Tutorial – Introduction to OpenGL

In this session we will go over some of the steps needed to create our own OpenGL projects that allow us to experiment with modern OpenGL rendering techniques.

This tutorial will assume you are developing on a Windows-based computer using Visual Studio 2013 as your IDE. However, the code we will be using and writing is cross-platform and has been built on OSX and Linux. If you wish to work on a platform other than Windows with Visual Studio 2013 then you will have to make the necessary modifications to this tutorial.

Throughout this subject we will be making use of a few open-source third-party libraries that will make it easier for us to build OpenGL applications. For the moment we will just be using 3 third-party libraries and a class provided by AIE: **GLM** (http://glm.g-truc.net/0.9.6/index.html & https://github.com/g-truc/glm), **GLFW** (http://www.glfw.org/ & https://github.com/glfw/glfw), an OpenGL extension wrapper created by the **OpenGL Loader Generator** (https://bitbucket.org/alfonse/glloadgen/wiki/Home) and placed on the portal page under this session, and some additional code we have created for you at **AIE** (https://github.com/AcademyOfInteractiveEntertainment/aieutilities).

GLM is an OpenGL Mathematics library that has been developed to mimic the math capabilities of OpenGL's GLSL language. It is cross-platform and header-only, meaning we do not need to compile it. However, due to it needing to support many different platforms and IDEs that have different support for various C++ features (i.e. C++11/14) we may need to merge against a specific release of GLM. You will be advised by your teacher if this is necessary.

GLFW is an OpenGL framework library that can create a window for us to render in to and gives us access to other features, such as input.

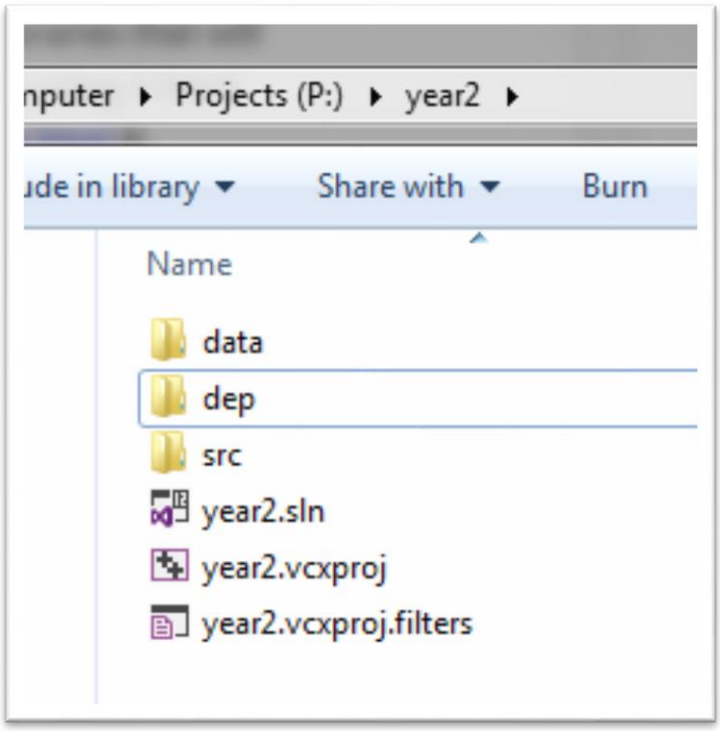
The **AIE** utilities github repository simply contains a single class for now called **Gizmos**. **Gizmos** is a stand-alone class that can easily display simple 3-D and 2-D primitives. We will need to include the .cpp and .h for it in to our projects.

Finally, OpenGL is a large library, with many different features, some of which are deprecated. We will make use of the **OpenGL Loader Generator** that can be used to give us a single header and C file that will only include features from the version we want, without deprecated items. As mentioned these are hosted on the portal page for your convenience.

Project Setup:

We first need to create our own OpenGL projects.
It is recommended you host your project on a version control server. **Github** (https://github.com) and **Bitbucket** (https://bitbucket.org) are both free services that are ideal.

