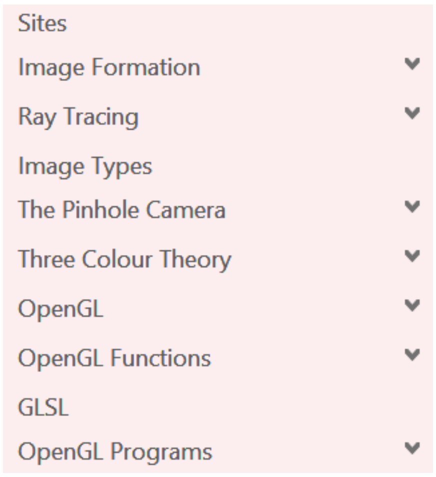
Week 1


Sites
Image Formation
Ray Tracing
Image Types
The Pinhole Camera
Three Colour Theory
OpenGL
OpenGL Functions
GLSL
OpenGL Programs

Sites

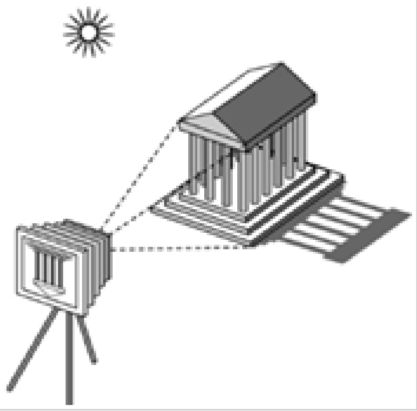
[Handbook]

[CSSE]

Image Formation involves forming 2D images similar to a camera.

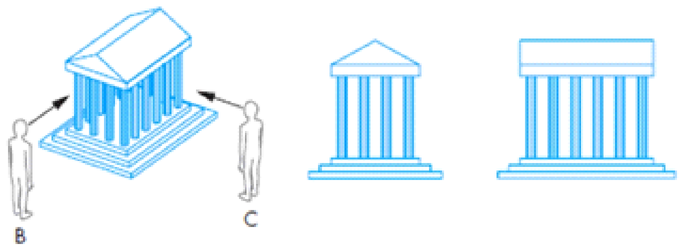
Three *independent* elements:

1. Viewer
2. Objects
3. Light Sources



The viewer:

- Forms the image of our objects
- Decides which side(s) of the object(s) will be visible
- Is the most important element
- May be a human, camera or digitizer



Objects:

- Are *defined/approximated* by a **set of locations** (vertices)
- Have attributes that govern how light interacts with them
  - E.g. Colour determines the light frequency that reflects from it

Light Sources

- Light sources can emit light as a set of discrete frequencies or within a band range.
- Geometric optics models light sources as emitters of light energy, each of which have a fixed intensity.
- Light travels in straight lines from its source to the objects.
- An ideal point source emits energy from a single location at one or more frequencies equally in all directions.
- A particular source is characterized by:
  - The intensity of light that it emits at each frequency
  - That light's directionality

Advantages of Independence

It is **advantageous to have each element independent** because:

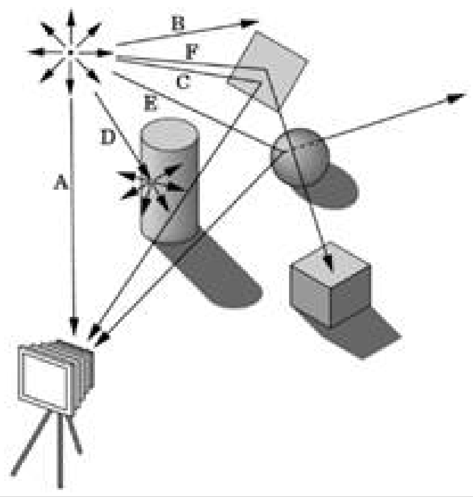
- Leads to simple software API
  - Each part can be specified separately
  - Implementation determines image
- Each separate element can be modelled separately
- Fast hardware implementation
- 2D graphics becomes a special case of 3D graphics. The 3rd dimension becomes non-existent and so the 3D scene is modelled with 2D objects.

