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

**LAB 2**

**Graphics Setup**

**Issue 1.0**

Contents

[1. Introduction 1](#_Toc36032929)

[2. Application flow 1](#_Toc36032930)

[3. Creating the new project 1](#_Toc36032931)

[4. Files creation 2](#_Toc36032932)

[4.1 “TutorialView.java” class 2](#_Toc36032933)

[4.2 Altering “SimpleTriangle.java” 5](#_Toc36032934)

[4.3 Altering “NativeLibrary.java” 6](#_Toc36032935)

[4.4 Altering “native-lib.cpp” 6](#_Toc36032936)

#  Introduction

While the previous section of the lab created a foundation for the application itself, this part will focus more on setting up an environment ready to develop and display OpenGL ES graphics. Again, nothing flashy will be produced as the end product, but it is a precursor to the future labs.

#  Application flow

It is useful to know the flow of an application in order to see how the graphics fit into the process.

* When the application starts, control will be passed to the *onCreate* function in the “GraphicsSetup.java” file.
* This function creates an instance of a view class (yet to be created) and will set the display to the view class.
* When the view class is created, it will call an initialization function that will set up the surface that will be rendered. This involves setting an EGLContextFactory, EGLConfiguration, and then a Renderer.
* The renderer has two important functions. One of these is called when the surface size is initialized or changed and then the other function is called every time a new frame is ready to be drawn. These functions typically reference other functions that are written in the native side of your application. This is achieved by using a different class that holds function prototypes for all your native functions.

The native side then takes over and renders the scene.

#  Creating the new project

We will be using the code created on a previous project, a starting point for this project. So, create a new project. The project should be called *“*GraphicsSetup” and thus the company name should be called “com.arm.malideveloper.graphicssetup”**.** Remember to select Native C++.

Rename the MainActivity to “SimpleTriangle” in order to have a more intuitive title.

Once the project has been created, you can copy over the content of the native-lib.cpp file, but ensure you change the package names included to match the new package names. Copy over the NativeLibrary class and also the code from FirstNative and place it into our new SimpleTriangle class. Be sure to edit the package names and refactor the code so that it fits our new project structure.

Once we have our previous project implemented into our new application, we need to create a third Java file. Create a new file called “TutorialView.java” in the same place as our SimpleTriangle.java and NativeLibrary.java files.

#  Files creation

## “TutorialView.java” class

As seen in the application flow segment earlier, theTutorialView class is going to be used as the view that contains all of our graphics. For this to work, we need to configure the surface that we are going to be drawing to, and we will also need to set up the context so that it can make use of OpenGL ES 2.0.

As mentioned in the first lab, we must always include what package the file belongs to, so that is our first line of code. The rest of this section is imports, much like before, so that we have the required libraries to work with OpenGL, etc. The imports include things such as GLSurfaceView, which is what our class will be derived from, and a variety of other EGL and OpenGL ES libraries to enable us to make use of OpenGL as mentioned previously.

**package** com.arm.malideveloper.graphicssetup;

**import** android.content.Context;

**import** android.opengl.GLSurfaceView;

**import** javax.microedition.khronos.egl.EGL10;

**import** javax.microedition.khronos.egl.EGLConfig;

**import** javax.microedition.khronos.egl.EGLContext;

**import** javax.microedition.khronos.egl.EGLDisplay;

**import** javax.microedition.khronos.opengles.GL10;

**class** TutorialView **extends** GLSurfaceView

{

We also declare our class as an extension of GLSurfaceView; this will provide us with access to its implementation of functions and allow us to override methods if we need to. Again, this is a way of letting OpenGL do some of the work for us.

Next, we define several variables that will be important later when we set up our surface. We will need to provide some parameters; this will include what format we want the surface to be in. For this example, we are going to use RGBA 8888, which is 32-bit colour made up of 8 red bits, 8 green bits, 8 blue bits, and 8 alpha bits.

**protected** **int** redSize = 8;

**protected** **int** greenSize = 8;

**protected** **int** blueSize = 8;

**protected** **int** alphaSize = 8;

**protected** **int** depthSize = 16;

**protected** **int** sampleSize = 4;

**protected** **int** stencilSize = 0;

**protected** **int**[] value = **new** **int** [1];

We will set the int *“depthSize”* to 16 as this will later be used to display objects in the correct order, etc. The *“sampleSize”* variable is used to enable anti-aliasing, which is the smoothing of graphics. Mali platforms typically provide at least 4x anti-aliasing; we will enable it. Finally, the stencil buffer is an advanced concept and will be disabled for now.

