***Introduction to Graphics and Mobile Gaming***

**LAB 3**

**SimpleTriangle**

**Issue 1.0**

1. Contents

[1. Introduction 1](#_Toc36027156)

[2. Creating a new Project + Native includes 1](#_Toc36027157)

[3. Shader loading 1](#_Toc36027158)

[4. Graphics and rendering 4](#_Toc36027159)

[5. Last touches 5](#_Toc36027160)

# Introduction

Now that we have set up the basics of our application and developed a basic understanding of shaders and how they are useful in graphics, we can start to apply this knowledge to creating interesting things! We will begin with the creation of a simple triangle. This lab will primarily alter/add code in the native-lib.cpp file.

# Creating a new Project + Native includes

The previous project will act as the starting point of this one. Simply copy the code across or create a new project and include the files. Remember to rename the project and everything specific to the name to SimpleTriangle, otherwise things such as packages and C functions in Java will not work.

Also, in our native-lib.cpp file, we need several includes so we have the adequate support to complete the project:

**#include** <jni.h>

**#include** <android/log.h>

**#include** <GLES2/gl2.h>

**#include** <GLES2/gl2ext.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <math.h>

**#include** <unistd.h>

**#define** LOG\_TAG "libNative"

**#define** LOGI(...) \_\_android\_log\_print(ANDROID\_LOG\_INFO, LOG\_TAG, \_\_VA\_ARGS\_\_)

**#define** LOGE(...) \_\_android\_log\_print(ANDROID\_LOG\_ERROR, LOG\_TAG, \_\_VA\_ARGS\_\_)

# Shader loading

Now that we have an understanding of shaders and how they can be implemented using OpenGL ES, let’s get started with coding them for our simple triangle! The following code is to be added to the native-lib.cpp file.

We start by creating an array that will be our vertex shader source code. The first line of the shader code declares an attribute called “vPosition”. The main function then begins. The next line of note sets the final position of the vertex using our attribute.

**static** **const** **char** glVertexShader[] =

"attribute vec4 vPosition;\n"

"void main()\n"

"{\n"

" gl\_Position = vPosition;\n"

"}\n";

Next, we need to write the code for our fragment shader. Again, we begin by creating the array that will house the source code for the shader. The first line of the shader code sets the precision the GPU will uphold when calculating floats. We want to assign the colour red to each fragment of our triangle. We write the line that will achieve this within our fragment shader.

**static const char** glFragmentShader[] =

"precision mediump float;\n"

"void main()\n"

"{\n"

"gl\_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n"

"}\n";

The next task is writing a function that will load our shaders and compile them, ready to be used in a program.

GLuint **loadShader**(GLenum shaderType, **const** **char**\* shaderSource)

{

GLuint shader = glCreateShader(shaderType);

**if** (shader)

{

glShaderSource(shader, 1, &shaderSource, NULL);

glCompileShader(shader);

GLint compiled = 0;

glGetShaderiv(shader, GL\_COMPILE\_STATUS, &compiled);

**if** (!compiled)

{

GLint infoLen = 0;

glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &infoLen);

**if** (infoLen)

{

**char** \* buf = (**char**\*) **malloc**(infoLen);

**if** (buf)

{

glGetShaderInfoLog(shader, infoLen, NULL, buf);

LOGE("Could not Compile Shader %d:\n%s\n", shaderType, buf);

**free**(buf);

}

glDeleteShader(shader);

shader = 0;

}

}

}

**return** shader;

}

At first, this may look like a lot of code and seem quite daunting, but once we break it down, it is relatively simple!

The first thing we do is create a new shader object. Assign the newly created shader to a GLuint called *“shader”*. We then test to see if we created a valid shader object before continuing. If the shader object is valid, we use *glShaderSource* to use the shader source code that was passed into the function initially. This source code would be the char array defined earlier on. Once we have selected the shader source, we need to compile it. Next, we check if the compile was successful.

We now need to write the function that will create our program so that we can attach our shaders to it and display them in the scene.

GLuint **createProgram**(**const** **char**\* vertexSource, **const** **char** \* fragmentSource)

{

GLuint vertexShader = loadShader(GL\_VERTEX\_SHADER, vertexSource);

**if** (!vertexShader)

{

**return** 0;

}

GLuint fragmentShader = loadShader(GL\_FRAGMENT\_SHADER, fragmentSource);

**if** (!fragmentShader)

{

**return** 0;

}

GLuint program = **glCreateProgram**();

**if** (program)

{

glAttachShader(program, vertexShader);

glAttachShader(program, fragmentShader);

**glLinkProgram**(program);

GLint linkStatus = GL\_FALSE;

**glGetProgramiv**(program , GL\_LINK\_STATUS, &linkStatus);

**if**( linkStatus != GL\_TRUE)

{

GLint bufLength = 0;

**glGetProgramiv**(program, GL\_INFO\_LOG\_LENGTH, &bufLength);

**if** (bufLength)

{

**char**\* buf = (**char**\*) **malloc**(bufLength);

**if** (buf)

{

**glGetProgramInfoLog**(program, bufLength, NULL, buf);

LOGE("Could not link program:\n%s\n", buf);

**free**(buf);

}

}

**glDeleteProgram**(program);

program = 0;

}

}

**return** program;

}

Both the fragment shader and the vertex shader source are passed into the “createProgram”function. We call the “loadShader” function we just created twice: once for the vertex shader and once for the fragment shader.

