Game Engine Programming
Lab 3: An OpenGL Refresher

Karsten Pedersen
Department of Creative Technology

October 16, 2018

In this lab we aim to refresh our knowledge of the industry standard graphics API OpenGL used to draw and interact with 3D objects. *OpenGL* as an API is extremely flexible but also comes with quite a bit of complexity. Developing with *OpenGL* can often require some initial time to get used to how it works and also to set up the data used for the rendering. However it is also quite easy to become rusty if not used in your day to day projects.

The first thing we will need to do is to ensure that the project is set up and ready to use *OpenGL*. To just get a window appearing on the screen we will be using *SDL 2* because not only is the code required to do so fairly trivial but also because you are already familiar with it. Other libraries that are also common for this task include *[Free]Glut*, *GLFW*, *SFML* and *Allegro*. Open the provided *Microsoft Visual Studio* project or generate the *CMake* build system. The code inside ***main.cpp*** should be familiar to you by now. It simply opens up the *SDL 2* Window and not much else. You will see that we are not even creating an ***SDL_Renderer*** because we are going to be using *OpenGL* directly rather than using the basic renderer provided by *SDL 2* (which uses either *OpenGL* or *DirectX* underneath depending on platform).

Note:
Microsoft Windows provides a version of *OpenGL* which is extremely old (version **1.2**) and is too inflexible to make what we see today as modern games. Luckily a much newer implementation of *OpenGL* is provided by the graphics card manufacturer's driver (i.e *NVIDIA*, *AMD*, *Intel*, *MESA*). As of writing we are at around version **4.6** which only the very latest hardware supports. However the techniques and functionality covered in these labs was actually provided by version **2.1** which means that your code will work on almost all hardware found in the wild, even on mobile devices such as *Android* and *iOS* via *OpenGL ES*.

For convenience, rather than use a specific header and library (i.e ***GL/gl.h***) from each different vendor's SDK, we instead use a 3rd party library called *Glew* (Open**GL** **E**xtension **W**rangler). This library links the vendor's specific implementation with the platforms implementation at runtime so we can just use *OpenGL* as usual and not worry about the details.

First, lets begin by including the *Glew* header file into our project so we can start using *OpenGL* functions.

```
#include <GL/glew.h>
```

We now need to instruct *SDL 2* to create a window that is suitable for creating an *OpenGL* rendering context. To do this we need to modify the ***SDL_CreateWindow*** function call and pass ***SDL_WINDOW_OPENGL*** as a parameter.

```
SDL_Window *window = SDL_CreateWindow("Triangle",
  SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
  WINDOW_WIDTH, WINDOW_HEIGHT,
  SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
```

Next we need to create an *OpenGL* rendering context within the created *SDL 2* window and because *Glew* loads the *OpenGL* library and extensions at runtime we also need to initialise it. In your

project, just after where you open the *SDL 2* window using ***SDL_CreateWindow***, add the following function calls.

```
if(!SDL_GL_CreateContext(window))
{
  throw std::exception();
}

if(glewInit() != GLEW_OK)
{
  throw std::exception();
}
```

Note:
This must be **after** the call to open the window because otherwise there is no window for *OpenGL* to bind a context to and the call will fail.

With this in place we are now ready to start with *OpenGL*. What we are first going to do to confirm everything is working is change the screen to the colour red. The following listing will first set the current *OpenGL* clear colour to red, will then actually instruct *OpenGL* to clear the screen and finally it will atomically swap the *OpenGL* memory buffer with that of the screen buffer (to eliminate flicker). You will want to place this code where you would usually start drawing to the screen in a standard *SDL 2* application.

```
glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
SDL_GL_SwapWindow(window);
```

Note:
Unlike when using *SDL* where colours usually range between ***0*** and ***255*** (maximum range of an ***unsigned char***), *OpenGL* instead deals with floating point values and so not only is a value between ***0*** and ***1*** such as ***0.332*** possible, it also simplifies many calculations. Floating point values are also handled very quickly on a maths co-processor (which is essentially exactly what a graphics card is).

With that in place, compile the project and run it. You should hopefully see a similar output to **Figure 1**.

