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

**LAB 9**

**Mipmapping and Compressed Textures**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc36110844)

[2 New texture function 1](#_Toc36110845)

[3 Loading textures 2](#_Toc36110846)

[4 Adjusting the rest of the code 2](#_Toc36110847)

[5 Compressed texture loading 4](#_Toc36110848)

[6 Final bit of code 5](#_Toc36110849)

# Introduction

As mentioned at the start of the previous lab, bandwidth is a major issue when developing for mobile devices. Bandwidth is a limited resource, and we wish to reduce it as much as we can. In this lab, we will look at how mipmapping and compressed textures combat bandwidth usage.

In the lectures, we have covered the concept of mipmapping and also compressed textures, but as a brief reminder:

* Mipmapping is a technique that takes an original, full-scale, high resolution image and generates several lower-res images to accompany the original image in a set.
* We can provide OpenGL ES with compressed textures that have a much smaller file size than uncompressed version, therefore saving a lot of bandwidth.

Up until now, we have used *“void glGenerateMipmap(GLenum target)”* to load a GL\_TEXTURE\_2D. But in this lab, we will be loading our mipmapped textures manually.

# New texture function

void loadTexture( const char \* texture, unsigned int level, unsigned int width, unsigned int height)

{

 GLubyte \* theTexture;

 theTexture = (GLubyte \*)malloc(sizeof(GLubyte) \* width \* height \* CHANNELS\_PER\_PIXEL);

 FILE \* theFile = fopen(texture, "r");

 if(theFile == NULL)

 {

 LOGE("Failure to load the texture");

 return;

 }

 fread(theTexture, width \* height \* CHANNELS\_PER\_PIXEL, 1, theFile);

 /\* Load the texture. \*/

 glTexImage2D(GL\_TEXTURE\_2D, level, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, theTexture);

 /\* Set the filtering mode. \*/

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

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

 free(theTexture);

}

We have modified the “loadTexture” function from the previous iterations.

We modified the function so that it now takes in the file name of the texture, as well as the level, width, and height. This is because we will need to be able to load a series of textures using the same function. When we are defining our textures for mipmapping, we have to provide the “level” argument in our call to *“glTextImage2D”.* The base texture (regular resolution) is level 0.

Each texture is expected by OpenGL ES to be half the size of the previous one: 512 – 256 – 128, etc. This has to go down to and include a 1x1 texture. If not all levels are provided, then OpenGL ES will throw errors.

Another change we have made is to the *“glTexParameteri”* call, which now includes mipmapping. We use “GL\_NEAREST\_MIPMAP\_NEAREST” which means that is uses the mipmap that is closest in terms of size to the object without any interpolation between the two. It also uses the pixel nearest to the desired pixel in the texture, again with no interpolation.

# Loading textures

 glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

 /\* Generate a texture object. \*/

 glGenTextures(2, textureIds);

 /\* Activate a texture. \*/

 glActiveTexture(GL\_TEXTURE0);

 /\* Bind the texture object. \*/

 glBindTexture(GL\_TEXTURE\_2D, textureIds[0]);

 /\* Load the Texture. \*/

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level0.raw", 0, 512, 512);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level1.raw", 1, 256, 256);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level2.raw", 2, 128, 128);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level3.raw", 3, 64, 64);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level4.raw", 4, 32, 32);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level5.raw", 5, 16, 16);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level6.raw", 6, 8, 8);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level7.raw", 7, 4, 4);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level8.raw", 8, 2, 2);

 loadTexture("/data/data/com.arm.malideveloper.mipmapping/files/level9.raw", 9, 1, 1);

This code will change depending on the name of your project package.

We have added some code to load the textures mipmaps in our *“setupGraphics”* function in ***“native-lib.cpp”***. We start by generating two texture IDs. This is because we will use the second for the compressed textures later on. We can see that we load a texture for each resolution version of the mipmaps, all the way from the full 512 x 512 down to 1x1. Don’t forget to instantiate texture Ids array globally

These textures were added using the asset manager in the Android SDK

# Adjusting the rest of the code

GLfloat squareVertices[] = { -1.0f, 1.0f, 1.0f,

 1.0f, 1.0f, 1.0f,

 -1.0f, -1.0f, 1.0f,

 1.0f, -1.0f, 1.0f,

 };

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

 1.0f, 1.0f,

 0.0f, 0.0f,

 1.0f, 0.0f,

 };

GLushort indicies[] = {0, 2, 3, 0, 3, 1};

We will need to change our visual example in order to display the effect that mipmapping has on our graphics. We will generate a single square that moves closer and further away from the screen. The mipmap will adjust as the size of the square varies.

