Reuse existing C code with the Android NDK

Learn how to use the Android Native Developer's Kit

Skill Level: Intermediate

Frank Ableson
Entrepreneur
Navitend

12 Apr 2011

The Android Software Developer Kit (SDK) used by the majority of Android application developers requires the use of the Java™ programming language. However, there is a large body of C language code available online. The Android Native Developer Kit (NDK) permits an Android developer to reuse existing C source code within an Android application. In this tutorial, you will create an image processing application in the Java programming language that uses C code to perform basic image processing operations.

Section 1. Before you start

One of the motivations for exploring the NDK in the first place is the opportunity to leverage open source projects, many of which are written in C. After completing this tutorial, you will have learned how to create a Java Native Interface (JNI) library, written in C and compiled with the Native Development Kit (NDK), and incorporate the library into an Android application written in the Java language. The application demonstrates how to perform basic image processing operations against raw image data. You will also learn how to extend the Eclipse build environment to integrate an NDK project into an Android SDK project file. From this foundation, you will be better equipped to port existing open source code to the Android platform.

About this tutorial

This tutorial introduces the Android NDK within the Eclipse environment. The NDK is used to add functionality to an Android application using the C programming language. The tutorial begins with a high-level look at the NDK and its common usage scenarios. From there, the topic of image processing is introduced, followed by an introduction and demonstration of this tutorial's application: IBM Photo Phun. This application is a mix of SDK-based Java code and NDK-compiled C code. The tutorial moves on to introduce the Java Native Interface (JNI), which is the technology of interest when working with the NDK. A look ahead to the completed project's source files provides a roadmap for the application constructed here. Then, in a step-by-step manner, you will construct this application. The Java class and C source files are explained. To conclude, the Eclipse build environment is customized to integrate the NDK tool chain directly into the easy-to-use Eclipse build process.

Prerequisites

To follow this tutorial, you should be comfortable constructing Android applications with the Android SDK and have a basic familiarity with the C programming language. In addition, you will need the following:

- Eclipse and Android Developer Tools (ADT) — Primary code editor, Java Compiler, and Android Development Tools Plug-in
- Android Software Developer Kit (SDK)
- Android Native Developer Kit (NDK)
- PNG Image — Image used for testing image processing operations

I created the code samples for this tutorial on a MacBook Pro with Eclipse V3.4.2 and Android SDK V8, which supports the Android release labeled 2.2 (Froyo). The NDK release used in this tutorial is r4b. The code requires version r4b or later because the image handling capabilities of the Android NDK are not available in prior releases of the NDK.

See Resources for links to these tools.

Section 2. The Android NDK

Let's begin with a look at the Android NDK and how it can be used for enhancing the Android platform. While the Android SDK provides a very rich programming environment, the Android NDK broadens the horizons and can speed up the delivery of desired functionality by bringing in existing source code, some of which may be proprietary and some of which may be open source code.

The NDK

The NDK is a software release available as a free download from the Android website. The NDK includes all the components necessary to incorporate functionality written in C into an Android application. The initial release of the NDK offered only the most primitive of functionality with significant constraints. With each successive release, the NDK has expanded its capabilities. As of r5 of the NDK application, authors can write a significant portion of an application directly in C, including user interface and event-handling capability. The features enabling the image handling functionality demonstrated here were introduced with the r4b version of the NDK.

Two common uses of the NDK are to increase application performance and to leverage existing C code by porting it to Android. Let's look first at performance improvement. Writing code in C does not guarantee a significant increase in performance. In fact, poorly written native code can actually slow down an application when compared to a well-written Java application. Application performance improvements are available when carefully crafted functions written in C are leveraged to perform memory-based or computationally intensive operations like those demonstrated in this tutorial. In particular, algorithms that leverage pointer arithmetic are particularly ripe for use with the NDK. The second common use case for the NDK is to port an existing body of C code written for another platform, such as Linux®. This tutorial demonstrates the NDK in a manner that highlights the performance and the re-use cases.

The NDK contains a compiler and build scripts, allowing you to focus on the C source files and leave the build magic to the NDK installation. The NDK build process is easily incorporated into the Eclipse development environment, which is demonstrated in the section on Customizing Eclipse.

Before jumping into the application itself, let's take a brief detour to discuss some fundamentals of digital image processing.

Fundamentals of digital imaging processing

An enjoyable aspect of modern computer technology is the advent and ubiquity of digital photography. There is more to digital photography than simply catching your kid doing something cute. Digital images are found everywhere from candid cellphone shots to high-end wedding albums, deep-space images, and numerous other applications. Digital images are easy to capture, exchange, and even alter. Modifying a digital image is of interest to us here and represents the core functionality of the tutorial's sample application.

Digital image manipulation occurs in myriad ways, including but not limited to the following operations:

- **Cropping** — Extracting a portion of an image
- **Scaling** — Changing the size of an image
- **Rotating** — Changing the orientation of an image
- **Conversion** — Converting from one format to another
- **Sampling** — Changing the density of an image
- **Mixing/Morphing** — Changing the appearance of an image
- **Filtering** — Extracting elements of an image, such as colors or frequencies
- **Edge detection** — Used for machine vision applications to identify objects within an image
- **Compressing** — Reducing the storage size of an image
- Enhancing an image through pixel operations:
    - Histogram equalization
    - Contrast
    - Brightness

Some of these operations are performed on a pixel-by-pixel basis, while others involve matrix math to work on small sections of the image at a time. Regardless of the operations, all image processing algorithms involve working with raw image data. This tutorial demonstrates the use of pixel and matrix operations in the C programming language, running on an Android device.

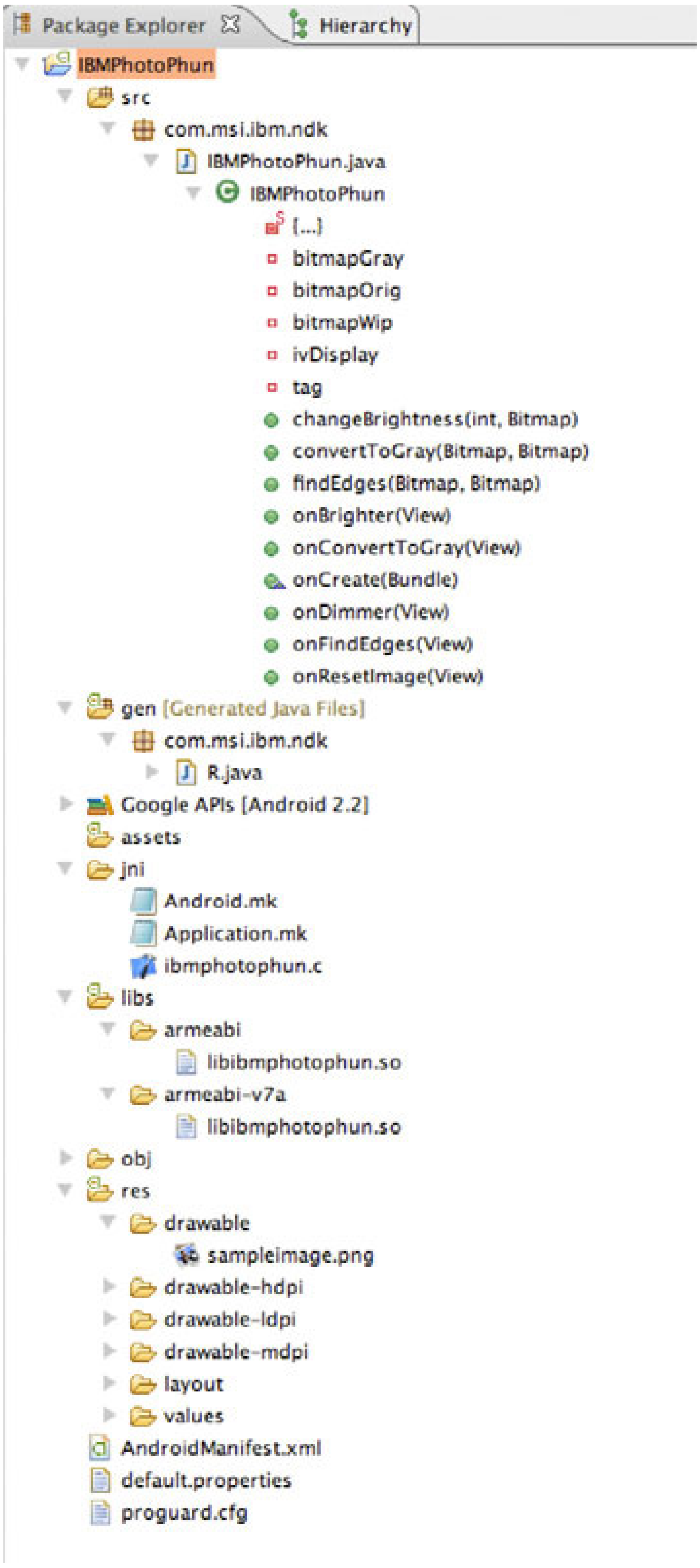
Section 3. The application architecture

This section explores the architecture of the tutorial's sample application, beginning with a high-level glance at the completed project, then progressing through each of the major steps in its construction. You can follow along step by step to reconstruct the application yourself or you can download the complete project from the Resources section.

The completed project

This tutorial demonstrates the construction of a simple image processing application, IBM Photo Phun. Figure 1 shows a screenshot from the Eclipse IDE with the project expanded to see the source and output files.

Figure 1. Eclipse project view



```
Package Explorer   Hierarchy
▼ IBMPhotoPhun
  ▼ src
    ▼ com.msi.ibm.ndk
      ▼ IBMPhotoPhun.java
        ▼ IBMPhotoPhun
            {...}
            bitmapGray
            bitmapOrig
            bitmapWip
            ivDisplay
            tag
            changeBrightness(int, Bitmap)
            convertToGray(Bitmap, Bitmap)
            findEdges(Bitmap, Bitmap)
            onBrighter(View)
            onConvertToGray(View)
            onCreate(Bundle)
            onDimmer(View)
            onFindEdges(View)
            onResetImage(View)
  ▼ gen [Generated Java Files]
    ▼ com.msi.ibm.ndk
      ▶ R.java
  ▶ Google APIs [Android 2.2]
    assets
  ▼ jni
      Android.mk
      Application.mk
      ibmphotophun.c
  ▼ libs
    ▼ armeabi
        libibmphotophun.so
    ▼ armeabi-v7a
        libibmphotophun.so
  ▶ obj
  ▼ res
    ▼ drawable
        sampleimage.png
    ▶ drawable-hdpi
    ▶ drawable-ldpi
    ▶ drawable-mdpi
    ▶ layout
    ▶ values
    AndroidManifest.xml
    default.properties
    proguard.cfg
```

The application's UI is constructed with traditional Android development techniques, using a single layout file (main.xml) and a single Activity, implemented in IBMPhotoPhun.java. A single C source file, located in a folder named jni, beneath the project's main folder, contains the image processing routines. The NDK tool chain compiles the C source file into a shared library file named libibmphotophun.so. The compiled library file(s) are stored in the libs folder. A library file is created for each target hardware platform or processor architecture. Table 1 enumerates the application's source files.

