Community Experience Distilled

Android Application Programming with OpenCV 3

Build Android apps to capture, manipulate, and track objects in 2D and 3D

Joseph Howse

[PACKT] open source community experience distilled
PUBLISHING

Android Application Programming with OpenCV 3

Build Android apps to capture, manipulate, and track objects in 2D and 3D

Joseph Howse

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Android Application Programming with OpenCV 3

First published: September 2013

Second Edition: June 2015

Production reference: 1230615

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-538-7

www.packtpub.com

Credits

Author
Joseph Howse

Reviewers
Jared Burrows
Arjun Comar
Manav Kedia
Yati Sagade

Commissioning Editor
Veena Pagare

Acquisition Editor
Vivek Anantharaman

Content Development Editor
Parita Khedekar

Technical Editor
Ryan Kochery

Copy Editor
Dipti Mankame

Project Coordinator
Milton Dsouza

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Joseph Howse lives in Canada. During the cold winters, he grows a beard and his four cats grow thick coats of fur. He combs the cats every day. Sometimes, the cats pull his beard.

Joseph has authored *OpenCV for Secret Agents*, *OpenCV Android Application Programming*, and *OpenCV Computer Vision with Python*. When he is not writing books or grooming cats, Joseph provides consulting, training, and software development services. His company is Nummist Media (`http://nummist.com`).

I dedicate my work to Sam, Jan, Bob, Bunny, and the cats, who have been my lifelong guides and companions.

OpenCV Android Application Programming is now in its second edition. I am indebted to all the editors and technical reviewers who have contributed to planning, polishing, and marketing both the editions of the book. These people have guided me with their experience and have saved me from sundry errors and omissions. Please meet the second edition's technical reviewers by reading their biographies here!

I want to thank the readers and everybody else at Packt Publishing and the OpenCV communities. We have done so much together and our journey continues!

About the Reviewers

Jared Burrows started working on Android development in 2011 when he got his first smartphone. He learned Java quickly and started putting applications on Google Play (the Android market). During this time, he was in a college and was interning at Northrop Grumman; currently, he works there as a full-time software engineer.

As his programming skills have matured through the years, he has produced 1-2 new apps each year and constantly remains active on websites such as StackOverflow, developing a good reputation and helping others. When he bought a Google Glass back in 2013, the first thing he created with an open source repository on GitHub named OpenQuartz, and he has worked on implementing OpenCV into a few example applications with Google Glass.

His applications on Google Play are available at `https://play.google.com/store/apps/developer?id=Burrows+Apps`.

His example applications with Google Glass + OpenCV are available at `https://github.com/jaredsburrows/OpenQuartz`.

Manav Kedia is a final-year undergraduate student of the department of Computer Science and Engineering at the Indian Institute of Technology, Kharagpur. He has interned with Adobe Research Labs, Bengaluru, and ETH Zurich previously. He is a passionate programmer and software enthusiast. Android application development is his forte among other things, in which he has bagged laurels from various hackathons organized by IBM and Shephertz. He is proficient in programming languages such as C++, Java, Python, AngularJS, and MySQL. He always ventures into new stuff. You can reach him at `manavkedia1993@gmail.com`.

I would like to thank the author for this brilliantly written book. Reviewing this book was a great learning experience. I would like to thank Milton Dsouza for coordinating with me throughout the review. I would also like to thank my parents and friends for supporting me in everything I do.

Yati Sagade is a programmer interested in, and working on, problems around image analysis and computer vision. He has developed several computer vision apps on the Android platform, including a work-in-progress "Air Piano" app named Dirac, which along with his other projects can be found at `https://github.com/yati-sagade/`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book will show you how to use OpenCV in an Android app that displays a camera feed, saves and shares photos, manipulates colors and edges, and tracks real-world objects in 2D or 3D. Integration with OpenGL is also introduced so that you can start building augmented reality (AR) apps that superimpose virtual 3D scenes onto tracked objects in the camera feed.

OpenCV is an open-source, cross-platform library that provides building blocks for computer vision experiments and applications. It offers high-level interfaces to capture, process, and present image data. For example, it abstracts away details about camera hardware and array allocation. OpenCV is widely used in both academia and industry.

Android is a mobile operating system that is mostly open source. For Java developers, it offers a high-level application framework called Android SDK. Android apps are modular insofar as they have standard, high-level interfaces to launch each other and share data. Mobility, a high level of abstraction, and data sharing are great starting points for a photo sharing app, similar to the one we will build.

Although OpenCV and Android provide a lot of high-level abstractions (and a lot of open source code for curious users to browse), they are not necessarily easy to use for newcomers. Setting up an appropriate development environment and translating the libraries' broad functionality into application features are both daunting tasks. This concise book helps us by placing an emphasis on a clean setup, clean application design, and a simple understanding of each function's purpose.

The need for a book on this subject is particularly great because OpenCV's Java and Android bindings are quite new and their documentation is not yet mature. Little has been written about the steps for integrating OpenCV with Android's standard camera, media, and graphics APIs. Surely, integration is a major part of an app developer's work, so it is a major focus of this book.

By the end of our journey together, you will have a taste of the breadth of application features that are made possible by integrating OpenCV with other Android libraries. You will have your own small library of reusable classes that you can extend or modify for your future computer vision projects. You will have a development environment and the knowledge to use it, and you will be able to make more apps!

What this book covers

Chapter 1, *Setting Up OpenCV*, covers the steps to set up OpenCV and an Android development environment, including Eclipse and Android SDK.

Chapter 2, *Working with Camera Frames*, shows how to integrate OpenCV into an Android Java app that can preview, capture, save, and share photos.

Chapter 3, *Applying Image Effects*, explores OpenCV's functionality to manipulate color channels and neighborhoods of pixels. The Apache Commons Math library is also introduced. We expand our app to include channel-mixing filters, "curve" filters, and a filter that darkens edges.

Chapter 4, *Recognizing and Tracking Images*, demonstrates the steps to recognize and track a known target (such as a painting) when it appears in a video feed. We expand our app so that it draws an outline around any tracked target.

Chapter 5, *Combining Image Tracking with 3D Rendering*, improves upon our previous tracking technique by determining the target's position and rotation in real 3D space. We expand our app so that it sets up an OpenGL 3D scene with the same perspective as the Android device's real camera. Then, we draw a 3D cube atop any tracked target.

Chapter 6, *Mixing Java and C++ via JNI*, demonstrates the use of Java Native Interface (JNI) to call C++ functions from Java. We convert some of our application's filters to C++ in order to learn about writing efficient, cross-platform code with OpenCV's C++ interface.

What you need for this book

This book provides setup instructions for OpenCV and an Android development environment, including Eclipse and Android SDK. The software is cross-platform, and the instructions cover Windows, Mac, and Linux. Other Unix-like environments may work, too, if you are willing to do your own tailoring of the setup steps.

You need a mobile device running Android 2.2 (Froyo) or greater, and it must have a camera. Preferably, it should have two cameras, front and rear.

Who this book is for

This book is great for Java developers who are new to computer vision and who like to learn through application development. It is assumed that you have previous experience in Java but not necessarily Android. A basic understanding of image data (for example, pixels and color channels) would be helpful, too.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "It will interface with the other apps on the device, via Android's `MediaStore` and `Intent` classes."

A block of code is set as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
  "http://schemas.android.com/apk/res/android"
  package="com.nummist.secondsight"
  android:versionCode="1"
  android:versionName="1.0">
```

When we wish to draw our attention to a particular part of a code block, the relevant lines or items are set in bold:

```
android:label="@string/app_name"
android:screenOrientation="landscape">
<intent-filter>
```

Any command-line input or output is written as follows:

```
$ cd /etc/udev/rules.d/
$ sudo touch 51-android.rules
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "These steps should be repeated for all the native (C++) projects, which include **OpenCV Sample – face-detection** and **OpenCV Tutorial 2 - Mixed Processing**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let's know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com` and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/B04598_Graphics.pdf`.

Errata

Although we took every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of the copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem. You can also contact the author directly at `josephhowse@nummist.com` or you can check his website, `http://nummist.com/opencv/`, for answers to common questions about the book.

1
Setting Up OpenCV

This chapter is a quick guide to setting up a development environment for Android and OpenCV. We will also look at the OpenCV sample applications, documentation, and community.

By the end of this chapter, our development environment will include the following components:

- **Java Development Kit (JDK) 7**: This includes tools for Java programming. JDK 7 is the exact version that we require. The more recent version, JDK 8, is not yet supported for Android development.
- **Cygwin 1.7 or greater (Windows only)**: This is a compatibility layer that provides Unix-like programming tools on Windows. We need it in order to develop in C++ on Android.
- **Android Software Development Kit (Android SDK) r24.0.2 or greater**: This includes tools for programming Android apps in Java.
- **Android Native Development Kit (Android NDK) r10d or greater**: This includes tools for programming Android apps in C++.
- **Eclipse 4.4.2 (Luna) or greater**: This is an **integrated development environment** (**IDE**). Although Google has started to recommend Android Studio as an IDE for Android development, Eclipse is still supported too. The OpenCV library and official samples are preconfigured as Eclipse projects, so for our purposes, Eclipse is a bit more convenient than Android Studio.
- **Java Development Tools (JDT)**: This is an Eclipse plugin for Java programming (already included in most Eclipse distributions).
- **C/C++ Development Tooling (CDT) 8.2.0 or greater**: This is an Eclipse plugin for C/C++ programming.

- **Android Development Tools (ADT) 24.0.2 or greater**: This is an Eclipse plugin for Android programming.
- **OpenCV4Android 3.0 or greater**: This is OpenCV's Android version, including Java and C++ libraries.

At the time of writing, OpenCV4Android's latest version is 3.0. This book targets version 3.0, but it also includes comprehensive notes on the differences between OpenCV 3.x and OpenCV 2.x. The author's website, `http://nummist.com/opencv`, offers two sets of code bundles: one for OpenCV 3.x (tested with 3.0) and another for OpenCV 2.x (tested with 2.4.9).

There are many possible ways to install and configure these components. We will cover several common setup scenarios, but if you are interested in further options, see OpenCV's official documentation at `http://docs.opencv.org/doc/tutorials/introduction/android_binary_package/O4A_SDK.html`.

System requirements

All the development tools for Android and OpenCV are cross-platform. The following operating systems are supported with almost identical setup procedures:

- Windows XP or a later version
- Mac OS 10.6 (Snow Leopard) or a later version
- Debian Wheezy or a later version, including derivatives such as Ubuntu 12.04 (Pangolin) or a later version
- Many other Unix-like systems (though not specifically covered in this book)

To run the OpenCV samples and, later on, our own application, we should have an Android device with the following specifications:

- Android 2.2 (Froyo) or greater (required)
- Camera (required): front and rear cameras (recommended)
- Autofocus (recommended)

Android Virtual Devices (**AVD**s) are not recommended. Some parts of OpenCV rely on low-level camera access and might fail with virtualized cameras.

Setting up a development environment

We are going to install various components of a development environment separately and configure them to work together. Broadly, this task has the following two stages:

1. Set up a general-purpose Android development environment.
2. Set up OpenCV for use in this environment. We may use a prepackaged, preconfigured version of OpenCV, or alternatively, we may configure and build OpenCV from source.

Let's start by looking at the setup steps for a general-purpose Android development environment. We will not delve into very much detail here because good instructions are available at the given links and, as an Android or Java developer, you have probably been through similar steps before.

If you already have an Android development environment or another Java development environment and you just want to add components to it, some of the following steps might not apply to you.

Here are the steps:

1. Download and install Oracle JDK 7 from `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`. Alternatively, on Debian or Ubuntu, install Oracle JDK 7 from the WebUpd8 PPA, as described at `https://launchpad.net/~webupd8team/+archive/ubuntu/java`. Although most Linux distributions have OpenJDK in their standard repositories, Oracle JDK is recommended instead for Android development.
2. Download Eclipse and unzip it to any destination, which we will refer to as `<eclipse>`. Many up-to-date Eclipse distributions are available at `http://www.eclipse.org/downloads/`. Of these, Eclipse IDE for Java Developers is a good choice as a foundation for an Android development environment.
3. We now need to set up Android SDK and the ADT plugin for Eclipse. Go to `http://developer.android.com/sdk/index.html#Other` and download **SDK Tools Only**. Install or unzip it to any destination, which we will refer to as `<android_sdk>`. Open Eclipse and install the ADT plugin according to the official instructions at `http://developer.android.com/sdk/installing/installing-adt.html`. Restart Eclipse. A window, **Welcome to Android Development**, should appear. Click on **Use Existing SDKs**, browse to `<android_sdk>`, and click on **Next**. Close Eclipse.