Ray Tracing

Ray Tracing is an image formation technique in which rays of light are followed from their source to *determine which ones enter the camera lens*

Ray Tracing is **complex** because rays of light may:

- Have multiple interactions with objects
- Get absorbed
- Go to infinity
- Never end up entering the camera lens



Usage

Ray Tracing is advantageous because:

- It is more **physically based**
- It is simple for **simple objects** (e.g. polygons and quadrics) with simple point sources
- We can produce **global lighting effects** such as shadows and multiple reflections
- Practical ray tracing with GPUs is getting close to real time

But, ray tracing is still slow and thus not suited for interactive applications

So, we use OpenGL instead of Ray Tracing

Image Types

An image may be a:

- Luminance image
  - Grayscale images (black, grey, white)
  - Monochromatic
  - Values are grey levels
- Colour image
  - Has perceived attributes of hue, saturation and lightness

The Pinhole Camera

The **Pinhole Camera** is the simplest and most common imaging system/camera model

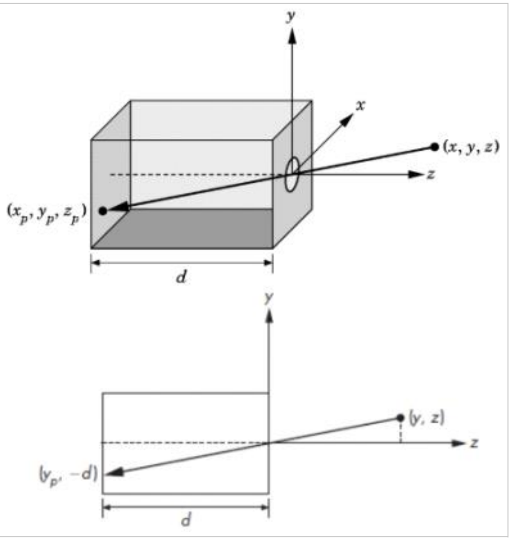
All light in the box passes through one point

The projection of the point (x, y, z) gives $(x_p, y_p, -d)$.

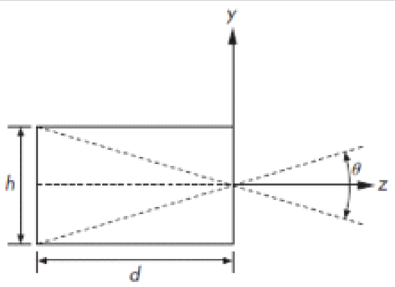
(The subscript p stands for projection)

We use the trigonometric equations of simple perspective:

$\frac{x}{z} = \frac{x_p}{z_p}$	$\frac{y}{z} = \frac{y_p}{z_p}$	$z_p = -d$



Properties

The field/angle of view (θ) of our camera is the angle made by the largest object that our camera can image on its film plane.	 $\theta = 2\tan^{-1}\frac{h}{2d}.$

The ideal pinhole camera has an infinite depth of field - everything in front of the camera is in focus

Disadvantages

The pinhole camera has **two disadvantages:**

- It admits <u>only a single ray</u> from a point source—almost no light enters the camera
- The camera has a <u>fixed angle of view</u>

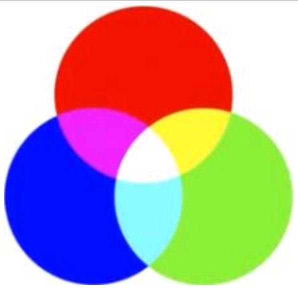
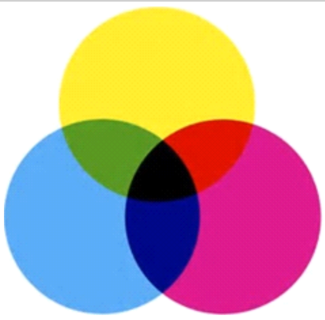
Three Colour Theory