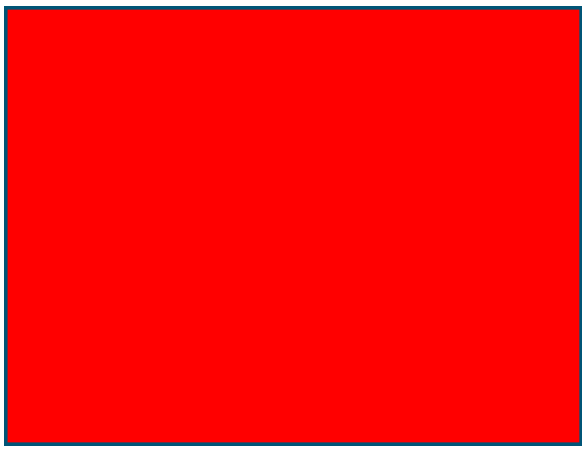

Figure 1: *A screen shot of our OpenGL program simply clearing the screen red*

Now we can finally begin using the core functionality of *OpenGL*. In 3D graphics almost everything is made up of triangles so lets start with trying to draw one. There is quite a lot of initial work that needs to be done before we can actually view an image on the screen. **Figure 2** should give you an overview of the tasks required to do this. This same code can be reused however so after the initial hurdle, you may find it surprisingly easy to extend.

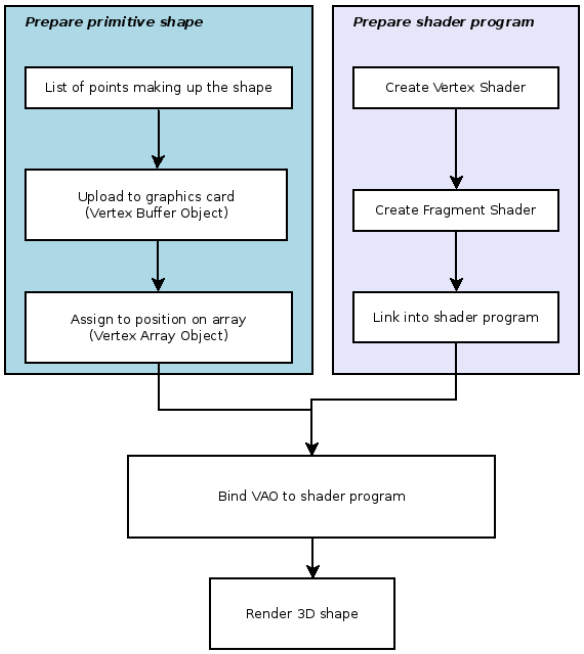

Figure 2: *Diagram showing the series of tasks in order to render a triangle*

Preparing the Primitive Shape Data

We will start with preparing the primitive shape. To do this we need a list of points that describe our shape (in our case our single triangle). We will use 3D points (include z component) because we aim to get into drawing 3D objects. For 2D components such as GUI, you will likely not include the z component to save memory on the GPU.

```
const GLfloat positions[] = {
  0.0f, 0.5f, 0.0f,
  -0.5f, -0.5f, 0.0f,
  0.5f, -0.5f, 0.0f
};
```

Note:
Don't get too hung up on the *OpenGL* types. For example ***GLfloat*** is just a typedef. On almost all platforms it will be a typedef of the standard ***float***. The only reason for *OpenGL* to do this is because it is designed to work on an extremely large range of platforms unlike i.e *Microsoft DirectX*. For this reason it cannot guarantee or make the assumption that the platform will provide certain functionality. In this case it would typedef to the next best thing (such as a ***double***).

When we run the program, this data is in our RAM (or stack). We want to upload it to the GPU. The following listing will create a new *Vertex Buffer Object* to store data in, bind it so that it is the active *VBO* for operations, upload the data to the currently bound *VBO* and finally unbind the buffer again to return the state back to normal.

```
GLuint positionsVboId = 0;

// Create a new VBO on the GPU and bind it
glGenBuffers(1, &positionsVboId);

if(!positionsVboId)
{
  throw std::exception();
}

glBindBuffer(GL_ARRAY_BUFFER, positionsVboId);

// Upload a copy of the data from memory into the new VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW);

// Reset the state
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Note:
The ***GLuint*** is again, basically just a typedef of an ***unsigned int***. The reason why we do not have a pointer is because pointers can only point to memory in RAM or on the stack. The actual memory we are referring to is on the GPU so instead the best we can do is use an integer ID and copy over the raw memory containing the data for the points. This is also what makes systems programming languages (such as *C*, *C++* and *Rust*) so crucial for games and embedded programming because non-native things such as *Microsoft C#/VB.NET* and *Java* use garbage collectors (which cannot function on the GPU) and dealing with raw memory in these languages is inefficient or impossible (instead they have to call into *C* anyway).

Now we need to create the *Vertex Array Object (VAO)*. At the moment you might be wondering why this is needed because the data is already on the GPU. In older implementations of *OpenGL* it was not needed and depending on your drivers it might work without. However when you start needing to include texture coordinates, vertex normals, lightmap coordinates, it will start to make more sense why it is now a required step. **Figure 3** may help explain the interactions between these concepts. The following listing will create the *VAO*, bind it, bind the *VBO* and assign our *VBO* containing the vertex positions to be our first entry (position ***0***). The position ***0*** will then be flagged as *enabled* so that it will be streamed into the shader as an attribute stream later on (This step is required to prevent unused streams from passing garbage data into the shader). Finally we unbind the *VBO* and unbind the *VAO* to reset to the default state.

