Chapter 8

Working with Android NDK and Renderscript

Developers typically write Android apps entirely in Java. However, situations arise where it's desirable (or even necessary) to express at least part of the code in another language (notably C or C++). Google addresses these situations by providing the Android Native Development Kit (NDK) and Renderscript.

Android NDK

The Android NDK complements the Android SDK by providing a toolset that lets you implement parts of your app using native code languages such as C and C++. The NDK provides headers and libraries for building native activities, handling user input, using hardware sensors, and more.

Many developers believe that the NDK exists to boost app performance. Although performance can improve, it can also worsen because transitions from the Dalvik virtual machine (VM) equivalent of compiled Java code to native code via the Java Native Interface (JNI) will add overhead, which impacts performance.

NOTE: Code running inside of Dalvik already experiences a performance boost thanks to the Just-In-Time compiler that was integrated with Dalvik in Android 2.2.

The NDK is used in the following scenarios:

- Your app contains CPU-intensive code that doesn't allocate much memory. Code examples include physics simulation, signal processing, huge factorial calculations, and testing huge integers for primeness. Renderscript (discussed later in this chapter) is probably more appropriate for addressing at least some of these examples.
- You want to ease the porting of existing C/C++-based source code to your app. Using the NDK can help to speed up app development by letting you keep most or all of your app's code in C/C++. Furthermore, working with the NDK can help you keep code changes synchronized between Android and non-Android projects.

CAUTION: Think carefully about integrating native code into your app. Basing even part of an app on native code increases its complexity and makes it harder to debug.

Installing the NDK

If you believe that your app can benefit from being at least partly expressed in native code, you'll need to install the NDK. Before doing so, you need to be aware of the following software and system requirements:

- A complete Android SDK installation (including all dependencies) is required. Version 1.5 or later of the SDK is supported.
- The following operating systems are supported: Windows XP (32-bit), Windows Vista (32- or 64-bit), Windows 7 (32- or 64-bit), Mac OS X 10.4.8 or later (x86 only), and Linux (32- or 64-bit; Ubuntu 8.04, or other Linux distributions using glibc 2.7 or later).
- For all platforms, GNU Make 3.81 or later is required. Earlier versions of GNU Make might work but have not been tested. Also, GNU Awk or Nawk is required.
- For Windows platforms, Cygwin (1.7 or higher) is required to support debugging. Before Revision 7 of the NDK, Cygwin was also required to build projects by supplying `make` and `awk` tools.

- The native libraries created by the Android NDK can be used only on devices running specific minimum Android platform versions. The minimum required platform version depends on the CPU architecture of the devices you are targeting. Table 8-1 details which Android platform versions are compatible with native code developed for specific CPU architectures.

Table 8-1. *Mappings Between Native Code CPU Architectures and Compatible Android Platforms*

Native Code CPU Architecture Used	Compatible Android Platforms
ARM, ARM-NEON	Android 1.5 (API Level 3) and higher
x86	Android 2.3 (API Level 9) and higher
MIPS	Android 2.3 (API Level 9) and higher

  These requirements mean that you can use native libraries created via the NDK in apps that are deployable to ARM-based devices running Android 1.5 or later. If you are deploying native libraries to x86- and MIPS-based devices, your app must target Android 2.3 or later.

- To ensure compatibility, an app using a native library created via the NDK must declare a `<uses-sdk>` element in its manifest file, with an `android:minSdkVersion` attribute value of `"3"` or higher. Example:

```
<manifest>
  <uses-sdk android:minSdkVersion="3" />
  ...
</manifest>
```

- If you use the NDK to create a native library that uses the OpenGL ES APIs, the app containing the library can be deployed only to devices running the minimum platform versions described in Table 8-2. To ensure compatibility, make sure that your app declares the proper `android:minSdkVersion` attribute value.

Table 8-2. *Mappings Between OpenGL ES Versions, Compatible Android Platforms, and Uses-SDK*

OpenGL ES Version Used	Compatible Android Platforms	Required `uses-sdk` Attribute
OpenGL ES 1.1	Android 1.6 (API Level 4) and higher	`android:minSdkVersion ="4"`
OpenGL ES 2.0	Android 2.0 (API Level 5) and higher	`android:minSdkVersion ="5"`

- Additionally, an app using the OpenGL ES APIs should declare a `<uses-feature>` element in its manifest, with an `android:glEsVersion` attribute that specifies the minimum OpenGL ES version required by the app. This ensures that Google Play will show your app only to users whose devices can support your app. Example:

```
<manifest>
  <uses-feature android:glEsVersion="0x00020000" />
  ...
</manifest>
```

- If you use the NDK to create a native library that uses the Android API to access `android.graphics.Bitmap` pixel buffers, or utilizes native activities, the app containing the library can be deployed only to devices running Android 2.2 (API level 8) or higher. To ensure compatibility, make sure that your app declares a `<uses-sdk android:minSdkVersion="8" />` element in its manifest.

Point your browser to `http://developer.android.com/tools/sdk/ndk/index.html` and download one of the following NDK packages for your platform—Revision 8b is the latest version at the time of writing:

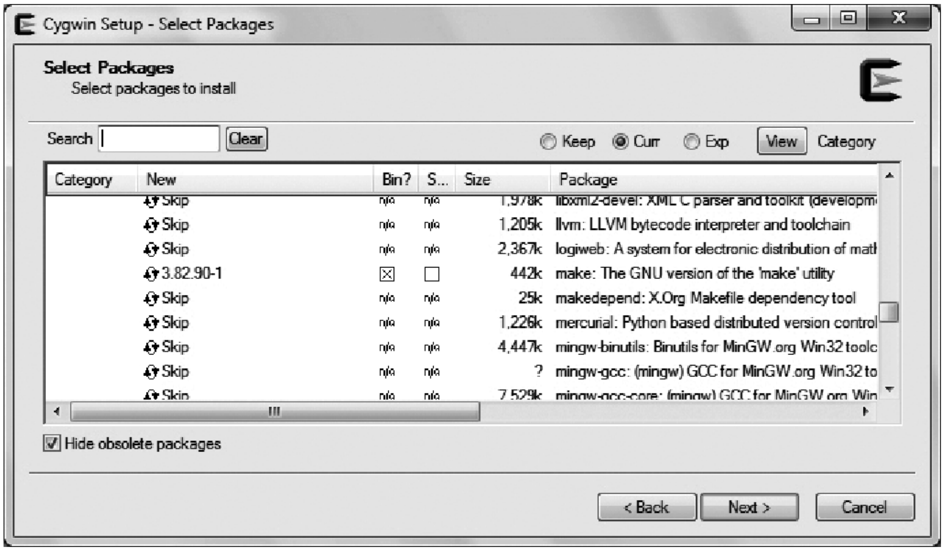
- `android-ndk-r8b-windows.zip` (Windows)
- `android-ndk-r8b-darwin-x86.tar.bz2` (Mac OS X: Intel)
- `android-ndk-r8b-linux-x86.tar.bz2` (Linux 32-/64-bit: x86)

After downloading your chosen package, unarchive it and move its `android-ndk-r8b` home directory to a more suitable location, perhaps to the same directory that contains the Android SDK's home directory.

INSTALLING CYGWIN

Cygwin is a collection of tools that provides a Linux look-and-feel environment for Windows. Complete the following steps to install Cygwin 1.7 or higher when Windows is your platform:

1. Point your browser to http://cygwin.com/.
2. Click the setup.exe link and save this file to your hard drive.
3. Run this program on your Windows platform to begin installing Cygwin version 1.7.16-1 (the latest version at the time of writing). If you choose a different install location (C:\cygwin is the default), make sure that the directory path contains no spaces.
4. When you reach the Select Packages screen, select the Devel category and look for an entry in this category whose Package column presents make: The GNU version of the "make" utility. In the entry's New column, click the word Skip; this word should change to 3.82.90-1. Also, the Bin? column's check box should be checked—see Figure 8-1.



Category	New	Bin?	S...	Size	Package
	Skip	n/a	n/a	1,978k	libxml2-devel: XML C parser and toolkit (developm
	Skip	n/a	n/a	1,205k	llvm: LLVM bytecode interpreter and toolchain
	Skip	n/a	n/a	2,367k	logiweb: A system for electronic distribution of matl
	3.82.90-1	☒	☐	442k	make: The GNU version of the 'make' utility
	Skip	n/a	n/a	25k	makedepend: X.Org Makefile dependency tool
	Skip	n/a	n/a	1,226k	mercurial: Python based distributed version control
	Skip	n/a	n/a	4,447k	mingw-binutils: Binutils for MinGW.org Win32 toolc
	Skip	n/a	n/a	?	mingw-gcc: (mingw) GCC for MinGW.org Win32 to
	Skip	n/a	n/a	7,529k	mingw-gcc-core: (mingw) GCC for MinGW.org Win

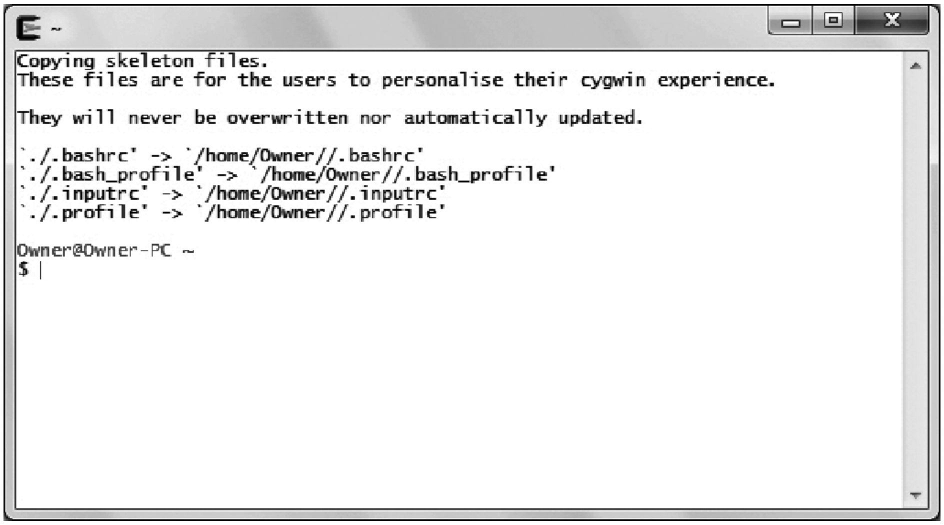
☑ Hide obsolete packages

Figure 8-1. *Make sure that 3.82.90-1 appears in the New column and that the check box in the Bin? column is checked before clicking Next.*

5. Click the Next button and continue the installation.

When installation finishes, Cygwin gives you the opportunity to override its defaults of creating an icon on the desktop and of adding an icon to the Start Menu. After choosing to override these or not, click Finish.

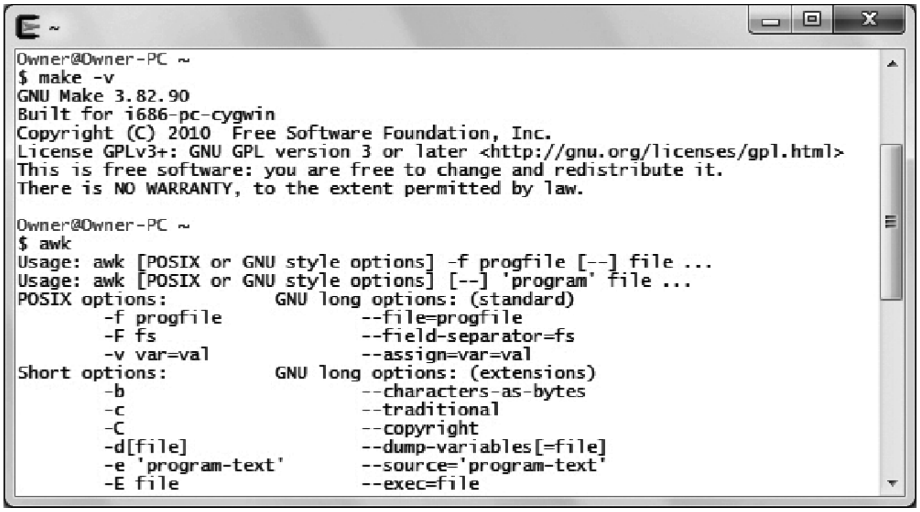
Assuming that you've kept the defaults, click the desktop icon. You should see the Cygwin console (which is based on the Bash shell) shown in Figure 8-2.



```
Copying skeleton files.
These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

`./.bashrc' -> `/home/Owner//.bashrc'
`./.bash_profile' -> `/home/Owner//.bash_profile'
`./.inputrc' -> `/home/Owner//.inputrc'
`./.profile' -> `/home/Owner//.profile'

Owner@Owner-PC ~
$ |
```

Figure 8-2. *Cygwin's console displays initialization messages the first time it starts running.*

If you want to verify that Cygwin provides access to GNU Make 3.81 or later and GNU Awk, enter the commands shown in Figure 8-3.



```
Owner@Owner-PC ~
$ make -v
GNU Make 3.82.90
Built for i686-pc-cygwin
Copyright (C) 2010  Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Owner@Owner-PC ~
$ awk
Usage: awk [POSIX or GNU style options] -f progfile [--] file ...
Usage: awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options: (standard)
        -f progfile             --file=progfile
        -F fs                   --field-separator=fs
        -v var=val              --assign=var=val
Short options:          GNU long options: (extensions)
        -b                      --characters-as-bytes
        -c                      --traditional
        -C                      --copyright
        -d[file]                --dump-variables[=file]
        -e 'program-text'       --source='program-text'
        -E file                 --exec=file
```

Figure 8-3. *The `Awk` tool doesn't display a version number.*

You can learn more about Cygwin by checking out http://`cygwin.com` as well as Wikipedia's Cygwin entry (http://`en.wikipedia.org/wiki/Cygwin`).

Exploring the NDK

Now that you've installed the NDK on your platform, you might want to explore its home directory to discover what the NDK offers. The following list describes those directories and files that are located in the home directory for the Windows-based NDK:

- `build` contains the files that compose the NDK's build system.
- `docs` contains the NDK's HTML-based documentation files.
- `platforms` contains subdirectories that contain header files and shared libraries for each of the Android SDK's installed Android platforms.
- `prebuilt` contains binaries (notably `make.exe` and `awk.exe`) that let you build NDK source code without requiring Cygwin.
- `samples` contains various sample apps that demonstrate different aspects of the NDK.

