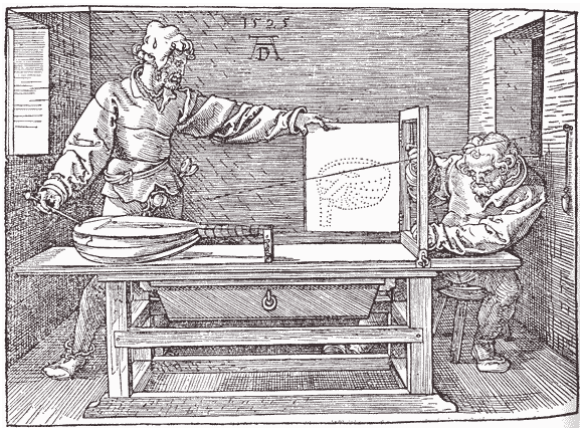
CS324 Computer Graphics – OpenGL Labs

July, 2014



Version 2.4
03/10/2016

Contents

1 Introduction

1.1 What is OpenGL?

OpenGL stands for Open Graphics Library and is a device independent API which allows the writing of graphics applications by accessing the functionality of graphics hardware, such as GPUs. It is designed to be efficient and models hardware functionality. Because of this, it sometimes feels very low-level. Surprisingly, it consists of only about 200 functions.

OpenGL is based on a stable standard and now includes abilities to access the programmable elements of GPUs (shaders) by the GLSL (GL Shading Language). More on that later in the course...

1.2 Programmer's models of OpenGL

From a programmer's point of view, OpenGL is a just a set of functions which you can access through C or C++: these functions take parameters and mostly produce graphical output. For example, we can draw a line by specifying its end points, describing its colour and then sending it to OpenGL to draw in a given window on the screen. Most graphics APIs therefore separate the specification of the graphics object (e.g. line) from its attributes and how it should be displayed. This last part matters as how we "see" the line depends on our "viewpoint" (or camera position).

1.2.1 Simple Graphics Pipeline

In the course, we will study computer graphics by considering it as a pipeline of operations performed consecutively on some data, to produce other data. A simple graphics pipeline takes in vertices at one end and produces pixels at the other. This might seem strange at first, but as our display is a grid of *pixels* (of different colours), that is always going to be the output of a graphics pipeline.

The input is **vertices**, which is just a fancy way of saying points with co-ordinates (two-dimensional, or three dimensional). So, in our line example, the input to a graphics pipeline is the endpoints of the line (two vertices) and the output is a set of pixels which connect the endpoints on the screen. But to produce the output, a graphics pipeline like OpenGL has to process the geometry (the vertices), rasterise them into a pixel grid, set the colours and produce an output suitable for the device (the **framebuffer** memory), which is then mapped to the display by the video function of the graphics card. So the pipeline consists of a Geometry Processor, Rasterizer, Fragment Processor and a Framebuffer (note we talk about a pixel with a colour as a **fragment**).

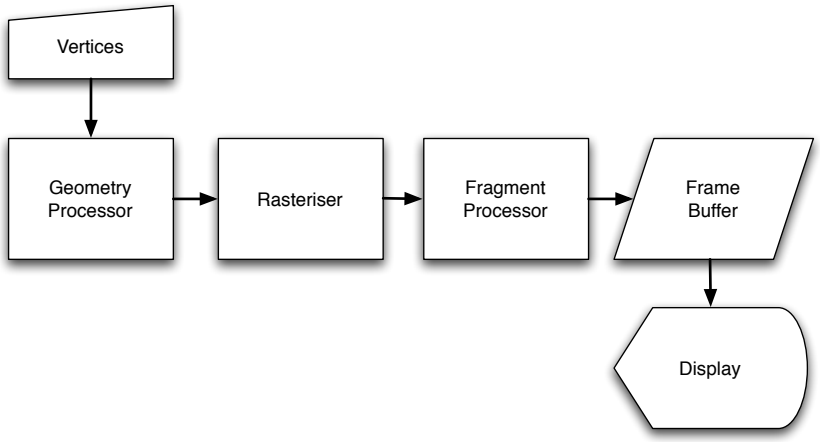


Figure 1: Diagram of a simple graphics pipeline.

1.2.2 OpenGL State Machine

Another view of OpenGL is as a State Machine where all the OpenGL API functions are modifying the state. The inputs are vertices and the outputs are pixels. We can then see that the API functions variously, specify input states and values (vertices), but mostly control the state by forcing OpenGL to produce outputs or change state by specifying attributes (e.g. colours), or the position and orientation of cameras etc.

1.3 What is in OpenGL?

There are about 200 functions in OpenGL, e.g. `glColor3f()`[1] which sets the current drawing colour, and they can be divided into groups:

- Primitive functions (**primitives** in computer graphics are the graphics objects). There are functions to specify geometric primitives such as points, line and polygons, but also bit maps (texture maps).
- Attribute functions which control the appearance of primitives. There are functions to set the colours and widths of lines for example. In 3D, we can set the surface properties of polygons.

[1]Many of the OpenGL functions are named with something like `3f` or `2i` or `4d`, etc., where the number is the dimension of the function (and usually the number of parameters it takes), and the `i|f|d` stand of for the type the function operates on, i.e. `i` is short for `int`, `f` is short for `float` and d is short for `double`.

- Viewing functions which specify the **cameras** from which the graphics scene is viewed. Cameras have positions and directions and specify to OpenGL precisely how vertices are projected onto the output window.
- Control functions which enable and disable certain OpenGL features, such as hidden line removal or blending.
- Query functions which allow OpenGL state variables to be read or the capabilities/conformance of the particular OpenGL implementation you are using to be determined.
- Input and Window functions. These are function which are not part of OpenGL but are essential to allow us to use OpenGL interactively, e.g. take mouse and keyboard input from an OpenGL window and position and resize the window. These are in an auxiliary library called GLUT[2].

1.4 Which Version?

OpenGL up to and including version 2.1 implements the "fixed" pipeline standard (known as **immediate mode**). After version 3.0, OpenGL also support the **programmable pipeline**. The programmable pipeline uses what are known as Shaders (which are small programs) to control the GPU hardware. Shaders are more efficient and flexible than the immediate mode architecture and many current OpenGL implementations support fixed pipelines using shaders. You can think of the GPU as a separate processor and it's programmed by the OpenGL shading language, GLSL. Let's forget about shaders for the moment and get started with our first OpenGL example.

1.5 References

I've used a number of source references for these tutorials.

1. Most of the labs follow the order of the chapters in the *OpenGL: A Primer*, but Edward Angle (available in its 3rd edition from Amazon for about £23).
2. Other material, such as some of the GLSL examples are based on online tutorials, e.g. OGLDEV at `http://ogldev.atspace.co.uk/`, but there are plenty of others.
3. A comprehensive source for OpenGL documentation and coding resources can be found at `opengl.org`.

[2]you would have to be a masochist not to use GLUT and try to write your own, but other windowing APIs can be used with OpenGL.

4. Two invaluable books are what is known as the Red Book, the *OpenGL Progamming Guide*, which describes the immediate mode of OpenGL (online version at `http://www.glprogramming.com/red/`), and the Orange Book, *OpenGL Shading Language*, by Rost for using the programmable pipeline.

5. The OpenGL 3D Shading Tutorials at `http://www.lighthouse3d.com/` are great too.

1.6 Acknowledgements

I'd like to thank James Archbold and James Dickson[3] for carefully working through these rather mad set of labs, debugging them and making some very useful and important corrections. Thanks guys!

[3]they are happy to be called by the surnames, or James A and James D!

2 Lab 1 - Getting Started

The aim of this lab is to get you started with OpenGL with a simple "Hello, World!" example. In this lab we will learn about:

- compiling a simple C++ program using a **Makefile**
- using GLUT include files
- using GLUT **callback functions** and re-draw events
- creating a graphics window
- drawing simple primitives with `glBegin()` and `glEnd()`
- working with vertices and colours

2.1 Downloading and compiling the example source files

Start by changing to your home directory using `cd` < *enter* >, then `unzip` archive of the laboratory work files from the `/modules/cs324` directory, like this:

```
$ cd
$ unzip /modules/cs324/cs324-labs.zip
```

This will create a sub-directory called `cs324-labs` and directories below that, one for each lab. Still using the `bash` terminal shell, `cd` into the first lab directory:

```
$ cd cs324-labs/lab-1
```

Use `ls` to list the files in this directory. You will see that there are C++ source programs with extensions `.cpp` and Unix Makefiles: `Makefile.linux` and `Makefile.OSX`. To compile programs on linux, you can use the `make` command with the `-f` option to use the `.linux` makefile and name the build target[4], e.g.

```
$ make -f Makefile.linux simple
```

If you don't want to keep typing the `-f` option, you can make a symbolic link using `ln` to link `Makefile` to `Makefile.linux` (or `Makefile.OSX` on Macs), like this:

```
$ ln -fs Makefile.linux Makefile
```

Now to *make* targets, you can simply type:

[4]On Macs, you need to use the `Makefile.OSX` makefile.

```
$ make simple
```

If all has worked, you can run the first program from the command line:

```
$ ./simple
```

To stop this program, use the kill key combination `CTRL-C` in the `bash` terminal window.

2.1.1 Unix manual pages using `man`

You can quickly find help on any of the OpenGL functions by searching the `man` pages, e.g.

```
$ man glVertex

GLVERTEX(3G)                                              GLVERTEX(3G)

NAME
       glVertex2d,  glVertex2f,  glVertex2i,  glVertex2s, ...
       ... - specifies a vertex

C SPECIFICATION
       void glVertex2d( GLdouble x,
                        GLdouble y )
       void glVertex2f( GLfloat x,
                        GLfloat y )
       void glVertex2i( GLint x,
                        GLint y )
       void glVertex2s( GLshort x,
                        GLshort y ) ...
```

2.2 Program `simple`

Here is a full listing of the first program which in the C++ source file `simple.cpp`:

```
1  //
2  // Here are the required include files
3  // note that glut.h itself includes glu.h and gl.h
4  #ifdef __APPLE__
5  #include <GLUT/glut.h> // if we are on a Mac
6  #else
7  #include <GL/glut.h> // on linux
8  #endif
9
10 // ...define what OpenGL must do when asked to redraw
      the window
11 //  note we can call this function anything we want but
      remember to
12 //  set your chosen name in the glutDisplayCallbackFunc
      () in main()
13 //
14 void display()
15 {
16   glClear(GL_COLOR_BUFFER_BIT); // clear the window to
        default background colour
17
18   glColor3f(1.0f, 0.0f, 0.0f); // set RGB values of
        colour to draw
19   glBegin(GL_LINES); // draw lines between pairs of
        points
20     glVertex2f(-0.5f, -0.5f); // from endpoint
21     glVertex2f(0.5f, 0.5f); // to endpoint
22   glEnd();
23
24   glFlush(); // force the output of any draw operations
25 }
26
27
28 //
29 // ...every C/C++ program needs a main()
30 //  argc is the number of command line arguments
31 //  argv[] is an array of null terminated strings of
      command line arguments
```

```
32 //       note that argv[0] is the program name
33 //
34 int main(int argc, char* argv[])
35 {
36   glutInit(&argc, argv); // we can pass certain X
       windows attributes to GLUT
37
38   glutCreateWindow("Simple"); // a named window of
       default size and position
39   glutDisplayFunc(display); // set a "display" or "
       redraw" callback
40
41   glutMainLoop(); // go into the main loop and wait for
       window events...
42
43   return 0; // safe to return 0 to shell unless error
44 }
```

Let's go through the program carefully. If you are unfamiliar with C or C++ you need to first know that **entry point** to the code is through the `main()` method, which is the second method listed (at the end). `main()` takes two arguments, `argc, argv`, which are command line arguments including the program name in `argv[0]`. Note that C strings are `NULL` terminated arrays, hence the type of `argv` is an array of string pointers `char*`.

2.2.1 OpenGL Initialisation and Window Creation

OpenGL initialisation: the first actual statement and the following three will be seen in all of the OpenGL examples and are calls to GLUT window creation and event handling functions. Note that the prefix `glut` to these functions tells us they belong to GLUT. Roughly speaking, GULT functions call GL functions. An OpenGL program will be a combination of GLUT/GL and GLU. We pronounce these prefixes as **glut**, **gee-ell**[5] and **glue** or **G-L-U**.

`glutInit()` takes the `argc, argv` parameters and initialises the GLUT window handler functions and GL. This allows you to pass environment-dependent to your program from the command line, for example, X-Windows type window attributes to `glutInit()` such as *-iconic*, *-display* and *-geometry*, but let's not worry about that as it's not important for us. Note that `glutInit()` must be the first thing called before any other GLUT function is used.

[5]not related to **Kal-el**, as far as I know

`glutCreateWindow()` creates an OpenGL context along with a window of default size and position and gives it a title (given as an literal string argument). In this example, we expect our graphics **window** to have the title "Simple".

2.2.2 Window Event Handler and GLUT Event Loop

OK, now that we've asked for a graphics window, now what? Well, you need to understand once we've created a window, to draw anything (graphics **primitives**) we have to tell GLUT the name of a drawing method it will call to draw the window contents – this can be the name of any function in our program which returns `void` (nothing).

`glutDisplayFunc(display)` tells GLUT that `display` is the name of the function which it should use whenever the window manager wants to update the graphics in the graphics window (which we've just created). If we don't set `display` as the **callback function**, we won't see anything! Note that at anytime you can request a display event by calling `glutPostRedisplay()`. As we will see later, this function is useful when we want to respond to mouse or keyboard events and update the graphics.

The program is almost finished! The final statement calls a function *from which we don't expect to return.*

`glutMainLoop()` calls the OpenGL event handler loop which will trap window, mouse and keyboard events and pass them to any callbacks we have specified, such as `display()`. We will see later in the labs how to handle keyboard and mouse events to allow us to interact with the displayed graphics. As mentioned, we don't expect to return from `glutMainLoop()`, so code following this line will not execute. The GLUT loop will gracefully close our program if we try to close the window or send it a terminate signal.

2.3 Display Handler (drawing geometry)

Now we can look at the other method in our program, which is called `display`[6]. It is a `void` method which does the actual graphics. Remember that `display` is called every time there is a draw event. So for example, it will be called first time the window is drawn, whenever, the window is moved, resized or revealed, etc.

`display()` does not take any arguments and returns none. Each time `display` is called you are obliged to redraw everything you want to see, every time. This is a scary thought because, this means that if you are drawing 1,000,000 polygons in the display window, you need to repeat this each and every time the draw event

[6]You call it what you like, but the name has to be registered in the `main()` using `glutDisplayFunc()`

is received. OpenGL provides some efficient ways to specify arrays of vertex lists to the GPU.

`glClear(GL_COLOR_BUFFER_BIT)` clears the colours of the **display buffer** to the **clear colour**. By default this is black.

`glColor3f()` set the current drawing colour and this variant takes three, floating point parameters for the red, green and blue values. These are in the range $[0, 1]$. White is when all the three colours have value `1.0f`. In C++, the `f` after the number designates it as a single-precision floating point literal (hard-coded value).

Finally, we get to draw something and this is done within the pair of commands:

```
glBegin(<primitve type>)
  // specify vertices and colours here
glEnd();
```

In the example, the `<primitive type>` is set to be `GL_LINES`, which means that the subsequent pairs of vertices are interpreted as the end points of **separate lines**.

`glVertex2f()` takes two floating point arguments (we know this because the naming convention that the function ends with `2` and then `f`). If I want to specify two integer vertices, I would use `glVertex2i()`. If it was three doubles, I would use `glVertex3d()` and so on.

`glEnd()` tells GL that the current **display list** (vertices) is finished and can be drawn into the buffer.

`glFlush()` forces GL to actually draw all the display lists to the frame buffer and in this program, the line will appear.

Note that `display()` draws directly to the frame buffer, so when `glFlush()` is called, the windowing system will show the drawing taking place. For small amounts of **geometry** being **posted** to the buffer, this is not a problem. However, if we were drawing thousands or millions of lines, this would be painful to watch because the video card would keep refreshing the display each time the frame buffer was changed by a new primitive. In the next program, we will use what is known as **double buffering** to avoid this problem.

2.4 First Exercises

1. If you haven't already, compile and run `simple`. Look at the code, where is the red line being drawn? What happens if you resize the window?

2. Modify the `display()` method to draw a line between two other end points. What happens if you specify endpoints which are not in the range `[-1,1]`?

3. Make the line have a yellow colour. Note that you can use `glLineWidth()`

to change the width of the line. Try moving `glLineWidth(2.0f)` before and after the `glColour3f()` statement, and see how the line changes.

4. Draw a square using pairs of calls to `glVertex2f()` within the `glBegin()` and `glEnd()` statements. See if you can reduce the number of `glVertex` calls. (Hint: check the man page for `glBegin`).

5. Add the line `#include <iostream>` to the top of the file. Add debug statements to `display()` to investigate how often and when it is called by the window manager. If you've never used C++ before, you will want to look up the `cout` family of commands for printing to the console. For example,

```
std::cout << "Hello, World!" << std::endl;
```

writes the text `Hello, World!` to the console followed by a line break.

2.5 Program `double`

Here is a full listing of `double.cpp`, which is a lot like the first program but with some important new things to help you better specify the placement and size of the graphics *window* and uses what is known as **double buffering**:

```
1  #ifdef __APPLE__
2  #include <GLUT/glut.h>
3  #else
4  #include <GL/glut.h>
5  #endif
6
7  // redraw callback
8  void display()
9  {
10   glClear(GL_COLOR_BUFFER_BIT);
11
12   // draw stuff -- into the backbuffer
13
14   glColor3f(1.0f, 1.0f, 1.0f); // white
15
16   glBegin(GL_POLYGON); // filled polygon
17     glVertex2f(-0.5f, -0.5f);
18     glVertex2f(0.5f, -0.5f);
19     glVertex2f(0.5f, 0.5f);
20     glVertex2f(-0.5f, 0.5f);
21   glEnd();
22
23   glutSwapBuffers(); // swap the backbuffer with the
       front
24 }
25
26
27 //
28 int main(int argc, char* argv[])
29 {
30   glutInit(&argc, argv);
31
32   // set the GL display mode: double buffering and RGBA
       colour mode
33   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA); // flags
       bitwise OR'd together
```

```
34
35    glutInitWindowSize(512, 512); // window size
36    glutInitWindowPosition(50, 50); // where on screen
37
38    glutCreateWindow("Double Buffer");
39
40    glutDisplayFunc(display); // set display callback
41
42    glClearColor(0.0f, 1.0f, 0.0f, 0.0f); // clear colour
         -- green!
43
44    glutMainLoop(); // go into the main loop and wait for
         window events...
45
46    return 0;
47 }
```

This program is very much like our first example, `simple.cpp`. In fact, you will see that almost all of the programs in the subsequent labs are going to have the code that is in `double.cpp` with the main changes in the display handler.

Before we discuss **double buffering**, we need to highlight the new things in `main()`. Remember that the purpose of the GLUT statements in `main()` are a **preamble** or **mantra** to summon the GLUT gods to our will. After the call to `glutInit()` we have three new statements (lines 33 to 36):

```
  glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA);

  glutInitWindowSize(512, 512); // do this before
     glutCreateWindow()
  glutInitWindowPosition(50, 50);
```

The first is a call which specifies that we want to apply double buffering by the use of two frame buffers, a **back** buffer and a **front buffer**. The basic idea is that we will draw our geometry in the current back buffer, and when we are ready, we will ask GLUT to show this buffer (making it the front buffer) and make available the previous front buffer ready for our next draw event. This sound complicated, but we simply replace the call to `glFlush()` at end of the display handler with `glutSwapBuffers()`.

The argument to `glutInitDisplayMode()` also `OR`s in a second **flag**, `GLUT_RGBA` which additionally tells GLUT to make the display buffers support the three colours (RGB) as before and a transparency channel (**alpha channel**), hence the `RGBA`. We will look at transparency another time. Actually, by default we get

`GLUT_RGB`, but it is normal to use `GLUT_RGBA` and does no harm.

The other two new statements give us control over the graphics window we want to create by allowing us to say how big it should be (in pixels wide and pixels tall), and where the top-left hand corner should be on your computer screen.

In `display()`, there are a few changes to what we had before. We are now drawing a polygon using `GL_POLYGON` and as mentioned, we have to use `glutSwapBuffers()` instead of `glFlush()`. You will only notice any change if you were pushing out a lot of geometry in the display list generation.

2.6 More Exercises

1. Create a new program based on `double.cpp` to draw a circle in the window of a given radius. Hint: look at my program `circle-points.cpp` which has code in it to generate vertices of a circle.

2. Create a program to draw a square sided spiral in the window.

3. Create a program to draw a graph of a cubic function, $y = x^3$, in the range $x = -1$ to $x = 1$ and a graph of $sin(3x)$ for the same range.

3 Lab 2 - Worlds and Windows

In this lab we are going to look at a really important concept in Computer Graphics, that of how we specify the mapping between the coordinates we use to represent our graphical objects or **models** (our points, lines, robot arms, racing cars, Tie Fighters and assorted weaponry...) and the coordinates of the pixel buffer or **window**. So we have a fundamental mapping or **transformation** between **World Coordinates** and **Window Coordinates**.

Why is this useful and necessary? Well the fact is that we want to fix a measurement system in the world we want to represent so we can put things where we want, at the size and scale we have determined, but these coordinates can't possibly be represented by our display which has a limited resolution. OpenGL gives us control over the mapping or the **world to window transformation**. In graphics, this is known as the **projection transformation** which in 2D is an **orthographic projection**.

You may (or may not) have wondered why it was that in program `double` in the first lab, our window was set to be 512×512 pixels in size, but we were drawing geometry in the range $[-1, 1]$. Did you not think that was odd and particularly unhelpful? The reason for this is that by default, OpenGL sets the projection transformation to map the corners of your window (however big or small you make it) to the range $[-1, 1]$ in both X and Y directions.

In this lab, as well as the projection transformation, we will look at how OpenGL exposes a related transformation of the viewing pipeline, the **model transformation** which allows us to place and rotate our objects into the world, before the projection takes place. This lab is about:

- Orthographic projection using `gluOrtho2D()` OpenGL's projection matrix
- Matrix stacks and how to manipulate them to transform objects we want draw
- Rotations and translating of objects using `glRotatef()` and `glTranslatef()`

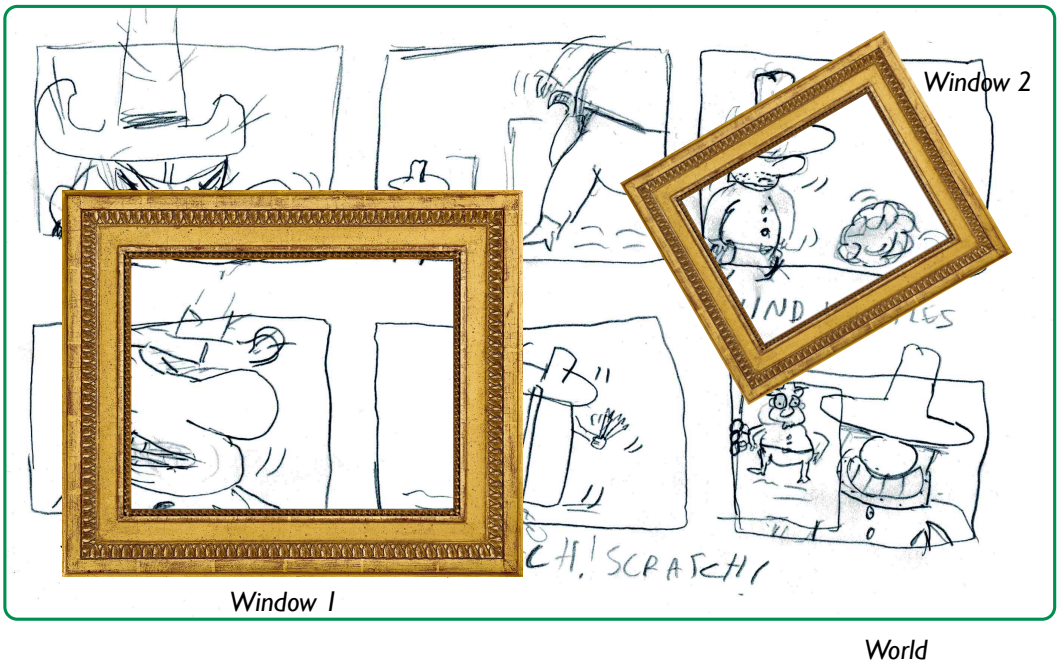


Figure 2: A Window on the World

3.1 Program `ortho`

This program is called `ortho` because it sets up and uses the fundamental 2D viewing method of **orthographic projection**. In 2D, this projection is just a translation and a scaling. Remember, what we want to achieve is an independence from the World Coordinates we want to work in and the rectangular area of the screen (in pixels) we want to use as our output graphics window.

