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

**LAB 5**

**Textured Cube**

**Issue 1.0**

1. Contents

[1. Introduction 1](#_Toc36042162)

[2. Project setup 1](#_Toc36042163)

[3. Load texture 1](#_Toc36042164)

[4. Code updates 3](#_Toc36042165)

[4.1 Shaders 3](#_Toc36042166)

[4.2 Setup graphics 4](#_Toc36042167)

[4.3 Texture coordinates 5](#_Toc36042168)

[4.4 Enabling attributes and passing sampler locations 6](#_Toc36042169)

# Introduction

In the previous tutorial, we created a 3D cube that rotates. We will now build upon this project, as it previously only had a colour; however, we want it to look a bit nicer. We will attempt to give our cube a texture!

A texture is an image that you wish to be displayed on a model. These images can be of any format, providing that the code required to load that format is present.

# Project setup

For this project, we will carry on from the previous cube example. This makes sense because we are going to be using a cube in this example too. Feel free to copy the previous project so in the end you can have both. We will call this project “TexturedCube”!

From the previous project, we need to add two new native files. Like the previous example, we are not going to touch the Java side as the setup code stays the same. The two added native files are Texture.cpp and Texture.h, so right-click on your cpp folder and select new c++ file and then give them the name just mentioned. Like before, we need to add the Texture.cpp file to our CMakeLists, so that it gets compiled when we build the project.

**add\_library**( # Sets the name of the library.  
 native-lib  
  
 # Sets the library as a shared library.  
 SHARED  
  
 # Provides a relative path to your source file(s).  
 native-lib.cpp  
 Matrix.cpp

Texture.cpp)

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

# Load texture

Now that we have added our new texture classes, we need to populate them with some code! We start in the Texture.cpp file by writing our “loadSimpleTexture” function, which, as the name suggests, will be the function that loads the texture we wish to give our cube/model. In this function, we load the image, convert it to a suitable format for OpenGL ES, and then load the texture into the GPU’s memory so we can use it in our shaders.

**#include** "Texture.h"

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

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

**#include** <cstdio>

**#include** <cstdlib>

GLuint **loadSimpleTexture**()

{

/\* Texture Object Handle. \*/

GLuint textureId;

/\* 3 x 3 Image, R G B A Channels RAW Format. \*/

GLubyte pixels[9 \* 4] =

{

18, 140, 171, 255, /\* Some Colour Bottom Left. \*/

143, 143, 143, 255, /\* Some Colour Bottom Middle. \*/

255, 255, 255, 255, /\* Some Colour Bottom Right. \*/

255, 255, 0, 255, /\* Yellow Middle Left. \*/

0, 255, 255, 255, /\* Some Colour Middle. \*/

255, 0, 255, 255, /\* Some Colour Middle Right. \*/

255, 0, 0, 255, /\* Red Top Left. \*/

0, 255, 0, 255, /\* Green Top Middle. \*/

0, 0, 255, 255, /\* Blue Top Right. \*/

};

The first thing we do is sort out all of the includes we need for the class. We start by including the header file for our texture class and then the basics that we have imported in the previous examples.

The next thing we do is define a new variable called “textureId” which holds the ID of the texture so we can reference it later on.

After this, we define another variable that contains the actual image data. We will be using the RAW image format that is expected as colour data one after the other. OpenGL ES expects the first bit of data to represent the bottom left so we have ordered the data in order to fulfil these expectations. Since we are using the RGBA colour format, each row has a red, green, blue, and alpha element; these numbers range from 0 to 255.