- `sources` contains the source code and prebuilt binaries for various shared libraries, such as `cpufeatures` (detect the target device's CPU family and the optional features it supports) and `stlport` (multiplatform C++ standard library). Android NDK 1.5 required that developers organize their native code library projects under this directory. Starting with Android NDK 1.6, native code libraries are stored in `jni` subdirectories of their Android app project directories.
- `tests` contains scripts and sources to perform automated testing of the NDK. They are useful for testing a custom-built NDK.
- `toolchains` contains compilers, linkers, and other tools for generating native ARM (Advanced RISC Machine, the CPU used by Android—`http://en.wikipedia.org/wiki/ARM_architecture`) binaries on Linux, OS X, and Windows (with Cygwin) platforms.
- `documentation.html` is the entry point into the NDK's documentation.
- `GNUmakefile` is the default make file used by GNU Make.
- `ndk-build` is a shell script that simplifies building machine code.
- `ndk-build.cmd` is a Windows `cmd.exe` script that invokes the `prebuilt\windows\bin\make.exe` executable.
- `ndk-gdb` is a shell script that easily launches a native debugging session for your NDK-generated machine code. (Cygwin is required to run this script on Windows platforms.)
- `ndk-stack.exe` lets you filter stack traces as they appear in the output generated by `adb logcat` and replace any address inside a shared library with the corresponding values. In essence, it lets you observe more readable crash dump information.
- `README.TXT` welcomes you to the NDK, and it refers you to various documentation files that inform you about changes in the current release (and more).
- `RELEASE.TXT` contains the NDK's release number (`r8b`).

Each of the `platforms` directory's subdirectories contains header files that target stable native APIs. Google guarantees that all later platform releases will support the following APIs (see also `http://developer.android.com/tools/sdk/ndk/overview.html#tools`):

- Android logging (`liblog`)
- Android native app APIs
- C library (`libc`)
- C++ minimal support (`stlport`)
- JNI interface APIs
- Math library (`libm`)
- OpenGL ES 1.1 and OpenGL ES 2.0 (3D graphics libraries) APIs
- OpenSL ES native audio library APIs
- Pixel buffer access for Android 2.2 and above (`libjnigraphics`)
- Zlib compression (`libz`)

CAUTION: Native system libraries that are not in this list are not stable and may change in future versions of the Android platform. Do not use them.

Greetings from the NDK

Perhaps the easiest way to become familiar with NDK programming is to create a small app that calls a native function that returns a Java `String` object. For example, Listing 8-1's `NDKGreetings` single-activity-based app calls a native `getGreetingMessage()` method to return a greeting message, which it displays via a dialog box.

Listing 8-1. *Receiving Greetings from the NDK*

```
package ca.tutortutor.ndkgreetings;

import android.app.Activity;
import android.app.AlertDialog;

import android.os.Bundle;
```

```
public class NDKGreetings extends Activity
{
   static
   {
      System.loadLibrary("NDKGreetings");
   }

   private native String getGreetingMessage();

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      String greeting = getGreetingMessage();
      new AlertDialog.Builder(this).setMessage(greeting).show();
   }
}
```

Listing 8-1's `NDKGreetings` class reveals the following three important features of every app that incorporates native code:

- Native code is stored in an external library that must be loaded before its code can be invoked. Libraries are typically loaded at class-loading time via `System.loadLibrary()`. This method takes a single `String` argument that identifies the library without its `lib` prefix and `.so` suffix. In this example, the actual library file is named `libNDKGreetings.so`. (If the library cannot be located, an instance of the `java.lang.UnsatisfiedLinkError` class is thrown, which causes Android to terminate your app.)
- One or more native methods are declared that correspond to functions located within the library. A native method is identified to Java by prefixing its return type with the keyword `native`.
- A native method is invoked like any other Java method. Behind the scenes, Dalvik makes sure that the corresponding native function (expressed in C/C++) is invoked in the library.

Listing 8-2 presents the C source code to a native code library that implements `getGreetingMessage()` via the JNI.

Listing 8-2. *Implementing a Greetings Response to Dalvik*

```
#include <jni.h>

jstring
   Java_ca_tutortutor_ndkgreetings_NDKGreetings_getGreetingMessage(JNIEnv* env,
                                                                    jobject this)
{
   return (*env)->NewStringUTF(env, "Greetings from the NDK!");
}
```

Listing 8-2 first specifies an `#include` preprocessor directive that includes the contents of the `jni.h` header file when the source code is compiled. This file specifies various JNI constants, types, and function prototypes.

Listing 8-2 then declares the native function equivalent of Listing 8-1's `getGreetingMessage()` method. This native function's header reveals several important items:

- The native function's return type is specified as `jstring`. This type is defined in `jni.h` and represents Java's `java.lang.String` object type at the native code level.
- The function's name must begin with the Java package and class names that identify where the associated native method is declared.
- The type of the function's first parameter, `env`, is specified as a `JNIEnv` pointer. `JNIEnv`, which is defined in `jni.h`, is a C struct that identifies JNI functions that can be called to interact with Java.
- The type of the function's second parameter, `this`, is specified as `jobject`. This type, which is defined in `jni.h`, identifies an arbitrary Java object at the native code level. The argument passed to this parameter is the implicit `this` instance that the Java VM passes to any Java instance method.

The function de-references its `env` parameter in order to call the `NewStringUTF()` JNI function. `NewStringUTF()` converts its second argument, a C string, to its `jstring` equivalent (where the string is encoded via the Unicode UTF encoding standard), and it returns this equivalent Java string, which is then returned to Java.

NOTE: When working with the JNI in the context of the C language, you must de-reference the `JNIEnv` parameter (`*env`, for example) in order to call a JNI function. Also, you must pass the `JNIEnv` parameter as the first argument to the JNI function. In contrast, C++ doesn't require this verbosity: you don't have to de-reference the `JNIEnv` parameter, and you don't have to pass this parameter as the first argument to the JNI function. For example, Listing 8-2's C-based `(*env)->NewStringUTF(env, "Greetings from the NDK!")` function call is expressed as `env->NewStringUTF("Greetings from the NDK!")` in C++.

Building and Running NDKGreetings with the Android SDK

To build `NDKGreetings` with the Android SDK, first use the SDK's `android` tool to create an `NDKGreetings` project. Assuming a Windows platform, a `C:\prj\dev` hierarchy in which the `NDKGreetings` project is to be stored (in `C:\prj\dev\NDKGreetings`), and an Android 4.1 platform target that corresponds to integer ID 1 (execute `android list targets` to obtain the correct ID), execute the following command (spread across two lines for readability) to create `NDKGreetings`:

```
android create project -t 1 -p C:\prj\dev\NDKGreetings -a NDKGreetings
                       -k ca.tutortutor.ndkgreetings
```

This command creates various directories and files within `C:\prj\dev\NDKGreetings`. For example, the `src` directory contains the `ca\tutortutor\ndkgreetings` directory structure, and the final `ndkgreetings` directory contains a skeletal `NDKGreetings.java` source file. Replace this skeletal file's contents with Listing 8-1.

Then create a `jni` directory within `C:\prj\dev\NDKGreetings`, and copy Listing 8-2 to `C:\prj\dev\NDKGreetings\jni\NDKGreetings.c`. Also, copy Listing 8-3 to `C:\prj\dev\NDKGreetings\jni\Android.mk`, which is a GNU Make file (explained in the NDK documentation) that's used to create the `libNDKGreetings.so` library.

Listing 8-3. *A Make File for* `NDKGreetings`

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

include $(CLEAR_VARS)

LOCAL_MODULE    := NDKGreetings
LOCAL_SRC_FILES := NDKGreetings.c
```

```
include $(BUILD_SHARED_LIBRARY)
```

Execute the following command from within the `C:\prj\dev\NDKGreetings` directory:

```
\android-ndk-r8b\ndk-build
```

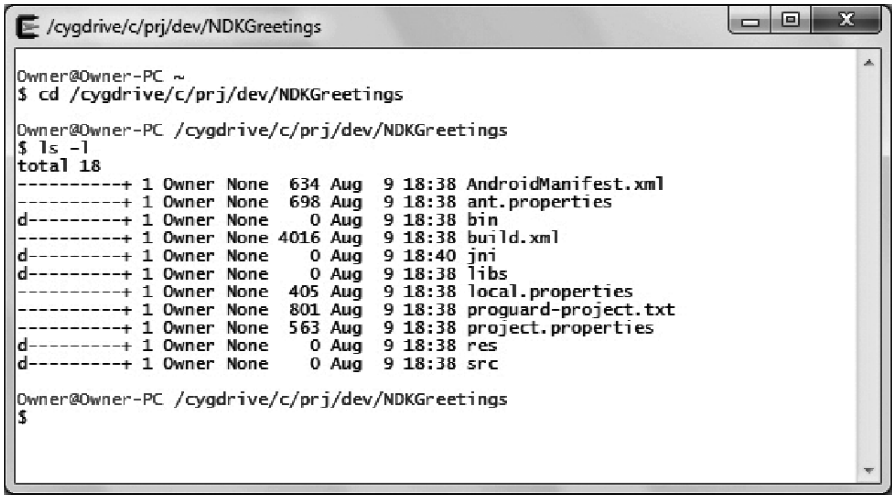
This command launches (on Windows) the `ndk-build.cmd` script to build the library. When the build is successful, the following messages are output:

```
Compile thumb  : NDKGreetings <= NDKGreetings.c
SharedLibrary  : libNDKGreetings.so
Install        : libNDKGreetings.so => libs/armeabi/libNDKGreetings.so
```

This output indicates that `libNDKGreetings.so` is located in the `armeabi` subdirectory of your `NDKGreetings` project directory's `libs` subdirectory.

NOTE: If you observe a "No rule to make target" message instead of the output above, the cause is most likely extra spaces in `Android.mk`.

Alternatively, you can use Cygwin (assuming that it has been installed as previously discussed) to accomplish this task. Run Cygwin (if it is not running) and, from within the Cygwin command window, set the current directory to `C:\prj\dev\NDKGreetings`. See Figure 8-4.



```
Owner@Owner-PC ~
$ cd /cygdrive/c/prj/dev/NDKGreetings

Owner@Owner-PC /cygdrive/c/prj/dev/NDKGreetings
$ ls -l
total 18
----------+ 1 Owner None  634 Aug  9 18:38 AndroidManifest.xml
----------+ 1 Owner None  698 Aug  9 18:38 ant.properties
d---------+ 1 Owner None    0 Aug  9 18:38 bin
----------+ 1 Owner None 4016 Aug  9 18:38 build.xml
d---------+ 1 Owner None    0 Aug  9 18:40 jni
d---------+ 1 Owner None    0 Aug  9 18:38 libs
----------+ 1 Owner None  405 Aug  9 18:38 local.properties
----------+ 1 Owner None  801 Aug  9 18:38 proguard-project.txt
----------+ 1 Owner None  563 Aug  9 18:38 project.properties
d---------+ 1 Owner None    0 Aug  9 18:38 res
d---------+ 1 Owner None    0 Aug  9 18:38 src

Owner@Owner-PC /cygdrive/c/prj/dev/NDKGreetings
$
```

Figure 8-4. *The path to `/prj/dev/NDKGreetings` begins with a `/cygdrive/c` prefix.*

Assuming that the NDK home directory is `android-ndk-r8b` and that it's located in the root directory of the C drive, execute the following command (in Cygwin) to build the library:

```
../../../android-ndk-r8b/ndk-build
```

You should observe the same output messages as previously shown.

Assuming that `C:\prj\dev\NDKGreetings` is current, execute the following command (from Cygwin's shell or the normal Windows command window) to create `NDKGreetings-debug.apk`:

```
ant debug
```

This APK file is placed in the `NDKGreetings` project directory's `bin` subdirectory. To verify that `libNDKGreetings.so` is part of this APK, run the following command from `bin`:

```
jar tvf NDKGreetings-debug.apk
```

You should observe a line containing `lib/armeabi/libNDKGreetings.so` among the `jar` command's output.

To verify that the app works, start the emulator, which you can accomplish at the command line by executing the following command:

```
emulator -avd AVD1
```

This command assumes that you've created the AVD1 device configuration as specified in Chapter 1.

Install `NDKGreetings-debug.apk` on the emulated device via the following command:

```
adb install NDKGreetings-debug.apk
```

This command assumes that `adb` is located in your path. It also assumes that `bin` is the current directory.

When `adb` indicates that `NDKGreetings-debug.apk` has been installed, navigate to the app launcher screen and click the NDKGreetings icon. Figure 8-5 shows you the result.

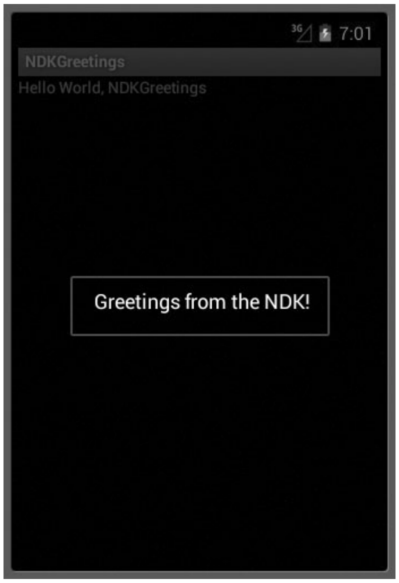


Figure 8-5. *Press the Esc key (in Windows) to make the dialog box go away.*

The dialog box displays the "Greetings from the NDK!" message that was obtained by calling the native function in the native code library. It also reveals a faint "Hello World, NDKGreetings" message near the top of the screen. This message originates in the project's default `main.xml` file that's created by the `android` tool.

Building and Running NDKGreetings with Eclipse

To build `NDKGreetings` with Eclipse, first create a new Android project as described in Chapter 1's Recipe 1-10. For your convenience, the steps that you need to follow to accomplish this task are provided here:

1. Start Eclipse if it is not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting *New Project* dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.

4. On the resulting *New Android App* dialog box, enter **`NDKGreetings`** into the Application Name textfield. This entered name also appears in the Project Name textfield, and it identifies the folder/directory in which the `NDKGreetings` project is stored.
5. Enter **`ca.tutortutor.ndkgreetings`** into the Package Name textfield.
6. Via Build SDK, select the appropriate Android SDK to target. This selection identifies the Android platform you'd like your app to be built against. Assuming that you've installed only the Android 4.1 platform, only this choice should appear and be selected.
7. Via Minimum SDK, either select the minimum Android SDK on which your app runs or keep the default setting.
8. Leave the "Create custom launcher icon" check box checked if you want a custom launcher icon to be created. Otherwise, uncheck this check box when you supply your own launcher icon.
9. Leave the "Mark this project as a library" check box unchecked because you are not creating a library.
10. Leave the "Create Project in Workspace" check box checked, and click Next.
11. On the resulting Configure Launcher Icon pane, make suitable adjustments to the custom launcher icon and then click Next.
12. On the resulting Create Activity pane, leave the Create Activity check box checked, make sure that BlankActivity is selected, and click Next.
13. On the resulting New Blank Activity pane, enter **`NDKGreetings`** into the Activity Name textfield, and **`main`** into the Layout Name textfield. Keep all other settings and click Finish.

Then use Eclipse's Package Explorer to locate the `NDKGreetings.java` source file node. Double-click this node and replace the skeletal contents shown in the resulting edit window with Listing 8-1.

Using Package Explorer, create a jni folder node below the NDKGreetings project node, add an `NDKGreetings.c` file subnode of jni, replace this node's empty contents with Listing 8-2, add a new `Android.mk` file subnode of jni, and replace its empty contents with Listing 8-3.

At this point, you can use Cygwin to create the library file, or you can create a builder to do this for you. To use Cygwin, launch this tool if it is not running, and use the `cd` command to change to the project's folder (for example, `cd /cygdrive/c/users/owner/workspace/NDKGreetings`). Then execute `ndk-build` as demonstrated in the previous section (for example, `/cygdrive/c/android-ndk-r8b/ndk-build`). If all goes well, the `NDKGreetings` project directory's `libs` subdirectory should contain an `armeabi` subdirectory, which should contain a `libNDKGreetings.so` library file.

Complete the following steps to create a builder:

1. Right-click the NDKGreetings node, and select Properties from the resulting pop-up menu.
2. Select Builders on the resulting *Properties for NDKGreetings* dialog box.
3. On the resulting Builders pane, click the New button.
4. On the resulting *Choose configuration type* dialog box, select Program and click OK.
5. On the resulting *Edit Configuration* dialog box, choose whatever name you want for the builder (or keep the default), enter **`C:\android-ndk-r8b\ndk-build.cmd`** (or your equivalent) into the Location textfield, enter **`${workspace_loc:/NDKGreetings}`** into the Working Directory textfield, and click the OK button to close this dialog box.
6. Click the OK button to close the *Properties for NDKGreetings* dialog box.

To run `NDKGreetings` from Eclipse, select Run from the menubar, and then select Run from the drop-down menu. If a *Run As* dialog box appears, select Android Application and click OK. Eclipse launches `emulator` with the AVD1 device, installs `NDKGreetings.apk`, and runs this app, whose output appears in Figure 8-6.

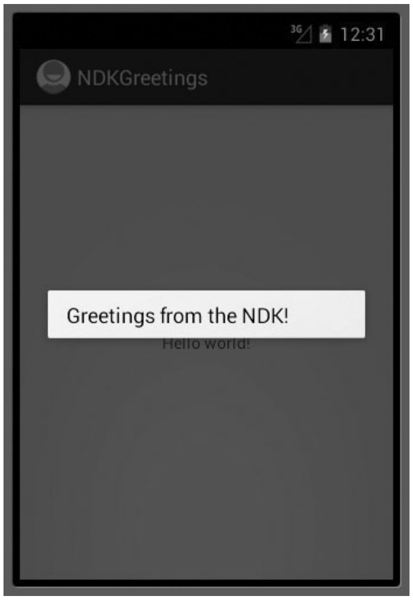


Figure 8-6. *`NDKGreetings`'s user interface looks different because of an Eclipse-generated custom theme.*

Sampling the NDK

The `samples` subdirectory of the NDK installation's home directory contains several sample apps that demonstrate different aspects of the NDK:

- `bitmap-plasma`: An app that demonstrates how to access the pixel buffers of Android `android.graphics.Bitmap` objects from native code, and uses this capability to generate an old-school "plasma" effect.
- `hello-gl2`: An app that renders a triangle using OpenGL ES 2.0 vertex and fragment shaders. (If you run this app on the Android emulator, you will receive an error message stating that the app has stopped unexpectedly when the emulator doesn't support OpenGL ES 2.0 hardware emulation.)
- `hello-jni`: An app that loads a string from a native method implemented in a shared library and then displays it in the app's user interface. This app is similar to `NDKGreetings`.

- hello-neon: An app that shows how to use the cpufeatures library to check CPU capabilities at runtime, and then uses NEON (a marketing name of a SIMD instruction set for the ARM architecture) intrinsics if supported by the CPU. Specifically, the app implements two versions of a tiny benchmark for an FIR filter loop (http://en.wikipedia.org/wiki/Finite_impulse_response): a C version and a NEON-optimized version for devices that support it.
- native-activity: An app that demonstrates how to use the native-app-glue static library to create a *native activity* (an activity implemented entirely in native code).
- native-audio: An app that demonstrates how to use native methods to play sounds via OpenSL ES.
- native-plasma: A version of bitmap-plasma implemented with a native activity.
- san-angeles: An app that renders 3D graphics through the native OpenGL ES APIs, while managing the activity life cycle with an android.opengl.GLSurfaceView object.
- two-libs: An app that loads a shared library dynamically and calls a native method provided by the library. In this case, the method is implemented in a static library imported by the shared library.

You can use Eclipse to build these apps. For example, carry out the following steps to build san-angeles:

1. Start Eclipse if it is not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting *New Project* dialog box, expand the Android node in the wizard tree (if necessary), select the Android Project from Existing Code branch below this node, and click the Next button.
4. On the resulting Import Projects pane, click the Browse button.
5. On the resulting *Browse for Folder* dialog box, select the NDK's san-angeles directory, which is under the samples directory. Click OK to close this dialog box.

6. Back on the Import Projects pane, check the "Copy projects into workspace" check box and click the Finish button. A `com.example.SanAngeles.DemoActivity` node should appear in Package Explorer. Furthermore, a `com.example.SanAngeles.DemoActivity` project directory should appear in the workspace. This directory contains a separate copy of the NDK's `san-angeles` project.
7. Right-click the `com.example.SanAngeles.DemoActivity` node, and select Properties from the resulting pop-up menu.
8. On the resulting *Properties for com.example.SanAngeles.DemoActivity* dialog box, select Builders.
9. On the resulting Builders pane, click the New button.
10. On the resulting *Choose configuration type* dialog box, select Program and click OK.
11. On the resulting *Edit Configuration* dialog box, choose whatever name you want for the builder (or keep the default), enter **`C:\android-ndk-r8b\ndk-build.cmd`** (or your equivalent) into the Location textfield, enter **`${workspace_loc:/com.example.SanAngeles.DemoActivity}`** into the Working Directory textfield, and click the OK button to close this dialog box.
12. Close the Properties for com.example.SanAngeles.DemoActivity dialog box by clicking OK.

With `com.example.SanAngelese.DemoActivity` as the selected node in Package Explorer, select Run from the menubar and Run from the drop-down menu. If a *Run As* dialog box appears, select Android Application and click OK. If you encounter a dialog box claiming that your project has errors, close this dialog box and select Run again.

This time, Eclipse should launch `emulator` with the AVD1 device that you created in Chapter 1. It should install `DemoActivity.apk` on this device and run this app. After unlocking the home screen, you should see a continuously moving screen with content similar to that shown in Figure 8-7 (it may take a few moments to appear).

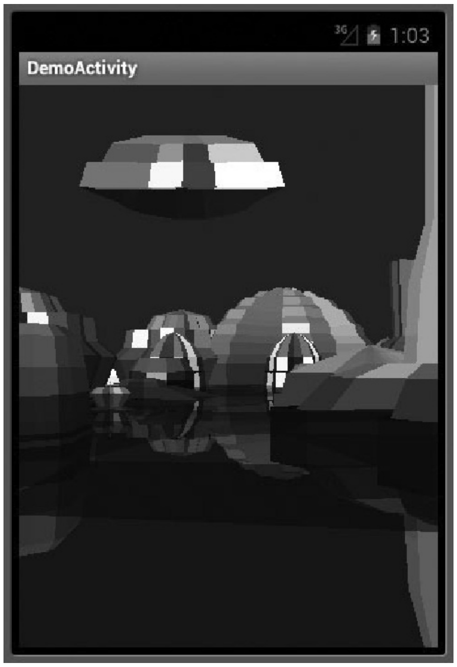


Figure 8-7. *`DemoActivity` takes you on a tour of a three-dimensional city.*

8-1. Discovering Native Activities

Problem

You know that Android supports native activities and you want to learn more about them.

Solution

A *native activity* is an activity that's implemented entirely in native code. First appearing in Android 2.3 (API Level 9) via the `android.app.NativeActivity` class, and in Revision 5 of the NDK, which provides support for developing them, native activities let you implement apps in C/C++ without writing any Java code.

NOTE: A `NativeActivity` instance is equivalent to an `android.app.Activity` instance that performs JNI calls to native code.

How It Works

`NativeActivity` is a helper class that lets you write a completely native activity and, by extension, a completely native app. It handles communication between the Android framework and your native code. You don't have to subclass it or call its methods. Instead, create your native app and declare it to be native in `AndroidManifest.xml`.

Native activities don't change the fact that Android apps still run in their own VMs, where they are sandboxed from other apps. Because of this, you can still access Android framework APIs through the JNI. However, there are also native interfaces that you can use to access sensors, input events and assets, and so on.

NOTE: To learn what APIs can be accessed by native activities, check out the list of stable native APIs that was presented earlier in this chapter.

The NDK offers two choices for developing a native activity:

- *Low-level*: The `native_activity.h` header file (located in `platforms/android-9/arch-arm/usr/include/android` and similar subdirectories of the NDK's home directory) defines the native version of the `NativeActivity` class. It contains the callback interface and data structures that you need to create your native activity. Because your app's main thread handles callbacks, your callback implementations must not be blocking. If they block, you might receive "Application Not Responding" errors because the main thread will be unresponsive until the callback returns. Check out the comments in `native_activity.h` for more information.

- *High-level*: The android_native_app_glue.h header file (located in the sources/android/native_app_glue subdirectory of the NDK's home directory) defines a static helper library built on top of native_activity.h. It spawns another thread to handle callbacks and input events. This spawned thread is used to prevent any callbacks from blocking the main thread, and it adds some flexibility in how you implement callbacks, so you might find this programming model a bit easier to implement. You can modify the android_native_app_glue.c source file (located in the same directory) when you need to change its functionality. Check out the comments in android_native_app_glue.h for more information.

You will learn more about native activities in the next two recipes, which show you how to develop similar native activities in low-level and high-level contexts. Furthermore, each recipe shows you how to develop its low-level or high-level native activity by using the Android SDK and Eclipse.

8-2. Developing Low-Level Native Activities

Problem

You want to learn how to develop low-level native activities, which are based on the native_activity.h header file.

Solution

Create a low-level native activity project as if it were a regular Android app project. Then modify its AndroidManifest.xml file appropriately, and introduce a jni subdirectory of the project directory that contains the native activity's C/C++ source code along with an Android.mk make file.

The modified AndroidManifest.xml file differs from the regular AndroidManifest.xml file in the following ways:

- A <uses-sdk android:minSdkVersion="9"/> element precedes the <application> element; native activities require at least API Level 9.
- An android:hasCode="false" attribute appears in the <application> tag because native activities don't contain source code.

- The `<activity>` element's `android:name` attribute contains the value "`android.app.NativeActivity`". When Android discovers this value, it locates the appropriate entry point in the native activity's library.
- A `<meta-data>` element precedes the `<intent-filter>` element. `<meta-data>` specifies an `android:name="android.app.lib_name"` attribute and an `android:value` attribute whose value is the name of the native activity's library (without a `lib` prefix and a `.so` suffix).

Your native activity's C/C++ source file must define the following entry-point method:

```
void ANativeActivity_onCreate(ANativeActivity* activity, void* savedState,
                              size_t savedStateSize)
```

This method declares the following parameters:

- `activity`: This is the address of an `ANativeActivity` structure. `ANativeActivity` is defined in the NDK's `native_activity.h` header file, and it declares various members, including `callbacks` (an array of pointers to callback functions; you can set these pointers to your own callbacks), `internalDataPath` (the path to the app's internal data directory), `externalDataPath` (the path to the app's external [removable/mountable] data directory), `sdkVersion` (the platform's SDK version number), and `assetManager` (a pointer to an instance of the native equivalent of the app's `android.content.res.AssetManager` class for accessing binary assets bundled into the app's APK file).
- `savedState`: This is your activity's previously saved state. If the activity is being instantiated from a previously saved instance, `savedState` will be non-`NULL` and will point to the saved data. You must make a copy of this data when you need to access it later, because memory allocated to `savedState` will be released after you return from this function.
- `savedStateSize`: This is the size (in bytes) of the data pointed to by `savedState`.

> **NOTE:** When you launch an app that is based on a native activity, an instance of the `android.app.NativeActivity` class is created. Its `onCreate(Bundle)` method uses the JNI to call `void ANativeActivity_onCreate(ANativeActivity*, void*, size_t)`.

`void ANativeActivity_onCreate(ANativeActivity*, void*, size_t)` should override any needed callbacks. It must also create a thread that promptly responds to input events in order to prevent an "Application Not Responding" error from occurring.

> **NOTE:** `void ANativeActivity_onCreate(ANativeActivity*, void*, size_t)` and your callback methods must not delay their execution; otherwise, an "Application Not Responding" error will occur.

Finally, the `Android.mk` file is nearly identical to what you've already seen. However, this file will most likely include a `LOCAL_LDLIBS` entry that identifies any required libraries. (These libraries will undoubtedly include the standard `libandroid.so` library.)

How It Works

Consider an `LLNADemo` project that demonstrates low-level native activities. Listing 8-4 presents the contents of this project's solitary `llnademo.c` source file.

Listing 8-4. *Examining a Native Activity from a Low Perspective*

```
#include <android/log.h>
#include <android/native_activity.h>
#include <pthread.h>

#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO, \
                                             "llnademo", \
                                             __VA_ARGS__))

AInputQueue* _queue;
pthread_t thread;
pthread_cond_t cond;
pthread_mutex_t mutex;
```

```
static void onConfigurationChanged(ANativeActivity* activity)
{
   LOGI("ConfigurationChanged: %p\n", activity);
}

static void onDestroy(ANativeActivity* activity)
{
   LOGI("Destroy: %p\n", activity);
}

static void onInputQueueCreated(ANativeActivity* activity, AInputQueue* queue)
{
   LOGI("InputQueueCreated: %p -- %p\n", activity, queue);
   pthread_mutex_lock(&mutex);
   _queue = queue;
   pthread_cond_broadcast(&cond);
   pthread_mutex_unlock(&mutex);
}

static void onInputQueueDestroyed(ANativeActivity* activity, AInputQueue* queue)
{
   LOGI("InputQueueDestroyed: %p -- %p\n", activity, queue);
   pthread_mutex_lock(&mutex);
   _queue = NULL;
   pthread_mutex_unlock(&mutex);
}

static void onLowMemory(ANativeActivity* activity)
{
   LOGI("LowMemory: %p\n", activity);
}

static void onNativeWindowCreated(ANativeActivity* activity,
                                  ANativeWindow* window)
{
   LOGI("NativeWindowCreated: %p -- %p\n", activity, window);
}

static void onNativeWindowDestroyed(ANativeActivity* activity,
                                    ANativeWindow* window)
{
   LOGI("NativeWindowDestroyed: %p -- %p\n", activity, window);
}

static void onPause(ANativeActivity* activity)
{
   LOGI("Pause: %p\n", activity);
}
```

