# COSC342: OpenGL Introduction

Window Creation, OpenGL, Shader Introduction

## Objectives

- Open a window and create its OpenGL context
- Draw red triangle
- Pass parameters to a GLSL shader
- Render a textured cube.

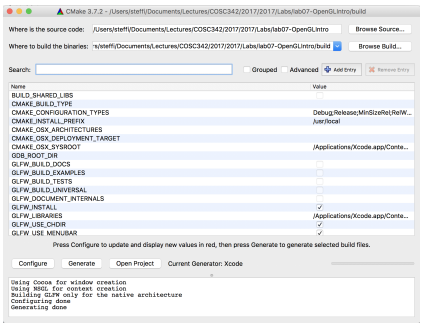
## Introduction

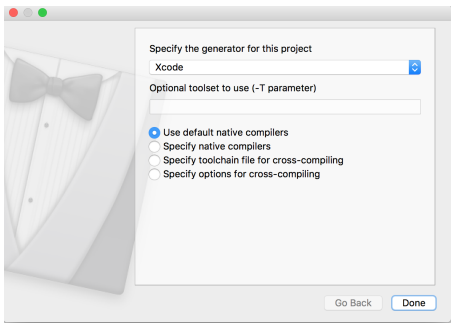
The following two labs will give you the background for writing your own render engine. Make sure that you copy the source files from the `/home/cshome/coursework/342/pickup/labs/lab08-OpenGLIntro/` directory into a directory within your home directory. Otherwise you may not be able to compile the code.

## Part 1: Window Creation

### Basics before starting

Similar to the last labs, the first step is to create the development project files. Open CMake.app from the Applications folder. Put the source code folder location for lab08-OpenGLIntro into source directory and add a location to where CMake should build the project files (e.g. lab08-OpenGLIntro/build). Click **Configure** and allow to create a new directory if the directory did not exist before.



Select the IDE you want to use, e.g. XCode and wait until configuring is finished. CMake will show settings and path relevant to project (we will leave them to the default values). Press **Generate** to create the project files.

In the following we will explain how to compile the project with XCode, but you can also go to the Terminal and use make to compile the project. However, please note that now the project files are getting more complex with multiple applications and several header and source files to edit during the lab, so XCode is recommended.

**XCode** Use Finder navigate to the build directory (e.g /build). Open XCode project file: `COSC_Lab_OpenGL1.xcodeproj`. In XCode click on "project navigator" and navigate to Part01. Have a look at the source file SimpleWindow.cpp. Much of this lab will involve you looking inside the files that have been provided. To compile and execute the code, go to "Product" and "Scheme" and select Part01 as scheme in the list. Then go to "Product" and click on "Build". Finally run the app by selecting "Product" - "Run".

**MAKE (advanced)** Change into the `lab08-OpenGLIntro/build` directory and type "`make`" to build the code, and then you should be able to type "`./Part01`" to run the executable code that has been generated. Part02 and Part03 require shaders and textures from the corresponding subfolders, so you need to navigate to the subfolder (e.g. Part02) and call *../build/Part*02 from there. If you want to avoid that, you can also use the shell scripts within the build directory (e.g. launch-Part02.sh. This script sets the correct working directories automatically. To run the shell script use "`./launch-Part02.sh`".

**Notes about CMakeList.txt** Now that our project files are getting more complex, we will have a closer look into the CMakeList file. The CMakeList.txt contains the input into the CMake build system. Commands are:

- *find_package* finds required libraries on the computer and sets the path e.g. *find_package*(OpenGL REQUIRED).
- *add_subdirectory* configures the compilation of external directories, e.g. *add_subdirectory*(external).
- *include_directories* sets the include directories so that the compiler finds header files.
- *add_executable* adds an executable to the project using the specified source files.

If you want to add source files to your project use *add_executable* to add new files, e.g.:

```
add_executable(Part01
  Part01/simpleWindow.cpp
  common/myNewClass.hpp
  common/myNewClass.cpp
)
```

After changing CMakeList.txt you need to configure and generate again.

## Source Code

Information:

- It's mostly C++ code.
- It uses inheritance to represent objects within a scenegraph but also to reuse code.
- It uses the OpenGL libraries, GLEW (GL extensions) and GLFW a multi-platform library for creating windows, contexts and receiving input and events.