The human eye has two types of sensors:

- Rods
  - Monochromatic
  - Night vision
- Cones
  - Color sensitive
  - Three types of cones
  - Three values (tristimulus values) are sent to the brain

Therefore, we only need **three primary colors** (RGB) to model all colors

Additive and Subtractive Colour

Additive Colour • Colors formed by adding amounts of three primaries • Primaries are RGB colors • Used by monitors • Black background	
Subtractive Colour • Form a color by filtering white light • Filters are Cyan, Magenta and Yellow • Used by printers • White background	

OpenGL

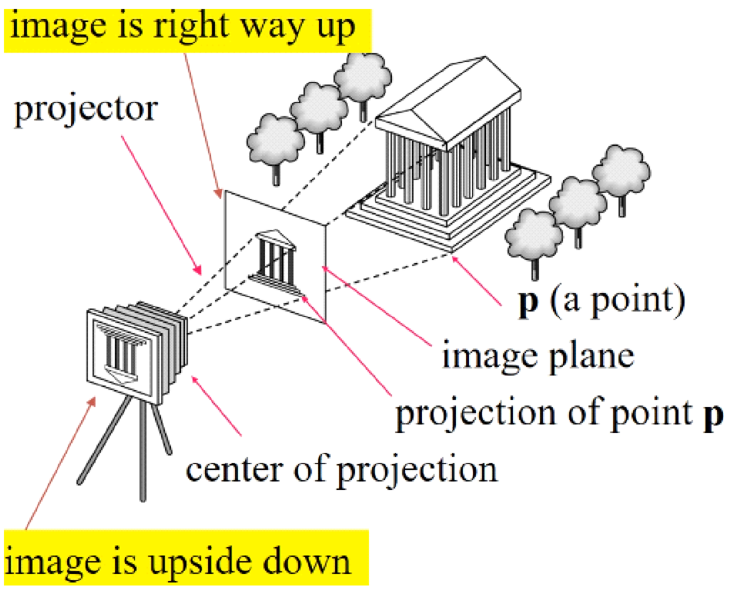
OpenGL is a *platform independent graphics API* (Application Programmer's Interface) that:

- Gets close enough to the hardware to get excellent performance and real-time rendering
- Focuses on rendering
- Is easy to use
- Provides a link between the low-level graphics hardware and the high-level application program that you write

Synthetic Camera Model

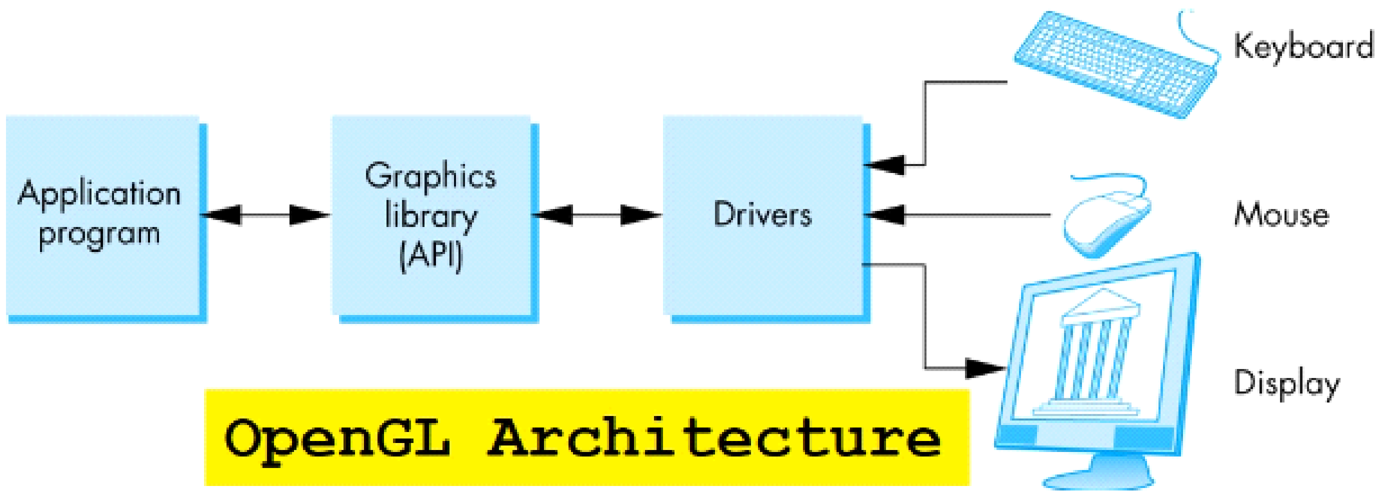
OpenGL uses the **synthetic pin hole camera model**