```
static void onResume(ANativeActivity* activity)
{
   LOGI("Resume: %p\n", activity);
}

static void* onSaveInstanceState(ANativeActivity* activity, size_t* outLen)
{
   LOGI("SaveInstanceState: %p\n", activity);
   return NULL;
}

static void onStart(ANativeActivity* activity)
{
   LOGI("Start: %p\n", activity);
}

static void onStop(ANativeActivity* activity)
{
   LOGI("Stop: %p\n", activity);
}

static void onWindowFocusChanged(ANativeActivity* activity, int focused)
{
   LOGI("WindowFocusChanged: %p -- %d\n", activity, focused);
}

static void* process_input(void* param)
{
   while (1)
   {
      pthread_mutex_lock(&mutex);
      if (_queue == NULL)
         pthread_cond_wait(&cond, &mutex);
      AInputEvent* event = NULL;
      while (AInputQueue_getEvent(_queue, &event) >= 0)
      {
         if (AInputQueue_preDispatchEvent(_queue, event))
            break;
         AInputQueue_finishEvent(_queue, event, 0);
      }
      pthread_mutex_unlock(&mutex);
   }
}

void ANativeActivity_onCreate(ANativeActivity* activity,
                              void* savedState,
                              size_t savedStateSize)
{
   LOGI("Creating: %p\n", activity);
   LOGI("Internal data path: %s\n", activity->internalDataPath);
   LOGI("External data path: %s\n", activity->externalDataPath);
   LOGI("SDK version code: %d\n", activity->sdkVersion);
   LOGI("Asset Manager: %p\n", activity->assetManager);

   activity->callbacks->onConfigurationChanged = onConfigurationChanged;
   activity->callbacks->onDestroy = onDestroy;
   activity->callbacks->onInputQueueCreated = onInputQueueCreated;
   activity->callbacks->onInputQueueDestroyed = onInputQueueDestroyed;
   activity->callbacks->onLowMemory = onLowMemory;
   activity->callbacks->onNativeWindowCreated = onNativeWindowCreated;
   activity->callbacks->onNativeWindowDestroyed = onNativeWindowDestroyed;
   activity->callbacks->onPause = onPause;
   activity->callbacks->onResume = onResume;
   activity->callbacks->onSaveInstanceState = onSaveInstanceState;
   activity->callbacks->onStart = onStart;
   activity->callbacks->onStop = onStop;
   activity->callbacks->onWindowFocusChanged = onWindowFocusChanged;

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&cond, NULL);
   pthread_create(&thread, NULL, process_input, NULL);
}
```

Listing 8-4 begins with three `#include` directives that (before compilation) include the contents of three NDK header files for logging, native activities, and Portable Operating System Interface (POSIX) threading.

NOTE: If you are unfamiliar with POSIX, check out Wikipedia's "POSIX" entry (http://`en.wikipedia.org/wiki/POSIX`).

Listing 8-4 next declares a `LOGI` macro for logging information messages to the Android device's log (you can view this log by executing `adb logcat`). This macro refers to the `int __android_log_print(int prio, const char* tag, const char* fmt, ...)` function (prototyped in the `log.h` header file) that performs the actual writing. Each logged message must have a priority (such as `ANDROID_LOG_INFO`), a tag (such as `llnademo`), and a format string defining the message. Additional arguments are specified when the format string contains format specifiers (such as `%d`).

Listing 8-4 then declares a `_queue` variable of type `AInputQueue*`. (`AInputQueue` is defined in the `input.h` header file, which is included by the `native_activity.h` header file.) This variable is assigned a reference to the input queue when the queue is created, or it is assigned `NULL` when the queue is destroyed. The native activity must process all input events from this queue to avoid an "Application Not Responding" error.

Three POSIX thread global variables are now created: thread, cond, and mutex. The variable thread identifies the thread that is created later on in the listing, and the variables cond and mutex are used to avoid busy waiting and to ensure synchronized access to the shared _queue variable, respectively.

A series of "on"-prefixed callback functions follows. Each function is declared static to hide it from outside of its module. (The use of static isn't essential but is present for good form.)

Each "on"-prefixed callback function is called on the main thread and logs some information for viewing in the device log. However, the void onInputQueueCreated(ANativeActivity* activity, AInputQueue* queue) and void onInputQueueDestroyed(ANativeActivity* activity, AInputQueue* queue) functions have a little more work to accomplish:

- onInputQueueCreated(ANativeActivity*, AInputQueue*) must assign its queue argument address to the _queue variable. Because _queue is also accessed from a thread apart from the main thread, synchronization is required to ensure that there is no conflict between these threads. Synchronization is achieved by accessing _queue between pthread_mutex_lock(&mutex) and pthread_mutex_unlock(&mutex) calls. The former call locks a *mutex* (a program object used to prevent multiple threads from simultaneously accessing a shared variable); the latter call unlocks the mutex. Because the non-main thread waits until _queue contains a non-NULL value, a pthread_cond_broadcast(&cond) call is also present to wake up this waiting thread.
- onInputQueueDestroyed(ANativeActivity*, AInputQueue*) is simpler, assigning NULL to _queue (within a locked region) when the input queue is destroyed.

The non-main thread executes the void* process_input(void* param) function. This function repeatedly executes int32_t AInputQueue_getEvent(AInputQueue* queue, AInputEvent** outEvent) to return the next input event. The integer return value is negative when no events are available or when an error occurs. When an event is returned, it is referenced by outEvent.

Assuming that an event has been returned, int32_t AInputQueue_preDispatchEvent(AInputQueue* queue, AInputEvent* event) is called to send the event (if it is a keystroke-related event) to the current input method editor to be consumed before the app. This function returns 0 when the event was not predispatched, which means that you can process it right now.

When a nonzero value is returned, you must not process the current event so that the event can appear again in the event queue (assuming that it does not get consumed during predispatching).

At this point, you could do something with the event (when it is not predispatched). Regardless, you lastly call `void AInputQueue_finishEvent(AInputQueue* queue, AInputEvent* event, int handled)` to finish the dispatching of the given event. A `0` value is passed to `handled` to indicate that the event has not been handled in your code.

Finally, Listing 8-4 declares `void ANativeActivity_onCreate(ANativeActivity*, void*, size_t)`, which logs a message, overrides most of the default callbacks (you could also override the rest when desired), initializes the mutex and the condition variable, and finally creates and starts the thread that runs `void* process_input(void*)`.

Listing 8-5 presents this project's `Android.mk` file.

Listing 8-5. *A Make File for `LLNADemo`*

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := llnademo
LOCAL_SRC_FILES := llnademo.c
LOCAL_LDLIBS := -llog -landroid
include $(BUILD_SHARED_LIBRARY)
```

This make file presents a `LOCAL_LDLIBS` entry, which identifies the `liblog.so` and `libandroid.so` standard libraries that are to be linked against.

Building and Running LLNADemo with the Android SDK

To build `LLNADemo` with the Android SDK, first use the SDK's `android` tool to create an `LLNADemo` project. Assuming a Windows platform, a `C:\prj\dev` hierarchy in which the `LLNADemo` project is to be stored (in `C:\prj\dev\LLNADemo`), and an Android 4.1 platform target corresponding to integer ID 1 (execute `android list targets` to obtain the correct ID), execute the following command (spread across two lines for readability) to create `LLNADemo`:

```
android create project -t 1 -p C:\prj\dev\LLNADemo -a LLNADemo
                       -k ca.tutortutor.llnademo
```

This command creates various directories and files within `C:\prj\dev\LLNADemo`. To reduce the size of the APK file, you can delete the `src` directory because this directory and its contents will not be needed. Also, you can delete all directories underneath res except for values, because they and their contents will not be needed.

Create a jni directory underneath LLNADemo, and copy Listings 8-4 and 8-5 to the llnademo.c and Android.mk files, respectively, which are stored in this directory. Then replace the contents of AndroidManifest.xml file with Listing 8-6.

Listing 8-6. *A Manifest File for LLNADemo*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="ca.tutortutor.llnademo"
          android:versionCode="1"
          android:versionName="1.0">
  <uses-sdk android:minSdkVersion="9"/>
  <application android:label="@string/app_name" android:hasCode="false">
    <activity android:name="android.app.NativeActivity"
              android:label="@string/app_name"
              android:configChanges="orientation">
      <meta-data android:name="android.app.lib_name"
                 android:value="llnademo"/>
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

An android:configChanges="orientation" attribute has been added to the <activity> tag so that void onConfigurationChanged(ANativeActivity* activity) is invoked when the device orientation changes (from portrait to landscape, for example). As an exercise, remove this attribute and observe how the log messages change.

With the C:\prj\dev\LLNADemo (or your equivalent) directory current, execute a command similar to that shown below to build the library:

```
\android-ndk-r8b\ndk-build
```

If all goes well, you should see the following messages:

```
Compile thumb : llnademo <= llnademo.c
SharedLibrary  : libllnademo.so
Install        : libllnademo.so => libs/armeabi/libllnademo.so
```

You should also observe an armeabi directory in libs, and a libllnademo.so file should be in armeabi.

Now execute the following command to build the project:

```
ant debug
```

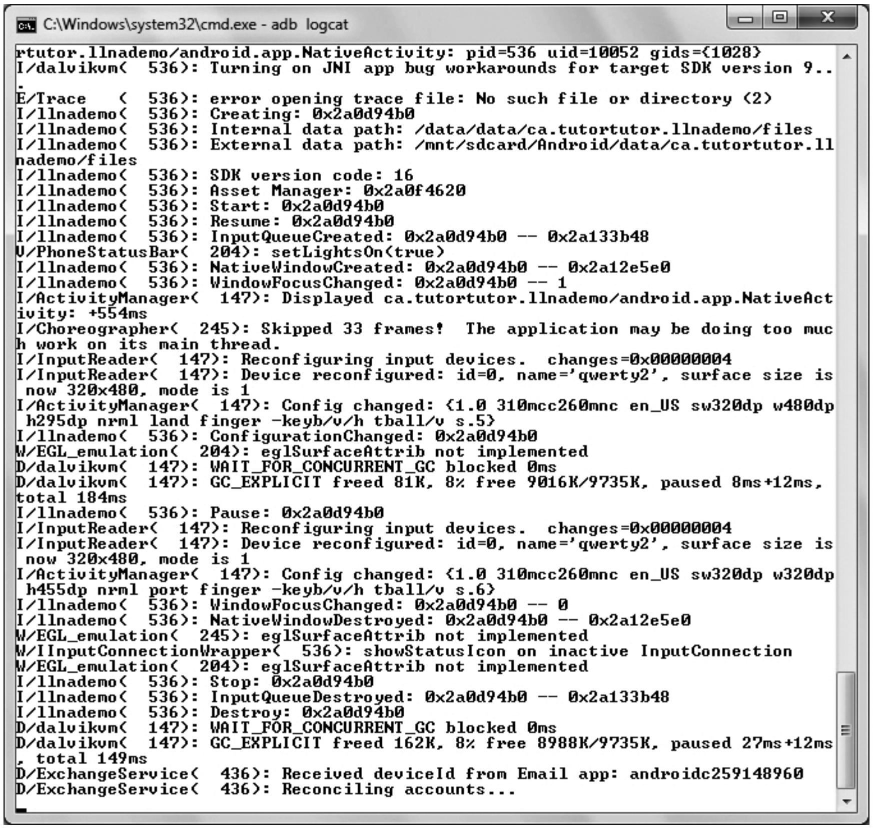
Assuming success, execute the following command to install the `LLNADemo-debug.apk` file on the current device:

```
adb install bin\LLNADemo-debug.apk
```

Before launching this app, start AVD1 (created in Chapter 1). Then execute the following command in a separate window so that you can view the log output:

```
adb logcat
```

Launch `LLNADemo` and you should observe a black screen. Press the Esc key and you should revert to the app launcher. Figure 8-8 shows you a portion of the log related to these events.



```
rtutor.llnademo/android.app.NativeActivity: pid=536 uid=10052 gids={1028}
I/dalvikvm(  536): Turning on JNI app bug workarounds for target SDK version 9..
.
E/Trace   (  536): error opening trace file: No such file or directory (2)
I/llnademo(  536): Creating: 0x2a0d94b0
I/llnademo(  536): Internal data path: /data/data/ca.tutortutor.llnademo/files
I/llnademo(  536): External data path: /mnt/sdcard/Android/data/ca.tutortutor.ll
nademo/files
I/llnademo(  536): SDK version code: 16
I/llnademo(  536): Asset Manager: 0x2a0f4620
I/llnademo(  536): Start: 0x2a0d94b0
I/llnademo(  536): Resume: 0x2a0d94b0
I/llnademo(  536): InputQueueCreated: 0x2a0d94b0 -- 0x2a133b48
V/PhoneStatusBar(  204): setLightsOn(true)
I/llnademo(  536): NativeWindowCreated: 0x2a0d94b0 -- 0x2a12e5e0
I/llnademo(  536): WindowFocusChanged: 0x2a0d94b0 -- 1
I/ActivityManager(  147): Displayed ca.tutortutor.llnademo/android.app.NativeAct
ivity: +554ms
I/Choreographer(  245): Skipped 33 frames!  The application may be doing too muc
h work on its main thread.
I/InputReader(  147): Reconfiguring input devices.  changes=0x00000004
I/InputReader(  147): Device reconfigured: id=0, name='qwerty2', surface size is
 now 320x480, mode is 1
I/ActivityManager(  147): Config changed: {1.0 310mcc260mnc en_US sw320dp w480dp
 h295dp nrml land finger -keyb/v/h tball/v s.5}
I/llnademo(  536): ConfigurationChanged: 0x2a0d94b0
W/EGL_emulation(  204): eglSurfaceAttrib not implemented
D/dalvikvm(  147): WAIT_FOR_CONCURRENT_GC blocked 0ms
D/dalvikvm(  147): GC_EXPLICIT freed 81K, 8% free 9016K/9735K, paused 8ms+12ms,
total 184ms
I/llnademo(  536): Pause: 0x2a0d94b0
I/InputReader(  147): Reconfiguring input devices.  changes=0x00000004
I/InputReader(  147): Device reconfigured: id=0, name='qwerty2', surface size is
 now 320x480, mode is 1
I/ActivityManager(  147): Config changed: {1.0 310mcc260mnc en_US sw320dp w320dp
 h455dp nrml port finger -keyb/v/h tball/v s.6}
I/llnademo(  536): WindowFocusChanged: 0x2a0d94b0 -- 0
I/llnademo(  536): NativeWindowDestroyed: 0x2a0d94b0 -- 0x2a12e5e0
W/EGL_emulation(  245): eglSurfaceAttrib not implemented
W/IInputConnectionWrapper(  536): showStatusIcon on inactive InputConnection
W/EGL_emulation(  204): eglSurfaceAttrib not implemented
I/llnademo(  536): Stop: 0x2a0d94b0
I/llnademo(  536): InputQueueDestroyed: 0x2a0d94b0 -- 0x2a133b48
I/llnademo(  536): Destroy: 0x2a0d94b0
D/dalvikvm(  147): WAIT_FOR_CONCURRENT_GC blocked 0ms
D/dalvikvm(  147): GC_EXPLICIT freed 162K, 8% free 8988K/9735K, paused 27ms+12ms
, total 149ms
D/ExchangeService(  436): Received deviceId from Email app: androidc259148960
D/ExchangeService(  436): Reconciling accounts...
_
```

