Using the Android NDK for Graphics

This document .elaborates on the NDK example from the book and provided up-to-the-minute details of the Eclipse build environment.

This example covers the use of OpenGLES version 1 (without shaders.)

3D Rendering with the NDK

All Android native code is compiled into a shared object library (a .SO library) using the Android NDK tools and libraries. The functions that are required to be externally visible have special wrapper methods written for them using the notation of the Java Native Interface (the JNI) that makes them executable within any Java program where they are declared as `private static native void MethodName(...)` `...` class methods.

The starter example that we are going to examine in this section should give provide a good framework for porting desktop programs to Android devices. The structure of the example follows the metaphor of the GLUT library described in the book. In this case, four functions perform the tasks of, initialization, viewport sizing, scene drawing and, resource release. The commensurate native C language function are called, `nativeIni(), nativeResize(), nativeDraw()` and `nativeRelease()`.

The example is going to be based on a slightly modified version of the SanAngeles program that ship with the NDK (the SanAngles example is a well know benchmark program for OpenGLES, it has been discussed in the book and multiple platform versions may be found on our web-site. Before looking at the details it is worth saying a little about the function naming, and argument handling convention that must be adopted when linking Android JAVA VM code with code compiled into a native shared object library, this is the JNI, see the boom for more details.

Bulding a native library

As a UNIX kernel lies at the heart of Android, the NDK works best when it used on systems with a UNIX kernel, this makes development on WIndows somewhat problematic as it is necessary to install one of the UNIX environments. e.g mingw in order to run the cross-platform compiler that produces native code.

The NDK offers a neat makefile build system that sets appropriate paths to locate the Android headers and compiler flags to target the output to a shared object (.SO) library that can be linked to the main Android project . Where possible, the shared object library should be built by setting up a makefile that follows the NDK's template. The makefiles are quite short, listing 0.1, shows an example that builds a library called "nkd1.so" from the single source file "native.c".

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_LDLIBS := -llog
LOCAL_MODULE := ndk1           # name of SO library
LOCAL_SRC_FILES := native.c    # list of source files
include $(BUILD_SHARED_LIBRARY)
```

Listing 0.1: The SO makefile only needs to list the input files and name the output library.

JNI nomenclature and NDK naming conventions

The Java Native Interface (JNI) is not an Android specific specification, it applies to any situation where Java code running in a VM wishes to access native code written in a native language, typically C. The NDK uses the JNI as its definition of how to define function names and arguments in native code so that they can be used with the method names and calling conventions in the Java Because Java uses very different storage conventions for its variables, for example, Java strings are not simply *byte character arrrays*, the matching of parameters passed back and forth will need to undergo a translation and use a highly structured naming convention. This is well documented in the JNI specification [1].

One of the trickiest things to do with the NDK is to pass data and make function calls back and forth between the native functions and the Android API. The rules of the JNI describe how to interpret data passed to the native functions, and how to package data to be returned to the JAVA code. (See the book for an elaboration on this process.)

We refer you to the specification [1] for full details, however the example in the book includes, passing integers, floats, and text strings, so with a bit of intelligent guessing you should be able to work out how to use two, thee, or more arguments in the functions. (See the book section on using the JNI.)

The NDK OpenGL example

Considering again the SanAngles example we shall use this section to show how the Android specific sections can be used to design the program so that it conforms to the familiar trilogy of OpenGL tasks: 1) setup, 2) render, and 3) resize, that should have a familiar ring from the book and that mapped

so neatly onto the GLUT. When we have extracted the Android essence and reorganized along the lines of the GLUT you should be able to slot in the visualization aspects of any OpenGL program following the GLUT metaphor.

Android NDK based OpenGL applications that are fundamentally going to take over the full device screen require only the simplest of JAVA activity code to be written. The book presented a technique for calling JAVA methods from the native code, so if there is something (e.g. accelerometer data access) that it is easier to do through the normal Android SDK it can be done that way; Although, it is possible to access the accelerometers through NDK function calls too.

The Activity code The JAVA code for the main activity and the `GLSurfaceView` class is taken from that discussed in the book, only the names have been changed and four static methods added to wrap the native functions for: initialization, resizing, rendering and close-down. The code for the revised activity and its class is presented in listings 0.2 and 0.3.

```
package org.openfx.example.GLESdemo;
// GLES imports
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
// Android imports
import android.app.Activity;
import android.opengl.GLSurfaceView;

public class DemoActivity extends Activity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 // create the OpenGL output surface
 mGLView = new DemoGLSurfaceView(this);
 setContentView(mGLView);
 }

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

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

 // this object represents the GLES enabled surface
 private GLSurfaceView mGLView;

 static {
  System.loadLibrary("glesdemo");
 }
}
```

Listing 0.2: The main application activity code.