Imagine a rectangular picture frame on a large, flat piece of paper on a desk. Imagine that this paper has a drawing on it. As we move the frame around, it behaves like a 2D *camera*, it will "capture" only parts of the drawing as we move it around. We can pan or **translate** our camera, but we might allow it to tilt or **rotate** it. If we stop and put is somewhere, we are specifying its position. If we further allow the ability to stretch our frame (in width and height), then we will capture more or less of the world.

In OpenGL, we manipulate our **window** on the **world** with `gluOrtho2D()`. This takes four parameters which are the bottom-left and top-right hand corner of our imaginary picture frame in world coordinates. By default, OpenGL, sets up the frame to have corners $(-1, -1)$ and $(1, 1)$. What this means is that it will make your world have the range $[-1, 1]$ in X and $[-1, 1]$ in Y. That is all you are allowed to see.

So the next question is, what about the `glutInitWindowSize()` in the `main()`

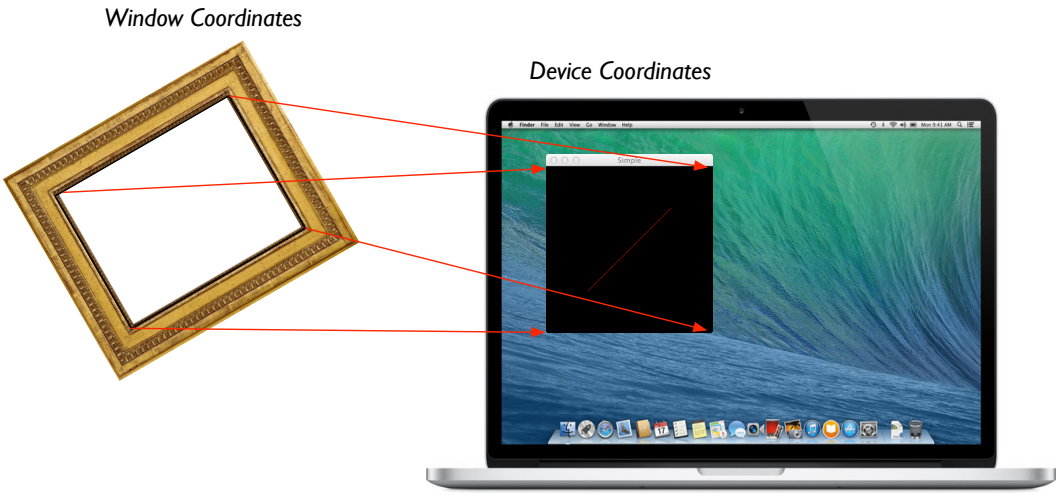


Figure 3: Window to Device coordinates done by OpenGL

method, what does that do? Well, this instructs GLUT to liaise with your windowing system and GPU to map a pixel buffer of a given **pixel** size on to your computer screen. What is hidden from you is that in the viewing pipeline, OpenGL, will eventually map the corners of the orthographic frame (specified in world coordinates) onto the your graphics window. This second coordinate mapping is how OpenGL achieves **device independence** and it's not something you need to worry about. The main thing is that whatever size of pixel window you specify in `glutInitWindowSize()`, OpenGL will ensure that the world coordinates of the bottom-left corner will be at bottom left of your screen, and the world coordinates of the top-right corner will be at the top-right of your screen.

Let's look at the code of `ortho` in detail.

```
1  #ifdef __APPLE__
2  #include <GLUT/glut.h>
3  #else
4  #include <GL/glut.h>
5  #endif
6
7  void display()
8  {
9    // clears to current background colour
10   glClear(GL_COLOR_BUFFER_BIT);
11
12   // drawing colour
```

```
13    glColor3f(1.0f, 1.0f, 1.0f);
14
15    // we should see this in our window
16    glBegin(GL_LINE_LOOP);
17      glVertex2i(250, 250);
18      glVertex2i(750, 250);
19      glVertex2i(500, 750);
20    glEnd();
21
22    glutSwapBuffers();
23  }
24
25  void init()
26  {
27    // select viewing transformation matrix to modify
28    glMatrixMode(GL_PROJECTION);
29
30    // set identity on matrix
31    glLoadIdentity();
32
33    // specify bottom-left and top-right corners
34    //     of window in world coordinates
35    // takes LEFT, RIGHT, BOTTOM, TOP
36    gluOrtho2D(0, 1000, 0, 1000);
37
38    // set background colour and transparency
39    // takes RGBA values
40    glClearColor(0.0f, 0.0f, 1.0f, 0.0f); // blue
41  }
42
43  int main(int argc, char* argv[])
44  {
45    glutInit(&argc, argv);
46    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA); // flags
        bitwise OR'd together
47
48    // display window size is in pixels on screen
49    glutInitWindowSize(512, 512);
50    glutInitWindowPosition(50, 50);
51
```

```
52    glutCreateWindow("Ortho2D ");
53
54    glutDisplayFunc(display);
55    init(); // initialise the viewing
56    glutMainLoop();
57
58    return 0;
59 }
```

`main()` Same as previously except before going into the `glutMainLoop()`, we call a new method, `init()` to specify a particular projection for our world.

`init()` All projection transformations are controlled in OpenGL by the projection **matrix**. OpenGL has a method called `glMatrixMode()` which takes as arguments the name of a matrix we want to change, in this case it is `GL_PROJECTION`, the other we will meet later is selected by `GL_MODELVIEW`. The matrices are **state variables** of OpenGL, and we can affect them setting their values or multiplying them by a new matrix. Matrices are stored in *stacks* – each matrix mode has a separate stack, which we will talk more about shortly.

`glLoadIdentity()` overwrites the current matrix with the **identity matrix**.

`gluOrtho2D()` specifies the sides of the window (or 2D camera) given in world coordinates (i.e. the coordinates we want to use to draw stuff in `display()`). The window is given sides with X-min and X-max, and Y-min and Y-max values, or LEFT, RIGHT, BOTTOM, TOP. Note that this is a GLU function and what it actually does is to create a transformation matrix from the given corners and multiply it on to the `GL_PROJECTION` matrix. We will learn exactly what matrix it creates later on in the course. For now, don't worry about it.

We also do a couple of other things, which is to specify a background colour for our graphics, using `glClearColor()`. This is the colour used when `glClear()` is called in `display()`.

The final thing to say about this program is to note the coordinates which are used in the vertices, drawn in `display`. We can now use our world coordinates, knowing full well where our viewing window is in relation to what we are attempting to draw.

3.2 Exercise

1. Modify `ortho` to move your 2D window elsewhere in the world, i.e. *pan* your camera (Hint: move is slightly to keep parts of your drawing in the display). Does it do what you expect?

3.3 Program `rotate`

As well as the `GL_PROJECTION` matrix, we also have another OpenGL matrix we can manipulate, called the **model matrix**. This is accessed by the `GL_MODELVIEW` flag.

```
glMatrixModel(GL_MODELVIEW); // manipulate model
   matrix
glLoadIdentity(); // null operation
```

The model matrix operates on any display geometry *before* it is drawn into the window. So in effect, it changes the input World Coordinates to new World Coordinates. It is easier to think of it as the transformation which *puts* the model into the scene. This is really useful, as it allows us to put the same objects repeatedly at different positions and orientations (and scales if we like), all over our world. Imagine that you are Jon Snow and you need a few more Crows to help fight the White Walkers, well you could apply 100 multiple model transformations to yourself in a spread around you at say one meter intervals? Now wouldn't that help? Of course, each model transformation for Jon2 and Jon3 etc. would be a different translation in x and y from your central position.

A really nice feature of these OpenGL state matrices is they are actually the top of a `matrix stack` (or a stack of matrices). OpenGL has two methods to *push* and *pop* to effectively save and restore the current matrix transformation state:

```
glMatrixMode(GL_MODELVIEW);

glPushMatrix(); // push down the current matrix

//... do something drawing

glPopMatrix(); // pop off the last used matrix
```

These stacks operate on both the `GL_PROJECTION` and `GL_MODELVIEW` matrices.

Why is this useful? Well it means, that to continue our GoT example, we can do a `glPushMatrix()` and then apply a different `glTranslatef()` for each Jon, draw him, and glPopMatrix(), ready for a new one.

```
glMatrixMode(GL_MODELVIEW);

// lots of useful Crows in a fight
for (size_t i=0;i<max_jon_snows;i++)
{
  glPushMatrix();
    glTranslatef(...); // placement
```

```
      draw_jon_snow();
    glPopMatrix();
  }
```

There are a number of built in transformations you can use that are applied to the currently selected state matrix. These are `glTranslatef()`, `glRotatef()`, `glScalef()`, which you can look up in the `man` pages.

Here is the full listing of `rotate.cpp`:

```
1  #ifdef __APPLE__
2  #include <GLUT/glut.h>
3  #else
4  #include <GL/glut.h>
5  #endif
6
7  #include <stdlib.h>
8  #include <stddef.h>
9
10 void draw_triangle()
11 {
12   // in model cooridnates centred at (0,0)
13   static float vertex[3][2] =
14     {
15       {-1.0f, -1.0f},
16       {1.0f, -1.0f},
17       {0.0f, 1.0f}
18     };
19
20   glBegin(GL_LINE_LOOP);
21     for (size_t i=0;i<3;i++)
22       glVertex2fv(vertex[i]);
23   glEnd();
24 }
25
26 void display()
27 {
28   glClear(GL_COLOR_BUFFER_BIT);
29
30   glColor3f(1.0f, 1.0f, 1.0f);
31
32   glLineWidth(2.0f);
33
```

```
34    // work on MODELVIEW matrix stack
35    glMatrixMode(GL_MODELVIEW);
36    glPushMatrix();
37
38      // translate it to where we want it
39      glTranslatef(500.0f, 500.0f, 0.0f);
40
41      // scale it up
42      glScalef(250.0f, 250.0f, 1.0f);
43
44      // rotate anticlockwise in-plane by 60 degrees
45      glRotatef(60.0f, 0.0f, 0.0f, 1.0f);
46
47
48      // render primitives
49      draw_triangle();
50
51    glPopMatrix(); // done with stack
52
53    glutSwapBuffers();
54  }
55
56  void init()
57  {
58    glMatrixMode(GL_PROJECTION);
59    glLoadIdentity();
60    gluOrtho2D(0, 1000, 0, 1000);
61    glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
62  }
63
64  int main(int argc, char* argv[])
65  {
66    glutInit(&argc, argv);
67    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA);
68    glutInitWindowSize(512, 512);
69    glutInitWindowPosition(50, 50);
70    glutCreateWindow("Rotate");
71    glutDisplayFunc(display);
72
73    init();
```

```
74    glutMainLoop();
75
76    return 0;
77 }
```

In the code, I have used `glRotatef()` which specifies a rotation matrix multiplied into the current stack. It takes an angle, in degrees, and an axis around which to rotate. In 2D, this is simply the z-axis `(0.0f, 0.0f, 1.0f)`. We will learn about 3D rotations in the course.

3.4 Exercises

1. Modify the `display()` method of `rotate()` and test out other rotation angles.

2. Is the order of the transformations given important? (remember the presented transformation is applied to the top of the stack).

3. Modify the program to draw 5 squares in the window at 5 random positions and angles using `glPushMatrix()` and `glPopMatrix()` appropriately. You can use `drand48()` to return a random number in the range $[0.0, 1.0]$ and scale it accordingly like this:

```
float rand_x = float(drand48() * 512); //
   uniform between 0 and 512
float rand_angle = float(drand48() * 360); //
   uniform between 0 and 360
```

4. Make a new program called `tree.cpp`, copying `rotate.cpp`, and modify it to generate a simple fern-like fractal. A fractal is a self-simlar pattern and can be produced by rotating, translating and scaling a given primitive recursively.

4 Lab 3 - Interactivity

An important part of using computer graphics for visualisation, games, etc. is interaction. As we have seen, core OpenGL does not do this, but capturing and handling events from the user is provided through GLUT functions. We have used the display callback to capture re-draw window events and draw some simple graphics.

In this lab, we will look at further callback mechanisms aimed at capturing mouse, keyboard events, and also installing an **idle routine**, which is ideal for animation. We will also look at how to draw text and make OpenGL **display lists**.

In this lab we will learn about:

- `glutKeyboardFunc()` to install a keyboard (**fast key**) handler.
- `glutMouseFunc()` and its sister `glutMouseMotionFunc()` to capture and use mouse button and pointer motion events.
- `glutIdleFunc()` to install an **idle** routine to allow us to use time when the user is not doing anything else, perhaps to animate the graphics.
- how to make and render **display lists** for geometry which does not change often.
- how to write text into the graphics window.

4.1 Program `keyboard`

We'll dive straight in and look at the example program, `keyboard.cpp`, to see how to capture and use keyboard events[7].

In `main()`, there are now two further callbacks defined:

```
// handlers for keyboard input
glutKeyboardFunc(keyboard);
glutSpecialFunc(special);
```

These are two function which both have `void` returns and are given three arguments: `key`, `x` and `y` position when a key is pressed. In this program, we only use the `key` character argument:

[7] OpenGL does not have **widgets** but there are ways to make pop-up menus in GLUT, but it is far too complicated to explain here and overly cumbersome for most small programs.

```
void keyboard(unsigned char key, int, int)
{
    switch (key)
    {
        case 'q': exit(1); // quit!

        // clockwise rotate
        case 'r': g_angle += g_angle_step; break;
    }

    glutPostRedisplay(); // force a redraw
}
```

A `switch` statement is used to check the values of `key`. In this case, if it is `'q'` the program is forced to exit using the system `exit()` call. If it is the `'r'` character, we increment the global `g_angle` variable by `g_angle_step`, these are all defined a the top of the program:

```
// values controlled by fast keys
float g_angle = 0.0f;
float g_xoffset = 0.0f;
float g_yoffset = 0.0f;

// increments
const float g_angle_step = 10.0f; // degrees
const float g_offset_step = 32.0f; // world coord units
```

Similarly, the `special()` method, responds to the arrow keyboard keys:

```
// any special key pressed like arrow keys
void special(int key, int, int)
{
    // handle special keys
    switch (key)
    {
        case GLUT_KEY_LEFT: g_xoffset -= g_offset_step;
           break;
        case GLUT_KEY_RIGHT: g_xoffset += g_offset_step;
            break;
        case GLUT_KEY_UP: g_yoffset += g_offset_step;
           break;
```

```
        case GLUT_KEY_DOWN: g_yoffset -= g_offset_step;
           break;
    }

    glutPostRedisplay(); // force a redraw
}
```

The `man` page on `glutSpecialFunc()` will show you what are the other `GLUT_` constants defined.

Note that in both the handlers, we use the `glutPostRedisplay()` to post a redisplay event so that the program will call our `display()` method and re-draw the triangle with the new parameters. If you look in `display()`, I've used the `g_angle` as a parameter to `glRotatef()`, to change the angle of rotation, and `g_xoffset` and `g_yoffset` to induce different translations before the triangle is drawing.

```
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glTranslatef(500.0f+g_xoffset, 500.0f+g_yoffset,
            0.0f);
        glScalef(100.0f, 100.0f, 1.0f);
        glRotatef(g_angle, 0.0f, 0.0f, 1.0f);
        draw_triangle();
    glPopMatrix();
```

And that's all there is to it. If you compile and run the program, you can use the keyboard key `'r'` to spin the triangle, the arrow keys to move it and, when the excitement is over, `'q'` to quit.

4.2 Exercises

1. Modify the code to trap the keys `'+'` and `'-'` and make them change the size of the triangle by some factor bigger and smaller respectively.

2. Add debug to `keyboard()` to print out the X and Y positions. You will have to declare the parameters in the function definition, like this:

```
void keyboard(unsigned char key, int x, int y)
{
  // blah...
}
```

4.3 Program `mouse`

The next program to look at is `mouse.cpp` which installs a mouse event handler. First look at `main()`:

```
    // mouse event handlers
    glutMouseFunc(mouse_click);
    glutPassiveMotionFunc(mouse_motion);
```

The first event handler which I've called, `mouse_click()`, needs to have three parameters:

- `int button` which returns which button was pressed, e.g. `GLUT_RIGHT_BUTTON` or `GLUT_LEFT_BUTTON`, etc.
- `int state` which tells us if the button was clicked down, `GLUT_DOWN` or was just released, `GLUT_UP`. Normally, we will want to perform an action when `state==GLUT_UP`.
- `int x`, `int y` are the X and Y mouse pointer position when the event was triggered.

These cases can be sorted out in `mouse_click()`, for example:

```
void mouse_click(int button, int state, int x, int y)
{
    if (button==GLUT_LEFT_BUTTON) // left button event
    {
        if (state==GLUT_UP) // was released
        {
            g_cursor_x = x; // save x and y positions
            g_cursor_y = y;

        }
    }
    glutPostRedisplay(); // cue a redraw
}
```

Similarly, the `glutPassiveMotionFunc()` function (in `main()` can install a handler which will be called every time the mouse is swept over our graphics window, and it will always tell you the current mouse pointer position. I've called this `mouse_motion()`:

```
void mouse_motion(int x, int y)
{
```

```
    std::cerr << "\t mouse is at (" << x << ", "
                                   << y << ")" << std::
                                      endl;
    //...
}
```

4.4 Exercises

1. Compile and run the program `mouse` and click the left mouse button in the window and see what happens. If you think you understand what and how it is doing things, modify it to move the triangle to the mouse click position.

2. Now use the `mouse_motion()` handler to continuously update the global cursor position and on the screen, draw a small cross to show the cursor as it moves.

3. Modify the program to rotate the tip of the triangle toward the position of the cursor. Hint: you can find the angle to rotate by using `atan2()`, e.g.

```
const float RAD_TO_DEG = 180.0/(atan(1)*4);
float angle_to_spin = RAD_TO_DEG * atan2(y -
   g_cursor_y, x - g_cursor_x);
```

4.5 Program `idle`

GLUT allows us to install a method which will be called when the user is idle, i.e. there are no events being generated by the user, and this allows your graphics program to perform **background** tasks, such as animating the graphics. It is very simple to set up and use. When ever we want to set up the callback, we make a call to `glutIdleFunc()` with the pointer to our handler function:

```
glutIdleFunc(idle);
```

If at any time we want to de-install the function, we can pass `NULL` to the callback function:

```
glutIdleFunc(NULL);
```

Look at the code to `idle.cpp`. In that I've used `glutIdleFunc()` to install and de-install my handler `idle()` each time the `'a'` key is pressed on the keyboard.

Note that to ensure that the idle handler is not being called when the window is not visible e.g. when minimzed, it is good practice to deinstall it within another callback function which detects visibility events, `glutVisibilityFunc()`:

```
void visibility(int vis)
{
    if (vis==GLUT_VISIBLE)
       glutIdleFunc(idle);
    else
       glutIdleFunc(NULL);
}
```

4.6 Exercises

1. Modify `idle.cpp` add a statement to `idle()` to increment the angle variable, `g_angle`. Compile it and check that `'a'` will start and stop the rotation of the triangle.

2. Make a new program called `bounce.cpp` to bounce a square around the edges of the screen using a GLUT idle handler.

4.7 Display Lists

Display lists are a way to pre-assemble and store OpenGL commands so that when the time comes to draw them, OpenGL can execute them efficiently. It is an ideal way to render objects which which will not change, other than by transformations

due to the GL_MODELVIEW or GL_PROJECTION matrices. Display lists can also be used to pre-compile state and attribute changes, like colours.

In the program idle.cpp, I've used a display list to pre-compile the drawing of the triangle. Look at the method, make_triangle(), which is based on the previous draw_triangle().

```
unsigned int make_triangle()
{
    static float vertex[3][2] =
        {
            {-1.0f, -1.0f},
            {1.0f, -1.0f},
            {0.0f, 1.0f}
        };

    // request a single display list handle
    unsigned int handle = glGenLists(1);

    glNewList(handle, GL_COMPILE);

    glBegin(GL_LINE_LOOP);
        for (size_t i=0;i<3;i++)
            glVertex2fv(vertex[i]);
    glEnd();
    glEndList();

    return handle;
}
```

As before, we use the local static array to make a template triangle, but this time we have used the glGenLists() function to get GL to give us a **handle** (an unsigned int). We then use this handle to with glNewList() to start the process of compiling geometry we want to draw. Note that at this point, nothing is being drawn. We are giving GL a series of commands which are being recorded until it sees a call to glEndList(). We then return the handle from this method so that it can be kept by our application. If we make more calls to glGenList() we will receive other handle names, and we can release a given handle at anytime using glDeleteLists().

In idle.cpp, we make the triangle display list in the init() function, so that it is ready to be used at the next display event. We *use* the display list, in display(), replacing the draw_triangle() call by:

```
    // execute a pre-compiled display list
    glCallList(g_the_triangle);
```

At this point, OpenGL knows to run the set of pre-compiled commands we used before. Display lists are important to know and use, so you may want to look at the Red Book information about them.

4.8 Drawing Text: Programs `bitmap_text` and `stoke_text`

Writing text into a graphics window is problematic and not a satisfactory experience. This is because, OpenGL does not natively support things like antialiased, scaleable fonts. GLUT does however have some fairly crude bitmap and vector fonts which we can use.

To use the bitmap fonts, we can combine our knowledge of display lists to make things a little simpler. Look at program `bitmap_text.cpp`. The font characters are first compiled as a set of display lists starting at a given **handle base**.

```
unsigned int make_bitmap_text()
{
    unsigned int handle_base = glGenLists(256);

    for (size_t i=0;i<256;i++)
    {
        // a new list for each character
        glNewList(handle_base+i, GL_COMPILE);
            glutBitmapCharacter(
               GLUT_BITMAP_TIMES_ROMAN_10, i);
        glEndList();
    }
    return handle_base;
}
```

Then to use them, `draw_text()` takes a C-string (`char*` array) and uses the character values to automatically call the correct pre-compiled list:

```
void draw_text(const char* text)
{
    glListBase(g_bitmap_text_handle);
    glCallLists(int(strlen(text)), GL_UNSIGNED_BYTE,
       text);
}
```

If you compile and run `text.cpp` you will see the results are rather disappointing. The problem is that because each character is a bitmap, although you can set its colour, you can't scale it or rotate it. And you have to use the `glRasterPos` to move the text.