4. From the Eclipse menu system, navigate to **Windows | Android SDK Manager**. Select and install additional SDK packages according to the official instructions at `https://developer.android.com/sdk/installing/adding-packages.html`. In particular, we will need the most recent versions of the following packages: the latest Android API, such as Android 5.1.1 (API 22), Android SDK Tools, Android SDK Platform-tools, Android SDK Build Tools, and Android Support Library. After the packages are installed, close Eclipse.
5. If we are using Windows, download and install Cygwin from `http://cygwin.com/install.html`.
6. Download Android NDK from `http://developer.android.com/tools/sdk/ndk/index.html`. Unzip it to any destination, which we will refer to as `<android_ndk>`.
7. Edit your system's `Path` (in Windows) or `PATH` (in Mac, Linux, or other Unix-like systems) to include `<android_sdk>/platform-tools`, `<android_sdk>/tools`, and `<android_ndk>`. Also, create an environment variable named `NDKROOT` and set its value to `<android_ndk>`. (If you are unsure how to edit `Path`, `PATH` or other environment variables, see the tips in the boxes on this page and next page.)

Editing environment variables on Windows

The system's `Path` variable and other environment variables can be edited in the **Environment Variables** window of the **Control Panel**.

On Windows Vista/7/8, open the **Start** menu and launch the **Control Panel**. Now, go to **System and Security | System | Advanced system settings**. Click on the **Environment Variables** button.

On Windows XP, open the **Start** menu and go to **Control Panel | System**. Click on the **Advanced** tab. Click on the **Environment Variables** button.

Now, under **System variables**, select an existing environment variable, such as `Path`, and click on the **Edit** button. Alternatively, make a new environment variable by clicking on the **New** button. Edit the variable's name and value as needed. For example, if we want to add `C:\android-sdk\platform-tools` and `C:\android-sdk\tools` to `Path`, we should append `;C:\android-sdk\platform-tools;C:\android-sdk\tools` to the existing value of `Path`. Note the use of semicolons as separators.

To apply the changes, click on all the **OK** buttons until we are back in the main window of the **Control Panel**. Now, log out and log in again.

Editing environment variables on Mac

Edit `~/.profile`.

To append to an existing environment variable in `~/.profile`, add a line such as `export PATH=$PATH:~/android-sdk/platform-tools:~/android-sdk/tools`. This example appends `~/android-sdk/platform-tools` and `~/android-sdk/tools` to `PATH`. Note the use of colons as separators.

To create a new environment variable in `~/.profile`, add a line such as `export NDKROOT=~/android-ndk`.

Save your changes, log out, and log in again.

Editing environment variables on Linux

Edit either `~/.profile` (as described previously for Mac) or `~/.pam_environment` (as described next). Note that `~/.profile` and `~/.pam_environment` use slightly different formats for variables.

To append to an existing environment variable in `~/.pam_environment`, add a line such as `PATH DEFAULT=${PATH}:~/android-sdk/platform-tools:~/android-sdk/tools`. This example appends `~/android-sdk/platform-tools` and `~/android-sdk/tools` to `PATH`. Note the use of colons as separators.

To create a new environment variable in `~/.pam_environment.`, add a line such as `NDKROOT DEFAULT=~/android-ndk`.

Save your changes, log out, and log in again.

Now we have an Android development environment, but we still need OpenCV. We may choose to download a prebuilt version of OpenCV, or we may build it from source. These options are discussed in the following two subsections.

Getting prebuilt OpenCV4Android

The prebuilt versions of OpenCV4Android can be downloaded from `http://sourceforge.net/projects/opencvlibrary/files/opencv-android/`. Look for files that have `opencv-android` in the name, such as `OpenCV-3.0.0-android-sdk.zip` (the latest version at the time of writing). Download the latest version and unzip it to any destination, which we will refer to as `<opencv>`.

Building OpenCV4Android from source

Alternatively, the process for building OpenCV4Android from **trunk** (the latest, unstable source code) is documented at `http://code.opencv.org/projects/opencv/wiki/Building_OpenCV4Android_from_trunk`. For a summary of the process, continue reading this section. Otherwise, skip ahead to *Building the OpenCV samples with Eclipse*, later in this chapter.

Since trunk contains the latest, unstable source code, there is no guarantee that the build process will succeed. You may need to do your own troubleshooting if you want to build from trunk.

To build OpenCV from source, we need the following additional software:

- **Git**: This is a **Source Control Management** (**SCM**) tool, which we will use to obtain OpenCV's source code. On Windows or Mac, download and install Git from `http://git-scm.com/`. On Linux, install it using your package manager. For example, on Debian or Ubuntu, open Terminal and run `$ sudo apt-get install git-core`.
- **CMake**: This is a set of build tools. On Windows or Mac, download and install CMake from `http://www.cmake.org/cmake/resources/software.html`. On Linux, install it using your package manager. For example, on Debian or Ubuntu, open Terminal and run `$ sudo apt-get install cmake`.
- **Apache Ant 1.8.0 or greater**: This is a set of build tools for Java. On Linux, just install Ant using your package manager. For example, on Debian or Ubuntu, open Terminal and run `$ sudo apt-get install ant`. On Windows or Mac, download Ant from `http://ant.apache.org/bindownload.cgi` and unzip it to any destination, which we will refer to as `<ant>`. Make the following changes to your environment variables:
  - Add `<ant>/bin` to `Path` (Windows) or `PATH` (Unix).
  - Create a variable `ANT_HOME` with the value `<ant>`.
- **Python 2.6 or greater (but not 3.0 or greater)**: This is a scripting language that is used by some of the OpenCV build scripts. An appropriate version of Python comes preinstalled on Mac and most Linux systems, including Debian and Ubuntu. On Windows, download and install Python from `http://www.python.org/getit/`. If you have installed multiple versions of Python on your system, ensure that an installation of Python 2.6 or greater (but not 3.0 or greater) is the only one in `Path` (Windows) or `PATH` (Unix). The OpenCV build scripts do not run properly with Python 3.0 or greater.

Once we have these prerequisites, we can download the OpenCV source code to any location, which we will refer to as `<opencv_source>`. Then, we can build it using an included script. Specifically, we should take the following steps:

On Windows, open Git Bash (Git's command prompt). On Mac, Debian, Ubuntu, or other Unix-like systems, open Terminal (or another command-line shell).

Run these commands:

```
$ git clone git://code.opencv.org/opencv.git <opencv_source>
$ cd <opencv_source>/platforms
$ sh ./scripts/cmake_android_arm.sh
$ cd build_android_arm
$ make -j8
```

The `-j8` flag specifies that the `make` command will use 8 threads, which is typically a good number for a quad-core processor. For a dual-core processor, a better choice might be the `-j4` flag (4 threads).

If all goes well, we should get a build of OpenCV4Android in `<opencv_source>/platforms/build_android_arm`. We can move it elsewhere if we wish. We will refer to its final location as `<opencv>`.

You might wonder what the `cmake_android_arm.sh` build script is doing. Actually, it just creates a build directory and runs a CMake command to populate the directory with a particular configuration of OpenCV. Here are the entire contents of the script file:

```
#!/bin/sh
cd `dirname $0`/..

mkdir -p build_android_arm
cd build_android_arm

cmake -DCMAKE_BUILD_WITH_INSTALL_RPATH=ON -
DCMAKE_TOOLCHAIN_FILE=../android/android.toolchain.cmake $@ ../..
```

Advanced users, who are familiar with CMake, might want to copy and modify this script to create a custom configuration of OpenCV. Refer to the code in `<opencv_source>/CMakeLists.txt` for definitions of OpenCV's CMake options.

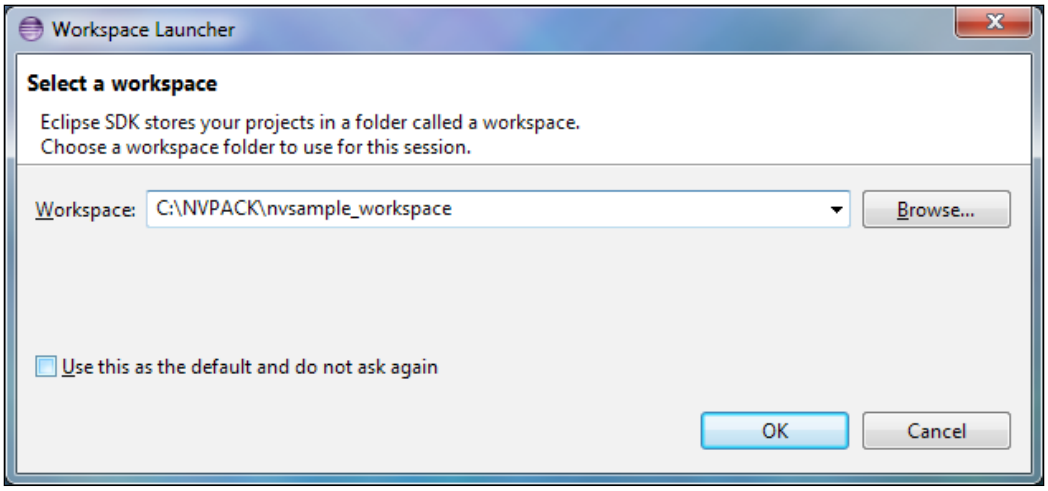
The preceding steps use the `cmake_android_arm.sh` script to produce an OpenCV4Android build for ARM, which is the architecture of most Android phones and tablets. Alternatively, you can use the `cmake_android_x86.sh` script for x86 or the `cmake_android_mips.sh` script for MIPS. Note that the name of the build directory also changes according to the architecture.

Building the OpenCV samples with Eclipse

Building and running a few sample applications is a good way to test that OpenCV is correctly set up. At the same time, we can practice using Eclipse.

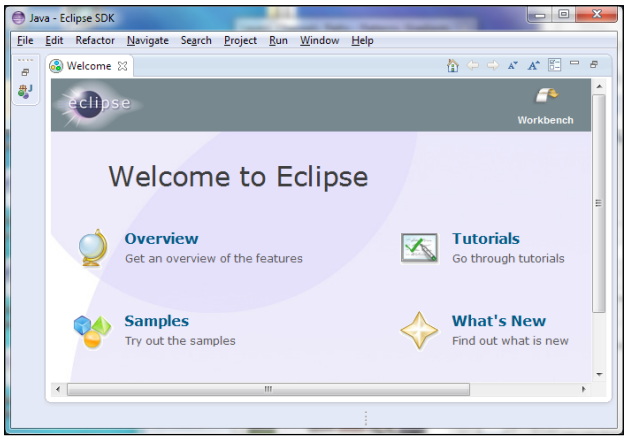
Let's start by launching Eclipse. The Eclipse launcher should be located at `<eclipse>/eclipse.exe` (Windows), `<eclipse>/Eclipse.app` (Mac), or `<eclipse>/eclipse` (Linux). Run it.

We should see a window called **Workspace Launcher**, which asks us to select a workspace. A **workspace** is the root directory for a set of related Eclipse projects. Enter any location you choose.