Recall that C++ code generally has the following structure:

- Consists of a couple of header files (hpp) and source files (cpp)
- `#include`—these are header files to include.
- the `main` function—this is the first function called when the program is run and contains the render loop

## Overview Program Structure

If you track through the program to the "`main()`" function, we can see the initialisation steps:

- In the `initWindow` function a window is created using glfw.
- Sets the input mode using `glfwSetInputMode` to ensure that keyboard input can be captured.
- Calls renderLoop. In the first exercise, this will be a nearly empty while loop that checks for keyboard input. Nothing will happen in the renderLoop in this first exercise, but later on this will contain all the code for rendering objects to the screen. You can close the window pressing the ESCAPE button.

### OpenGL specifics

The main strength of OpenGL is its relative machine independence—one problem that can exist between different machines relates to the 'size' of variables—on some machines `int` is 16-bits, and on others it is 32-bits.

To provide a uniform look, OpenGL has its own types—and it's recommended that you use these where possible.

The main types are: `GLint`, (a 32-bit integer), and `GLfloat` (a 32-bit float).

## Exercise

- Create a fullscreen window by changing the window width and height to the appropriate values and use glfwGetPrimaryMonitor() as third parameter in glfwCreateWindow. e.g. window = glfwCreateWindow( 2880, 1800, $windowName.c\_str()$, glfwGetPrimaryMonitor(), NULL);
- Change the background color from dark blue background to dark green by modifying glClearColor in simpleWindow.cpp.

# Part 2: Drawing coloured triangles and quads

Now that you've created a basic window —let's move into part 2, and the world of shapes. Just as in Part 1, `build` the executable for Part02 (again by selecting the scheme for Part02 in XCode), and run it. The window will show a red triangle. In the following, we will have a look how the triangle is rendered and later on adjust the code to draw multiple triangles and change their colours as well quads.

## Red Triangle

In the main function, a triangle and a basicshader object are created. BasicShader is inherited from class Shader and each object has a shader object as member variable. In this example the shader object is added to the triangle object and later used for transforming the triangles vertices to the screen using the model-view projection matrix (MVP - refer to lecture about the graphics pipeline and the vertex shader). The triangle object is then added to our simple scenegraph (class Scenegraph) . All objects that are part of the screengraph are rendered in the main loop. The rendering function of the scenegraph class takes care of the rendering and also passes the MVP matrix to the shaders. We will have a detailed explanation of the shader code later.
⇒ *Read through the program code, and see what it is doing.*

```
1    // create a triangle and set a shader
     Triangle* myTriangle = new Triangle();
3    //create basic shader
     Shader* shader = new Shader( "basicShader");
```

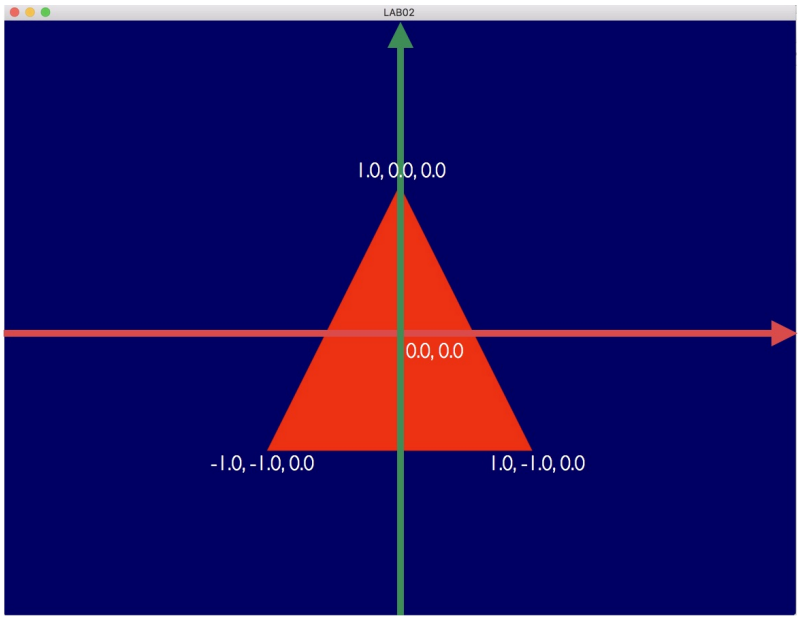
Within the triangle class definition (Triangle.cpp) we have the details how a triangle is generated and rendered. In the init function a buffer with vertices is created (vertex buffer object). Have a look at the organisation of the data within the buffer: Each vertex (a point in 3D) has 3 coordinates: x, y and z. Use the right-hand rule to get a better spatial understanding.

- X is your thumb and points to the right.
- Y is your index and points up.
- Z is your middle finger. With the thumb to the right and ndex finger up, the middle finger will point to your back.

