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

**LAB 1**

**First Android Native Application**

**Issue 1.0**

1. Contents

[1 Introduction 1](#_Toc36029285)

[1.1 Lab overview 1](#_Toc36029286)

[2 Requirements 1](#_Toc36029287)

[3 Creating the project 2](#_Toc36029288)

[3.1 Setup 2](#_Toc36029289)

[3.2 Creating required files 4](#_Toc36029290)

[4 Adding Code 4](#_Toc36029291)

[4.1 “FirstNative.java” 4](#_Toc36029292)

[4.2 “NativeLibrary.java” 5](#_Toc36029293)

[4.3 “native-lib.cpp” 6](#_Toc36029294)

[4.4 “CMakeLists.txt” file 6](#_Toc36029295)

[5 Running the application 7](#_Toc36029296)

# Introduction

The aim of these labs is to provide the student with an understanding of the basics of graphics and application development, with a focus on OpenGL ES and Mali-based technology. We will cover several topics core to graphics development and provide a solid foundation for the game design segment of the course.

## Lab overview

This lab will focus on the very basics of application development and provide a core structure for us to work with that will be built upon in the future labs.

An android application generally consists of two parts, a native side and an Android side. Typically, the native side is written in C or C++, whilst the Android side is written in Java. The Android portion of the code handles any application-specific aspects; this includes things such as setting up the surface and scene. The native portion of the code focuses more on rendering the visuals that we want to appear in our scene; this includes objects, textures, etc.

# Requirements

In this lab, we will be using the following hardware and software:

* **Android Studio**
* **An Android device**

You can download Android Studio free from here, under the Android Studio Downloads section: <https://developer.android.com/>

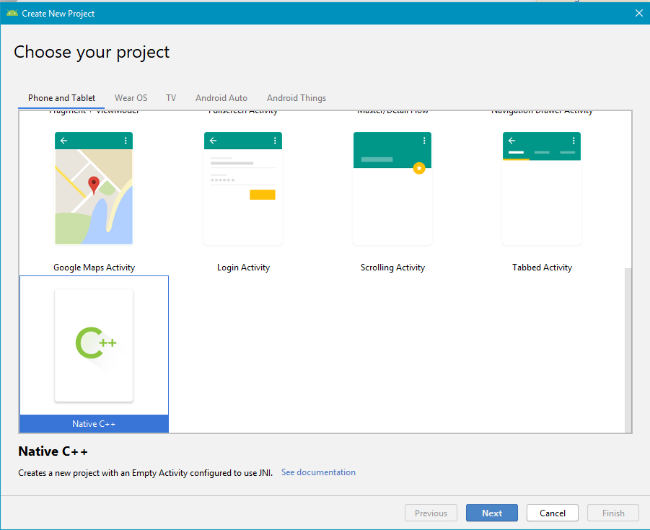
# Creating the project

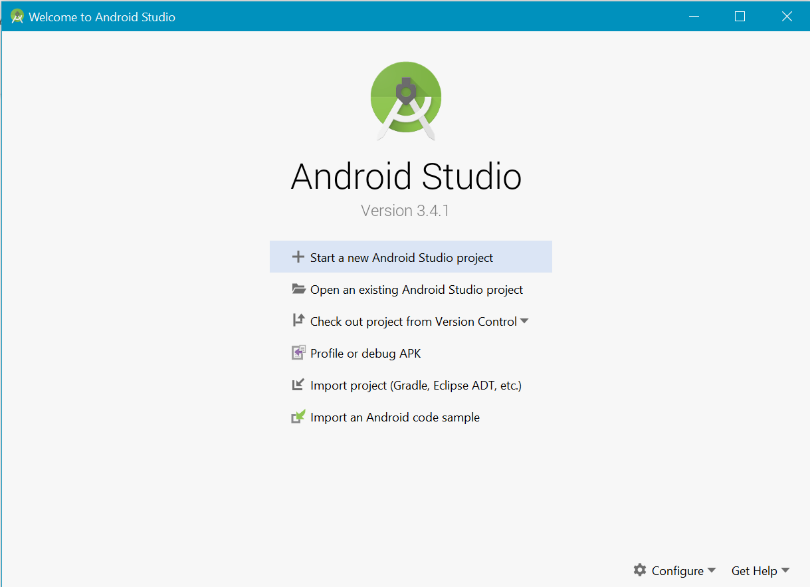
## Setup

Before we start developing our application, we need to setup a project in Android Studio.

