The native-activity sample resides under the NDK installation root, in samples/native-activity. It is a very simple example of a purely native application, with no Java source code. In the absence of any Java source, the Java compiler still creates an executable stub for the virtual machine to run. The stub serves as a wrapper for the actual, native program, which is located in the .so file.

The app itself simply renders a color onto the entire screen, and then changes the color partly in response to movement that it detects.

AndroidManifest.xml

An app with only native code must not specify an Android API level lower than 9, which introduced the[NativeActivity](http://developer.android.com/ndk/guides/concepts.html#naa) framework class.

<uses-sdk android:minSdkVersion="9" />

The following line declares android:hasCode as false, as this app has only native code–no Java.

<application android:label="@string/app\_name"

android:hasCode="false">

The next line declares the NativeActivity class.

<activity android:name="android.app.NativeActivity"

Finally, the manifest specifies android:value as the name of the shared library to be built, minus the initiallib and the .so extension. This value must be the same as the name of LOCAL\_MODULE in Android.mk.

<meta-data android:name="android.app.lib\_name"

android:value="native-activity" />

Android.mk

This file begins by providing the name of the shared library to generate.

LOCAL\_MODULE := native-activity

Next, it declares the name of the native source-code file.

LOCAL\_SRC\_FILES := main.c

Next, it lists the external libraries for the build system to use in building the binary. The -l (link-against) option precedes each library name.

* log is a logging library.
* android encompasses the standard Android support APIs for NDK. For more information about the APIs that Android and the NDK support, see [Android NDK Native APIs](http://developer.android.com/ndk/samples/stable_apis.html).
* EGL corresponds to the platform-specific portion of the graphics API.
* GLESv1\_CM corresponds to OpenGL ES, the version of OpenGL for Android. This library depends on EGL.

For each library:

* The actual file name starts with lib, and ends with the .so extension. For example, the actual file name for the log library is liblog.so.
* The library resides in the following directory, NDK root: <ndk>/platforms/android-<sdk\_version>/arch-<abi>/usr/lib/.

LOCAL\_LDLIBS := -llog -landroid -lEGL -lGLESv1\_CM

The next line provides the name of the static library, android\_native\_app\_glue, which the application uses to manage NativeActivity lifecycle events and touch input.

LOCAL\_STATIC\_LIBRARIES := android\_native\_app\_glue

The final line tells the build system to build this static library. The ndk-build script places the built library (libandroid\_native\_app\_glue.a) into the obj directory generated during the build process. For more information about the android\_native\_app\_glue library, see its android\_native\_app\_glue.h header and corresponding .csource file.

$(call import-module,android/native\_app\_glue)

For more information about the Android.mk file, see [Android.mk](http://developer.android.com/ndk/guides/android_mk.html).

Application.mk

This line defines the minimum level of Android API Level support.

APP\_PLATFORM := android-10

Because there is no ABI definition, the build system defaults to building only for armeabi.

main.c

This file essentially contains the entire progam.

The following includes correspond to the libraries, both shared and static, enumerated in Android.mk.

#include <EGL/egl.h>

#include <GLES/gl.h>

#include <android/sensor.h>

#include <android/log.h>

#include <android\_native\_app\_glue>

The android\_native\_app\_glue library calls the following function, passing it a predefined state structure. It also serves as a wrapper that simplifies handling of NativeActivity callbacks.

void android\_main(struct android\_app\* state) {

Next, the program handles events queued by the glue library. The event handler follows the state structure.

struct engine engine;

// Suppress link-time optimization that removes unreferenced code

// to make sure glue isn't stripped.

app\_dummy();

memset(&engine, 0, sizeof(engine));

state->userData = &engine;

state->onAppCmd = engine\_handle\_cmd;

state->onInputEvent = engine\_handle\_input;

engine.app = state;

The application prepares to start monitoring the sensors, using the APIs in sensor.h.

engine.sensorManager = ASensorManager\_getInstance();

engine.accelerometerSensor =

ASensorManager\_getDefaultSensor(engine.sensorManager,

ASENSOR\_TYPE\_ACCELEROMETER);

engine.sensorEventQueue =

ASensorManager\_createEventQueue(engine.sensorManager,

state->looper, LOOPER\_ID\_USER, NULL, NULL);

Next, a loop begins, in which the application polls the system for messages (sensor events). It sends messages to android\_native\_app\_glue, which checks to see whether they match any onAppCmd events defined inandroid\_main. When a match occurs, the message is sent to the handler for execution.

while (1) {

// Read all pending events.

int ident;

int events;

struct android\_poll\_source\* source;

// If not animating, we will block forever waiting for events.

// If animating, we loop until all events are read, then continue

// to draw the next frame of animation.

while ((ident=ALooper\_pollAll(engine.animating ? 0 : -1, NULL,

&events,

(void\*\*)&source)) >= 0) {

// Process this event.

if (source != NULL) {

source->process(state, source);

}

// If a sensor has data, process it now.

if (ident == LOOPER\_ID\_USER) {

if (engine.accelerometerSensor != NULL) {

ASensorEvent event;

while (ASensorEventQueue\_getEvents(engine.sensorEventQueue,

&event, 1) > 0) {

LOGI("accelerometer: x=%f y=%f z=%f",

event.acceleration.x, event.acceleration.y,

event.acceleration.z);

}

}

}

// Check if we are exiting.

if (state->destroyRequested != 0) {

engine\_term\_display(&engine);

return;

}

}

Once the queue is empty, and the program exits the polling loop, the program calls OpenGL to draw the screen.

if (engine.animating) {

// Done with events; draw next animation frame.

engine.state.angle += .01f;

if (engine.state.angle > 1) {

engine.state.angle = 0;

}

// Drawing is throttled to the screen update rate, so there

// is no need to do timing here.

engine\_draw\_frame(&engine);

}

}