Now we need to create the constructor for the class. The class is instantiated with a Context passed into it. OpenGL/EGL Contexts store all states associated with an instance of OpenGL. Essentially, it holds everything related to OpenGL. The constructor then calls two functions that we will define later. These two functions are related to EGL, which acts as an interface between the windowing system and the drawing commands of OpenGL ES. The last call is to set the renderer, which will also be defined later.

 **public** TutorialView(Context context)

 {

 **super**(context);

 setEGLContextFactory(**new** ContextFactory());

 setEGLConfigChooser(**new** ConfigChooser());

 setRenderer(**new** Renderer());

 }

We now define a new class, within the TutorialView class, called “ContextFactory” which is an implementation of GLSurfaceView.EGLContextFactory. This class has two functions; the first is called *“createContext”* and simply creates the Context our program will be using. The creation of the context takes in an attribute list, which in this case, consists of an “EGL\_CONTEXT\_CLIENT\_VERSION” (we assign 2 to this value) and then “EGL10.EGL\_NONE” which signifies the end of the list. This context is then returned to the caller. We also have the “destroyContext” function, which simply takes in a context and a display and then destroys it.

**private** **static** **class** ContextFactory **implements** GLSurfaceView.EGLContextFactory

{

 **public** EGLContext createContext(EGL10 egl, EGLDisplay display, EGLConfig eglConfig)

 {

 **final** **int** EGL\_CONTEXT\_CLIENT\_VERSION = 0x3098;

 **int**[] attrib\_list = {EGL\_CONTEXT\_CLIENT\_VERSION, 2, EGL10.***EGL\_NONE*** };

 EGLContext context = egl.eglCreateContext(display, eglConfig, EGL10.***EGL\_NO\_CONTEXT***, attrib\_list);

 **return** context;

 }

 **public** **void** destroyContext(EGL10 egl, EGLDisplay display, EGLContext context)

 {

 egl.eglDestroyContext(display, context);

 }

}

We now define another class within TutorialView. This class is called *“ConfigChooser”* and it handles all the variables we defined earlier on in the TutorialView class. As seen in the definition of this new class, it implements “GLSurfaceView.EGLConfigChooser”. The first function in the class is called *“chooseConfig”.* This function begins by creating an attribute list using the variables we mentioned earlier. This will be our desired configuration (Note: we did not set “EGL\_RENDERABLE\_TYPE” – this is the interface and will be set to OpenGL ES 2.0).

The next lines deal with the number of supported configurations. We call *“eglChooseConfig”* with NULL as a parameter so that it returns the number of configs available. Next, we create an EGLConfig array which will hold every configuration that matches our requirements. One issue is that despite telling EGL the specifics of our ideal configuration, it might give us something greater than the configuration we ask for.

**protected** **class** ConfigChooser **implements** GLSurfaceView.EGLConfigChooser

{

 **public** EGLConfig chooseConfig(EGL10 egl, EGLDisplay display)

 {

 **final** **int** EGL\_OPENGL\_ES2\_BIT = 4;

 **int**[] configAttributes =

 {

 EGL10.***EGL\_RED\_SIZE***, redSize,

 EGL10.***EGL\_GREEN\_SIZE***, greenSize,

 EGL10.***EGL\_BLUE\_SIZE***, blueSize,

 EGL10.***EGL\_RENDERABLE\_TYPE***, EGL\_OPENGL\_ES2\_BIT,

 EGL10.***EGL\_SAMPLES***, sampleSize,

 EGL10.***EGL\_DEPTH\_SIZE***, depthSize,

 EGL10.***EGL\_STENCIL\_SIZE***, stencilSize,

 EGL10.***EGL\_NONE***

 };

 **int**[] num\_config = **new** **int**[1];

 egl.eglChooseConfig(display, configAttributes, **null**, 0, num\_config);

 **int** numConfigs = num\_config[0];

 EGLConfig[] configs = **new** EGLConfig[numConfigs];

 egl.eglChooseConfig(display, configAttributes, configs, numConfigs, num\_config);

 **return** selectConfig(egl, display, configs);

}

The *“selectConfig”* function is the solution to this as we need to select an appropriate configuration:

**public** EGLConfig selectConfig(EGL10 egl, EGLDisplay display, EGLConfig[] configs)

 {

 **for**(EGLConfig config : configs)

 {

 **int** d = getConfigAttrib(egl, display, config, EGL10.***EGL\_DEPTH\_SIZE***, 0);

 **int** s = getConfigAttrib(egl, display, config, EGL10.***EGL\_GREEN\_SIZE***, 0);

 **int** r = getConfigAttrib(egl, display, config, EGL10.***EGL\_RED\_SIZE***,0);

 **int** g = getConfigAttrib(egl, display, config, EGL10.***EGL\_GREEN\_SIZE***, 0);

 **int** b = getConfigAttrib(egl, display, config, EGL10.***EGL\_BLUE\_SIZE***, 0);

 **int** a = getConfigAttrib(egl, display, config, EGL10.***EGL\_ALPHA\_SIZE***, 0);

 **if** (r == redSize && g == greenSize && b == blueSize && a == alphaSize && d >= depthSize && s >= stencilSize)

 **return** config;

 }

 **return** **null**;

}

As stated previously, the aim of this function is to go through each configuration and find all matches or greater, but select the one most suited to our wishes. This is achieved using the *“getConfigAttrib”* function, which acts as a string comparison between the attributes, locating the most alike configuration, returning if it finds a match.