An alternative is to use **vectorised fonts** or **stroke fonts**, which because they are drawing with lines, behave like any other geometry.

Look now at `stroke_text.cpp`. I've gone back to just drawing the text directly (instead of using display lists, but you could if you wanted). This time, `draw_text()` makes calls to `glutStrokeCharacter()` for each character in the given string `text`:

```
void draw_text(const char* text)
{
    size_t len = strlen(text);
    for (size_t i=0;i<len;i++)
        glutStrokeCharacter(GLUT_STROKE_ROMAN, text[i]);
}
```

Now, because the GLUT function actually draws lines for each character in the chosen font, we can just use this as if we were drawing geometry. We have to be careful to scale the font so it is the correct size. I've empirically chosen `0.5f` for the scale in `display()`

```
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glTranslatef(500.0f, 500.0f, 0.0f);
        glScalef(0.5f, 0.5f, 1.0f);
        draw_text("Hello, World!");
    glPopMatrix();
```

4.9 Exercises

1. Write a program that makes two or more display lists of geometry of your choosing and use it to render multiple versions of the object on a display. Compare the performance with drawing your objects directly in the display callback.

2. Modify the `draw_text()` method in `stroke_text.cpp` to allow fonts to be centred. You can find out the size of a character by using `glutStrokeCharacterWidth()`. You will need to get the size (in pixel units) of the whole string before you can centre it, perhaps like this:

```
    size_t len = strlen(text);
    int total_width = 0;
    for (size_t i=0;i<len;i++)
        total_width += glutStrokeWidth(
           GLUT_STROKE_ROMAN, text[i]);

    // next use glTranslatef() to centre the string
       prior to drawing it
```

5 Lab 4 - 3D Graphics, Cameras and Viewing

Computer graphics is all about representing things and visualising them as realistically as we see them in the real world. Each of our two eyes works like a **pin-hole camera**, where light reflected form objects in the world goes in straight lines through a **pupil** (the lens) and is focussed on our **retina**. If you have ever made a pin-hole camera with a box and a piece of tracing paper as the imager, you might remember that the pictures are **projected** up-side down. This is because the **projection plane** or **view plane** is behind the pin hole, the **centre of projection**: all straight lines from points in the scene project through the pin-hole, onto the view plane.

It may be obvious, but remember that although the world is three dimensional (3D), our view of it is two dimensional, 2D. The imager (or retina) is a 2D surface.

If you are an artist and hold up an empty picture frame in front of you, you can see a window on the world, and light from objects being **projected** on to an imaginary **view plane** in *front of* the centre of projection. This second example is exactly how you should think about how computer graphics calculates a 2D projection of objects the graphics application models. In fact, it's all a process of calculating exactly where **vertices** (3D points) project onto a given 2D surface (the **view plane**).

OpenGL is set up to perform projection calculations efficiently using its viewing pipeline. In the earlier labs, we learned how to draw 2D World Coordinate primitives (points, lines, polygons, etc.) onto a 2D world. In this lab, we will learn

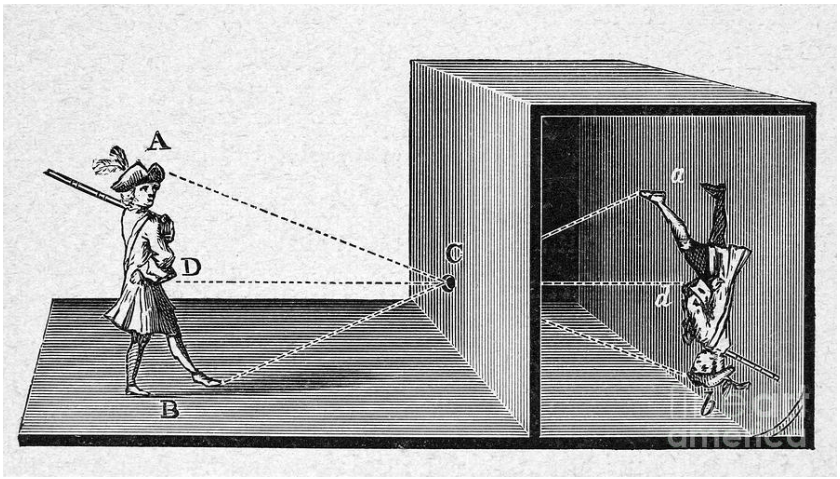


Figure 4: A pin-hole camera: view plane behind centre of projection. This sketch depicts a **camera obscura**, which means hidden room and was famously used by the Dutch artist Vermeer. That's where we get the name **camera** from.

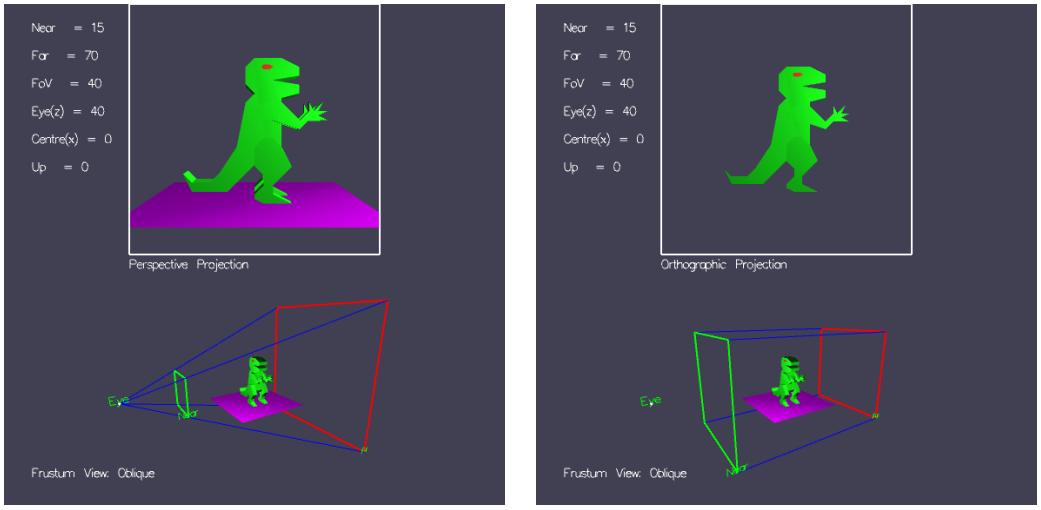


Figure 5: Perspective and Orthographic Projection. Note that the **view volume** changes shapes. The projection plane in both cases is the FRONT plane of the view volume. Note also how the view volume is a **frustum** (truncated pyramid) in perspective projection and a **right parallelepiped.**

the fundamentals of setting up the GL "pin-hole" camera model and getting it to project 3D primitives (3D vertices, lines in three space) onto a view plane. Specifically, we will learn about:

- Orthographic and Perspective projection in 3D
- Specifying the position and direction (orientation) of a camera
- Drawing 3D geometry: wireframe models
- Moving our camera around the scene

5.1 Perspective versus Orthographic Projection

We take for granted that when you perform projection, there is the effect of **perspective**; things further away look smaller than things closer to you. We use this fact to estimate distances, and work out what is in front of what. There are other cues too, like **parallax**, **shading**, **shadows**, **texture**, **depth of focus**, **fog** and so on. In computer graphics, these cues can be simulated, but for now, we will focus on simple **wireframe** modelling, which is often used in computer aided design (CAD) modelling.

Before we look at perspective projection, we will use **parallel** or **orthographic projection**. This has the property that all projection lines run parallel onto the

view plane. This sounds dumb, but is it roughly what happens when we look at objects close up and is used in for example 3D CAD drawings and medical image visualisation because it allows us to take measurements of sizes off the image: there is no size distortion despite the projection.

5.2 Program `ortho3d`

This program demonstrate how to set up the OpenGL camera and use orthographic or parallel projection to map 3D vertices onto the view plane.

5.2.1 Cameras

The camera set up is performed by `gluLookAt()` which takes the **eye position** (as X, Y, Z World Coordinates), and what is called the **reference point** of the camera, i.e. where the camera is looking, again in X, Y and Z coordinates. Taken together, the eye and the reference tell us where the camera is pointing. Then we have to say which way the camera is up by giving a nominal **up vector**, which is usually the Y-axis, i.e. `(0, 1, 0)`. In our 3D world coordinate system, the X axis points to the right, Y-axis points up and Z-axis is coming *out of the screen*, towards us.

Note that `gluLookAt()` can be applied to either the projection matrix or the model matrix. Either way it is concatenated into the pipeline, however, applying it to the `GL_MODELVIEW` matrix is something we might do if we intend to move the camera around. In `ortho3d` you will find the `gluLookAt()` in the display handler:

```
// position and orient camera
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(1, 1, 2, // eye position
          0, 0, 0, // reference point
          0, 1, 0  // up vector
    );
```

5.2.2 Orthographic or Parallel Projection

Orthographic projection is specified in a round about way, by specifying the **view volume**. Remember that in 2D we use `gluOrtho2D` to specify the bounds of our **window** on the world by giving the LEFT, RIGHT, BOTTOM and TOP edges of the rectangular window? Well, in 3D, we use `glOrtho()` (GL not GLU!) to specify a box with sides LEFT, RIGHT, BOTTOM, TOP as before but also a NEAR and FAR distance form the **eye position**. In the `init()` method you will see:

```
// set orthographic viewing
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// specify a projection with
```

Figure 6: The Utah Teapot

```
//    this view volume, centred on origin
// takes LEFT, RIGHT, BOTTOM, TOP, NEAR and FAR
glOrtho(-2.0, 2.0, -2.0, 2.0, -4.0, 4.0);
```

This does two things: one, it says to OpenGL to **clip** or exclude all geometry which lies outside this box (**right parallelepiped** if you really want to know); and two, to use **parallel projection**. Note that in orthographic projection, it doesn't matter *where* the view plane is in relation to the eye, the projection looks the same regardless[8]. Note that `glOrtho()` sets up a matrix which is be applied to the `GL_PROJECTION` state matrix.

5.2.3 Drawing 3D Geometry

Having set up a camera, we need to draw 3D geometry. This is easy, because we just specify vertices in 3D World Coordinates using, for example, `glVertex3f()`, and decide how they are to be assembled by giving an argument to `glBegin()`, e.g. `GL_POINTS` to draw 3D points, `GL_LINES` to draw lines and `GL_TRIANGLES` for triangles, etc.

It is quite tedious to make 3D models from scratch for testing, so GLUT has a number of useful 3D models built in, including the famous UTAH teapot. Here are some functions to try in your display handler: `glutWireCube()`, `glutWireSphere()`, `glutWireTorus()` and `glutWireTeapot()`. Use `man` to find out their arguments etc. You many need to use `glScalef()` to make them bigger or smaller.

[8]however, geometry in front of the NEAR plane is clipped

5.3 Exercises

1. Change the eye position in the call to `gluLookAt()`, does it do what you expect?

2. Add a special key handler to trap events from the arrow keys to move the eye position in X and Y.

3. Add a keyboard handler to move the position of the NEAR plane back and forth. You will need to call `glOrtho()` on the correct matrix each time the NEAR plane position is changed before posting a redisplay. Can you make the model disappear?

5.4 Program `perspective`

This program demonstrates how to do perspective projection. It uses the same camera set up as in `ortho3d.cpp` but uses `gluPerspective()` to set up the projection so that we can view objects centred at the origin.

5.4.1 Perspective Projection

To do perspective projection, there is a helper function to multiply the `GL_PROJECTION` matrix by the correct transformation called `glFrustum()`, which also takes the same arguments as `glOrtho()` but this time makes the projection a perspective one.

It's called `glFrustum()` because the view volume created by perspective projection is like a truncated square based pyramid – a **frustum**. This tapering of the volume near the viewer is caused of course by the projection lines converging at the eye position. It is important to note that the NEAR plane becomes the **view plane** in perspective projection. This also has the curious effect that as you move this plane nearer to the eye, you get more of the world projected onto the window (your **field of view** or FOV becomes larger). Conversely, if you move the view plane further from your eye, you will see less and less of the world. You have to be careful to make sure that NEAR $<$ FAR else nothing will get projected as the view volume will be completely collapsed.

`glFrustrum()` is quite difficult to use because of the perspective effect. Therefore, GLU provides a helper called `gluPerspective()` that uses different parameters. Instead of the LEFT, RIGHT, TOP, BOTTON, NEAR and FAR parameters that `glFrustum()` uses, it replaces those with a field of view (FOV) and aspect ratio and NEAR and FAR.

```
gluPerspective(40.0, 1.0f, 1.0, 5.0);
```

This is much easier to deal with and in most cases, a FOV with about 40 degrees and an aspect ratio of 1.0 is satisfactory. That just leaves NEAR and FAR. Well, FAR can be set be some distance from the objects of interest and NEAR is the position of the view plane (in world units) away from your eye position.

5.4.2 Reshape Callback

There is another callback handler to detect when the users **resizes** the window. This is notified to the user method installed by `glutReshapeFunc()`. In the `main()` of `perspective.cpp`, I make the call:

```
glutReshapeFunc(reshape);
```

The `reshape()` function is then given the dimensions of the resized window, which can be used to reset the GL **viewport**. We've not discussed viewports, but this sets up the final mapping which allows OpenGL map window coordinates to **normalised device coordinates**. All we need to know is that any reshape handler must make this call:

```
glViewport(0, 0, w, h);
```

After this, the perspective projection is set using `gluPerspective()`. So, like in `ortho3d.cpp`, we are setting up the projection matrix in `reshape()` and not in `init()`.

5.4.3 Solid Geometry

In this program, I've already made a keyboard handler to trap the `'t'` key to display the Utah teapot, rendered with lighting. If you look in `init()` I've made a call to `init_lights()` and `init_material()`, and have been deliberately hidden away as we don't need to know about these things yet!

Having initialised the lights and material properties, however, you can display shaded, solid geometry, like the Utah teapot, `glutSolidTeapot()`. Note that to draw lines etc., we have to **turn off** the lighting, with `glDisable(GL_LIGHTING)` and re-enable it afterwards, with `glEnable(GL_LIGHTING)`:

```
if (!g_draw_teapot)
{
    glDisable(GL_LIGHTING);
    glutWireCube(0.5);
    glEnable(GL_LIGHTING);
}
else
    glutSolidTeapot(0.4);
```

5.5 Exercises

1. Compile and test the program `perspective`. Try chaining the eye Z value with the fast-keys, `'z'/'Z'`. What do you notice? Why is the object appearing to spin?

2. Build a triangle in three space, and present the vertices in counter-clockwise order. Then test it out with and without the lighting enabled.

3. Try other solid geometry objects, such as `glutSolidCone`, `glutSolidTorus` and `glutSolidDodecahedron`.

6 Lab 5 - Model Transformations

This lab is all about using 3D transformations, particularly translations and rotations. We've already used the functions `glTranslatef()` and `glRotatef()` which build matrices that perform the respective transformations. Also, we've seen that OpenGL maintains two matrix stacks which apply to different parts of the viewing pipeline. Recall that to apply a transformation to the vertices of our objects, we need to use the `GL_MODELVIEW` stack; whereas, to set up the projection type (orthographic and perspective), we use the `GL_PROJECTION` matrix stack:

```
glMatrixMode(GL_MODELVIEW); // do something to our
   model
glPushMatrix(); // push down the current matrix{\bf
  glRotatef(30.0, 1, 0, 0); // rotate by 30 degree
     about the X axis
  draw_something();
glPopMatrix(); // undo the effect of the rotation
```

The `glPushMatrix()` and `glPopMatrix()` is a critical part of this process. What `glPushMatrix()` does is to copy and push whatever is the current model matrix (or projection matrix) down on the stack, and then any transformation we apply are multiplied in. When we then generate geometry (by drawing something or calling a display list), the vertices of this geometry feel the effect of our transformation, such as the rotation in the above example. Then after we are done, we can discard our transformation by using `glPopMatrix()`, which then restores the previous matrix.

This push-pop bracketing of transformation allows us to **only** affect certain geometry, or if we nest the push-pop operations, then we can have a hierarchy of related objects whose positions, orientation and sizes are relative to each other. For example, imagine I want to draw a robot arm which consists of several parts which are connected together but allowed to move. There might be a shoulder joint, an upper arm, a lower arm, and a hand. Now, because these things are connected, if I more the upper arm around the shoulder, all the other parts will move with it. Using OpenGL matrix stack, we can code it like this:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();

  // move to where you want the shoulder
  // draw the shoulder
  // move to the top of the shoulder (*)
```

```
glPushMatrix();

  // apply rotation to move lower arm (*)
  // draw the upper arm
  // move to the end of the upper arm (*)

  glPushMatrix();

    // apply rotation of the elbow joint (*)
    // draw the lower arm
    // move the end of the lower arm (*)

    // connect the hand
    glPushMatrix();

      // draw the hand and so on

    glPopMatrix(); // undo motion of hand
  glPopMatrix(); // undo effect of lower arm
     rotation
glPopMatrix(); // undo effect of upper arm motion
glPopMatrix(); // undo effect of shoulder set up
```

What we are doing is both connecting together the parts of our robot arm, and allowing the control of the joints. In this lab, we will look at one program which implements a simple robot arm and the exercises will be aimed at making it work. But before we do that, a word about rotations.

6.1 Rotations in 3D

We've already seen the use of a rotation when we used it to rotate a square in 2D, in program `lab-2/rotate.cpp`. There we were rotating in the **X-Y plane**, so the way it was used was:

```
glRotatef(angle, 0, 0, 1); // rotate about the Z axis
```

Because in OpenGL the Z-axis is coming out of the screen, 2D rotations are always around it. Recall that a 2D rotation is about a **pivot point**, such as the origin, but a 3D rotation must be about **an axis**. Here, the effect is the same because a 3D rotation about the Z-axis is the same as 2D rotation about the origin. So the axis is important and we normally chose the **principal axes**:

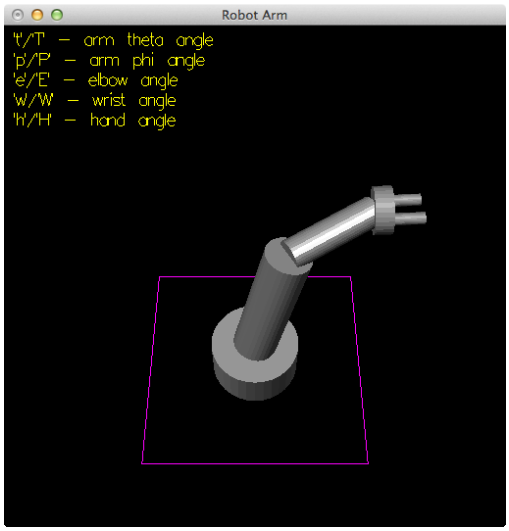


Figure 7: When it's working it should look like this.

X-axis, Y-axis and Z-axis. Note also that the `angle` of rotation is specified in degrees and causes an **anti-clockwise** rotation about the chosen axis.

6.2 Order of Matrix Stack Evaluations

In vector algebra, when we write something like:

$$\mathbf{y} = BA\mathbf{x}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors and A and B are matrices, the order of evaluation is **right-to-left**: apply transformation A to $\mathbf{x}$ and *then* apply matrix B.

In OpenGL, if we use the matrix stack, then any new matrix is **right** multiplied by the one that is already there, so actually, if the old matrix on the stack was M and the new one is T then we get $M \times T$ replacing the top. This still does what we expect, in that any subsequent geometry has the combined transformations applied to it.

Remember that if we use the functions `glTranslatef()`, `glRotatef()` or `glScalef()` we are creating matrices and multiplying them with the matrix on top of the stack.

6.3 Program `robot-arm`

Let's look at my program `robot-arm.cpp`. It's quite long, but don't worry we'll break it down.

6.3.1 The Robot Arm Parts

The `main()` function should be familiar and there is nothing new there. In `init()`, there is a call to `make_robot_arm()` which is at top of the file. This method makes the parts of the robot arm and compiles them into display lists with handles having a list base of `g_robot` plus one of a number of constants: `SHOULDER`, `ARM`, `FOREARM`, and `HAND`. Basically, this means that we can draw the individual parts efficiently by making the calls like:

```
glCallList(g_robot + SHOULDER); // draw shoulder
glCallList(g_robot + HAND); // draw hand
```

Now the point about `make_robot_arm()` is that all the parts are the correct size but in graphics parlance, defined in **model** coordinates. This means that to use them, you need to translate, rotate them to attach them together.//

Why have I done this? Is it because I am cruel? No, this is how we must do it to allow us to **articulate** the various parts of the arm. If I had created a single object which was the robot arm, then it would be rigid and only move as one object.

6.3.2 Building the Arm

Now we know how to draw named parts of the arm (using `glCallList()`), we have to build it. I've started doing this in `display()` and marked in comments where the function calls should go and hinted at what they should contain.

Can you see the nested structure of this code and as was mentioned in the introduction to the lab, we use the push-pop bracketing to build our arm? Note that I've provided some constant variables which give you the important dimensions of the parts.