Table 1. The required application source files

File	Comment
IBMPhotoPhun.java	Extends the Android Activity class for UI and application logic
Ibmphotophun.c	Implements image processing routines
main.xml	Home page of the application UI
AndroidManifest.xml	Deployment descriptor for the Android application
sampleimage.png	Image used for demonstration purposes (feel free to substitute an image of your own)
Android.mk	Makefile snippet used by the NDK to construct the JNI library

If you don't have a working Android development environment, now is a great time to install the Android tools. For more information on how to set up an Android development environment, see Resources for links to the required tools, plus some introductory articles and tutorials on developing applications for Android. Having a familiarity with Android is helpful in understanding this tutorial.

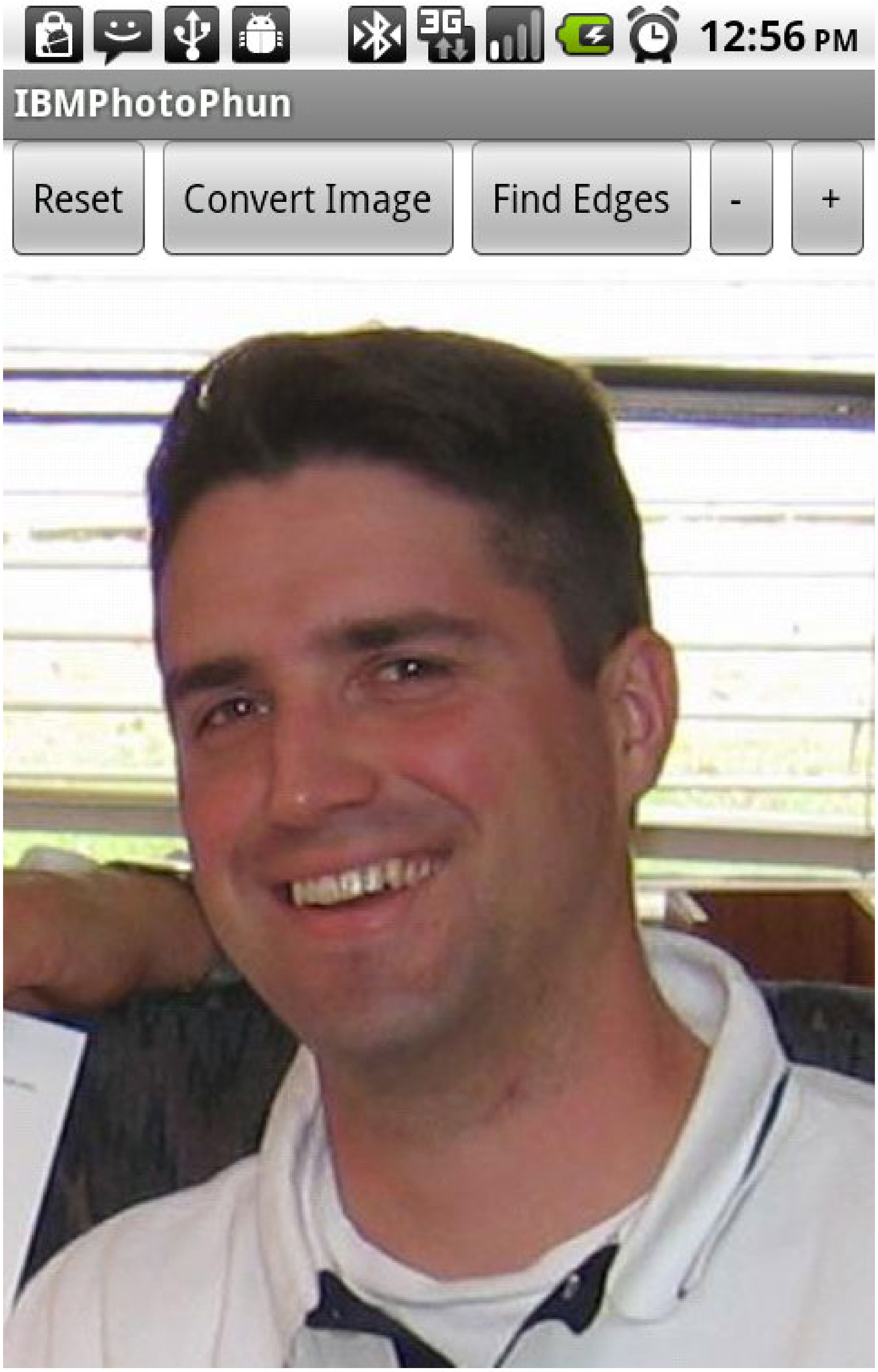
Now that you have an overview of the architecture and the application, you can see how it looks when running on an Android device.

Demonstrating the application

Sometimes it's helpful to begin with the end in mind, so before you dive into the step-by-step process of creating this application, have a quick look at it in action. The following screenshots were captured from a Nexus One running Android 2.2 (Froyo). The images were captured using the Dalvik Debug Monitor Service (DDMS) tool, which installs as part of the Android Developer Tools Eclipse plug-in.

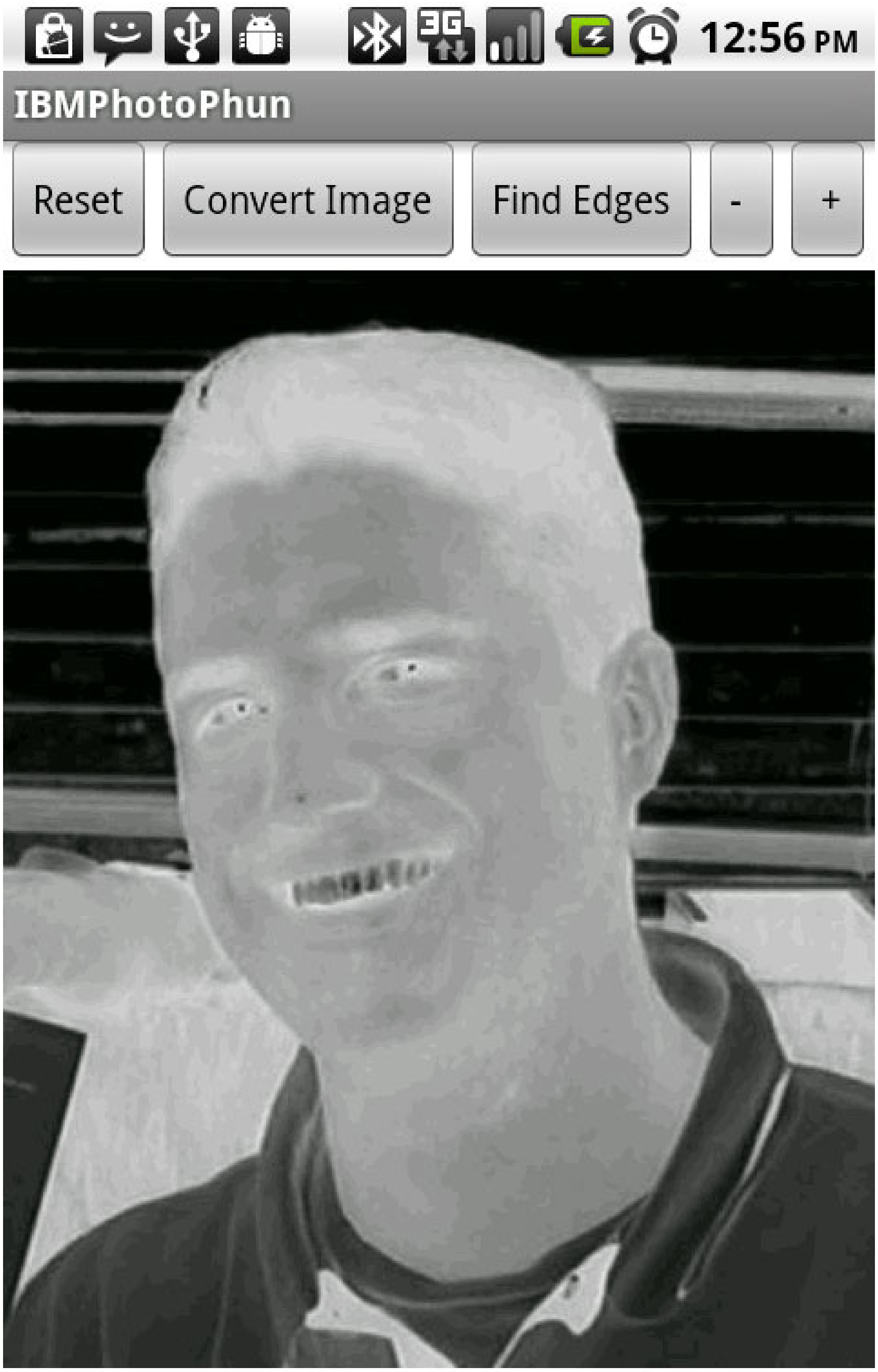
Figure 2 shows the home screen of the application with the sample image loaded. One quick look at the image and you will understand how I wound up as a programmer and not on the set of some television program, thanks to my "fit for radio" face. Please feel free to substitute your own image when building the application yourself.

