# Lab 1 - Introduction to OpenGL

In this lab we are going to create a very basic fixed function pipeline OpenGL 1.1 Application. This is a gentle introduction to OpenGL and also the cross-platform SDL Library(this will allows us to create a window, handle input events and other bits of functionality to create a game).

Remember to regularly commit and sync your code with GitHub during the course of the Lab session.

## 1. Create your GitHub Repo

Create a new GitHub Repository called **GP2Labs-*yourusername*** make sure you pick a **gitignore** for Visual Studio.

You should now clone this repository using GitHub for Windows/Mac, navigate to the directory.

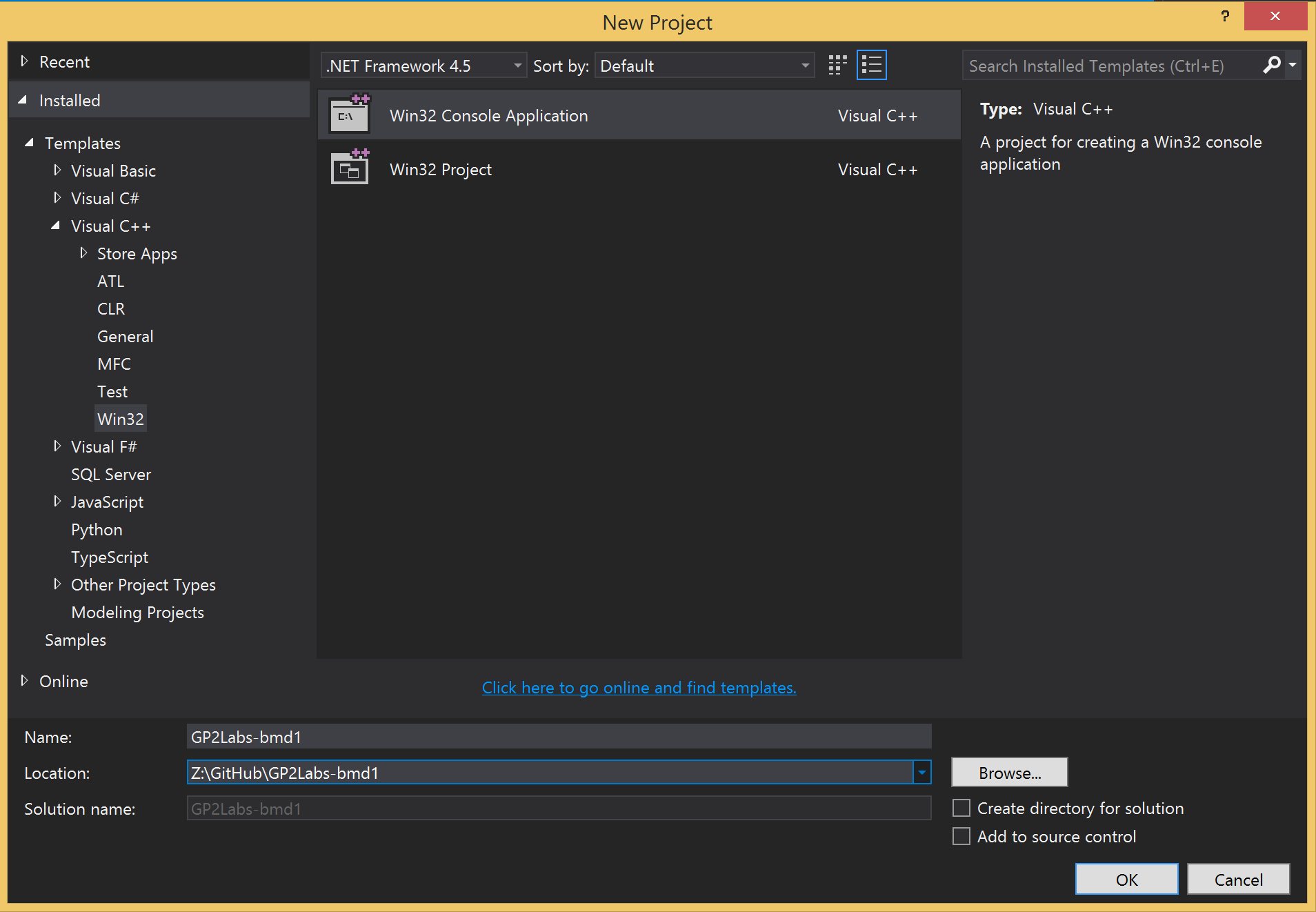
If you don’t know how to create a Repository on GitHub you should follow the instructions in the “**GitHub Exercise**” document on GCULearn.