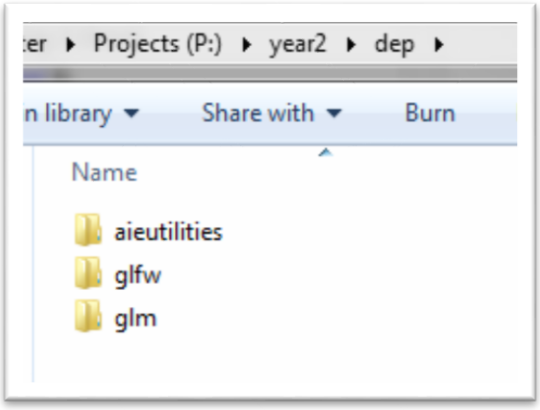
Create a new Console Application project and set up a suitable folder structure to work from. Setting up a good structure to begin with helps the project development. Below is a potential setup:



In this example we have a Solution and Project file located in the root folder. Within that is a **data** folder that would contain all of our program's art assets that we may use.

There is also a **src** folder that would contain the actual code for our project. You should place the loader generator files within this folder (**gl_core_4_4.h** and **gl_core_4_4.c**). We will create most of our code within this folder.

Also in the root folder is a **dep** folder, which is short for *dependencies*. It is within this folder that we would place third-party code libraries:

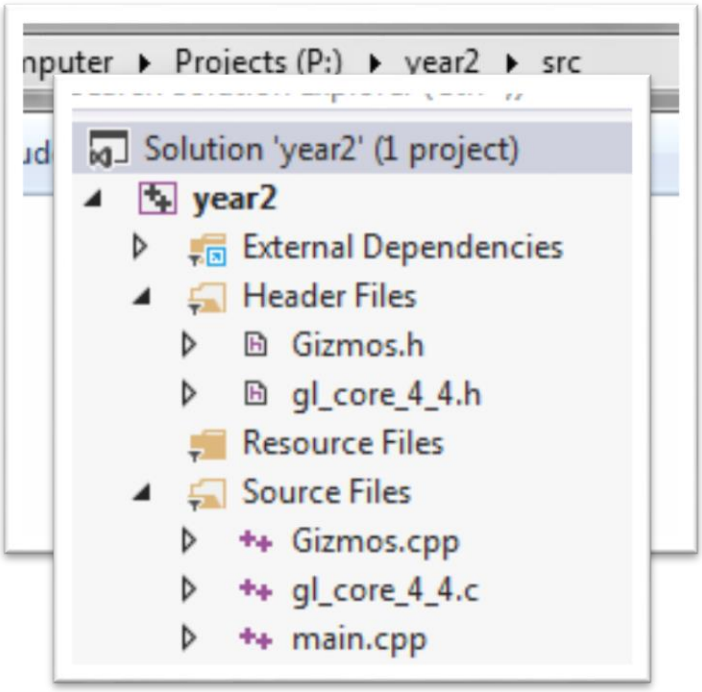


Be sure to correctly set the Include directories and Library directories to path to these locations. For now you should only need to link with **glfw3.lib** and **opengl32.lib** as **GLM** is a header-only library. Add the **Gizmos.cpp** file from the **AIE** utilities to your project but leave it where it is located.

Gizmos.cpp requires that an include path be set to the location of the **gl_core_4_4.h** file. If you wish to use a different generated OpenGL file than the one it is requesting you can, but be sure to edit **Gizmos.cpp**. For the include path just add the **src** folder as an include search path for your project.

Once we have the dependencies and folders setup we can create a **main.cpp** within our **src** folder. Do so, and add a simple main function:

```
int main() {
    return 0;
}
```



Creating an OpenGL Window:

Before we can draw anything we must first create a Window. A Window is a container for an OpenGL context that we can draw to. Almost every program has a window. The program you are reading this document in is a window, with this document contained within it!

