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

**LAB 7**

**Normal Mapping**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc36050315)

[2 File loading 1](#_Toc36050316)

[3 Texture loading 3](#_Toc36050317)

[4 Normals, tangents, and binormals 4](#_Toc36050318)

[5 Shaders 7](#_Toc36050319)

[5.1 Vertex shader 7](#_Toc36050320)

[5.2 Fragment shader 8](#_Toc36050321)

[6 More code 9](#_Toc36050322)

# Introduction

When dealing with mobile and computer graphics, there is an important decision about complexity and performance. In order for graphics to look realistic, they require many vertices, etc. However, this is taxing to the performance aspect of the application and can make it run much slower. Normal mapping provides us with the ability to generate graphics that look complex, whilst actually being more efficient in terms of performance.

This works by combining simple geometry with surface normals, providing a more complex version of the same geometry. The process would be something like:

* Create more complex geometry, including the direction of all the surface normals.
* Save the normal in a texture.
* Apply the texture to a simple geometry, making it look more realistic.

Ensure that you rename parts that reference the name of the previous project such as “SimpleTriangle” or rename the project if you are working on one file.

# File loading

Previously, we looked at textures and manually wrote our own in the code; however, this was a small texture and did not require much effort. But for larger and more realistic textures, this is not practical. One way around this is including the texture as an asset in the apk. This will compress the texture and mean that it is always available to the application.

Textures can simply be stored in an *“assets*” folder in the Android Studio project to be included in the apk. To create an asset folder, press alt+insert when in the project view and navigate to folder-> asset folder. This will automatically place this folder where it should be located (this is in app/src/main).

The difficult part is getting them out of the apk. This must be done in the Java side. We will be modifying ***“NormalMapping.java”*** or the main activity of your application***:***

**import** android.os.Bundle;

**import** android.app.Activity;

**import** android.util.Log;

**import** android.content.res.AssetManager;

We start by importing the following libraries

**private** **static** String *LOGTAG* = "NormalMapping";

**private** **static** String *assetDirectory* = **null**;

**protected** TutorialView graphicsView;

**private** **static** android.content.Context *applicationContext* = **null**;

Then, we add two new variables to the code; these are *“assetDirectory”* and *“applicationContext”.* One is a string that contains the location of where we wish to store our application data and the other is an instance of *“android.content.context”* which we will use to access the AssetManager. Access to the AssetManager will allow us to get the files out of the apk.

Next, we alter the code in our *“onCreate”* method:

applicationContext = getApplicationContext();

*assetDirectory* = *applicationContext*.getFilesDir().getPath() + "/";

extractAsset("normalMap256.raw");

This portion of code begins by initializing the *“applicationContext”* variable by calling the *“getApplicationContext”* function (this is provided by Android). We then use it to get the path to where we should store everything. We assign this path to the “assetDirectory” string for future use. Then, we call the *“extractAsset”* function (still to be written) with the name of the asset we wish to retrieve.

Now we need to define the *“extractAsset”* function. This function will check if the file we wish to extract is on the filesystem. If it is, then we do not need to do anything as the file is already extracted. However, if it is not present, we then need to extract the file.

**private** **void** extractAsset(String assetName)

{

File fileTest = **new** File(*assetDirectory* + assetName);

**if**(fileTest.exists())

{

Log.*d*(*LOGTAG*,assetName + " already exists no extraction needed\n");

}

**else**

{

Log.*d*(*LOGTAG*, assetName + " doesn't exist extraction needed \n");

The next bit of code is called when the file needs extraction:

try

{

RandomAccessFile out = new RandomAccessFile(*assetDirectory* + assetName,"rw");

AssetManager am = *applicationContext*.getResources().getAssets();

/\* [tryCatchExtractAsset] \*/

/\* [readWriteFile] \*/

InputStream inputStream = am.open(assetName);

byte buffer[] = new byte[1024];

int count = inputStream.read(buffer, 0, 1024);

while (count > 0)