```
const float g_shoulder_height = 0.2f;
const float g_arm_length = 0.6f;
const float g_forearm_length = 0.4f;
const float g_hand_thickness = 0.075f;
```

I've also started making some variables for the joint angles that you need to add fast-keys for:

```
// rotation angles of arm
float g_theta = 0.0f;
float g_phi = 0.0f;
// float g_elbow_angle = 0.0f;
// float g_wrist_angle = 0.0f;
// float g_hand_angle = 0.0f;
```

6.3.3 Articulating the Arm

To articulate the arm, you need allow the joints to rotate. More specifically you want to apply a rotation at each level of the model hierarchy, just before you start building the next part of the arm. Actually, in the pseudo code in the introduction, I've marked with a `(*)` where in the `display()` method you need to add the transformations. And, you can see how I've done this for the shoulder joint:

```
// ...
   // draw shoulder
   glCallList(g_robot + SHOULDER);

   // move to top of shoulder using
      g_shoulder_height
   // glTranslatef(...)

   glPushMatrix();

       // get ready to apply the arm rotations
       glRotatef(g_phi, 0, 1, 0);
       glRotatef(g_theta, 1, 0, 0);

       // draw the arm...
```

Following the template should allow you to complete the exercises. Good luck!

6.4 Exercises

1. Complete the attachment code in `display()` to build a complete arm. It should stack together, one piece on top of the other.

2. Try out the arm rotations which I've written: this should articulate the arm using the `'t'/'T'` and `'p'/'P'` fast-keys. Look at the `glRotatef()` calls which do this. What would happen if the order of the two rotations are swapped?

3. Add a rotation and control to change the angle between the lower and upper arms, the elbow joint. Use the fast-key `'e'/'E'` for this angle.

4. Add a rotation to rotate the wrist joint side-to-side and to spin the hand. Use the fast-key `'w'/'W'` and `'h'/'H'` for these angles as hinted at in the code.

7 Lab 6 - Lights and Materials

This lab is all about how to add realism to our 3D rendered geometry using the graphics lighting models which are built into OpenGL. Actually, we've already used lighting in a couple of our programs where we showed **solid geometry**, such as when we used `glutSolidTeapot()`. For example, in program `lab-4/perspective.cpp` with the `'t'` fast-key, and for the lighting of the robot arm in program `lab-4/robot-arm.cpp`. In both programs, I deliberately hid away the setting up of lights and materials in the header file, `lights_material.h`. Now's the time to open up that box and see what's what.

In real life, we take for granted the effects of light around us. We see the world because surfaces around us reflect light in the scene. Light rays, whether they are from the sun or artificial lights, such as light bulbs, are reflected off surfaces around us and the effects are mainly determined by three things: where the light is coming from, how the surface is angled relative to us, and the **surface properties**. For example, if the light is behind us when we are reading a book, it will be well lit. If a surface is shiny, like a silver pot, we will see bright spots where the the light sources are mirrored in it.

In this lab, we will learn specifically about the following things:

- how the effect of light on a surface is approximated by the **Phong model**
- the concepts of **surface normals** and **lighting vectors**
- surface properties: **diffusivity** and **specularity**
- how to position and move **point lights** in OpenGL

7.1 Modelling Direct Lighting

In the natural world, surfaces reflect light from **primary** light sources, such as the sun's rays, or artificial lights, but also secondary light sources, such as other surfaces. In fact, light will continue to "bounce" around the scene until all photons escape or are absorbed. Of course, certain surfaces will **reflect** more light than others: white surfaces reflect much more than a dark ones. A shiny mirror will reflect almost all incident light.

This model is known as a **global** model or lighting because we are trying to consider *all* surfaces as secondary, tertiary etc., light sources. To model such interactions would be both complicated and time-consuming. OpenGL does not try to do this sort of lighting. OpenGL only models **direct** illumination, i.e. the amount of incident light which is reflected from a surface which goes to the viewer and enters the camera. Furthermore, it only uses **point light sources**:

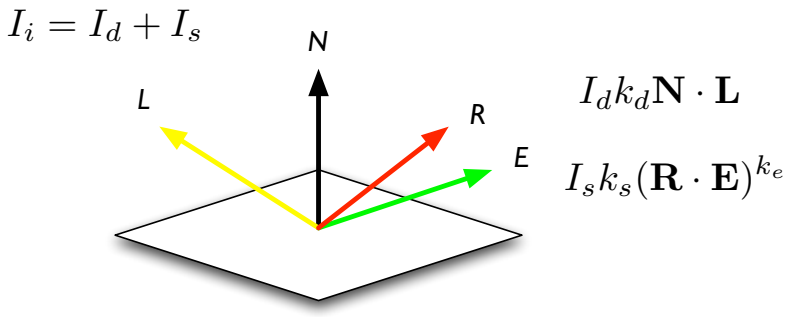


Figure 8: A local surface model shows the incident **light vector**, L, which is the direction from the surface centre to the light source; the surface's **normal vector**, N; the reflected light vector R, which is L mirrored in N; and the **eye vector**, E. Note all the vectors point away from the surface.

light sources that emit light in all direction equally.[9]. For the purposes of light calculations it is assumed that all light rays incident on the surface are parallel to each other (this is a good approximation if the light is far away, like the sun). Mathematically, it is the path from a light source (**a point in space**), to a flat surface and reflected toward the **eye position**. This is known as a **local** lighting model.

Although **local** lighting sounds like a cop-out, the effects it can generate are varied and convincing, despite the fact that the effect of light **scatter** around the scene is being ignored[10]. The key to understanding the local model used by OpenGL, is some basic 3D geometry and we need to know three quantities, which I will define here, and is shown diagrammatically in the figure 8:

1. **the light vector** this is the direction from the centre of the surface element to the light position. We can denote this vector by **L** and has unit length.

2. **the eye vector** this a the direction from the centre of the surface element to the eye position, denoted by **E**. It too has unit length.

3. **the surface normal vector** which is the perpendicular vector of the surface. Remember that the surface is flat, so its tilt is given by its **normal vector**, denoted by **N**.

OpenGL uses a lighting model which combines the three geometric parameters: **L**, **N**, and **E** and with two properties of a surface which we are allowed to change:

[9]actually, OpenGL does allow the range of angles to be restricted by a cone, which models a **spot light**

[10]ray-tracing, is a technique to model all light/surface interactions in a scene and uses multiple local models to accumulate the surface to surface reflections

1. **diffuse reflection coefficient** that models how **matt** the surface is: like paper, cloth, wood, plastic or rubber and some dull metals.

2. **specular reflection coefficient** that models how **shiny** the surface is: like silverware, metals, glass, varnished objects, crockery, etc.

A formula devised by Bui Tong Phong, known as the Phong Lighting Model, is used to combine these parameters like this:

$$I_r = I_i \left(k_d \mathbf{L}^T \mathbf{N} + k_s (\mathbf{R}^T \mathbf{E})^{k_e} \right),$$

where I_r is the reflected light energy, I_i is the incident light energy. k_d and k_s are the diffuse and specular properties of the surface. Here, $\mathbf{L}$ is the light vector. $\mathbf{R}$ is the mirror vector: the vector $\mathbf{L}$ reflected in the surface normal $\mathbf{N}$. Note that there is a third "surface" property which can be controlled and is known as the **specular exponent**, k_e, and it controls the size of the specular reflection.

The interesting thing about the Phong model is that: the diffuse contribution to the reflected energy does **not** depend on the eye position[11], whereas, conversely, the specular part entirely does depend on it. This means that matt surfaces look the same, wherever you stand, but shiny ones change as you move. Does that concur with your experience of the world?

Now the Phong Model goes further than this and adds to this an **ambient** lighting term, which is there to mop up all the scatter which is not being modelled and can be written as $k_a I_a$, which is an ambient coefficient times an ambient light energy in the scene, I_a. Generally, this term is kept small relative to the diffuse and specular parts.

7.2 Smooth or Interpolated Shading

OpenGL shading calculations, by default, are done on a per-vertex basis and when it comes to polygons, the Phong lighting calculation is performed with the **last** vertex of a polygon. As a consequence, each polygon will have a flat or **constant shading** value. To approximate smoothly shaded objects, we can either increase the density of the polygons, making each one smaller and smaller OR we can use **interpolated shading**. This is achieved by setting a flag using:

```
glShadeModel(GL_SMOOTH);
```

(the default mode is `GL_FLAT`). Smooth shading uses Henri Gouraud's method, which first calculates the surface normals adjoining each vertex, and averages them to find the vertex normals; the colour of each vertex is found according to the Phong reflection model, and colour values for a surface are interpolated between its vertices. This has the desired effect of smoothing across the edges.

[11] **Lambert's Law**

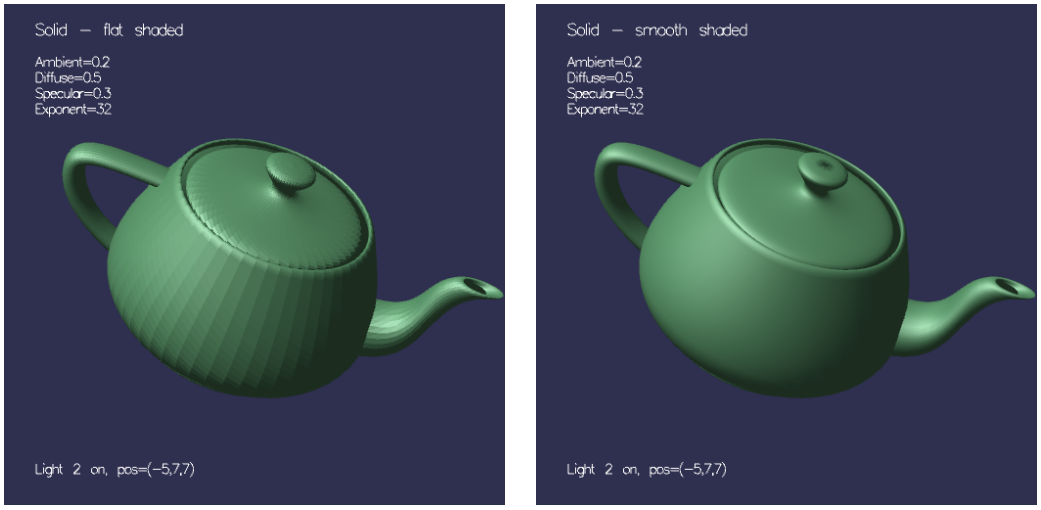


Figure 9: Flat and Smooth Shaded Geometry

7.3 Hidden Surface Removal by Depth Buffering

In human vision, one of the big depth cues is that solid surfaces hide things behind them. When we watch TV or look at a photograph, we get a sense of what is in front of what, even though the images are two dimensional because of **occlusion** boundaries. For computer graphics, this complicates the shading of polygons during rendering: we need to know which parts of which polygons are in front of others and make sure that hidden parts are not seen. The methods used to solve this is known as **hidden surface removal**.

When we send geometry through the OpenGL viewing pipeline, it has 3D coordinates, and we know where the camera is, so we know which vertex is closer and which farther away. If we paint a polygon with colours, pixels which are **rasterised**, i.e. the **fragments**, also have depths. In fact, for a flat surface, these depths change linearly in two directions, as measured from the eye position.

OpenGL uses a buffer to store fragment depth values as they are rasterised, called the **buffer** or the **z-buffer**. This buffer which is the same size as the RGB pixel buffer (i.e. the output window size), has floating point values in the range $[0, 1]$, where 0 is depths at the NEAR plane and 1, depths at the FAR plane. The hidden surface removal algorithm is then very simple:

1. Initialise the depth buffer to 1 (the FAR plane)

2. Rasterize each new fragment and tested its depth value with that stored in the z-buffer. (1) If it is less than the value, put the fragment colour into the RGB buffer, and overwrite the corresponding z-buffer value; (2) Else, discard the fragment.

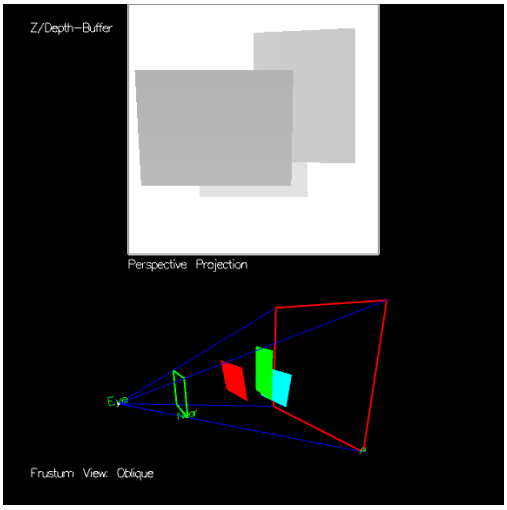


Figure 10: Hidden Surface Removal using Z-buffering.

Look at the examples in figure 10 where I've shown the z-buffer contents of rendering three rectangles, which occlude each other. You can see the resultant depth values: the darker the values, the closer is the fragment to the eye position.

Let's now look at an example program and we will see the Phong model in action.

7.4 Program `phong.cpp`

This program draws a Utah Teapot in 3D and positions a single point light source, with some ambient and diffuse energy. It then gives all the surface polygons the same **material** (or surface) properties. Let's look at it in detail.

`main()` should be familiar, but note that now there is an extra flag, `GLUT_DEPTH` added to `glutInitDisplayMode()`, this makes OpenGL set up and use its z-buffering for hidden surface removal. Next, `init()` sets up a point light source:

```
void init()
{
    // we can do this once if we don't intend to change
       the light properties
    // incident energy properties of a light
    float light_ambient[] = {0.1, 0.1, 0.1, 1.0};
    float light_diffuse[] = {0.5, 0.5, 0.5, 1.0};
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
```

```
    // fix the light position
    float light_position[] = {1.0, 1.0, 2.0, 0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    // enable lighting and turn on the light0
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // so that hidden surfaces are removed
    glEnable(GL_DEPTH_TEST);

    // mode of shading
    glShadeModel(GL_FLAT); // can be GL_FLAT, GL_SMOOTH
}
```

The function `glLightfv()` takes the name of a light `GL_LIGHT0` and a parameter we want to change, e.g. `GL_AMBIENT`, or `GL_POSITION` and then uses the floating point arrays `light_ambient`, `light_diffuse` and `light_position` (which all have **four** elements) to set the light. The first two set the brightness of the light and therefore control the I_a and I_d values in the Phong formula.

We also give the light a colour by using the three channel parameters. The position is just `X, Y, Z` in **world coordinates**. Next, `glEnable()` is used to turn on lighting globally and turn on a specified light, `GL_LIGHT0`, as we only have one.

Next, the z-buffer tests are enabled using the flag `GL_DEPTH_TEST`. We will see when we look at blending transparent geometry, that it is necessary to turn-off depth tests during compositing of fragments.

Once the light is set up, it will light any polygon which has surface properties set and use its surface normal to calculate Phong shading values. If our program just uses one global material, we could initialise it here, and all the rendered geometry would look the same. However, it is common to put material properties near where the geometry is being created or displayed, namely in the display callback. In this program, `display()` contains:

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    // position and orient camera
  gluLookAt(...); // as before

    // set the surface properties
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
```

```
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glutSolidTeapot(0.4);
    glutSwapBuffers();
}
```

Note first the use of the flag, `GL_DEPTH_BUFFER_BIT`, in the `glClear()` call. This is vital to reset the depth buffer at the same time as the colour buffer. If we don't do this, things will look very strange.

Next, `glMaterialfv()` is used to set the various properties of the **front** surfaces of any geometry. Note that we define the **front surface** as that facing the viewer: i.e. its surface normal is pointing **towards** the camera. Or, mathematically $\mathbf{N} \cdot \mathbf{E}$ is **positive**. The `man` page on `glMaterialfv` will tell you more about how we can set back and both face properties. Finally, the Utah Teapot is rendered.

Note that `glutSoldTeapot()` not only sends the vertices of the object (as **quads** in fact), but also works out and sends `glNormalfv()` commands, so that each surface element has the correct surface normal with it. If you draw your own polygons, or a surface mesh, you too must find out and give OpenGL the surface normal vectors (see the exercise below).

7.5 Exercises

1. Compile `phong.cpp` and run it. Add fast-key control variables for the diffuse, specular and specular exponent properties of the teapot to be changed. Just set all three colour channels to be the same.

2. Add a fast-key variable to change the ambient level of the light, keeping the surface material ambient property to be small, e.g. `0.1f`.

3. Add a fast-key toggle to flip between flat and smooth shading.

4. Move the light position to the back of the teapot by changing its Z value. Put this under a fast-key control and watch how the lighting changes. Note that you will need to re-issue the `glLightfv(GL_LIGHT0, GL_LIGHT_POSITION, light_position)` call in the keyboard handler before posting a redisplay.

5. Comment out the call to `glutSolidTeapot()` and now put in a single triangle of your choosing. To get the shading to work, you need to give OpenGL a surface normal using:

   ```
   glNormal3f(dx, dy, dz);
   ```

You can work this out using a **cross-product** calculation. I've made a function for you in `cross_product.h`, so please use it and test out your result. Add some more triangles and see if you believe the shading of them.

7.6 Program `disco.cpp`

With this program, we will animate the lights around an object and change their colours, and convert them into spot lights. We will also define a number of different material properties and allow the the user to change them.

To set up lights and materials, I've written two helper functions `set_light()` and `set_material()` which takes references to the structures `light_t` and `material_t`. There are defined at the top of the source file:

```
// properties struct
typedef struct materials_t
{
    float ambient[4];
    float diffuse[4];
    float specular[4];
    float shininess;
};

// light struct
typedef struct light_t
{
    size_t name;
    float ambient[4];
    float diffuse[4];
    float specular[4];
    float position[4];
};
```

We can then create variables of these structure types, such as the pre-defined lights and materials, and instantiate the member values:

```
// predefined materils to use
const materials_t brass = {
    {0.33f, 0.22f, 0.03f, 1.0f}, // ambient
    {0.78f, 0.57f, 0.11f, 1.0f}, // diffuse
    {0.99f, 0.91f, 0.81f, 1.0f}, // specular
    27.8f // specular exponent
};
```

```
// predefined light
light_t light_1 = {
    GL_LIGHT1,
    {0.0f, 0.0f, 0.0f, 1.0f},
    {0.0f, 0.0f, 1.0f, 1.0f},
    {1.0f, 1.0f, 1.0f, 1.0f},
    {0.5f, 0.75f, -0.5f, 1.0f}
}
```

When we want to use them, we just call the appropriate function with the named material or light, e.g.

```
set_light(light_0);
set_material(brass);
```

You can look at the code of these methods to see that they just make all the tedious `glMaterialfv` and `glLightfv()` calls. Having set all this up, three lights are set up in the `init()` function. Then, before we draw the teapot, we choose which material we want and make the call to `set_material()` (in the display handler). I've also put in code to mark the position of the lights using `GL_POINTS`:

```
    // show where the lights are applying any current
       rotation
    glDisable(GL_LIGHTING);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // perform any transformation of light positions
       here

        glPointSize(3.0f);
        glBegin(GL_POINTS);
            glVertex3fv(light_0.position);
            glVertex3fv(light_1.position);
            glVertex3fv(light_2.position);
        glEnd();

    glPopMatrix();
    glEnable(GL_LIGHTING);
```

I've left a comment where you should put any transformations of the lights by using, for example, `glRotatef()` and then re-setting the light position using

`glLightfv()`.

7.7 Exercises

1. Allow the user to swap between the material properties set up in the program.

2. Use the `glutIdleFunc()` callback installer to set up an `idle()` function to move the position of the given light sources in a circle above the tea-pot. Note that `glLightfv()` will be affected by the `GL_MODELVIEW` matrix stack, so you can use the `glRotatef()` around the Y axis to spin the lights.

3. Turn on and off the three lights in some random way as you spin them: for `GL_LIGHT0` use `glEnable(GL_LIGHT0)` and `glDisable(GL_LIGHT0)`.

4. Make the lights into spot lights by using the parameters: `GL_SPOT_DIRECTION`, and `GL_SPOT_CUTOFF`, e.g.

```
glLightf(GL_LIGHT1, GL_SPOT_DIRECTION,
   spot_direction); // direction
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45); //
   degrees
```

   (Hint: I've already put the code in the `set_light()` method for you!)

8 Lab 7 - Texturing

Texturing or **texture mapping** is a powerful way of adding realism to computer graphics without the need for significantly more vertices and pixels colours being specified. Texture mapping combines both the vertex (geometry) and fragment functions (colours and shading) of the graphics pipeline to allow image values (as pixel colours) to be mapped to fragments when a polygon is rasterised.

We have already seen that to add lighting effects to a solid object, in addition to the world coordinates of the vertices of the mesh, each polygon of the mesh, needs a **surface normal**. Typically, a display list is built like this:

```
glBegin(GL_POLYGON);
  glVertex3fv(vertex_1);
  glNormal3fv(normal_1);
  glVertex3fv(vertex_2);
  glNormal3fv(normal_2);
  glVertex3fv(vertex_3);
  glNormal3fv(normal_3);
  // .. and so on
glEnd();
```

This information is used to interpolate shading by calculating a Phong Lighting value at each vertex, and shading values are interpolated along the edges, and across the scan lines during **scan conversion**. In texture mapping, as well as normal information, we provide **texture coordinates** at each vertex, which specify a location on a image or **texture**. And instead of colouring fragments according to a lighting calculation, the rateriser produces fragments by **looking up** and interpolating values from corresponding **source texture coordinates** off the texture.

For each vertex of the display list, we associate a source texture coordinate which is a 2D location[12] using the function `glTexCoord2fv()`. For example, we would write:

```
glBegin(GL_POLYGON);
  glVertex3fv(vertex_1);
  glTexCoord2fv(tex_coord_1);
  glVertex3fv(vertex_2);
  glTexCoord2fv(tex_coord_2);
  glVertex3fv(vertex_3);
  glTexCoord2fv(tex_coord_3);
  // .. and so on
glEnd();
```

[12] actually, it can be 1D and 3D, but we're not going to look at that

Notice that **target coordinates** (the vertices in world coordinates) are 3D and **source coordinates** as 2D.

For example, look at figure 11, where I set up texture mapping to take image pixels from the Mandrill image (on the right), to corresponding fragments on the triangle (which is in 3D), on the left. If I move the geometry, the mapping is recalculated. Further more, we can choose any three points on the source texture. See the second example in the same figure.

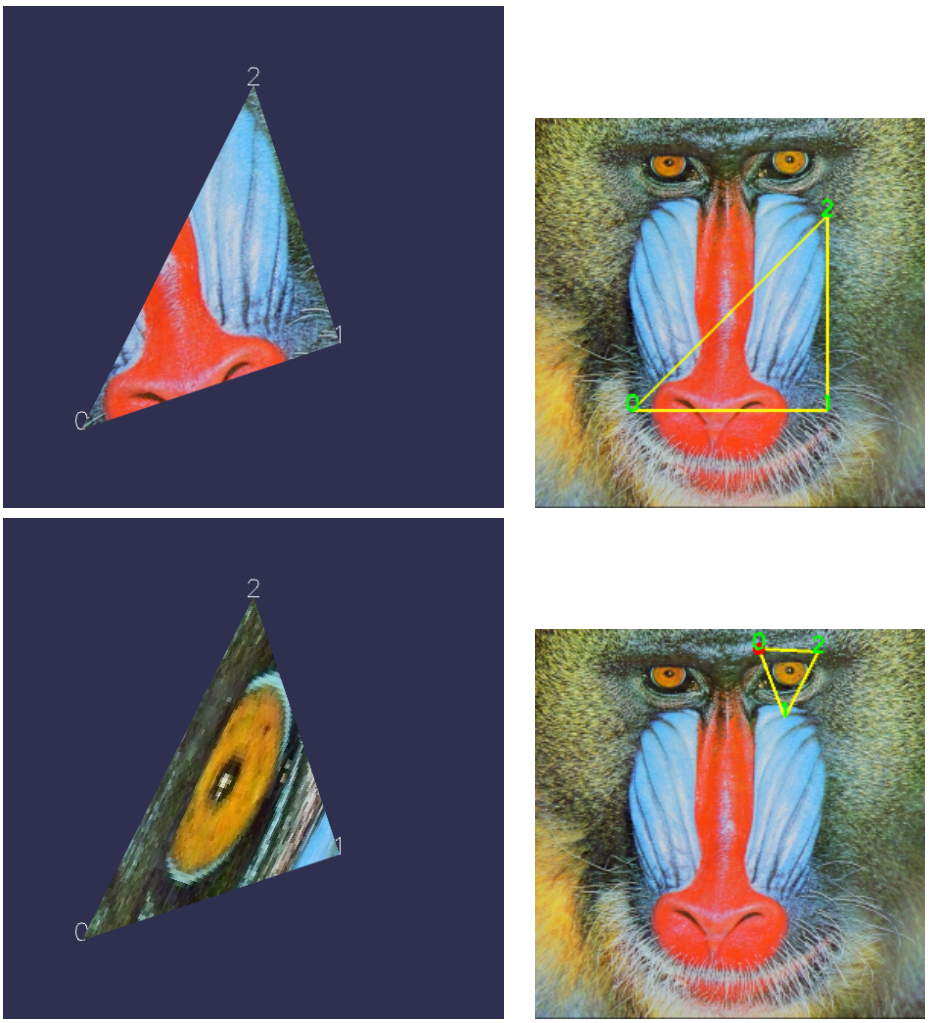


Figure 11: Texture mapping example showing the effect of moving the **source texture coordinates**. If the triangle on the left is moved, the mapping is maintained. Colours from the image are given to fragments and the triangle is "shaded".

The effect of texture mapping is visually appealing and can be used to model the appearance of natural surfaces: wood grain, carpet, cloth, grass. More dra-

matically, if we associate polygons with clouds of vertices which move together, **particle systems**, then we can produce the effects of fire, water, smoke and so on.