Then, we use the OpenGL function “glCreateProgram()” to create a new program and assign it to a GLuint called “program”. We need to attach both of our shaders to our newly created program. Once we have attached our shaders to the program, we need to link them using “glLinkProgram”. The remainder of the code is simply checking that everything was successful and looking out for errors.

# Graphics and rendering

We now need to write our “setupGraphics” function and also the “renderFrame”function.

GLuint simpleTriangleProgram;

GLuint vPosition;

**bool** **setupGraphics**(**int** w, **int** h)

{

simpleTriangleProgram = createProgram(glVertexShader, glFragmentShader);

**if** (!simpleTriangleProgram)

{

LOGE ("Could not create program");

**return** **false**;

}

vPosition = **glGetAttribLocation**(simpleTriangleProgram, "vPosition");

**glViewport**(0, 0, w, h);

**return** **true**;

}

Before we start writing the “setupGraphics” function, we declare two variables:

* *simpleTriangleProgram* – This will be the program we create and attach the shaders to.
* *vPosition* – This will be the location of where the GPU is expecting the vertex data for the shader.

The function itself starts by creating a program object using the vertex and fragment shader (char arrays) we declared earlier on and assigning it to the variable we just defined. We test to see if the program was created and assigned successfully; if it was not, then we send an error message to Logcat and return false. If the creation was successful, we then call “glGetAttribLocation()” to get the location for where our *vPosition* data should be. Finally, we pass dimensions into the “glViewport()” function using the height and width (h and w) that were passed into the function initially.

The first section of the following code we need to add is an array of floats that define the three points (vertices) of our triangle. Each couple of floats is a vertex. The values can range from -1.0 to 1.0 and the screen goes from bottom left (-1.0f, -1.0f) to top right (1.0f, 1.0f). This is a simple 2D triangle, so we do not need to concern ourselves with the Z axis.

**const** GLfloat triangleVertices[] = {

0.0f, 1.0f,

-1.0f, -1.0f,

1.0f, -1.0f

};

**void** **renderFrame**()

{

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

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

**glUseProgram**(simpleTriangleProgram);

**glVertexAttribPointer**(vPosition, 2, GL\_FLOAT, GL\_FALSE, 0, triangleVertices);

**glEnableVertexAttribArray**(vPosition);

**glDrawArrays**(GL\_TRIANGLES, 0, 3);

}

The second section is the function that handles the drawing of our graphics. In order to make our red triangle stand out, we will begin the function by using the “glClearColor” function to set the clear colour to black. We next call “glClear” which will clear the depth buffer and the colour buffer each frame. The next thing we do is select which program we want to display for our scene. We select the program we have previously attached our shaders to as this is what we want to show. After this we link the attribute in the shader to the data (float array) we just defined above. This is done using the “glVertexAttribPointer()” and passes in the expected location of the data, the number of elements per vertex, “GL\_FALSE” as we do not need to normalize the data, and finally the data itself. The last line in this section simply draws out the array, stating that it should be drawn as triangles at element 0, containing 3 vertices.

# Last touches

We just need to make some final, small alterations now, and we are ready to go. First, we need to alter the “init” and “step” function definitions.

**extern "C"**{  
 **JNIEXPORT void JNICALL** Java\_com\_arm\_malideveloper\_simpletriangle\_NativeLibrary\_init(  
 JNIEnv \* env, jclass type, jint width, jint height){  
 setupGraphics(width, height);  
 };  
 **JNIEXPORT void JNICALL** Java\_com\_arm\_malideveloper\_simpletriangle\_NativeLibrary\_step(  
 JNIEnv \* env, jclass type){  
 renderFrame();  
 };  
  
};

As we can see, we have updated them to call our newly created functions. The “init” function calls “setupGraphics” when the application starts up. Then, “step” calls “renderFrame” for every frame.

Then, to finish, we need to make one change to the “CMakeLists.txt” file.

**target\_link\_libraries**( # Specifies the target library.  
 native-lib  
  
 # Links the target library to the log library  
 # included in the NDK.  
 ${log-lib}  
 **GLESv2**)

We simply added “GLESv2” to the “target\_link\_libraries”section as we are now making use of the GLES libraries to generate graphics.