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

**LAB 4**

**Simple Cube Rotation**

**Issue 1.0**

1. Contents

[1. Introduction 1](#_Toc36039603)

[2. Project setup 1](#_Toc36039604)

[3. Matrices 1](#_Toc36039605)

[3.1 General information 1](#_Toc36039606)

[3.2 Identity function 2](#_Toc36039607)

[3.3 Translation function 3](#_Toc36039608)

[3.4 Matrix multiplication 3](#_Toc36039609)

[3.5 Scaling 4](#_Toc36039610)

[3.6 Rotation 4](#_Toc36039611)

[3.7 Update “Matrix.h” 5](#_Toc36039612)

[3.8 Projection function 6](#_Toc36039613)

[4 Shaders 7](#_Toc36039614)

[5 Extension of setup graphics (native-lib.cpp) 8](#_Toc36039615)

[6 Creating the Cube 9](#_Toc36039616)

[7 Drawing the scene 10](#_Toc36039617)

# Introduction

In the previous lab, we looked at how to create a simple 2D triangle to display in our application. In this lab, we will look at developing more complex graphics such as 3D cubes and how we can manipulate them to make them move, rotate, etc. The outcome of this lab should be a 3D spinning cube in our application.

# Project setup

For this project, we will carry on from the triangle example. You can either just continue with your current project or make a copy of the project and call it “SimpleCube”. For this tutorial, we have made a new copy. If you have any issues working out how to do this, please consult the other tutorials. Ensure that you rename parts that reference the name of the previous project such as “SimpleTriangle”.

We need to add two new native files. We could have placed all our new functions in the native-lib.cpp, but for ease of use and readability, we have separated them out. Navigate to the cpp folder, right-click, and select new c++ file. The file name should be Matrix.cpp, also make sure to attach a header file to it which will be Matrix.h. We also need to edit the CMakelists.txtfile; it should look like this:

**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)

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

# Matrices

## General information

We have already covered matrices in the lecture, but this section will act as a reminder. A matrix is an array of numbers typically formatted in a grid. In this tutorial, we will be dealing with a 4x4 matrix.

We will touch on three matrices that will be used when developing our application/graphics. These are:

* Model Matrix
* View Matrix
* Projection Matrix

The model matrix is typically concerned with where an object should be drawn to the screen; this helps with movement, scaling, and rotation.

The view matrix is concerned with handling the movement of the camera and how we will see the scene.

Finally, the projection matrix is concerned with the depth of the scene, making objects that are closer to us appear larger than those in the background.

## Identity function

We are now going to add code to our “Matrix.cpp” file that we created during the project setup. We will begin by coding a “matrixIdentityFunction”. This function creates an identity matrix, which is a diagonal line of 1s from the top left of the matrix to the bottom right, with 0s filling the rest. When this matrix is used, it will make sure nothing is happening, such as translation, rotation, or scaling.

**void** **matrixIdentityFunction**(**float**\* matrix)

{

**if**(matrix == NULL)

{

**return**;

}

matrix[0] = 1.0f;

matrix[1] = 0.0f;

matrix[2] = 0.0f;

matrix[3] = 0.0f;

matrix[4] = 0.0f;

matrix[5] = 1.0f;

matrix[6] = 0.0f;

matrix[7] = 0.0f;

matrix[8] = 0.0f;

matrix[9] = 0.0f;

matrix[10] = 1.0f;

matrix[11] = 0.0f;

matrix[12] = 0.0f;

matrix[13] = 0.0f;

matrix[14] = 0.0f;

matrix[15] = 1.0f;

}

OpenGL places the elements into a matrix differently to conventional mathematics. Instead of elements going up along the row, they are placed in columns, e.g.:

[0] [4] [8] [12]

[1] [5] [9] [13]

[2] [6] [10] [14]

[3] [7] [11] [15]

## Translation function

Translation is the act of moving something in the X, Y, or Z axis; or a mixture of the three. We only need to alter the values of the numbers in the furthest right column in order to translate the object. Element 12 (referring to the diagram above) determines how far the object will be moved on the X axis. Element 13 determines the Y axis, and element 14 controls the Z axis. Finally, element 15 will need to be set 1 to in order for the math to work; however, this is already the case.

**void** **matrixTranslate**(**float**\* matrix, **float** x, **float** y, **float** z)

{

**float** temporaryMatrix[16];

matrixIdentityFunction(temporaryMatrix);

temporaryMatrix[12] = x;

temporaryMatrix[13] = y;

temporaryMatrix[14] = z;

matrixMultiply(matrix,temporaryMatrix,matrix);

}

The function starts by creating a temporary matrix and filling it with the identity matrix. We then assign the changes we want in the coordinates to their respective positions in the matrix, as discussed previously. Finally, we multiply the temporary matrix with the matrix we wish to translate and then assign the result back to the matrix.