In this lab, we will first learn how to:

- read and install a texture map and specify source and target co-ordinates
- map a 3D object with multiple textures
- use a way to automatically generate source coordinates for standard objects
- we will experiment with environment mapping making objects appear reflective

8.1 Program `texture`

To set up to use texture mapping, we need to first load an image in some standard image format, like `BMP`, `GIF`, `JPEG`, or `PNG` etc. and then send its pixel values, stored as an image array, to OpenGL. I have provide a way to load `PNG` images in `png_load.h` and we use `glTexImage2D()` to generate image pixel values. Because it is expensive to keep sending all the image values to the graphics card every time they are need, it is essential to **bind** the texture as a named object (a bit like what we did with display lists) using `glGenTextures()` and `glBindTexture()`. Then, we enable texture mapping and present the texture co-ordinates at the same time as the target coordinates (world co-ordinates of our object), using `glEnable(GL_TEXTURE_2D)`.

Let's look at the important parts of `texture.cpp` in more detail. The texture is loaded and bound in `load_and_bind_texture()` which takes a filename and returns a texture handle. After the call to `png_load()` we have the important OpenGL parts:

```
    unsigned int tex_handle = 0;

    // request one texture handle
    glGenTextures(1, &tex_handle);

    // create a new texture object and bind it to
       tex_handle
    glBindTexture(GL_TEXTURE_2D, tex_handle);

  // storage and environment incantations...

    glTexImage2D(GL_TEXTURE_2D, 0,
```

```
                GL_RGB, width, height, 0,
                GL_RGB, GL_UNSIGNED_BYTE, image_buffer);

    // free the image buffer memory
    // ...

    return tex_handle;
```

The key statements are to use `glGenTextures()` to request a texture handle (a bit like using `glGenLists()`). Then the call to `glBindTexture()` tells GL that we want to do state changes on that texture handle: such as set its **interpolation** and repetition etc., but most important of all, send the image values to texture memory using `glTexImage2D()`. This last function declares what kind of image data is being presented in the `image_buffer`. In this case, we have a 2D image, of RGB values, where each value is a `char` (`GL_USIGNED_BYTE`). We then return the handle and keep it ready to use when we want to map image values from this texture. Note that normally, we would only bind image values that we are going to use. So if the texture coordinates are fixed, we would take a bounding box around it and bind those pixel values only to conserve texture memory. In our example, we are binding the whole image because I want you to choose arbitrary source texture coordinates on it.

Now we turn to the actual display part. In this program we set up a 2D orthogonal projection window, using `gluOrtho2D()`. This is actually done in the `reshape()` call back and if the user changes the size of the window, then we store the new window sizes `g_window_width` and `g_window_height`. We'll see why in a second.

Then, to map the texture we do the following in `display()`:

```
    // enable texturing
    glEnable(GL_TEXTURE_2D);

    // select which texure to render
    glBindTexture(GL_TEXTURE_2D, g_the_tex);

    // specify texture coordinates
    glBegin (GL_QUADS);
        glTexCoord2f (0.0f,0.0f); // lower left corner
        glVertex2i(0, 0);
        glTexCoord2f (1.0f, 0.0f); // lower right corner
        glVertex2i(g_window_width, 0);
        glTexCoord2f (1.0f, 1.0f); // upper right corner
        glVertex2i(g_window_width, g_window_height);
```

```
        glTexCoord2f (0.0f, 1.0f); // upper left corner
        glVertex2i(0, g_window_height);
    glEnd ();

    glDisable(GL_TEXTURE_2D);
```

The key parts are to inform OpenGL that we want to map textures with `glEnable(GL_TEXTURE_2D)`, then to select the texture we want to map with `glBindTexture()`[13], and specify the **target**/**source** coordinate pairs with `glBegin()`/`glEnd()` commands. Note that the **source coordinates** are in he range $[0, 1]$, i.e. they are normalised. As you can see though, here is where we use the `g_window_width` and `g_window_height` variables to stretch the texture across all the way to the corners of the window.

8.1.1 Loading Images from files

I've provided a header file, `png_load.h` which has a single method than can load `PNG` format images into an image array pointed to by `char*` pointer. It also returns the `width` and `height` of the image. We use `png_load()` in `load_and_bind_texture()` to load the named image and bind it to texture memory:

```
    char* image_buffer = NULL; // the image data
    int width = 0;
    int height = 0;

    // read in the PNG image data into image_buffer
    if (png_load(filename, &width, &height, &
       image_buffer)==0)
    {
        fprintf(stderr, "Failed to read image texture
           from %s\n", filename);
        exit(1);
    }

    // use the image values to create a texture map
    //...

    free(image_buffer); // free up the image memory
```

[13] I know that might be confusing, as we've already "bound" this texture already! However, this is to prevent another texture being bound in the interim and being drawn by accident.

8.2 Exercises

1. Compile `texture.cpp`, run it to see what it does. Experiment with changing the size of the window and watch how the texture stays stuck to the window edges.

2. Experiment with changing the source and target coordinates. Recall that the source coordinates are **normalised**. So if you want half way across the image you need `0.5f`. Normalised X is `pixel_x/image_width`, and normalised Y is `pixe_y/image_height`.

8.3 Program `camper-van`

`camper-left` `camper-front` `camper-right` `camper-back`

Figure 12: VW Camper Van: three views as source textures.

The aim or this program is to demonstrate texture mapping multiple textures to different parts of a 3D object. For this example, I've chosen some images of a classic old VW Camper Van, and your job is to complete the rendering of it onto the cuboid I've programmed.

8.4 Exercises

1. Compile program `camper-van.cpp` as it stands and see what it does. You can use the fast-key, `' '`, to spin it around the Y axis.

2. Complete the rendering of the other 4 sides of the van by selecting the appropriate source coordinates. To draw the images to the correct scale, you need to normalise the width and height – in this case, this work has been done for you, and you need only refer to the correct elements of `g_van_source_coords`. Otherwise, you could use some photo manipulation program, like GIMP, to find out the pixel coordinates for the quads and then normalise them. All three images are 547×411 in size.

8.5 Program `generate`

As we have seen, other than in the most simple objects, generating the source coordinates for textures and corresponding them to target (object) coordinates is tedious and error prone. In may cases, it is useful to approximate the mapping via an intermediate surfaces, or an **O-mapping**. This involves setting a formula for the texture coordinate generated based on some simple **bounding geometry**, like a sphere, cylinder or the faces of a cube.

In OpenGL, the function `glTexGeni()` can generate the `S` and `T` coordinates (i.e. those that lie in the plane of the image) automatically. We tell it which mode to use, such as `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR` or `GL_SPHERE_MAP`. The last mode is interesting because it presumes that texture is a sphere map: a flattening of the surface of a sphere into a disk (look at `images/sphere-map.png`, for example.

Program `generate.cpp` uses automatic texture coordinate generation to render a solid teapot. Look at the display callback and you will see the following just before the point where the `glutSolidTeapot()` call is made:

```
glBindTexture(GL_TEXTURE_2D, g_tex[g_curr]);

// g_gen_mode sets type of O-mapping
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
   g_gen_mode);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
   g_gen_mode);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_2D);

glTexEnvi(GL_TEXTURE_ENV,
   GL_TEXTURE_ENV_MODE, GL_DECAL);
glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_WRAP_T, GL_REPEAT);
```

The `glBindTexture()` selects the texture. Then we enable the generation of two axes of coordinate generation: `GL_S` and `GL_T`. I've made `gen_mode` a variable which you can use fast-keys to change. Then, we enable the texturing and the co-ordinate generation. Finally, before drawing the object, we ensure that the textures are "painted" on the object without modulation (this can be changed

to `GL_MODULATE` from `GL_DECAL`, if for example we want to combine shading with texturing). Lastly, we set how we want the textures to be wrapped around the object if the source coordinates fall off the end in the auto generation: `GL_REPEAT` causes the image values to be taken back to the start of the image.

8.6 Exercises

1. Compile and run `generate.cpp` and see what it does. Play around with the various options and look in the code to where and how they are being controlled. Note that there are no calls to `glTexCoord2f()` in the code!

2. Notice how with the right texture, such as `clouds.png`, the **sphere-map** mode causes the images to appear to be reflective. This is known as environment mapping and is used efficiently render shiny objects.

3. Look a the `man` page of `glTexEnvi` and `glTexParameteri` to understand better the various options being set.

4. Try other standard solid objects and other textures of your choosing.

9 Lab 8 - Blending

In this lab we will look at how **transparent** surfaces can be added to rendered scenes. OpenGL handles transparency by attributing fragments with an **opacity** value, usually denoted by the letter `A`. Together with the three colour channels, the opacity or `alpha` value makes a quartet of values. Indeed, the frame buffer has a separate pixel buffer, which can hold floating point opacity values in the range $[0, 1]$. Note that mathematically, transparency is the *complement* of opacity:

$$\text{Transparency} = 1.0 - \text{Opacity}$$

Note further that:

- OpenGL uses the term `ALPHA` and the letter `A` to refer to the opacity values
- an `ALPHA` value of 1.0 is a fully opaque fragment, whereas 0.0 is fully transparent.

We can issue commands to "colour" polygons with `RGBA` values, e.g.

```
glColor4f(0.0f, 1.0f, 0.0f, 0.5f); // half transparent
    green fragments will be generated
```

When we start a display pass, we would clear the frame buffer channels to the background colour and an opacity of `1.0f` which defines the background to be fully opaque.

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background is
    fully opaque
```

The rasteriser and fragment processor will then apply a blending operation to combine or **composite** the new colours and opacities with those that are already in the frame buffer. This process of **compositing** gets its name from what filmmakers used to do when layering one filmed scene on top of another to create special effects. This is precisely the same process, except in computer graphics we perform it digitally.

By default, blending is turned off and all fragment colours simply overwrite those that are already there, albeit using any depth test for hidden surface removal. To use blending, we have to enable it:

```
glEnable(GL_BLEND);
```

Then, from this point on or until it is disabled, all fragments are blended with the existing values in the frame buffer according to compositing or **blending rules**. The algorithm works like this: the value already in the `RGBA` buffer is called the

destination colour, and the new fragment value the **source colour** (here the term *colour* includes the opacity value).

The blending rule is defined by `glBlendFunc()` which specifies modal constants that say how the source and destination values should be combined. One rule which works for most situations is:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

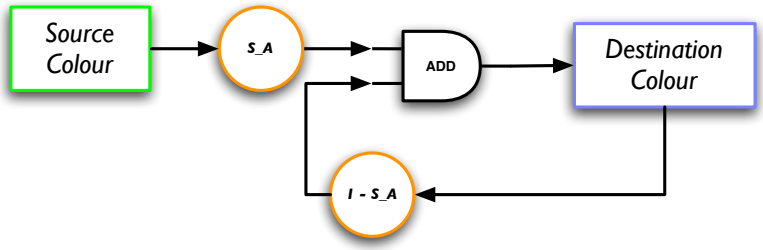


Figure 13: The blending operation with **source** and **destination** factors, `GL_SOURCE_ALPHA` and `GL_ONE_MINUS_SOURCE_ALPHA`.

The first argument defines the source blending factor and the second the destination blending factor. These factors are applied to all the 4-channels. So here, it would use the following formulae:

$$(R,G,B)_{D\text{new}} = A_S \times (R,G,B)_S + (1 - A_S) \times (R,G,B)_{D\text{old}}$$
$$A_{D\text{new}} = A_S \times A_S + (1 - A_S) \times A_{D\text{old}}$$

Here, I've denoted the destination with the subscript D and the source by S, also by new, I mean the new destination. Intuitively, the second argument factor $(1 - A_S)$ simply says that the more the incoming fragment is transparent, $A_S \to 0.0$, the more the existing colour mixes with the incoming. Note here, that it's not just the colours which get combined, it is also the incoming alpha values. This can be interpreted by understanding that each time we add more compositing layers to our result, the greater the opacity of the colour increases, until it becomes 1.0.

A note of caution however, the order of fragments presentation matters. This means that to get blending to work correctly each time we have to present fragments back to front: the ones farthest away first. However, the z-buffer hidden surface removal does not guarantee it (can you think of an example to prove this assertion?)

Another rule is `glBlendFunc(GL_SRC_ALPHA, GL_ONE)`, which make the destination factor 1.0. This second rule is curious and when used darkens the output and we must start with black in the background for it to work. It does have the advantage that order does not matter.

Blending is really important to produce natural looking graphics but it is difficult to get right. In this lab we will specifically:

- learn how to blend simple polygons to see the effect of the blending factors and opacity
- find out how to combine transparent and opaque geometry by changing the depth buffer write mode
- blend textures to produce interesting graphics effects and learn how to do billboarding
- implement the basics of a particle system

9.1 Program `blend`

Program `blend.cpp` tests out the blending capabilities of OpenGL. It aims to render a well lit, solid teapot as fully opaque, and then place a semi-transparent green quadrilateral in front of it. The fast-key, `' '`, is used to spin the scene, so we can see the effect of the blending.

`init()` This sets up the blending factors to allow for the **over composite operation**, where transparent geometry must be drawn in back-to-front order:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Then, in `display()` after the teapot is drawn, a single `GL_QUAD` is place in front of it, coloured green and given an opacity of 0.5:

```
// add in transparent geometry in front of teapot
glDisable(GL_LIGHTING);

glColor4f(0.0f, 1.0f, 0.0f, 0.5f);

glBegin(GL_QUADS);
    glVertex3f(-0.5f, -0.5f, 0.5f);
    glVertex3f(0.5f, -0.5f, 0.5f);
    glVertex3f(0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, 0.5f);
glEnd();

glEnable(GL_LIGHTING);
```

Note also the lighting is *disabled*, as the quad is not given any material properties (but it could be if required).

9.2 Exercises

1. Add a fast-key control `'o'/'O'` to control the opacity of the green quad. Increment the opacity in small amounts, e.g. 0.1 steps, and make sure it always stays in the range: $[0, 1]$.

2. Alter the material properties, which can be found at the top of the program as usual, of the teapot to give it transparency. You need to just change the 4th argument of the material properties. Just changing the ambient and diffuse properties should be enough to see a change. How does the blending work now? Is it satisfactory?

3. What happens if you use the a different blending factor combination, e.g. `GL_SRC_ALPHA`, `GL_ONE`?

9.3 Program `mixed`

When we mix opaque and transparent geometry, because of the way that OpenGL implements hidden surface removal, using **depth** buffers, it cannot ensure that all fragments that fall on a given pixel are composited in back-to-front order. This is problematic, as the only proper solution is to first sort all the geometry and/or fragments in depth order, then composite them. This would be expensive to do. Alternatively, the depth buffer can be extended to be a **priority queue**, sorted by depth. Again, this is possible but often not practical.

An approximate way to combine opaque and transparent geometry in OpenGL is to prevent translucent fragments from writing into the depth buffer, this then stops their depth values interfering with other transparent geometry. The trick is then to render all opaque objects **first** and then the transparent objects. To make the depth test `read-only`, we use

```
glDepthMask(GL_FALSE); // make depth buffer read-only
// draw transparent geometry
glDepthMask(GL_TRUE); // make it read-write again
```

9.4 Exercise

1. Compile program `mixed.cpp` and add in the `glDepthMask()` commands in the appropriate place. Confirm that the green and red quads are being blended correctly with the opaque teapot. Are they being blended correctly with each other?

9.5 Program `particles`

Particle systems are used in computer graphics to model natural phenomena like: fire, smoke, water, vapour, etc. A particle system is essentially a set of point masses whose kinematic behaviour has been modelled. Usually particles are given positions and motion vectors, but they may also have mass, have collisions and follow some rules. Each particle will be displayed in some way: the simplest thing to do is to show each particle as a point with colour. To produce more convincing effects, particles are usually associated with a quadrilateral and bound to a **texture**. The textures might be quite small, e.g. 64×64, and be given colour and transparency and a lifetime parameter. This allows complex phenomena such as fire and smoke to be modelled.

Program `particles.cpp` make a simple particle system consisting of a set of translucent textures. Each particle's parameters are held in a structure:

```
// particle information as struct
typedef struct particle_t
{
    unsigned int tex; // texture handle
    float colour[4]; // colour and transparency
    float position[3]; // world coordinate
    float direction[3]; // direction of travel
    float size; // in world dimensions
};
```

Each particle is associated with its own GL texture, `tex`, which is created and bound into texture memory when the particle is created. It also has a colour with transparency, a size (in world dimensions), and a position and motion vector. The aim is to create a set of particles at random positions and then animate their motions by moving them in a random direction. Note that the colour and transparency applies to the whole texture, and this is achieved by making the texture values all `ALPHA` channel values. The image values for the texture are not loaded from a file (although they could be), but created internally by the method `blob_image()`.

The texture image is made using a Gaussian function, which tails off in amplitude at the edges of the texture, so this has the effect of making the texture opaque in the middle and transparent at the edges. The texture image has 4-values per pixel:

```
    for (size_t i=0;i<size;i++)
    for (size_t j=0;j<size;j++,p+=4)
    {
        const float x = float(j) - float(size/2);
```

```
            const float y = float(i) - float(size/2);
            const float g = exp(-2*(x*x+y*y)/sd2); //
               gaussian value
            im[p+0] = 0; // red
            im[p+1] = 0; // green
            im[p+2] = 0; // blue
            im[p+3] = (unsigned char)(255 * g); // alpha
               value
        }
```

A set of particles is made by `make_particles()`, and each one is bound using `glTexImage2D()`. Note that for this particular application, I've used some special incantation to allow us to change the colour of the whole texture, and blend it correctly with other fragments using the lines:

```
        const float env_colour[4] = {0.0f, 0.0f, 0.0f, 1.0f
           }; // env colour
           // ...
           glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
              GL_BLEND);
           glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR,
               env_colour);
```

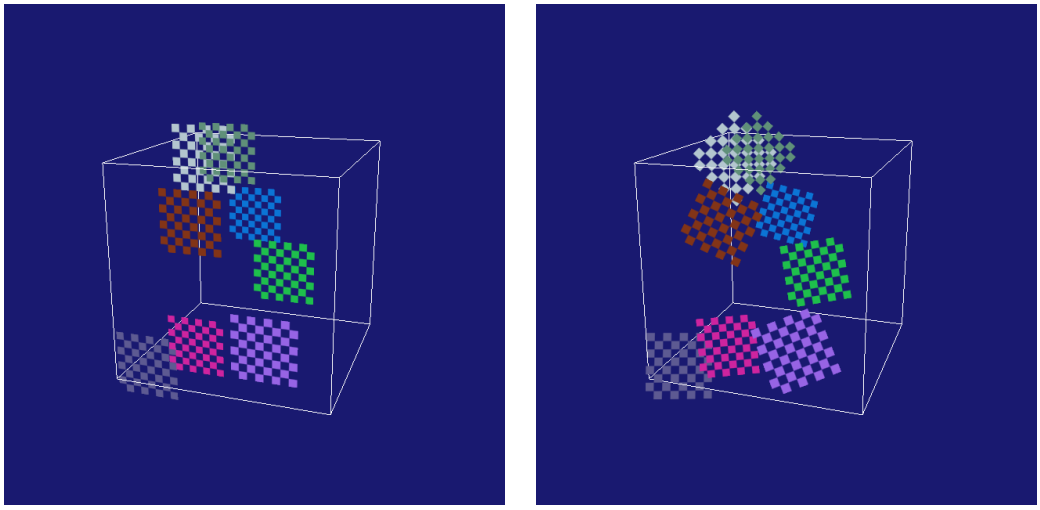
This is quite hard to explain, and it's not necessary for you to know exactly why, other than it works and is different from what you might do for most texture images[14]

9.5.1 Billboarding

When mapping 2D textures into a scene, the target coordiantes are usually the surface quadrilaterals of objects or parts of a mesh. This means that when the object is rotated or the viewpoint changed, the mapping continues to work as OpenGL will apply the model/view transformation to the texture target coordinates. However, when using textures in particle systems, it is desirable to keep texture target polygons facing the viewer: a process know as **billboarding**. All the rendered textures then lie **plane-parallel** to the view plane, positioned at the correct coordinates.

Billboarding requires the effect of the model/view transformation to be undone, while preserving any translation of the target coordinates. In our example, we

[14]there is more information about this in the Red Book in the section on Decal and Blend Texture Functions.



Not viewplane parallel *Billboarding*

Figure 14: Texture billboarding is when all textures are turned to face the viewer. In these figures, the textures are shown as chequer boards.

are using `gluLookAt()` to move the viewer to the `eye` position at `(1, 1, 3)`, and then applying a rotation about the Y-axis of angle, `g_spin`.

Since all the textures are posted with quads facing out, i.e. have normals `(0, 0, 1)` (the Z-axis), we can calculate a rotation which will turn them towards the current eye position. This is done by the method `rotate_towards_viewer()` which takes the texture position, the eye position and the plane normal (z-axis), and calculates the axis and angle of rotation which will perform the rotation. Note that the required rotation is different for each texture and and must be bracketed by `glPushMatrix()` and `glPopMatrix()` operations.

If you look at this method, it does the following:

1. Calculate the vector from the texture centre (`pos`) to the `eye` position which is simply: `v = eye - pos`, and then it normalises it (makes it unit length) using the `normalize()` method (found in the `vm.h` header file).

2. Next, the axis of rotation is calculated as the `cross product`: `normal` $\times$ `v`. Recall that the cross product always gives you a vector perpendicular to the two input vectors. The result is the axis we want rotate the texture normal around.

3. Finally, the angle of rotation is given by the angle between `v` and `normal` and is calculated by the arccosine of the inner product (or dot product) of `v` and `normal`: `angle = acos( v dot normal )`.

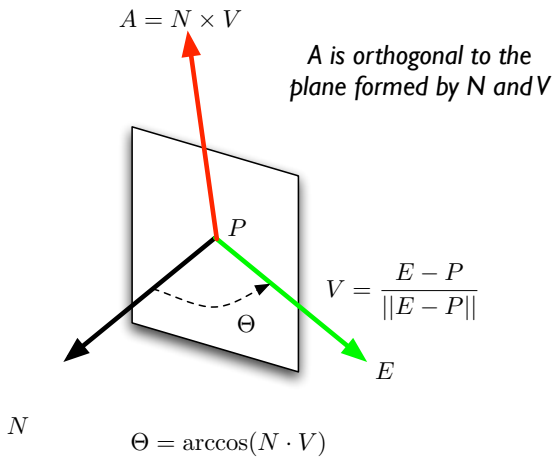


Figure 15: Calculating the rotation to billboard a plane at P, with normal N to point it towards the viewer at E.

To apply the rotation, `glRotatef()` is used and the texture becomes billboarded.

9.5.2 Particle Motion

Particles are given motion by applying an increment in their positions using their `direction` vectors (which were set randomly when they were created). The motions are applied using an idle callback, in `idle()`. Note that, in this example, I have made the particles bounce around a unit cube by making sure that their directions are mirrored each time the box boundary is reached.

9.6 Exercises

1. Compile and run `particles.cpp`. You can use the fast key, `' '`, to spin the cube.

2. Comment out the texture mapping and comment in the code which draws the edge of each texture map. You should then be able to see how the billboarding is working.

3. Change the transparency of the particles based on their lifetime. Use an exponential decay function of the form:

$$A(t) = \exp(-\lambda t) \tag{1}$$

   where t is the lifetime and λ is the half-life.

4. Use the lifetime of the particle to change its colour.

5. Modify the code to detect when two particles are close to each other and work out a way to make them collide and bounce off each other.

10 Lab 9 - Vertex and Fragment Shaders

This and the subsequent labs are dedicated to generating computer graphics using small programs associated with objects we want to visualise, called **shaders**. Shaders are the modern way to program computer graphics and since about 2006, have replaced fixed pipeline programs (such as those we've looked in the previous labs). However, the principles we've learnt still apply and indeed, applications can still be written in OpenGL and to a certain extent, GLUT.

After introducing the concept of **programmable pipelines**, we will look at some basic shaders, how they are programmed in OpenGL.

In this lab, we will specifically look at vertex and fragment shaders:

- create a **program object**, compile shader source code, link and **attach** shader objects within an OpenGL application
- look at a simple **pass-through** vertex and fragment shader, which once installed, perform identically to the OpenGL fixed pipeline
- learn how per-vertex (Gouraud) and per-pixel (Phong) shading can be achieved using vertex and fragment shaders
- understand how to pass data to shader objects from the main application and change their behaviour

10.1 Fixed and Programmable Pipelines

Recall that in the introduction, I talked about one view of OpenGL: as a **pipeline** of operations which takes vertices and their attributes as input, and generates coloured pixels on the output graphics window. OpenGL can be set up to perform operations which transform the vertices from Model Coordinates to World Coordinates, then to Camera or Eye Coordinates, and since vertices are given as parts of primitives (e.g. triangles), the OpenGL rasterisation operation will generate filled primitives, and interpolate the colours and lighting information to finally produce and set pixels in the frame buffer and map these to the output window. So the principal operations are: transformations of vertices to the viewing coordinates, the assembly of the primitives, the rasterisation and shading of the faces, that generate **fragments**.

Up until OpenGL version 2.0, the pipeline of operations could be controlled by changing OpenGL state, such as the transformation stacks, using different primitives, and altering the lights and materials or blending in textures, but the pipeline is essentially *fixed*. From version 2.1, OpenGL was officially extended to include a **programmable pipeline**, which allowed users to write small bits of C-like code to perform their own sets of vertex and fragment operations.

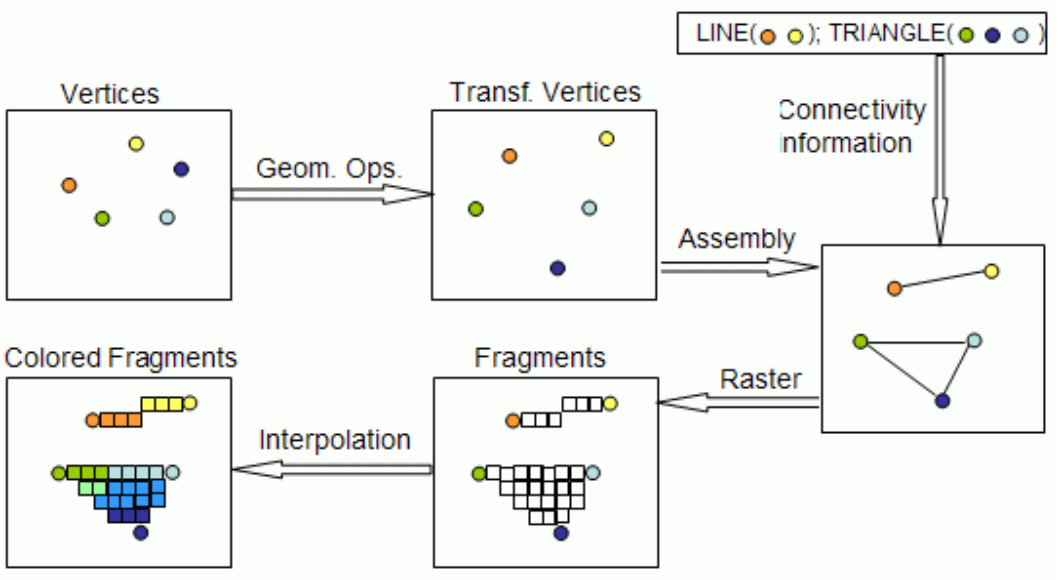


Figure 16: Operations of a Fixed Pipeline. Illustration taken from `www.lightouse3d.com`.

These programs are known as **shaders** and are written in a language called OpenGL Shading Language, or GLSL[15]. GLSL is based on ANSI-C but includes a number of C++ like type instantiations, vector and matrix algebra operator overloading, and some rather interesting and useful operators, such as swizzling (we'll come to that shortly). Here's some GLSL:

```
1 void main(void)
2 {
3     // transform the vertex using the modelview and
        projection matrix
4     gl_Position =  gl_ModelViewProjectionMatrix *
        gl_Vertex;
5
6     gl_FrontColor = gl_Color; // pass out the input
        colour to geom/frag shader
7 }
```

As mentioned, shaders are small programs (function or methods, if you like), and they are associated with a **program object** within the application, and may have one or more shaders linked into it. The application can have any number of program objects, and these will generally be associated with graphical objects

[15]I've heard that this is sometimes pronounced *gee-ell-slang*: what, really?

that are rendered in a particular way. The most important thing to know about shaders is that they are run on **every vertex**, **every primitive** and/or **every fragment** as the scene is being rendered. This lays open the full power of the GPU which is, for all intents and purposes, a special purpose parallel processor[16], operating on all the vertices, and all the fragments in one go to produce the frame buffer output.

10.2 Vertex, Geometry and Fragment Shaders

There are three main types of shaders which can be used by the graphics programmer to replace three important parts of the graphics pipeline. All the shaders take inputs and produce outputs (figure 17) by passing vertex and colour information from shader to shader, accessing various common state variables on the way:

- **vertex shader**: this is used to perform transformations of vertices from model coordinates to view or eye coordinates by applying the ModelView and Projection transformations. By default it takes one vertex and outputs a transformed vertex. Optionally, it can also read and write colour attributes associated with the input vertex. If we use a vertex shader and we don't want to do anything special, we must at least apply the viewing transformations. Vertex shaders operate on a **per-vertex** basis.

- **geometry shader**: this is used to manipulate the vertices at the level of the primitive to which they belong and all the vertices belonging to a given primitive are available to read by the geometry shader. For example, if a `GL_TRIANGLE` is draw by the application, then the three vertices of the triangle will be presented to any geometry shader installed by the user. The geometry shader can then either pass these on, delete them, or create new primitives.

- **fragment shader**: this is used to program operations on the pixels or fragments generated by the rasterisation operation, and hence operates on a **per-pixel** basis. The fragments are produced by the rasterisation and interpolation *after* the vertices have been transformed[17]. The main role of a fragment shader is to perform lighting and texturing operations on a per-pixel basis. If a fragment shader is installed by the user, then it must at least generate an output fragment colour.

[16] high performance computing now use GPUs as to perform computation in applications traditionally handled by the central processing unit in what is known as **GPGPU** - General-purpose computing on GPUs

[17] actually, we don't have to assume this is so, as we can perform all the same transformations on a per-pixel basis if we want to!

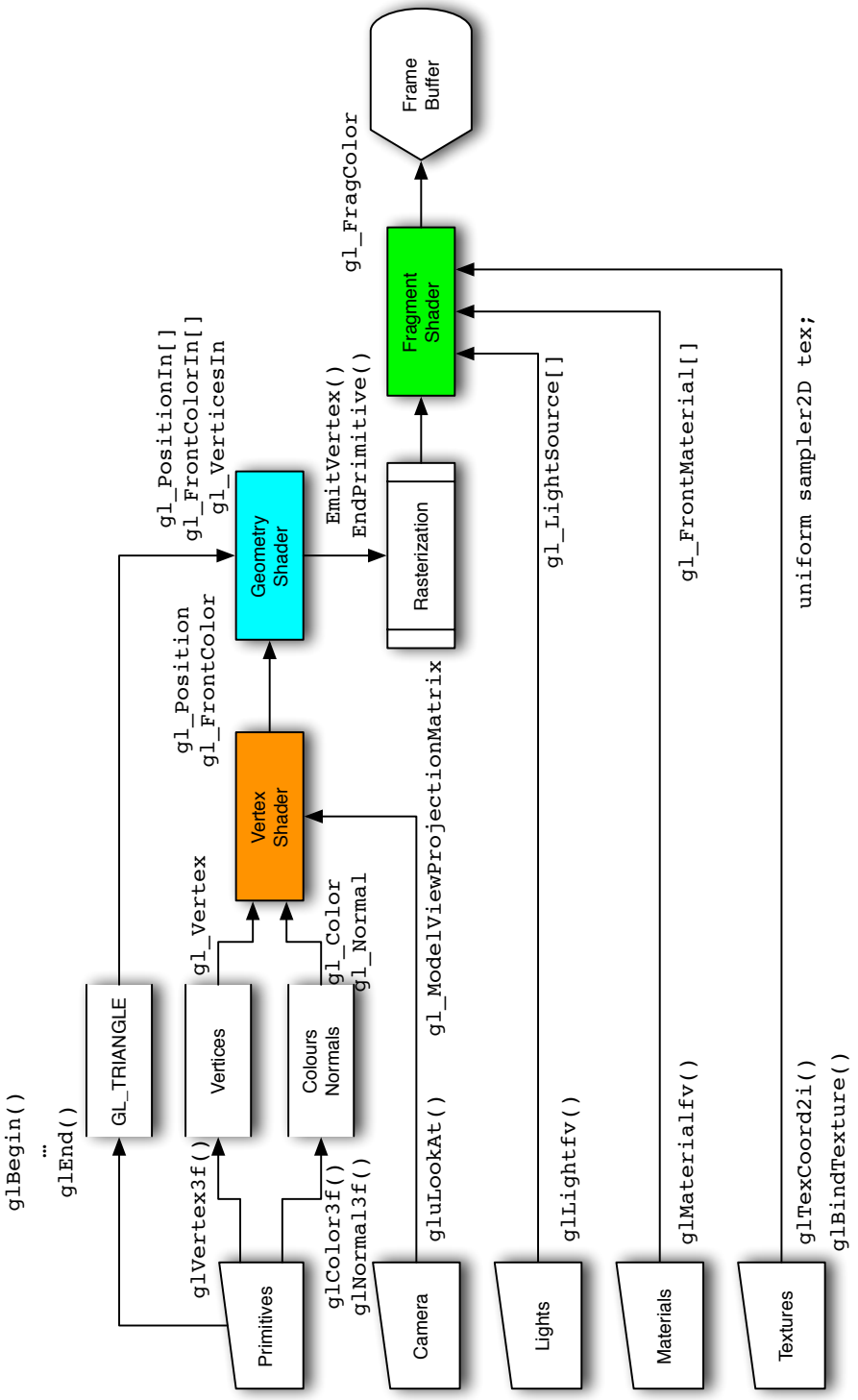


Figure 17: Programmable Pipeline showing operation of vertex, geometry and fragment shaders. The coloured boxes are programmable units.

Before we go on to look at the features and syntax of GLSL itself, we need to know how it integrates into OpenGL.

10.3 Compiling, Linking and Attaching Shader Objects

GLSL programs or shaders are compiled by the OpenGL application program itself. In fact, shaders can be written as literal strings in the source code, or read-in by the application from separate files. Once the source code (as a `NULL` terminated string of characters) is read in, it needs to be **compiled** and then **linked** into a **program object**. All of these steps have to done prior to installing or **attaching** the shader to the OpenGL application.

OpenGL provides a set of functions to do this. Here is an example listing of the minimum set of calls we need to make to create a program object, compile and link two shaders (note that `GLuint` is the same as an `unsigned int`):

```
// the shaders
const char vertex_source[] = {"void main(void) {}"};
const char frag_source[] = {"void main(void) {}"};

GLuint program_obj = glCreateProgram();
GLuint vertex_obj = glCreateShader(GL_VERTEX_SHADER);
GLuint frag_obj = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vertex_obj, 1, &vertex_source);
glShaderSource(frag_obj, 1, &frag_source);

glCompileShader(vertex_obj); // hope it compiles ok
glCompileShader(frag_obj);

glAttachShader(program_obj, vertex_obj);
glAttachShader(program_obj, frag_obj);

glLinkProgram(program_obj); // hope it links ok

glUseProgram(program_obj);
```

However, since we are compiling source code, we might have syntax errors, so it is good idea to check for errors after compilation and after linking using, for example:

```
int compiled = 0;
glGetShaderiv(vertex_obj, GL_COMPILE_STATUS, &compiled);
```

We can also printout the error logs and debug the code accordingly. A typical way to do this compilation-linking and error checking is written into the source file `create_and_compile_shaders.h` which has a utility method that takes just the filenames of the shaders you want to use and returns a program object ready to use.

```
// compile and attach shaders found in two source files,
    foo.vert and foo.frag
#include "create_and_compile_shaders.h"

// ...
unsigned int program_obj = create_and_compile_shaders(
        "foo.vert", NULL, "foo.frag");
```

10.4 GLSL Versions

All of the code examples I have provided in these labs use GLSL version 1.20. We can be explicit about this by using the directive `#version 120`, which I have done in some cases such as the geometry shaders. Version 1.20 has some limited support for geometry shaders. OpenGL 2.1's fixed pipeline allows state to be used in them. This means that the `GL_MODELVIEW` and `GL_PROJECTION` matrices are visible to the shaders. From version 1.40 the matrix stack is not available and instead, the programmer has to manage the projection and model view matrices explicitly in order to comply with the new OpenGL 3.1 standard. Note also that the GLSL extended in some very useful ways with some of the 1.20 syntax being deprecated.

For a fuller description of the different version and what this means for the programmer, please refer to the following resources:

1. The Wikipedia page: `en.wikipedia.org/wiki/OpenGL_Shading_Language`
2. Matt DesLauriers GitHub page `github.com/mattdesl/lwjgl-basics/wiki/GLSL-Versions` which has a very nice little guide on the main differences for the user.

As I said, in the following, I will assume you are using OpenGL 2.1 and GLSL 1.20.

10.5 Cross-platform support with GLEW

GLEW[18] is a third-party and is a cross-platform wrapper to allow the support of GL extensions (GLEW stands for the OpenGL Wrangler Library). It is useful to

[18] pronounced "glue"!

be able to port code from say Mac OS X to linux and not have to keep rewriting bits when some of the GPU/GLSL functionality is not implemented or versions are different. With GLEW, you can test whether a particular call or functionality is present. I've had to use GLEW to allow you to run stuff on Linux, and if you look at the application programs in this and the subsequent labs, you will see at the top

```
#include <GL/glew.h> // the glew header file
```

and in the `main()` function the following:

```
GLenum err = glewInit();
if (GLEW_OK!=err)
{
    fprintf(stderr, "Error: %s\n",
       glewGetErrorString(err));
    exit(1);
}
fprintf(stderr, "Using GLEW %s\n", glewGetString(
   GLEW_VERSION));
```

which initialises the `GLEW` and allows us to carry on as normal, more or less. Don't worry about this too much as once this is done, the rest is the same. If you do want to know more then take a look at `glew.sourceforge.net`.

Note that I have put the header and libraries in `/modules/cs324/glew` and `Makefile.linux` automatically builds with the headers and links in `libGLEW.so`. GLEW is a dynamic library, meaning it is linked to your program when your program runs. This means you need to add the GLEW lib directory to your dynamic library search path. In `bash` you can set this environment variable like this:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/modules/cs324
   /glew/lib
```

You can add this line to your `.bashrc` which is in your home directory. Another way is to add the libary path to your program's runtime search path. To do this, the provided Makefiles supply the argument `-Wl,rpath,/modules/cs324/glew/lib` the GCC at link time.

10.6 GLSL Language Basics

If you are like me, you will want to skip this section and just get on and try the example shaders which I have written, and pick up the details of the language syntax as you go. In this section, I will run through the basics, which at least you can quickly scan and refer to if you get stuck. There are plenty of good on-line resources on GLSL for reference (however, be careful to know which version you are using and referring to. In these labs we are using version 1.2 (or 120)). This reference card is useful `dac.escet.urjc.es/rvmaster/rvmaster/asignaturas/g3d/glsl_quickref.pdf`.

10.6.1 Simple and Extended Types

GLSL supports only `bool`, `int` and `float` for the simple types[19]. For floating point literals, we have to use the double notation, i.e. `1.0`, `4.5` etc.. However, it has a number of specific vector and matrix types:

- `vec3` for a 3D vector
- `mat4` for a 4×4 matrix

As with OpenGL, additionally GLSL allows for 1D, 2D and 4D versions of vectors, namely: `vec1`, `vec2`, `vec4`; and 2D, 3D versions of matrices: `mat2`, `mat3`. These types can be instantiated as if they were `structs` or classes:

```
vec4 v = vec4(1.0, 0.0, -3.0, 2.0); // a 4d vector
mat2 m = mat2(1.0, 0.0, 0.0, 1.0); // the 2x2 identity
    matrix
```

We can declare arrays of fixed dimensionality too, e.g.

```
float f[3] = {1, 2, 3};
vec2 v[2];
v[0] = vec2(1.0, 0.0);
```

So the indexing operator `[]` works as you would expect. For matrices, you uses the index operators work in row major order:

```
float c = m[1][2]; // coefficient in row 1 and column
   2
```

There is one more type which is important to know about, these create **sampler** variables, e.g. `sampler2D` would be a 2D image sampler. Samplers are associated with textures in the application program:

[19] and `void`

```
uniform sampler2D f_tex; // my texture
```

Samplers are used with the texture sampler function, e.g. `texture2D()`, which produces a `vec4` output of a texture colour value given a coordinate.

10.6.2 Operators and Swizzling

Vector and matrix type components can be accessed in two ways:

- by the index operator, e.g. `v[0]`, `v[3]`, etc.
- by named fields, e.g. `v.x`, `v.y`
- in groups of components by the **swizzling** operator: e.g. `v.xy` would produce a `vec2(v.x, v.y)` . Note that the components can be given in any combination, and any order.

We can equivalently use the field names `r g b a`, if we are referring to colours and `s t p q` when referring to texture coordinates.

We can assign parts of a vector or matrix to another by combining the instantiation with a swizzle:

```
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);
vec3 v3_swizzled = v.zyx; // swizzled version of v

float v4_w = v4.w; // just the 4th component, named w

v4.x = v4.x + 1; // add 1 to the first component of v4
```

The fourth component is the homogenous co-ordinate, `w`.

Component-wise mathematical operations are supported by overloaded arithmetic operators:

```
v_c = v_a + v_b; // add two vectors

v_d = v_a * v_b; // component wise multiplication

float s = dot(v_a, v_b); // dot product by builtin
   function
```

Matricies too have their arithmetic operators overloaded, e.g.

```
mat4 m = gl_ModelViewProjectionMatrix;
mat4 v = gl_Vertex;
```

```
vec4 v_eye = m * v;
```

10.6.3 Type Qualifiers

GLSL supports the `const` qualifier and this works as you would expect:

```
const float pi = 3.1415926; // declare a constant
```

It also has two important special modifiers: `uniform` and `varying`, which we will use.

- `uniform`: is a variable used in the shader which can be accessed in the OpenGL main program (through special functions), but will be read-only in the shader
- `varying`: is a variable that outputs its value from the vertex shader to the geometry and/or fragment shader. This value must be written by the vertex shader. A `varying` variable is read-only in the fragment shader.

For example, in a vertex shader we might write:

```
1 uniform float v_time; // time value from main program
2 varying vec4 my_colour; // pass to fragment shader
```

Then in the fragment shader, we must receive the `my_colour` value:

```
varying vec4 my_colour; // passed from vertex shader
```

There is another type qualifier, `attribute` which allows per-vertex attribute to be sent with the display list to the vertex shader. These are read-only in the vertex shader.

10.6.4 Built-in Standard Functions

All the mathematical and scientific functions you might need, such as those in `math.h` are built-in to GLSL:

- trignometric functions: `sin()`, `cos()`, `tan()`, `atan()`, etc.
- common ones: `abs()`, `sign()`, `min()`, `max()`, etc.
- vector algebra: `dot()`, `cross()`, `length()`, `normalize()`, etc.

More than these, GLSL provides some specialised **geometric functions**, which allow certain useful vector calculations to be performed, and come up a lot:

- `reflect()` reflect one vector in another, to find the mirror direction. Used in specular lighting calculations.
- `refract()` refract one vector given a normal and a refraction index `eta`, used in ray casting through transparent media.
- `ftransform()` is a short hand for the combined model view and projection transformation and applied to vertices only.

As mentioned above, the `texture2D()` is a function which can sample a given texture. It might be used like this in a fragment shader:

```
uniform sampler2D f_tex0;

gl_FragColor = texture2D(f_tex0, gl_TexCoord[0].st);
```

Note that `st` are the normalised source coordinates of the texture.

10.6.5 Control Structures and User Defined Functions

GLSL has the same control structures as C/C+, namely:

- loops can specified using `for (;;)` statements; `while...do` and `do...while`
- program branching and selection is done using `if` and `if...else`; and `switch...case` syntax

We can have user defined functions which take arguments by value and support return types. Note that there are no pointer-types or reference types. GLSL supports `in`, `out` and `inout` qualifiers to name the input and output parameters of a function. For example,

```
float degrees_to_radians(in float deg)
{
  return 4*atan(1)*deg/180.0;
}

void add(in float a, in float b, out float c)
{
  c = a + b;
}

void plus_plus(inout int x)
{
  x = x+1;
}
```

10.6.6 Built-in Uniform Variables

Shaders have read and sometimes write access to certain variables (state) associate with the incoming vertex, primitive or fragment. These are known as **built-in uniform** variables and can be recognised as they have the prefix `gl_` in front of them: e.g. `gl_Vertex` is the incoming vertex, which is of type `vec4`.

We can separate these variables by which shader has access to them, and whether they are input (read-only) or output, (read and write). The principal ones to know about are summarised in table 1:

Shader	*Uniform variable*	*RO/RW*
Vertex	`attribute vec4 gl_Vertex`	RO
	`attribute vec4 gl_Color`	RO
	`attribute vec3 gl_Normal`	RO
	`attribute vec3 gl_MultiTexCoord0`[20]	RO
	`varying vec4 gl_Position`	RW
	`varying vec4 gl_FrontColor`	RW
	`varying vec4 gl_TexCoord[]`	RW
Geometry	`attribute int gl_VerticesIn`	RO
	`attribute vec4 gl_PositionIn[]`	RO
	`attribute vec4 gl_FrontColorIn[]`	RO
	`vec4 gl_Position`	RW
	`vec4 gl_FrontColor`	RW
Fragment	`attribute vec4 gl_Vertex`	RO
	`varying vec4 gl_Color`	RO
	`varying vec4 gl_TexCoord[]`	RO
	`vec4 gl_FragColor`	RW

Table 1: The main built-in uniform variables of vertex, geometry and fragment shaders.

10.7 Programs `simple.cpp` `pass-through.vert` `pass-through.frag`

In the following programs, we will look at the OpenGL main application (in the C++ file, `.cpp`) and the shader programs. I've used the naming convention with extensions as follows:

- `.vert` for the vertex shader program source
- `.geom` for the geometry shader program source
- `.frag` for the fragment shader source

So, in the lab directory, there are various shader programs and main application programs. In all cases, to use the shaders you will have to compile the main program using the `Makefile` and then to run the program with two or more shaders, e.g.

```
1 $ make simple
2 $ ./simple pass-through.vert pass-through.frag
```

This will cause program `simple` to load, compile, link and run the given vertex and fragment shaders. Note that some of the programs, like `simple`, will actually produce visible output using the fixed pipeline only (for comparison).

