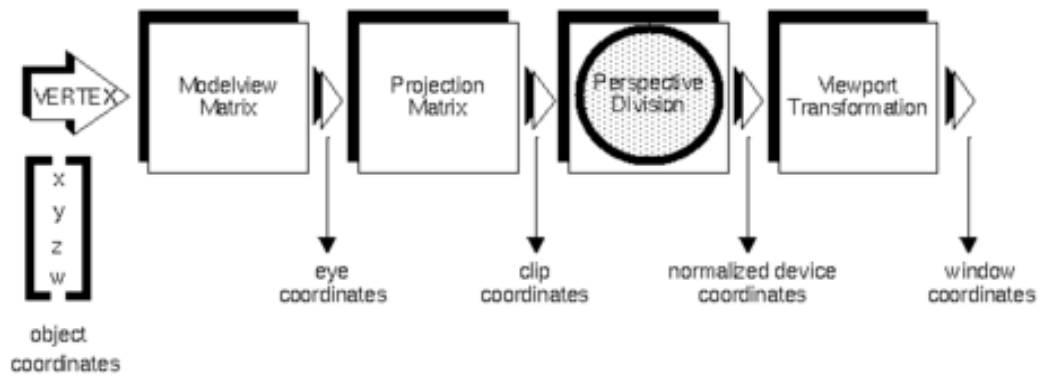


Figure 13.1: Rendering pipeline showing only vertex transformations (model, view, projection, perspective, viewport).

Figure 13.2 from the OpenGL Blue Book (Dave Shreiner (Ed.) and OpenGL ARB 2000) shows a much more detailed view of the pipeline.

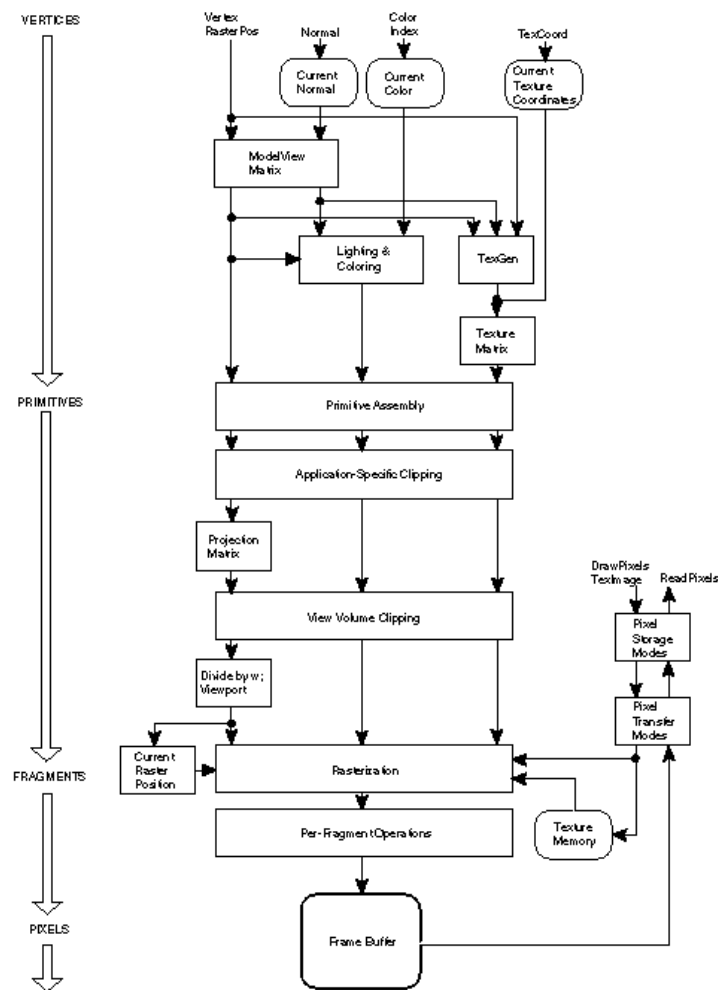


Figure 13.2: Rendering pipeline in detail (Dave Shreiner (Ed.) and OpenGL ARB 2000).

Figure 13.3 from the popular Nehe Productions tutorial (Rudolf 2007) shows a similar detail view; this may be more readable by some people.

Up to now, we have been dealing with a *fixed-function* pipeline; we can program whatever the GPU (Graphics Processing Unit) manufacturers together with the OpenGL API developers have decided they should let us program via calls to the drivers via the API. But, since 2000, GPUs have been getting more powerful (*) than CPUs and it was decided to allow GPUs to be user programmable. I can think of two main reasons: (i) to allow application programmers to implement fancy effects not possible in the fixed-pipeline; (ii) to be able to implement these effects at acceptable frame rates by taking advantage of the performance (speed and parallelism) of the GPU at doing graphics operations.

(*) *Anyone who has had to uprate their PC power supply to deal with a new graphics card can testify to the electrical current that they draw; and high current means one of two things, or both: fast clock speed; many processing units.*

The speed performance of GPUs is now so great that it is enticing some non-graphics application programmers (e.g. in image processing) to attempt to take advantage of it. But it's not something that you are likely to be able to program database applications in; plus, getting data to and from the GPU is always going to be a problem.