Figure 2. Home screen of the IBM Photo Phun application


12:56 PM
IBMPhotoPhun
Reset
Convert Image
Find Edges
-
+

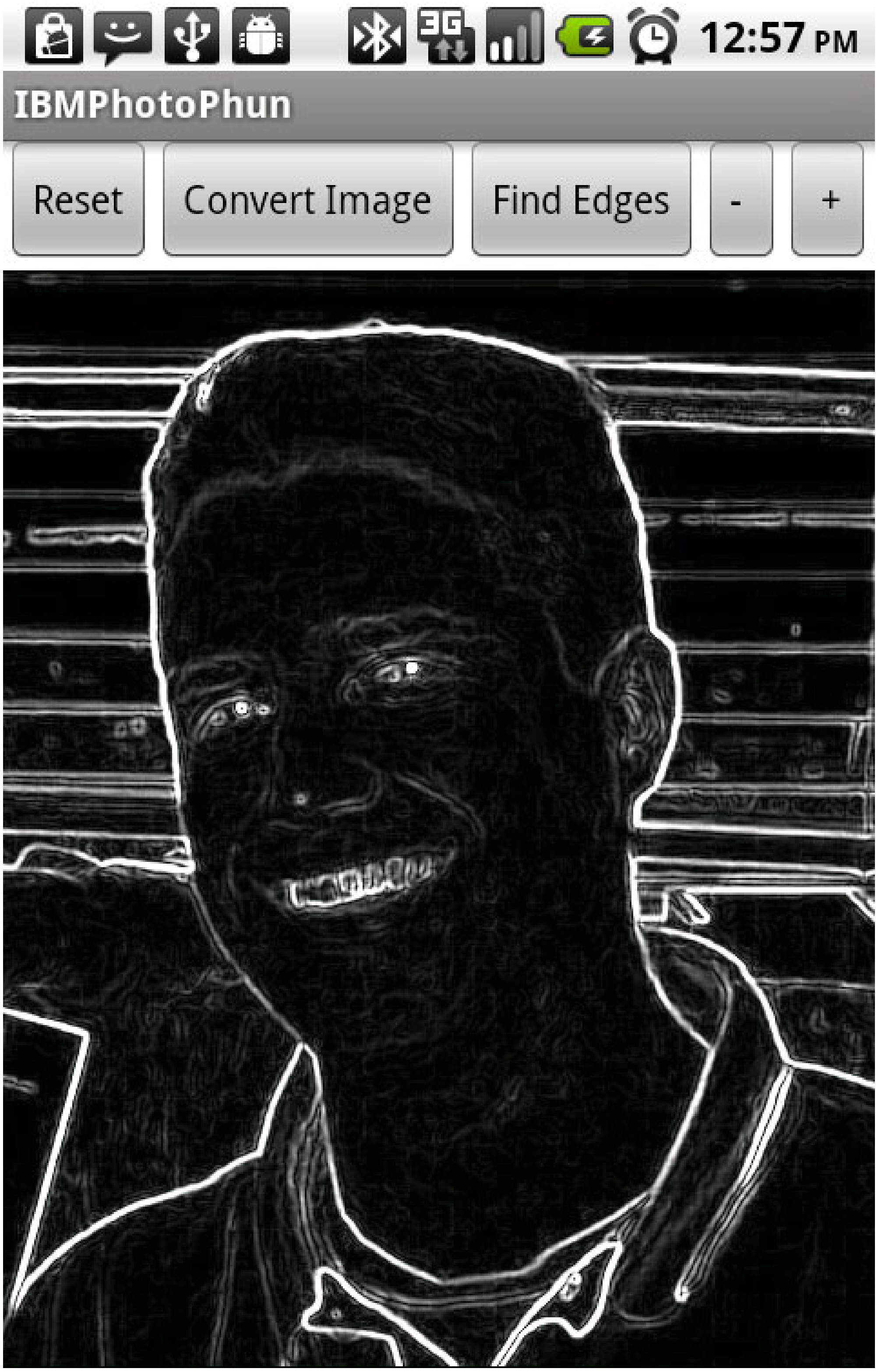
The buttons across the top of the screen allow you to change the image. The first button, **Reset**, restores the image to this original color image. Selecting the **Convert Image** button converts the image to grayscale, as shown in Figure 3.

Figure 3. Grayscale image


12:56 PM
IBMPhotoPhun
Reset
Convert Image
Find Edges
-
+

The **Find Edges** button starts with the original color image, converts it to grayscale, then performs a Sobel Edge Detection algorithm. Figure 4 shows the results of the edge detection algorithm.

Figure 4. Detecting the edges


12:57 PM
IBMPhotoPhun
Reset
Convert Image
Find Edges
-
+

Edge-detection algorithms are often used in machine vision applications as a preliminary step in a multi-step image processing operation. From this point, the final two buttons allow you to make the image darker or lighter by changing the brightness of each pixel. Figure 5 shows a brighter version of the grayscale image.

Figure 5. Increased brightness

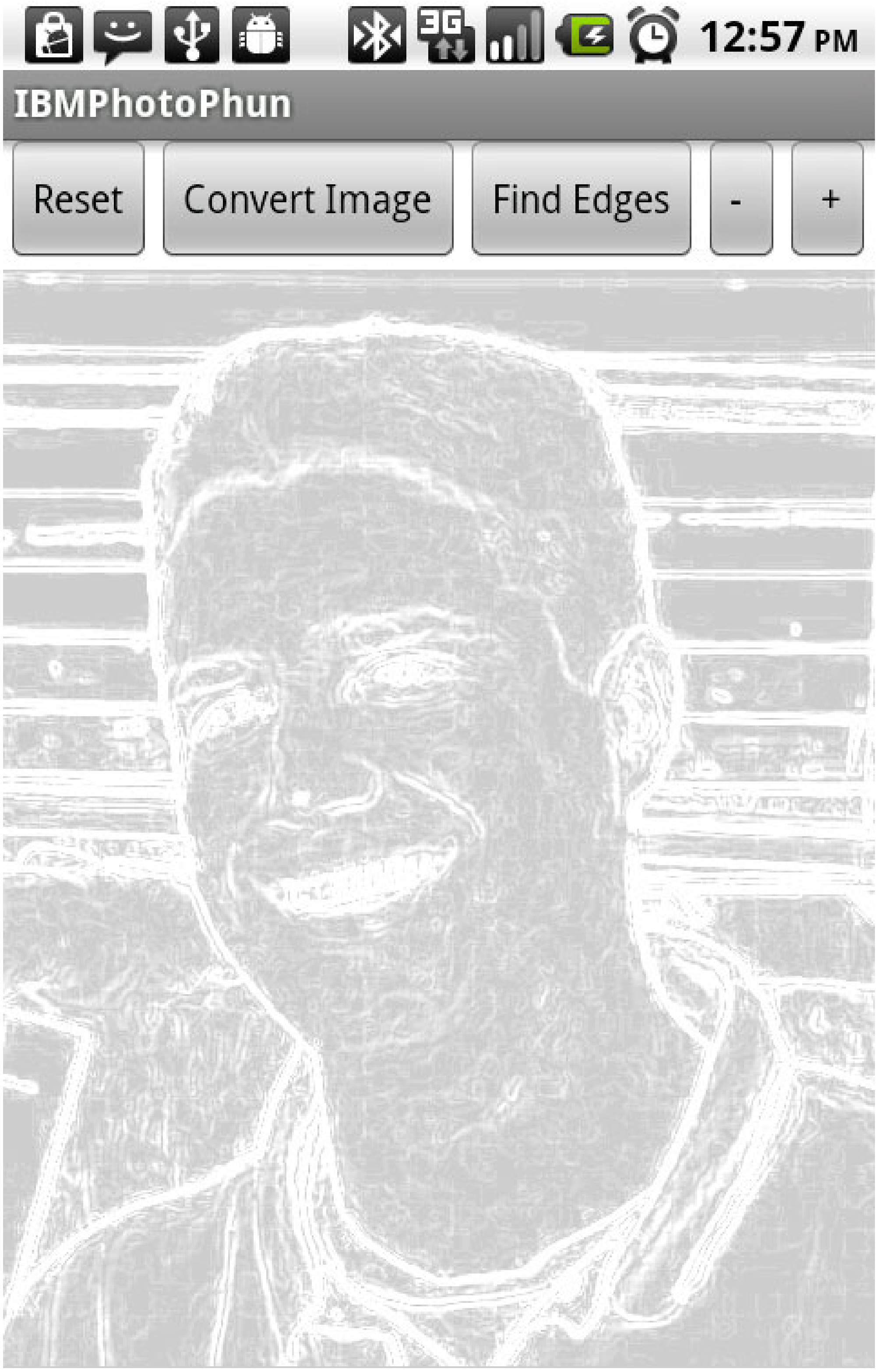
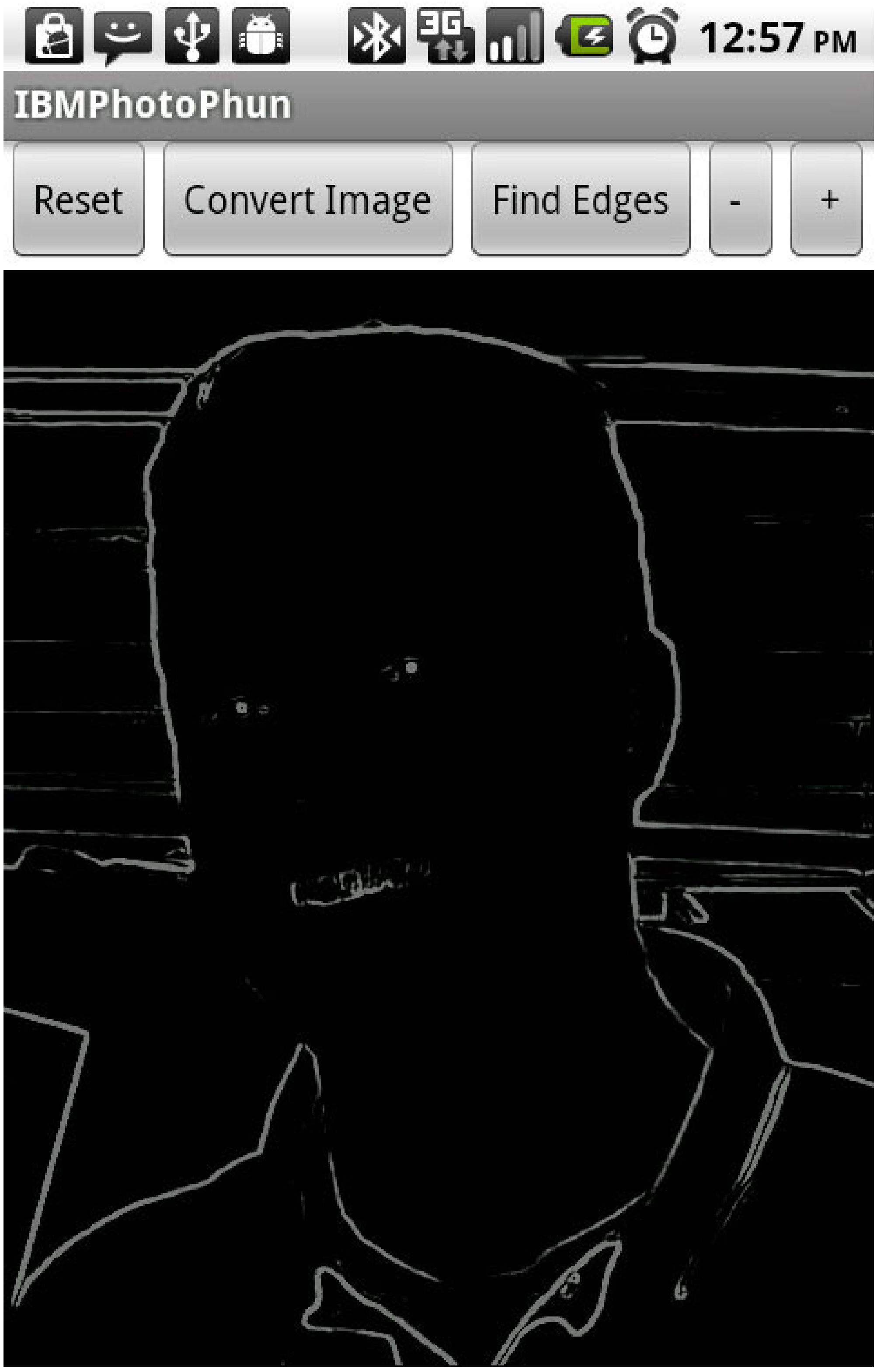

12:57 PM
IBMPhotoPhun
Reset
Convert Image
Find Edges
-
+

Figure 6 shows the edges image a bit darker.