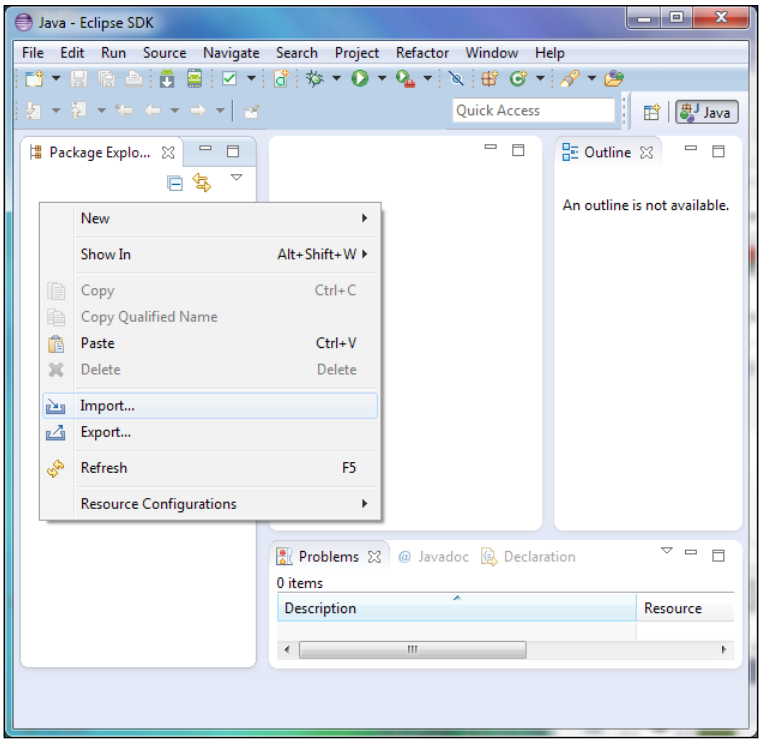


We can return to **Workspace Launcher** anytime via the menu: **File** | **Switch Workspace** | **Other...**.

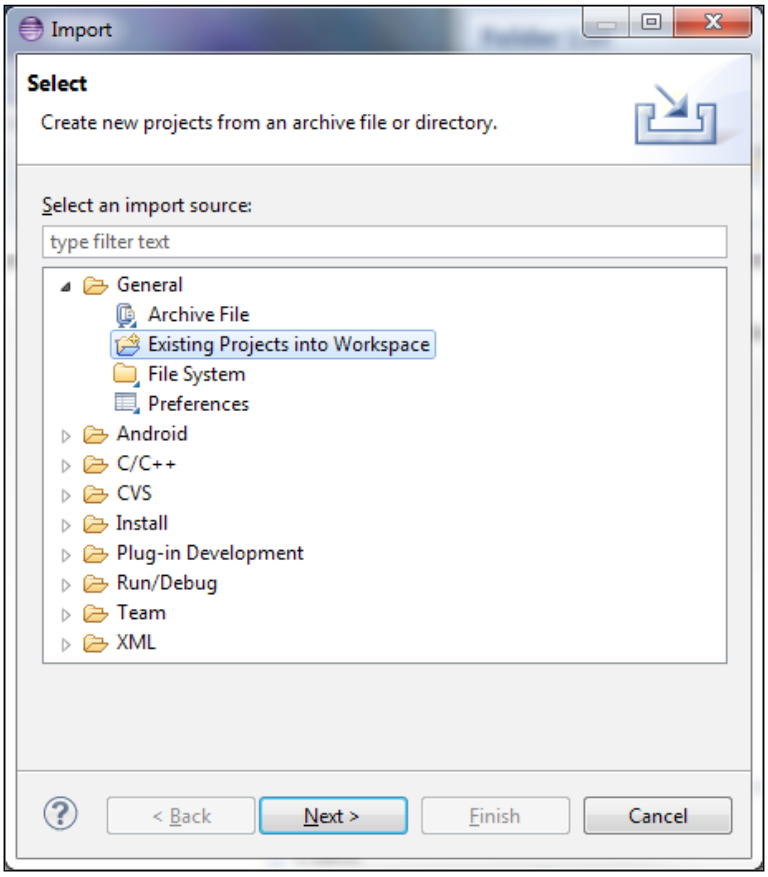
If the **Welcome to Eclipse** screen appears, click on the **Workbench** button:



Now, we should see a window with several panels, including **Package Explorer**. If we are not using TAPD, we need to import the OpenCV sample projects into our new workspace. Right-click on **Package Explorer** and select **Import...** from the context menu:



The **Import** window should appear. Navigate to **General** | **Existing Projects into Workspace**, and then click on **Next >**:

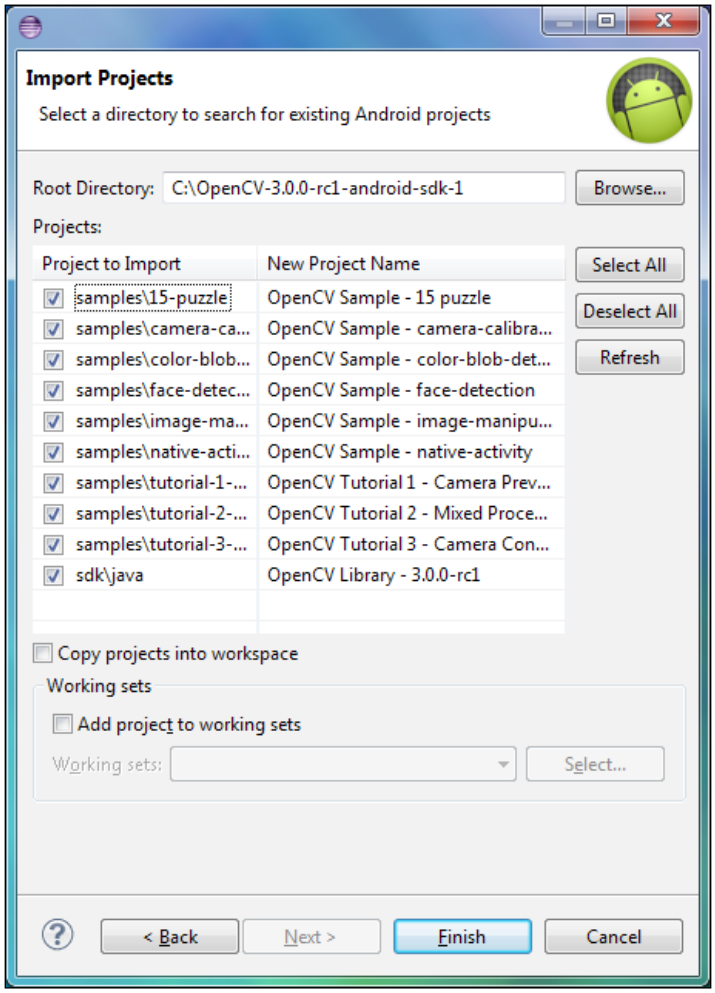


On the second page of the **Import** window enter `<opencv>` in the **Select root directory:** field. Under the **Projects:** label, a list of detected projects should appear. (If not, click on **Refresh**.) The list should include the OpenCV library, samples, and tutorials. They should all be selected by default.

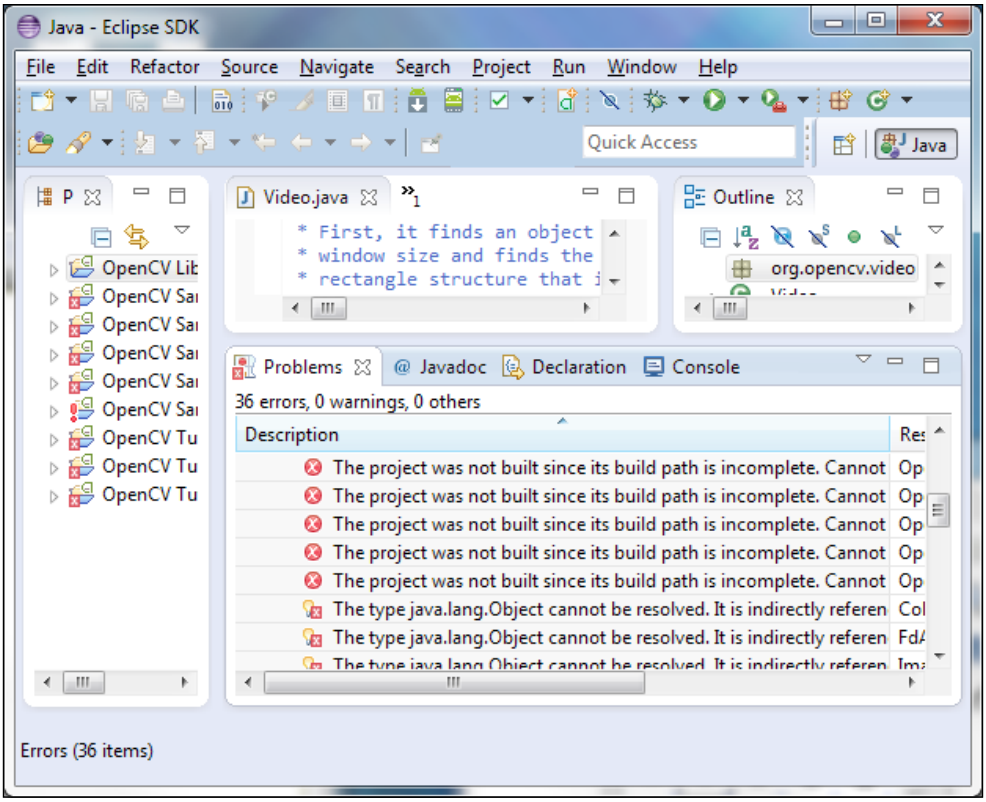
Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

This means that Eclipse has found the OpenCV library, samples, and tutorials and has recognized them as Eclipse projects. *Do not* select **Copy projects into workspace** because the OpenCV sample and tutorial projects rely on a relative path to the library project, and this relative path will not be preserved if the projects are copied into the workspace. Click on **Finish** to import the projects:



Once the projects are imported, we might need to fix some configuration issues. Our development environment might have different paths and different versions of Android SDK than the ones in the samples' default configuration:



Any resulting errors will be reported in the **Problems** tab. For likely solutions, see the section *Troubleshooting Eclipse projects*, later in this chapter.

We should first resolve any errors in the **OpenCV Library** project as the samples and tutorials depend on the library.

Once the OpenCV projects no longer show any errors, we can prepare to test them on an Android device. Recall that the device must have Android 2.2 (Froyo) or greater and a camera. To let Eclipse communicate with the device, we must enable the device's USB debugging option. On the Android device, perform the following steps:

1. Open the **Settings** app.
2. On Android 4.2 or greater, go to the **About phone** or **About tablet** section and tap **Build number** seven times. This step enables the **Developer options** section.
3. Go to the **Developer options** section (on Android 4.0 or greater) or the **Applications | Development** section (on Android 3.2 or less). Enable the **USB debugging** option.

Now, we need to install an Android app called OpenCV Manager 3, which takes care of checking for OpenCV library updates when we run any OpenCV application. At the time of writing, OpenCV Manager 3 is not yet available from the Play Store. However, in the `<opencv>/apk` folder of our development environment, we can find prebuilt application bundles (`.apk` files) for various architectures. Choose an `.apk` file whose name matches your Android device's architecture. At the time of writing, ARMv7-A is a popular architecture for Android devices. For this architecture, OpenCV 3.0 offers the `OpenCV_3.0.0_Manager_3.00_a.apk` file. Open a command prompt and enter a command, such as the following, to install the appropriate `.apk` to your Android device via USB:

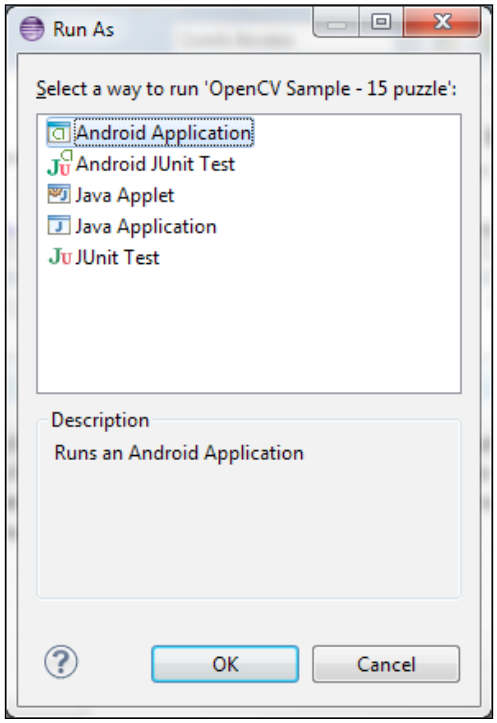
```
$ adb install <opencv>/apk/OpenCV_3.0.0_Manager_3.00_armeabi-v7a.apk
```

If the installation succeeds, the terminal should print `Success`.