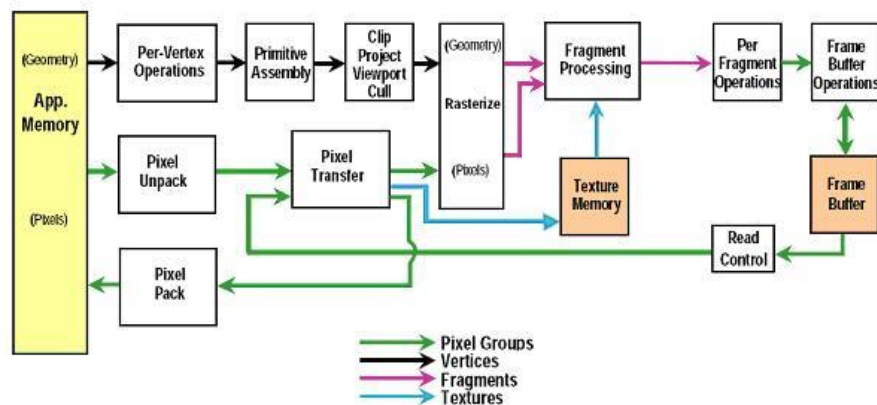


Figure 13.3: Rendering pipeline in more detail (Rudolf 2007).

13.2.2 Shaders and GLSL

Confusingly, user programs for GPUs are called *shaders*; but they are *programs* and their only connection with *shading* is that shading is an important aspect of computer graphics.

OpenGL Shading Language (GLSL — pronounced GLSL or *GLSlang*) essentially gives us another API — one allows us to program (alter) certain parts of the pipeline.

Apart from GLSL there are two other *high-level* shading languages: (i) Microsoft's High Level Shading Language (HLSL) which works with DirectX and WGL (Windows Graphics Foundation — which DirectX may be renamed to); (ii) nVidia's Cg. These are very similar in principle to GLSL and converting between them is easy.

The languages are called *high-level* in the sense that C and C++ are *high-level*, i.e. compared to *assembly* and machine code (*low-level*).

Before GLSL came along, there was a sort of (low-level) shader assembly language (ARB_vertex_shader, ARB_fragment_shader).

GLSL is based on C and C++.

Programmable Shading Pipeline GLSL allows programming of just two regions of the pipeline, namely the *vertex transformation* region and the *fragment processing* region; the corresponding programs are called *vertex shader* and *fragment shader*. From the application programmer's point of view, a vertex shader and a fragment shader are separate programs; but, obviously because of the *pipeline*, a vertex shader may communicate with a fragment shader via shared data variables, and the client application program can communicate with both — again via shared data variables.

In Figure 13.2 a vertex shader corresponds to the processing boxes (square cornered rectangles) *before* Primitive Assembly, i.e. Per-Vertex Operations in Figure 13.3.

In Figure 13.2 a fragment shader corresponds to the processing box Per-Fragment Operations, i.e. Fragment Processing in Figure 13.3.

In the next section, we'll review the complete pipeline in some detail and thereby give ourselves a clear idea of the (separate) responsibilities of vertex and fragment shaders.

Roughly speaking, a vertex shader is responsible for taking all the vertices specified in `glVertex*` calls and transforming them to *window* coordinates.

The term *fragment* corresponds roughly to *potential pixel plus some attributes*; in fact, the term *pixel shader* is often used.

Figure 13.4 gives a summarising view of the pipeline showing where the two shaders operate: vertex, *Vertex Transformation*; fragment: *Fragment Texturing and Coloring*. The corresponding diagram in Figure 13.5 gives a visual depiction of the pipeline, showing (top) the vertex transformation region and (bottom) the fragment processing region.

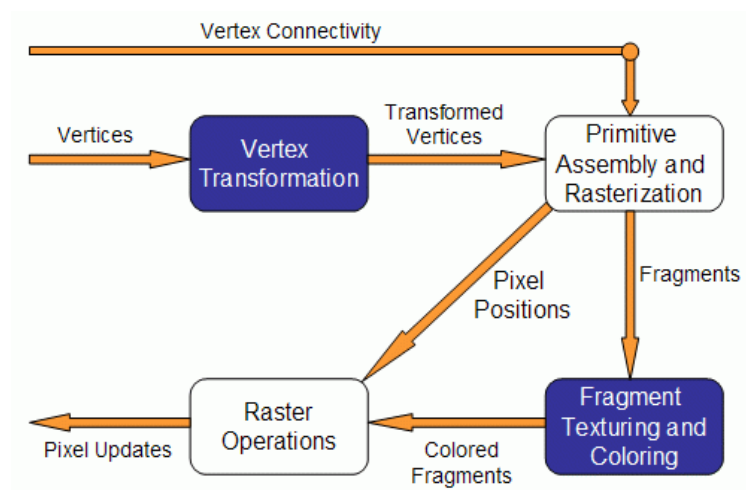


Figure 13.4: Simplified pipeline highlighting vertex and fragment handling parts of the pipeline (Fernandes 2007).

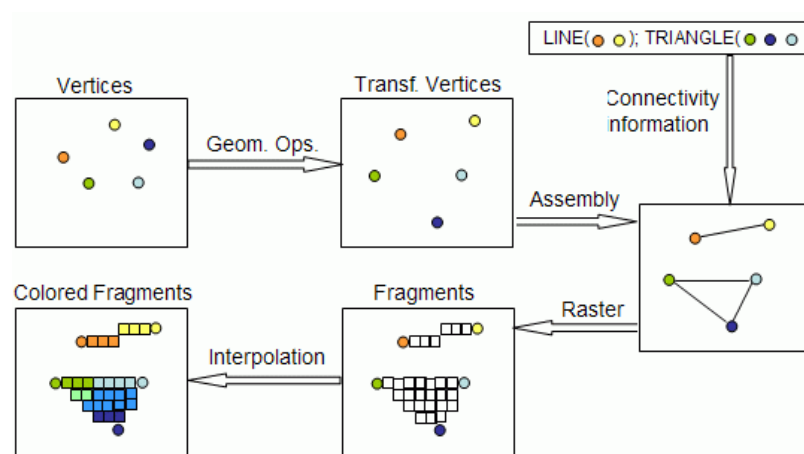


Figure 13.5: Visual depiction of what goes on in the pipeline (Fernandes 2007).