Figure 6. Decreased brightness


12:57 PM
IBMPhotoPhun
Reset
Convert Image
Find Edges
-
+

Now it is your turn.

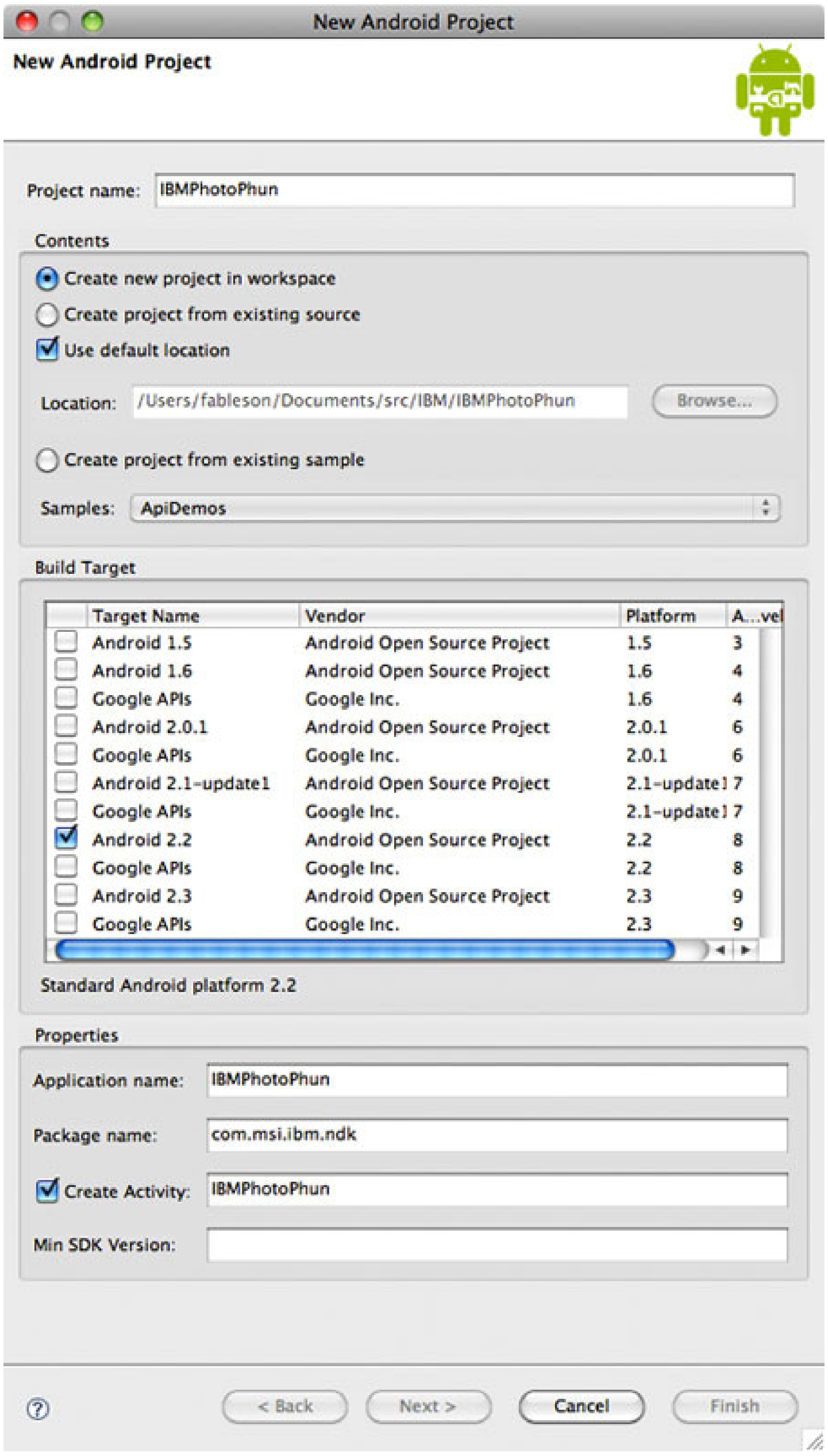
Section 4. Creating the application

In this section, we will create the application by leveraging the tools provided in the Eclipse ADT plug-in. Even if you aren't familiar with creating applications for Android, you should be able to follow along quite readily and learn from this section. The Resources section contains helpful articles and tutorials on the basics of creating Android applications.

ADT new project wizard

Creating the application within the Eclipse IDE is very straightforward, thanks to the ADT new project wizard, shown in Figure 7.

Figure 7. Creating a new Android project



New Android Project

Project name: IBMPhotoPhun

Contents

● Create new project in workspace
○ Create project from existing source
✓ Use default location

Location: /Users/fableson/Documents/src/IBM/IBMPhotoPhun

○ Create project from existing sample

Samples: ApiDemos

Build Target

	Target Name	Vendor	Platform	A...vel
	Android 1.5	Android Open Source Project	1.5	3
	Android 1.6	Android Open Source Project	1.6	4
	Google APIs	Google Inc.	1.6	4
	Android 2.0.1	Android Open Source Project	2.0.1	6
	Google APIs	Google Inc.	2.0.1	6
	Android 2.1-update1	Android Open Source Project	2.1-update1	7
	Google APIs	Google Inc.	2.1-update1	7
✓	Android 2.2	Android Open Source Project	2.2	8
	Google APIs	Google Inc.	2.2	8
	Android 2.3	Android Open Source Project	2.3	9
	Google APIs	Google Inc.	2.3	9

Standard Android platform 2.2

Properties

Application name: IBMPhotoPhun

Package name: com.msi.ibm.ndk

✓ Create Activity: IBMPhotoPhun

Min SDK Version:

When filling out the new project wizard, provide the following information:

1. Valid project name.
2. Build target. Note that for this project, you must use Android V2.2 or Android V2.3 as the target SDK platform level.
3. Valid application name.
4. Package name.
5. Activity name.

Once you have populated the wizard screen, select **Finish**. Clicking the **Next** button prompts for creating a "Test" project to accompany this project, which is a useful step, but for another day.

Once the project is populated into Eclipse, you are ready to implement the source files necessary for this application. You will begin with the UI elements of the application.

Implementing the user interface

The UI for this application is rather straightforward. It contains a single `Activity` with a handful of `Button` widgets and an `ImageView` widget to display the chosen image. Like many Android applications, the UI is defined in the main.xml file, shown in Listing 1.

Listing 1. UI layout file, main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#ffffffff"
    >
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="center"

    >

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/btnReset"
    android:text="Reset"
    android:visibility="visible"
    android:onClick="onResetImage"
/>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/btnConvert"
    android:text="Convert Image"
    android:visibility="visible"
    android:onClick="onConvertToGray"
/>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/btnFindEdges"
    android:text="Find Edges"
    android:visibility="visible"
    android:onClick="onFindEdges"
/>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/btnDimmer"
    android:text="- "
    android:visibility="visible"
    android:onClick="onDimmer"
/>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/btnBrighter"
    android:text=" +"
    android:visibility="visible"
    android:onClick="onBrighter"
/>
</LinearLayout>
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="centerCrop"
    android:layout_gravity="center_vertical|center_horizontal"
    android:id="@+id/ivDisplay"
/>
</LinearLayout>
```

Note the use of two `LinearLayout` elements. The outer element controls the vertical flow of the UI and the inner `LinearLayout` is set up for horizontal management of its children. The horizontal layout element holds all of the `Button` widgets across the top of the screen. The `ImageView` is set up to center the contained image and has an `id` attribute, permitting you to manipulate its contents during run time.

Each of the `Button` widgets has an `onClick` attribute. The value of this attribute must correspond to a public void method within the containing `Activity` class, which takes a single `View` argument. This approach is a quick and easy means of setting up click handlers without the trouble of defining anonymous handlers or getting access to the element during runtime. See Resources for more information on this approach to handling `Button` presses.

Once the UI has been defined in the layout file, the `Activity` code must be written to work with the UI. This is implemented in the file IBMPhotoPhun.java, where the

`Activity` class is extended. You can see the code in Listing 2.

Listing 2. IBM Photo Phun imports and class declaration

```
/*
 * IBMPhotoPhun.java
 *
 * Author: Frank Ableson
 * Contact Info: fableson@navitend.com
 */

package com.msi.ibm.ndk;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.graphics.BitmapFactory;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;
import android.view.View;
import android.widget.ImageView;


public class IBMPhotoPhun extends Activity {
    private String tag = "IBMPhotoPhun";
    private Bitmap bitmapOrig = null;
    private Bitmap bitmapGray = null;
    private Bitmap bitmapWip = null;
    private ImageView ivDisplay = null;



    // NDK STUFF
    static {
        System.loadLibrary("ibmphotophun");
    }
    public native void convertToGray(Bitmap bitmapIn,Bitmap bitmapOut);
    public native void changeBrightness(int direction,Bitmap bitmap);
    public native void findEdges(Bitmap bitmapIn,Bitmap bitmapOut);
    // END NDK STUFF



    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Log.i(tag,"before image stuff");
        ivDisplay = (ImageView) findViewById(R.id.ivDisplay);


        // load bitmap from resources
        BitmapFactory.Options options = new BitmapFactory.Options();
        // Make sure it is 24 bit color as our image processing algorithm
                // expects this format
        options.inPreferredConfig = Config.ARGB_8888;
        bitmapOrig = BitmapFactory.decodeResource(this.getResources(),
R.drawable.sampleimage,options);
        if (bitmapOrig != null)
            ivDisplay.setImageBitmap(bitmapOrig);

    }

    public void onResetImage(View v) {
        Log.i(tag,"onResetImage");
        ivDisplay.setImageBitmap(bitmapOrig);

    }

    public void onFindEdges(View v) {
        Log.i(tag,"onFindEdges");

        // make sure our target bitmaps are happy
        bitmapGray = Bitmap.createBitmap(bitmapOrig.getWidth(),bitmapOrig.getHeight(),
Config.ALPHA_8);
        bitmapWip = Bitmap.createBitmap(bitmapOrig.getWidth(),bitmapOrig.getHeight(),
Config.ALPHA_8);
        // before finding edges, we need to convert this image to gray
        convertToGray(bitmapOrig,bitmapGray);
        // find edges in the image
        findEdges(bitmapGray,bitmapWip);
        ivDisplay.setImageBitmap(bitmapWip);