Figure 8-8. *`LLNADemo` logs various messages during its execution.*

Building and Running LLNADemo with Eclipse

To build `LLNADemo` with Eclipse, first create a new Android project as described in Chapter 1's Recipe 1-10. For your convenience, the steps that you need to follow to accomplish this task are provided here:

1. Start Eclipse if it is not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting *New Project* dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.
4. On the resulting *New Android App* dialog box, enter **`LLNADemo`** into the Application Name textfield. This entered name also appears in the Project Name textfield, and it identifies the folder/directory in which the `LLNADemo` project is stored.
5. Enter **`ca.tutortutor.llnademo`** into the Package Name textfield.
6. Via Build SDK, select the appropriate Android SDK to target. This selection identifies the Android platform you'd like your app to be built against. Assuming that you've installed only the Android 4.1 platform, only this choice should appear and be selected.
7. Via Minimum SDK, either select the minimum Android SDK on which your app runs or keep the default setting. (Don't go lower than API Level 9.)
8. Leave the "Create custom launcher icon" check box checked if you want a custom launcher icon to be created. Otherwise, uncheck this check box when you supply your own launcher icon.
9. Leave the "Mark this project as a library" check box unchecked because you are not creating a library.
10. Leave the "Create Project in Workspace" check box checked, and click Next.

11. On the resulting Configure Launcher Icon pane, make suitable adjustments to the custom launcher icon; click Next.
12. On the resulting Create Activity pane, leave the Create Activity check box checked, make sure that BlankActivity is selected, and click Next.
13. On the resulting New Blank Activity pane, enter **`LLNADemo`** into the Activity Name textfield. Keep all other settings and click Finish.

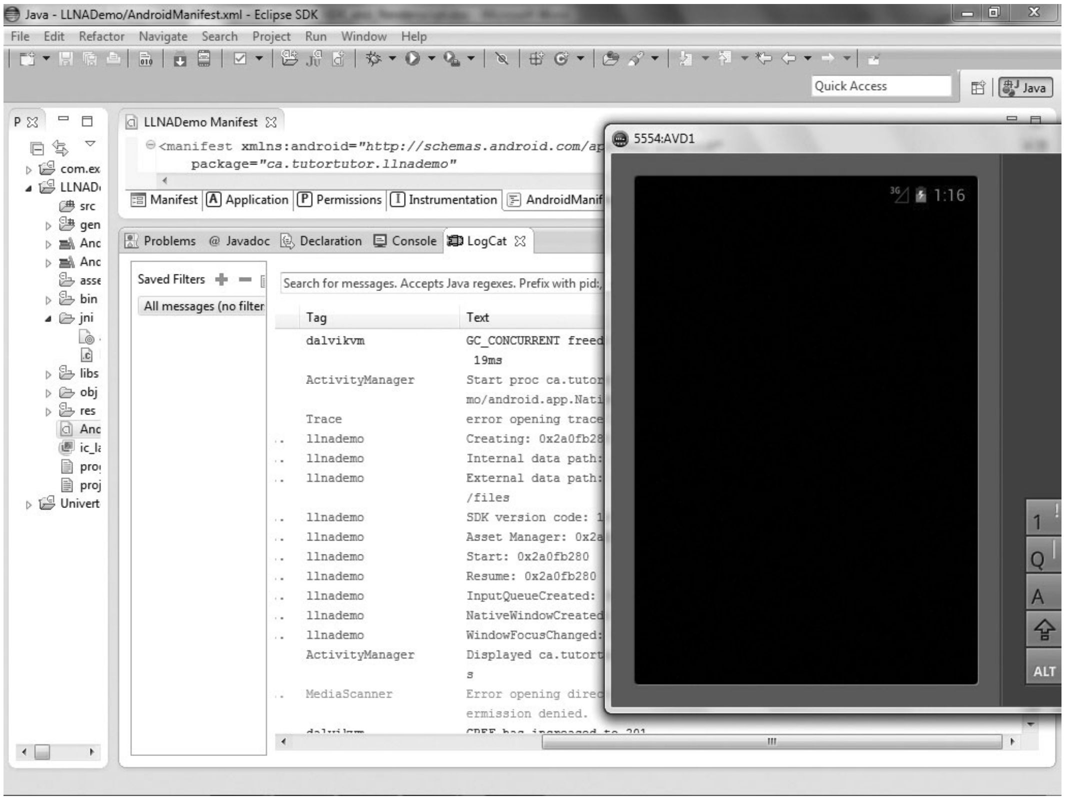
Using Package Explorer, create a jni folder node below the LLNADemo project node, add an `LLNADemo.c` file subnode of jni, replace this node's empty contents with Listing 8-4, add a new `Android.mk` file subnode of jni, and replace its empty contents with Listing 8-5.

Let's create a builder to create the library file. Complete the following steps:

1. Right-click the LLNADemo node, and select Properties from the resulting pop-up menu.
2. Select Builders on the resulting *Properties for LLNADemo* dialog box.
3. On the resulting Builders pane, click the New button.
4. On the resulting *Choose configuration type* dialog box, select Program and click OK.
5. On the resulting *Edit Configuration* dialog box, choose whatever name you want for the builder (or keep the default), enter **`C:\android-ndk-r8b\ndk-build.cmd`** (or your equivalent) into the Location textfield, enter **`${workspace_loc:/LLNADemo}`** into the Working Directory textfield, and click the OK button to close this dialog box.
6. Click the OK button to close the *Properties for LLNADemo* dialog box.

Finally, using Eclipse's built-in manifest editor, make the necessary changes to `AndroidManifest.xml` that were presented earlier in this recipe.

To run `LLNADemo` from Eclipse, select Run from the menubar, and then select Run from the drop-down menu. If a *Run As* dialog box appears, select Android Application and click OK. Eclipse launches `emulator` with the AVD1 device, installs `LLNADemo.apk`, and runs this app, whose output appears in Figure 8-9.



```
<manifest xmlns:android="http://schemas.android.com/ap
    package="ca.tutortutor.llnademo"
```

```
   Tag                  Text
   dalvikvm             GC_CONCURRENT freed
                         19ms
   ActivityManager      Start proc ca.tutor
                        mo/android.app.Nati
   Trace                error opening trace
.. llnademo             Creating: 0x2a0fb28
.. llnademo             Internal data path:
.. llnademo             External data path:
                        /files
.. llnademo             SDK version code: 1
.. llnademo             Asset Manager: 0x2a
.. llnademo             Start: 0x2a0fb280
.. llnademo             Resume: 0x2a0fb280
.. llnademo             InputQueueCreated:
.. llnademo             NativeWindowCreated
.. llnademo             WindowFocusChanged:
   ActivityManager      Displayed ca.tutort
                        s
.. MediaScanner         Error opening direc
                        ermission denied.
```

Figure 8-9. *`LLNADemo` logs various messages during its execution.*

NOTE: When the target SDK is set to API Level 13 or higher (Eclipse defaults the target SDK to 15) and you haven't included `screenSize` with `orientation` in the value assigned to `<activity>`'s `configChanges` attribute (`"orientation|screenSize"`), you will not see "ConfigurationChanged" messages in the log when you change the device orientation.

8-3. Developing High-Level Native Activities

Problem

You want to learn how to develop high-level native activities, which are based on the android_native_app_glue.h header file.

Solution

The development of a high-level native activity is very similar to that of a low-level native activity. However, a new source file and a new Android.mk file are required.

How It Works

Consider an HLNADemo project that demonstrates high-level native activities. Listing 8-7 presents the contents of this project's solitary hlnademo.c source file.

Listing 8-7. *Examining a Native Activity from a High Perspective*

```
#include <android/log.h>
#include <android_native_app_glue.h>

#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO, \
                                             "hlnademo", \
                                             __VA_ARGS__))

static void handle_cmd(struct android_app* app, int32_t cmd)
{
   switch (cmd)
   {
      case APP_CMD_SAVE_STATE:
           LOGI("Save state");
           break;

      case APP_CMD_INIT_WINDOW:
           LOGI("Init window");
           break;

      case APP_CMD_TERM_WINDOW:
           LOGI("Terminate window");
           break;

      case APP_CMD_PAUSE:
           LOGI("Pausing");
           break;
      case APP_CMD_RESUME:
           LOGI("Resuming");
           break;

      case APP_CMD_STOP:
           LOGI("Stopping");
           break;

      case APP_CMD_DESTROY:
           LOGI("Destroying");
           break;

      case APP_CMD_LOST_FOCUS:
           LOGI("Lost focus");
           break;

      case APP_CMD_GAINED_FOCUS:
           LOGI("Gained focus");
   }
}

static int32_t handle_input(struct android_app* app, AInputEvent* event)
{
   if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION)
   {
      size_t pointerCount = AMotionEvent_getPointerCount(event);
      size_t i;
      for (i = 0; i < pointerCount; ++i)
      {
         LOGI("Received motion event from %zu: (%.2f, %.2f)", i,
              AMotionEvent_getX(event, i), AMotionEvent_getY(event, i));
      }
      return 1;
   }
   else if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_KEY)
   {
      LOGI("Received key event: %d", AKeyEvent_getKeyCode(event));
      if (AKeyEvent_getKeyCode(event) == AKEYCODE_BACK)
         ANativeActivity_finish(app->activity);
      return 1;
   }
   return 0;
}

void android_main(struct android_app* state)
{
   app_dummy(); // prevent glue from being stripped

   state->onAppCmd = &handle_cmd;
   state->onInputEvent = &handle_input;

   while(1)
   {
      int ident;
      int fdesc;
      int events;
      struct android_poll_source* source;

      while ((ident = ALooper_pollAll(0, &fdesc, &events, (void**)&source)) >=
0)
      {
         if (source)
            source->process(state, source);

         if (state->destroyRequested)
            return;
      }
   }
}
```

Listing 8-7 begins in a nearly identical fashion to Listing 8-4. However, the previous `native_activity.h` header file has been replaced by `android_native_app_glue.h`, which includes `native_activity.h` (along with `pthread.h`). A similar `LOGI` macro is also provided.

The `void handle_cmd(struct android_app* app, int32_t cmd)` function is called (on a thread other than the main thread) in response to an activity command. The `app` parameter references an `android_app struct` (defined in `android_native_app_glue.h`) that provides access to app-related data, and the `cmd` parameter identifies a command.

NOTE: Commands are integer values that correspond to the low-level native activity functions that were presented earlier (`void onDestroy(ANativeActivity* activity)`, for example). The `android_native_app_glue.h` header file defines integer constants for these commands (`APP_CMD_DESTROY`, for example).

The `int32_t handle_input(struct android_app* app, AInputEvent* event)` function is called (on a thread other than the main thread) in response to an input event. The `event` parameter references an `AInputEvent` struct (defined in `input.h`) that provides access to various kinds of event-related information.

The `input.h` header file declares several useful input functions, beginning with `AInputEvent_getType(const AInputEvent* event)`, which returns the type of the event. The return value is one of AINPUT_EVENT_TYPE_KEY for a key event and AINPUT_EVENT_TYPE_MOTION for a motion event.

For a motion event, the size_t AMotionEvent_getPointerCount(const AInputEvent* motion_event) function is called to return the number of *pointers* (active touch points) of data contained in this event (this value is greater than or equal to 1). This count is repeated, with each touch point's coordinates being obtained and logged.

NOTE: Active touch points and AMotionEvent_getPointerCount(const AInputEvent*) are related to *multitouch*. To learn more about this Android feature, check out "Making Sense of Multitouch" (http://android-developers.blogspot.ca/2010/06/making-sense-of-multitouch.html).

For a key event, the int32_t AKeyEvent_getKeyCode(const AInputEvent* key_event) function returns the code of the physical key that was pressed. Physical key codes are defined in the keycodes.h header file. For example, AKEYCODE_BACK corresponds to the back button on the device.

The key code is logged and is then compared with AKEYCODE_BACK to find out if the user wants to terminate the activity (and, by extension, the single-activity app). If so, the void ANativeActivity_finish(ANativeActivity* activity) function (defined in native_activity.h) is invoked with app->activity referencing the activity to be finished.

After processing a mouse or key event, handle_input(struct android_app*, AInputEvent*) returns 1 to indicate that it has handled the event. If the event was not handled (and should be handled by default processing in the background), this function returns 0.

NOTE: You can comment out handle_input(struct android_app*, AInputEvent*)'s if (AKeyEvent_getKeyCode(event) == AKEYCODE_BACK), followed by ANativeActivity_finish(app->activity);, followed by return 1; statements, and let return 0; cause default processing to finish the activity when the back button is pressed.

The void android_main(struct android_app* state) function is the entry point. It first invokes a native glue function called app_dummy(), which doesn't do anything. However, app_dummy() must be present to ensure that the Android build system includes the android_native_app_glue.o module in the library.

NOTE: See http://blog.beuc.net/posts/Make_sure_glue_isn__39__t_stripped to learn more about this oddity.

The android_app struct provides an onAppCmd field of type void (*onAppCmd)(struct android_app* app, int32_t cmd) and an onInputEvent field of type int32_t (*onInputEvent)(struct android_app* app, AInputEvent* event). The addresses of the aforementioned functions are assigned to these fields.