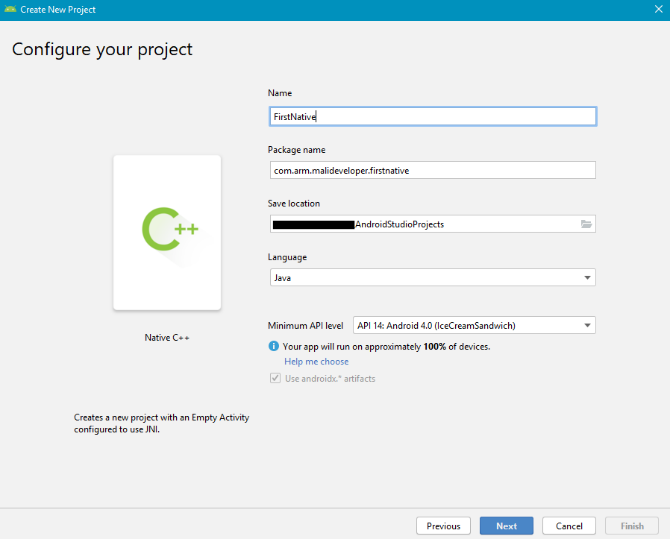
Start by opening Android Studio and then click on ‘*Start a new Android Studio project’*. This will bring up a new window in which we need to setup our new project.

In the Phone and Tablet category, select Native C++. This option will make an android project with an empty activity with C++ support. The application will be called “FirstNative”.

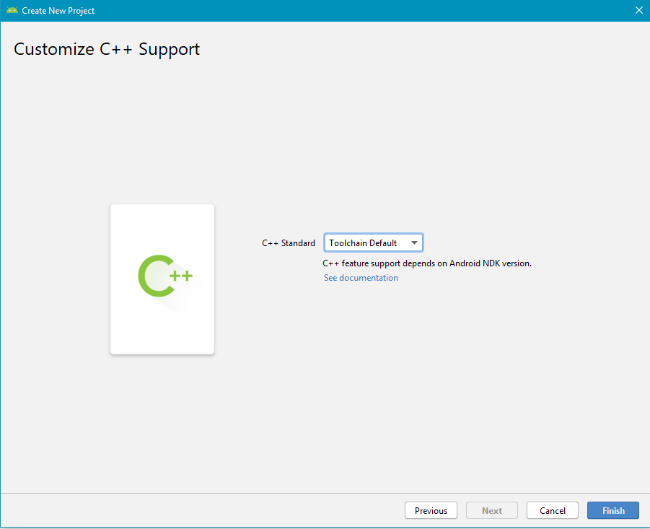




***Figure 1 – Creating Our FirstNative Project***

The package name is the reverse of your company name which should follow the convention: the application name and then the company/organization website. Taking Arm as an example, “FirstNative.malideveloper.arm.com” the package name will be “com.arm.malideveloper.firstnative”. Obviously, this is up to you, and you can make it what you want, as long as it includes full stops. Please select the Language as Java and a minimum API 14 to ensure that the application will be able to run on almost every phone.