    }
    public void onConvertToGray(View v) {
        Log.i(tag,"onConvertToGray");

        bitmapWip = Bitmap.createBitmap(bitmapOrig.getWidth(),bitmapOrig.getHeight(),
Config.ALPHA_8);
        convertToGray(bitmapOrig,bitmapWip);
        ivDisplay.setImageBitmap(bitmapWip);
    }

    public void onDimmer(View v) {
        Log.i(tag,"onDimmer");

        changeBrightness(2,bitmapWip);
        ivDisplay.setImageBitmap(bitmapWip);
    }
    public void onBrighter(View v) {
        Log.i(tag,"onBrighter");

        changeBrightness(1,bitmapWip);
        ivDisplay.setImageBitmap(bitmapWip);
    }
}
```

Let's break this down into a few notable comments:

1. There are a handful of member variables:
    - `tag` — This is used in all logging statements to help filter the `LogCat` during debugging.
    - `bitmapOrig` — This `Bitmap` holds the original color image.
    - `bitmapGray` — This `Bitmap` holds a grayscale copy of the image and is only used temporarily during the `findEdges` routine.
    - `bitmapWip` — This `Bitmap` holds the grayscale image that is modified when brightness values are modified.
    - `ivDisplay` — This `ImageView` is a reference to the `ImageView` defined in the main.xml layout file.

2. The "NDK Stuff" section includes four lines:
   - The library containing our native code is loaded with a call to `System.loadLibrary`. Note that this code is contained in a block marked as "static." This causes the library to be loaded when the application is started.
   - Prototype declaration for `convertToGray` — This function takes two parameters. The first is a color `Bitmap` and the second is a `Bitmap` that is populated with a grayscale version of the first.
   - Prototype declaration for `changeBrightness` — This function takes two parameters. The first is an integer representing up or `down`. The second is a `Bitmap` that is modified on a pixel by pixel basis.
   - Prototype declaration for `findEdges`. This takes two parameters. The first is a grayscale `Bitmap` and the second is a `Bitmap` that receives the "edges only" version of the image.
3. The `onCreate` method inflates the layout identified by `R.layout.main`, obtains a reference to the ImageView widget (`ivDisplay`), then loads the color image from the resources.
   - The `BitmapFactory` method takes an `options` parameter that permits you to load the image in the ARGB format. "A" stands for alpha channel and "RGB" stands for red, green, blue, respectively. Many open source image processing libraries expect a 24-bit color image, eight bits each for red, green, and blue, with each pixel consisting of the RGB triplet. Each value ranges from 0 to 255. Images on the Android platform are stored as a 32-bit integer as alpha, red, green, blue.
   - Once the image is loaded, it is displayed in the `ImageView`.
4. The balance of the methods in this class correspond to "click handlers" for the `Button` widgets:
   - `onResetImage` loads the original color image into the `ImageView`.
   - `onConvertToGray` creates the target `Bitmap` as an 8-bit image and calls the `convertToGray` native function. The resulting image (bitmapWip) is displayed in the `ImageView`.
   - `onFindEdges` creates two intermediate `Bitmap` objects, converts the color image to a grayscale image and calls the `findEdges` native function. The resulting image (bitmapWip) is displayed in the `ImageView`.
   - `onDimmer` and `onBrighter` each invoke the `changeBrightness`

method to modify the image. The resulting image (bitmapWip) is displayed in the `ImageView`.

That about wraps up the UI code. It is now time for you to implement the image process routines, but first, we need to create the library itself.

Section 5. Creating the NDK files

Now that the Android application's UI and application logic are in place, you need to implement the image processing functions. In order to do this, you need to create a Java-native library with the NDK. In this case, you will use some public domain C code to implement the image processing functions and package them into a library usable by the Android application.

Building the native library

The NDK creates shared libraries and relies on a makefile system. To build the native library for this project, you need to perform the following steps:

1. Create a new folder named jni beneath your project file.
2. Within the jni folder, create a file named Android.mk, which contains the makefile instructions to properly build and name your library.
3. Within the jni folder, create the source file, which is referenced in the Android.mk file. The name of the C source file for this tutorial is ibmphotophun.c.

Listing 3 contains the Android.mk file contents.

Listing 3. Android.mk file

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

Among other things, this makefile (snippet) instructs the NDK to:

1. Compile the ibmphotophun.c source file into a shared library.

2. Name the shared library. By default, the shared library naming convention is lib<modulename>.so. So, the resulting file here is named libibmphotophun.so.

3. Specify the required "input" libraries. The shared library relies upon two built-in library files for logging (liblog.so) and jni graphics (libjnigraphics.so). The logging library permits you to add entries to the `LogCat`, which is helpful during the development phase of your project. The graphics library provides routines for working with Android bitmaps and their image data.

The ibmphotophun.c source file contains a few C include statements and the definition of the argb type, which corresponds to the Color data type in the Android SDK. Listing 4 shows ibmphotophun.c without the image routines, which are presented next.

Listing 4. Ibmphotophun.c macros and includes

```
/*
 * ibmphotophun.c
 *
 * Author: Frank Ableson
 * Contact Info: fableson@msiservices.com
 */

#include <jni.h>
#include <android/log.h>
#include <android/bitmap.h>

#define  LOG_TAG    "libibmphotophun"
#define  LOGI(...)  __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define  LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)

typedef struct
{
    uint8_t alpha;
    uint8_t red;
    uint8_t green;
    uint8_t blue;
} argb;
```

The `LOGI` and `LOGE` macros make calls to the Logging facility and are equivalent in functionality to `Log.i()` and `Log.e()` respectively in the Android SDK. The `argb` data type defined with the `typedef` struct keywords allows the C code to access the four data elements of a single pixel stored in a 32-bit integer. The three include statements provide the necessary declarations to the C compiler for jni glue, logging, and bitmap handling, respectively.

It is now time for you to implement some image processing routines, but before we examine the code itself, you need to understand the naming convention of JNI functions.

When Java code calls a native function, it maps the function name to an expanded, or decorated, function, which is exported by the JNI shared library. Here is the convention: `Java_fully_qualified_classname_functionname`.

For example, the `convertToGray` function is implemented in the C code as `Java_com_msi_ibm_ndk_IBMPhotoPhun_convertToGray`.

The first two arguments to the JNI functions include a pointer to the JNI environment and to the calling-class object instance. For more information about JNI, please see the Resources section.

Building the library is quite simple. Open a terminal (or DOS) window and change directory to the jni folder where you have stored these files. Make sure that the NDK is in your path and execute the ndk-build script. This script contains all the glue necessary to build the library. The resulting library is placed in the libs folder on the same level as the jni folder (<project folder>/libs/, for example).

When the Android application is packaged by the ADT plug-in for Eclipse, the library files are included and "wired up" automatically for you. One library file is generated for each supported hardware platform. The correct library is loaded at runtime.

Let's look at how the image processing algorithms are implemented.

Implementing the image processing algorithms

The image processing routines used by this application were adapted from a variety of public domain and academic routines, along with my own experience as an image processing hobbyist. Two of the functions use pixel operations and the third utilizes a minimal matrix approach. Let's look first at the `convertToGray` function in Listing 5.

Listing 5. convertToGray function

```
/*
convertToGray
Pixel operation
*/
JNIEXPORT void JNICALL Java_com_msi_ibm_ndk_IBMPhotoPhun_convertToGray(JNIEnv
* env, jobject  obj, jobject bitmapcolor,jobject bitmapgray)
{
    AndroidBitmapInfo  infocolor;
    void*              pixelscolor;
    AndroidBitmapInfo  infogray;
    void*              pixelsgray;
    int                ret;
    int             y;
    int             x;
    LOGI("convertToGray");
    if ((ret = AndroidBitmap_getInfo(env, bitmapcolor, &infocolor)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }


    if ((ret = AndroidBitmap_getInfo(env, bitmapgray, &infogray)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }


    LOGI("color image :: width is %d; height is %d; stride is %d; format is %d;flags is
%d",infocolor.width,infocolor.height,infocolor.stride,infocolor.format,infocolor.flags);
    if (infocolor.format != ANDROID_BITMAP_FORMAT_RGBA_8888) {
        LOGE("Bitmap format is not RGBA_8888 !");
        return;
    }


    LOGI("gray image :: width is %d; height is %d; stride is %d; format is %d;flags is
%d",infogray.width,infogray.height,infogray.stride,infogray.format,infogray.flags);
    if (infogray.format != ANDROID_BITMAP_FORMAT_A_8) {
        LOGE("Bitmap format is not A_8 !");
        return;
    }


    if ((ret = AndroidBitmap_lockPixels(env, bitmapcolor, &pixelscolor)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    if ((ret = AndroidBitmap_lockPixels(env, bitmapgray, &pixelsgray)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    // modify pixels with image processing algorithm

    for (y=0;y<infocolor.height;y++) {
        argb * line = (argb *) pixelscolor;
        uint8_t * grayline = (uint8_t *) pixelsgray;
        for (x=0;x<infocolor.width;x++) {
            grayline[x] = 0.3 * line[x].red + 0.59 * line[x].green + 0.11*line[x].blue;
        }

        pixelscolor = (char *)pixelscolor + infocolor.stride;
        pixelsgray = (char *) pixelsgray + infogray.stride;
    }

    LOGI("unlocking pixels");
    AndroidBitmap_unlockPixels(env, bitmapcolor);
    AndroidBitmap_unlockPixels(env, bitmapgray);