{

out.write(buffer, 0, count);

count = inputStream.read(buffer, 0, 1024);

}

out.close();

inputStream.close();

}

catch(Exception e)

{

Log.*e*(*LOGTAG*, "Failure in extractAssets(): " + e.toString() + " " + *assetDirectory*+assetName);

}

if(fileTest.exists())

{

Log.*d*(*LOGTAG*,"File Extracted successfully");

}

}

}

In order to extract the file to the filesystem, we start by creating a file that will act as the extracted version. Next, we get an instance of AssetManager (using the application context we defined earlier). We then have some generic code that enables us to read and write to the output file. Once this is finished, we close the files and ensure any exceptions are caught, finally testing to see if we successfully extracted the file.

# Texture loading

Now that we have extracted the files, we need to edit our “loadTexture” function to load textures from files, as opposed to hardcoded values.

#include **<android/log.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\_\_)

#define TEXTURE\_WIDTH 256

#define TEXTURE\_HEIGHT 256

#define CHANNELS\_PER\_PIXEL 3

GLubyte \* theTexture;

GLuint loadTexture()

{

static GLuint textureId;

theTexture = (GLubyte \*)malloc(sizeof(GLubyte) \* TEXTURE\_WIDTH \* TEXTURE\_HEIGHT \* CHANNELS\_PER\_PIXEL);

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);

FILE \* theFile = fopen("/data/data/com.arm.malideveloper.openglessdk.normalmapping/files/normalMap256.raw", "r");

if(theFile == NULL)

{

LOGE("Failure to load the texture");

return 0;

}

fread(theTexture, TEXTURE\_WIDTH \* TEXTURE\_HEIGHT \* CHANNELS\_PER\_PIXEL, 1, theFile);

/\* Load the texture. \*/

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, TEXTURE\_WIDTH, TEXTURE\_HEIGHT, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, theTexture);

/\* Set the filtering mode. \*/

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

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

free(theTexture);

return textureId;

}

*Note that some of the code is specific to file names/paths and should be adapted for your own environment.*

Instead of an array that contains all the data for our texture, we now just define a pointer that we wish to fill with the data. The initial part of the function is still the same as it was, except for opening the file. We have hardcoded this path in for simplicity and understanding.

After the file has been opened, we malloc some space for it. Due to the file being raw, the data is presented exactly the same way as the texture example. RGB values, each a byte in size. The texture we are using is 256 x 256, so this needs to be multiplied by 3 (each pixel has 3 components). We then read the texture into the space we just allocated again, only reading 256 x 256 x 3 bytes. This information is then fed into glTexImage2D; then, we must free the space that was allocated as the texture has been copied into the GPU.

# Normals, tangents, and binormals

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 normals[] = // CODE NEEDED HERE

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

};

GLfloat tangents[] = {-1.0f, 0.0f, 0.0f, /\* Back \*/

-1.0f, 0.0f, 0.0f,

-1.0f, 0.0f, 0.0f,

-1.0f, 0.0f, 0.0f,

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

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

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

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, -1.0f, /\* Right \*/

0.0f, 0.0f, -1.0f,

0.0f, 0.0f, -1.0f,

0.0f, 0.0f, -1.0f,

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

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f, /\* Bottom \*/

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f,

1.0f, 0.0f, 0.0f

};

GLfloat biNormals[] = { 0.0f, 1.0f, 0.0f, /\* Back \*/

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

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

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

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

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f, /\* Right \*/

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 1.0f, 0.0f,

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

0.0f, 0.0f, -1.0f,

0.0f, 0.0f, -1.0f,

0.0f, 0.0f, -1.0f,

0.0f, 0.0f, 1.0f, /\* Bottom \*/

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 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

};

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

In this case, the vertices are the same as they were in the cube tutorials, 4 defined per face to generate a cube. **The code for the normal array is not provided; you need to complete this section**. The array is different to the lighting example. The normal are defined as 4 per face; unlike the lighting example, they are all the same and all point perpendicular to the face. The top face has 4 pointing in the Y direction; the back face has 4 pointing in the negative Z direction, and so on. The next array we define is for the colours; this is also the same as the simple cube example. A solid colour is defined for each face.