We use an imaginary plane to turn the image the right way up



Architecture

Wednesday, November 20, 2019 3:19 PM



Versions

Early versions came with default shaders

Latest versions are completely **shader-based**

- No default shaders
- Each application must provide both a vertex and a fragment shader
    - i.e. you must additionally write these shader programs

OpenGL ES

- The ES indicates that it is suitable for Embedded Systems
- V1.0 is a simplified version of OpenGL 2.1
- V2.0 is a simplified version of OpenGL 3.1

OpenGL 4.1 and 4.2 additionally have geometry shaders and tessellation.

OpenGL 4.6 (2017) is the latest version

WebGL

WebGL is not included in the curriculum

WebGL
- Is a derivative of OpenGL ES version 2.0
- Provides JavaScript bindings for OpenGL functions, allowing an HTML page to render images using any GPU resources available on the computer where the web browser is running

Libraries

OpenGL Core Library

- Provides basic functionality (e.g. primitives)
- OpenGL32 on Windows
- GL on most Unix/Linux systems (libGL.a)

OpenGL Utility Library (GLU)

- Provides functionality in OpenGL core but avoids having to rewrite code
- Can define textures and mip maps
- Will only work with legacy code

Links with Windowing Systems

- GLX for X Window systems
- WGL for Windows
- AGL for Macintosh

OpenGL Utility Toolkit (GLUT)

- Provides functionality common to all window systems
  - Open a window
  - Get input from mouse and keyboard
  - Menus
  - Event-driven
- Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
  - No slide bars
- Was created long ago and has been unchanged - has deprecated functions
- FreeGLUT updates GLUT, and has more capabilities (e.g. context checking)

OpenGL Extension Wrangler (GLEW)

- Makes it easy to access OpenGL extensions available on a particular system
- It avoids having to have specific entry points in Windows code
- The application just needs to include `glew.h` and run a `glewinit()`

Usage Process

1. Decide on your objects, lights and camera
2. Work out which OpenGL functions are required
3. Include the libraries that contain your required functions

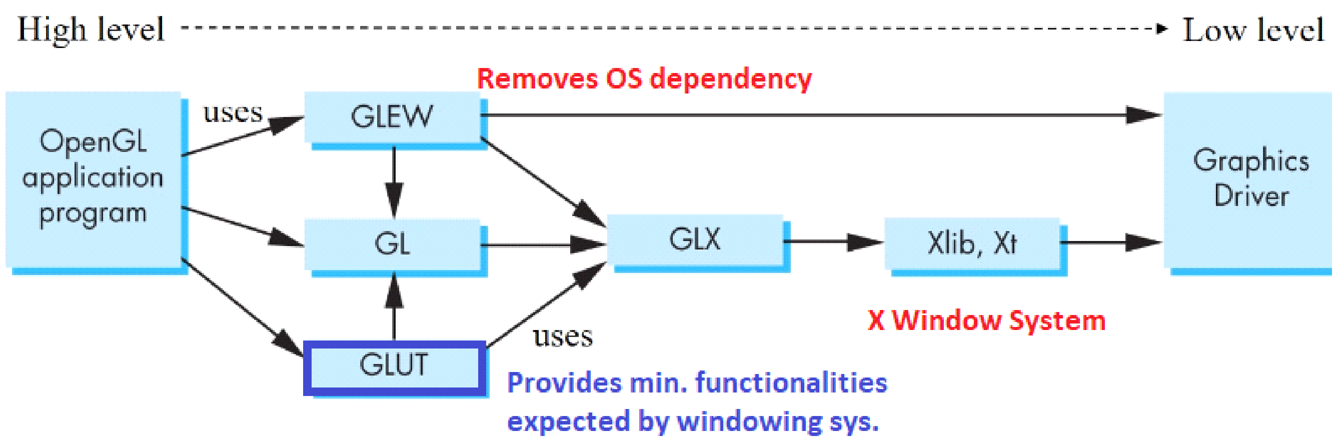
These are all practical issues and you will have the OpenGL documentation to help you