A pair of nested loops is now entered. The inner loop repeatedly invokes the int ALooper_pollAll(int timeoutMillis, int* outFd, int* outEvents, void** outData) function (defined in looper.h) to return the next event; this function returns a value greater than or equal to 0 when an event is ready for processing.

The event is recorded in an android_poll_source structure, whose address is stored in outData. Assuming that outData contains a non-NULL address, android_poll_source's void (*process)(struct android_app* app, struct android_poll_source* source) function is invoked to process the event. Behind the scenes, either handle_cmd(struct android_app*, int32_t) or handle_input(struct android_app*, AInputEvent*) is invoked; it depends on which function is appropriate for handling the event.

Finally, the destroyRequested member of the android_app structure is set to a nonzero value, as a result of a call to ANativeActivity_finish(ANativeActivity*) (or default processing in lieu of this function). This member is checked during each loop iteration to ensure that execution exits quickly from the nested loops and android_main(struct android_app*), because the app is ending. Failure to exit android_main(struct android_app*) in a timely fashion can result in an "Application Not Responding" error.

Listing 8-8 presents this project's Android.mk file.

Listing 8-8. *A Make File for HLNADemo*

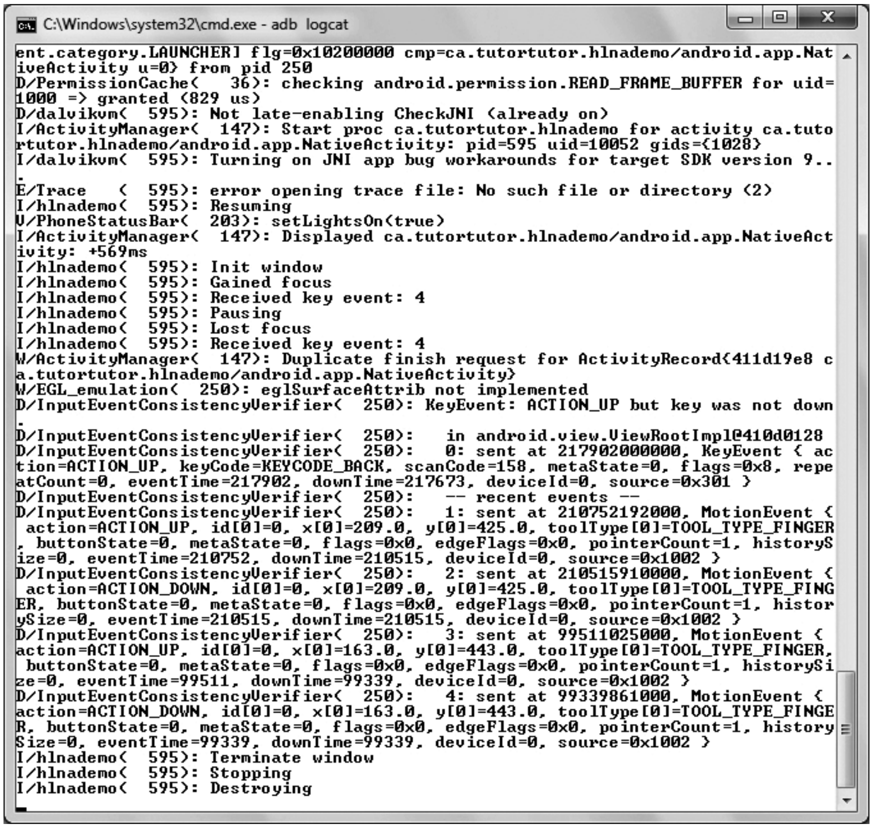
```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := hlnademo
LOCAL_SRC_FILES := hlnademo.c
LOCAL_LDLIBS := -landroid
LOCAL_STATIC_LIBRARIES := android_native_app_glue
include $(BUILD_SHARED_LIBRARY)
$(call import-module,android/native_app_glue)
```

This make file is similar to the make file presented in Listing 8-5. However, there are some differences:

- The `LOCAL_LDLIBS` entry no longer contains `-llog` because the logging library is linked to the `android_native_app_glue` library when this library is built.
- A `LOCAL_STATIC_LIBRARIES` entry identifies `android_native_app_glue` as a library to be linked to the `hlnademo` module.
- A `$(call import-module,android/native_app_glue)` entry includes the `Android.mk` file associated with the `android_native_app_glue` module so that this library can be built.

Building and Running HLNADemo with the Android SDK

Build `HLNADemo` as if you were building `LLNADemo` (change each `llnademo` reference to `hlnademo` in the manifest and use the updated `Android.mk` file presented in Listing 8-8). Then launch `HLNADemo` and you should observe a black screen. Figure 8-10 shows you a portion of the log that reveals messages presented in Listing 8-7.



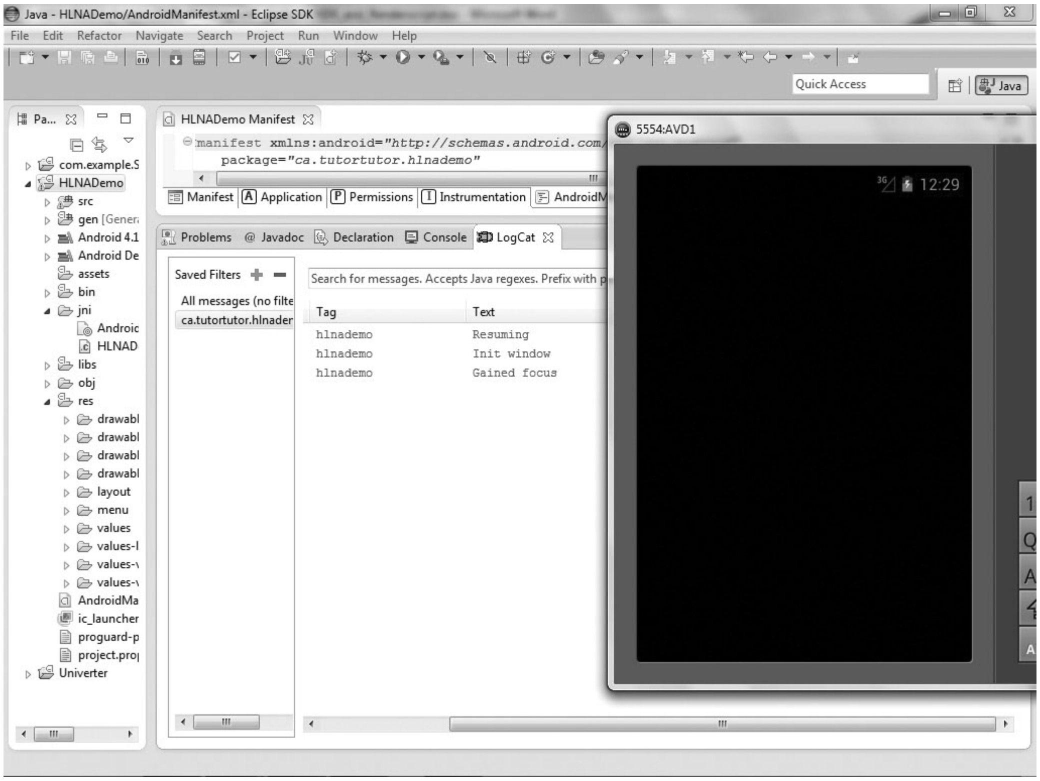
C:\Windows\system32\cmd.exe - adb logcat

```
ent.category.LAUNCHER] flg=0x10200000 cmp=ca.tutortutor.hlnademo/android.app.Nat
iveActivity u=0} from pid 250
D/PermissionCache(   36): checking android.permission.READ_FRAME_BUFFER for uid=
1000 => granted (829 us)
D/dalvikvm(  595): Not late-enabling CheckJNI (already on)
I/ActivityManager(  147): Start proc ca.tutortutor.hlnademo for activity ca.tuto
rtutor.hlnademo/android.app.NativeActivity: pid=595 uid=10052 gids={1028}
I/dalvikvm(  595): Turning on JNI app bug workarounds for target SDK version 9..
.
E/Trace   (  595): error opening trace file: No such file or directory (2)
I/hlnademo(  595): Resuming
V/PhoneStatusBar(  203): setLightsOn(true)
I/ActivityManager(  147): Displayed ca.tutortutor.hlnademo/android.app.NativeAct
ivity: +569ms
I/hlnademo(  595): Init window
I/hlnademo(  595): Gained focus
I/hlnademo(  595): Received key event: 4
I/hlnademo(  595): Pausing
I/hlnademo(  595): Lost focus
I/hlnademo(  595): Received key event: 4
W/ActivityManager(  147): Duplicate finish request for ActivityRecord{411d19e8 c
a.tutortutor.hlnademo/android.app.NativeActivity}
W/EGL_emulation(  250): eglSurfaceAttrib not implemented
D/InputEventConsistencyVerifier(  250): KeyEvent: ACTION_UP but key was not down
.
D/InputEventConsistencyVerifier(  250):   in android.view.ViewRootImpl@410d0128
D/InputEventConsistencyVerifier(  250):   0: sent at 217902000000, KeyEvent { ac
tion=ACTION_UP, keyCode=KEYCODE_BACK, scanCode=158, metaState=0, flags=0x8, repe
atCount=0, eventTime=217902, downTime=217673, deviceId=0, source=0x301 }
D/InputEventConsistencyVerifier(  250):   -- recent events --
D/InputEventConsistencyVerifier(  250):   1: sent at 210752192000, MotionEvent {
 action=ACTION_UP, id[0]=0, x[0]=209.0, y[0]=425.0, toolType[0]=TOOL_TYPE_FINGER
, buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0, pointerCount=1, historyS
ize=0, eventTime=210752, downTime=210515, deviceId=0, source=0x1002 }
D/InputEventConsistencyVerifier(  250):   2: sent at 210515910000, MotionEvent {
 action=ACTION_DOWN, id[0]=0, x[0]=209.0, y[0]=425.0, toolType[0]=TOOL_TYPE_FING
ER, buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0, pointerCount=1, histor
ySize=0, eventTime=210515, downTime=210515, deviceId=0, source=0x1002 }
D/InputEventConsistencyVerifier(  250):   3: sent at 99511025000, MotionEvent {
action=ACTION_UP, id[0]=0, x[0]=163.0, y[0]=443.0, toolType[0]=TOOL_TYPE_FINGER,
 buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0, pointerCount=1, historySi
ze=0, eventTime=99511, downTime=99339, deviceId=0, source=0x1002 }
D/InputEventConsistencyVerifier(  250):   4: sent at 99339861000, MotionEvent {
action=ACTION_DOWN, id[0]=0, x[0]=163.0, y[0]=443.0, toolType[0]=TOOL_TYPE_FINGE
R, buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0, pointerCount=1, history
Size=0, eventTime=99339, downTime=99339, deviceId=0, source=0x1002 }
I/hlnademo(  595): Terminate window
I/hlnademo(  595): Stopping
I/hlnademo(  595): Destroying
```

Figure 8-10. *`HLNADemo` logs various messages during its execution.*

Building and Running HLNADemo with Eclipse

Build `HLNADemo` as if you were building `LLNADemo` (change each `llnademo` reference to `hlnademo` in the manifest and use the updated `Android.mk` file presented in Listing 8-8). Then launch `HLNADemo` and you should observe a black screen. Figure 8-11 shows you a portion of the log that reveals messages presented in Listing 8-7.



```
manifest xmlns:android="http://schemas.android.com/
    package="ca.tutortutor.hlnademo"
```

```
Tag         Text
hlnademo    Resuming
hlnademo    Init window
hlnademo    Gained focus
```

Figure 8-11. `HLNADemo` *logs various messages during its execution.*

Renderscript

You can use the Android NDK to perform rendering and data-processing operations quickly. However, there are three major problems with this approach:

- *Lack of portability*: Your apps are constrained to run on only those devices to which the native code targets. For example, a native library that runs on an ARM-based device won't run on an x86-based device.
- *Lack of performance*: Ideally, your code should run on multiple cores, be they CPU, GPU, or DSP cores. However, identifying cores, farming out work to them, and dealing with synchronization issues isn't easy.

- *Lack of usability*: Developing native code is harder than developing Java code. For example, you often need to create JNI glue code, which is a tedious process that can be a source of bugs.

Google's Android development team created Renderscript to address these problems, starting with lack of portability, then lack of performance, and finally lack of usability.

Renderscript consists of a language based on C99 (a modern dialect of the C language), a pair of compilers, and a runtime that collectively help you achieve high performance and visually compelling graphics via native code but in a portable manner. You get native app speed along with SDK app portability, and you don't have to use the JNI.

NOTE: Although it has been present since Android 2.0, Renderscript was not made public until Android 3.0, where it is used to implement live wallpapers and more.

Renderscript combines a graphics engine with a compute engine. The graphics engine helps you achieve fast 2D/3D rendering, and the compute engine helps you achieve fast data processing. Performance is achieved by running threads on multiple CPU, GPU, and DSP cores. (The compute engine is currently confined to CPU cores.)

TIP: The compute engine is not limited to processing graphics data. For example, it could be used to model weather data.

Exploring Renderscript Architecture

Renderscript adopts an architecture in which the low-level Renderscript runtime is controlled by the higher-level Android framework. Figure 8-12 presents this architecture.

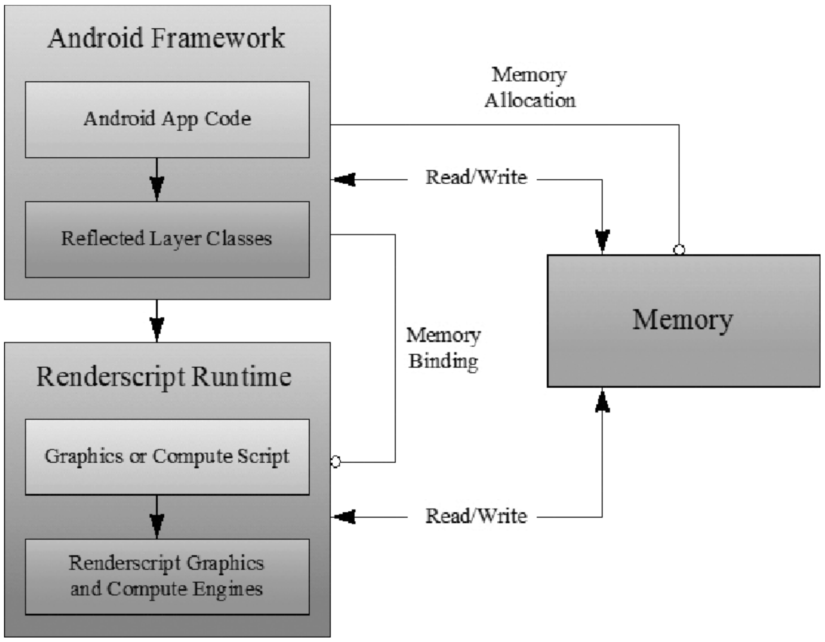


Figure 8-12. *Renderscript architecture is based on the Android framework and the Renderscript runtime.*

The Android framework consists of Android apps running in the Dalvik VM that communicate with graphics or compute scripts running in the Renderscript runtime via instances of reflected layer classes. These classes serve as wrappers around their scripts that make this communication possible. The Android build tools automatically generate the classes for this layer during the build process. These classes eliminate the need to write JNI glue code, which is commonly done when working with the NDK.