}
```

This function takes two arguments from the calling Java code: a color `Bitmap` in the ARGB format and an 8-bit grayscale `Bitmap` that receives a grayscale version of the color image. Here is a walk-through of the code:

1. The `AndroidBitmapInfo` structure, defined in bitmap.h, is helpful for learning about a `Bitmap` object.

2. The `AndroidBitmap_getInfo` function, found in the jnigraphics library, obtains information about a specific `Bitmap` object.
3. The next step is to ensure that the bitmaps passed into the `convertToGray` function are of the expected format.
4. The `AndroidBitmap_lockPixels` function locks down the image data so you can perform operations directly on the data.
5. The `AndroidBitmap_unlockPixels` function unlocks previously locked pixel data. These functions should be called as a "lock/unlock pair".
6. Sandwiched between the lock and unlock functions you see the pixel operations.

Pointer fun

Image processing applications in C typically involve the use of pointers. Pointers are variables that "point" to a memory address. The data type of a variable specifies the type and size of memory you are working with. For example a `char` represents a signed 8-bit value, so a `char` pointer (`char *`) allows you to reference an 8-bit value and perform operations through that pointer. The image data is represented as `uint8_t`, which means an unsigned 8-bit value, where each byte holds a value ranging from 0 to 255. A collection of three 8-bit unsigned values represents a pixel of image data for a 24-bit image.

Working through an image involves working on the individual rows of data and moving across the columns. The `Bitmap` structure contains a member known as the stride. The stride represents the width, in bytes, of a row of image data. For example, a 24-bit color plus alpha channel image has 32 bits, or 4 bytes, per pixel. So an image with a width of 320 pixels has a stride of 320*4 or 1,280 bytes. An 8-bit grayscale image has 8 bits, or 1 byte, per pixel. A grayscale bitmap with a width of 320 pixels has a stride of 320*1 or simply 320 bytes. With this information in mind, let's look at the image processing algorithm for converting a color image to a grayscale image:

1. When the image data is "locked," the base address of the image data is referenced by a pointer named `pixelscolor` for the input color image and `pixelsgray` for the output grayscale image.
2. Two `for-next` loops allow you to iterate over the entire image.
    1. First, you iterate over the height of the image, one pass per "row."

Use the `infocolor.height` value to get the count of the rows.

2. On each pass through the rows a pointer is set up to the memory location corresponding to the first "column" of image data for the row.

3. As you iterate over the columns for a particular row, you convert each pixel of color data to a single value representing the grayscale value.

4. When the complete row is converted you need to advance the pointers to the next row. This is done by jumping forward in memory by the stride value.

For all pixel-oriented image processing operations, you follow the above format. For example, consider the `changeBrightness` function shown in Listing 6.

Listing 6. changeBrightness function

```
/*
changeBrightness
Pixel Operation
*/
JNIEXPORT void JNICALL Java_com_msi_ibm_ndk_IBMPhotoPhun_changeBrightness(JNIEnv
* env, jobject  obj, int direction,jobject bitmap)
{
    AndroidBitmapInfo  infogray;
    void*              pixelsgray;
    int                ret;
    int             y;
    int             x;
    uint8_t save;




    if ((ret = AndroidBitmap_getInfo(env, bitmap, &infogray)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }


    LOGI("gray image :: width is %d; height is %d; stride is %d; format is %d;flags is
%d",infogray.width,infogray.height,infogray.stride,infogray.format,infogray.flags);
    if (infogray.format != ANDROID_BITMAP_FORMAT_A_8) {
        LOGE("Bitmap format is not A_8 !");
        return;
    }


    if ((ret = AndroidBitmap_lockPixels(env, bitmap, &pixelsgray)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    // modify pixels with image processing algorithm


    LOGI("time to modify pixels....");
    for (y=0;y<infogray.height;y++) {
        uint8_t * grayline = (uint8_t *) pixelsgray;
        int v;
        for (x=0;x<infogray.width;x++) {
            v = (int) grayline[x];

            if (direction == 1)
                v -=5;
            else
                v += 5;
            if (v >= 255) {
                grayline[x] = 255;
            } else if (v <= 0) {
                grayline[x] = 0;
            } else {
                grayline[x] = (uint8_t) v;
            }
        }

        pixelsgray = (char *) pixelsgray + infogray.stride;
    }

    AndroidBitmap_unlockPixels(env, bitmap);


}
```

This function operates in a manner very similar to the `convertToGray` function with the following distinctions:

1. This function requires only a single grayscale bitmap. The image passed in is modified in place.
2. The function adds or subtracts a value of 5 from each pixel on each pass. This constant may be changed. I used 5 because it made the image change noticeably with each pass without having to press the plus or minus buttons excessively.
3. The pixel values are constrained between 0 and 255. Be careful when performing these operations with unsigned variables directly as it is easy to "wrap around." My initial effort on the `changeBrightness` function resulted in adding 5 to a value such as 252 and winding up with 2. The effect was fun to watch, but not what I was after. That is why I am using the integer named v and casting the pixel data to the signed integer and then comparing that value to 0 and 255.

There remains one more image processing algorithm to examine: the `findEdges` function, which works a little bit differently from the prior two pixel-oriented functions. Listing 7 shows the `findEdges` function.

Listing 7. The findEdges function detects outlines within an image

```


/*
findEdges
```

```
Matrix operation
*/
JNIEXPORT void JNICALL Java_com_msi_ibm_ndk_IBMPhotoPhun_findEdges(JNIEnv
* env, jobject  obj, jobject bitmapgray,jobject bitmapedges)
{
    AndroidBitmapInfo  infogray;
    void*              pixelsgray;
    AndroidBitmapInfo  infoedges;
    void*              pixelsedge;
    int                ret;
    int             y;
    int             x;
    int             sumX,sumY,sum;
    int             i,j;
    int                Gx[3][3];
    int                Gy[3][3];
    uint8_t            *graydata;
    uint8_t            *edgedata;


    LOGI("findEdges running");

    Gx[0][0] = -1;Gx[0][1] = 0;Gx[0][2] = 1;
    Gx[1][0] = -2;Gx[1][1] = 0;Gx[1][2] = 2;
    Gx[2][0] = -1;Gx[2][1] = 0;Gx[2][2] = 1;



    Gy[0][0] = 1;Gy[0][1] = 2;Gy[0][2] = 1;
    Gy[1][0] = 0;Gy[1][1] = 0;Gy[1][2] = 0;
    Gy[2][0] = -1;Gy[2][1] = -2;Gy[2][2] = -1;


    if ((ret = AndroidBitmap_getInfo(env, bitmapgray, &infogray)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }


    if ((ret = AndroidBitmap_getInfo(env, bitmapedges, &infoedges)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }



    LOGI("gray image :: width is %d; height is %d; stride is %d; format is %d;flags is
%d",infogray.width,infogray.height,infogray.stride,infogray.format,infogray.flags);
    if (infogray.format != ANDROID_BITMAP_FORMAT_A_8) {
        LOGE("Bitmap format is not A_8 !");
        return;
    }

    LOGI("color image :: width is %d; height is %d; stride is %d; format is %d;flags is
%d",infoedges.width,infoedges.height,infoedges.stride,infoedges.format,infoedges.flags);
    if (infoedges.format != ANDROID_BITMAP_FORMAT_A_8) {
        LOGE("Bitmap format is not A_8 !");
        return;
    }



    if ((ret = AndroidBitmap_lockPixels(env, bitmapgray, &pixelsgray)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    if ((ret = AndroidBitmap_lockPixels(env, bitmapedges, &pixelsedge)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }
    // modify pixels with image processing algorithm


    LOGI("time to modify pixels....");

    graydata = (uint8_t *) pixelsgray;
    edgedata = (uint8_t *) pixelsedge;

    for (y=0;y<=infogray.height - 1;y++) {
        for (x=0;x<infogray.width -1;x++) {
            sumX = 0;
            sumY = 0;
            // check boundaries
            if (y==0 || y == infogray.height-1) {
                sum = 0;
            } else if (x == 0 || x == infogray.width -1) {
                sum = 0;
            } else {
                // calc X gradient
                for (i=-1;i<=1;i++) {
                    for (j=-1;j<=1;j++) {
                        sumX += (int) ( (*(graydata + x + i + (y + j)
* infogray.stride)) * Gx[i+1][j+1]);
                    }
                }

                // calc Y gradient
                for (i=-1;i<=1;i++) {
                    for (j=-1;j<=1;j++) {
                        sumY += (int) ( (*(graydata + x + i + (y + j)
* infogray.stride)) * Gy[i+1][j+1]);
                    }
                }

                sum = abs(sumX) + abs(sumY);

            }

            if (sum>255) sum = 255;
            if (sum<0) sum = 0;

            *(edgedata + x + y*infogray.width) = 255 - (uint8_t) sum;



        }
    }

    AndroidBitmap_unlockPixels(env, bitmapgray);
    AndroidBitmap_unlockPixels(env, bitmapedges);