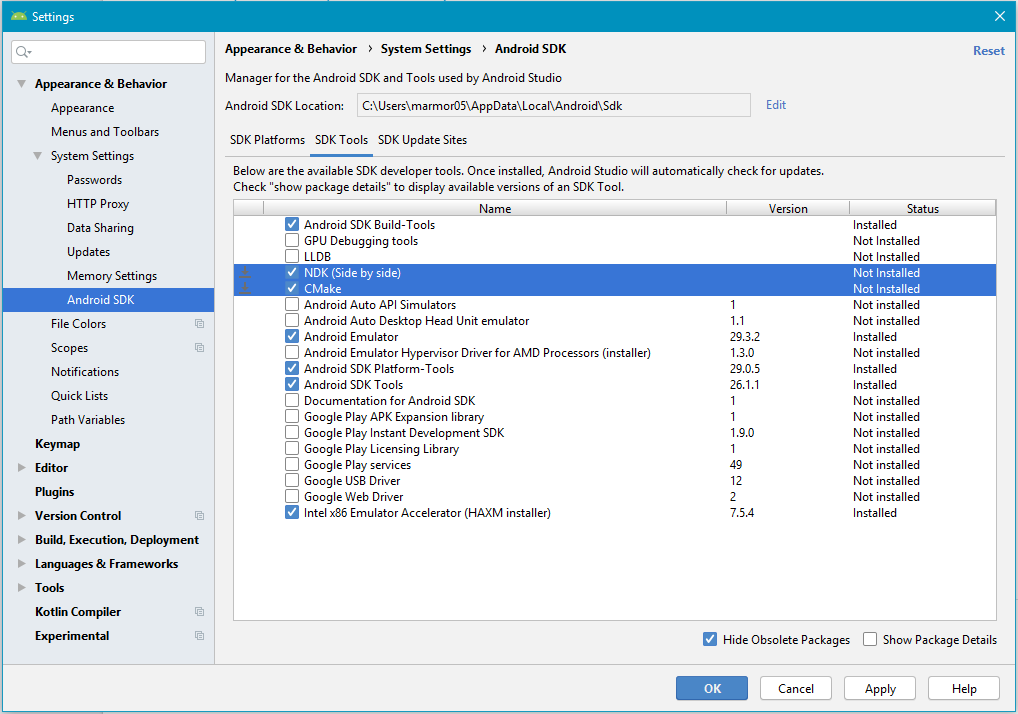
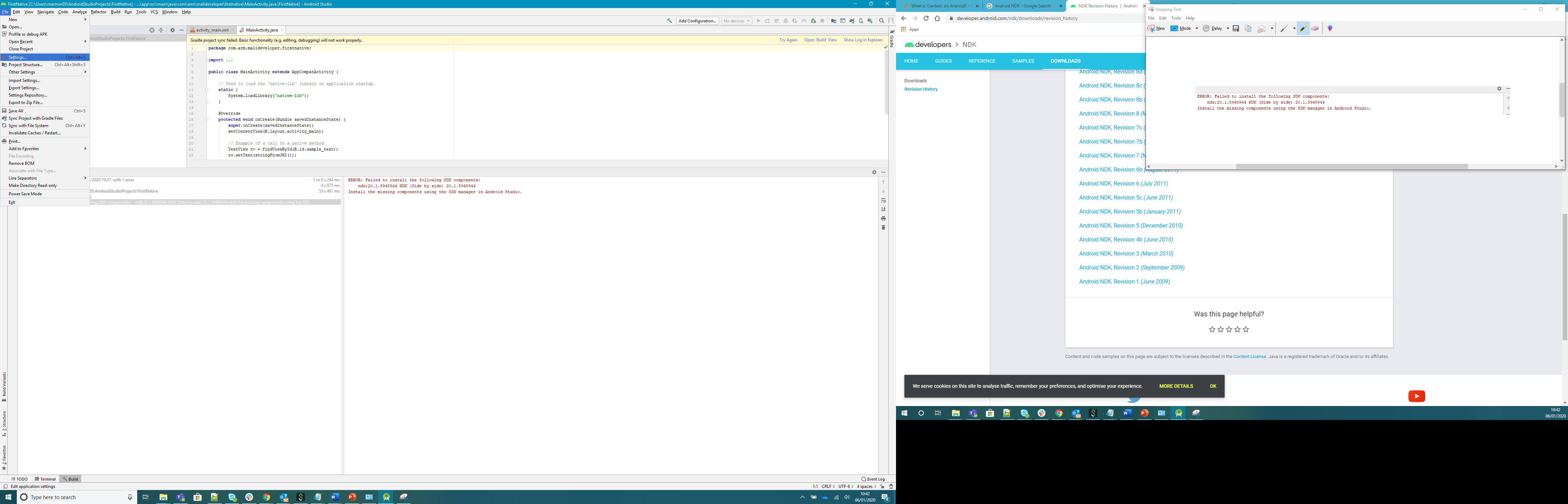
***Figure 2 – Configuring Our Project***