The first new section that we must add is the *“tangents”* array, which holds the tangents to each of the faces. Much like the normals array, 4 are defined per face and they are all the same. The front face has a tangent positive in the X direction; this lies on the plane of the face and is perpendicular to the normal, which is defined in positive Z direction, etc.

Next, we need to define the biNormal array. This has to be perpendicular to both, the normal and the tangent. For the front, back, right, and left faces, this is the positive Y direction. The top and bottom are positive in the X direction.

Finally, we define the texture coordinates, which are the same as the textured cube example.

# Shaders

## Vertex shader

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

"attribute vec4 vertexPosition;\n"

"attribute vec2 vertexTextureCord;\n"

"attribute vec3 vertexNormal;\n"

"attribute vec3 vertexColor; \n"

"attribute vec3 vertexTangent;\n"

"attribute vec3 vertexBiNormal;\n"

"varying vec2 textureCord;\n"

"varying vec3 varyingColor; \n"

"varying vec3 inverseLightDirection;\n"

"varying vec3 inverseEyeDirection;\n"

"uniform mat4 projection;\n"

"uniform mat4 modelView;\n"

"void main()\n"

"{\n"

" vec3 worldSpaceVertex =(modelView \* vertexPosition).xyz;"

" vec3 transformedVertexNormal = normalize((modelView \* vec4(vertexNormal, 0.0)).xyz);"

" inverseLightDirection = normalize(vec3(0.0, 0.0, 1.0));\n"

" inverseEyeDirection = normalize((vec3(0.0, 0.0, 1.0)- worldSpaceVertex ).xyz);\n"

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

" textureCord = vertexTextureCord;\n"

" varyingColor = vertexColor;\n"

" vec3 transformedTangent = normalize((modelView \* vec4(vertexTangent, 0.0)).xyz);\n"

" vec3 transformedBinormal = normalize((modelView \* vec4(vertexBiNormal, 0.0)).xyz);\n"

" mat3 tangentMatrix = mat3(transformedTangent, transformedBinormal, transformedVertexNormal);\n"

" inverseLightDirection =inverseLightDirection \* tangentMatrix;\n"

" inverseEyeDirection = inverseEyeDirection \* tangentMatrix;\n"

"}\n";

You may notice there is slightly less in the vertex shader this time than the lighting tutorial. This is because some of the calculations have been moved to the fragment shader. At the start, we declare all our attributes, etc. as we have in previous examples. Then, the first line of the main function creates a temporary variable that is equal to the current vertex that has been transformed by the *modelView* matrix. This will be used to help work out the eye vector from the current vertex.

The next line of code transforms the vertex normal. This is similar to the calculation performed in the lighting tutorial. Despite the complex normals we are going to be using are defined in the normal map, we will still need to use the real vertex normal in order to create the matrix that will move the light and eye vectors into the tangent space.

The next two lines provide definitions of our variables *“inverseLightDirection”* and *“inverseEyeDirection”.* The light direction variable uses the same code as was used in the lighting tutorial. Because we are dealing with directional light, we do not require a vector from the vertex to the light, as all light is treated as being parallel direction vectors.

The next three lines are standard of what we have seen in the previous tutorials, so we will not explain them. Next, we define *“transformedTangent”* and *“transformedBinormal”*. We normalize the initial tangents and biNormals using the modelView matrix and assign it.

The final three lines are responsible for converting the eyeVector and lightVector into tangent space. We start by creating a transformation matrix that consists of the tangent, binormal, and normal vectors. We then multiply the light and the eye by this new matrix.