```
class DemoGLSurfaceView extends GLSurfaceView {

 DemoRenderer mRenderer; // protocol methods

 public DemoGLSurfaceView(Context context) {
  super(context);
  mRenderer = new DemoRenderer();
  setRenderer(mRenderer);
 }
}

class DemoRenderer implements GLSurfaceView.Renderer {
 public void onSurfaceCreated(GL10 gl, EGLConfig config) {
  // call our GL initialization code
  nativeInit();
 }
 public void onSurfaceChanged(GL10 gl, int w, int h) {
  // respond to change of display surface size
  native Resize(w, h);
 }
 public void onDrawFrame(GL10 gl) {
  // render now!
  nativeRender();
 }
 // these are the method that
 private static native void nativeInit();
 private static native void nativeResize(int w, int h);
 private static native void nativeRender();
 private static native void nativeDone();
}
```

Listing 0.3: The OpenGL class and surface configuration methods

The OpenGL native code The native code builds into the shared object library from as many source files as may be required. The NDK makefile is easily adapted for a this project, see listing 0.4, the only points of note are the collection of source files and the linkage with the GLES library.

There is actually very little additional code that needs to be considered as we are not going into the details of the OpenGL code used by the SanAngeles or any other OpenGL example. The power of this example is that that it can be adapted to cover just about anything that can be done on the desktop with the GLUT. At the example's heart are just three functions:

```
appInit();
appDeInit();
appRender();
```

A framework from which to extend these functions is given in listing 0.5.

The final two listings for this program connect the device independent OpenGL code with the main Activity's methods. We have shown in the bok how to call native functions as if they were JAVA static class methods, and this example is no different. In listing 0.6 the global variables used by the application timer are defined, and the native methods for initialization and close-down are provided. Listing 0.7 completes the picture by building the links to the code that resizes the viewport and renders the scene at the current time.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := glesdemo

LOCAL_CFLAGS := -DANDROID_NDK \
 -DDISABLE_IMPORTGL

LOCAL_SRC_FILES := \
 source1.c \
 source2.c \
 android-interface.c

# MAKE SURE THE GLES LIBRARY IS USED !!!
LOCAL_LDLIBS := -lGLESv1_CM -ldl -llog

include $(BUILD_SHARED_LIBRARY)
```

Listing 0.4: The makefile for the native shared object library .

Breaking down the example into these short stages identifies the places in the code that can be the jumping off point for enhancement.

```
// Functions called from the app framework.

// globlal variables and funciton prototypes
// would be declared here
extern int gAppAlive;

void appInit(){
 // app specific code HERE
}

void appDeinit() {
 // app specific code HERE
}

// render whatever we like !!!!
void appRender(long tick, int width, int height) {
 if (!gAppAlive) return;

 // Prepare OpenGL ES for rendering of the frame.
 glViewport(0, 0, width, height);
 glClearColorx((GLfixed)(0.1f * 65536),
 (GLfixed)(0.2f * 65536),
 (GLfixed)(0.3f * 65536), 0x10000);

 // now draw - this might typically be:
 glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 gluPerspective(45, (float)width / height, 0.5f, 150);
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();

 // Draw HERE

}
```

Listing 0.5: The device independent application functions for initialization, rendering and closure.

```
#include <jni.h>
#include <sys/time.h>
#include <time.h>
#include <android/log.h>
#include <stdint.h>

int gAppAlive = 0; // the GO code

// global variables for screen size and timers
static int sWindowWidth = 320;
static int sWindowHeight = 480;
static int sDemoStopped = 0;
static long sTimeOffset = 0;
static int sTimeOffsetInit = 0;
static long sTimeStopped = 0;

/* Call to initialize the graphics state */
void
Java_org_openfx.example_GLESdemo_DemoRenderer_nativeInit
 (JNIEnv* env ) {
 // this is a call defined in jni.h to initialize the
 // GLES functions for Android NDL
 importGLInit();
 // custion initialize
 appInit();
 // reset the timer and set the "GO" code
 gAppAlive = 1;
 sDemoStopped = 0;
 sTimeOffsetInit = 0;
}

/* Call to finalize the graphics state */
void
Java_org_openfx_example_GLESdemo_DemoRenderer_nativeDone
 ( JNIEnv* env ){
 appDeinit(); // custom funciton
 importGLDeinit(); // NDK function
}
```

Listing 0.6: The NDK function interfaces for initialization and close-down.

```
/* call to resize the view */
void Java_org_openfx_example_GLESdemo_DemoRenderer_nativeResize
 ( JNIEnv* env, jobject thiz, jint w, jint h ){
 // set the global variables
 sWindowWidth = w;
 sWindowHeight = h;
}

/* Call to render the next GL frame */
void Java_org_openfx_example_GLESdemo_DemoRenderer_nativeRender
(JNIEnv* env){
 long curTime;
 /* if sDemoStopped is TRUE, then we re-render the same frame
 * on each iteration.
 */
 if (sDemoStopped) {
  curTime = sTimeStopped + sTimeOffset;
 } else {
  curTime = _getTime() + sTimeOffset;
  if (sTimeOffsetInit == 0) {
  sTimeOffsetInit = 1;
  sTimeOffset = -curTime;
  curTime = 0;
  }
 }
 // call the OpenGL renderer
 appRender(curTime, sWindowWidth, sWindowHeight);
}
```

Listing 0.7: The NDK function interfaces for rendering and display size change.

Bibliography

[1] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Palo Ato CA: Sun Microsystems Inc, 1999.