As seen above, we can reduce a lot of the complex vertices and indices to a much simpler version.

matrixPerspective(projectionMatrix, 45, (**float**)width / (**float**)height, 0.1f, 170);

As we are moving the object further away, we need to adjust the Zfar distance in the *“matrixPerspective*” call. Otherwise, the object will get clipped before it shows some of the mipmap levels.

 **float** distance = 1;

 **float** velocity = 0.1;

 GLuint textureModeToggle = 0;

We add a few new globals too. These are: a distance global, a velocity global (how much to increase or decrease the distance by), and an integer that is used as a toggle to show whether we are using compressed textures or not.

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

Another adjustment we must make is to the translation function. We need to make it take into account our new distance variable and remove the rotation functions as we are not using angle anymore.

 **glUniform1i**(samplerLocation, textureModeToggle);

 **glDrawElements**(GL\_TRIANGLES, 6, GL\_UNSIGNED\_SHORT, indicies);

 distance += velocity;

 **if** (distance > 160 || distance < 1)

 {

 velocity \*= -1;

 textureModeToggle = !textureModeToggle;

 }

The final small adjustment we have to make is how we move the object at the end of the frame. We provide a range of acceptable values, and once the distance falls outside of this range, we toggle as to whether we are using compressed textures, and the sign of the velocity to make the object move in the opposite direction. We also change the sampler location from 0 to compressed. This will have been touched on in the lectures.

# Compressed texture loading

void loadCompressedTexture( const char \* texture, unsigned int level)

{

 GLushort paddedWidth;

 GLushort paddedHeight;

 GLushort width;

 GLushort height;

 GLubyte \* textureHead = (GLubyte \*)malloc(sizeof (GLubyte) \* 16);

 GLubyte \* theTexture;

 FILE \* theFile = fopen(texture, "r");

 if(theFile == NULL)

 {

 LOGE("Failure to load the texture");

 return;

 }

 fread(textureHead, 16, 1, theFile);

 paddedWidth = (textureHead[8] << 8) | textureHead[9];

 paddedHeight = (textureHead[10] << 8) | textureHead[11];

 width = (textureHead[12] << 8) | textureHead[13];

 height = (textureHead[14] << 8) | textureHead[15];

 theTexture = (GLubyte \*)malloc(sizeof(GLubyte) \* ((paddedWidth \* paddedHeight) >> 1));

 fread(theTexture, (paddedWidth \* paddedHeight) >> 1, 1, theFile);

 /\* Load the texture. \*/

 glCompressedTexImage2D(GL\_TEXTURE\_2D, level, GL\_ETC1\_RGB8\_OES, width, height, 0, (paddedWidth \* paddedHeight) >> 1, theTexture);

 /\* Set the filtering mode. \*/

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

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

 free(textureHead);

 free(theTexture);

}

The *“loadCompressedTexture”* function is a bit more complicated than the original *“loadTexture”* function. This is because we are using the ETC file format instead of raw. We should be familiar with the ETC format as we have looked at it in the lecture.

We extract the relevant parts from the corresponding position in a typical ETC file—if you have forgotten the format of an ETC file refer back to the lecture!

Other than the extraction of data from the ETC file being different, the function is similar to the *“loadTexture”* function.

# Final bit of code

 **glActiveTexture**(GL\_TEXTURE1);

 /\* Bind the texture object. \*/

 **glBindTexture**(GL\_TEXTURE\_2D, textureIds[1]);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level0.pkm",0);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level1.pkm",1);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level2.pkm",2);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level3.pkm",3);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level4.pkm",4);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level5.pkm",5);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level6.pkm",6);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level7.pkm",7);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level8.pkm",8);

 loadCompressedTexture("/data/data/com.arm.malideveloper.mipmapping/files/level9.pkm",9);

We finally need to add the code that will make use of our newly created *“loadCompressedTexture”* function. WE call *“glActivetexture”* with “GL\_TEXTURE1” and then bind the second texture ID to it. This means that we have all the original textures on GL\_TEXTURE0 and all the compressed textures on GL\_TEXTURE1.

We mentioned that we have a variable known, which gets toggled in the frame. This means we can use this value to determine which texture unit to load from, just like we did in the glUniform1i call earlier. When you run the application, you should see a square that goes further into the distance. The first texture you should see has a 1 on it. The textures should then count up to 5. Each number responds to the mipmap level that OpenGL ES is currently using. The square should then move closer again counting down from 5 to 0. That is all there is to compressed textures and mipmapping; use these new techniques to reduce the bandwidth in all your applications.