## Fragment shader

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

"precision mediump float;\n"

"uniform sampler2D texture;\n"

"varying vec2 textureCord;\n"

"varying vec3 varyingColor;\n"

"varying vec3 inverseLightDirection;\n"

"varying vec3 inverseEyeDirection;\n"

"varying vec3 transformedVertexNormal;\n"

"void main()\n"

"{\n"

" vec3 fragColor = vec3(0.0,0.0,0.0); \n"

" vec3 normal = texture2D(texture, textureCord).xyz;"

" normal = normalize(normal \* 2.0 -1.0);"

/\* Calculate the diffuse component. \*/

" vec3 diffuseLightIntensity = vec3(1.0, 1.0, 1.0);\n"

" float normalDotLight = max(0.0, dot(normal, inverseLightDirection));\n"

" fragColor += normalDotLight \* varyingColor \*diffuseLightIntensity;\n"

/\* Calculate the ambient component. \*/

" vec3 ambientLightIntensity = vec3(0.1, 0.1, 0.1);\n"

" fragColor += ambientLightIntensity \* varyingColor;\n"

/\* Calculate the specular component. \*/

" vec3 specularLightIntensity = vec3(1.0, 1.0, 1.0);\n"

" vec3 vertexSpecularReflectionConstant = vec3(1.0, 1.0, 1.0);\n"

" float shininess = 2.0;\n"

" vec3 lightReflectionDirection = reflect(vec3(0) - inverseLightDirection, normal);\n"

" float normalDotReflection = max(0.0, dot(inverseEyeDirection, lightReflectionDirection));\n"

" fragColor += pow(normalDotReflection, shininess) \* vertexSpecularReflectionConstant \* specularLightIntensity;\n"

" /\* Make sure the fragment colour is between 0 and 1. \*/"

" clamp(fragColor, 0.0, 1.0);\n"

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

"}\n";

The fragment shader code is very similar to what we had previously in the vertex shader for the lighting tutorial. The first line of the main function sets the starting fragment colour to black. The next two lines acquire the normal from the normal map texture. Then, everything else is essentially identical to the lighting example.

# More code

We need to define all the variables needed in the ***native-lip.cpp***:

GLuint vertexLocation;

GLuint samplerLocation;

GLuint projectionLocation;

GLuint modelViewLocation;

GLuint textureCordLocation;

GLuint colorLocation;

GLuint textureId;

GLuint vertexNormalLocation;

GLuint tangentLocation;

GLuint biNormalLocation;

These are the variable names that we use throughout the example.

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

textureCordLocation = **glGetAttribLocation**(glProgram, "vertexTextureCord");

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

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

samplerLocation = **glGetUniformLocation**(glProgram, "texture");

vertexNormalLocation = **glGetAttribLocation**(glProgram, "vertexNormal");

colorLocation = **glGetAttribLocation**(glProgram, "vertexColor");

tangentLocation = **glGetAttribLocation**(glProgram, "vertexTangent");

biNormalLocation = **glGetAttribLocation**(glProgram, "vertexBiNormal");

These are the calls we make to obtain the locations of data.

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

**glEnableVertexAttribArray**(vertexLocation);

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

**glEnableVertexAttribArray**(textureCordLocation);

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

**glEnableVertexAttribArray**(colorLocation);

**glVertexAttribPointer**(vertexNormalLocation, 3, GL\_FLOAT, GL\_FALSE, 0, normals);

**glEnableVertexAttribArray**(vertexNormalLocation);

**glVertexAttribPointer**(biNormalLocation, 3, GL\_FLOAT, GL\_FALSE, 0, biNormals);

**glEnableVertexAttribArray**(biNormalLocation);

**glVertexAttribPointer**(tangentLocation, 3, GL\_FLOAT, GL\_FALSE, 0, tangents);

**glEnableVertexAttribArray**(tangentLocation);

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

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

These are the lines that supply data in the render function!