You should now commit and sync your codebase. At this point you should ask the Lab Demonstrator for help if you are unable to sync your project.

## 2. Create Basic Project

Open up Visual Studio 2013 and create a **Visual C++** -> **Win32** -> **Console Application**

**Don’t hit** OK yet, make sure you change the location to the Github directory/GP2Labs-yourusername. The name of the project should be **GP2Labs-*yourusername***



Click on the **OK** buttonand you will be taken to the Win32 Application Wizard, click on the **Next** button**, UNTICK** **Precompiled header** & **Security Development Life(SDL) checks** and finally **TICK** **Empty Project**. You can now click finish.  
  
One you have done this, switch over to GitHub for Windows to commit and sync your changes. You should always make sure that you enter a meaningful commit message.

## 3. Create Basic Application Framework

We are now going to add our “main” file, this will contain the entry point to our application and will also be used to initialize a Window and setup OpenGL.

**Right Click** on the project and select **Add**->**New Item**, select **C++ File**. In the Dialog box that pops up ensure that the name of the file is **main.cpp**, this will add an empty source file to the project.

### 3.1 Adding IOStream header

At the top of the file add the following code snippet

//Header files

#include <iostream>

//Global variables go here

This includes a header file which contains Input/Output streams which allows to output information to the console(or output window). This is a very useful feature for debugging and making sure everything is in good working order.

### 3.2 Adding Window Initialisation Function

We are now going to add a function which will be called to create a window, add a couple of new lines below the **Global variables** comment and add the following code snippet

//Global functions

void InitWindow(int width, int height, bool fullscreen) {

}

This function takes in two integers(window width & height) and a boolean(determines if the window should be fullscreen or windowed). The body of the function will be filled out in **Step 4 – Initialise Window**

### 3.3 Adding Cleanup Function

Now we should add a function, which will be called to gracefully cleanup our application, add the following code snippet just a few lines after the above **InitWindows** Function

void CleanUp() {

}

This function will be called when we are exiting our Game Loop(see point 6) and will be used to cleanup any memory allocated in the exercise.

### 3.3 Main Function

The main function is the entry point to all standard C++ Applications, this the first function called by the Operating System when you application is launched. From this entry point we can call our own functions & create instances of objects.

Add the following code snippet a few lines after the cleanup function.

//Main Method - Entry Point

int main(int argc, char \* arg[]) {

return 0;

}

The entry point receives two variables; an interger which tracks the number of command line arguments sent to the application and a pointer to array of characters(strings) which contain the command line arguments. The function returns a integer which usually determines if the application has run successfully(zero) or unsuccessfully(negative).   
  
A quick word about command line arguments, the arguments are passed to the entry point in an OS dependent manner. For example, in Windows you pass the arguments in via a shortcut.

## 4. Link to SDL Library

The SDL2 library is a cross platform multimedia library which is widely used in the Games Industry to create Games for multiple platforms. For example Valve have adopted SDL for the Source Engine so that their titles can work on Mac, Linux and Windows.

SDL2 contains all the functionality we need to create a game including Window creation, User Input support(keyboard, mouse, joypad & touch), basic image loading, 2D Graphics and low level audio support. On top of all these features it interfaces very well with most Graphical APIs and has built in support to easily create OpenGL contexts(for access to graphics pipeline).

