Android Development: Part One

This workshop will introduce you to the nature of the Android development platform. We begin with an overview of the platform's development history and some discussion of its associated hardware architecture. We go on to address the unique peculiarities of the Android Platform's software architecture, the nature of the Dalvik 'process virtual machine' on which Android applications run, and two variants of the Android Software Development Kit. The Android Native Development Kit is introduced and contrasted with the SDK.

We go on to set up an Android Development Project using the standard SDK and present some sample Android programs designed to get you thinking about how to get started with the Android Platform. The workshop concludes with some discussion of what will be covered in Part Two of this mini-series of workshops, and some references and recommended reading for those who wish to explore this area in greater details in their own time.

Android: "It's Linux, Jim, but not as we know it…"

Android, a Linux-based operating system for low-power touchscreen devices, began development in 2003 and was nominally released in 2007, with the first handsets coming out late in 2008. Essentially a smartphone and tablet OS, Android supports ARM, MIPS and x86 hardware platforms – of those, ARM is generally viewed as the most commonly targeted. To give some context to how the OS and its associated hardware have evolved in the four years since its release, the images below show the first Android smartphone and one of the most recent Android offerings from the same company, HTC.

HTC Dream (2008)
528 MHz CPU
256 MB Storage
192 MB RAM
3.2" HVGA Screen

HTC DLXJ (2012)
1.5 GHz Quad-Core CPU
16 GB Storage
2 GB RAM
5.0" 1080p HD Screen

The explosion of hardware development in the mobile platform market has dwarfed the incremental performance increases of desktop hardware across the same period. Commensurate with this, the capabilities of operating systems targeted at such hardware have been extended and expanded, as has their profile. As an example, Microsoft's latest iteration of the Windows operating system was designed with touch screen interfaces as a key aspect of future technology, and kernels were released for both the x86 and ARM architectures.

All of this combines to make the understanding of mobile platform development a useful quality for any aspiring game developer. As such, you should consider this tutorial an encouragement to explore all mobile platforms on your own time, to develop for yourself a small portfolio of mobile software. Not only is it directly relevant to the releases of many smaller studios, but it demonstrates that you have experience working within tight computational constraints which, in turn, suggests you know how to write efficient and elegant game code.

So, without further ado, let's dive into what development on Android actually means.

The Android SDK, or How I Learned to Stop Caring and Love the Dalvik Machine

The first thing that we need to understand about Android is that it is an operating system in two parts. Actually, that's an oversimplification – depending on how you slice it, it can be an operating system of a half dozen parts – but for our purposes we only consider two aspects. The background, pared-down Linux kernel (with no X Window System support), and the Dalvik (Java) Virtual Machine.

Almost all development for Android is undertaken via the Dalvik VM. The VM executes Dalvik 'dex-code' (Dalvik Executable), normally translated from Java bytecode. We say normally because, within the Android SDK, there is the option to inject libraries into one's Java-based Android program, through clever use of the Java Native Interface, to facilitate support of other programming languages such as C. This process has been used to port old-school PC titles to the Android platform, but it is never as straightforward as simply 'import library, ctrl-c, ctrl-v, compile'. Well, *never* say never – a simple, ASCII tic-tac-toe simulator would probably find that development cycle reasonable.

For the most part, however, Android applications are run within the Dalvik machine, making calls to the native level (for purposes such as OpenGL ES rendering) via a set of pre-existing libraries (some of which will be presented later in this tutorial). This can lead to some interesting memory management issues, given that the Dalvik machine has its own garbage collection and memory allocation schema – but these, also, will be discussed later in the tutorial.

From the perspective of getting started with Android development, I recommend one (or, preferably, both) of two development environments. For people comfortable with Visual Studio, NVidia has a plug-in for Tegra development (Tegra being NVidia's mobile chipset technology) that supports Android development in the MS VS environment. This is useful, in that it allows you to develop in an environment you're already comfortable with. The down side of this plug-in is that it does not allow for emulation of Android hardware on your PC. So, while you can test your code for compilation within the Visual Studio environment, if you want to actually see what your code *does* you have to plug in an Android device.

If you want to get this plug-in, you are required to register for NVidia's Registered Developer Program. This process normally takes a few hours (up to a day) and, depending on which 'areas' of NVidia development you express interest in during registration, can aid you in more than just Android development (for example, it is also required to obtain the NVidia CUDA development plug-ins for Visual Studio). The actual installer is relatively small, but that's because it downloads the bulk of the material you need after you start installing. Once it's installed, you will find an option in Create New Project that allows you to select Android. Within that, you'll find basic templates and some simple sample programs to get yourself started. You'll also find a base project for Android with Native Library – more on this later.