13.2.3 Pipeline, Detailed Review

We now review the OpenGL fixed-function pipeline in some detail; see (Shreiner et al. 2008b). In the diagrams below, rounded rectangles represent *data* and square cornered rectangles represent processes.

Figure 13.6 shows that OpenGL takes in *vertex data* (vertex coordinates, colours, normals, texture coordinates) and *pixel data* (textures).

Here, from Figure 3.1 is an example of vertex data. In examples involving lighting, lights and materials are also specified here.

```
glColor3f (1.0, 1.0, 1.0);
glBegin(GL_POLYGON);
    glVertex3f (0.25, 0.25, 0.0); glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0); glVertex3f (0.25, 0.75, 0.0);
glEnd();
```

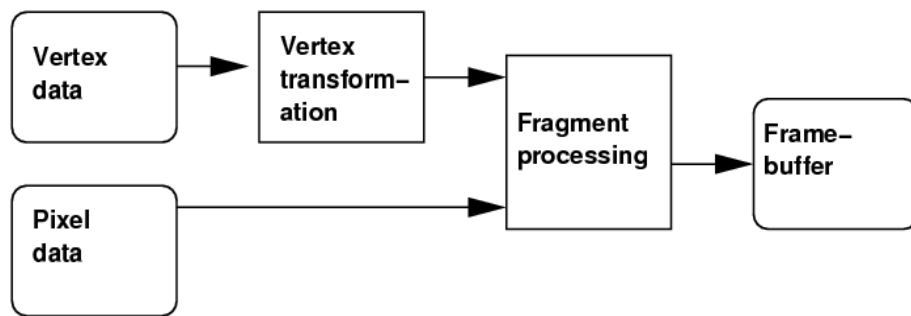


Figure 13.6: Another overview of the OpenGL pipeline.

Vertex Processing We now look in detail at vertex processing in Figure 13.7; a vertex shader can replace the transformation stages shown enclosed in the dotted rectangle.

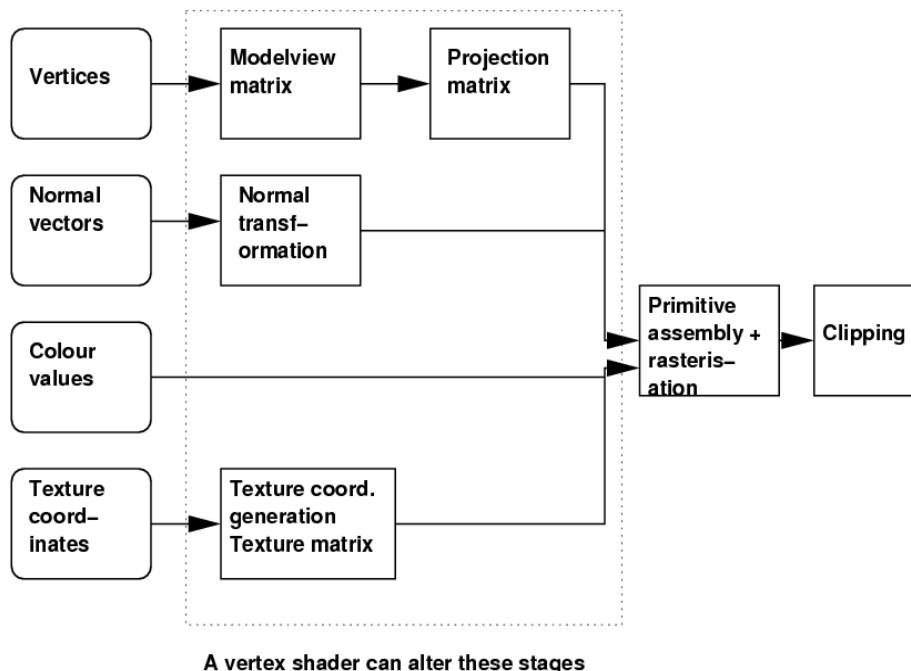


Figure 13.7: Vertex transformation pipeline.

Primitive assembly refers to the assembly of *primitives* (GL_LINES, GL_POLYGON etc.) from the vertices given.

Rasterisation refers to *drawing* the primitives onto a discrete pixel grid — recall *raster graphics* versus *vector graphics*.

Clipping refers to what is done when any of the vertices are outside the *view volume* (recall `glOrtho`, `glFrustum`, `glPerspective`). Clipping must be performed on primitives, not on individual vertices. In Figure 13.8 we start off with a triangle (`glbegin{GL_TRIANGLES} ... glEnd()`) whose vertices are v_1, v_2, v_3 . The clipping algorithm must consider each line (pair of vertices) and provide substitute vertices as appropriate. Clipping is a huge subject; see any general computer graphics book, e.g. (Foley et al. 1990).