/\* [placeTextureInMemory] \*/

/\* Use tightly packed data. \*/

**glPixelStorei**(GL\_UNPACK\_ALIGNMENT, 1);

/\* Generate a texture object. \*/

glGenTextures(1, &textureId);

/\* Activate a texture. \*/

glActiveTexture(GL\_TEXTURE0);

/\* Bind the texture object. \*/

glBindTexture(GL\_TEXTURE\_2D, textureId);

/\* Load the texture. \*/

**glTexImage2D**(GL\_TEXTURE\_2D, 0, GL\_RGBA, 3, 3, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, pixels);

/\* Set the filtering mode. \*/

**glTexParameteri**(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

**glTexParameteri**(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

**return** textureId;

}

The next thing we need to do is add the image data to memory so it can be accessed. We begin by using the “glPixelStorei” function as this will pack our data tightly and save some space.

The line after generates the ID number that we assign to the variable we declared earlier on. Some parameters will be required, and these are: number of textures being generated and the place to store them. We only have one texture and created a texture object handle earlier on.

After this, we need to activate the texture and bind it to a specific unit type. We have two choices for the binding texture type. These are GL\_TEXTURE\_2D or GL\_TEXTURE\_CUBE\_MAP. For this instance, we want GL\_TEXTURE\_2D as it is the most frequently used.

From here, we can upload the texture using the “glTexImage2D” function. As with the previous function call, we provide the unit type that we wish to use. The next parameter that we use is the number of mipmaps to be passed in—this will be discussed in later tutorials. Next is the internal format (RGBA); then, the 7th parameter is the same as both, the internal format and the format of the image must be the same. Parameters 4 and 5 are the width and height of the image. The sixth is a border for the image—this will simply be set to zero. The final two parameters are the type of data we will use and the actual data.

The final two function calls define what happens when we need to stretch or shrink the image. For instance each of our cubes faces are not going to be 3 x 3 pixels. This means we are going to have to stretch the image. We have told OpenGL ES in this instance to use the pixel that is closest to the one it should pick. So roughly the cube will be divided into 9 equal portions of colour.

# Code updates

## Shaders

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

"attribute vec4 vertexPosition;\n"

"attribute vec2 vertexTextureCord;\n"

"varying vec2 textureCord;\n"

"uniform mat4 projection;\n"

"uniform mat4 modelView;\n"

"void main()\n"

"{\n"

" gl\_Position = projection \* modelView \* vertexPosition;\n"

" textureCord = vertexTextureCord;\n"

"}\n";

**static** **const** **char** glFragmentShader[] =

"precision mediump float;\n"

"uniform sampler2D texture;\n"

"varying vec2 textureCord;\n"

"void main()\n"

"{\n"

" gl\_FragColor = texture2D(texture, textureCord);\n"

"}\n";

Now that we have set up our texture function, we need to alter the shader code to incorporate the use of textures for our model. We have removed the references to the colour attribute from the previous example because we do not need to provide per vertex colour data if we are using a texture. Instead, we supply a per vertex texture coordinate. This value will state where in the texture the vertex should take its colour from. Texture coordinates go from 0,0 in the bottom left-hand corner to 1,1 in the top right-hand corner. We add a varying, which is a vec2 attribute; we will pass this to the fragment shader too.

We also have a new uniform value in the fragment shader source code, of type ‘sampler2D’. If we had more than one texture, we would use this attribute to decide which texture to use.

We assign the result of a function called ‘texture2D’. This function takes in two parameters, which are the sampler and the texture coordinate that tells the unit where to go in the texture.

## Setup graphics

**GLuint textureId;**

**bool** **setupGraphics**(**int** width, **int** height)

{

glProgram = createProgram(glVertexShader, glFragmentShader);

**if** (!glProgram)

{

LOGE ("Could not create program");

**return** **false**;

}

vertexLocation = glGetAttribLocation(glProgram, "vertexPosition");

textureCordLocation = glGetAttribLocation(glProgram, "vertexTextureCord");

projectionLocation = glGetUniformLocation(glProgram, "projection");

modelViewLocation = glGetUniformLocation(glProgram, "modelView");

samplerLocation = glGetUniformLocation(glProgram, "texture");

/\* Setup the perspective. \*/

matrixPerspective(projectionMatrix, 45, (**float**)width / (**float**)height, 0.1f, 100);