The other, more common, platform for Android development comes from http://developer.android.com/SDK. On this site you will find a complete 'bundle' that includes an Android-centric Eclipse install, and all of the libraries you should need for getting started writing for Android. Unlike the NVidia SDK, this download doesn't require a 'vetted' registration procedure, and you can be writing your first Android app within ten minutes of clicking the download link.

The Eclipse-based Android SDK does support development of Android Apps using C/C++, in the same fashion that the Tegra toolkit does. For more information on how to set this up, given it can be a little fiddly, an excellent tutorial can be found at

http://mhandroid.wordpress.com/2011/01/23/using-eclipse-for-android-cc-development/

If you have any problems installing either (or both!) of these environments, the first lab session of the day (11am-12noon) has been allocated to helping you get up and running.

The Native Development Kit and You

There are two commonly employed ways to use C/C++ code in an Android application. The first, mentioned above, is to include libraries that access the JNI and give you simple C/C++ support. The second is to use what is called the Android Native Development Kit (or NDK). You can obtain the NDK from android.com, but it should be approached with caution.

It is easy to envision the NDK as a sort of panacea – the idea being that you can simply copy and paste a program you have developed in C++, wholesale, compile and run. This is rarely, if ever, the case. Partially, this is because Android uses a pared-down, non-standard C library (known as Bionic, for those who're interested) – this can require you to 'walk three sides of a square' when restructuring your existing codebase to suit it. On the other hand, if you have developed your entire codebase with the Android NDK in mind, making the change in the other direction (from Android to PC) is a simpler prospect.

We will discuss the NDK in Part Two of this tutorial series, along with other subjects directly relevant to the Team Project (such as particle effects in OpenGL ES). For now, though, it is recommended that you consider carefully which path you wish to follow in the context of how you make your game 'portable' between the PC and Android platforms.

Possibly the simplest way is to write two versions of each function and class you create – one in standard C++, and another in Java. While this does front-load additional work for your team, it reduces the chance that you will forge ahead with C++ development in the hope of 'wedging it into the NDK' towards the end of your development cycle, only to find that this is impractical/impossible.

"Hello, World!" and Other Animals

Below is a link to a very useful tutorial that leads you, step by step, through the creation of a "Hello, World!" example using the Eclipse-based Android SDK.

http://developer.android.com/training/basics/firstapp/index.html

Your first task this morning is to work through that example methodically. If you've already done that on your own time, then either move on to the next task (discussed below) or, if you have the NVidia Tegra SDK, compare its "Hello, World!" example to this one. Look at the differences, investigate what the NVidia version does within its "HelloJni.c" file.

The next sample program for you to work through is a simple OpenGL ES program designed to make you think about the differences (and similarities) between using OpenGL within the Android environment, and using it in your existing PC-based software projects. The book this sample code is drawn from has been *very* useful to me in exploring the capabilities of OpenGL ES, and is included in the references for those who are interested in reading it for themselves.