If you don’t know how to link to an SDK/Library you should read the following document on GCU Learn – **Linking to SDKs**

### 4.1 SDL2 Include(header) Directory

You should edit the **Additional Include Directories** setting of the Project in Visual Studio so it points to the include directory of SDL2, this directory should be **C:\SDL2-2.0.3\include** in the lab machines(your home setup may be different but I recommend that you change it to match the lab)

### 4.2 SDL2 Library Directory

You should now edit the **Additional Library Directories** properties of the Project in Visual Studio so that it points to the directory which contains the SDL2 libraries, this should be **C:\SDL2-2.0.3\lib\x86**  in the lab

### 4.3 SDL2 Libraries

We now need to link against the SDL2 libraries, add **SDL2.lib** & **SDL2main.lib** to the **Additional Dependencies** property of the project. Lastly you should copy the **SDL2.dll** from C**:\SDL2-2.0.3\lib\x86**  to the Debug folder of your project(you may need to copy this to the release folder if you build a release version of your project)

## 5. Initialise SDL2, Create Window and Cleanup

We are now ready to create our SDL2 Window, this is actually fairly easy compare to creating a Win32 Windows.

### 5.1 Add Include statement for SDL2

In Visual Studio move to the top of the Win32 Window and add the following code snippet just after the include statement for **iostream**

//header for SDL2 functionality

#include <SDL.h>

### 5.2 Window Global variable & Constants

We now need to add a variable that will point to our created window and additionally we will add a couple of constants which will make it easier to control the Width and Height of our window.

Add the following code snippet just after the **Globals go here!!** Comment

//Pointer to our SDL Windows

SDL\_Window \* window;

//Constants to control window creation

const int WINDOW\_WIDTH = 640;

const int WINDOW\_HEIGHT = 480;

### 5.3 Filling out our Window Creation function

Navigate to the **InitWindow** function and inside the curly braces( **{** & **}**) add the following code to create a window

//Create a window

window = SDL\_CreateWindow(

"Lab 1", // window title

SDL\_WINDOWPOS\_CENTERED, // x position, centered

SDL\_WINDOWPOS\_CENTERED, // y position, centered

width, // width, in pixels

height, // height, in pixels

SDL\_WINDOW\_OPENGL // flags

);

Most of the variables passed into this function should be self explanatory, the last variable may require a bit more explanation. This **flags** value can be any value of SDL\_WindowFlags(<https://wiki.libsdl.org/SDL_WindowFlags>), these flags control how our window is created(fullscreen, support for OpenGL etc etc). These values can be OR'd together, this will allow us to create a fullscreen Window that supports OpenGL(SDL\_WINDOW\_FULLSCREEN|SDL\_WINDOW\_OPENGL)

### 5.4 Initialise SDL2

Before we can call the above function we need to first initialise the SDL Library, this is a prerequisite before we call any SDL functions. Navigate to the **main** function and just after the **opening** curly brace add the following code snippet

// init everything - SDL, if it is nonzero we have a problem

if (SDL\_Init(SDL\_INIT\_EVERYTHING) != 0)

{

std::cout << "ERROR SDL\_Init " << SDL\_GetError() << std::endl;

return -1;

}

This shouldn't be too dificult to understand, this will initialise the whole of the SDL library. Check the following page on the SDL2 docs for an explanation of the init flags passed into **SDL\_Init** function**-** <https://wiki.libsdl.org/SDL_Init>

### 5.5 Calling our Window Creation function

Now that SDL2 has been initialised we can call our **InitWindow** function, just after the closing curly brace of the above if statement, add the following code

You should notice that we pass in the WINDOW HEIGHT and WIDTH constants to control the dimensions of the windows, at the moment the last parameter(controls window mode) is not used so we pass in false.

InitWindow(WINDOW\_WIDTH, WINDOW\_HEIGHT, false);

### 5.6 Cleanup

Our **InitWindow** functionallocates some memory, the other issue you need ot be aware of is that the initialisation of SDL also allocates some memory and needs to be gracefully shutdown in order to reclaim this memory.

Navigate to the actual **InitWindow** function(outside **main** function) and just after the closing curly brace add the following function.