GLFW makes it easy for us to create a Window to render in to. Without **GLFW** we would have to deal with a lot of Win32 code specific to Windows, or OSX code specific to OSX, etc

(All of the code below will be modifying our main.cpp)

First we need to include **GLFW** and our **OpenGL** header:

```
#include <gl_core_4_4.h>
#include <GLFW/glfw3.h>

int main() {
    return 0;
}
```

GLFW has two methods to initialise and terminate its internal systems; **glfwInit()** and **glfwTerminate()**.

```
#include <gl_core_4_4.h>
#include <GLFW/glfw3.h>

int main() {

    if (glfwInit() == false)
        return -1;

    // the rest of our code goes here!

    glfwTerminate();
    return 0;
}
```

Now that GLFW's systems are ready we can create a window. To create a window we must specify its resolution, its title, and then optionally we can specify which monitor the window should appear on. We simply set **nullptr** for the monitors and the window will appear on the primary display. Once we have created the window, which has its handle stored as a **GLFWwindow** pointer, we can assign it as the primary window to render in to. More complicated window set ups can allow for multiple windows that can be rendered in to.

```
#include <gl_core_4_4.h>
#include <GLFW/glfw3.h>

int main() {

    if (glfwInit() == false)
        return -1;

    GLFWwindow* window = glfwCreateWindow(1280, 720,
                                          "Computer Graphics",
                                          nullptr, nullptr);

    if (window == nullptr) {
        glfwTerminate();
        return -2;
    }

    glfwMakeContextCurrent(window);

    // the rest of our code goes here!

    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

We now have a window that can handle OpenGL rendering, however we first need to fix up the OpenGL extensions.

In the past this was a complicated process, but thanks to tools like the OpenGL Loader Generator it is a simple process to remap all of OpenGL's function calls to the correct versions and feature sets. After we have a window created we simply call **ogl_LoadFunctions()**:

```
if (ogl_LoadFunctions() == ogl_LOAD_FAILED) {
     glfwDestroyWindow(window);
     glfwterminate();
     return -3;
 }
```

We can test what version of OpenGL our program is running by calling the loader methods **ogl_GetMajorVersion()** and **ogl_GetMinorVersion()**:

```
auto major = ogl_GetMajorVersion();
auto minor = ogl_GetMinorVersion();
printf("GL: %i.%i\n", major, minor);
```

If we run our program we see that a window flickers into existence and then disappears. This is because we need to add a loop to our program.

We can easily loop while the window is open. To close the window we could simply press alt-F4, or the X button at the top of the window, or we could add a check to see if the ESCAPE key has been pressed on the keyboard. GLFW has a few functions for testing the state of the keyboard and mouse, so we simple loop while the window is still open and the ESCAPE key hasn't been pressed.

For GLFW to correctly check the keyboard it needs to poll for any events sent to it from the operating system, which includes keyboard presses, so at the end of the loop we call **glfwPollEvents()** so that all operating system messages and events are handled correctly.

We also call **glfwSwapBuffers()** on our window, within the loop. This updates the monitors display but swapping the rendered back buffer. If we did not call this then we wouldn't be able to see anything rendered by us with OpenGL.

```
while (glfwWindowShouldClose(window) == false &&
    glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS) {

    // our game logic and update code goes here!
    // so does our render code!

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

If you run our application and it compiles successfully then you should now be able to see a window appear that displays nothing. Congratulations, you just completed what once used to be a very long setup task for every game!

Next we will set about displaying useful graphics.

Adding visuals with Gizmos:

Having a window is one thing, but using it to display graphics is the aim here!

To begin with let's make sure some OpenGL calls work; we'll set a background clear colour and then clear the back-buffer every frame.

Every time we draw we must wipe the screen clean to display something, else we can end up with leftover visuals from the previous frame. Not every game must clear the buffer, but in our case we must.

Just before we start looping we should specify the colour we will clear our screen to. The method **glClearColor()** takes 4 float variables that represent the red, green, blue and alpha colour channels:

```
glClearColor( 0.25f, 0.25f, 0.25f, 1 );
```

With the colour set we can now add a glClear() call to the start of our loop, before any other call:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Running the program now we see the screen is a different colour. Try a few more colours.

GL_COLOR_BUFFER_BIT informs OpenGL to wipe the back-buffer colours clean.
GL_DEPTH_BUFFER_BIT informs it to clear the distance to the closest pixels. If we didn't do this then OpenGL may think the image of the last frame is still there and our new visuals may not display.

Your teacher will explain more about the depth buffer and its usage. By default it is off, so we must enable it.

To enable the depth buffer we need to add the following call to our code. For the moment put it just after the **glClearColor()** call:

```
glClearColor( 0.25f, 0.25f, 0.25f, 1 );
glEnable(GL_DEPTH_TEST); // enables the depth buffer
```

Changing colour is a little boring, so let's bring in the Gizmo class.