## Matrix multiplication

As seen in the translate function, there is a call to a “matrixMultiply” function. We will need to implement this ourselves for it all to work. Before we write the function itself, we will refresh our memory from the lecture and look at matrix multiplication.

Suppose we have two matrices that we wish to multiply together:

3 4 1 7 1 9

1 7 3 x 2 3 4

5 6 2 5 1 6

We need to multiply each element in the rows of matrix 1 with each of the elements of the corresponding columns in matrix 2. So, in this case, we get:

[(3\*7)+(4\*2)+(1\*5)=34] [(3\*1)+(4\*3)+(1\*1)=16] [(3\*9)+(4\*4)+(1\*6)=49] [34] [16] [49]

[(1\*7)+(7\*2)+(3\*5)=36] [(1\*1)+(7\*3)+(3\*1)=25] [(1\*9)+(7\*4)+(3\*6)=55] = [36] [25] [55]

[(5\*7)+(6\*2)+(2\*5)=57] [(5\*1)+(6\*3)+(2\*1)=25] [(5\*9)+(6\*4)+(2\*6)=81] [57] [25] [81]

We must create a function that will carry out this mathematical operation for us.

**void** **matrixMultiply**(**float**\* destination, **float**\* operand1, **float**\* operand2)

{

**float** theResult[16];

**int** row, column = 0;

**int** i,j = 0;

**for**(i = 0; i < 4; i++)

{

**for**(j = 0; j < 4; j++)

{

theResult[4 \* i + j] = operand1[j] \* operand2[4 \* i] + operand1[4 + j] \* operand2[4 \* i + 1] + operand1[8 + j] \* operand2[4 \* i + 2] + operand1[12 + j] \* operand2[4 \* i + 3];

}

}

**for**(**int** i = 0; i < 16; i++)

{

destination[i] = theResult[i];

}

}

This function doesn’t need any explanation as it does not really relate to OpenGL ES; it simply performs the multiplication of two matrices as previously described. However, the function needs to be placed before the translate function, so it is defined before it is to be called.

## Scaling

Now we need to write a function for another transformation; this will be for scaling. Instead of altering three of the elements in the right column like we did for translation, when trying to scale, we alter the values that lay on the diagonal line going from top left to bottom right.

**void** **matrixScale**(**float**\* matrix, **float** x, **float** y, **float** z)

{

**float** tempMatrix[16];

matrixIdentityFunction(tempMatrix);

tempMatrix[0] = x;

tempMatrix[5] = y;

tempMatrix[10] = z;

matrixMultiply(matrix, tempMatrix, matrix);

}

We assign element 0 the scaling of X, element 5 the scaling of Y, and element 10 the scaling of Z. This has to be done like this for the math to work. As with the other functions we create temporary matrix so that we do not tamper with the original matrix/data. We then multiply them together.

## Rotation

In order to perform rotations, there will be some trigonometry involved; this won’t be touched in the lab notes. But we have looked at these calculations in the lectures so feel free to refer back to them to understand these functions a bit better.

The following code snippets offer the code for rotation in the X, Y, and Z axis.

**void** **matrixRotateX**(**float**\* matrix, **float** angle)

{

**float** tempMatrix[16];

matrixIdentityFunction(tempMatrix);

tempMatrix[5] = **cos**(matrixDegreesToRadians(angle));

tempMatrix[9] = -**sin**(matrixDegreesToRadians(angle));

tempMatrix[6] = **sin**(matrixDegreesToRadians(angle));

tempMatrix[10] = **cos**(matrixDegreesToRadians(angle));

matrixMultiply(matrix, tempMatrix, matrix);

}

**void** **matrixRotateY**(**float** \*matrix, **float** angle)

{

**float** tempMatrix[16];

matrixIdentityFunction(tempMatrix);

tempMatrix[0] = **cos**(matrixDegreesToRadians(angle));

tempMatrix[8] = **sin**(matrixDegreesToRadians(angle));

tempMatrix[2] = -**sin**(matrixDegreesToRadians(angle));

tempMatrix[10] = **cos**(matrixDegreesToRadians(angle));

matrixMultiply(matrix, tempMatrix, matrix);

}

**void** **matrixRotateZ**(**float** \*matrix, **float** angle)

{

**float** tempMatrix[16];

matrixIdentityFunction(tempMatrix);

tempMatrix[0] = **cos**(matrixDegreesToRadians(angle));

tempMatrix[4] = -**sin**(matrixDegreesToRadians(angle));

tempMatrix[1] = **sin**(matrixDegreesToRadians(angle));

tempMatrix[5] = **cos**(matrixDegreesToRadians(angle));

matrixMultiply(matrix, tempMatrix, matrix);