//Used to cleanup once we exit

void CleanUp()

{

SDL\_DestroyWindow(window);

SDL\_Quit();

}

The call to SDL\_DestroyWindow will clean up memory allocated by the call to create the window and the SDL\_Quit will cleanup any memory allocated when the SDL Library is initialised. The other thing that should be highlighted is that we carry out deletation in reverse order of creation.

We are now ready to call our **CleanUp** function, navigate to the **main** function and add the code snippet just after the call to **InitWindow** and before **return 0;** line.

CleanUp();

Build and run the application, you should see the Window open and then close very quickly, the reason this occurs is that the Window dosen’t block code execution. We will need to add a **while** loop to control the lifetime of the application, in Games this loop is called the **Game Loop**.

## 6. Game Loop

As mentioned in the previous section, the Game Loop will control the lifetime of our application and should in essence keep looping until the user quits the games. In our case we are going to wait until the user clicks the close button on the window(in future labs we will change this to when the use selects a quit option on a menu or in a popup dialogue box).  
  
6.1 Running - Global Variable

We require a global variable which will control the Game Loop, the best option for this will be a Boolean variable, this variable should be initially set to **true** but when we are ready to exit the game we should set it to **false.**

Navigate to the top of **main.cpp** file and add the following next to the other global variables(WINDOW\_WIDTH & WINDOW\_HEIGHT)

bool running = true;

### 6.2 SDL Event Variable

Most Window based applications(Linux, Win32, Mac) are event driven, this means that during an applications lifecycle events will be generated by the Users interaction with the window or other components of the application(button, text box, etc). The SDL library has an Event structure(a class with only variables) which encapsulates all the events that can be generated by interacting with an SDL Window. We need to add a variable of type **SDL\_Event** which will be used to hold the event data generated by our Window.  
  
Navigate to the **main** function and add a couple of new lines between the calls to **InitWindow** and **Cleanup** functions. Now add the following code snippet in the gap you have just created to create an instance of **SDL\_Event** called event.

SDL\_Event event;

### 6.3 GameLoop

We are now going to entre our Game Loop, to do this you should add a **while** loop and use the running variable as the condition for the loop. Insert the following code snippet after the above the SDL\_Event variable

while (running)

{

}

}

}

### 6.4 Polling Events

Events in SDL like other Window System are added to an event queue, it is up to the programmer to decide how these events are handled. In SDL there is two ways to access these events, **Peek** and **Poll**. **Peek** is used to check the event but will not remove it; **Poll** will check the event but will also remove it from the queue. **Poll** is the preferred method of handling events in a real time system such as Games, the reason for this is that the event is guaranteed to be removed and not clog up execution of the application.

Add the following code just inside the above while loop(in-between the curly braces)

while (SDL\_PollEvent(&event)) {

}

The above snippet will poll the event queue(SDL\_PollEvent), if there is nothing in the queue then the function will return 0. If there are events in the queue then the function will return a positive value. The function will also fill out the **SDL\_Event** variable with the details of the event.

### 6.5 Handling Events

It is now up to use how we handle the event, the kind of events we can receive are Window Events(close, resize, move, minimize and maximize), keyboard, mouse & joypad events. In the above the **SDL\_PollEvent** while loop, we need to check what type of event we are getting, respond to it and optionally retrieve values from the result.  
  
Add the following code snippet to the above **SDL\_PollEvent** while loop

//Get event type

if (event.type == SDL\_QUIT || event.type == SDL\_WINDOWEVENT\_CLOSE) {

//set our boolean which controls the game loop to false

running = false;

}

This snippet checks the event type and if the type is **SDL\_QUIT** or **SDL\_WINDOWEVENT\_CLOSE** then we set the **running** Boolean to false. These events are generated when the user clicks on the close button on the top right of the window.

Build and run the application, you should see a window appear on the screen. Click on the close button and the SDL Window should close.

## 7. Link to OpenGL

All the above setup has now got us to the point where we can start working with OpenGL. The thing to understand about the OpenGL SDK is that it comes packaged with the Platform SDK for Windows(which is installed with Visual Studio 2013) and is automatically added to the System Path of the Operating System. This means we don’t need to add the Additional Include & Libraries to the project properties in Visual Studio 2013. We still need to link to the libraries though.