References

OpenGL ES for Android: A Quick-Start Guide, by Kevin Brothaler

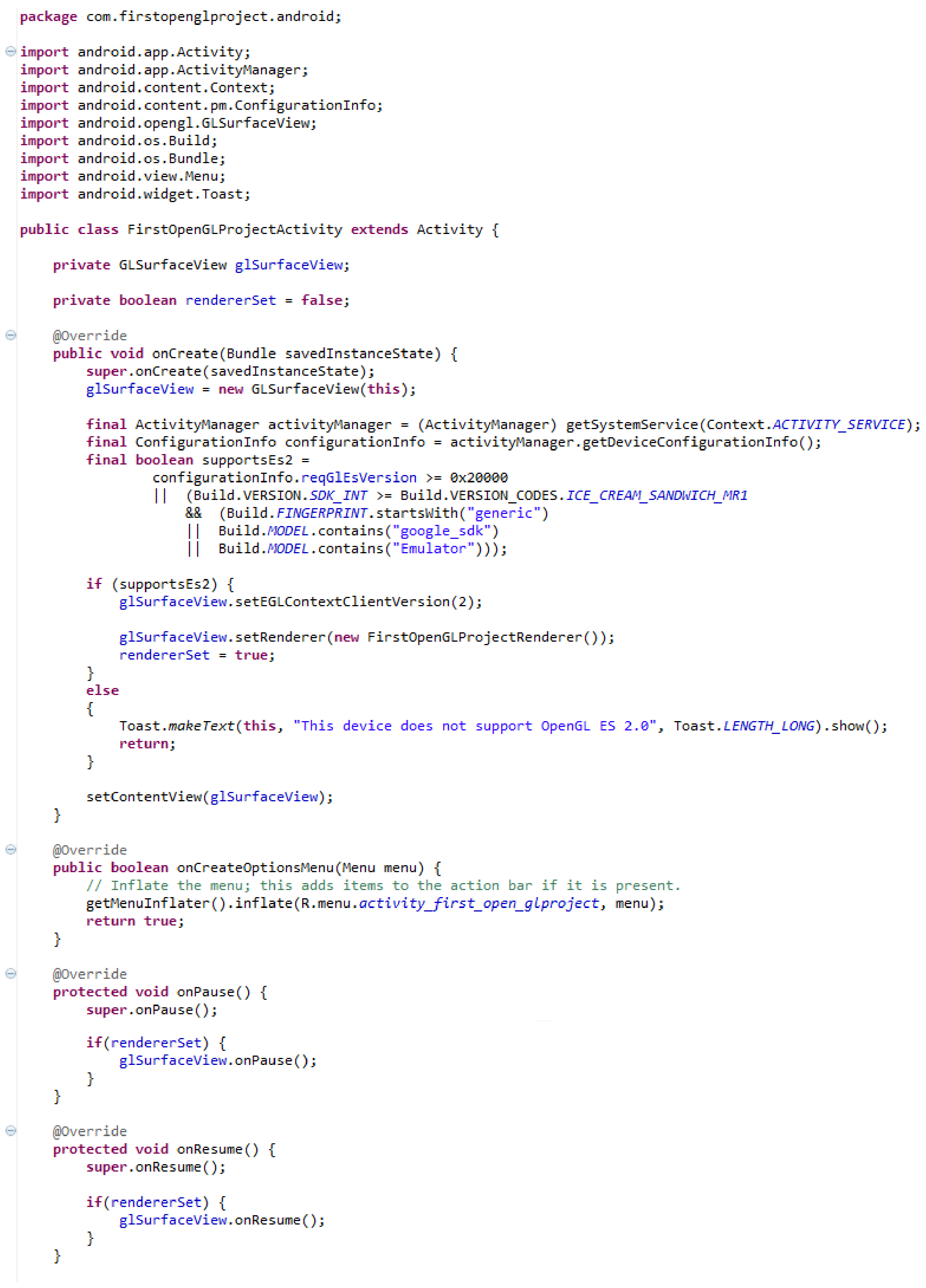
This is a 'Beta-book', in that it hasn't yet been officially published. It covers much of the material you have already explored as part of your Graphics module last semester, but does so from the perspective of developing for Android within the SDK. The simple OpenGL ES sample program below is drawn from it.

Pro OpenGL ES for Android (Professional Apress), by Mike Smithwick & Mayank Verma

A very interesting book that introduces OpenGL ES to the reader, discusses the matrix mathematics behind rendering, and includes some interesting examples from which to draw inspiration.