We should include the **Gizmo.h** file but also include **GLM** as we will be using its vector and matrix classes. **GLM** encases everything within a GLM namespace, so we will specify we are using particular classes out of it like so: **using glm::vec3**

Add the following to the top of your main.cpp along with your other includes:

```
#include <aie/Gizmos.h>
#include <glm/glm.hpp>
#include <glm/ext.hpp>

using glm::vec3;
using glm::vec4;
using glm::mat4;
```

Now we can create Gizmos.

The Gizmo class is a helper class that many studios have in some form or implementation. It is simply a tool to easily visualise shapes and lines. Studios use similar methods for displaying connections between waypoints, line-of-sight for enemy characters, debug collision and trigger volumes and many other useful things. The Gizmo class is designed to be self-contained.

Because the Gizmo class display primarily 3-D shapes we will need to setup a 3-D virtual camera. We will explain the details of the camera transforms in a later session. For now we simply have a **View** transform that represents the location of the camera, and a **Projection** transform that represents the lens of the camera.

Add the following code after the **ogl_LoadFunctions()** call but before the loop:

```
Gizmos::create();

mat4 view = glm::lookAt(vec3(10,10,10), vec3(0), vec3(0,1,0));
mat4 projection = glm::perspective(glm::pi<float>() * 0.25f,
                                   16/9.f, 0.1f, 1000.f);
```

This code initialises all of our gizmos and sets up our virtual camera.

Next, after the loop but before we terminate the GLFW window we need to destroy the gizmos:

```
Gizmos::destroy();
```

The Gizmo class works by allowing us to add geometric shapes and lines to it. We can then draw anything that has been added to it. The shapes remain until we call **Gizmos::clear()** which removes all shapes and lines, and we can add new ones.

Within the loop, after the **glClear()** but before the **glfwSwapBuffers()**, add the following:

```
    while (glfwWindowShouldClose(window) == false &&
        glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS) {

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        Gizmos::clear();

        Gizmos::addTransform(glm::mat4(1));

        vec4 white(1);
        vec4 black(0,0,0,1);

        for (int i = 0; i < 21; ++i) {
            Gizmos::addLine(vec3(-10 + i, 0, 10),
                            vec3(-10 + i, 0, -10),
                            i == 10 ? white : black);

            Gizmos::addLine(vec3(10, 0, -10 + i),
                            vec3(-10, 0, -10 + i),
                            i == 10 ? white : black);
        }

        Gizmos::draw(projection * view);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

Now, run your program and you should see a grid displayed in 3-D.

Experiment with the Gizmos class. Experiment with changing parameters to the virtual camera transforms. You should look over the GLM documentation for further details.

Solar System:

Using the Gizmo class and GLM mat4 transforms, attempt to create a Solar System. That is a Sun, surrounded by spinning planets, which also have spinning moons.

If you look at many of the Gizmo add methods that have a default parameter for a mat4 pointer that is **nullptr** or **mat4(1)** by default. This parameter can help you spin the planets and moons.

An Application Class:

Creating an entire program within a main() is silly as it will get quite large, and there is no neat way for us to differentiate code from different sessions and tutorials.

But we could create a class that we could inherit from for each session!

Many game engines and applications have a form of Application class. This is a base class that has create / destroy or startup / shutdown methods, along with update / draw methods. Games then derive from this class an implement their own startup / shutdown, loading and unloading resources specific to that game, and then an update that updates that particular game logic, and finally a draw or render method that displays the game in its certain way.

You are to create your own Application class that should have those 4 types of methods mentioned, which should ideally be setup as pure abstract methods.

Add whatever common code you think all of your applications will use. You may find new things to add to the base class at a later date.

Create a derived version of the class and have it implement the pure abstract methods to display everything we have gone over within this session. That is the startup initialises the Gizmos and possibly the window, and the shutdown destroys them.

Modify your main to only call the Application's methods. Perhaps the following could be a starting example:

```
#include "MyApplication.h"

int main() {

    Application* theApp = new MyApplication();
    if (theApp->startup() == true) {
        while (theApp->update() == true)
            theApp->draw();
        theApp->shutdown();
    }

    delete theApp;
    return 0;
}
```