}

## Update “Matrix.h”

We must update the “Matrix.h” file to include our new functions:

#ifndef MATRIX\_H  
#define MATRIX\_H  
#include <math.h>  
  
void matrixIdentityFunction(float\* matrix);  
  
void matrixTranslate(float\* matrix, float x, float y, float z);  
  
void matrixMultiply(float\* destination, float\* operand1, float\* operand2);  
  
void matrixFrustum(float\* matrix, float left, float right, float bottom, float top, float zNear, float zFar);  
  
void matrixPerspective(float\* matrix, float fieldOfView, float aspectRatio, float zNear, float zFar);  
  
void matrixRotateX(float\* matrix, float angle);  
  
void matrixRotateY(float\* matrix, float angle);  
  
void matrixRotateZ(float\* matrix, float angle);  
  
void matrixScale(float\* matrix,float x, float y, float z);  
  
float matrixDegreesToRadians(float degrees);  
  
#endif  
  
class Matrix {  
  
};

## Projection function

The next function we need to add is the “matrixPerspective” function.

**void** **matrixPerspective**(**float**\* matrix, **float** fieldOfView, **float** aspectRatio, **float** zNear, **float** zFar)

{

**float** ymax, xmax;

ymax = zNear \* tanf(fieldOfView \* M\_PI / 360.0);

xmax = ymax \* aspectRatio;

matrixFrustum(matrix, -xmax, xmax, -ymax, ymax, zNear, zFar);

}

The first parameter that we send to the function is the matrix that we wish to do the projection with. The second parameter is the field of view; this is the angle that you should be able to see through. 45 degrees is standard, but when you have finished the tutorial, feel free to adjust to see the effect.

The next parameter is the aspect ratio; this is usually the width/height of your screen or surface. This helps the application to look correct on a variety of devices, which is very important considering the vast array of different devices that are available in the mobile space.

The final two parameters are called zNear and zFar. zNear is how close an object has to be to the camera before it disappears. zFar is the opposite, how far away an object should be from the camera before it is no longer drawn.

This function calls a *“*matrixFrustum” function that actually creates the viewing frustum, which is what you will see of the scene. We will only provide the code and not go into too much detail regarding this. This function needs to be before the perspective function.

**void** **matrixFrustum**(**float**\* matrix, **float** left, **float** right, **float** bottom, **float** top, **float** zNear, **float** zFar)

{

**float** temp, xDistance, yDistance, zDistance;

temp = 2.0 \*zNear;

xDistance = right - left;

yDistance = top - bottom;

zDistance = zFar - zNear;

matrixIdentityFunction(matrix);

matrix[0] = temp / xDistance;

matrix[5] = temp / yDistance;

matrix[8] = (right + left) / xDistance;

matrix[9] = (top + bottom) / yDistance;

matrix[10] = (-zFar - zNear) / zDistance;

matrix[11] = -1.0f;

matrix[14] = (-temp \* zFar) / zDistance;

matrix[15] = 0.0f;

}

Finally, we need to add a simple mathematical function that will convert degrees to radians for us (needs to be placed before the rotation functions):

float matrixDegreesToRadians(float degrees)

{

return M\_PI \* degrees / 180.0f;

}

# Shaders

We previously set up our shaders; we are now going to alter this to have some more exciting visuals that we need to produce. We will start by looking at the vertex shader code:

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

"attribute vec4 vertexPosition;\n"

"attribute vec3 vertexColour;\n"

"varying vec3 fragColour;\n"

"uniform mat4 projection;\n"

"uniform mat4 modelView;\n"

"void main()\n"

"{\n"

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

" fragColour = vertexColour;\n"

"}\n";

The first line declares the input position for the vertices and the second line associates a colour with each vertex. We then declare a varying. This allows the transferal of values into the fragment shader such as fragment colour.

Next, two uniform variables are declared: “projection” and “modelView”. A uniform is something static that can be applied to multiple things, or everything, not just one specific attribute. It acts like a global variable and can be accessed by both the fragment shader and the vertex shader; however, it is read only. We will store our matrices here that were discussed earlier on. In this case, the model and view matrices will be combined into “modelView”.

In the main function of the shader code, we assign a value to the “gl\_Position”attribute, but instead of simply passing in the vertexPosition like previously, we multiply it by the projection and modelView uniforms. The ordering of this is important. The final line sets the varying to the attribute that we wish to use in our fragment shader.

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

"precision mediump float;\n"

"varying vec3 fragColour;\n"

"void main()\n"

"{\n"

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

"}\n";

The fragment shader remains fairly similar to the previous code. Note how we have the varying “fragColour”from the vertex shader, and we assign it to the first three elements of “gl\_FragColor”, setting the final element as 1.0 (alpha).