If more windows prompt you for choices, select the default options. For example, use the C++ standard – default toolchain. Then, click Finish.

***Figure 3 –Customize C++ Support***

You should now have a project containing an empty activity called “MainActivity” which you can find in “FirstNative” > “app” > “src” > “main” > “java” … > “MainAcitivity.java”.

**Gradle may throw an error** when trying to build the empty project. This is most likely due to needing the Android NDK. Go to “File” > “Settings” > “Android SDK” > “SDK Tools” tab. Select NDK, “CMake,” then accept the agreement and install them.

***Figure 4 –Accessing SDK Tools***

**A screenshot of a cell phone

Description automatically generated**Once sorted, Gradle will automatically build your project, and you may see errors appear, such as not having your NDK set; to fix this go to File -> Project Structure -> SDK Location and then set your NDK location there.

The project should now be created, and we should be able to see it in the project explorer window to the left in Android Studio. It should look like the image on the right; if it does not, select the option Project from the dropdown list just above the explorer window.

***Figure 5 – Our FirstNative Project Display***

## Creating required files

A screenshot of a cell phone

Description automatically generatedFor this initial lab, we will need to work with a total of three files. Two of these files will be Android related and one will be native. The Activity class we just created, called “MainActivity”, is one of the Android files and can be seen in app > src > main > java > com.arm.malideveloper.firstnative > MainActivity.

To make the next file, we right-click on the “com.arm.malideveloper.firstnative” folder, select New,and then select Java Class. A new window will appear asking for the name of the new class; we call it “NativeLibrary”. This will be placed alongside your FirstNative file, as seen in the picture.

***Figure 6 – Location of Java Files***

The third file we need handles the native side of the application, and as such, it is located somewhere else. Android Studio creates this since we have added C++ support, and it is located here: app > src > main > cpp > native-lib.cpp. In here, there is some placeholder C++ code, which creates a HelloWorld string. We will adapt this later for our own needs, but first, we need to adapt our “MainActivity” file.

In order to make use of intuitive names, please first rename the “MainActivity” to “FirstNative”. This can be done by right-clicking on the “MainActivity” in app > java > com.arm.malideveloper.firstnative > MainActivity and then select Refactor and Rename and type in the new name. Now click on the Refactor button. This will open a prompt called “Refactoring Preview” which displays all places where refactoring is needed in order for this name change to take place. Android Studio will make these changes automatically once you hit the Do Refactor button.

# Adding Code

## “FirstNative.java”

We are now going to write the code for our FirstNative.java file. Remember that this file is a part of the Android side and will perform most of the application-specific setup and interactions with Android. The first thing we should do is clear the existing code from the FirstNative.java class. We want to be working from an empty class.

To start our code, we need to let the application know which package FirstNative.java belongs to. This first line (as seen above) indicates that this class belongs to the package we created for our project. We must include this line in all future java source files.

**package** com.arm.malideveloper.firstnative;

The next few lines import various things that will help us develop the application. Due to this class being the initial point for the application/activity, we need to import the activity class.

**import** android.os.Bundle;

**import** android.app.Activity;

**import** android.util.Log;

We next need to define the class. We declare “FirstNative” as an extension of the Activity class because this lets Android know that it is the initial class. The next line declares a string that we will use when we handle the logging of information.

