OpenGL

# Pre-requisite:

1. GLFW library: download it from <https://www.glfw.org/> (download the precompiled binaries for your platform)
   1. Documentation: <https://www.glfw.org/documentation.html>
   2. <https://www.glfw.org/docs/latest/>
   3. The architecture of the binaries depends on the architecture of your application.
2. Visual Studio – IDE for C++ development
3. OpenGL functions documentation: <http://docs.gl/>

# Setting up Visual studio solution for the OpenGL programming:

1. Create an empty visual C++ general project.
2. Setting up GLFW library in your project
   1. Create a dependencies/GLFW/ folders in your solution dir
   2. Open project properties in VS
   3. All Configurations – Win32
3. C++ Settings:
   1. Additional Include directories: $(SolutionDir)dependencies/GLFW/include
4. Linker Settings:
   1. Additional Library directories: $(SolutionDir)dependencies/GLFW/vs-2012/
5. Additional Dependencies: glfw3.lib;opengl32.lib;user32.lib;gdi32.lib;shell32.lib
6. Build and check if your configuration is correct

# Chapter 1: Getting started with OpenGL

You need to test if the openGL using GLFW is working properly. For this the boiler plate code:

#include <GLFW/glfw3.h>

int main(void)

{

GLFWwindow\* window;

/\* Initialize the library \*/

if (!glfwInit())

return -1;

/\* Create a windowed mode window and its OpenGL context \*/

window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

if (!window)

{

glfwTerminate();

return -1;

}

/\* Make the window's context current \*/

glfwMakeContextCurrent(window);

/\* Loop until the user closes the window \*/

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

/\* Swap front and back buffers \*/

glfwSwapBuffers(window);

/\* Poll for and process events \*/

glfwPollEvents();

}

glfwTerminate();

return 0;

}

This boiler plate code should open a window for you with the title “Hello World”. You can further test this, by drawing a triangle on this window using the classical/legacy openGL method. For this enter the following code after glClear()

//using legacy openGL way of drawing triangles

glBegin(GL\_TRIANGLES);

glVertex2f(-0.5f, -0.5f);

glVertex2f(0.0f, 0.5f);

glVertex2f(0.5f, -0.5f);

glEnd();

This will make sure that the GLFW and openGL is setup in your development environment correctly.

Basic workflow of OpenGL:



# Chapter 2: How to use the OpenGL specifications that is implemented by the graphics driver?

Note: In chapter 1, we did not use any function that was part of graphics driver’s implementation for the rendering using the openGL specification.

Every operating system has their own rendering engine and graphics library that they implement but using them means that the application will not be cross platform and we shall not be using the specifications of openGL.

The specifications of OpenGL are implemented in the graphics driver by the manufacturers of graphics card. So to use the openGL specified functions, it is required to access the dll(s) which contains all the implementations, provide function pointers to such functions/constants/macros etc. to the developers so that these functions can be used to render objects on the graphics window.

GLEW is one such library which does this internal linking (which is OS and driver specific) and providing the requested pointers without going to pain of collecting these pointers manually before using.

Download Glew: [http://glew.sourceforge.net/](https://www.youtube.com/redirect?v=H2E3yO0J7TM&redir_token=LnOanrWIsKWvUqbfF_qy55qP2Rp8MTU0NjU5NDk3N0AxNTQ2NTA4NTc3&event=video_description&q=http%3A%2F%2Fglew.sourceforge.net%2F)

Add this glew library to your project:

1. Extract and copy the glew library to the dependencies folder of the solution.
2. Rename to shorter version : glew
3. Add include dir: $(SolutionDir)dependencies\glew\include\GL
4. Add include library dir: $(SolutionDir)dependencies\glew\lib\Release\Win32\
5. Add the lib to be linked: glew32s.lib
6. In the pre-processor definition add: GLEW\_STATIC

Read the documentation properly so that the initial setting up is not messed up. The above details will help you to avoid such configuration related mistakes.

glewInit should be used only after creating valid OpenGL context

In A/CAE, most of the implementation is from <http://www.qt-project.org/> the library which is similar to glew. Glew and QT are not compatible with each other. The macros from Glew are undefined when the QT is to be used.