Memory management is controlled at the VM level. The app is responsible for allocating memory and binding this memory to the Renderscript runtime so that the memory can be accessed by the script. (The script can define simple [nonarray] fields for its own use, but that's about it.)

Apps make asynchronous calls to the Renderscript runtime (via the reflected layer classes) to make allocated memory available to and start executing their scripts. They can subsequently obtain results from these scripts without having to worry about whether or not the scripts are still running.

When you build an APK, the LLVM (Low-Level Virtual Machine) front-end compiler (see the `llvm-rs-cc` tool in Appendix B) compiles the script into a file of device-independent bitcode that is stored in the APK. (The reflected layer class is also created.) When the app launches, a small LLVM back-end compiler on the device compiles the bitcode into device-specific code, and it caches the code on the device so that it doesn't have to be recompiled each time you run the app. This is how portability is achieved.

> **NOTE:** As of Android 4.1, the graphics engine has been deprecated. App developers told the Android development team that they prefer to use OpenGL directly because of its familiarity. Although the graphics engine is still supported, it will probably be removed in a future Android release. For this reason, the rest of this chapter focuses only on the compute engine.

Exploring Compute Engine-Based App Architecture

A compute engine-based app consists of Java code and an `.rs` file that defines the compute script. The Java code interacts with this script by using APIs defined in the `android.renderscript` package. Key classes in this package are `RenderScript` and `Allocation`:

- `RenderScript` defines a context that is used in further interactions with Renderscript APIs (and also the compute script's reflected layer class). A `RenderScript` instance is returned by invoking this class's `static RenderScript create(Context ctx)` factory method.
- `Allocation` defines the means for moving data into and out of the compute script. Instances of this class are known as *allocations*, where an allocation combines an `android.renderscript.Type` instance with the memory needed to provide storage for user data and objects.

The Java code also interacts with the compute script by instantiating a reflected layer class. The name of the class begins with `ScriptC_` and continues with the name of the `.rs` file containing the compute script. For example, if you had a file named `gray.rs`, the name of this class would be `ScriptC_gray`.

The C99-based `.rs` file begins with two `#pragma`s that identify the Renderscript version number (currently 1) and the app's Java package name. Several additional items follow:

- `rs_allocation` directives that identify the input and output allocations created by the app and bound to the Renderscript code
- an `rs_script` directive that provides a link to the app's `ScriptC_`*`script`* instance so that compute results can be returned to this instance
- optional simple variable declarations whose values are supplied by the app
- a `root()` function that is called by each core to perform part of the overall computation
- a `noargument init` function with a `void` return type that's indirectly invoked from the Java code to execute `root()` on multiple CPU cores

At runtime, a Java-based activity creates a `Renderscript` context, creates input and output allocations, instantiates the `ScriptC_`-prefixed layer class, uses this object to bind the allocations and `ScriptC_` instance, and invokes the compute script, which results in the script's `init` function being invoked.

The `init` function performs additional initialization (as necessary) and executes the `rsForEach()` function with the `rs_script` value and the `rs_allocation` input/output allocations. `rsForEach()` causes the `root()` function to be executed on the device's available CPU cores. Results are then sent back to the app via the output allocation.

Figure 8-13 illustrates this scenario.

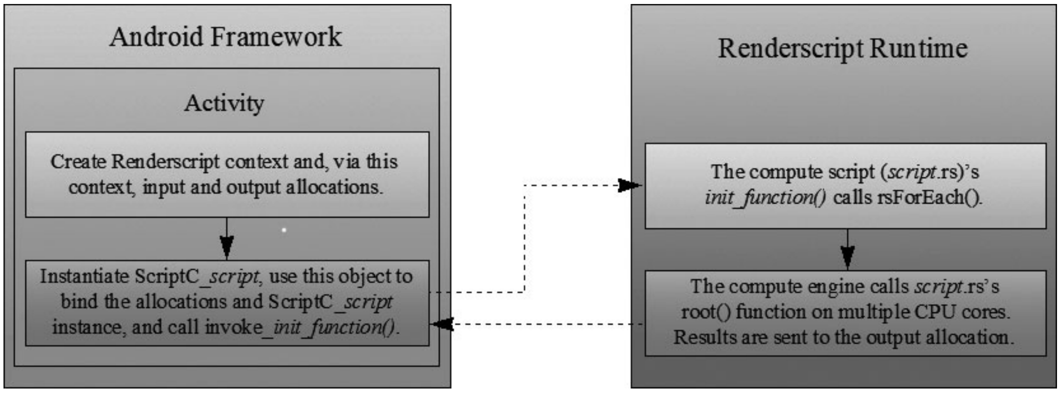

Android Framework
Activity
Create Renderscript context and, via this context, input and output allocations.
Instantiate ScriptC_script, use this object to bind the allocations and ScriptC_script instance, and call invoke_init_function().
Renderscript Runtime
The compute script (script.rs)'s init_function() calls rsForEach().
The compute engine calls script.rs's root() function on multiple CPU cores. Results are sent to the output allocation.

Figure 8-13. *Compute engine–based app architecture can be partitioned into four major tasks.*

Grayscaling Images with Renderscript

Perhaps the easiest way to become familiar with the compute side of Renderscript is to create a small app that performs a simple image-processing operation, such as grayscaling an image. Listing 8-9 presents the source code to a `GrayScale` app that lets you view an image of the Sun, grayscale the image, and view the result.

Listing 8-9. *Viewing Original and Grayscaled Images of the Sun*

```
package ca.tutortutor.grayscale;

import android.app.Activity;

import android.os.Bundle;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;

import android.renderscript.Allocation;
import android.renderscript.RenderScript;

import android.view.View;

import android.widget.ImageView;

public class GrayScale extends Activity
{
   boolean original = true;

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      final ImageView iv = new ImageView(this);
      iv.setScaleType(ImageView.ScaleType.CENTER_CROP);
      iv.setImageResource(R.drawable.sol);
      setContentView(iv);
      iv.setOnClickListener(new View.OnClickListener()
                            {
                               @Override
                               public void onClick(View v)
                               {
                                  if (original)
                                     drawGS(iv, R.drawable.sol);
                                  else
                                     iv.setImageResource(R.drawable.sol);
                                  original = !original;
                               }
```

```
                             });
   }

   private void drawGS(ImageView iv, int imID)
   {
     Bitmap bmIn = BitmapFactory.decodeResource(getResources(), imID);
     Bitmap bmOut = Bitmap.createBitmap(bmIn.getWidth(), bmIn.getHeight(),
                                        bmIn.getConfig());
     RenderScript rs = RenderScript.create(this);
     Allocation allocIn;
     allocIn = Allocation.createFromBitmap(rs, bmIn,
                                           Allocation.MipmapControl.MIPMAP_NONE,
                                           Allocation.USAGE_SCRIPT);
     Allocation allocOut = Allocation.createTyped(rs, allocIn.getType());
     ScriptC_grayscale script = new ScriptC_grayscale(rs, getResources(),
                                                      R.raw.grayscale);
     script.set_in(allocIn);
     script.set_out(allocOut);
     script.set_script(script);
     script.invoke_filter();
     allocOut.copyTo(bmOut);
     iv.setImageBitmap(bmOut);
   }
}
```

Listing 8-9 declares an activity class named GrayScale. This class overrides the onCreate(Bundle) method, declares a void drawGS(ImageView iv, int imID) method that grayscales an image identified by resource ID imID, and then assigns the result to the android.widget.ImageView instance identified by iv.

onCreate(Bundle) creates the activity's user interface based on an ImageView instance whose contents are set to the drawable resource identified by R.drawable.sol. The image is scaled uniformly (maintaining the image's aspect ratio) so that each dimension is at least as large as the corresponding screen dimension (less any padding).

A click listener is registered with the imageview widget. An initial click on this widget causes the listener to invoke drawGS(ImageView, int)and to grayscale the image and update the widget with the grayscaled result. A second click causes the original image to be displayed. Subsequent clicks continue this alternating pattern of behavior.

drawGS(ImageView, int) first invokes the android.graphics.BitmapFactory class's static Bitmap decodeResource(Resources res, int id) method. When passed an android.content.res.Resources instance (obtained via android.content.Context's Resources getResources() method) and the resource ID of the desired drawable resource (R.drawable.sol), this method returns an `android.graphics.Bitmap` instance containing the contents of this resource.

`drawGS(ImageView, int)` next invokes `BitMap`'s `static Bitmap createBitmap(int width, int height, Bitmap.Config config)` method to create and return an empty bitmap with the same dimensions and configuration as the previous bitmap (which contains the contents of the Sun image).

A `RenderScript` context object is created. This object is then passed as the first argument to `Allocation`'s `static Allocation createFromBitmap(RenderScript rs, Bitmap b, Allocation.MipmapControl mips, int usage)` method, which creates an allocation that stores the drawable's bitmap. Three additional arguments are passed:

- `bmIn` identifies the bitmap source of the allocation.
- `Allocation.MipmapControl.MIPMAP_NONE` specifies that no *mipmaps* (precalculated, optimized collections of images that accompany a main texture, which are intended to increase rendering speed and reduce aliasing artifacts) will be generated, and the type generated from the incoming bitmap will not contain additional levels of detail.
- `Allocation.USAGE_SCRIPT` specifies that the allocation is to be bound to and accessed by the compute script.

The created `Allocation` object serves as the input allocation, from where the compute script obtains its input data for processing. A second `Allocation` object for storing computed output is now created, by invoking `Allocation`'s `static Allocation createTyped(RenderScript rs, Type type)` method with the following arguments:

- `rs` identifies the Renderscript context.
- `allocIn.getType()` returns the type of the input allocation. The type describes the layout of the input allocation's bitmap.

At this point, the reflected `ScriptC_grayscale` class, which provides communication between the app and the compute script, is instantiated with the following arguments:

- `rs` identifies the Renderscript context.
- `getResources()` returns a `Resources` instance for accessing resources.
- `R.raw.grayscale` identifies a `grayscale.bc` bitcode resource file that is stored in the APK's `res/raw` directory.

As you will soon discover, the compute script contains `in`, `out`, and `script` fields. These fields are accessible to the app (for initialization) via `ScriptC_grayscale`'s `set_in()`, `set_out()`, and `set_script()` methods. These methods are called to communicate the input/output allocations and the `ScriptC_grayscale` instance to the compute script.

Along with `set_in()`, `set_out()`, and `set_script()` methods, the LLVM front-end compiler creates an `invoke_filter()` method that is now called to execute the compute script. (The `filter` portion of this name matches the `noargument init` function in the compute script.)

One of the nice things about Renderscript is that the app can immediately request script output without having to wait for the script to finish. In this case, the app invokes `Allocation`'s `void copyTo(Bitmap b)` method to copy results from the output allocation to the `Bitmap` instance passed to this method, which happens to be the empty bitmap.

Finally, the formerly empty bitmap is assigned to the imageview widget via `ImageView`'s `void setImageBitmap(Bitmap bm)` method, and the grayscaled result is seen.

Now that you've explored the Java side of this app, Listing 8-10 introduces you to the C99 compute script side.

Listing 8-10. *Grayscaling an Image*

```
#pragma version(1)
#pragma rs java_package_name(ca.tutortutor.grayscale)

rs_allocation in;
rs_allocation out;
rs_script script;

const static float3 gsVector = {0.3f, 0.6f, 0.1f};

void root(const uchar4* v_in, uchar4* v_out)
{
   float4 f4 = rsUnpackColor8888(*v_in);
   *v_out = rsPackColorTo8888((float3) dot(f4.rgb, gsVector));
}

void filter()
{
   rsDebug("RS_VERSION = ", RS_VERSION);
#if !defined(RS_VERSION) || (RS_VERSION < 14)
   rsForEach(script, in, out, 0);
#else
   rsForEach(script, in, out);
#endif
}
```

Listing 8-10 first presents two #pragmas that respectively identify the Renderscript version number and ca.tutortutor.grayscale as the Java package with which this compute script associates.

A pair of rs_allocation directives and an rs_script directive follow. These directives introduce variables that will reference the input/output allocations and the ScriptC_grayscale instance.

Next, gsVector, a vector of three floating-point values (indicated by the float3 type), is defined. gsVector is initialized to 0.3f (red), 0.6f (green), and 0.1f (blue), which indicate the percentages of pixel color components that contribute to the overall gray value.

The subsequent void root(const uchar4* v_in, uchar4* v_out) function is invoked for each pixel and on each available CPU core. Each pixel is processed independently of other pixels.

The v_in parameter is a pointer to a uchar4 structure that holds the four eight-bit values (red, green, blue, and alpha color components) of the incoming pixel. The v_out parameter is a pointer to a similar structure for storing the pixel's resulting components.

root(const uchar4*, uchar4*) first unpacks *v_in to a float4 value (float4 identifies a vector of four floating-point values). This task is accomplished by invoking Renderscript's float4 rsUnpackColor8888(uchar4 color) function.

The RGB components of this vector are accessed by specifying f4.rgb and are then multiplied by gsVector's three components, by invoking Renderscript's float dot(float lhs, float rhs) (dot product) function.

NOTE: dot() follows a pattern in which it accepts 1, 2, 3, or 4 components as arguments. This pattern lets you specify scalar values (such as float result = dot(1.0f, 2.0f);) or vectors with no more than four components (such as float3 result = dot(f4.rgb, gsVector);).

The final task is to pack dot()'s return value into a uchar4 value by invoking Renderscript's uchar4 rsPackColorTo8888(float3 color) function; then assign this value to *v_out. The (float3) cast is redundant but necessary.

Finally, the `void filter()` function is invoked as a result of the app executing `script.invoke_filter();`. (Note the pattern in which the name of this function is appended to `invoke_`.)

`filter()` first executes `rsDebug("RS_VERSION = ", RS_VERSION);` to output the value of the `RS_VERSION` constant to the log. You can invoke one of Renderscript's overloaded `rsDebug()` functions to output debugging information.

`RS_VERSION` is a special constant that is set to the SDK version number. `filter()` contains `#if` and `#else` directives that help the compiler choose a different version of `rsForEach()` to call based on this constant's existence and value.