Modern OpenGL

- Allows programs to achieve fast graphics performance by using the **GPU rather than the CPU**
- Allows applications to *control the GPU through shader programs*

It is the application's job is to send data to GPU - which then does all the rendering

Software Organization



- The application programs can use GLEW, GL, GLUT functions but not GLX etc. ones
- The lower level parts are specific to operating systems and drivers
- Programs can be recompiled with GLUT libraries for other operating systems

Data Types

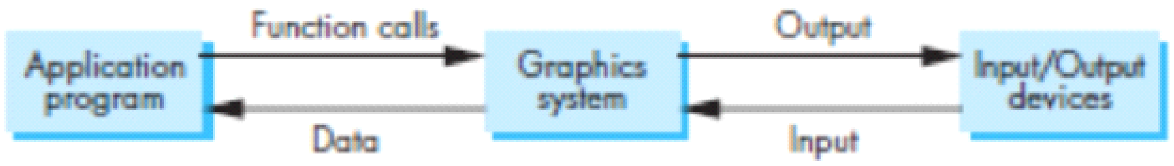
In OpenGL, we use basic OpenGL types such as:
GLfloat, GLdouble, GLint, etc. (equivalent to C/C++ equivalents)

Additional data types are supplied in header files vec.h and mat.h from Angel and Shreiner:
vec2, vec3, mat2, point2, point3, etc.
(These are vector points, matrices etc.)

OpenGL Functions

Friday, 14 February 2020 7:26 AM

State Machine



Graphics system as a black box

We can think of the entire graphics system as a black box that contains a finite-state machine.

Input

- Comes from the application program
- Can change the state of the machine
- Can cause the machine to produce a visible output.

Two Function Types

From the API perspective, there are two types of graphics functions

1. Functions that **define primitives** that flow through the state machine:
   - Define how vertices are processed
   - Can cause output if the primitive is visible
2. Functions that **change the machine's state** or return state information:
   - Transformation functions
   - Attribute functions
   - In OpenGL, most state variables are *defined by the application and sent to the shaders*