#include <glew.h>

#include <GLFW/glfw3.h>

#include <iostream>

int main(void)

{

GLFWwindow\* window;

/\* Initialize the library \*/

if (!glfwInit())

return -1;

/\* Create a windowed mode window and it's OpenGL context \*/

window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

if (!window)

{

glfwTerminate();

return -1;

}

/\* Make the window's context current \*/

glfwMakeContextCurrent(window);

//glewInit() uses the valid OpenGL context. without creating the valid context, the glew returns error

//later you shall not be able to use the api(s) related to openGL

if (glewInit() != GLEW\_OK)

std::cout << "Error" << std::endl;

//printing the openGL driver version. its the driver version on your system

std::cout << glGetString(GL\_VERSION) << std::endl;

/\* Loop until the user closes the window \*/

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

//using legacy openGL way of drawing triangles

glBegin(GL\_TRIANGLES);

glVertex2f(-0.5f, -0.5f);

glVertex2f(0.0f, 0.5f);

glVertex2f(0.5f, -0.5f);

glEnd();

/\* Swap front and back buffers \*/

glfwSwapBuffers(window);

/\* Poll for and process events \*/

glfwPollEvents();

}

glfwTerminate();

return 0;

}

# Chapter 3: Using Modern GL way of rendering

Steps:

1. Create an OpenGL buffer
2. Binding the buffer to the context
3. Adding the data to the buffer
4. Draw function

Buffer – the memory block reserved in the vRAM of GPU

Shader – the program/block of code that is running on GPU.

All the graphics related work or extensive operations related work should be implemented in the shader so that the performance of computation is improved. GPUs are clocked at higher speeds compared to CPU. This way the extensive computation works can be moved to GPU to get quicker results and reduce the utilization of your CPU. This will internally help in increasing the performance of your application.

#include <glew.h>

#include <GLFW/glfw3.h>

#include <iostream>

int main(void)

{

GLFWwindow\* window;

/\* Initialize the library \*/

if (!glfwInit())

return -1;

/\* Create a windowed mode window and it's OpenGL context \*/

window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

if (!window)

{

glfwTerminate();

return -1;

}

/\* Make the window's context current \*/

glfwMakeContextCurrent(window);

//glewInit() uses the valid OpenGL context. without creating the valid context, the glew returns error

//later you shall not be able to use the api(s) related to openGL

if (glewInit() != GLEW\_OK)

std::cout << "Error" << std::endl;

//printing the openGL driver version. its the driver version on your system

std::cout << glGetString(GL\_VERSION) << std::endl;

//building the buffer

unsigned int buffer;

float positions[6] = {

-0.5f, -0.5f,

0.0f, 0.5f,

0.5f, -0.5f

};

//no of buffers required and the id of the buffers

//pointer to unsigned int

glGenBuffers(1, &buffer);

//binding the buffer to openGL

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

//adding the data to buffer

glBufferData(GL\_ARRAY\_BUFFER, sizeof(positions), positions, GL\_STATIC\_DRAW);

/\* Loop until the user closes the window \*/

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

/\*actual draw call to draw the traingle

this will draw the triangles, but you would see black screen

This is perfectly normal as nothing extra is specified like color, etc.

\*/

glDrawArrays(GL\_TRIANGLES, 0, 3);

/\* Swap front and back buffers \*/

glfwSwapBuffers(window);

/\* Poll for and process events \*/

glfwPollEvents();

}

glfwTerminate();

return 0;

}

Binding is very essential in OpenGL. It indicates draw function to use which data. Without binding the data, you cannot expect OpenGL to know which data it should use to draw/render on the window.

OpenGL works based on states. So before drawing anything on window, you need to specify which buffer data you want to use, for which context and which shader you want to execute for draw.

A good documentation source: <http://docs.gl/>

# Chapter 4: Vertex Attributes

Vertex – It is considered as an object containing lot of information such as position, color, texture, normal, etc. It is more than just a point having a position values.

Stride – amount of memory blocks to be moved to reach to the next vertex/attribute.

Pointer – offset between the attributes of a single vertex from where you want to start reading the data.

In OpenGL, when you are binding the data and specifying the vertex attributes, you need to architect the memory layout for your data. This is required because the same data structure needs to be used to interpret the results in the shader that you will be implementing (glVertexAttribPointer).

#include <glew.h>

#include <GLFW/glfw3.h>

#include <iostream>

int main(void)

{

GLFWwindow\* window;

/\* Initialize the library \*/

if (!glfwInit())

return -1;

/\* Create a windowed mode window and it's OpenGL context \*/

window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

if (!window)

{

glfwTerminate();

return -1;

}

/\* Make the window's context current \*/

glfwMakeContextCurrent(window);

if (glewInit() != GLEW\_OK)

std::cout << "Error" << std::endl;

unsigned int buffer;

float positions[6] = {

-0.5f, -0.5f,

0.0f, 0.5f,

0.5f, -0.5f

};

glGenBuffers(1, &buffer);

//binding the buffer to openGL

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

//adding the data to buffer

glBufferData(GL\_ARRAY\_BUFFER, sizeof(positions), positions, GL\_STATIC\_DRAW);

//setting up Vertex attribute

int stride = sizeof(float) \* 2; //amount of bytes for vertex

glVertexAttribPointer(0, 2, GL\_FLOAT, GL\_FALSE, stride, 0);

//enabling the vertex

glEnableVertexAttribArray(0);

/\* Loop until the user closes the window \*/

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

/\*actual draw call to draw the traingle

this will draw the triangles, but you would see black screen

This is perfectly normal as nothing extra is specified like color, etc.