### 7.1 OpenGL Libraries

We now need to link against the **OpenGL** libraries, add opengl32.lib & glu32.lib to the Additional Dependencies property of the project.

**You don’t need to copy any dlls to the debug or release folder as these are accessible via the System Path.**

## 8. OpenGL Initialisation and Cleanup

Now we are ready to initialise OpenGL, to do this we are going to create function which encapsulate the initialisation of OpenGL, in addition to this we are going to create Utility function which will be used to calculate the size of the viewport we are going to be rendering too, and finally we will create two functions which will update the state of our application and a render function which will draw to the screen.

### 8.1 OpenGL Header files

We need to add the include the statements for OpenGL, we require the SDL OpenGL header which contains Functions and Objects to handle the interface between OpenGL and SDL, this also contains all the functionality for working with OpenGL. We also require the OpenGL Utility header.

Navigate to the top of **main.cpp** and add the following include statement **just after the SDL include statement**

#include <SDL\_opengl.h>

#include <gl\GLU.h>

### 8.2 OpenGL Context Global Variable

The SDL OpenGL Context, while not used directly is needed in circumstances but is a prerequisite before we start interacting with OpenGL.

### 8.3 Our OpenGL Init and Viewport Functions

Navigate to the top of the main.cpp file and add the following functions after cleanup function.

//Function to initialise OpenGL

void initOpenGL()

{

}

//Function to set/reset viewport

void setViewport( int width, int height )

{

}

We will implement and call these functions in the next section, the reason we have a separate viewport function as we may want to change the viewport size(for example when the screen is resized).

### 8.4 Implementing our Init OpenGL Function

This function will create SDL OpenGL Context, while not used directly is needed in circumstances but is a prerequisite before we start interacting with OpenGL. Once the context has been created we will then setup some initial OpenGL states.

Inside the **initOpenGL** function add the following code snippet to create the SDL OpenGL Context and check for any errors.

//Create OpenGL Context

glcontext = SDL\_GL\_CreateContext(window);

//Something went wrong in creating the context, if it is still NULL

if (!glcontext)

{

std::cout << "Error Creating OpenGL Context " << SDL\_GetError() << std::endl;

}

We now need to add some code to set some initial OpenGL states, add the following code just after the **error checking if statement** inside the **initOpenGL** function

//Smooth shading

glShadeModel( GL\_SMOOTH );

//clear the background to black

glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );

//Clear the depth buffer to 1.0

glClearDepth( 1.0f );

//Enable depth testing

glEnable( GL\_DEPTH\_TEST );

//The depth test to use

glDepthFunc( GL\_LEQUAL );

//Turn on best perspective correction

glHint( GL\_PERSPECTIVE\_CORRECTION\_HINT, GL\_NICEST );

### 8.5 Implementing our Viewport Function

There is one last bit of setup we need to do before we can start using OpenGL. We require to define a Viewport Transformation and Perspective Transformation(These will be defined in more detail during Lecture 2).

Add the following code to the **setViewport** function to setup the Viewport Transformation

//screen ration

GLfloat ratio;

//make sure height is always above 0

if ( height == 0 ) {

height = 1;

}

//calculate screen ration

ratio = ( GLfloat )width / ( GLfloat )height;

//Setup viewport

glViewport( 0, 0, ( GLsizei )width, ( GLsizei )height );

Now after the call to the **glViewport** add the following code to setup the Perspective Transformation

//Change to poject matrix mode

glMatrixMode( GL\_PROJECTION );

glLoadIdentity( );

//Calculate perspective matrix, using glu library functions

gluPerspective( 45.0f, ratio, 0.1f, 100.0f );

//Swith to ModelView

glMatrixMode( GL\_MODELVIEW );

//Reset using the Indentity Matrix

glLoadIdentity( );

### 8.6 Calling our functions

We are now ready to call our setup functions, navigate to the **main** function and after call to **InitWindow** add the following code