In the array each vertex is represented by three subsequent floats (x,y,z). For example the first vertex is (-1,-1,0), also compare the coordinate system and the triangle vertices in the following Figure.

The next step is to create a vertexbuffer and pass our vertex data to vertex buffer.

```
g_vertex_buffer_data[0] = -1.0f, g_vertex_buffer_data[1]=  -1.0f,
    g_vertex_buffer_data[2]=0.0f;
2 g_vertex_buffer_data[3] = 1.0f, g_vertex_buffer_data[4]=  -1.0f,
    g_vertex_buffer_data[5]=0.0f;
g_vertex_buffer_data[6] = 0.0f, g_vertex_buffer_data[7]=  1.0f,
    g_vertex_buffer_data[8]=0.0f;
```




```
4
  // Generate buffer
6 glGenBuffers(1, &vertexbuffer);
  // bind buffer to pass following commands to our buffer
8 glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
  // Give our vertices to OpenGL.
10 glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
      g_vertex_buffer_data, GL_STATIC_DRAW);
```

The actual rendering happens inside the scenegraph's render method. The render method binds the corresponding shaders of each object and calls the object's render function. The camera parameter as input provides information about camera placement, orientation and settings (such as the field of view (FOV)).

```
1  void render(Camera* camera){
       for (int i=0;i<sceneObjects.size();i++)
3      {
           sceneObjects[i]->bindShaders();
5          sceneObjects[i]->render(camera);
       }
7  }
```

## Shaders

Have a look into the basic vertex and fragment shaders: basicshader.vert and basicshader.frag. In the basic vertex shader the incoming vertices are multiplied with the Model-View-Projection matrix that is passed to the shader. The parameter gl_Position is a GLSL built-in variable (contains the clip-space output position of the current vertex) that is then passed to the fragment shader.

```
gl_Position = MVP * vec4(vertexPosition_modelspace,1);
```

In the fragment shader the output colour for each fragment is computed. In our basic example here the output is set to a default red colour.

```
color = vec3(1.0,0.0,0.0);
```

Also have a look into the BasicShader class to understand how shaders are initialised and parameters are passed. The first step is the loading of the shaders. This happens in initShader using the LoadShaders method. LoadShaders takes two strings that contain the names of the vertex and the fragmentshader program code.

```
programID = LoadShaders(vertexshaderName.c_str(),
  fragmentshaderName.c_str());
```

Every time the shader should be used in the rendering, we let OpenGL know by using glUseProgram with the programID that we created using LoadShaders.

```
void bind(){
 // Use our shader
 glUseProgram(programID);
}
```

In order to pass parameters to the shader, we use glGetUniformLocation. This happens in after the creation of the shaders in the initShader method.

```
MatrixID = glGetUniformLocation(programID, "MVP");
```

If this variable changes we need to let the shader know by using the matching glUniformXXX function for our variable (in this example we have a 4x4 matrix, so we use glUniformMatrix4fv).

```
void updateMVP(glm::mat4 MVP){
 glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
}
```

## Notes on GLSL shaders

There are different types of variables available in GLSL, such as standard ones like float, bool, int, but also vec2,3,4 representing vectors and matrices (mat2,3,4) representing 2x2, 3x3 and 4x4 matrices. For vectors it is important to note that there are different ways how to access the vector elements (swizzling):

- r, g, b, a are used colours representing red, green, blue, alpha (blend factor).
- x, y, z, w are used for spatial coordinates like vectors and points.
- s, t, p, q are used for texture lookups.

For texture value (texel) access, samplers functions (e.g. sampler2D) are used. For our examples we will use sampler2D to access 2D textures.

### Exercise

- Change basicshader.frag to display a green triangle.
- Create a second triangle object and place both triangles 0.6 units apart from each other using the method setTranslate on the triangles.
- Write a class ColorShader derived from the class Shader. ColorShader should have a member colour (4 dimensional float vector variable for representing RGBA, e.g. glm::vec4(0.3,0.4,0.9,1.0))) and an additional method setColor. Remember from the OpenGL Essentials lecture how to pass parameters to a shader. (e.g. the colour variable can be passed to the shader using glUniform4f). Use ColorShader two create two triangles with different colours (pass two different colour values for each triangle. In order to use the ColorShader class in your project, add the created ColorShader.hpp and ColorShader.cpp to your project by updating the CMakeList.txt file (add files using *add_executable* for Part02 in CMakeList.txt, configure again using CMake and generate the project files again)
- Create a class Quad similar to the triangle class. A quad will consist of 2 triangles. In order to use the Quad class in your project, add the created Quad.hpp and Quad.cpp to your project by updating the CMakeList.txt file (add files using *add_executable* for Part02 in CMakeList.txt, configure again using CMake and generate the project files again). Finally create a quad object within your program and add it to the scenegraph. Make sure it will be rendered to the screen.