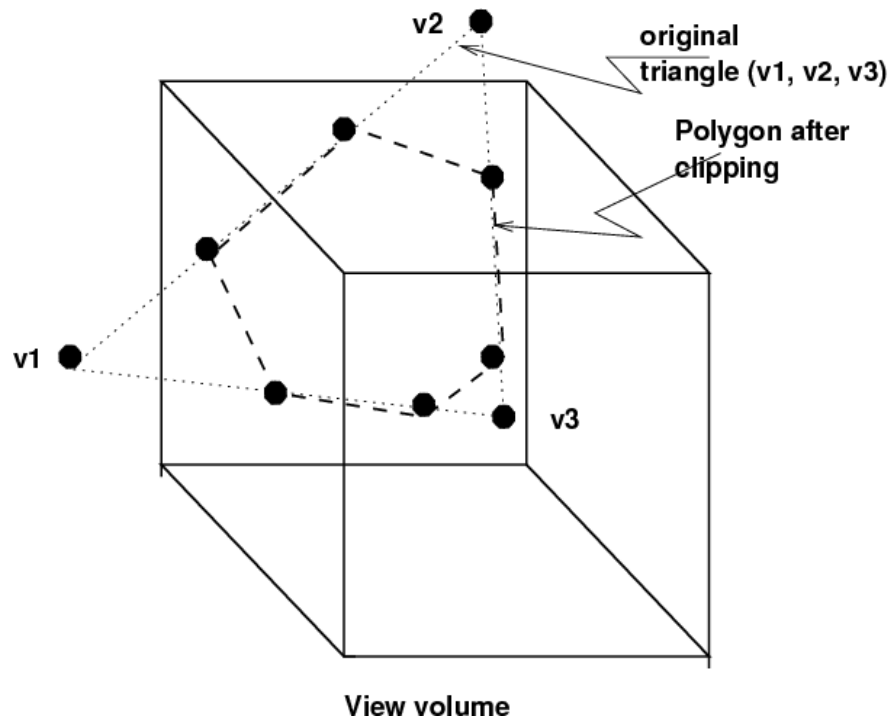


Figure 13.8: Clipping.

Figure 13.9 shows the vertex transformations in some detail. You should review this in conjunction with Chapter 5 of these notes and Chapters 5, 6 and 7 of the maths. notes (Campbell 2008a).

After primitive assembly, rasterisation, and clipping, have been performed, the resulting fragments (potential pixels with colour, depth) are passed on to the fragment processing stage.

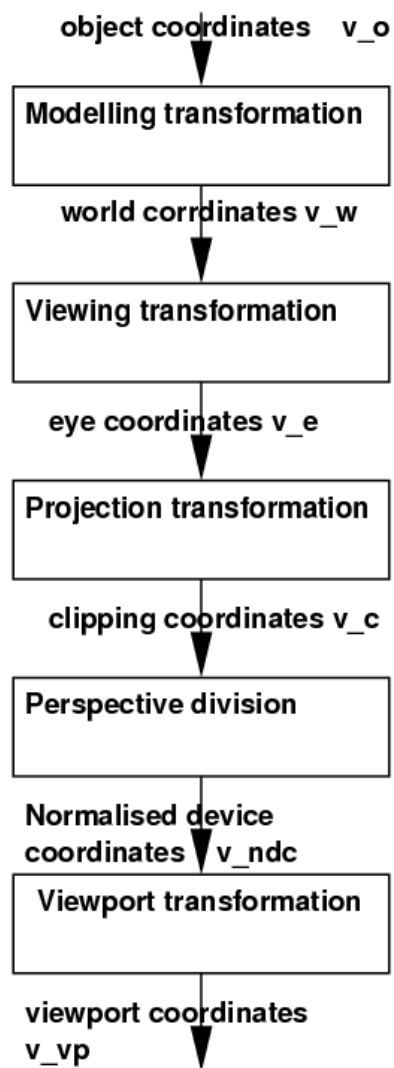


Figure 13.9: Vertex transformations in detail.

Fragment Processing Figure 13.7 shows the fragment processing part of the pipeline. A fragment shader can replace the processing stages shown enclosed in the dotted rectangle. Note. See *Primary and Secondary Colour Summation*; we have not covered *secondary colours*; this refers to some detail involved in mixing texture mapping and specular lighting, see (Shreiner et al. 2008b).

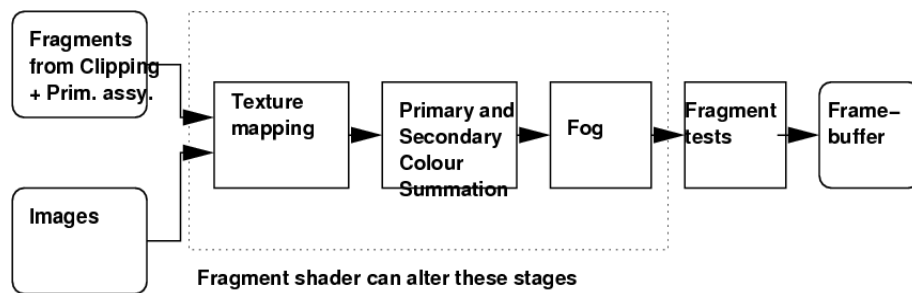


Figure 13.10: Fragment processing pipeline.

13.3 GLSL Shaders — Simple Examples

Now, we're going to dive straight in and give *Hello, World* style simple examples of vertex and fragment shader types. We'll give the shaders first and the OpenGL host program after, but you might want to have a quick look at Figures 13.19 to 13.22 just to see how the shaders are included.

13.3.1 Minimal Shaders

Minimal Vertex Shader Figure 13.11 shows a minimal vertex shader. The comments explain some of the code, and we'll talk through other aspects in the lecture. `gl_Position` (the vertex transformed by the modelview matrix *and* the projection matrix) is passed to the next stage of the pipeline; this is exactly the same as the fixed-function pipeline does.

```
// minimal vertex shader, minimal.vert, www.lighthouse3d.com

void main(){
    // the following three lines provide the same result

    // gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
    // gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = ftransform();
}
```

Figure 13.11: Minimal vertex shader, minimal.vert

Minimal Fragment Shader Figure 13.12 shows a minimal fragment shader; all objects are coloured a bluish colour.

```
// minimal.frag
// minimal fragment shader
// www.lighthouse3d.com

void main(){
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Figure 13.12: Minimal Fragment Shader, minimal.frag

13.3.2 Slightly more ambitious

Vertex Shader Figure 13.13 shows the vertex shader. The comments explain as well as we need at this stage. `gl_Position` and `gl_FrontColor` are altered; these are read by the fragment shader.