**glEnable**(GL\_DEPTH\_TEST);

**glViewport**(0, 0, width, height);

/\* Load the Texture. \*/

textureId = loadSimpleTexture();

**if**(textureId == 0)

{

**return** **false**;

}

**else**

{

**return** **true**;

}

}

For this function, we need to remove any calls relating to colour from the previous example and replace it with the texture equivalents and sample location. There are 5 lines of code that are required here. Two of them are obtaining attribute locations of vertex positions and texture coordinates. Three of them are to do with obtaining uniform locations for two of our matrices and the texture. We also add a call to the *“loadSimpleTexture*” function.

## Texture coordinates

GLfloat cubeVertices[] = {-1.0f, 1.0f, -1.0f, /\* Back. \*/

1.0f, 1.0f, -1.0f,

-1.0f, -1.0f, -1.0f,

1.0f, -1.0f, -1.0f,

-1.0f, 1.0f, 1.0f, /\* Front. \*/

1.0f, 1.0f, 1.0f,

-1.0f, -1.0f, 1.0f,

1.0f, -1.0f, 1.0f,

-1.0f, 1.0f, -1.0f, /\* Left. \*/

-1.0f, -1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,

-1.0f, 1.0f, 1.0f,

1.0f, 1.0f, -1.0f, /\* Right. \*/

1.0f, -1.0f, -1.0f,

1.0f, -1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

-1.0f, 1.0f, -1.0f, /\* Top. \*/

-1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, -1.0f,

-1.0f, - 1.0f, -1.0f, /\* Bottom. \*/

-1.0f, -1.0f, 1.0f,

1.0f, - 1.0f, 1.0f,

1.0f, -1.0f, -1.0f

};

GLfloat textureCords[] = { 1.0f, 1.0f, /\* Back. \*/

0.0f, 1.0f,

1.0f, 0.0f,

0.0f, 0.0f,

0.0f, 1.0f, /\* Front. \*/

1.0f, 1.0f,

0.0f, 0.0f,

1.0f, 0.0f,

0.0f, 1.0f, /\* Left. \*/

0.0f, 0.0f,

1.0f, 0.0f,

1.0f, 1.0f,

1.0f, 1.0f, /\* Right. \*/

1.0f, 0.0f,

0.0f, 0.0f,

0.0f, 1.0f,

0.0f, 1.0f, /\* Top. \*/

0.0f, 0.0f,

1.0f, 0.0f,

1.0f, 1.0f,

0.0f, 0.0f, /\* Bottom. \*/

0.0f, 1.0f,

1.0f, 1.0f,

1.0f, 0.0f

};

We need to make sure that we have a texture coordinate per vertex. This is so that OpenGL ES knows how it should map the image onto the triangles we are going to be drawing. We can define these coordinates the same we did the vertices; this means we only have to use 4 per face again. Going from 0 (bottom left corner) to 1 (top right corner) as it did previously. For example, the top left vertex for the front face (-1.0, 1.0, 1.0) matches the top left texture coordinate (0.0, 1.0).

## Enabling attributes and passing sampler locations

/\* [enableAttributes] \*/

**glVertexAttribPointer**(textureCordLocation, 2, GL\_FLOAT, GL\_FALSE, 0, textureCords);

**glEnableVertexAttribArray**(textureCordLocation);

**glUniformMatrix4fv**(projectionLocation, 1, GL\_FALSE,projectionMatrix);

**glUniformMatrix4fv**(modelViewLocation, 1, GL\_FALSE, modelViewMatrix);

/\* Set the sampler texture unit to 0. \*/

**glUniform1i**(samplerLocation, 0);

The last thing we need to do is alter our *“renderFrame”* function to take all these updates into account and apply our texture. Here, we enable the texture coordinate attributes and show OpenGL where they are. Then, finally we set the sampler from the fragment shader, using GL\_TEXTURE0.

We can now build and run the project, resulting in our application displaying a textured cube!