\*/

glDrawArrays(GL\_TRIANGLES, 0, 3);

/\* Swap front and back buffers \*/

glfwSwapBuffers(window);

/\* Poll for and process events \*/

glfwPollEvents();

}

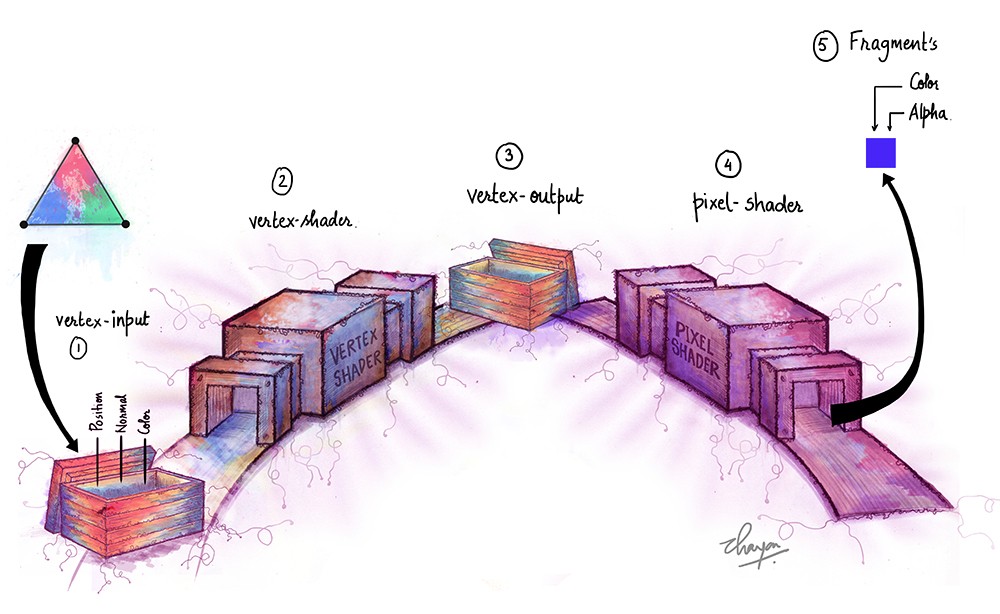
glfwTerminate();

return 0;

}

Note: Here, in this session the triangle seems to be rendered on your screen. This is because many GPU drivers come with default shader implementation. This is very specific to the GPU and its drivers.

# Chapter 5: Shaders - 1



Shaders – it’s the block of code which is running on GPU.

Shaders are of different types:

1. Vertex Shader
2. Fragment/Pixel shader
3. Tessellated shaders
4. And many more…

Shaders takes the data from the CPU, stores them on GPU and processes them and generates the values which are used for rasterization or showing up the pixels on the graphics window.

Vertex shader and fragment shaders are the primary shaders where the majority of work is done related to the computation of pixel position, lighting, color, orientation, etc.

**Vertex Shader:**

It is the first shader in the pipeline which collects the data from the CPU memory using the buffer that is initialized and makes it available in GPU. The basic task of vertex shader is to calculate the positioning of the pixel on graphics window. The calculation related to the transformations are to be done here. At the end of vertex shader, you end up with a wireframe model like structure. Location of all the pixels are finalized.

This means that vertex shader is called as many times as the vertices are available in the buffer. So for the triangle, vertex shader is called thrice.

**Fragment Shader:**

This version of shader is used to compute the color of an individual pixel that needs to be rendered on the screen. This means, that the shader is called for every pixel that will be rasterized/drawn on the graphics window. This also indicates that the fragment shader is an expensive operation compared to the vertex shader.

The color of the pixel is based on the lighting, texture, material, camera orientation and lot other manipulation. This shader should at the end compute the value of the color for individual pixel.

Check the source code: <https://github.com/ronak1009/learnOpenGL/tree/f1d5ec47d126d3d131b2d703601da7373e0a8c6d>

**Basic properties of Shaders:**

1. Programs compiling and running on GPU.
2. Starts by mentioning the version (#version 330 core)
3. List of inputs and outputs
4. main() function
5. Language of writing shaders: GLSL
6. Shaders are compiled, linked and binded to the context at run-time.
7. Vertex shader requires input data which is different fetched from the vertex data which is available from the buffer-data.
8. Fragment shader always need to give output in form of vec4 for the color. (r,g,b,alpha)
9. It’s possible to send data from one shader to another by specifying a particular structure for the output variable.