```
GLuint vaoId = 0;

// Create a new VAO on the GPU and bind it
glGenVertexArrays(1, &vaoId);

if(!vaoId)
{
  throw std::exception();
}

glBindVertexArray(vaoId);

// Bind the position VBO, assign it to position 0 on the bound VAO
// and flag it to be used
glBindBuffer(GL_ARRAY_BUFFER, positionsVboId);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
  3 * sizeof(GLfloat), (void *)0);

glEnableVertexAttribArray(0);

// Reset the state
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

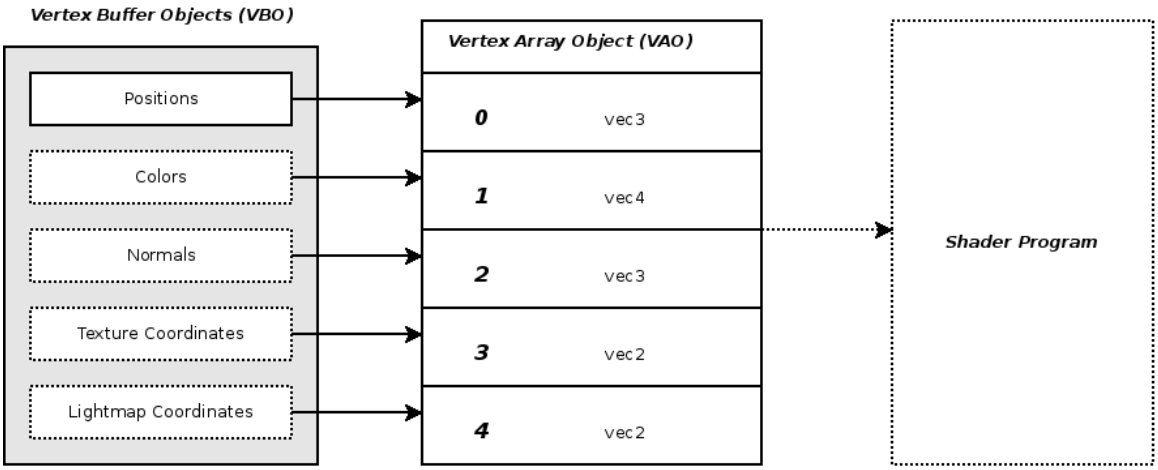

Figure 3: *Diagram showing the relationship between VBOs, VAOs and shader programs*

Preparing the Shader Program

We now have the data that we want to draw uploaded, however the GPU does not yet know how we want it to be drawn. What we need to do is write a very small program to upload to the GPU that uses the data to do the drawing. This small program is known as the shader program.
Shader programs are written using *GL Shader Language* (*GLSL*). The *Microsoft DirectX* equivalent is *High Level Shader Language* (*HLSL*) for example. The syntax of *GLSL* is very similar to *C/C++* but has many built in types and functions which are useful for graphics programming such as ***vec4*** and ***pow***. As with *OpenGL* itself, there are many different versions of *GLSL*. We will be using ***1.2*** which is fairly old. Again we will be doing this to reach maximum compatibility with different hardware and drivers. The very latest at the time of writing is ***4.6 core*** and there are a small number of differences between them that you may want to research if you are interested.

Shader programs are similar to standard *C/C++* programs in that they are made up of multiple objects. These objects consist of different shaders such as the *Vertex Shader*, *Fragment Shader*,

Tessellation Shader, etc. At the very least a program must have a *Vertex Shader* and a *Fragment Shader*. The *Vertex Shader* deals with the positions of the shape, such as the three points of a triangle whereas the *Fragment Shader* deals with the fragments within a shape, so drawing all the pixels that reside within the area a triangle. We will begin creating our shader program by first writing and compiling a simple *Vertex Shader*.

```
const GLchar *vertexShaderSrc =
  "attribute vec3 in_Position;              " \
  "                                         " \
  "void main()                              " \
  "{                                        " \
  "  gl_Position = vec4(in_Position, 1.0); " \
  "}                                        ";

// Create a new vertex shader, attach source code, compile it and
// check for errors.
GLuint vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, &vertexShaderSrc, NULL);
glCompileShader(vertexShaderId);
GLint success = 0;
glGetShaderiv(vertexShaderId, GL_COMPILE_STATUS, &success);

if(!success)
{
  throw std::exception();
}
```