//Call our InitOpenGL Function

initOpenGL();

//Set our viewport

setViewport(WINDOW\_WIDTH, WINDOW\_HEIGHT);

### 8.7 Cleanup OpenGL

In the creation of Context we have allocated some memory to handle OpenGL functionality. Navigate to the **Cleanup** function and add the following code to delete the SDL OpenGL Context **before** the call to **Destroy** the **Window**

SDL\_GL\_DeleteContext(glcontext);

Now make sure you build, commit and sync the code at this point

## 9. Drawing

### 9.1 Update and Render functions

We are going to add a couple of functions which will be used to update the state of the Game and then draw(render) anything to the screen.

Open **main.cpp**  and add the following function stubs near the other global functions

//Function to draw

void render()

{

}

//Function to update game state

void update()

{

}

The update function will remain empty at the moment

### 9.2 Render Function

Inside the Render function, add the following code to clear the screen to black.

//Set the clear colour(background)

glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );

//clear the colour and depth buffer

glClear( GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT );

//require to swap the back and front buffer

SDL\_GL\_SwapWindow(window);

The first line will set the Clear Colour State(**glClearColour**), the **glClear** will do the actual clearing(in this case we are clearing the depth buffer and the colour buffer), finally we have **SDL\_GL\_SwapWindow** which swap the back and front buffer and display everything to the screen. One thing to note, is that all drawing has to be done between **glClear** and **SDL\_GL\_SwapWindow**.

### 9.3 Calling Update and Render functions

We are now ready to call our update and render function, we should call these functions when there is now events in the SDL Event queue, you should place the following code before the last curly brace of the **while(running)** loop.

update();

render();

If you build and run the application, you should see a black screen. Can you change the colour of the screen?  
  
Make sure you change it back to black!

### 9.4 Drawing a Triangle

We are now at a stage where we can draw something, add the following code between the **glClear** and **SDL\_GL\_SwapWindow** function in the render function.

//Switch to ModelView

glMatrixMode( GL\_MODELVIEW );

//Reset using the Indentity Matrix

glLoadIdentity();

//Translate to -5.0f on z-axis

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

//Begin drawing triangles

glBegin(GL\_TRIANGLES);

glColor3f(1.0f, 0.0f, 0.0f); //Colour of the vertices

glVertex3f(0.0f, 1.0f, 0.0f); // Top

glVertex3f(-1.0f, -1.0f, 0.0f); // Bottom Left

glVertex3f(1.0f, -1.0f, 0.0f); // Bottom Right

glEnd();

Lets look at each one of these calls

**glMatrixMode(GL\_MODELVIEW);**

This switches the matrix mode state to the Model View mode, this mode is used to position our vertices in 3D space. We will often carry out a rotation, translation or scaling after this call.

**glLoadIdentity();**

This will push the identity matrix onto the current matrix(in this case ModelView , see above). The identity matrix is like setting a value to 1(a gross simplification but will do for now).

**glTranslatef(0.0f,0.0f,-5.0f);**

Will push a translation matrix onto the current matrix, in this case it will translate the current matrix -5.0f units.

**glBegin(GL\_TRIANGLES);**

This begins the drawing process, we pass in the primitive type we are going to draw.

**glColor(1.0f,0.0f,0.0f);**

This sets the colour of the vertices where each component is a floating point number(0.0f – 1.0f) and represents red, green and blue.

**glVertex3f(1.0f,0.0f,0.0f);**

This specifies a Vertex using an x. y and z positional values.

**glEnd();**

This ends the drawing process

# Exercise

1. Change the triangle to a right angle triangle
2. Give each vertex a different colour
3. Draw 2 triangles
4. Make the triangles two different colours
5. Add global values to position each triangle
6. Move these positions with a keypress

# Additional Reading

**OpenGL SDK Docs** - <http://bit.ly/GP2-SDK-Docs>

**SDL2 SDK Docs** - <http://bit.ly/GP2-SDL2-Docs>

**Game Development with SDL 2.0 -** <http://bit.ly/GP2-SDL2DevTalk>