# Extension of setup graphics (native-lib.cpp)

We have now added more inputs to our shaders, which means they need to be told where they can expect this data! To do this, we need to alter out “setupGraphics” function.

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

{

simpleCubeProgram = createProgram(glVertexShader, glFragmentShader);

**if** (simpleCubeProgram == 0)

{

LOGE ("Could not create program");

**return** **false**;

}

vertexLocation = **glGetAttribLocation**(simpleCubeProgram, "vertexPosition");

vertexColourLocation = **glGetAttribLocation**(simpleCubeProgram, "vertexColour");

projectionLocation = **glGetUniformLocation**(simpleCubeProgram, "projection");

modelViewLocation = **glGetUniformLocation**(simpleCubeProgram, "modelView");

/\* Setup the perspective \*/

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

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

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

**return** **true**;

}

As we can see, we locate the position of the vertex position and colour using “glGetAttribLocation” and assign it to some local variables. For the uniform matrices, we have to use the “glGetUniformLocation” function in order locate the data and then assign it to some more local variables. As our scene will be simple, we only need to set the perspective once.

The final alteration of this function is with “glEnable”. Now that we are creating a 3D cube, we tell OpenGL ES that we need to take depth into consideration. This means that only the front-facing fragments will be drawn.

We also need to declare some variables for these functions to work with:

GLuint simpleCubeProgram;

GLuint vertexLocation;

GLuint vertexColourLocation;

GLuint projectionLocation;

GLuint modelViewLocation;

float projectionMatrix[16];

float modelViewMatrix[16];

float angle = 0;

Also include “Matrix.h” at the top of the file!

# Creating the Cube

We now need to set up the vertices for our cube model. For this example, the cube will be made up of twelve triangles, each face of the cube being made up of two triangles.

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

};

Models will usually be generated by modeling software and tools; however, for the point of understanding how it all works, we are writing the vertices by hand. Now that it is a 3D model, we include the Z value in each trio to convey the depth.

We then define the colours for each of the vertices (rasterizing), using values for RGB.

GLfloat colour[] = {1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f,

1.0f, 1.0f, 0.0f,

1.0f, 1.0f, 0.0f,

1.0f, 1.0f, 0.0f,

0.0f, 1.0f, 1.0f,

0.0f, 1.0f, 1.0f,

0.0f, 1.0f, 1.0f,

0.0f, 1.0f, 1.0f,

1.0f, 0.0f, 1.0f,

1.0f, 0.0f, 1.0f,

1.0f, 0.0f, 1.0f,

1.0f, 0.0f, 1.0f

};

Instead of defining every vertex of every triangle, we define all of the unique points. This reduces the number of vertices we have to write out. We then pass in a group of indices that maps the vertices to their respective triangles.

GLushort indices[] = {0, 2, 3, 0, 1, 3, 4, 6, 7, 4, 5, 7, 8, 9, 10, 11, 8, 10, 12, 13, 14, 15, 12, 14, 16, 17, 18, 16, 19, 18, 20, 21, 22, 20, 23, 22};

# Drawing the scene

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

{

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

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

matrixIdentityFunction(modelViewMatrix);

matrixRotateX(modelViewMatrix, angle);

matrixRotateY(modelViewMatrix, angle);

matrixTranslate(modelViewMatrix, 0.0f, 0.0f, -10.0f);

**glUseProgram**(simpleCubeProgram);

**glVertexAttribPointer**(vertexLocation, 3, GL\_FLOAT, GL\_FALSE, 0, cubeVertices);

**glEnableVertexAttribArray**(vertexLocation);

**glVertexAttribPointer**(vertexColourLocation, 3, GL\_FLOAT, GL\_FALSE, 0, colour);

**glEnableVertexAttribArray**(vertexColourLocation);

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

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

**glDrawElements**(GL\_TRIANGLES, 36, GL\_UNSIGNED\_SHORT, indices);

angle += 1;

**if** (angle > 360)

{

angle -= 360;

}

}

As you can see, the “renderFrame”function has grown a bit since the previous tutorial. We clear out the depth and colour buffer again (depth buffer being important now as it is 3D, no longer 2D). We then set the “modelViewMatrix” using our identity matrix. Next, we call matrixRotate X and Y to display the functionality we have provided for the cube. In order for us to see it, we move the cube 10 units back so that the camera is not set inside the cube. The next section of code is the same as it was in the previous tutorial.

After this, we link the projection and modelView matrix with their counterparts in the shader.

We cannot use “glDrawArrays” like last time because it expects a full set of defined vertices, but as mentioned earlier, we only stated all the unique ones. So, we use a function called “glDrawElements”.This function takes into account how the vertices can be associated to more than one triangle using our indices array. This will draw our cube!

The final little bit of code makes sure that the angle is changing when we use our rotate functions.

You can now build and run the application to see a 3D rotating cube.