Let us look at the pass-through vertex and pass-through fragment shaders. These are the minimal shaders we need for the program to effectively behave like a fixed-pipeline renderer. Here is the complete listing of the vertex shader, `pass-through.vert`:

```
1 void main(void)
2 {
3     // transform the vertex using the modelview
4     //                     and projection matrix
5     gl_Position = gl_ModelViewProjectionMatrix *
          gl_Vertex;
6
7     // pass out the input colour to geom/frag shader
8     gl_FrontColor = gl_Color;
9 }
```

The whole shader program is specified in a `main(void)` method which neither takes nor returns any value. Next the shader calculates the input vertex `gl_Vertex`'s projected coordinate by applying the combined transformation matrix `gl_ModelViewProjectionMatrix`. The output is put into the only output value that a vertex shader must write, `gl_Position`. Then finally, the input vertex

colour attribute, `gl_Color` is copied to the output front-colour, `gl_FrontColor`. These colour values will be those set by the application program using `glColor4f`. What we don't see here is the types of the variables: the vertex and position are all `vec4`'s and the matrix is a `mat4`. The input and output colours are `vec4`. This shader is known as a pass-through shader because it has simply implemented the standard fixed pipeline operation of transforming all vertices into camera space and normalised viewing coordinates, and passing their colours on to the rasterisation and interpolation stage, i.e. fragment processing.

Let's now look at the pass-through fragment shader and what happens *after* the rasterisation stage. Remember that the fragment shader is called for **every** pixel or fragment which is generated by the rasteriser. Here is the full listing of `pass-through.frag`:

```
1 void main(void)
2 {
3     // interpolate the received colour
4     gl_FragColor = gl_Color;
5 }
```

This shader is even less exciting than its vertex sister! This minimal fragment shader, which is called for every shaded pixel works on the interpolated values after rasterisation. We have to generate a fragment colour and set `gl_FragColor`. The `uniform` variable `gl_Color` is the interpolated fragment colour.

10.8 Exercises

1. First compile and run `simple` without the shaders, i.e.

   ```
   $ make simple
   $ ./simple
   ```

   It should be displaying a triangle with interpolated colours inside a wire cube and if you hit the space bar, it will rotate it.

2. Look at the code where the command line arguments are passed to load, compile and link any shaders on the command line.

3. Now use the pass-through shaders by specifying them as command line arguments to `simple`:

   ```
   $ ./simple pass-through.vert pass-through.frag
   ```

   Satisfy yourself that the display output is identical to the fixed pipeline version (i.e. run without shaders).

4. Modify the fragment shader to set a fixed colour value, rather than the input value. You can create colours like this:

```
vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
gl_FragColor = red;
```

5. Modify the vertex shader to output a fixed colour of your choosing. What is the difference in between this and getting the fragment shader to do the same?

6. Modify the vertex shader to offset the input vertex and then transform it. You can translate the input vertex like this:

```
vec4 v_t = gl_Vertex + vec4(dx, dy, dz, 0.0);
```

   Can you get the triangle to move to where you want?

10.9 Programs `uniform.cpp` `rotate.vert`

With this program, we will communicate with a vertex shader by passing values to it from the main program. In fact, we will set up the `idle()` call back to increment a counter and pass this to the shader so we can get the shader to animate vertices.

To pass data to the shader, we have to use variables with the `uniform` qualifier. In the vertex shader, `rotate.vert`, I have set up two uniform variables:

```
uniform int v_ticks;
uniform bool v_toggle;
```

To put values into these variables, we have first **locate** them and then set their values. OpenGL has a pair of functions to do this. If you look now at `uniform.cpp` and the `init()` method, you will see where I've set a *handle* for these variables:

```
        // get the location of the ticks variable
        v_ticks_loc = glGetUniformLocation(g_program_obj
           , "v_ticks");

        // get the location of an action toggle
        v_toggle_loc = glGetUniformLocation(
           g_program_obj, "v_toggle");
```

Both of the "handles" are `unsigned int`'s. Note that we have to give the identifiers used in the vertex shader as arguments to `glGetUniformLocation()`. Now to set the values, we use `glUniform1i()`. See for example the `idle()` method:

```
// if shader program is installed
if (glIsProgram(g_program_obj))
{
    glUniform1i(v_ticks_loc, g_ticks); // send to
       vertex shader
    post = true;
}
```

This copies the `g_ticks` value to the shader uniform variable `v_ticks` using the pre-determined handle, `g_ticks_loc`.

Note that the exact form of `glUniform` used will depend on the type and dimensionality of the variable being set. The function name follows the same rules as `glColor`, i.e. we say the dimensionally as `1 2 3 4` and then the type as `i f v`. For matrices, we have to use the `glUniformMatrix` variants.

In the vertex shader, `rotate.vert`, the triangle is rotated around the X-axis by applying a rotation matrix to the input vertices. A user defined function, `rotation_matrix()` creates the appropriate matrix from a given axis and angle (in radians):

```
if (v_toggle) // only rotate if told to
{
    // set up a rotation matrix about X axis
    mat4 mr_x = rotation_matrix(vec3(1, 0, 0),
       deg_to_rad(float(v_ticks)));

    // apply it before model view
    pos = mr_x * gl_Vertex;
}
```

Note the use of the uniform variable, `v_toggle` to gate the rotations.

10.10 Exercises

1. Compile and run `uniform` with the `rotate.vert` vertex shader and the pass-through fragment shader, i.e.

   ```
   $ make uniform
   $ ./uniform rotate.vert pass-through.frag
   ```

   Use the space bar to stop and start the spinning of the cube.

2. Modify the vertex shader to change the colour of the geometry using the `v_ticks` variable. Note that it can take values between 0 and 999, so you need to normalise it to make a colour component.

3. Look at the shader, `pulse.vert`, and use that instead of `rotate.vert`. Can you figure out how is works? Try altering `freq` and `amp` values and look at the effect. Remember you don't need to do anything other than use the shader on the command line to test it.

10.11 Per-Vertex and Per-Pixel Shading

OpenGL's fixed pipeline can shade polygons using the Phong Lighting model and, as we have seen in Lab 6 (section 7), that it uses interpolated shading with **Gouraud**'s method. This is enabled by making the call `glShadeModel(GL_SMOOTH)`. Interpolated shading is a **per-vertex** interpolation of the Phong Lighting values, see figure 18(a).

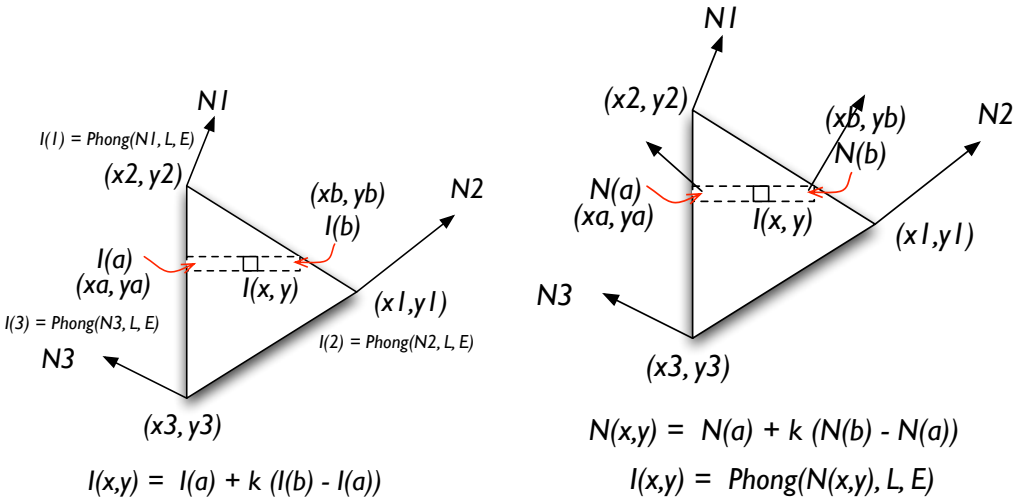


Figure 18: (a) Per-vertex versus (b) Per-pixel shading using Gouraud's and Phong's methods respectively.

The vertex normals at each vertex are averaged with other polygons which share that vertex (are incident at it), and these normals are used to work out the reflected value, I_r, at each vertex using Phong's formula:

$$I_r = I_i \left(k_d \mathbf{L}^T \mathbf{N} + k_s (\mathbf{R}^T \mathbf{E})^{k_e} \right),$$

where I_r is the reflected light energy, I_i is the incident light energy. k_d and k_s are the diffuse and specular properties of the surface. $\mathbf{R}$ is the mirror vector: the vector $\mathbf{L}$ reflected in the surface normal $\mathbf{N}$.

As we can see in figure 18, the shading values at each vertex are then interpolated by the rasterisation process to shade pixel values across the polygon. This process produces the identical values at edges shared by neighbouring triangles and so effects a smooth interpolation across the mesh.

A more accurate way to shade the polygon was proposed by Phong himself, and is known as Phong Shading, shown in figure 18(b). The main difference between Phong Shading and Gouraud's method is that instead of interpolating the lighting

values across the surface, it interpolates the vertex normals to calculate per-pixel normals, and then it uses these in the Phong's lighting calculation. This is known as **per-pixel** lighting, as the Phong lighting model is used at the pixel-level (as opposed to just once for each vertex). As you can imagine, this requires many more calculations. For a triangle which produces N fragments, it requires $3N$ more multiples to interpolate normals, and N more lighting calculations.

We can use a vertex shader to implement Gouraud Shading and a fragment shader to implement Phong Shading.

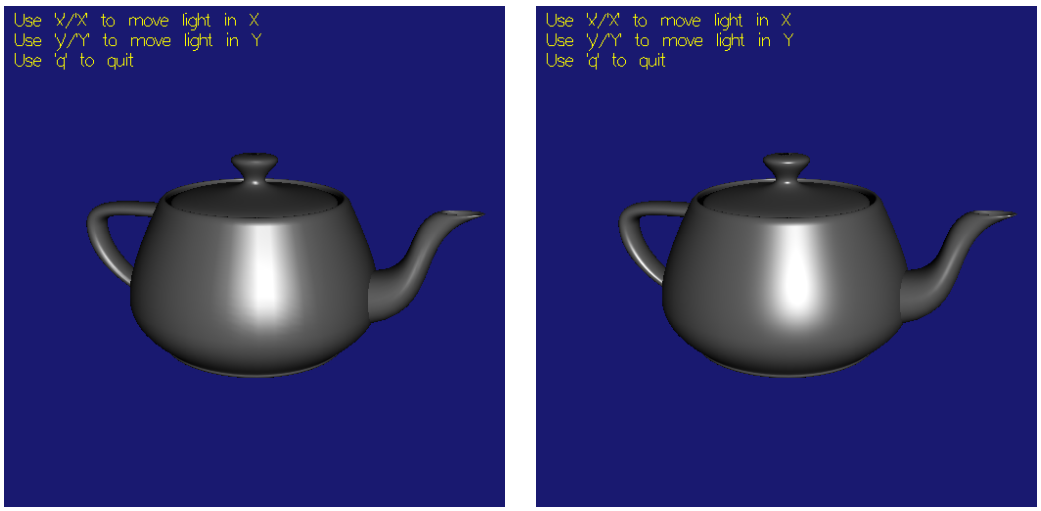


Figure 19: Per-vertex compared with Per-pixel shading using vertex and fragment shaders.

10.12 Program `phong.cpp` `phong.vert`

I've implemented per-vertex shading in the shader `phong.vert`, which can be run with the program `phong.cpp`. The shader source is given below (it's quite long but hopefully you should be able to read it and see how it is applying the Phong Shading model). Note that first light, `gl_LightSource[0]`, and the front material parameters, `gl_FrontMaterial`, are available through uniform structures, with fairly obvious field names[21]:

```
1
2 void main(void)
3 {
```

[21] you can look at a GLSL reference and find `struct gl_LightSourceParamters` and `struct gl_MaterialParameters`, if you want to know more.

```
4   vec4 pos;
5   vec3 normal;
6   vec3 light_direction, reflected;
7   vec4 ambient, diffuse, global_ambient, specular;
8   float N_dot_L; // lambertian
9   float N_dot_L_halfway; // reflected vector
      approximation
10  float N_dot_R; // specular
11
12  /* first transform the normal into eye space and
      normalize the result */
13  normal = normalize(gl_NormalMatrix * gl_Normal);
14
15  /* apply any model view transformation to input vertex
     */
16  pos = gl_ModelViewMatrix * gl_Vertex;
17
18  /* now normalize the light's direction after taking
      vertex pos
19    from the light away from pos */
20  light_direction = normalize(vec3(gl_LightSource[0].
      position)-vec3(pos));
21
22  /* compute the cos of the angle between the normal and
     lights direction.
23    The light is directional so the direction is
        constant for every vertex.
24    Since these two are normalized the cosine is the dot
        product.
25    We also need to clamp the result to the [0,1] range.
        */
26  N_dot_L = max(0.0, dot(normal, light_direction));
27
28  /* Compute the diffuse term */
29  diffuse = gl_FrontMaterial.diffuse * gl_LightSource
      [0].diffuse;
30
31  /* Compute the ambient and globalAmbient terms */
32  ambient = gl_FrontMaterial.ambient * gl_LightSource
      [0].ambient;
```

```
33    //global_ambient = gl_LightModel.ambient *
          gl_FrontMaterial.ambient;
34
35    /* Compute the specular term if N_dot_L is larger than
          zero */
36    specular = vec4(0.0, 0.0, 0.0, 0.0); // initialize
37    if (N_dot_L > 0.0)
38    {
39      // half way approximation
40      // normalize the half-vector, and then compute the
41      // cosine (dot product) with the normal
42      N_dot_L_halfway = max(0.0, dot(normal,
            gl_LightSource[0].halfVector.xyz));
43
44      // do it properly with reflected vector
45      //reflected = reflect(light_direction, normal);
46      //N_dot_R = max(0.0, dot(-normal, reflected));
47
48      // size of specularity in pow() bit
49      specular = gl_FrontMaterial.specular *
            gl_LightSource[0].specular *
50            pow(N_dot_L_halfway, gl_FrontMaterial.shininess)
                ;
51            //pow(N_dot_R, gl_FrontMaterial.shininess);
52    }
53
54    // vertex is now lit with Phong lighting value
55    gl_FrontColor =
56        //global_ambient +
57        ambient + // ambient
58        N_dot_L * diffuse +  // diffuse
59        specular; // specular term
60
61    // transform the vertex using the modelview and
          projection matrix
62    gl_Position =  gl_ModelViewProjectionMatrix *
          gl_Vertex;
63  }
```

10.13 Exercises

1. Compile and run `phong.cpp` with the per-vertex shader, like this:

```
$ ./phong phong.vert pass-through.frag
```

   Confirm that its lighting is the same as the fixed pipeline (run `phong` without any parameters). Move the light position using the fast keys, `'x'/'X'` and `'y'/'Y'`.

2. Add a fast key to change the specular exponent value in the material property and test the shader again.

10.14 Program `phong.cpp phong.frag`

To calculate per-pixel shading, we must implement the Phong lighting calculations in the *fragment* shader. I've done this for you in `phong.frag`. Take a look at the source file and you will notice that it looks remarkably like `phong.vert`. This is because most of the same uniform built-in variables available to us in the vertex shader are also exposed in the fragment shader. But remember that this program is going to be called for **every** fragment of any object rendering, after rasterisation takes place, so the GPU will be working much harder to perform these calculations.

We have to pass some information to the fragment shader about the vertex normals in eye space, which will be interpolated by the rasteriser. This is why we need to use a different vertex shader from the standard pass-through one and send the information through variables of type `varying`. I've named the paired vertex shader, `pass-through-phong.vert`:

```
1  varying vec4 pos_eye;
2  varying vec3 normal_eye;
3
4  void main(void)
5  {
6      // transform the vertex using the modelview and
          projection matrix
7      gl_Position =  gl_ModelViewProjectionMatrix *
          gl_Vertex;
8
9      // vertex position in eye coordinates
10     pos_eye = gl_ModelViewMatrix * gl_Vertex;
11
12     // normal in eye space
13     normal_eye = gl_NormalMatrix * gl_Normal;
```

```
14 }
```

Look at `phong.frag` now and note the following order of operations:

1. `normal` is the vertex normal (averaged between neighbours) and then interpolated in eye-coordinates.
2. Then we work out the light direction (note each light info is given in `gl_Lightsource[i]`), where `i` will be the light 0, 1, 2, etc.
3. Then we work out the diffuse and ambient components: note how the surface and light properties are multiplied: these are automatically per-component products.
4. Then if there is any Lambertian reflection `N_dot_L`, we look to use the Phong specular reflection calculation.
5. Note how the reflected vector can be approximated by half-way or using the built in `reflect()` function (how do you think it works exactly?)
6. Finally the front colour, `gl_FrontColour`, is calculated.

10.15 Exercises

1. Run the fragment shader with the modified pass-through vertex shader, like this:

```
$ ./phong pass-through-phong.vert phong.frag
```

   Compare the lighting produced by this shader with the per-vertex lighting. Move the light position using the fast keys, `'x'/'X'` and `'y'/'Y'`.

2. In `phong.frag` I've written two ways to calculate the mirror vector used in the specularity estimation: one using the `reflect()` function; the other using the **half-way vector** approximation. Compare the effect of using either `N_dot_L_halfway` or `N_dot_R` in the specularity calculation.
3. Add a fast key to change the specular exponent and compare the per-vertex and per-fragment shaders (you'll have to add it `phong.cpp`) – note the material properties are built in uniforms already.
4. How would this shader deal with multiple lights do you think?
5. Take a look at the two shaders `toon.vert` and `toon.frag` which can produce some interesting colour shading effects. Use these shaders with `phong` and see if you work out what they do and how they are doing it.

11 Lab 10 - Geometry Shaders

The programmable pipeline admits the creation of a special shader to handle geometry, in addition to the vertex and fragment shaders. This is known as the **geometry shader** and it sits logically between the vertex shader and the fragment shader.

The geometry shader is special because it gives access to primitives as generated by the main application. These are for example, sets of points, lines, line strips, triangles and triangle strips, specified by the application programmer using the modifiers given to the `glBegin()` function call, e.g. `glBegin(GL_TRIANGLE)` signals that there are three vertices to be regarded as forming a triangle.

Unlike the vertex shader which treats all vertices independently, the geometry shader is given access (through built-in uniform variables) to *all* the vertices of a primitive (and its associated vertex attributes, such as normals and colours). Furthermore, it can either send them back out without modification, modify their positions and attributes and then send them out, and/or, *create new geometry*.

Because of their ability to *generate* new geometry, geometry shaders can be used effectively to create all types of visual effects, such as explosions, particle effects, evaluated curves and surfaces, grass, hair and fur rendering. Geometry shaders are also useful for debugging shaders as they can annotate the rendering with, for example, colours and normals to reveal the values of attributes.

In this lab, we will examine and write geometry shaders and use them to create some interesting effects. Specifically, we will:

- learn about the built-in uniform variables passed between the vertex and geometry shader
- write a pass-through geometry shader and look at the vertex generation mechanism using the GLSL functions: `EmitVertex()` and `EndPrimitive()`
- use a geometry shader to "explode" an object
- use a geometry shader to show vertex normals on the Utah Teapot
- look at how "grass" can be modelled with a geometry shader

11.1 Pass-through geometry shader `pass-through.geom`

Let's look at a geometry shader. Here is the complete program of a pass-through geometry shader.

```
1 #version 120
2 #extension GL_EXT_gpu_shader4 : enable
```

```
3  #extension GL_EXT_geometry_shader4 : enable
4
5  void main()
6  {
7      for (int i=0; i<gl_VerticesIn;i++)
8      {
9          gl_Position = gl_PositionIn[i];
10         gl_FrontColor = gl_FrontColorIn[i];
11         EmitVertex();
12     }
13     EndPrimitive();
14 }
```

The first few lines are directives to indicate to the OpenGL driver that we are using GLSL version 1.20 and that the geometry shader extensions need to be enabled. This is not standard in version 1.20, so we have make this incantation. Then we have the shader code itself.

Like the vertex and fragment shaders, we have the program inside a `void main()`, then we see that there are some new uniform variables we can read and write, and two new built-in functions we can call:

- `gl_VerticesIn` is the number of vertices in the primitive being processed by the shader (this will depend on what was primitive was generated by the driving application).
- `gl_PositionIn[]` and `gl_FrontColorIn[]` are arrays containing vertex position and colours for each vertex of the primitive.
- `gl_Position` and `gl_FrontColor` are the output values we must set and will be passed to the fragment shader, after rasterization and interpolation.
- `EmitVertex()` pushes out the vertex at that point in the loop. If we don't issue an `EmitVertex()` call, then this vertex will be dropped by the shader and it will not be assembled and included in the rasterisation.
- `EndPrimitive()` signals that we are finished with processing the primitive.

A couple of things to note. Firstly, if we don't emit the incoming vertices then we won't see the primitive as intended by the calling application. However, we are free to use the incoming data to create our own geometry and set the `gl_Position` and `gl_FrontColor` of entirely new vertices and emit those. This is how we can make new geometry in a vertex shader.

We have to tell OpenGL which are the input and which are the output geometry types. OpenGL provides parameter setting functions, `glProgramParameteri()` to do this. In version 1.20, these part of the extension and called `glProgramParameteriEXT()`, for example we can write:

```
    glProgramParameteriEXT(program_obj,
       GL_GEOMETRY_INPUT_TYPE_EXT, GL_TRIANGLES);
    glProgramParameteriEXT(program_obj,
       GL_GEOMETRY_OUTPUT_TYPE_EXT, GL_TRIANLE_STRIP);
```

We also need to tell the GPU the maximum number of vertices we will output:

```
  int max_vertices = 32;
    glProgramParameteriEXT(program_obj,
       GL_GEOMETRY_VERTICES_OUT_EXT, max_vertices);
```

I've created a helper function called `set_geoemtry_shader_params()` to do this:

```
void set_geometry_shader_params(
            unsigned int program_obj,
            unsigned int in_type,
            unsigned int out_type,
            int max_vertices)
```

Going back to the vertex and geometry shader, in theory, we can communicate other attributes, such as vertex normals between them using variables of type `varying`. If we do this, the per-vertex variable is automatically made into an array type in the geometry shader. This is why `gl_Position` output from the vertex shader becomes `gl_PositionIn[]` in the geometry shader.[22]

To use the pass-through geometry shader, we can run it with the teapot program and the other null shaders:

```
$ make teapot
$ ./teapot pass-through.vert pass-through.geom pass-
   through.frag
```

Yes, it renders the teapot in wireframe! Don't fall off your chair.

[22]However, I've not succeeded in achieving this on Mac OS X using OpenGL 2.1 and GLSL version 1.20. We can get round it for the purposes of this lab and I believe it will work under Linux.

11.2 Programs `teapot.cpp explode.vert explode.geom`

The shaders `explode.vert` and `explode.geom` should be looked at together. Here is the the vertex shader:

```
1  uniform int v_ticks;
2
3  void main(void)
4  {
5      // transform the vertex using the modelview and
          projection matrix
6      gl_Position =  gl_ModelViewProjectionMatrix *
          gl_Vertex;
7
8      gl_FrontColor = gl_Color;
9
10     // push the normal displacement into the back colour
11     vec3 normal = gl_NormalMatrix * gl_Normal;
12     float scale = float(v_ticks)/2000.0;
13     gl_FrontSecondaryColor = vec4(scale * normal, 0.0);
14 }
```

You can see it uses the `v_ticks` uniform (incremented by the `idle()` function of the calling program `teapot.cpp`) to scale the vertex normal vector and put it into the `gl_FrontSecondaryColor` variable. I do this to get round the fact that I was not able to pass it as expected, but it does the job.