Supporting OpenCV 2.x applications

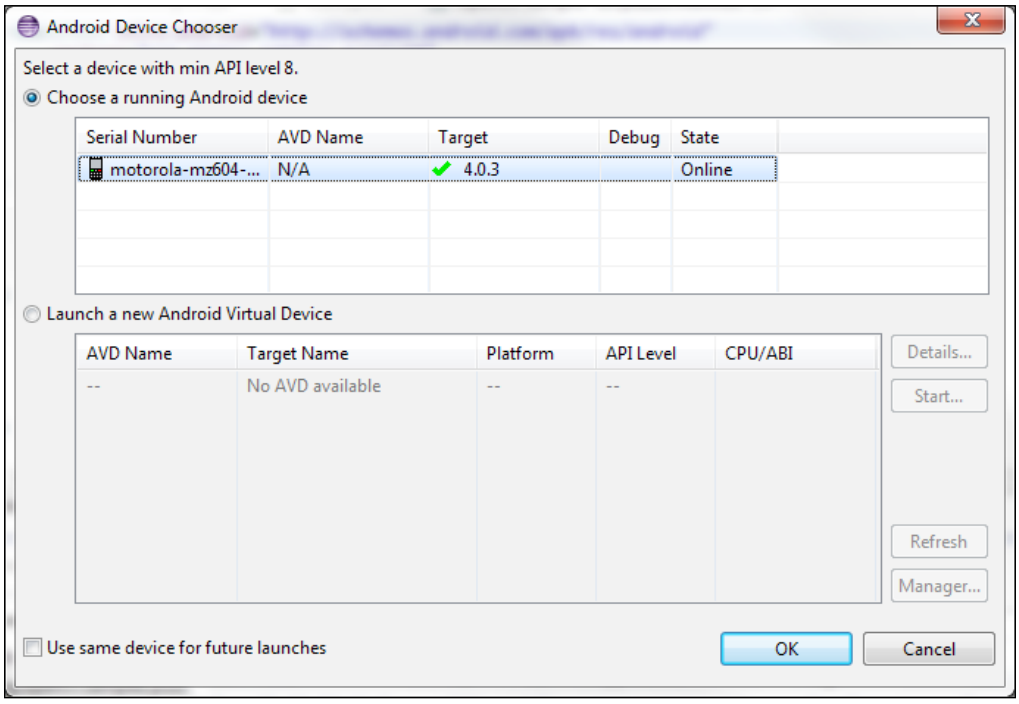
At the time of writing, the Play Store contains an older version of OpenCV Manager that supports OpenCV 2.x only. If you want to run both OpenCV 2.x and OpenCV 3.x applications, you can install this older version from the Play Store alongside OpenCV Manager 3. They do not conflict.

Plug the Android device into your computer's USB port. In Eclipse, select one of the OpenCV sample projects in **Package Explorer**. Then, from the menu system, navigate to **Run** | **Run as…** | **Android Application**:

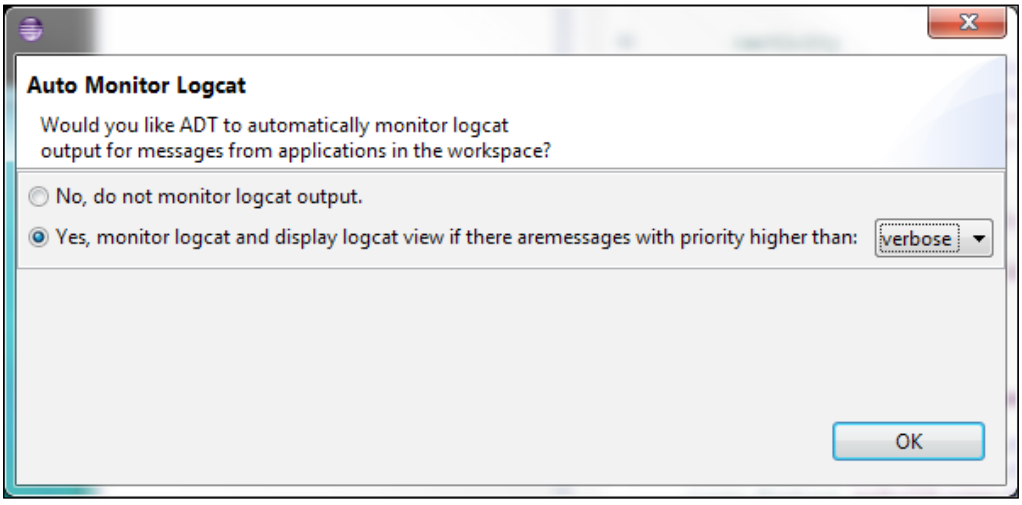


An **Android Device Chooser** window should appear. Your Android device should be listed under **Choose a running Android device**. If the device is not listed, refer to the section *Troubleshooting the USB connection*, later in this chapter.

Select the device and click on **OK**:



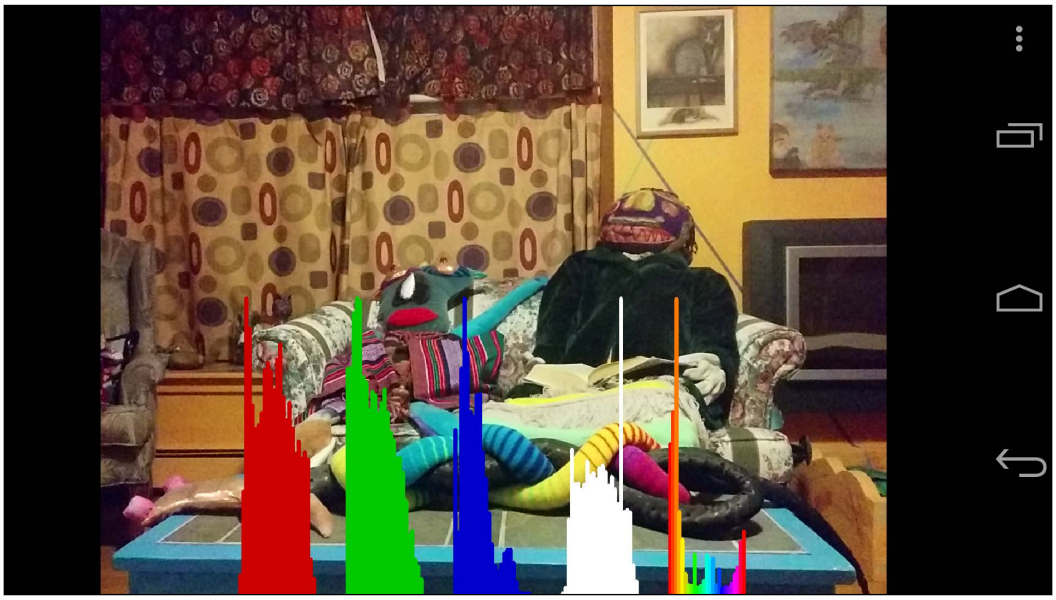
If the **Auto Monitor Logcat** window appears, select the **Yes** radio button and the **verbose** drop-down option, and click on **OK**. This option ensures that all the log output from the application will be visible in Eclipse:



On the Android device, you might get a message: **OpenCV library package was not found! Try to install it?** Make sure that the device is connected to the Internet and then touch the **Yes** button on your device. The Play Store will open to show an OpenCV package. Install the package and then press the hardware back button to return to the sample application, which should be ready for use.

For OpenCV 3.0, the samples and tutorials have the following functionality:

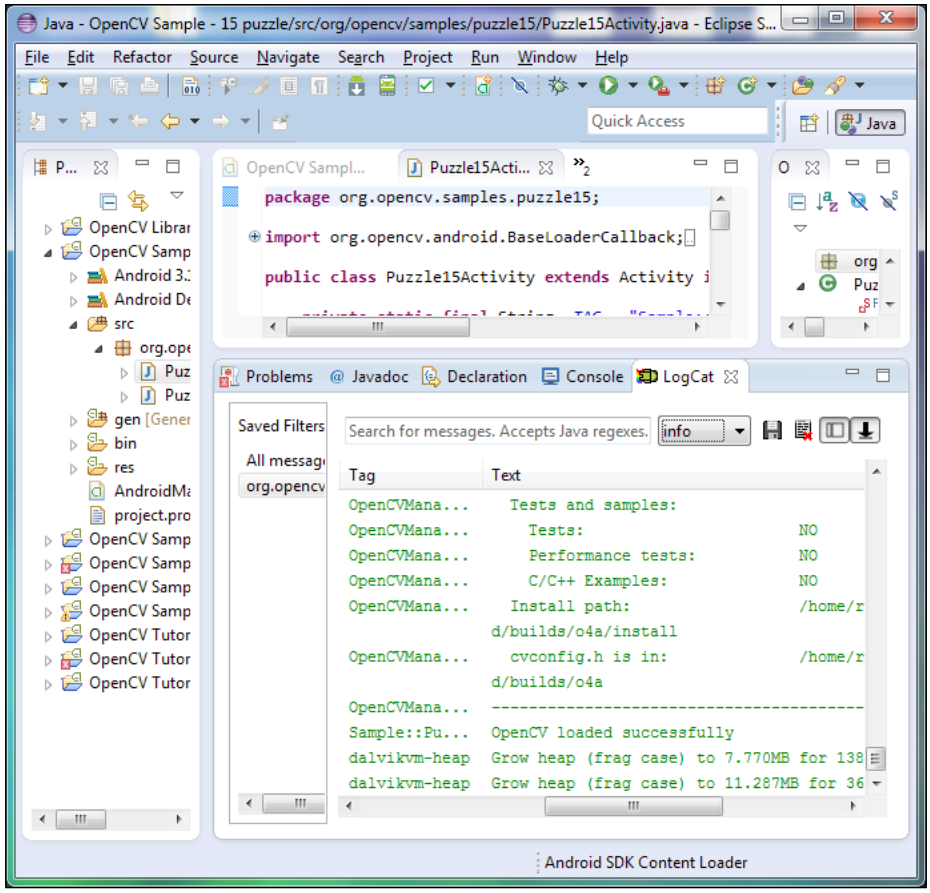
- **Sample–15 puzzle**: This splits up a camera feed to make a sliding-block puzzle. The user can swipe blocks to move them.
- **Sample–color-blob-detection**: This detects color regions in a camera feed. The user can touch anywhere to see the outline of a color region.
- **Sample–face-detection**: This draws green rectangles around faces in a camera feed.
- **Sample–image-manipulations**: This applies filters to a camera feed. The user can press the Android menu button to select from a list of filters. For example, one filter draws a color histogram (a bar chart of colors that are present in the image), as seen at the bottom of the following screenshot:



- **Sample – native-activity**: This displays a camera feed using native (C++) code.

- **Tutorial 1 – Camera Preview**: This displays a camera feed. The user can press the **...** menu to select a different camera feed implementation (Java or native C++).
- **Tutorial 2 – Mixed Processing**: This applies filters to a camera feed using native (C++) code. The user can press the **...** menu to select from a list of filters. One of the filters draws red circles around interest points or features in a camera feed. Generally speaking, interest points or features lie along the high-contrast edges in an image. They are potentially useful in image recognition and tracking applications, as we will see later in this book.
- **Tutorial 3 – Camera Control**: This applies filters to a camera feed, which has a customizable resolution. The user can press the **...** menu to select from a list of filters and a list of resolutions.

Try these applications on your Android device! While an application is running, its log output should appear in the **LogCat** tab in Eclipse:



Feel free to browse the projects' source code via **Package Explorer** to see how they were made. Alternatively, you might want to return to the official samples and tutorials later, once we have built our own application over the course of this book.