```
// color.vert
void main()
{
    //gl_Color = vec4(0.0, 1.0, 1.0, 1.0); // vertex attrib.
    gl_FrontColor = vec4(0.0, 1.0, 1.0, 1.0); // output to frag. shader

    // alter the vertex's x before transformation
    gl_Vertex.x *=3.0;
    // gl_Position is the output, so the line below has no effect
    // as gl_Position overwritten by gl_Position = ftransform();
    //gl_Position.x *= 2.0;

    gl_Position = ftransform();

    // alter the vertex's x after transformation
    //gl_Position.x *= 2.0;
}
```

Figure 13.13: Vertex shader, color.vert

Fragment Shader Figure 13.14 shows a minimal fragment shader; the entire object will be coloured *red*.

```
// color.frag
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Figure 13.14: Fragment Shader, color.frag

13.3.3 Diffuse lighting plus toon shading

Vertex Shader Figure 13.15 shows the vertex shader. First it extracts the light direction and normalises it. Next it transforms the normals; note: normals are transformed using a different matrix than the modelview-projection matrix product. You should be aware, see section 6.9, that if you want to use diffuse or specular lighting, then each vertex must be equipped with a normal, set using `glNormal*`. For this reason, these shaders do not work with the simple cube example, in `glsl1.cpp` make sure to replace the cube with the teapot, as follows; the teapot computes normals.

```
//drawCube();
glutSolidTeapot(1.0);

// toonf2.vert
// varying means global and accessible by the
// fragment shader
varying vec3 lightDir,normal;

void main(){
    lightDir = normalize(vec3(gl_LightSource[0].position));
    normal = gl_NormalMatrix * gl_Normal;

    gl_Position = ftransform();
}
```

Figure 13.15: Vertex shader, toonf2.vert

Fragment Shader Figure 13.16 shows a minimal fragment shader. First it normalises the normal; next it computes the lightness (brightness) of the surface based on the diffuse (Lambertian) lighting equation, see 6.9; then it chooses one of four colours based on what part of the intensity range (0 – 1) the lightness is, see Figure 13.17.

As an alternative, you can uncomment line `// a` to produce monochrome colouring; and if you uncomment line `// b` as well, you will get negative monochrome; both these are shown in Figure 13.18.

```

// toonf2.frag
varying vec3 lightDir,normal;

void main(){
    float intensity;
    vec4 color;

    // normalizing the lights position to be on the safe side
    vec3 n = normalize(normal);
    intensity = dot(lightDir,n);

    if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5) color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25) color = vec4(0.4,0.2,0.2,1.0);
    else color = vec4(0.2,0.1,0.1,1.0);

    /*
    if(intensity > 0.5) color = vec4(1.0,1.0,1.0,1.0);
    else color = vec4(0.0,0.0,0.0,1.0);
    */

    //color = vec4(intensity,intensity,intensity,1.0); // a
    //color = 1.0 -color; // b

    gl_FragColor = color;
}

```

Figure 13.16: Lighting + toon shading Fragment Shader, toonf2.frag



Figure 13.17: Diffuse intensity, toon shading; toonf2.vert, toonf2.frag.



Figure 13.18: (a) Monochrome diffuse intensity; (b) Negative of diffuse intensity.

13.3.4 OpenGL Program that Uses Shaders

Figures 13.19, 13.20, 13.21 and 13.22 shows a minimal example of an OpenGL program that uses shaders (those given in Figures 13.13 and 13.14).

```

/* glsl.cpp
   Simple Demo programs for GLSL
   from www.lighthouse3d.com with additions j.g.c. 2008-11-20
   Windows version j.g.c. 2008-11-22
   Make sure to include in your source files:
       glsl.cpp    GLee.c    GLutils.cpp
*/

#include <cstdio> #include <cstdlib> #include <cctype> // for tolower
#include "GLee.h"
#include <GL/glut.h>
#include "GLutils.h"

GLint loc; GLuint v, f, f2, p; float a = 0.0;

void changeSize(int w, int h) {
    if(h == 0) h = 1;
    float aspectRatio = 1.0* w / h;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity(); glViewport(0, 0, w, h);
    gluPerspective(45, aspectRatio, 1, 100);
    glMatrixMode(GL_MODELVIEW);
}

void drawCube() {
    float hd = 1.0; glColor3f(1,0,0);
    glBegin(GL_QUADS);
        glVertex3f(-hd,-hd,-hd); glVertex3f(-hd,hd,-hd);
        glVertex3f(hd,hd,-hd);    glVertex3f(hd,-hd,-hd);
    glEnd();
    // ... five other sides excluded
}

void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 5.0, 5.0,      // eye
              0.0, 0.0, 0.0,      // at
              0.0f, 1.0f, 0.0f); // up
    // light used only by toonf2 shaders
    GLfloat lpos[4] = {1.0,0.0,1.0,0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, lpos);
    glRotatef(a, 0., 1., 0.);
    drawCube();
    //glColor3f(0.,1.,1.); //glutSolidTeapot(1.0);
    a+= 0.01; // rotate
    glutSwapBuffers();
}