}
```

The `findEdges` routine shares much in common with the prior two functions:

1. Like the `convertToGray` function, this function takes two bitmap parameters, but in this case, both are grayscale.

2. The bitmaps are interrogated to ensure that they are of the expected format.

3. The bitmap pixels are locked and unlocked appropriately.

4. The algorithm iterates over the source image's rows and columns.

Unlike the prior two functions, this function compares each pixel to the pixels around it, rather than simply performing a mathematical operation on the pixel value itself. The algorithm implemented in this function is a variant of the Sobel Edge Detection algorithm. In this implementation, I am comparing each pixel to its neighbors with a border of one pixel in each direction. Variants of this and other algorithms may use larger "borders" to obtain different results. Comparing each pixel to its neighbors accentuates the contrast between pixels and in doing so highlights the "edges."

I am not going to go into the math involved in this algorithm for two reasons. First, it is beyond the scope of this tutorial to care about the math itself. And second, the exact purpose of this tutorial — to (re)use existing C source code — is demonstrated by using an existing image processing algorithm. You are able to obtain the desired results without reinventing the wheel or having to port this code to Java technology. C is an ideal environment for working with image data, thanks to pointer arithmetic.

For more information about image processing algorithms, please see Resources.

Section 6. Customizing Eclipse

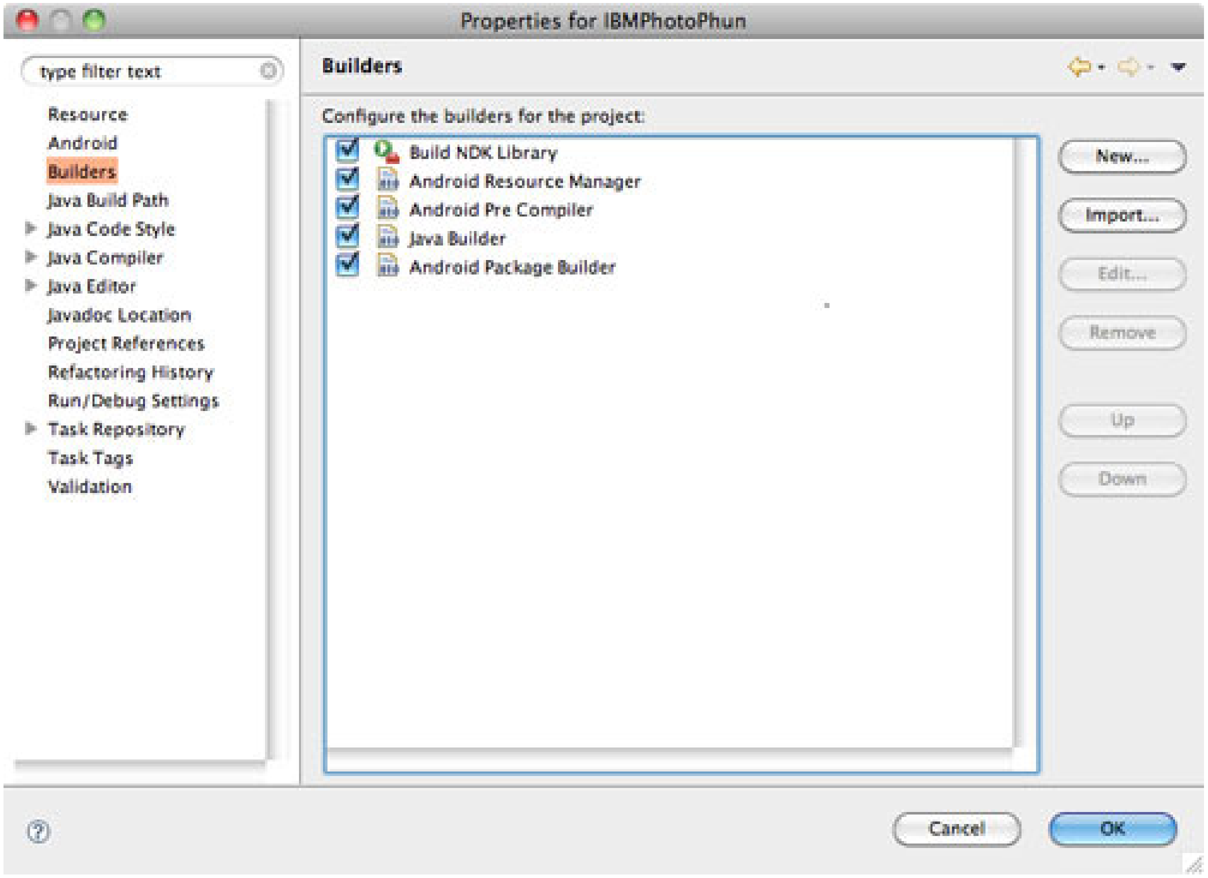
One of the enjoyable aspects of working with the Eclipse IDE is that you rarely have to compile. Anytime you save a file within the Eclipse IDE, your project is built automatically. That is great for the Android SDK (that is, Java) files and the Android XML files, but what about the NDK-built library? Let's find out.

Extending the Eclipse environment

As mentioned, building the native library is as simple as running the `ndk-build` command. However, when working with a project for anything other than a trivial exercise, it is a hassle to jump out to a terminal or command window and execute the `ndk-build` command, return to the Eclipse environment, and force a refresh by "touching" one of the project files, which forces a recompile and repackaging of the complete application. The solution is to extend the Eclipse environment by customizing the build settings for your Android project.

To modify the build settings, first view the properties of the Android project and select **Builders** in the list. Add a new Builder and move it to the top of the list, as shown in Figure 8.

Figure 8. Modifying the build settings



Properties for IBMPhotoPhun

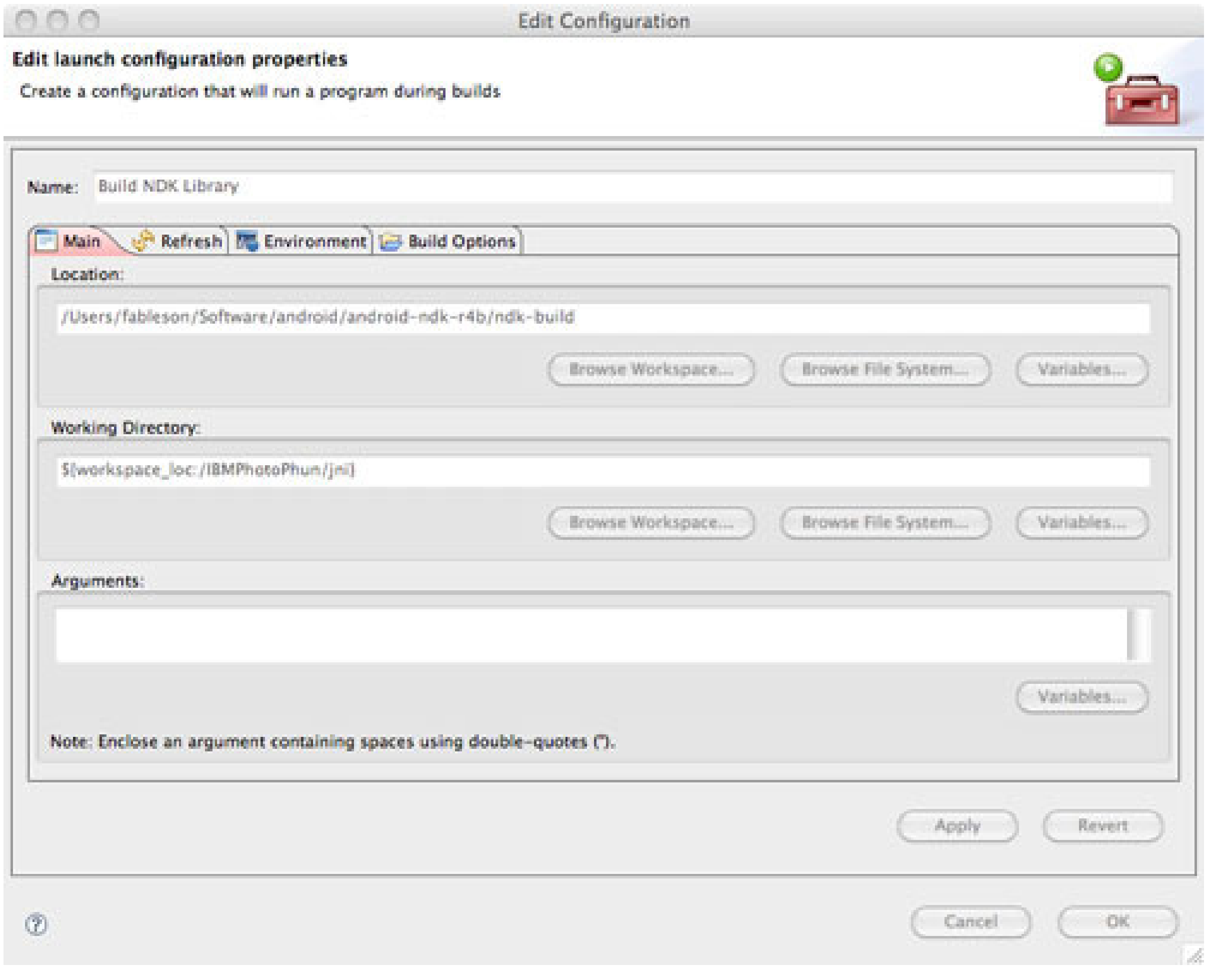
Builders

Configure the builders for the project:

- [✓] Build NDK Library
- [✓] Android Resource Manager
- [✓] Android Pre Compiler
- [✓] Java Builder
- [✓] Android Package Builder

Each builder has four configuration tabs. Give your builder a name, such as Build NDK Library, then populate the tabs. The first tab ("Main") specifies the executable tool location and working directory. Point the location to the ndk-build file and the working directory to your jni folder, as shown in Figure 9.

Figure 9. Setting up the Builder properties for the NDK


Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: Build NDK Library

Location:

/Users/fableson/Software/android/android-ndk-r4b/ndk-build

Working Directory:

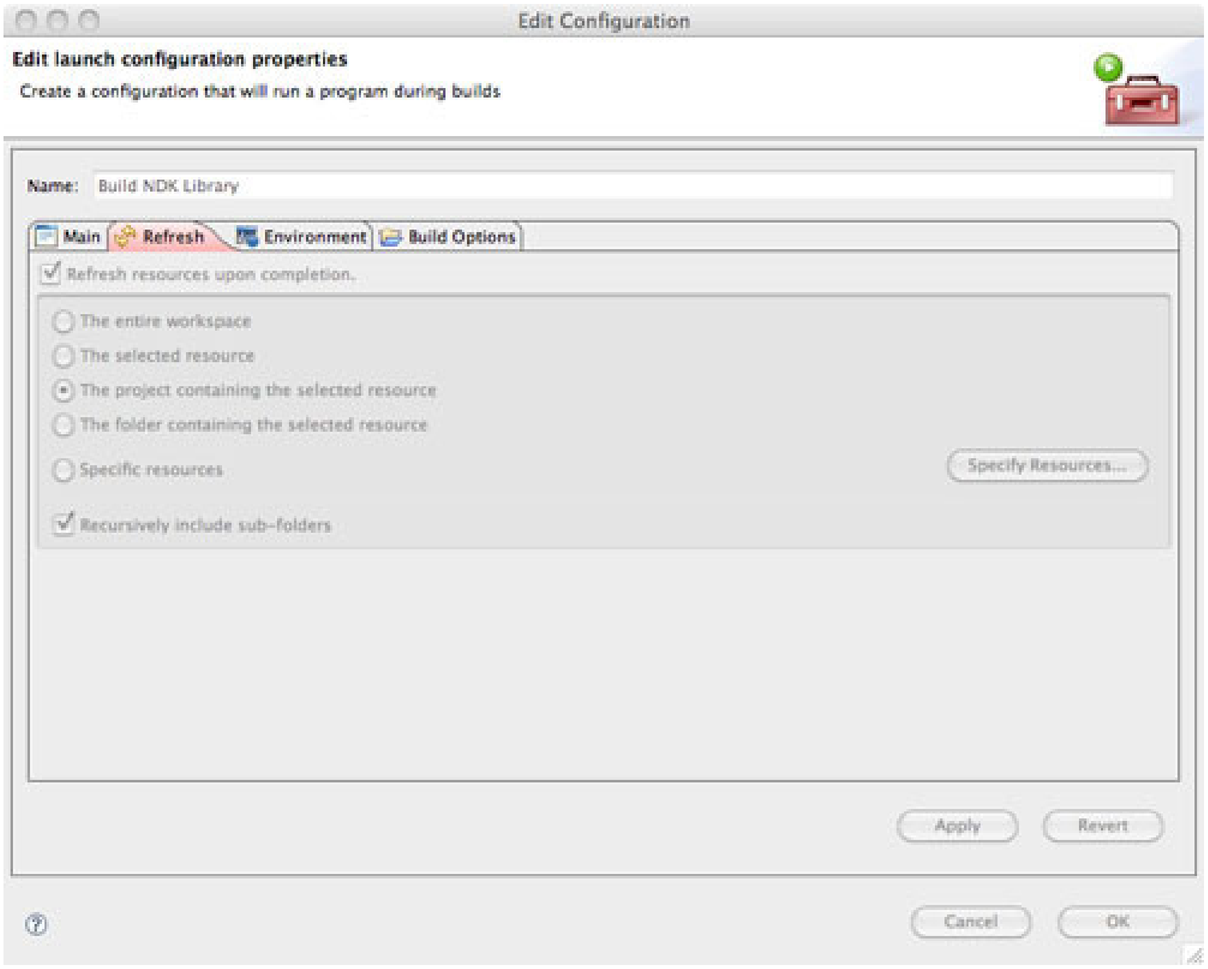
${workspace_loc:/IBMPhotoPhun/jni}

Arguments:

Note: Enclose an argument containing spaces using double-quotes (").

You only want the ndk-build to operate on this project and not others within your Eclipse workspace, so set this up on the Refresh tab, as shown in Figure 10.

Figure 10. Setting up the Refresh tab


Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: Build NDK Library

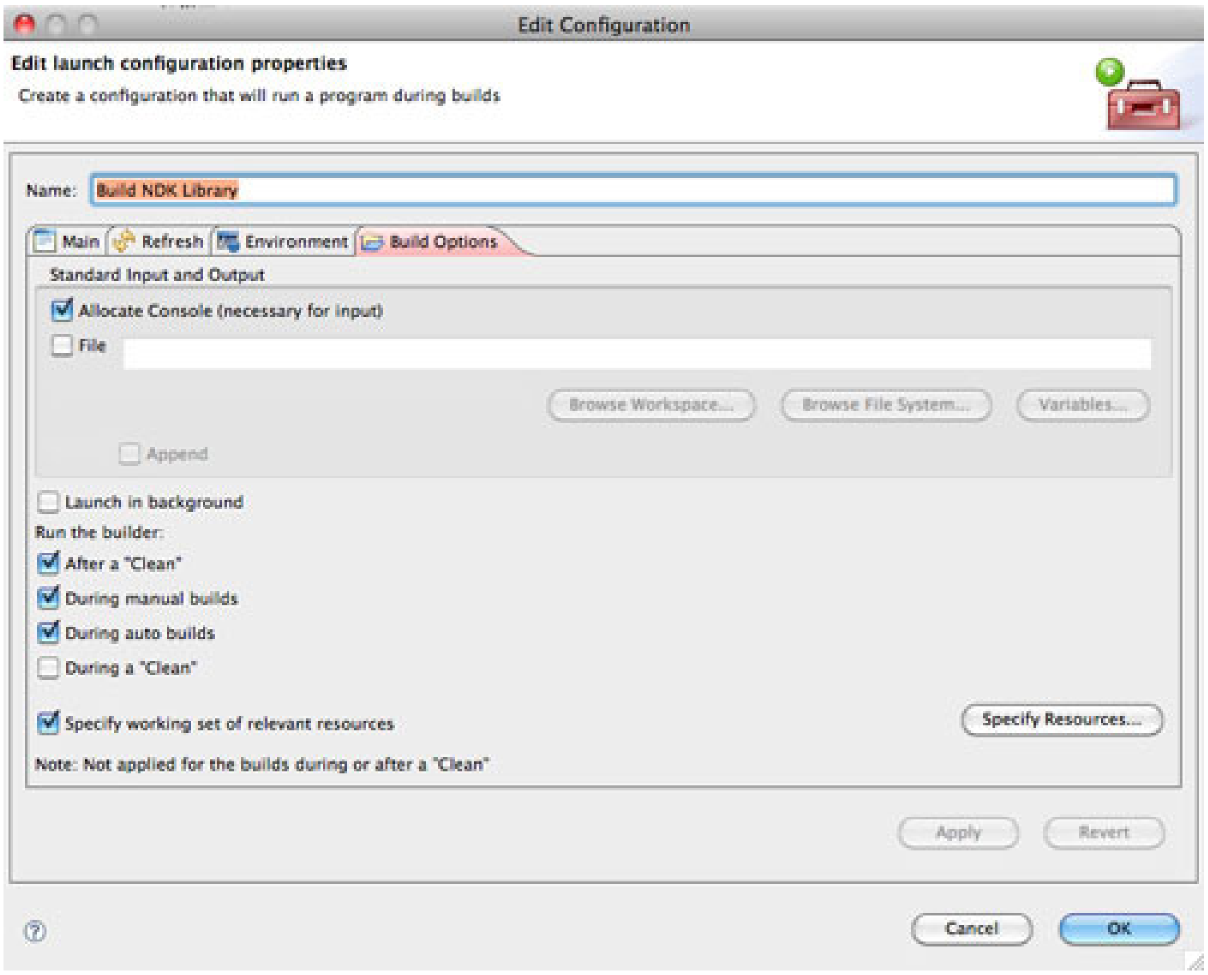
[✓] Refresh resources upon completion.

( ) The entire workspace
( ) The selected resource
(●) The project containing the selected resource
( ) The folder containing the selected resource
( ) Specific resources

[✓] Recursively include sub-folders

The only time you want the library to rebuild is when either the Android.mk file or the ibmphotophun.c file is modified. To set this up, choose the jni folder under the **Specify Resources** button on the Build Options tab. Also, specify when you want the build tool to run by checking off the appropriate times, as shown in Figure 10.

Figure 11. Setting up the build options



Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: Build NDK Library

Main | Refresh | Environment | Build Options

Standard Input and Output

- [✓] Allocate Console (necessary for input)
- [ ] File
- [ ] Append

- [ ] Launch in background

Run the builder:

- [✓] After a "Clean"
- [✓] During manual builds
- [✓] During auto builds
- [ ] During a "Clean"

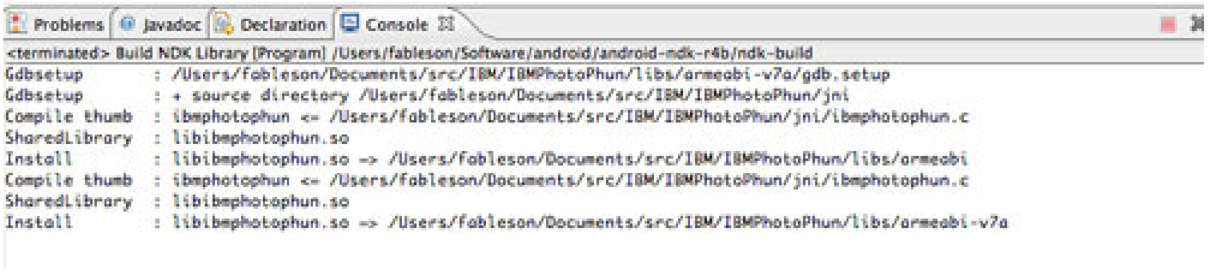
- [✓] Specify working set of relevant resources

Note: Not applied for the builds during or after a "Clean"

After clicking **OK** to confirm your settings, make sure that this NDK build tool is set up as the first entry in the list by selecting the **Up** button until it is at the top of the list of Builders, as shown in Figure 7.

To test that your Builder is set up properly, open the ibmphotophun.c source file within Eclipse by right-clicking on the source file and choose to open it with the Text Editor. Make a simple change and save the file. You should see the NDK tool-chain output in the console window, as shown in Figure 11. If your C code has errors they are shown in red.

Figure 12. NDK output shows in the console of the Eclipse IDE