Troubleshooting Eclipse projects

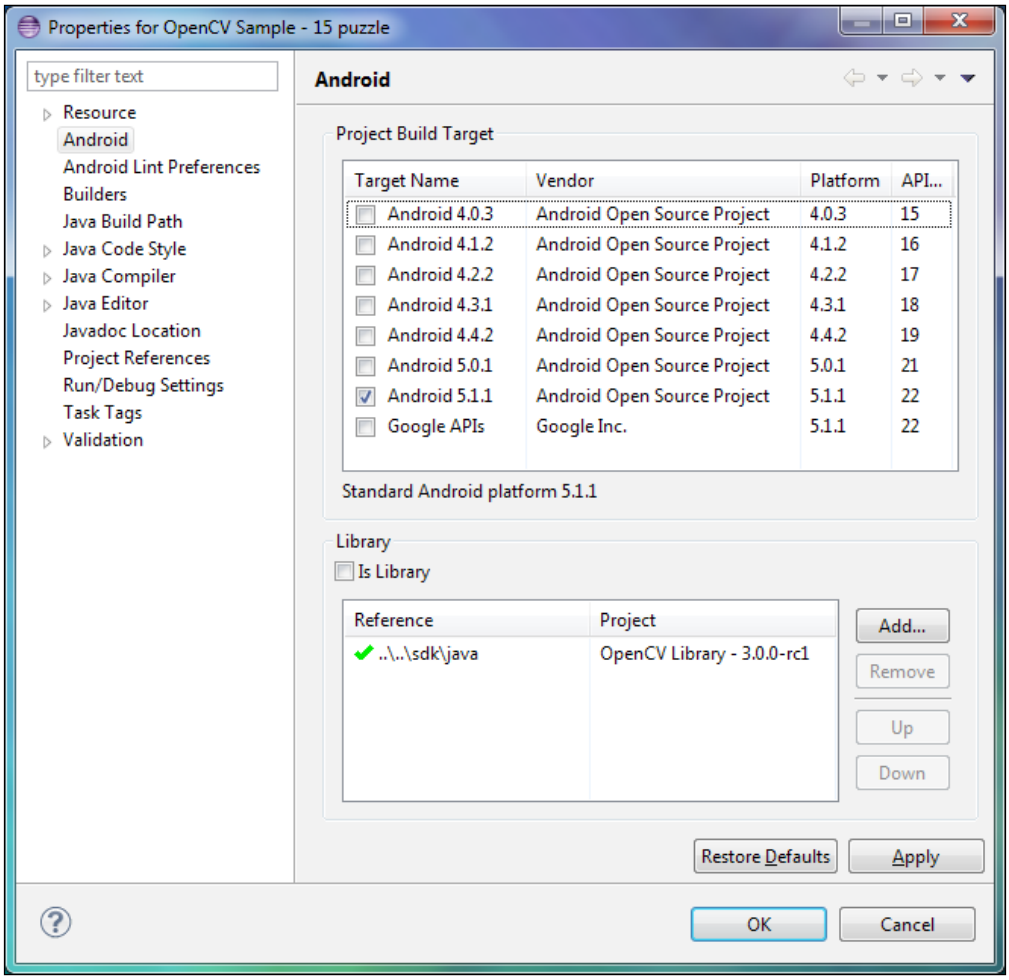
This section is not about troubleshooting Java code. Rather, it addresses a few common problems with the configuration and build process of Eclipse projects. You might encounter these problems when working with the OpenCV library, OpenCV sample projects, other imported projects, or even your own new projects.

Sometimes, Eclipse fails to recognize that a project needs to be rebuilt after the project or one of its dependencies has changed (or after a dependency has been imported). When in doubt, try cleaning all the projects by navigating to **Project** | **Clean…** | **Clean all projects** | **OK** in the menu system. This will force Eclipse to rebuild everything, thus ensuring that all errors, warnings, and successes are up-to-date.

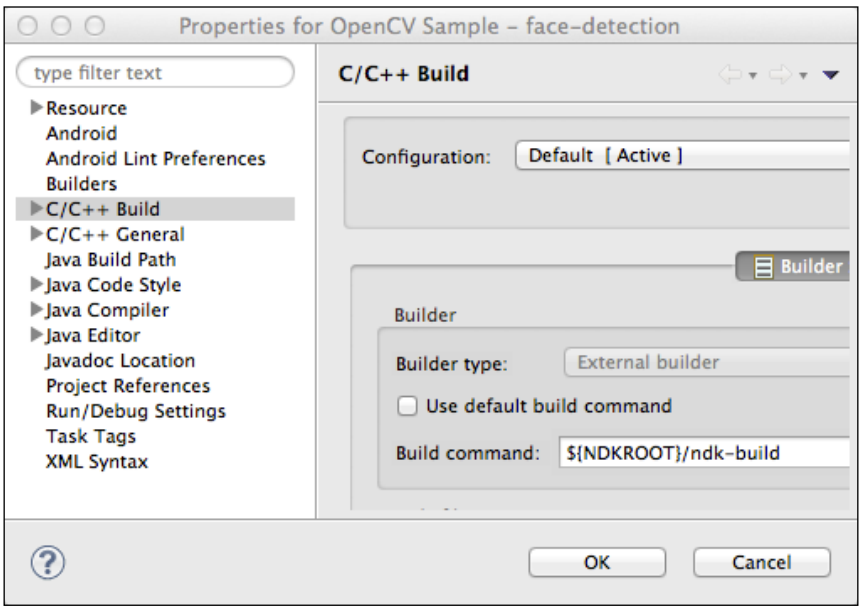
If a set of cleaned projects still has mysterious errors, then a configuration problem might be the cause.

The target Android version might not be properly specified. The symptoms are that imports from the `java` and `android` packages fail, and there are error messages such as **The project was not built since its build path is incomplete**. The solution is to right-click on the project in **Package Explorer**, select **Properties** from the context menu, select the **Android** section, and checkmark one of the available Android versions. These steps should be repeated for all the projects.

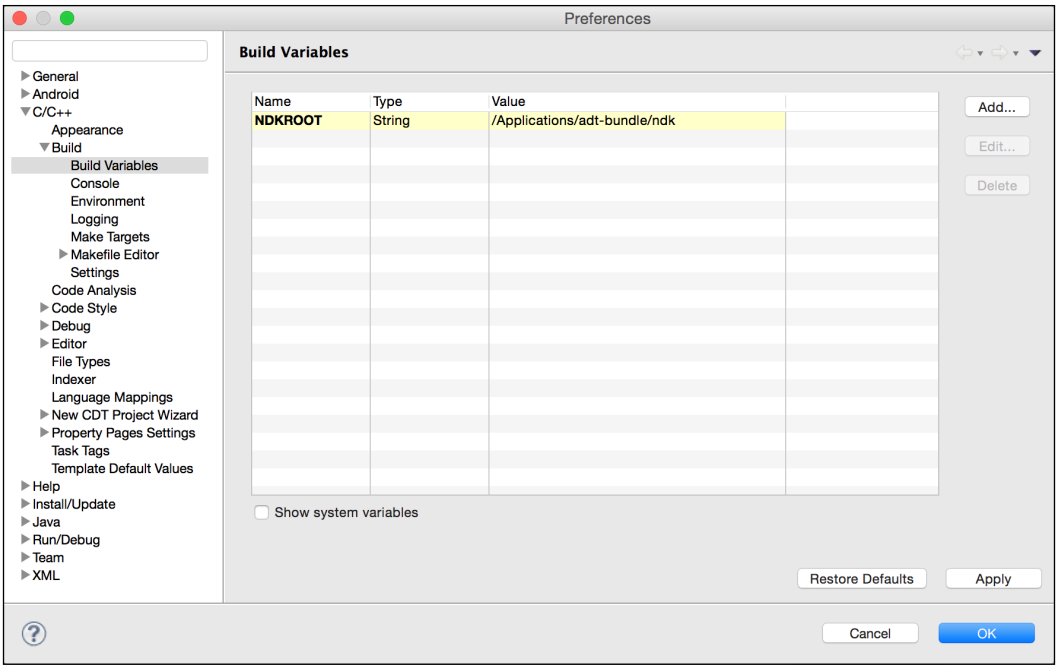
At compile time, OpenCV and its samples must target Android 3.0 (API level 11) or greater, though at runtime they also support Android 2.2 (API level 8) or greater:



If imported on Mac or Linux, the OpenCV C++ samples might be misconfigured to use the Windows build executable. The symptom is an error message such as **Program "/ndk-build.cmd" not found in PATH**. The solution is to right-click on the project in **Package Explorer**, select **Properties** from the context menu, select the **C/C++ Build** section, and edit the **Build command:** field to remove the `.cmd` extension. These steps should be repeated for all the native (C++) projects, which include **OpenCV Sample – face-detection** and **OpenCV Tutorial 2 - Mixed Processing**:



If we are still getting an error message such as **Program "/ndk-build.cmd" not found in PATH**, we can conclude that Eclipse is not recognizing the `NDKROOT` environment variable. As an alternative to relying on an environment variable, we can add `NDKROOT` as an Eclipse build variable to **Eclipse** | **Preferences** | **C/C++** | **Build** | **Build Variables**. (These preferences are shared across Eclipse projects.) As the variable's type, select **String**, and as its value, enter your NDK path (to which we previously referred as `<android_ndk>`):



Troubleshooting the USB connection

If your Android device does not appear in Eclipse's **Android Device Chooser** window or if your `adb` commands fail in the command prompt, the USB connection might be at fault. Specifically, USB communication with the Android device is controlled via a tool called **Android Debug Bridge** (**ADB**), and this tool (or some other component of the connection) might not be working as expected. Try the possible solutions in this section.

To verify whether a USB connection is working, run the following command in a command prompt:

```
$ adb devices
```

If the connection is working, the terminal should print the serial number and name of your connected Android device, such as `019d86b921300c7c device`

Many connection problems are intermittent and can be resolved by restoring the USB connection to an initial state. Try the following steps and, after each step, test whether the problem is resolved:

1. Unplug the Android device from the host computer's USB port. Then, plug it back in.
2. Disable and re-enable the device's **USB debugging** option, as described earlier in the section *Building the OpenCV samples with Eclipse*.
3. On Mac or Linux, run the following command in Terminal (or another command prompt):

```
sudo sh -c "adb kill-server && start-server"
```

Less commonly, connection problems might relate to drivers or permissions. A one-time setup process, as described next, should resolve such problems.

On Windows, we might need to manually install the USB drivers for the Android device. Different vendors and devices have different drivers. The official Android documentation provides links to the various vendors' driver download sites at `http://developer.android.com/tools/extras/oem-usb.html#Drivers`.

On Linux, before connecting an Android device via USB, we might need to specify the device's vendor in a permissions file. Each vendor has a unique ID number, as listed in the official Android documentation at `http://developer.android.com/tools/device.html#VendorIds`. We will refer to this ID number as `<vendor_id>`. To create the permissions file, open a command prompt application (such as Terminal) and run the following commands:

```
$ cd /etc/udev/rules.d/
$ sudo touch 51-android.rules
$ sudo chmod a+r 51-android-rules
```

Note that the permissions file needs to have root ownership, so we use `sudo` while creating or modifying it. Now, open the file in an editor such as gedit:

```
$ sudo gedit 51-android-rules
```