```

Figure 13.19: OpenGL Program using Shaders, part 1

```

void processNormalKeys(unsigned char key, int x, int y) {
    if (key == 27) exit(0);
    else if(tolower(key) == 'p') glUseProgram(p); // use shader program
    else if(tolower(key) == 'n') glUseProgram(0); // revert to fixed-function
}

void setShaders() {
    char *vs = NULL,*fs = NULL;

    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);

    const char* vertexShaderFilename =
"C:\\lect\\graphics2\\progs\\ch13shaders\\minimal.vert";
    const char* fragmentShaderFilename =
"C:\\lect\\graphics2\\progs\\ch13shaders\\minimal.frag";

    /*
const char* vertexShaderFilename = "color.vert";
const char* fragmentShaderFilename = "color.frag";
*/

    /*
const char* vertexShaderFilename = "toonf2.vert";
const char* fragmentShaderFilename = "toonf2.frag";
*/

    printf("%s, %s \n", vertexShaderFilename, fragmentShaderFilename);
    vs = textFileRead(vertexShaderFilename);
    fs = textFileRead(fragmentShaderFilename);

    const char *vv = vs;
    const char *ff = fs;
    printf("\nVertex shader ... \n %s \n", vv);
    printf("Fragment shader ... \n %s \n", ff);

    glShaderSource(v, 1, &vv, NULL);
    glShaderSource(f, 1, &ff, NULL);

    free(vs); free(fs);
    // ... continued
}

```

Figure 13.20: OpenGL Program using Shaders, part 2

```

// ... setShaders continued
glCompileShader(v);
GLint result;
glGetShaderiv(v, GL_COMPILE_STATUS, &result);
if(!result){
    printf("unsuccessful 1\n");
    reportShaderErrors(v);
    exit(EXIT_FAILURE);
}
glCompileShader(f);
glGetShaderiv(f, GL_COMPILE_STATUS, &result);
if(!result){
    printf("unsuccessful 2\n");
    reportShaderErrors(f);
    exit(EXIT_FAILURE);
}

p = glCreateProgram();
glAttachShader(p, f);
glAttachShader(p, v);

glLinkProgram(p);
glGetShaderiv(p, GL_LINK_STATUS, &result);
if(!result){
    printf("unsuccessful link\n");
    reportShaderErrors(v);
    exit(EXIT_FAILURE);
}

glUseProgram(p);
//loc = glGetUniformLocation(p, "time");
}

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100); glutInitWindowSize(320, 320);
    glutCreateWindow("Lighthouse 3D GLSL examples");
    version();

    glutDisplayFunc(renderScene); glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize); glutKeyboardFunc(processNormalKeys);

    glEnable(GL_DEPTH_TEST); glClearColor(1.0, 1.0, 1.0, 1.0);

    setShaders();
    glutMainLoop();
    return EXIT_SUCCESS;
}

```

Figure 13.21: OpenGL Program using Shaders, part 3

```

// in GLutils.cpp
char *textFileRead(const char *fn) {
    FILE *fp;
    char *content = NULL;
    int count= 0;

    if (fn == NULL){
        fprintf(stderr, "null file name\n");
        return content;
    }

    fp = fopen(fn,"rt");
    if (fp != NULL) {
        fseek(fp, 0, SEEK_END);
        count = ftell(fp);
        rewind(fp);
    }
    else{
        fprintf(stderr, "cannot open file %s\n", fn);
        return content;
    }
    if (count > 0) {
        content = (char *)malloc(sizeof(char) * (count+1));
        count = fread(content,sizeof(char),count,fp);
        content[count] = '\0';
    }
    fclose(fp);
    return content;
}

```

Figure 13.22: Program using Shaders, part 4, read shader source

Dissection

1. The OpenGL program creates a cube (or a teapot) and rotates it; note `glutIdleFunc(renderScene)`; that causes the scene to be drawn continually — `renderScene` increments the angle `a`.
2. The main action occurs in `setShaders`, see Figure 13.20.
3. `glCreateShader` allocates a shader object — handle for shader source.

```
v = glCreateShader(GL_VERTEX_SHADER);
f = glCreateShader(GL_FRAGMENT_SHADER);
```

- 4.
5. Next, the shader source code is read in by the `textFileRead` in Figure 13.22; that function finds out how long the array of characters (string) containing the shader program is (`count`; it allocates heap memory (+1 for the null terminator) and reads the string.

```
vs = textFileRead("color.vert");
fs = textFileRead("color.frag");
```

6. `glShaderSource` associates the shader source just read in with the shader object created above by `glCreateShader`.

We can had an array of strings, but in our case we have just one string, and `NULL` indicates that we do not give `length(s)` of the strings but indicate that it is `NULL \0`terminated.

```
//          shader-obj  count  string(s)  string length(s)
glShaderSource(v,      1,      &vv,      NULL);
glShaderSource(f, 1, &ff, NULL);
```

7. `glShaderSource` wants `const char *` so we do

```
const char *vv = vs;
const char *ff = fs;
```

8. Just to emphasise that the shaders are merely strings (with `\ns`) at the end of each line, we can print them:

```
printf("\nVertex shader ... \n %s \n", vv);
printf("Fragment shader ... \n %s \n", ff);
```

You'll see that when you run the program.

9. The shaders sources are copied to inside OpenGL, so we can free the memory allocated for them in this application program.

```
free(vs); free(fs);
```

10. Next we *compile* the two shaders (OpenGL does this — it is *not* C or C++ compilation and it is done a run-time, not compile time).

```
glCompileShader(v);  
// and later on ...
```

```
glCompileShader(f);
```

11. Check for errors and if necessary report them and exit.

```
glCompileShader(v);  
GLint result;  
glGetShaderiv(v, GL_COMPILE_STATUS, &result);  
if(!result){  
    printf("unsuccessful 1\n");  
    reportShaderErrors(v);  
    exit(EXIT_FAILURE);  
}
```

12. **Note carefully.** If the shader compiler doesn't like something, there will be a silent error and OpenGL will revert to the fixed-function pipeline; thus, the check above will save a lot of head scratching.