Assuming that `RS_VERSION` exists and has a value less than 14, the simplest variant of `rsForEach()` that can be called is `void rsForEach(rs_script script, rs_allocation input, rs_allocation output, const void* usrData)`.

> **NOTE:** `usrData` lets you pass a pointer to additional script-specific data to the `root()` function. You will see how to obtain this pointer in a `root()` context in Recipe 8-4.

If `RS_VERSION` contains a value that is 14 or higher, the simplest variant of the `rsForEach()` function that can be called is `void rsForEach(rs_script script, rs_allocation input, rs_allocation output)`.

> **NOTE:** You will encounter `rsForEach()` call examples on the Internet that do not consult `RS_VERSION`. However, not testing this constant via `#if` and `#else` means that you can run into a situation where the script compiles okay under the Android SDK or Eclipse but then fails to compile on the other development platform, with output messages similar to the following:
>
> ```
> note: candidate function not viable: requires 4 arguments, but
> 3 were provided
> ```
>
> ```
> note: candidate function not viable: requires 5 arguments, but
> 3 were provided
> ```

Regardless of the `rsForEach()` function that is called, its first argument is a reference to the script object on the Java side; its second argument, `in`, corresponds to `v_in`; and its third argument, `out`, corresponds to `v_out`.

Building and Running GrayScale with the Android SDK

To build GrayScale with the Android SDK, first use the SDK's android tool to create a GrayScale project. Assuming a Windows platform, a C:\prj\dev hierarchy in which the GrayScale project is to be stored (in C:\prj\dev\GrayScale), and an Android 4.1 platform target that corresponds to integer ID 1 (execute android list targets to obtain the correct ID), execute the following command (spread across two lines for readability) to create GrayScale:

```
android create project -t 1 -p C:\prj\dev\GrayScale -a GrayScale
                       -k ca.tutortutor.grayscale
```

Replace the contents of the src\ca\tutortutor\grayscale\GrayScale.java file with Listing 8-9. Also, create a grayscale.rs file with Listing 8-10's contents, and store this file in the src directory. (Renderscript source files are given the .rs extension and are stored in a project's src directory.) Finally, create a drawable-nodpi directory and copy a file named sol.jpg (presumably containing an image of the Sun) to this directory. (Android does not scale images stored in drawable-nodpi; the app takes care of the scaling.)

Execute the following command to build the project:

```
ant debug
```

You will probably discover the following warning message (spread across multiple lines for readability) and a failed build:

```
WARNING: RenderScript include directory
         'C:\prj\dev\GrayScale\${android.renderscript.include.path}'
         does not exist!
[llvm-rs-cc.exe] <built-in>:2:10: fatal: 'rs_core.rsh' file not found
```

Issue 34569 in Google's Android issues database (http://code.google.com/p/android/issues/detail?id=34569) offers a workaround: Add the following property (spread across multiple lines for readability) to the build.xml file that's located in the tools\ant subdirectory of your Android SDK home directory:

```
<property name="android.renderscript.include.path"
          location="${android.platform.tools.dir}/renderscript/include:
                    ${android.platform.tools.dir}/renderscript/clang-include"/>
```

Place this <property> element after the following <path> element:

```
<!-- Renderscript include Path -->
<path id="android.renderscript.include.path">
  <pathelement location="${android.platform.tools.dir}/renderscript/include" />
  <pathelement location="${android.platform.tools.dir}/renderscript/clang-
```

```
include" />
</path>
```

Reexecute `ant debug` and the build should succeed. Finally, install the `grayscale-debug.apk` file on AVD1 (see Chapter 1) and run the app. Figure 8-14 shows the result.

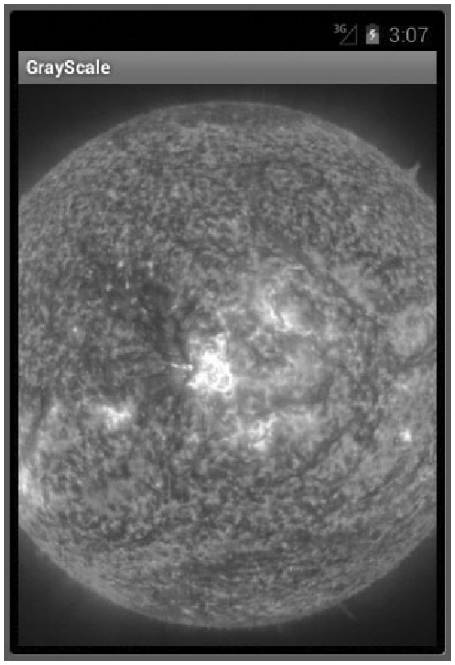


Figure 8-14. *Click this orange-colored Sun image to see the Sun in grayscale.*

Building and Running GrayScale with Eclipse

To build `GrayScale` with Eclipse, first create a new Android project as described in Chapter 1's Recipe 1-10. For your convenience, the steps that you need to follow to accomplish this task are provided here:

1. Start Eclipse if it is not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.

3. On the resulting *New Project* dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.
4. On the resulting *New Android App* dialog box, enter **`GrayScale`** into the Application Name textfield. This entered name also appears in the Project Name textfield, and it identifies the folder/directory in which the `GrayScale` project is stored.
5. Enter **`ca.tutortutor.grayscale`** into the Package Name textfield.
6. Via Build SDK, select the appropriate Android SDK to target. This selection identifies the Android platform you'd like your app to be built against. Assuming that you've installed only the Android 4.1 platform, only this choice should appear and be selected.
7. Via Minimum SDK, either select the minimum Android SDK on which your app runs or keep the default setting.
8. Leave the "Create custom launcher icon" check box checked if you want a custom launcher icon to be created. Otherwise, uncheck this check box when you supply your own launcher icon.
9. Leave the "Mark this project as a library" check box unchecked because you are not creating a library.
10. Leave the "Create Project in Workspace" check box checked, and click Next.
11. On the resulting Configure Launcher Icon pane, make suitable adjustments to the custom launcher icon, and click Next.
12. On the resulting Create Activity pane, leave the Create Activity check box checked, make sure that BlankActivity is selected, and click Next.
13. On the resulting New Blank Activity pane, enter **`GrayScale`** into the Activity Name textfield. Keep all other settings and click Finish.

Use Eclipse's Package Explorer to locate the `GrayScale.java` source file node. Double-click this node and replace the skeletal contents shown in the resulting edit window with Listing 8-9.

Next, create a `grayscale.rs` file node with Listing 8-10's contents under the `src` node, and introduce a `drawable-nodpi` directory node under the `res` directory node into which you must introduce a `sol.jpg` file.

To run `GrayScale` from Eclipse, select Run from the menubar, and then select Run from the drop-down menu. If a *Run As* dialog box appears, select Android Application and click OK. Eclipse launches `emulator` with the AVD1 device, installs `GrayScale.apk`, and runs this app, whose output appears in Figure 8-15.

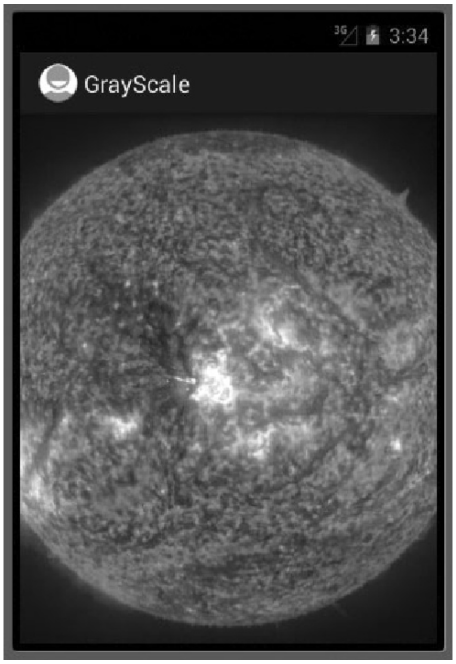


Figure 8-15. *The Sun in grayscale emerges after you click its orange-colored counterpart image.*

8-4. Learning More About Renderscript

Problem

You're intrigued by Renderscript and want to learn more about it. For example, you want to learn how to receive `rsForEach()`'s `usrData` value in the `root()` function.

Solution

The following resources will help you learn more about Renderscript:

- Romain Guy's and Chet Haase's "Learn about Renderscript" video (`http://youtube.com/watch?v=5jz0kSuR2j4`). This one-and-one-half-hour video covers the graphics and compute sides of Renderscript, and it is well worth your time.
- The Android documentation's Renderscript page (`http://developer.android.com/guide/topics/renderscript/index.html`) provides access to important compute information. It also provides access to Renderscript-oriented blog posts.
- The `android.renderscript` package documentation (`http://developer.android.com/reference/android/renderscript/package-summary.html`) can help you to explore the various types, with emphasis on the `RenderScript` and `Allocation` classes.
- The Renderscript reference page (`http://developer.android.com/reference/renderscript/index.html`) provides documentation on all of the functions that Renderscript makes available to your compute script.

Regarding `root()`, this function is minimally declared with two parameters that identify the input/output allocations, as in `void root(const uchar4* v_in, uchar4* v_out)`. However, you can specify three more parameters to obtain a `usrData` value and the x/y coordinates of the value passed to `v_in` in the input allocation, as follows:

```
void root(const uchar4* v_in, uchar4* v_out, const void* usrData, uint32_t x,
          uint32_t y)
```

How It Works

Although the void root(const uchar4*, uchar4*, const void*, uint32_t, uint32_t) function may look a little intimidating, it's not hard to use. For example, Listing 8-11 presents source code to a compute script that uses this expanded function to give an image a wavy appearance as if being seen in water.

Listing 8-11. *Waving an Image*

```
#pragma version(1)
#pragma rs java_package_name(ca.tutortutor.wavyimage)

rs_allocation in;
rs_allocation out;
rs_script script;

int height;

void root(const uchar4* v_in, uchar4* v_out, const void* usrData, uint32_t x,
          uint32_t y)
{
   float scaledy = y/(float) height;
   *v_out = *(uchar4*) rsGetElementAt(in, x, (uint32_t) ((scaledy+
                                   sin(scaledy*100)*0.03)*height));
}

void filter()
{
   rsDebug("RS_VERSION = ", RS_VERSION);
#if !defined(RS_VERSION) || (RS_VERSION < 14)
   rsForEach(script, in, out, 0);
#else
   rsForEach(script, in, out);
#endif
}
```

Listing 8-11's root() function ignores usrData (which isn't required), but it uses the values passed to x and y. It also uses the value passed to height, which represents the height of the image.

The function first uses height to scale the value passed to y to a floating-point value between 0 and 1. It then invokes Renderscript's const void* rsGetElementAt(rs_allocation, uint32_t x, uint32_t y) function to return the input allocation element that's located at position x and y, which is then assigned to *v_out.

The value passed to x, which happens to be the value in `root()`'s x parameter, is self-evident. However, the value passed to y may be a little harder to grasp. The idea is to vary the argument in a sinusoidal pattern so that returned pixels from the original image are chosen to yield a wavy appearance.

Listing 8-12 presents the source code to a `WavyImage` app that communicates with the compute script stored in `wavy.rs`.

Listing 8-12. *Viewing Original and Watery Images of the Sun*

```
package ca.tutortutor.wavyimage;

import android.app.Activity;

import android.os.Bundle;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;

import android.renderscript.Allocation;
import android.renderscript.RenderScript;

import android.view.View;

import android.widget.ImageView;

public class WavyImage extends Activity
{
   boolean original = true;

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      final ImageView iv = new ImageView(this);
      iv.setScaleType(ImageView.ScaleType.CENTER_CROP);
      iv.setImageResource(R.drawable.sol);
      setContentView(iv);
      iv.setOnClickListener(new View.OnClickListener()
                            {
                               @Override
                               public void onClick(View v)
                               {
                                  if (original)
                                     drawWavy(iv, R.drawable.sol);
                                  else
                                     iv.setImageResource(R.drawable.sol);
                                  original = !original;
```

```
                          }
                       });
   }

   private void drawWavy(ImageView iv, int imID)
   {
      Bitmap bmIn = BitmapFactory.decodeResource(getResources(), imID);
      Bitmap bmOut = Bitmap.createBitmap(bmIn.getWidth(), bmIn.getHeight(),
                                         bmIn.getConfig());
      RenderScript rs = RenderScript.create(this);
      Allocation allocIn;
      allocIn = Allocation.createFromBitmap(rs, bmIn,

Allocation.MipmapControl.MIPMAP_NONE,
                                            Allocation.USAGE_SCRIPT);
      Allocation allocOut = Allocation.createTyped(rs, allocIn.getType());
      ScriptC_wavy script = new ScriptC_wavy(rs, getResources(), R.raw.wavy);
      script.set_in(allocIn);
      script.set_out(allocOut);
      script.set_script(script);
      script.set_height(bmIn.getHeight());
      script.invoke_filter();
      allocOut.copyTo(bmOut);
      iv.setImageBitmap(bmOut);
   }
}
```

Listing 8-12 differs from Listing 8-9 mainly via `script.set_height(bmIn.getHeight());`, which passes the bitmap's height to the script's height field so that the script can scale the y value.

If you were to build and run this app (in the same manner as with `GrayScale`), and if you were to click the image of the Sun, you would see the result that's shown in Figure 8-16.

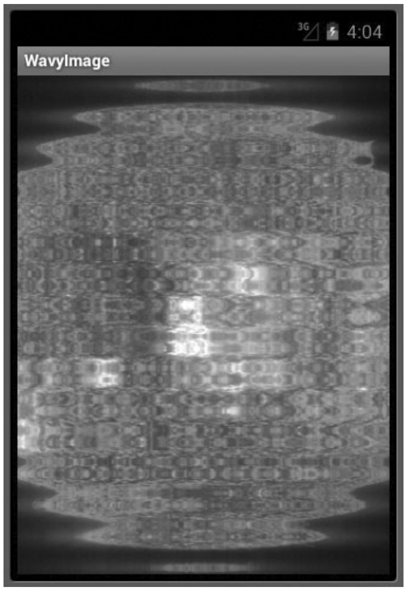


Figure 8-16. *The Sun has a wavy (or possibly watery) appearance.*

Summary

The Android NDK complements the Android SDK by providing a toolset that lets you implement parts of your app by using native code languages such as C and C++. The NDK provides headers and libraries for building native activities, handling user input, using hardware sensors, and more.

Renderscript consists of a language based on C99 (a modern dialect of the C language), a pair of compilers, and a runtime that collectively help you achieve high performance and visually compelling graphics via native code but in a portable manner. You get native app speed along with SDK app portability, and you don't have to use the JNI.