For each vendor, append a new line to the file. Each of these lines should have the following format:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="<vendor_id>", MODE="0666",
GROUP="plugdev"
```

Save the permissions file and quit the editor. Reboot.

On Mac, no special drivers or permissions are required.

Finding the documentation and help

The OpenCV Java API and C++ API are both relevant to Android. The Java API documentation is online at `http://docs.opencv.org/java/`, and an index of OpenCV4Android resources is online at `http://opencv.org/platforms/android.html`. The C++ API documentation is online at `http://docs.opencv.org/`. The following documents, which mostly use C++ code, are also available as downloadable PDF files:

- API reference: `http://docs.opencv.org/opencv2refman.pdf`
- Tutorials: `http://docs.opencv.org/opencv_tutorials.pdf`
- User guide (incomplete): `http://docs.opencv.org/opencv_user.pdf`

If the documentation does not seem to answer your question, try talking to the OpenCV community. Here are some sites where you will find helpful people:

- Official OpenCV forum: `http://www.answers.opencv.org/questions/`
- Jay Rambhia's blog: `http://jayrambhia.wordpress.com/`
- The support site for my OpenCV books: `http://nummist.com/opencv/`

Also, you can read or submit bug reports at `http://code.opencv.org/projects/opencv/issues?query_id=4`. Finally, if you need to take your issue to the highest authority, you can e-mail the OpenCV4Android developers at `android@opencv.org`.

Summary

By now, we should have an Android and OpenCV development environment that can do everything we need for the application described in this book's remaining chapters. Depending on the approach we took, we might also have a set of tools that we can use to reconfigure and rebuild OpenCV for our future needs.

We know how to build the OpenCV Android samples in Eclipse. These samples cover a different range of functionality to this book's project, but they are useful as additional learning aids. We also know where to find the documentation and help.

Now that we have the necessary tools and reference materials to hand, our first ambition as application developers is to control a camera! Throughout the next chapter, we will use Android SDK and OpenCV to preview, capture, and share photographs.

2
Working with Camera Frames

In this chapter, we will focus on building a basic photo capture app, which uses OpenCV to capture the frames of camera input. Our app will enable the user to preview, save, edit, and share photos. It will interface with the other apps on the device, via Android's `MediaStore` and `Intent` classes. Thus, we will learn how to build bridges between OpenCV and standard Android. In subsequent chapters, we will expand our app, using more functionality from OpenCV.

The complete Eclipse project for this chapter can be downloaded from the author's website. The project has two versions:

A version for OpenCV 3.x is located at `http://nummist.com/opencv/4598_02.zip`.

A version for OpenCV 2.x is located at `http://nummist.com/opencv/5206_02.zip`.

Designing our app – Second Sight

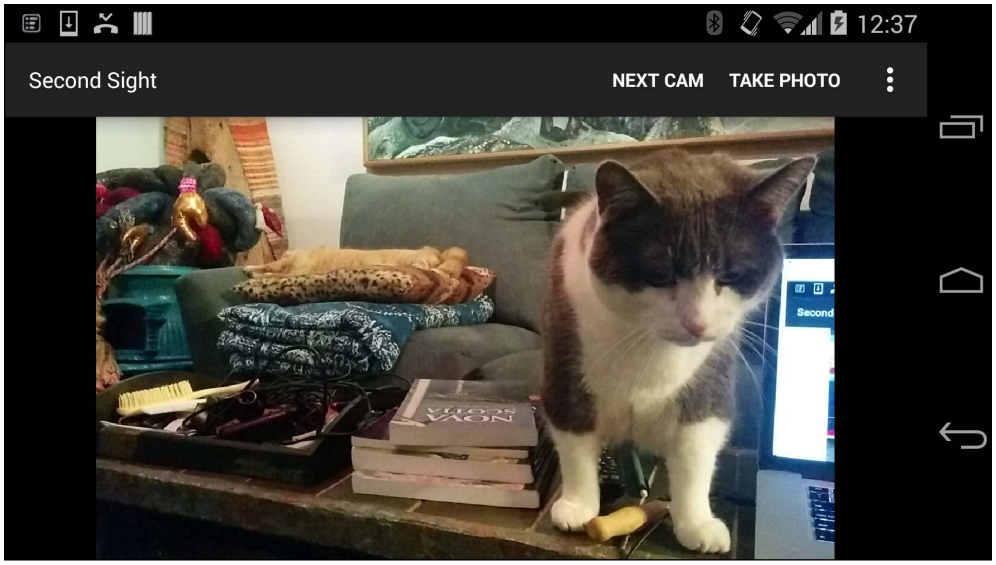
Let's make an app that enables people to see new visual patterns, animate and interact with these patterns, and share them as pictures. The idea is simple and versatile. Anyone, from a child to a computer vision expert, can appreciate the visual patterns. Through the magic of computer vision on a mobile device, any user can more readily see, change, and share hidden patterns in any scene.

For this app, I chose the name Second Sight, a phrase that is sometimes used in mythology to refer to supernatural and symbolic visions.

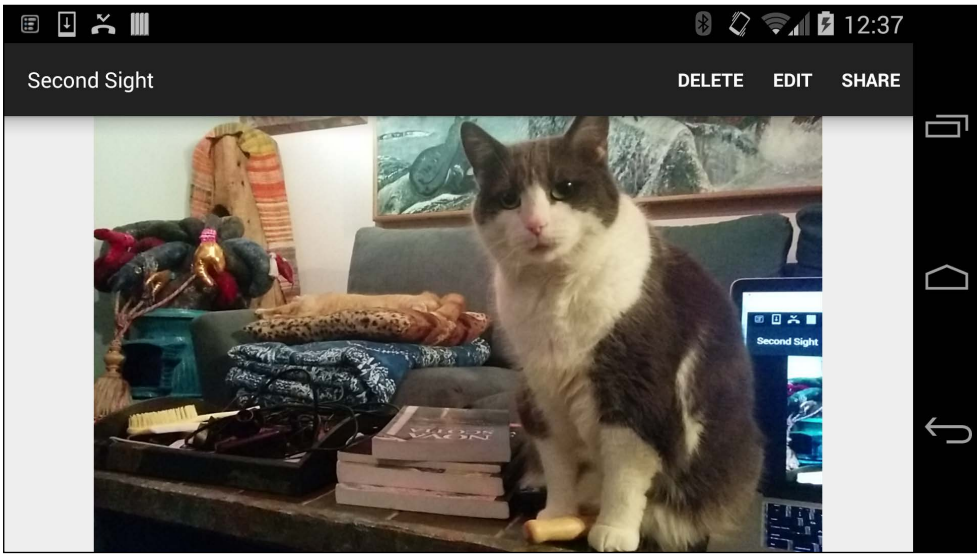
At its core, Second Sight is a camera app. It will enable the user to preview, save, and share photos. Like many other camera apps, it will also let the user apply filters to the previewed and saved photos. However, many of the filters will not be traditional photographic effects. For example, the more complex filters will enable the user to see the stylized edges or even rendered objects that blend with the real scene (**augmented reality**).

For this chapter, we will build the basic camera and sharing functions of Second Sight, without any filters. Our first version of the app will contain two activity classes named `CameraActivity` and `LabActivity`. The `CameraActivity` class will show the preview and provide menu actions so that the user may select a camera (if the device has multiple cameras), an image size (under the **...** section of the menu, if the camera supports multiple image sizes), and take a photo. Then, the `LabActivity` class will open to show the saved photo and provide the menu actions so that the user may delete the photo or send it to another app for editing or sharing.

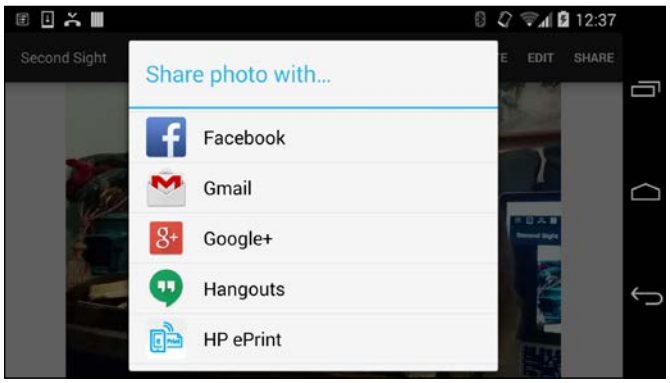
To get a better sense of our goal, let's look at some screenshots. Our first version of `CameraActivity` will appear as follows:



When the user clicks on the **Take Photo** menu item, the `LabActivity` class will be opened. It will look like the following screenshot:



When the user presses the **Share** menu item, an intent chooser (a dialog for choosing a destination app) will appear over the top of the photo, as in the following screenshot:

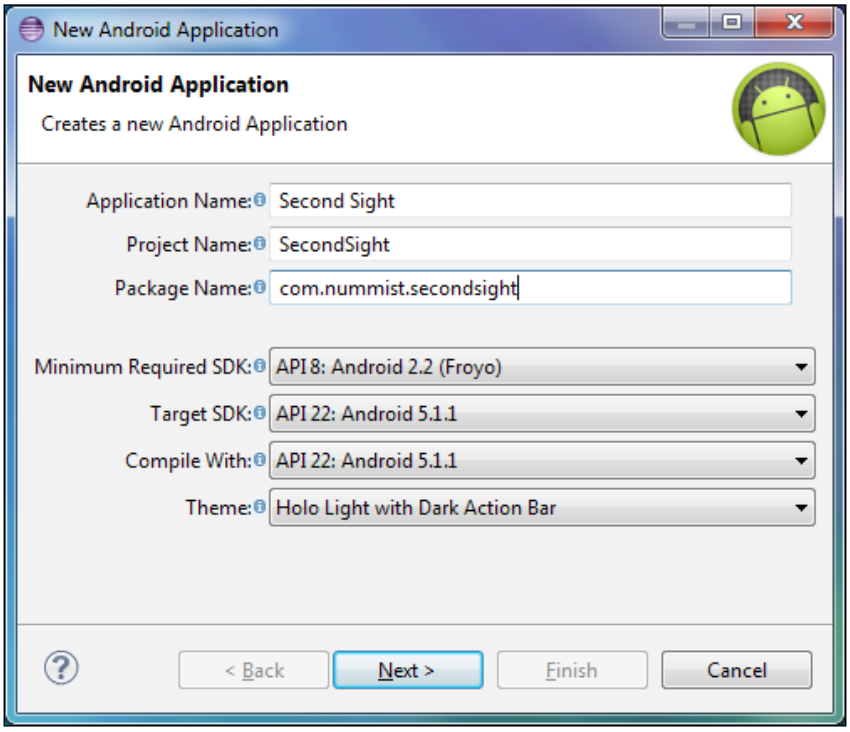


For example, by pressing the **Google+** tile, the user could open the photo in the Google+ app, in order to share it over the social network. Thus, we have a complete example of typical usage. With a few touch interactions, the user can snap a photo and share it.

Creating the Eclipse project

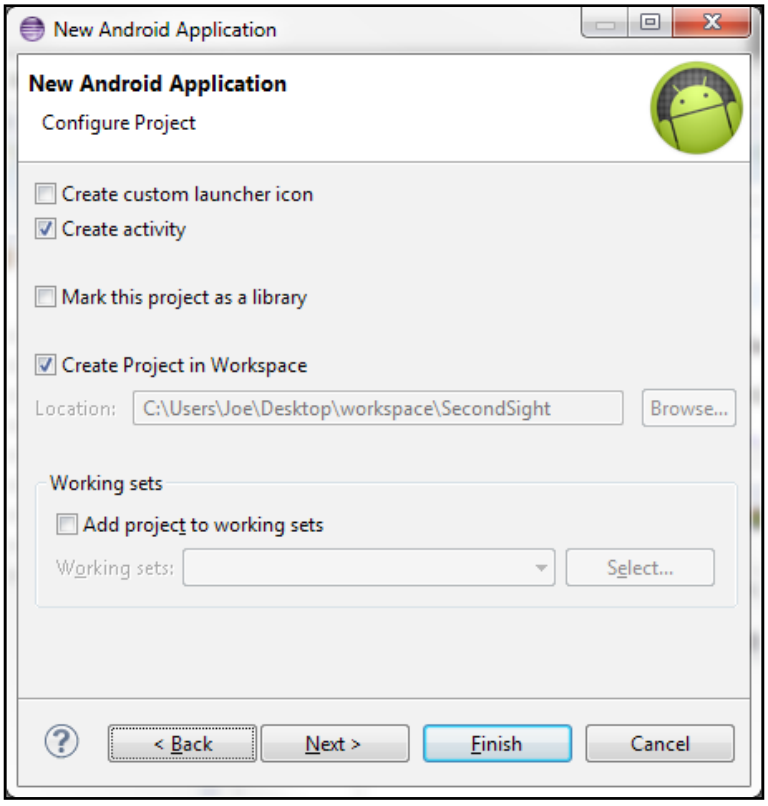
We need to create a new Eclipse project for our app. We may do this in the same workspace that we already used for the OpenCV library project and samples. Alternatively, if we use another workspace, we must import the OpenCV library project into this workspace too. (For instructions on setting the workspace and importing the library project, see the *Building the OpenCV samples with Eclipse* section of *Chapter 1*, *Setting Up OpenCV*.)

Open Eclipse in a workspace that contains the library project. Then, from the menu system, navigate to **File** | **New** | **Android Application Project**. The **New Android Application** window should appear. Enter the options that are shown in the following screenshot:

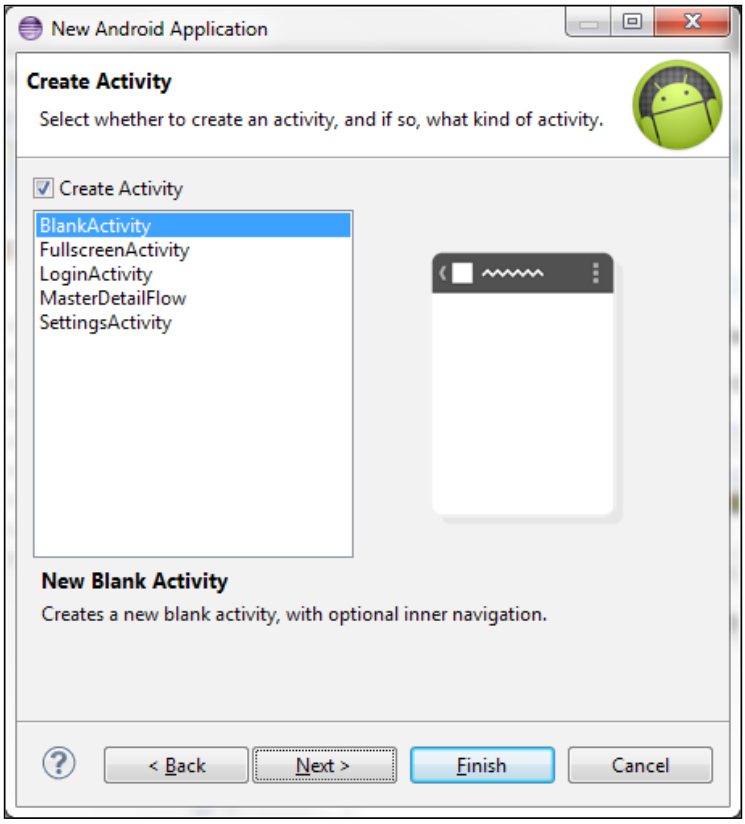


The **Target SDK** and **Compile With** fields should be set to **API 11: Android 3.0** or higher. It is safe to choose the most recent API version, which, at the time of writing, is **API 22: Android 5.1.1**. The **Minimum Required SDK** field should be left at the default, that is, **API 8: Android 2.2 (Froyo)**, because we will write fallbacks to enable our code to run on that version.

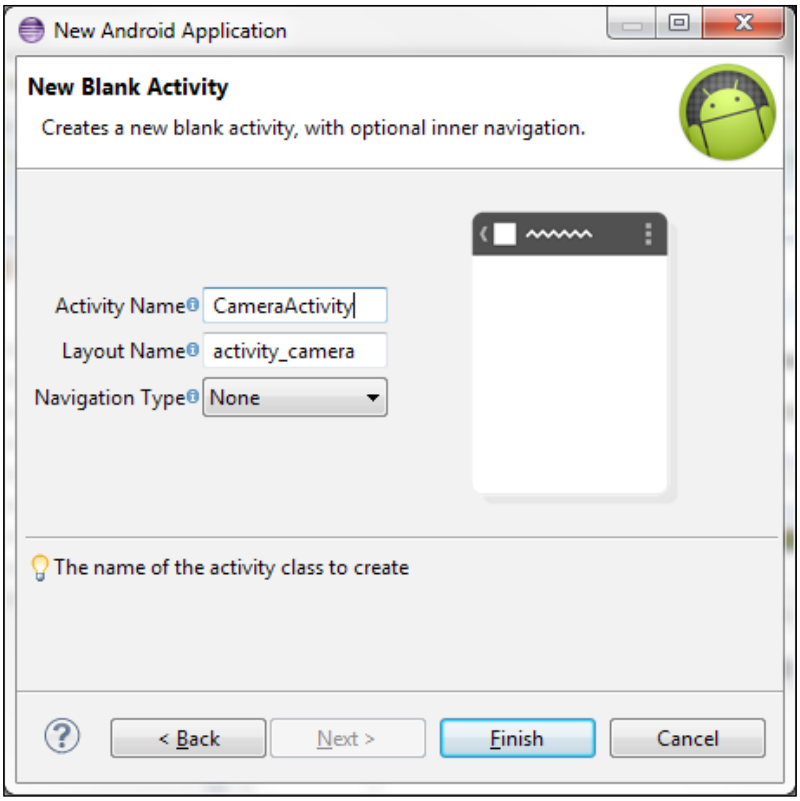
Click on the **Next** button. A checklist should appear. Ensure that the only checked options are **Create activity** and **Create Project in Workspace**, as in the following screenshot:



Click on the **Next** button. A list of activity templates should appear. Select **BlankActivity**, as in the following screenshot:



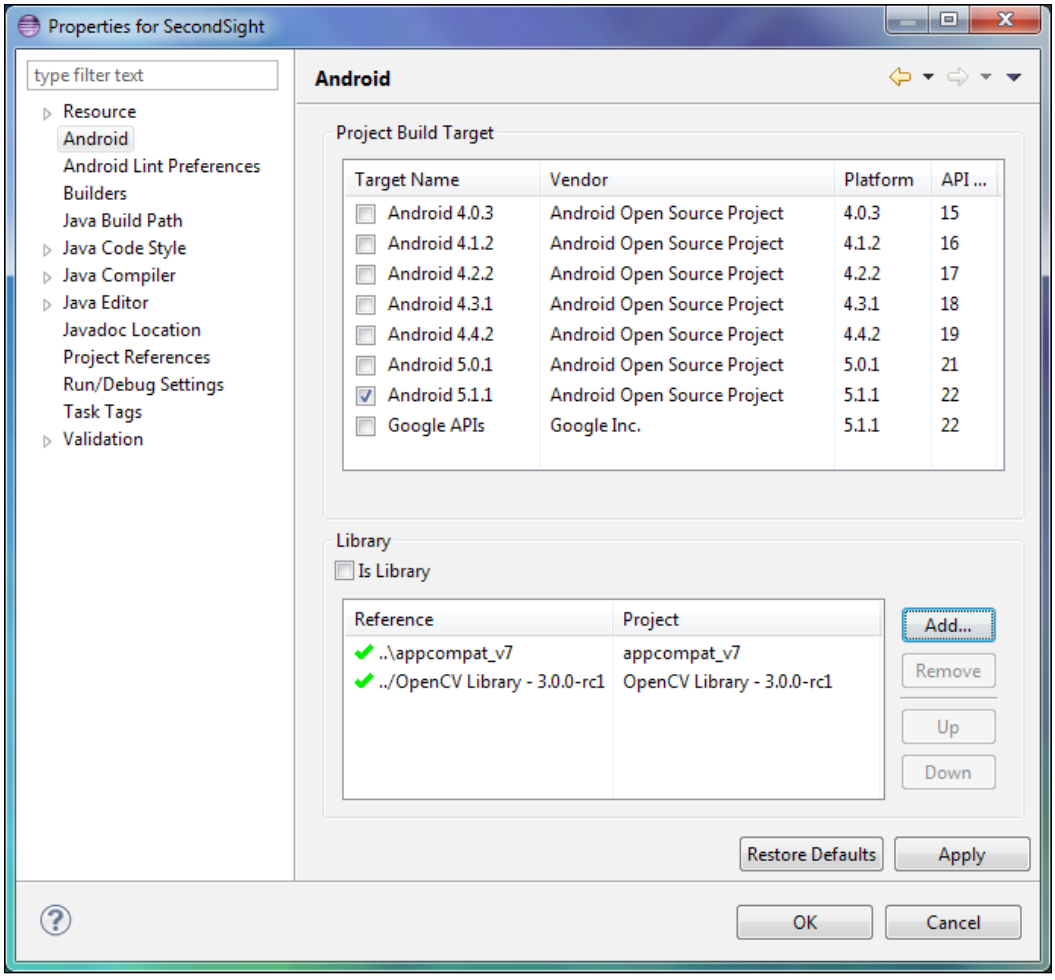
Click on the **Next** button. More options about the activity should appear.
Enter **CameraActivity** in the **Activity Name** field, as in the following screenshot:



Click on the **Finish** button. Our project is created. Also, you should see another new project, **appcompat_v7**, in the Package Explorer pane. The **v7 appcompat** library is part of the **Android Support Libraries**, which come with the Android SDK. The Support Libraries provide backward compatibility so that an app can use many new features of Android even when the user's device is running an old version of the operating system.

Remember that the workspace must contain the OpenCV library project also. If it is not yet present, import it as described in the *Building the OpenCV samples with Eclipse* section of *Chapter 1*, *Setting Up OpenCV*. Likewise, refer to *Chapter 1* for any other problems related to setting up the environment, using Eclipse, or configuring and troubleshooting a project.

We must specify that our app depends on OpenCV. Right-click on the **SecondSight** project in **Package Explorer** and, from the context menu, select **Properties**. The **Properties** window should appear. Go to its **Android** tab and use the **Add...** button to add a reference to the OpenCV library project. Confirm that the v7 appcompat library is also listed as a dependency. Once the references are added, the window should look like the following screenshot:



Click on **OK** to apply the new preferences.

We should be able to browse the contents of our `SecondSight` project in the **Package Explorer** pane. Let's remove and add some files. Perform the following steps:

1. Delete `res/layout/activity_camera.xml`. (Right-click on it, select **Delete** from the context menu, and click on **OK**.) The layout of our interface will be very simple, so it will be more convenient to create it in Java code instead of this separate XML file. However, if you do want an example of using OpenCV with an XML layout, you may refer to the sample apps that come with the library. See the *Building the OpenCV samples with Eclipse* section in *Chapter 1*, *Setting Up OpenCV*.
2. Delete the `res/values-v11` and `res/values-v14` folders. They contain the files that define alternative GUI styles for certain Android API levels. However, thanks to the v7 appcompat library, we can use a single style file, `res/values/styles.xml`, for all our supported API levels.
3. Create `src/com/nummist/secondsight/LabActivity.java`. (Right-click on **com.nummist.secondsight**, navigate to **New** | **Class** from the context menu, enter `LabActivity` in the **Name** field, and click on **Finish**.)
4. Create `res/menu/activity_lab.xml`. (Right-click on the parent folder, navigate to **New** | **Android XML File** from the context menu, enter `activity_lab` in the **File** field, and click on **Finish**.)

Now, we have the skeleton of our project. Throughout the rest of this chapter, we will edit several files to provide appropriate functionality and content.

Enabling camera and disk access in the manifest

The `AndroidManifest.xml` file (the **manifest**) specifies an Android app's requirements and components. Compared to the default manifest, the manifest in Second Sight needs to do the following additional work:

- Ensure that the device has at least one camera.
- Get permission to use the camera.
- Get permission to write files to permanent storage.
- Restrict the screen orientation to landscape mode because OpenCV's camera preview does not handle portrait mode well (at least, in OpenCV 2.x and OpenCV 3.0). See the following bug report, which describes the problem, and hints at the possibility of a fix in a future version of OpenCV 3.x: `http://code.opencv.org/issues/3565`.
- Register the second activity.

We can accomplish these tasks by editing the `uses-permission`, `uses-feature`, and `activity` tags in the manifest.

For details about the Android manifest, see the official documentation at `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

Open `AndroidManifest.xml`, which is under the project's root directory. View it in the source code mode by clicking on the tab labeled **AndroidManifest.xml**. Edit the file by adding the highlighted code in the following snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
  "http://schemas.android.com/apk/res/android"
  package="com.nummist.secondsight"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="22" />

  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name=
    "android.permission.WRITE_EXTERNAL_STORAGE" />

  <uses-feature android:name="android.hardware.camera" /
>
  <uses-feature android:name="android.hardware.camera.autofocus"
    android:required="false" />
  <uses-feature android:name="android.hardware.camera.flash"
    android:required="false" />

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="com.nummist.secondsight.CameraActivity"
      android:label="@string/app_name"
```

```
        android:screenOrientation="landscape">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name=
            "android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity
        android:name="com.nummist.secondsight.LabActivity"
        android:label="@string/app_name"
        android:screenOrientation="landscape">
      </activity>
    </application>
</manifest>
```