13. Then create a *program* object, and associate the compiled shaders with the program just created — this is an initial build — like building / linking of C++ programs. The commented line shows that we can have more than one of each shader.

```
p = glCreateProgram();  
glAttachShader(p, f);  
//glAttachShader(p, f2);  
glAttachShader(p, v);
```

14. Now we do a final *linking* to produce a final shader program; and don't forget to check for errors.

```
glLinkProgram(p);  
glGetShaderiv(p, GL_LINK_STATUS, &result);  
if(!result){  
    printf("unsuccessful link\n");  
    reportShaderErrors(v);  
    exit(EXIT_FAILURE);  
}
```

15. And tell OpenGL to use that program in place of the fixed-function pipeline.

```
glUseProgram(p);
```

Note. The complete fixed-function parts (vertex processing and fragment processing) will be replaced, so if you put something non-sensible in the shaders, you'll get a blank screen or non-sensible display.

16. We can revert (reset) back to the fixed-function pipeline using

```
glUseProgram(0);
```

and you can see that we do this in

```
void processNormalKeys(unsigned char key, int x, int y) {  
    if (key == 27) exit(0);  
    else if(tolower(key) == 'p') glUseProgram(p); // use shader program  
    else if(tolower(key) == 'n') glUseProgram(0); // revert to fixed-function  
}
```

17. All this is done in `setShaders` is done at run-time, so that if edit `color.vert` or `color.frag` but do not recompile (or rebuild) the main program, the effect of the edits will be seen.
18. A flowchart depiction of what we have just discussed is shown in Figure 13.23.

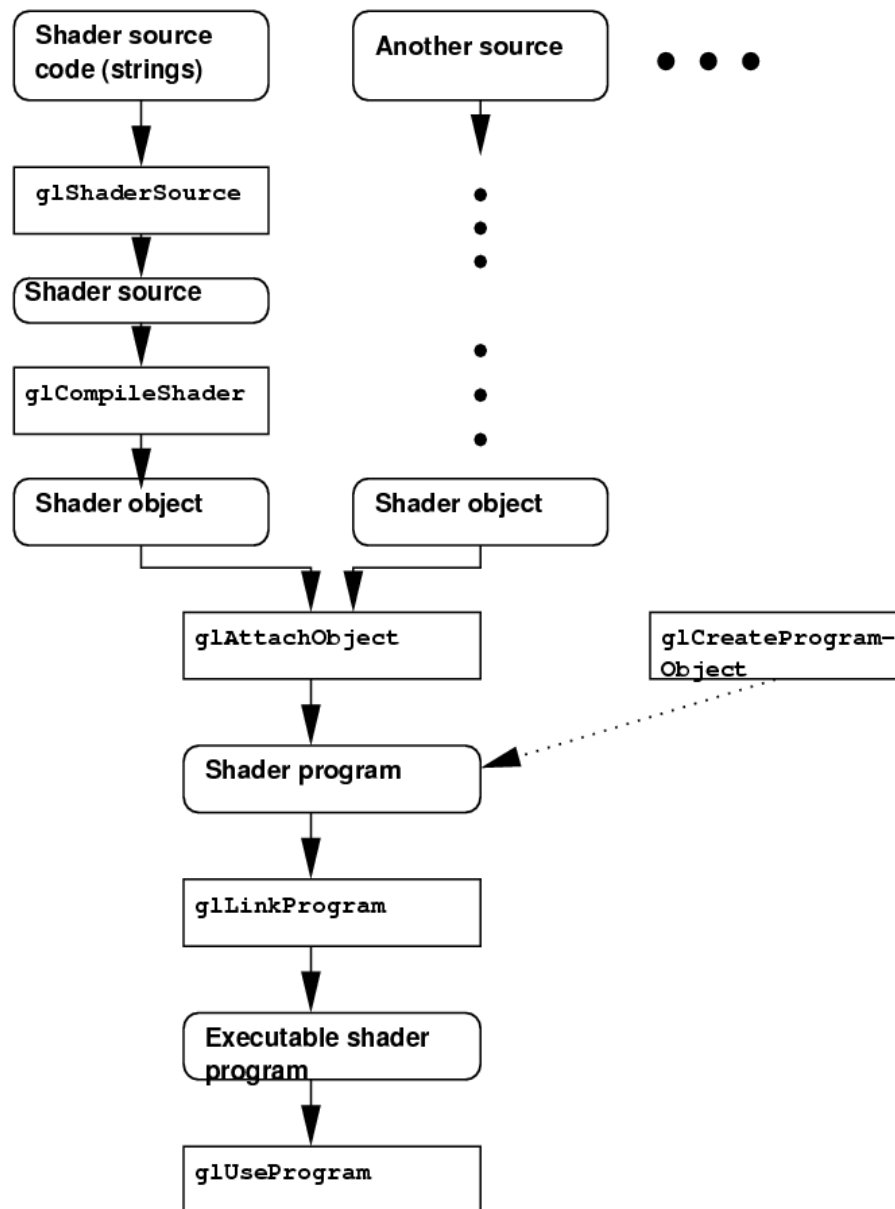


Figure 13.23: Shader creation flowchart.