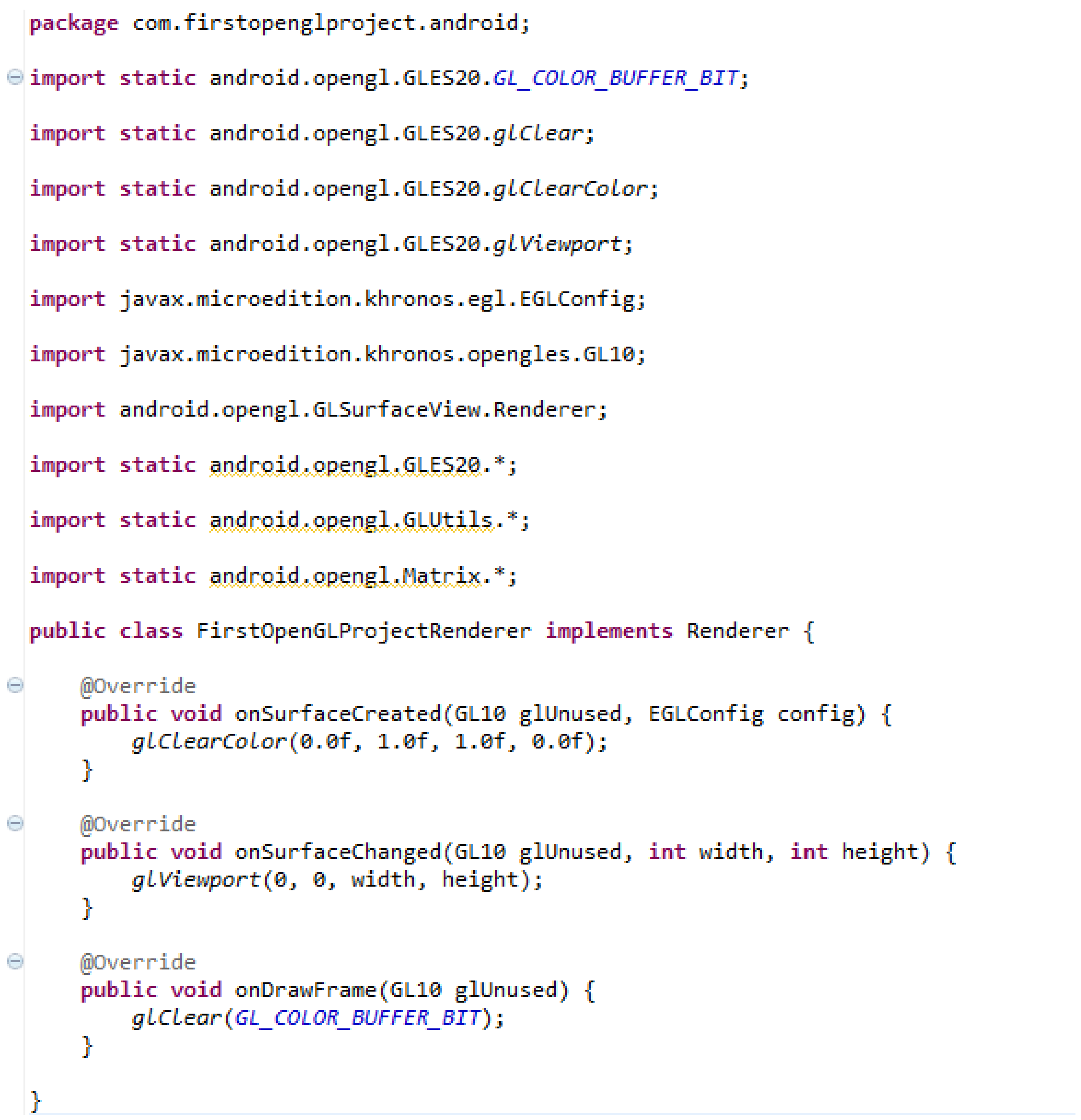
The Android Website

Silly as it sounds, some of the tutorials on this site are very, very useful in terms of getting to grips with Android development as a whole. Working through the tutorials is a good way to get your feet wet with Android development without investing a lot of your free time/money.



```java
package com.firstopenglproject.android;

import android.app.Activity;
import android.app.ActivityManager;
import android.content.Context;
import android.content.pm.ConfigurationInfo;
import android.opengl.GLSurfaceView;
import android.os.Build;
import android.os.Bundle;
import android.view.Menu;
import android.widget.Toast;

public class FirstOpenGLProjectActivity extends Activity {

    private GLSurfaceView glSurfaceView;

    private boolean rendererSet = false;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        glSurfaceView = new GLSurfaceView(this);

        final ActivityManager activityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        final ConfigurationInfo configurationInfo = activityManager.getDeviceConfigurationInfo();
        final boolean supportsEs2 =
                configurationInfo.reqGlEsVersion >= 0x20000
                ||  (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1
                    &&  (Build.FINGERPRINT.startsWith("generic")
                    ||  Build.MODEL.contains("google_sdk")
                    ||  Build.MODEL.contains("Emulator")));

        if (supportsEs2) {
            glSurfaceView.setEGLContextClientVersion(2);

            glSurfaceView.setRenderer(new FirstOpenGLProjectRenderer());
            rendererSet = true;
        }
        else
        {
            Toast.makeText(this, "This device does not support OpenGL ES 2.0", Toast.LENGTH_LONG).show();
            return;
        }

        setContentView(glSurfaceView);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_first_open_glproject, menu);
        return true;
    }

    @Override
    protected void onPause() {
        super.onPause();

        if(rendererSet) {
            glSurfaceView.onPause();
        }
    }

    @Override
    protected void onResume() {
        super.onResume();

        if(rendererSet) {
            glSurfaceView.onResume();
        }
    }
```

FirstOpenGLProjectActivity.java



```java
package com.firstopenglproject.android;

import static android.opengl.GLES20.GL_COLOR_BUFFER_BIT;

import static android.opengl.GLES20.glClear;

import static android.opengl.GLES20.glClearColor;

import static android.opengl.GLES20.glViewport;

import javax.microedition.khronos.egl.EGLConfig;

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

import static android.opengl.GLES20.*;

import static android.opengl.GLUtils.*;

import static android.opengl.Matrix.*;

public class FirstOpenGLProjectRenderer implements Renderer {

    @Override
    public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
        glClearColor(0.0f, 1.0f, 1.0f, 0.0f);
    }

    @Override
    public void onSurfaceChanged(GL10 glUnused, int width, int height) {
        glViewport(0, 0, width, height);
    }

    @Override
    public void onDrawFrame(GL10 glUnused) {
        glClear(GL_COLOR_BUFFER_BIT);
    }

}
```

FirstOpenGLProjectRenderer.java