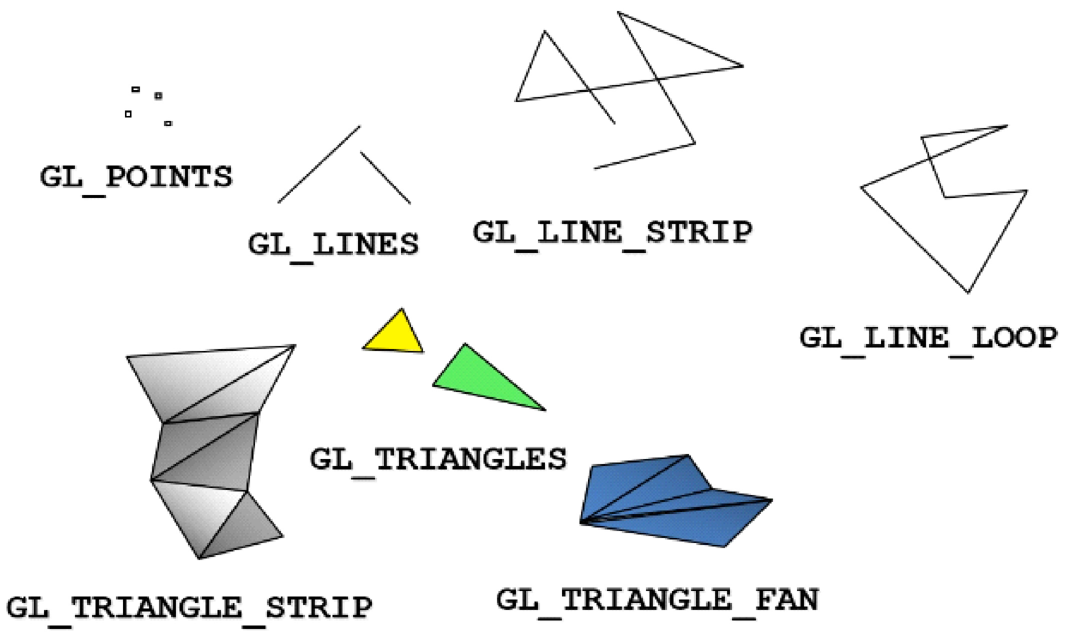
Due to this FSM implementation, most **parameters are persistent** - e.g. when we set the color, it becomes the current drawing color for everything below

Functions

OpenGL provides a range of functions for specifying:

- Primitives
  - Points
  - Line Segments
  - Triangles
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
- Input (GLUT)
- Query

Primitives


GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN

Attributes

Attributes are **properties associated with the primitives** that give them their different appearances

- Color (for points, lines, polygons)
- Size and width (for points, lines)
- Stipple pattern (for lines, polygons)
- Polygon mode
  - Display as filled: solid color or stipple pattern
  - Display edges
  - Display vertices

Note: glPointSize - The actual size that is displayed on the screen will be different

Lack of Object Orientation

OpenGL is **not object oriented** so that there are multiple functions for a given logical function.

The following are the same function but for different parameter types:

- glUniform3f
- glUniform2i
- glUniform3dv

The underlying storage mode is the same

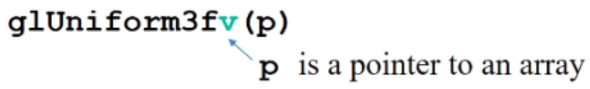
It is easy to create overloaded functions in C++ but the issue is efficiency

Format

glUniform3f(x,y,z)

- Belongs to core GL library
- Function name
- This is a function that **defines a uniform variable** for the current program object
- Uniform variables are used to *communicate with your vertex/fragment shaders* from outside
- Dimensions
- Float data type

A variant of the same function:



```
glUniform3fv(p)
```

p is a pointer to an array

OpenGL Constants

Most constants are defined in the include files
`gl.h`, `glu.h` and `glut. h`

`#include <GL/glut.h>` should automatically include the others

`gl.h`

- Has OpenGL primitives (e.g. GLfloat)
- Has functions like `glEnabIe` and `glCIear`

GLSL

GLSL = OpenGL Shading Language

- C-like language
  - Built-in Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Supports loops and if-else statements, but no recursion is allowed
- Similar to Nvidia's Cg and Microsoft HLSL

GLSL programs are run using a OpenGL program

Code sent to shaders as source code

New OpenGL functions to compile, link and get information to shaders

OpenGL Programs

Most OpenGL programs have a similar structure that consists of the following functions:

- **`main()`**
  - Creates the window
  - Calls the init() function
  - Specifies callback functions relevant to the application
  - Enters event loop in the last executable statement
- **`init()`**
  - Defines the vertices, attributes, etc. of the objects to be rendered
  - Specifies the shader programs
- **`display()`**
  - This is a callback function that defines what to draw whenever the window is refreshed (e.g. when the window is opened/changed/dragged)
  - Every glut program must have a display callback