```
<terminated> Build NDK Library [Program] /Users/fableson/Software/android/android-ndk-r4b/ndk-build
Gdbsetup       : /Users/fableson/Documents/src/IBM/IBMPhotoPhun/libs/armeabi-v7a/gdb.setup
Gdbsetup       : + source directory /Users/fableson/Documents/src/IBM/IBMPhotoPhun/jni
Compile thumb  : ibmphotophun <= /Users/fableson/Documents/src/IBM/IBMPhotoPhun/jni/ibmphotophun.c
SharedLibrary  : libibmphotophun.so
Install        : libibmphotophun.so => /Users/fableson/Documents/src/IBM/IBMPhotoPhun/libs/armeabi
Compile thumb  : ibmphotophun <= /Users/fableson/Documents/src/IBM/IBMPhotoPhun/jni/ibmphotophun.c
SharedLibrary  : libibmphotophun.so
Install        : libibmphotophun.so => /Users/fableson/Documents/src/IBM/IBMPhotoPhun/libs/armeabi-v7a
```

With the NDK stitched into your build process, you can focus on writing code and not concern yourself so much with the build environment. Need to make a change to the application logic? No problem, modify the Java code and save the file. Need to tweak the image processing algorithm? No worries, just modify the C routine and save the file. Eclipse and the ADT plug-in takes care of the rest for you.

Section 7. Summary

This tutorial presented an example of using the Android NDK to incorporate functionality in the C programming language. The functions used here represent a sampling of open source/public domain image processing algorithms. In a similar manner, any valid C code compatible for the Android platform may be utilized with the help of the Android NDK. In addition to the mechanics of using the NDK within Eclipse, you also learned some fundamental concepts around image processing.

Downloads

Description	Name	Size	Download method
Source code	os-androidndk-IBMPhotoPhun.source.zip	99.5KB	HTTP

Information about download methods

Resources

Learn

- Read "Writing C Code for Android," by Frank Ableson, at Linux Magazine.
- Learn more by reading the Edge Detection Tutorial, by Bill Green.
- *Image Processing in C,* by Dwayne Phillips, R&D Publications (ISBN 0-13-104548-2) offers more image processing information.
- See the Wikipedia page on Java Native Interface (JNI).
- Check out NDK at http://developer.android.com.
- Read "Wiring Up Android Buttons" from Linux Magazine to learn how to handle button presses in Android.
- Learn about developing Android applications using Eclipse in "Develop Android applications with Eclipse."
- Read "Networking with Android" to explore the networking capabilities of Android.
- Check out "Working with XML on Android" to learn about the different options for working with XML on Android and how to use them to build your own Android applications.
- *Android in Action, 2nd Edition,* by Frank Ableson, covers all aspects of Android development book.
- *Mobile Design and Development,* by Brian Fling, discusses practical guidelines, standards, techniques, and best practices for building mobile products.
- Get the Android SDK documentation.
- Visit the Open Handset Alliance, Android's sponsor.
- Read "Introduction to Android development" to get started developing Android applications.
- Read "Under the Hood of Native Web Apps for Android" at Linux Magazine, by Frank Ableson.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Stay current with developerWorks' Technical events and webcasts.
- Follow developerWorks on Twitter.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products, as well as our most popular articles and tutorials.
- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

- Download the Android SDK, access the API reference, and get the latest news on Android from the official Android developers' site. V1.5 and later will work. This tutorial used Android SDK V8, which supports the Android release labeled 2.2 (Froyo).
- Download the Android NDK. The NDK release used in this tutorial is r4b.
- Android is open source, which means that you can get the source code for it from the Android Open Source Project.
- Obtain the latest Eclipse IDE. V3.4.2 was used in this tutorial.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Frank Ableson

W. Frank Ableson is an entrepreneur living in northern New Jersey with his wife Nikki and their children. His professional interests include mobile software and embedded design. He is the author of *Unlocking Android* (Manning Publications, 2009) and *Android in Action* (Manning Publications, 2011) and he is the mobile editor for Linux Magazine.