Bibliography

- Akenine-Moeller, T. & Haines, E. (2002). *Real-time Rendering*, 2nd edn, Natick, MA: A.K. Peters.
- Angel, E. (2005). *Interactive Computer Graphics: a top-down approach using OpenGL*, 4th edn, Addison Wesley.
- Angel, E. (2008). *OpenGL: a primer*, 3rd edn, Addison Wesley. ISBN: 0321398114.
- Astle, D. & Hawkins, K. (2004). *Beginning OpenGL Game Programming*, Premier Press / Thompson Course Technology.
- Bourg, D. M. (2002). *Physics for Game Developers*, O'Reilly.
- Brackeen, D., Barker, B. & Vanhelsuwe, L. (2004). *Developing Games in Java*, New Riders Publishing (Pearson Education).
- Campbell, J. (2008a). Notes on Mathematics for 2D and 3D Graphics, *Technical report*, Letterkenny Institute of Technology. URL. <http://www.jgcampbell.com/msc2d3d/grmaths.pdf>.
- Campbell, J. (2008b). Notes on Mathematics for 2D and 3D Graphics, *Technical report*, Letterkenny Institute of Technology. URL. <http://www.jgcampbell.com/msc2d3d/grmaths.pdf>.
- Castleman, K. (1996). *Digital Image Processing*, Prentice Hall.
- Dave Shreiner (Ed.) and OpenGL ARB (2000). *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2 (The Blue Book)*, Addison Wesley.
- Davis, G. (2005). *Learning Java Bindings for OpenGL (JOGL)*, Lightning Source UK Ltd. ISBN = 42080362X, available for download purchase at: <http://www.genedavissoftware.com/books/jogl/index.html>.
- Eberly, D. H. (2004). *Game Physics*, Morgan Kaufmann. ISBN: 1-55860-740-4.
- Eberly, D. H. (2005). *The 3D Game Engine Architecture: Engineering Real-time Applications with Wild Magic*, Morgan Kaufmann. ISBN: 0-12-229064-X.
- Eberly, D. H. (2007). *3D Game Engine Design: a practical approach to real-time computer graphics*, 2nd edn, Morgan Kaufmann. ISBN: 0-12-229063-1.
- Fernandes, A. R. (2007). Gsl — tutorial, *Technical report*, Lighthouse3D.com. web: <http://www.lighthouse3d.com/opengl/gsl/>.
- Foley, J., van Dam, A., Feiner, S. & Hughes, J. (1990). *Computer Graphics: principles and practice*, 2nd edn, Addison Wesley.

- Foley, J., van Dam, A., Feiner, S., Hughes, J. & Phillips, R. (1994). *Introduction to Computer Graphics*, Addison Wesley. ISBN: 0-201-60921-5.
- Gonzalez, R. & Woods, R. (2002). *Digital Image Processing*, 2nd edn, Prentice Hall.
- Hearn, D. & Baker, M. P. (2003). *Computer Graphics with OpenGL*, Prentice Hall. ISBN: 0131202383.
- Hill, F. (2008). *Computer Graphics Using OpenGL*, 3rd edn, Prentice Hall.
- Hoffmann, B. (1975/1966). *About Vectors*, Dover.
- Koenig, A. & Moo, B. (2000). *Accelerated C++*, Addison-Wesley.
- LaMothe, A. (2002). *Tricks of the Windows Game Programming Gurus: Fundamentals of 2D and 3D Game Programming*, 2nd edn, SAMS.
- LaMothe, A. (2003). *Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization*, SAMS.
- Lengyel, E. (2004). *Mathematics for 3d Game Programming and Computer Graphics*, 2nd edn, Charles River Media.
- Llopis, N. (2003). *C++ for Game Programmers*, Charles River Media.
- Mark DeLoura (2000). *Game Programming Gems*, Charles River Media.
- Mark DeLoura (2001). *Game Programming Gems 2*, Charles River Media.
- Martz, P. (2006). *OpenGL Distilled*, Addison Wesley.
- McReynolds, T. & Blythe, D. (2005). *Advanced Graphics Programming Using OpenGL*, Morgan Kaufmann. ISBN: 1-55860-659-9.
- McShaffrey, M. (2005). *Game Programming Complete*, 2nd edn, Paraglyph Press.
- Murdock, K. (2004). *3ds Max 6 Bible*, Wiley.
- Rollings, A. & Morris, D. (2004). *Game Architecture and Design*, 2nd edn, New Riders Publishing (Pearson Education).
- Rost, R. J. (2006). *OpenGL Shading Language*, 2nd edn, Addison Wesley. ISBN 0-321-33489-2; web: <http://www.3dshaders.com/>.
- Rudolf, F. (2007). *Gls1 an introduction*, *Technical report*, Nehe Productions. web: <http://nehe.gamedev.net/data/articles/article.asp?article=21>.
- Schneider, P. J. & Eberly, D. H. (2003). *Geometric Tools for Computer Graphics*, Morgan Kaufmann. ISBN: 1-55860-594-0.
- Selman, D. (2002). *Java 3D Programming*, Manning.
- Shreiner, D., Woo, M., Neider, J. & Davis, T. (2008a). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1 (The Red Book)*, 6th edn, Addison Wesley. ISBN: 0-321-48100-3. Note that there is a very usable web version of Edition 1.1 at: <http://www.glprogramming.com/red/>.

- Shreiner, D., Woo, M., Neider, J. & Davis, T. (2008b). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1 (The Red Book)*, 6th edn, Addison Wesley. ISBN: 0-321-48100-3. Note that there is a very usable web version of Edition 1.1 at: <http://www.glprogramming.com/red/>.
- Vince, J. (2001). *Essential Mathematics For Computer Graphics Fast*, Springer-Verlag. ISBN: 1852333804.
- Watt, A. (2000). *3D Computer graphics*, 3rd edn, Addison-Wesley.
- Watt, A. & Policarpo, F. (2001). *3D Games, Volume 1*, Addison-Wesley.
- Wright, R. & Lipchak, B. (2004). *OpenGL Superbible*, 3rd edn, SAMS. ISBN: 0-672-32601-9.
- Wright, R., Lipchak, B. & Haemel, N. (2007). *OpenGL Superbible*, 4th edn, Addison-Wesley. ISBN: 0-321-49882-8; web: <http://www.starstonesoftware.com/OpenGL/>.
- Xiang, Z. & Plastock, R. (2000). *Computer Graphics (Schaum's Outlines)*, 2nd edn, McGraw-Hill.