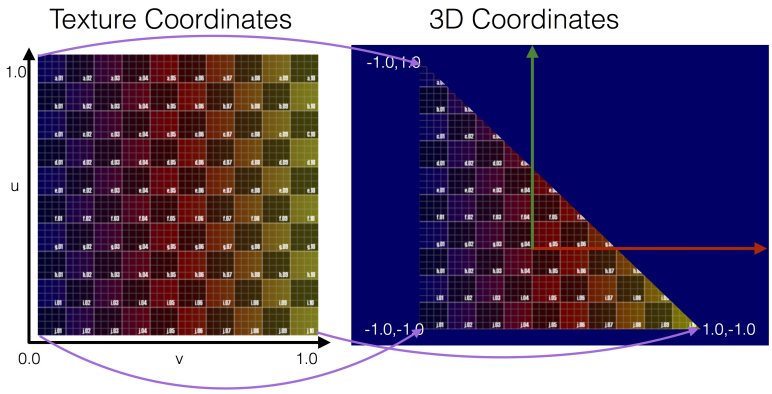
## Part 3: Using Textures

So far we only used simple colouring. In the third part of the lab, we will use texture to display image data on our rendered mesh. The background for adding textures are:

- Texture coordinates
- Texture loading
- Texture binding

## Texture coordinates

By using Textures coordinates (or UV coordinates) we tell OpenGL how the texture should be mapped on our mesh. For instance, for the previous triangle example, we need to give a textures coordinate for each vertex. This means OpenGL will know exactly which coordinate of the texture maps to which vertex of our triangle.




For this purpose, in addition to the vertex buffer, we have to create a buffer that holds the textures coordinates. In our example, we encapsulated these different buffers in an extra class called *Mesh*. The class *Mesh* allows to create mesh objects that have vertices, texture coordinates, normals, as well as indices (we will discuss the two latter ones in the next lab). We will use different VBO for vertices, texture coordinates and normals in this class. For setting texture coordinates, we use the method *setUVs*. As shown in the code snippet, a buffer is generated for the uv coordinates and the content of *m_uvs* is loaded into that buffer.

```
void Mesh:setUVs(std::vector<glm::vec2> uvs){
    std::copy(uvs.begin(), uvs.end(), m_uvs.begin());
    glGenBuffers(1, &m_uvBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_uvBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_uvs.size() * sizeof(glm::vec2),
     &m_uvs[0], GL_STATIC_DRAW);
}
```

## Texture loading

In order to use the texture itself, the first step is to pass the texture to OpenGL. The steps for passing the texture is 1) create texture with *glGenTextures* , bind it using *glBindTexture*, fill the texture with data *glTexImage2D*.

```
// Create an OpenGL texture
GLuint textureID;
glGenTextures(1, &textureID);

// "Bind" the newly created texture : all future texture
  functions will modify this texture
```

```
6   glBindTexture(GL_TEXTURE_2D, textureID);

8   // Give the image to OpenGL
    glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, width, height, 0, GL_BGR,
      GL_UNSIGNED_BYTE, data);
```

## Texture binding

Every time we want to use the texture, we have to bind it before rendering the object that should be textured. For this purpose we set the active texture using *glActiveTexture*. The function *glBindTexture* does the actual texture binding. We use the method bindTexture of Texture for encapsulating these steps:

```
1  void Texture::bindTexture(){
       // Bind our texture in Texture Unit 0
3      // we just use one texture unit here
       glActiveTexture(GL_TEXTURE0);
5      glBindTexture(GL_TEXTURE_2D, m_textureID);
   }
7
```

## Using texture in shader

To use the texture in the fragment shader, we use the texture coordinates (UV coordinates). For accessing the texture *sampler2D* is used and specify which texture is used (*myTextureSampler*). This will return a RGBA value that we will then use for setting the colour output.

```
   void main()
2  {
     // Output color = color of the texture at the specified UV
4    vec4 colorRGBA = texture( myTextureSampler, UV );
     //set output - at the moment only rgb
6    color = colorRGBA.rgb;
   }
8
```

## Exercise

- Change the fragment shader so that it displays the texture coordinates as colour values.
- Add a fixed colour value to the output fragment colour to create a coloured texture output.
- Replace the texture in the example with an photograph of your choice or the sample picture (testimage.bmp) inside the folder Part03 and map it on every side of the cube.