Then, we create our first function, which will also be the first function called when the application starts. This function is called “onCreate” and as its first act calls the “onCreate” method of the superclass; the reason this is done is to save time and let Android carry out some of the initial setup. The second command of the function outputs the given string in LogCat, which is Android’s logging tool. LogCat should be visible at the bottom as a part of the Android Monitor tab. The final line attempts to communicate with the native part of the application via the “NativeLibrary.java” class function “init()”; however, we are yet to write this class.

**public** **class** FirstNative **extends** Activity

{

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

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

{

**super**.onCreate(savedInstanceState);

Log.*d*(*LOGTAG*, "On Create Method Calling Native Library");

NativeLibrary.*init*();

}

The final two functions we need to add to our class are *“onPause()”* and *“onResume()”.* However, we will again leave most of this up to Android to handle and simply call the respective super methods for each. This will tell the application what to do upon occurrence of a pause or resume.

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

{

**super**.onPause();

}

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

{

**super**.onResume();

}

## “NativeLibrary.java”

The next file we need to populate is “NativeLibrary.java”. This is another Android/Java file but will act as our interface with the native side of the application; therefore, it is crucial that this is working and able to mediate between the Android and native parts. Again, we first want to clear the existing class and produce our own.

**package** com.arm.malideveloper.firstnative;

**public** **class** NativeLibrary

{

**static**

{

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

}

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

}

The first line of this class is the declaration of which package it belongs to. The next line that is noteworthy is *“System.loadLibrary(“Native”);”*. This will load our native library from the system. Finally, we declare a function shell called *“init()”* as we saw earlier. The function itself will be in the native library.

## “native-lib.cpp”

Now that we have both of our Java/Android files; it is time to write the code for our native side. As mentioned earlier, the native side will handle rendering the scene, etc., but for now, we wish to keep it simple. Once more we want to clear the existing content from the Native-lib class.

**#include** <jni.h>

**#include** <android/log.h>

As opposed to Java where we import files/libraries we need, we are now in C where we need to ‘#include’ them. We start by including “jni.h” which is a header file for the Java Native Interface. This will allow us to interact with the Java code. The second line includes a similar file to the log library added in the Android side, which allows logging for debugging, etc.

**#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\_\_)

**extern** "C" **JNIEXPORT** void **JNICALL**

Java\_com\_arm\_malideveloper\_firstnative\_NativeLibrary\_init(JNIEnv \*env, jclass type) {

LOGI("Hello from the Native Side!!");

}

For logging purposes, we have added some #defines. We then write the code for the function referenced in “NativeLibrary.java”. We initially label it as external C so that Java realizes it must handle it differently. We then put a simple command, print hello world, in the function definition so that we can test it works.

## “CMakeLists.txt” file

Android Studio uses CMake to compile C and C++ code into a native library. This currently does not require any editing for this lab, but we will need to look at this in more detail in later labs, so it is good to look at this code now.

**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).  
 **src/main/cpp/native-lib.cpp** )

This sets our native-lib location, name and sets the library as a shared library so that it can be used across the entire application. You may set the library to be either Shared or Static depending on your need. You may also define multiple libraries here, which we will look at in more detail in later labs.

**find\_library**( # Sets the name of the path variable.  
 **log-lib**  
  
 # Specifies the name of the NDK library that  
 # you want CMake to locate.  
 **log** )

This simply specifies the name of the public NDK library that you want to add. CMake already includes all system libraries in the search path as default, so no other prebuilt libraries need to be added here.

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

Finally, CMake specifies the libraries that should link to the target libraries. In this instance, and most other instances, this will link multiple libraries to the log library included in the NDK.

# Running the application

* Android Smartphone:

First, please ensure that your phone is in developer mode and that Debugging is enabled. Connect the phone to your PC via cable and then simply click the **RUN** button in Android Studio (this is shown as a green play button on the top menu). This may bring up a list of currently connected devices. Just pick the one you want to use and press **OK.** The application is automatically installed on the device and then started.

Please check the Log to see if the message is displayed, and therefore check if the app runs correctly.