 **private** **int** getConfigAttrib(EGL10 egl, EGLDisplay display, EGLConfig config,

 **int** attribute, **int** defaultValue)

 {

 **if** (egl.eglGetConfigAttrib(display, config, attribute, value))

 **return** value[0];

 **return** defaultValue;

 }

 }

The definition of the *“getConfigAttrib”* is simply a call to the EGL definition and then returning the response.

The last snippet of the ***TutorialView*** class is our definition of the Renderer class. This class consists of three methods: *“onDrawFrame”*, *“onSurfaceChanged”*, and *“onSurfaceCreated”*. These are fairly self-explanatory functions. *“onDrawFrame”* is called each time a new frame needs to be rendered and calls the *“step()”* function in our native library*. “onSurfaceChanged”* is called when the surface has been altered and calls the “init” function in our native library. *“onSurfaceCreated”*, as expected, is called when a surface is created, but we will leave this function for now.

 **private** **static** **class** Renderer **implements** GLSurfaceView.Renderer

 {

 **public** **void** onDrawFrame(GL10 gl)

 {

 NativeLibrary.*step*();

 }

 **public** **void** onSurfaceChanged(GL10 gl, **int** width, **int** height)

 {

 NativeLibrary.*init*(width,height);

 }

 **public** **void** onSurfaceCreated(GL10 gl, EGLConfig config)

 {

 }

}

## Altering “SimpleTriangle.java”

For this exercise, we will edit the code that was written for the class SimpleTriangle.java.

The imports stay the same, and as you can see in the snippet below, we add a new line to the first section of the class. This line creates a variable of type TutorialView (the class we just created). This is where we will display our graphics when we get around to creating them.

 **public** **class** SimpleTriangle **extends** AppCompatActivity

 {

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

 **protected** TutorialView graphicsView;

In the new version of the *“onCreate”* function, we create a new instance of TutorialView and assigns it to the variable we declared earlier on, *“graphicsView”*. We then set this newly created view to the content view, which means that Android will focus on it when displaying content.

 @Override **protected** **void** onCreate(Bundle savedInstanceState)

{

 **super**.onCreate(savedInstanceState);

 Log.*i*(*LOGTAG*, "Creating New Tutorial View");

 graphicsView = **new** TutorialView(getApplication());

 setContentView(graphicsView);

}

For these two functions (*onPause* and *onResume*), we call the super equivalents as we did in the “FirstNative” section, but this time, we also call the *“graphicsView”* functions so that the graphics pause and resume when the application does.

 @Override **protected** **void** onPause()

 {

 **super**.onPause();

 graphicsView.onPause();

 }

 @Override **protected** **void** onResume()

 {

 **super**.onResume();

 graphicsView.onResume();

 }

## Altering “NativeLibrary.java”

There are only very small changes to the NativeLibrary class from the “FirstNative” equivalent. The first change is the addition of two parameters *“width”* and *“height”* to the init function call. These values will be representative of the dimensions of the view. The other change is the addition of the step function call.

**public** **class** NativeLibrary

{

 **static**

 {

 System.*loadLibrary*("native-lib");

 }

 **public** **static** **native** **void** init(**int** width, **int** height);

 **public** **static** **native** **void** step();

}

## Altering “native-lib.cpp”

First, we must add "#include <unistd.h>” to the file. We then define the newly added function.

**extern "C"**{
 **JNIEXPORT void JNICALL** Java\_com\_arm\_malideveloper\_graphicssetup\_NativeLibrary\_init(JNIEnv \*env, jclass obj, jint width, jint height){};

 **JNIEXPORT void JNICALL** Java\_com\_arm\_malideveloper\_graphicssetup\_NativeLibrary\_step(JNIEnv \*env, jclass type){
 */\* Sleeping to avoid thrashing the Android log. \*/* sleep(5);
 **LOGI**("New Frame Ready to be Drawn!!!!");
 }
};

Take note that the names of the functions have changed since “FirstNative” as they are now in a new package.

Once this is completed, you can load the Application on your device. You still will not be able to view anything on the application, but our application is now ready to work with OpenGL ES Graphics coding in the next lab.