Now, if we look at the geometry shader, we will see how these scaled normals (which will get bigger and bigger with time) are used:

```
1  #version 120
2  #extension GL_EXT_gpu_shader4 : enable
3  #extension GL_EXT_geometry_shader4 : enable
4
5  void main()
6  {
7      // pass out the input vertex
8      for (int i = 0; i < gl_VerticesIn; ++i)
9      {
10         gl_Position = gl_PositionIn[i];
11         gl_FrontColor = gl_FrontColorIn[i];
12         EmitVertex();
13     }
14     EndPrimitive();
```

```
15
16      // duplicated and push away from original
17      for (int i = 0; i < gl_VerticesIn; ++i)
18      {
19          // normal coded in secondary colour!
20          vec4 motion = gl_FrontSecondaryColorIn[i];
21
22          gl_Position = gl_PositionIn[i] + motion;
23          gl_FrontColor = vec4(1.0, 0.0, 0.0, 0.5);
24          EmitVertex();
25      }
26      EndPrimitive();
27
28  }
```

The top of the shader is simply the same as the pass-through geometry shader. This means that we will see the original geometry as was, in its original colour. Then we repeat the exercise, this time we draw the geometry pushed out by the scaled normal, passed to us in the `gl_FrontSecondaryColorIn[]` array. The colour of the newly generated geometry is coloured red and made half transparent.

11.3 Exercises

1. First compile `teapot.cpp` and test out the explode geometry shader. You will need to run it like this:

```
$ ./teapot explode.vert explode.geom pass-through.
   frag
```

2. Experiment with `explode.geom`. Change it to not output all the incoming vertices and see what happens.

3. Change the colour of the output geometry based on the `v_ticks` value.

4. In `explode.geom`, use the `gl_Position[]` array to work our the primitive normals instead of the cheat that I used by making the vertex shader encode them into the `gl_FrontSecondaryColor`.

5. Look at `normals.vert` and `normals.geom`. Test it out with the teapot and confirm what it does.

6. Modify the geometry shader to colour the normal by its direction using the normal components to modify the RGB colour values of the front colour.

11.4 Programs `grass.cpp grass.vert grass.geom`

In the last example, I have used a geometry shader to "grow" blades of grass and then animate them using a timer tick from the OpenGL driver `idle()` method and some sine and cosine functions to rotate and scale the positions of the blade as it grows.

Each blade of grass starts off as a line using `GL_LINES` and so has two vertices in it. To start with we set the geometry shader params using my helper function in the `init()` method:

```
        set_geometry_shader_params(g_program_obj,
              GL_LINES, // input primitive
              GL_LINE_STRIP, // output primitive
              128 // max vertices
        );
```

To make things a bit more interesting, I have placed the "seeds" for the grass in a random grid positions in the `display()` method. Each blade is set up like this

```
  glBegin(GL_LINES);
        float rand_x = 2*float(drand48())-1.0f;
        float rand_z = 2*float(drand48())-1.0f;
;
        glColor4f(1, 1, 1, 0.5); // arbitrary colour
        glVertex3f(rand_x, -0.2f, rand_z);
        glNormal3f(0, 1, 0); // point up
        glVertex3f(rand_x, -0.25f, rand_z);
        glNormal3f(0, 1, 0); // point up
    glEnd();
```

Now, in the vertex shader, `grass.vert`, I have used the `gl_FrontSecondaryColor` to pass extra information from the vertex shader to the geometry shader, `grass.geom`. Because I want each blade of grass to grow and "wave" sinusoidally, I use the `v_ticks` uniform, which I know will go in angles round a circle, to give me a phase value in radians:

```
    vec3 rand = noise3(gl_Vertex.xyz);
    float phase = 2.0*atan(1.0)*4.0*(float(v_ticks)
       /180.0); // radians

    // encode values to pass to geometry shader
    gl_FrontSecondaryColor = vec4(rand, phase);
```

I have used a 3D noise generator (another one of the built-in functions), `noise3()` to give me a random 3D vector. (If it does not appear to work, then comment in the `my_noise3()` line which uses GLSL noise functions defined in the shader.)

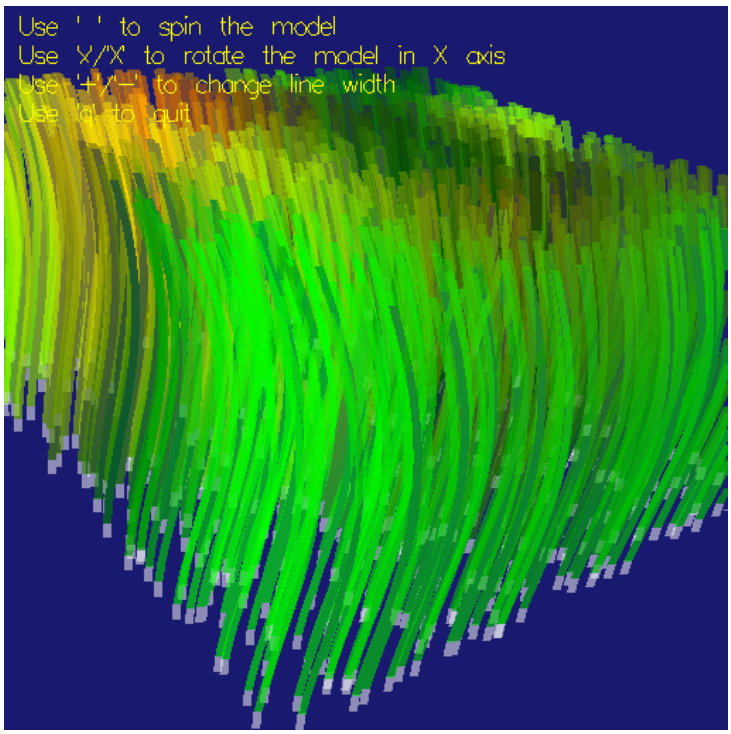


Figure 20: A rendering from my grass geometry shader.

To understand the geometry shader, `grass.geom`, you need to know the model for the grass blade I have designed.

- each blade starts to grow in the Y-direction in $\frac{1}{n}$ steps, where n is set to be 16.
- its X and Z positions are generated as points in a circle with radius `amp`, which itself varies linearly with the height of the blade
- the X and Z positions are given by `sin()` and `cos()` values modulated by the `phase` and `rand_x` and `rand_y` values form the vertex shader
- the colour is modified by the random values too to be shades of yellow and green

11.5 Exercises

1. Compile `grass.cpp` and run the grass geometry shader like this:

```
$ ./grass grass.vert grass.geom pass-through.frag
```

   Play around with the size of the blade and the amplitude value: `amp`.

2. Make the grass change colours in a different way.

3. Rewrite the grass shader to grow a blade of grass with quads rather than line segments. You will need to modify `grass.cpp` appropriately.

12 Lab 11 - Texture and Bump Mapping with Shaders

OK, we have 10 weeks but there are 11 labs, but all good tutorials must go up to 11, so here we are.

The final lab will look at how we perform texturing and **multi-texturing** using fragment shaders. Specifically, this lab considers:

- texture mapping and **multi-texturing**, which is combining textures on the same surface
- how to do texture mapping with **samplers** in fragment shaders
- how to perform **environment mapping** with shaders to imitate reflective surface properties
- how to specify and use **normal maps** or **bump maps**

12.1 Multi-texturing in OpenGL

Multi-texturing is the process of applying more than one texture map value to the surface of a polygon. When we looked at texture mapping, in section 8, we only considered the application of a single texture to a surface. Remember to do this we had to specify **source** and **target** texture co-ordinates. The source co-ordinates are the positions of 3D vertices, and the source coordinates are **normalised** co-ordinates specifying the positions of corners in the texture image.

In program **crate.cpp** I have textured a cube using a "crate" texture in `crate.png`[23]. But looking at the display method reveals that instead of using `glTexCoord2i()`, I have used `glMultiTexCoord2i()`, which takes **three** arguments, the first of which is the name of a **texture unit** in the GPU: `GL_TEXTURE0`. I have also bracketed the specification of source-target pairs with activating the texture unit used and then activating the next one:

```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, g_crate);
  //...
glActiveTexture(GL_TEXTURE1);
glDisable(GL_TEXTURE_2D);
```

[23] every good video game should have a crate, or a barrel.

The business of activating the next one indicates to the GPU that I will not be using any more. In fact, GPUs will have multiple texture units, and we can find out how many using the call I have made in the `init()` method:

```
int max_texture_units = 0;
glGetIntegerv(GL_MAX_TEXTURE_IMAGE_UNITS, &
   max_texture_units);
fprintf(stderr, "Max texture units is %d\n",
   max_texture_units);
```

Because we have multiple texture units, we can bind multiple textures and specify the same target coordinates, but different source coordinates. Now, look at the `display()` method of `crate-multi.cpp` which performs multi-texturing.

This time I activate two texture units and bind in two different textures:

```
// bind textures for multi-textuturing pipeline
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, g_crate);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
   GL_COMBINE);

glActiveTexture(GL_TEXTURE1);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, g_radioactive);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
   GL_MODULATE);
```

And the, when I render the faces of the cube, I give the source coordinates to both the texture units:

```
glMultiTexCoord2i(GL_TEXTURE0, 0, 0);
glMultiTexCoord2i(GL_TEXTURE1, 1, 0);
glVertex3fv(g_cube_vertices[g_cube_faces[f][0]])
   ;
// and so on
```

The crucial thing are the `glTexEnvi()` calls as each texture is bound. These tell the OpenGL **texture combiners**[24] how to combine the two textures values at the fragment stage. Here, the first should be painted on and the second should modulate the first. The problem with this way of multi-texturing is that it is

[24]see the insanely complicated advice on this page `www.opengl.org/wiki/Texture_Combiners`!

complicated to work out how to achieve the result we want. The situation is a lot simpler with shaders.

Compile and run `crate-multi.cpp` to see what it does.

12.2 Programs `texture.cpp` `texture.vert` `texture.frag`

Before we look at how to combine textures, we need to know how to simply map texture values onto fragments. As you may have guessed, this is a job for a fragment shader. In the introduction to GLSL earlier, I mentioned an extended type: `sampler2D`, but we've not used them yet. Samplers (for example variables of types `sampler2D`), are representations of textures within the shader. We usually declare a variable of type `sampler2D` as `uniform` and expect the main application to bind a texture to it.

This is the done as for any other uniform type:

- we obtains its location using `glGetUniformLocation()`
- we set its value to be the number of the texture unit we want the sampler to bind to, for example the call, `glUniform1i(f_tex0_loc, 0)`, indicates that texture unit 0 is going to be used

Then to do the texturing we proceed as before, by using `glActivateTexture()` and then binding the texture and specifying source and target coordinates.

Here is the complete listing of `texture.frag` which declares a sampler variable and uses it:

```
1 #version 120
2
3 // which texture?
4 uniform sampler2D f_tex0;
5
6 void main(void)
7 {
8     gl_FragColor = texture2D(f_tex0, gl_TexCoord[0].st);
9 }
```

In this example, the fragment colour is solely set to be the pixel from the presented texture, `f_tex0`. The actual look-up of the pixel colour value is performed by the built-in function `texture2D()` which takes the texture sampler handle, and the source coordinates from the transformed texture coordinates: `gl_TexCoord[0].st` using the `.st` twizzle. Each texture unit has its own coordinates, `gl_TexCoord[i].st`, where `i` is the number of texture unit.

Note that to get the correct texture co-ordinate value in the fragment shader, we **must** transform the texture coordinates, using the appropriate texture matrix, `gl_TextureMatrix[]` in the vertex shader. Here is what the `texture.vert` vertex shader looks like:

```
1 void main()
2 {
3     gl_Position = ftransform();
4     gl_TexCoord[0] = gl_TextureMatrix[0] *
          gl_MultiTexCoord0;
5 }
```

One final thing to say, in `texture.cpp`, I am auto-generating texture co-ordinates using `glTexGeni` as object linear. If you are not sure, look again at section 8.

Figure 21: Utah teapots with base texture and environment map.

12.3 Exercises

1. Compile and run the example texture mapping shader, like this:

```
$ make texture
$ ./texture texture.vert texture.frag
```

2. Modify the fragment shader to alter the hue of the texture by adding red to it. I've commented in some code to do this.

12.4 Environment Mapping

Program `teapot.cpp` to be used with the vertex shader `texture-env.vert` and the fragment shader `texture-env.frag` and these shaders demonstrate **environment mapping**. When we did environment mapping before in lab 7, section 8, we used the texture co-ordinate auto generation functions to create texture coordinates suitable for a `GL_SPHERE_MAP`. In the main application, we wrote something like this:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

We didn't examine closely what this does, but now is the chance to try to understand it. Sphere mapping is a way to use an intermediate surface, an O-map, on which the texture is hypothesised to be on, instead of a plane. In the case of a sphere-map, this is of course the surface of a sphere, and we pretend that our object lies in the middle of it. What the auto-coordinate generation does is to apply a formula to find out what the reflected vector of the **eye vector** in the surface normal point to and uses this to locate a point on the surface in `S` and `T` coordinates.

If we denote the direction vector to the vertex from the eye point, as a normalised vector form the origin $\mathbf{u}$, in eye-coordinates, this is simply the position of the vertex (as the eye is at the origin). In a shader, we can calculate it by:

```
vec3 u = normalize(vec3(gl_ModelViewMatrix * gl_Vertex
   ));
```

Then, next we need the normal to the surface, $\mathbf{n}$, at the vertex point, and again, in eye-coordinates, this is given by using the normal matrix:

```
vec3 n = normalize(gl_NormalMatrix * gl_Normal);
```

Next we need to find the mirror vector, $\mathbf{r}$, of the eye vector $\mathbf{u}$ in the surface normal $\mathbf{n}$, GLSL conveniently provides us with a function to do this[25]

```
vec3 r = reflect(u, n);
```

The last bit is magical. We define a scalar, and use it to give the texture coordinates for the sphere map as `S` and `T` by scaling and shifting the X and Y components of $\mathbf{r}$[26]:

[25] it implements the Householder perpendicular projector :$\mathbf{r} = \mathbf{u} - 2(\mathbf{n} \cdot \mathbf{u})\mathbf{n}$

[26] the `man` page of `glTexGen` will confirm that this is indeed what the `GL_SPHERE_MAP` modifier does!

```
float m = 2.0 * sqrt( r.x*r.x + r.y*r.y + (r.z+1.0)*(r.z
   +1.0) );
float s = r.x/m + 0.5;
float t = r.y/m + 0.5;
```

I have implemented this sphere-map coordinate generation in `texture-env.vert`, and it uses it to set the `.st` values of `gl_TexCoord[1]` ready to be used by the fragment shader, `texture-env.frag`. Here is the code of the fragment shader:

```
1  uniform sampler2D f_tex0;
2  uniform sampler2D f_tex1; // assume this is the
      environment map
3
4  void main (void)
5  {
6      vec4 colour = texture2D(f_tex0, gl_TexCoord[0].st);
7      vec4 env = texture2D(f_tex1, gl_TexCoord[1].st);
8
9      gl_FragColor = env; // environment map only
10     //gl_FragColor = colour*0.9 + env*0.1; // blended
11 }
```

By using two uniform samplers for two different textures, and two texture units in the main program, we can combine the texture values if we want, in any way we want. The first is the base colour and the second is the "specular" reflection. Two texture coordinates are used to look up then combined in the fragment shader.

12.5 Exercises

1. Compile and run the environment/multi-texture mapping shader with program `teapot.cpp`, like this:

   ```
   $ make teapot
   $ ./teapot texture-env.vert texture-env.frag
   ```

2. Change the line in the fragment shader where `gl_FragColor` is set to use both the base colour and the environment in some linear combination.

3. Try different textures in the main program (loaded in the `init()` function).

12.6 Multi-texturing with Diffuse Lighting

The previous example used multi-texturing to combine the texture pattern of the material with the environment (effectively the specular lighting). What if we wanted to have other diffuse properties, such as Lambertian shading, the base texture and the environment? This is not unreasonable.

Program `teapot.cpp` can be used with the shaders `texture-env-light.vert` and `texture-env-light.frag`. The main difference is in the fragment shader, `texture-env-light.frag` now calculates the Lambertian term (from the light and material properties) and adds these in some way to the base texture and the reflection map.

12.7 Exercises

1. Compile and run the new shaders with program `teapot.cpp`, like this:

```
$ make teapot
$ ./teapot texture-env-light.vert texture-env-light.
   frag
```

2. Comment in and out the different parts of `texture-env-light.frag` to change the base, diffuse and specular lighting of the teapot.

12.8 Normal (Bump) Mapping with Shaders

Normal, or bump mapping, is a way to make small alterations (perturbations) to these surface normals which have an effect on the lighting values and simulate the roughness of a surface. We can use a texture to encode a normal map which is a map of perturbations of to be applied to the interpolated surface normals, in the tangent coordinates of the surface. We've already seen that in a fragment shader, we have access to the per-pixel surface normals and we have already used it to perform per-pixel shading.

The theory of bump mapping shows us that small alterations in the height of a surface can be simulated by X and Z increments to the surface normal (in the tangent space). We can do this in a fragment shader by setting up a texture which has the normal (or bump) map encoded in two of its colour channels: usually the red and blue channels. See for example the images `normals.png` or `chesterfield.png`.

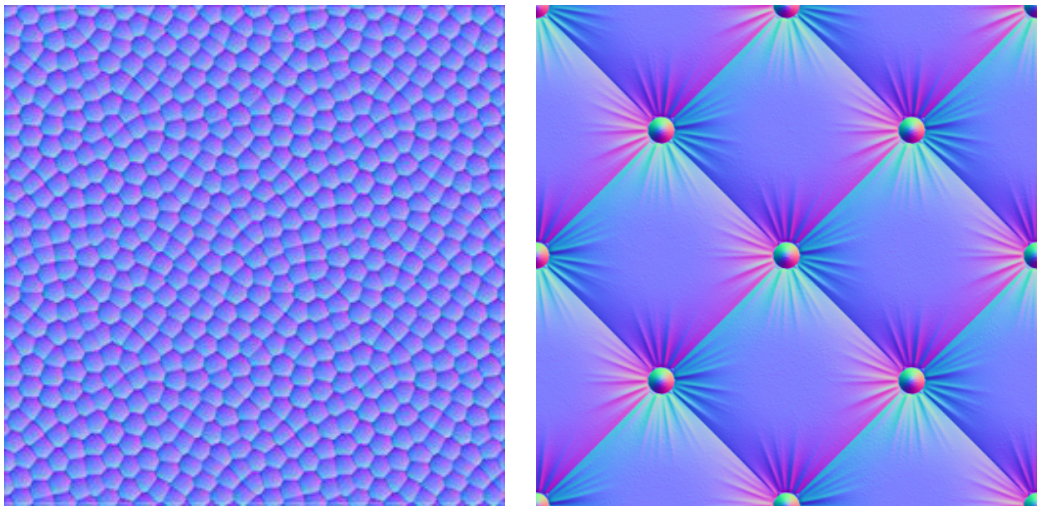


Figure 22: Two bump maps (`normals` and `chesterfield`) encoded as PNG images.

To apply the map, we use it to affect the lighting in the fragment shader by perturbing the normal calculation by the values in the normal texture map. The effect is more pronounced if we perform per-pixel lighting also.

Note that my implementation is not quite correct as the perturbation should be performed in tangent space, not eye-space. This is more complicated as we need to have the in-plane vectors of the surface, so called, tangent and `binormal` vectors. But without specifying them when the geometry is sent to the shader, it

is not usually easy to get these[27].

12.9 Program bump.cpp bump.vert bump.frag

Look at program `bump.cpp` which loads up a base texture and a normal map (`normal.png`). Then the bump map is applied, together with the base texture, by the fragment shader, `bump.frag`. The important lines are decoding the perturbation out of the texture's RGB values, and then applying it to the interpolated eye-space normal:

```
// Extract the normal from the normal map
vec3 p_normal = normalize(texture2D(f_tex1,
   gl_TexCoord[0].st).rgb * 2.0 - 1.0);

// apply as a pertubation to actual normal
vec3 normal2 = normalize(gl_NormalMatrix * (normal +
    p_normal));
```

I have also written two other shaders which perform per-pixel Phong lighting, which has a more spectacular effect with the bump-mapping, these are in `bump-light.vert` and `bump-light.frag`. The latter looks very similar to a per-pixel lighting shader with the small change that the surface normal is perturbed by the normal map before it is used. Now to try them out...

12.10 Exercises

1. First compile and run `bump` with `bump.vert` and `bump.frag`, like this:

```
$ make bump
$ ./bump bump.vert bump.frag
```

2. Now change the bump map being used by `bump.cpp` to be `chesterfield.png` and run it against `bump-light.vert` and `bump-light.frag`. Compare the results with 1.

```
$ make bump
$ ./bump bump-light.vert bump-light.frag
```

3. How would you make your own bump map? There is a plug-in to GIMP to do it. If you have GIMP, get the plug-in and try making your own bump map.

[27] a geometry shader could work these out and pass them to the fragment shader

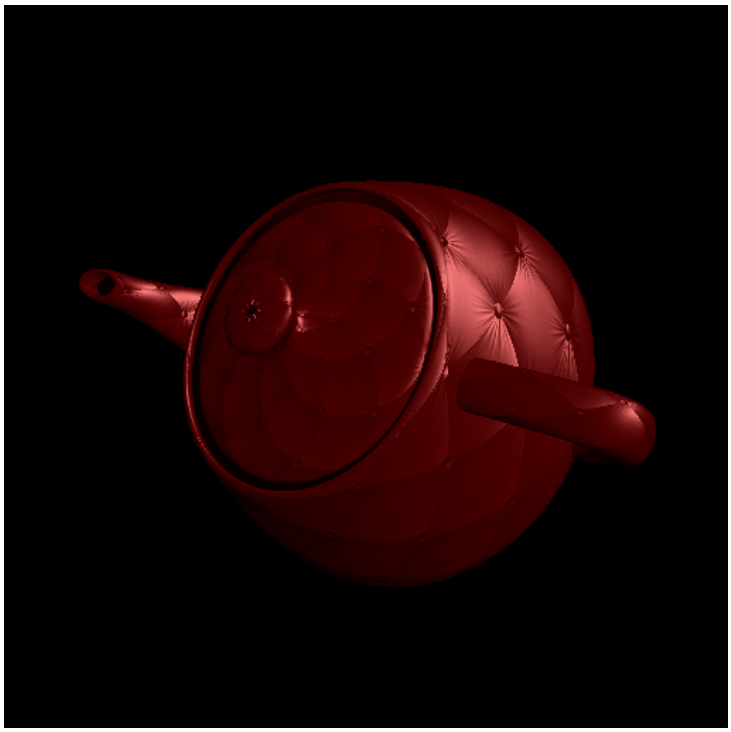


Figure 23: Utah teapot with a rather comfortable looking Chesterfield upholstery.

4. Search on-line on how to perform the correct tangent space calculations for bump-mapping, e.g. `content.gpwiki.org/index.php/OpenGL:Tutorials:GLSL_Bump_Mapping`.

Index