Example

`simple.cpp` generates a white square on a black background

Start

```
// Include GLUT header file
// Automatically includes (core) gl.h
# include <GL/glut.h>

// Textbook header file
#include "Angel.h"

// Set 'std' as the default namespace look in
// (e.g. std::cout becomes cout)
using namespace std

// Triangles are better to use, so we will use them to make the square
const int numTriangles = 2;

// Each triangle has 3 vertices
const int numVertices = NumTriangles * 3;

// An array of vec3 coordinates. The third dimension is zero as we need a
flat square. This shows how 2D graphics is a special case of 3D graphics
vec3 points[NumVertices] = {
    vec3( -0.5, -0.5, 0.0), vec3( 0.5, -0.5, 0.0), vec3( -0.5, 0.5, 0.0),
    vec3( 0.5, 0.5, 0.0 ), vec3( -0.5, 0.5, 0.0), vec3( 0.5, -0.5, 0.0 )
};
```

Initialization

```
void init(void)
{
   // We create objects in three steps in OpenGL
   // 1. Make variable holder -> 2. Define object type -> 3. Bind object

   // Create a vertex array object
   GLuint vao; // Made GL unsigned integer holder
   glGenVertexArrays(1, &vao); // Object type is now Vertex Array
   glBindVertexArray(vao); // Bind

   // Create and initialize a buffer object
   GLuint buffer; // Holder
   glGenBuffers(1, &buffer); // Type
   glBindBuffer(GL_ARRAY_BUFFER, buffer); // Bind

   // We create a buffer of the size we need to store the six points
   glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);

   // The shader programs are loaded/compiled to give a Gluint output
   GLuint program = InitShader("vertex.glsl", "fragment.glsl");

   // OpenGL will now use the shader program for shading
   glUseProgram(program);

   // Initialize the vertex position attribute from the vertex shader
   // "vPosition" is reference to the vertex.glsl program
   GLuint vPos = glGetAttribLocation(program, "vPosition");
   glEnableVertexAttribArray(vPos);

   // GL_FLOAT = Type of vertex positions
   // GL_FALSE = The data does not need normalization
   // 0 = There is no stride. Every point is drawn
   // BUFFER_OFFSET(0) = We are drawing from the start, so no offset
   glVertexAttribPointer(vPos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

   // Sets background to black (RGB values then opacity)
   glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

Vertex Shader

```
// Matches the name chosen in simple.cpp
// (i.e. the 2nd parameter passed to the glGetAttribLocation function)
attribute vec4 vPosition;

void main()
{
    // gl_Position is an inbult variable in GLSL that holds the vertex postions
    // It denotes the vertex co-ordinates in 4 dimensions
    gl_Position = vPosition;
}
```

Fragment Shader

```
// The fragment shader defines the color in which the fragments will be drawn in
void main()
{
   // gl_FragColor is an inbuilt variable in GLSL
   // It holds the colour as a 4D vector
   // We input white, so a white square is drawn on the black background
   gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0);

}
```

Display and Main

```
void display(void)
{
    // Clear the buffer
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw the two arrays as triangles
    // 0 = Specifies the starting index in the enabled array
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    // Draw now and flush the buffers
    glFlush();
}

int main(int argc, char ** argv)
{
    // Initialize glut
    glutInit( &argc, argv );

    // Define display mode
    glutInitDisplayMode(GLUT_RGBA);

    // Set window size
    glutInitWindowSize( 256, 256 );

    // Create a window with "simple" as its title
    glutCreateWindow ("simple");

    // Set up OpenGL state and initialize shaders
    init();

    // The previous 'display' function will be a callback
function,
    // meaning it will be called every time the window is
   refreshed.
    glutDisplayFunc(display);

    // Enter infinite event loop
    // If the window changes in any way, it will redraw it
    glutMainLoop();
}
```