This *GLSL* is the simplest example of a working *Vertex Shader*. Notice the variable-like declaration at the top of the *GLSL* called ***in_Position***? This is what allows us to read the stream of data from the *VBO* that we prepared earlier. The *GLSL* code currently states that it should use the ***vec3*** that it receives directly by assigning it to the final output vertex position (***gl_Position***). An awkward part here is that the *GLSL* is embedded into the *C/C++* code as a string. This makes multi-line entry a bit awkward. A more scalable solution is either using some ***MACRO*** trickery or simply loading the shader from a file. With that working, the next step is for the *Fragment Shader*. The code is very similar so make sure not to get the two confused.

```
const GLchar *fragmentShaderSrc =
  "void main()                          " \
  "{                                    " \
  "  gl_FragColor = vec4(0, 0, 1, 1); " \
  "}                                    ";

// Create a new fragment shader, attach source code, compile it and
// check for errors.
GLuint fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, &fragmentShaderSrc, NULL);
glCompileShader(fragmentShaderId);
glGetShaderiv(fragmentShaderId, GL_COMPILE_STATUS, &success);

if(!success)
{
  throw std::exception();
}
```

Again, the *GLSL* code is very simple and states that for any fragment it draws, use the colour blue (***0, 0, 1, 1***). Now we need to link our new shader objects together as part of our complete shader program. We do this by attaching the shader objects and linking the program. However, remember when we created our *VAO* we set the positions *VBO* to position ***0*** (the first and only location)? We need to instruct the shader whilst linking to put our input variable ***in_Position*** at that position

too so the correct attribute stream matches up.

```
// Create new shader program and attach our shader objects
GLuint programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);

// Ensure the VAO "position" attribute stream gets set as the first position
// during the link.
glBindAttribLocation(programId, 0, "in_Position");

// Perform the link and check for failure
glLinkProgram(programId);
glGetProgramiv(programId, GL_LINK_STATUS, &success);

if(!success)
{
  throw std::exception();
}

// Detach and destroy the shader objects. These are no longer needed
// because we now have a complete shader program.
glDetachShader(programId, vertexShaderId);
glDeleteShader(vertexShaderId);
glDetachShader(programId, fragmentShaderId);
glDeleteShader(fragmentShaderId);
```

And that is it for the preparation. Congratulations! We can now finally perform the drawing operation. In comparison this is an extremely simple task. We tell *OpenGL* to use our created shader, we tell it to use our created triangle data (*VAO*) and finally we tell it to draw. Then finally we reset the state back. So just under your ***glClear*** line add the following to receive a result similar to that shown in **Figure 4**.

```
// Instruct OpenGL to use our shader program and our VAO
glUseProgram(programId);
glBindVertexArray(vaoId);

// Draw 3 vertices (a triangle)
glDrawArrays(GL_TRIANGLES, 0, 3);

// Reset the state
glBindVertexArray(0);
glUseProgram(0);
```

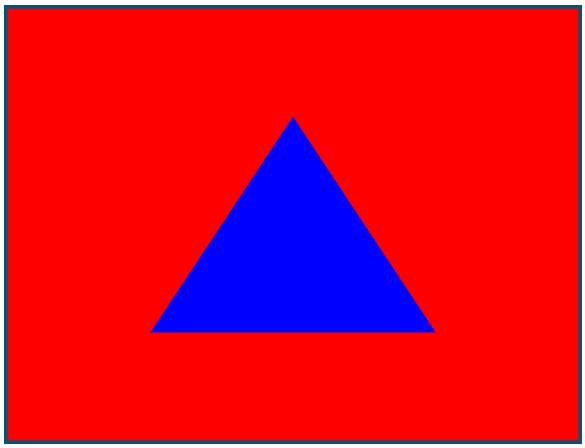

Figure 4: *A screen shot of our OpenGL program drawing a single triangle*

Note:
The important thing to note is that all of our data is now stored within the graphics card by the time it comes to drawing. This means that rather than sending across all the data each frame (as is the case with software rendered *SDL*), with *OpenGL* we can instead just refer to which buffer to draw (via an ID) which minimises the data sent between the CPU and GPU to help achieve maximum performance.

Have a play with the position coordinates that you upload to the GPU. You will notice that by default *OpenGL* coordinates are between ***-1*** and ***1*** in both the x and y axis. This is basically the *normalised device coordinates* and by using a different *Projection* matrix, this will change. We will cover this in a future lab.

There was a lot to cover in this lab. You will notice that you now have quite a bit of code in your ***main.cpp*** file. Much of the remaining Game Engine Programming unit will be looking into the ways to integrate OpenGL into games in a flexible and scalable way.