Adapting the code to OpenCV 2.x

Replace `android:targetSdkVersion="22"` with `android:targetSdkVersion="19"`. This change enables a backward compatibility mode, thereby avoiding a critical problem in OpenCV 2.x's loader on Android 5.x (Lollipop). The problem is described in the following Stack Overflow thread: `http://stackoverflow.com/questions/27470313/opencv-service-intent-must-be-explicit-android-5-0-lolipop`. (For OpenCV 3.x, this issue has been fixed.)

By requiring the feature `android.hardware.camera`, we are specifying that Google Play should only distribute our app to devices that have a rear-facing camera. For historical reasons, a front-facing camera does not satisfy the `android.hardware.camera` requirement. If we required a front-facing camera, we would instead specify the `android.hardware.camera.front` feature. If instead we required any (front- or rear-facing) camera, we could, in principle, specify the `android.hardware.camera.any` feature. However, in practice, Google Play erroneously fails to recognize this feature on most devices. Thus, the most practical filter is `android.hardware.camera`. Another alternative would be to omit the `uses-feature` tags altogether and instead test the device's hardware capabilities at runtime. Later in this chapter, in the section *Previewing and saving photos in CameraActivity*, we will see how to query the number of cameras and their capabilities.

Creating menu and string resources

Our app's menus and localizable text are described in XML files. Identifiers in these resource files are referenced by the Java code, as we will see later.

For details about Android app resources, see the official documentation at `http://developer.android.com/guide/topics/resources/index.html`.

First, let's edit `res/menu/activity_camera.xml` so that it has the following implementation, describing the menu items for `CameraActivity`:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <item
    android:id="@+id/menu_next_camera"
    app:showAsAction="ifRoom|withText"    android:title="@string/menu_
next_camera"{ }/>
  <item
    android:id="@+id/menu_take_photo"
    app:showAsAction="always|withText"
    android:title="@string/menu_take_photo"{ }/>
</menu>
```

Note that we use the `app:showAsAction` attribute to make menu items appear in the app's top bar, as seen in the earlier screenshots. For backward compatibility, this attribute is defined in the resources of the v7 appcompat library. By referencing the v7 appcompat library, our project merges the library's resources into the application's resources, and we define `app` to be the XML namespace for these resources using the attribute `xmlns:app="http://schemas.android.com/apk/res-auto"`, in the preceding code block.

Also note that the menu items for image sizes are not defined in the preceding code block. We will create these menu items programmatically based on the camera capabilities that we query at runtime.

Similarly, the menu items for `LabActivity` are described in `res/menu/activity_lab.xml`, as follows:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">>>
  <item
    android:id="@+id/menu_delete"
```

```
      app:showAsAction="ifRoom|withText"
      android:title="@string/delete" />
    <item
      android:id="@+id/menu_edit"
      app:showAsAction="ifRoom|withText"
      android:title="@string/edit" />
    <item
      android:id="@+id/menu_share"
      app:showAsAction="ifRoom|withText"
      android:title="@string/share" />
</menu>
```

To support an action bar, we must change the app's GUI styles. Edit the app's default style file, `res/values/styles.xml`, and change the declaration of `AppBaseTheme` to match the following:

```
    <style name="AppBaseTheme"
        parent="Theme.AppCompat.Light.DarkActionBar">
```

If you have not already done so, delete the `res/values-v11` and `res/values-v14` folders, which contain unneeded alternative styles for certain API levels. Thanks to the v7 appcompat library, we can use the default style file for all our supported API levels.

The strings of user-readable text, used in various places in the app, are described in `res/values/strings.xml` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Second Sight</string>
  <string name="delete">Delete</string>
  <string name="edit">Edit</string>
  <string name="menu_next_camera">Next Cam</string>
  <string name="menu_take_photo">Take Photo</string>
  <string name="menu_image_size">Size</string>
  <string name="photo_delete_prompt_message">This photo is saved
    in your Gallery. Do you want to delete it?</string>
  <string name="photo_delete_prompt_title">Delete photo?</string>
  <string name="photo_error_message">Failed to save photo</string>
  <string name="photo_edit_chooser_title">Edit photo
    with&amp;#8230;</string>
  <string name="photo_send_chooser_title">Share photo
    with&amp;#8230;</string>
```

```
    <string name="photo_send_extra_subject">My photo from Second
      Sight</string>
    <string name="photo_send_extra_text">Check out my photo from
      the Second Sight app! http://nummist.com/opencv/</string>
    <string name="share">Share</string>
</resources>
```

Having defined these boilerplate resources, we can proceed to the implementation of our app's functionality in Java.

Previewing and saving photos in CameraActivity

Our main activity, `CameraActivity`, needs to do the following:

- On startup, use OpenCV Manager 3 to ensure that the appropriate OpenCV shared libraries are available. (For more information about OpenCV Manager 3 and previous versions, refer to the *Building the OpenCV samples with Eclipse* section in *Chapter 1*, *Setting Up OpenCV*.)
- Display a live camera feed.
- Provide the following menu actions:
  - Switch the active camera (for a device that has multiple cameras)
  - Change the image size (for a camera that supports multiple image sizes)
- Save a photo and insert it into `MediaStore` so that it is accessible to apps such as Gallery. Immediately open the photo in `LabActivity`.

We will use OpenCV functionality wherever feasible, even though we could just use the standard Android libraries to display a live camera feed, save a photo, and so on. To get information about the capabilities of the device's camera or cameras, we will rely on a standard Android class called `Camera`.

Starting in API level 21 (Lollipop), the `Camera` class is deprecated in favor of a new package, `android.hardware.camera2`. (See the official documentation at `https://developer.android.com/reference/android/hardware/camera2/package-summary.html`.) However, so far, there is no backward compatibility for the `camera2` package before API level 21. Thus, we will use the deprecated `Camera` class in order to support more devices.

OpenCV provides an abstract class called `CameraBridgeViewBase`, which represents a live camera feed. This class extends Android's `SurfaceView` class so that its instances can be a part of the view hierarchy. Moreover, a `CameraBridgeViewBase` instance can dispatch events to any listener that implements one of two interfaces, either `CvCameraViewListener` or `CvCameraViewListener2`. Often, the listener will be an activity, as is the case with `CameraActivity`.

The `CvCameraViewListener` and `CvCameraViewListener2` interfaces provide callbacks to handle the start and stop of a stream of camera input and to handle the capture of each frame. The two interfaces differ in terms of the image format. `CvCameraViewListener` always receives an RGBA color frame, which is passed as an instance of OpenCV's `Mat` class. Conceptually, a `Mat` is a multidimensional array that may store pixel data. `CvCameraViewListener2` receives each frame as an instance of OpenCV's `CvCameraViewFrame` class. From the passed `CvCameraViewFrame`, we may get a `Mat` image in either RGBA color or grayscale format. Thus, `CvCameraViewListener2` is the more flexible interface, and it is the one we implement in `CameraActivity`.

Since `CameraBridgeViewBase` is an abstract class, we need an implementation. OpenCV provides two implementations: `JavaCameraView` and `NativeCameraView`. They are both Java classes, but `NativeCameraView` is a Java wrapper around a native C++ class. `NativeCameraView` potentially yields a higher frame rate, but it is prone to device-specific bugs and it also tends to break when new Android OS versions come out. Thus, for greater reliability, we use `JavaCameraView` in our app.

To support the interaction between OpenCV Manager and client apps, OpenCV provides an abstract class called `BaseLoaderCallback`. This class declares a callback method that is executed after OpenCV Manager ensures that the library is available. Typically, this callback is the appropriate place to enable the camera view and create any other OpenCV objects.

Now that we know something about the relevant OpenCV types, let's open `CameraActivity.java` and add the following declarations of our activity class and its member variables:

For brevity, the code listings in this book omit `package` and `import` statements. Eclipse should autogenerate `package` statements when you create files and `import` statements when you declare variables.

```
// Use the deprecated Camera class.
@SuppressWarnings("deprecation")
public class CameraActivity extends ActionBarActivity
    implements CvCameraViewListener2 {

  // A tag for log output.
  private static final String TAG =
    CameraActivity.class.getSimpleName();

  // A key for storing the index of the active camera.
  private static final String STATE_CAMERA_INDEX = "cameraIndex";

  // A key for storing the index of the active image size.
  private static final String STATE_IMAGE_SIZE_INDEX =
    "imageSizeIndex";

  // An ID for items in the image size submenu.
  private static final int MENU_GROUP_ID_SIZE = 2;

  // The index of the active camera.
  private int mCameraIndex;

  // The index of the active image size.
  private int mImageSizeIndex;

  // Whether the active camera is front-facing.
  // If so, the camera view should be mirrored.
  private boolean mIsCameraFrontFacing;

  // The number of cameras on the device.
  private int mNumCameras;

  // The camera view.
  private CameraBridgeViewBase mCameraView;

  // The image sizes supported by the active camera.
  private List<Size> mSupportedImageSizes;

  // Whether the next camera frame should be saved as a photo.
  private boolean mIsPhotoPending;

  // A matrix that is used when saving photos.
  private Mat mBgr;
```

```
    // Whether an asynchronous menu action is in progress.
    // If so, menu interaction should be disabled.
    private boolean mIsMenuLocked;

    // The OpenCV loader callback.
    private BaseLoaderCallback mLoaderCallback =
      new BaseLoaderCallback(this) {
  @Override
  public void onManagerConnected(final int status) {
    switch (status) {
    case LoaderCallbackInterface.SUCCESS:
      Log.d(TAG, "OpenCV loaded successfully");
      mCameraView.enableView();
      mBgr = new Mat();
      break;
    default:
      super.onManagerConnected(status);
      break;
    }
  }
```

The concept of states (varying modes of operation) is central to Android activities, and `CameraActivity` is no exception. When the user selects a menu action to switch the camera on or take a photo, the effects are not instantaneous. Actions affect the work that must be done in subsequent frames. Some of this work is even done asynchronously. Thus, many member variables of `CameraActivity` are dedicated to tracking the logical state of the activity.

Understanding asynchronous event collisions in Android

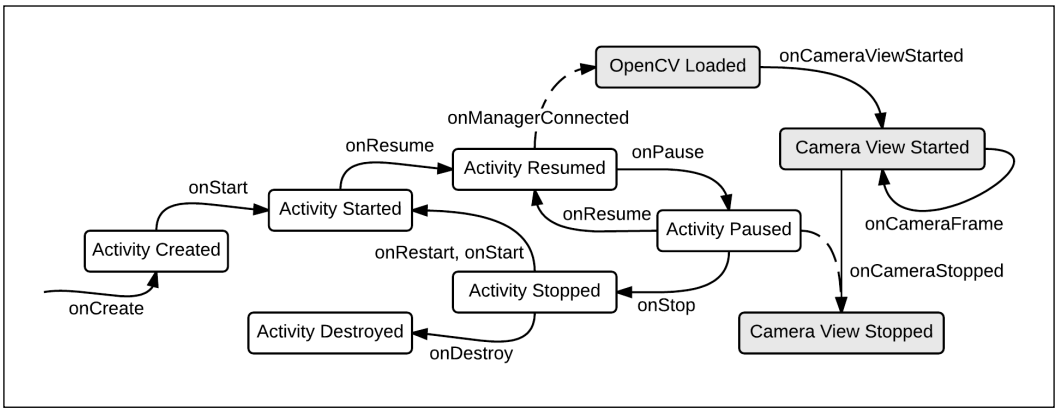
Many Android library methods such as `startActivity` do their work asynchronously, which means that they run on a background thread to allow the main (user interface) thread to continue processing events. This is to say, while the work is being carried out, the user may continue to use the interface, potentially initiating other work that is logically inconsistent with the first work.

For example, suppose that `startActivity` is called when a certain button is clicked. If the user presses the button multiple times, quickly, then more than one new activity may be pushed onto the activity stack. This behavior is probably not what the developer or user intended. A solution would be to disable the clicked button until its activity resumes. Similar considerations affect our menu system in `CameraActivity`.

Similar to any Android activity, `CameraActivity` also implements several callbacks that are executed in response to standard state changes, namely, changes in the activity lifecycle. Let's start by looking at the `onCreate` and `onSaveInstanceState` callbacks. These methods, respectively, are called at the beginning and end of the activity lifecycle. The `onCreate` callback typically sets up the activity's view hierarchy, initializes data, and reads any saved data that may have been written last time `onSaveInstanceState` was called.

For details about the Android activity lifecycle, see the official documentation at `http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle`.

The following diagram summarizes the various states in the Android activity lifecycle and the callbacks that are called during state transitions. (Some of these callbacks, such as `onStart`, are not implemented in `CameraActivity`; instead, we use the default implementation.) Moreover, the gray boxes in the diagram represent states in the lifecycle of the OpenCV library and camera view. A dotted line indicates a relationship between states in the activity lifecycle and OpenCV lifecycle.



In `CameraActivity`, the `onCreate` callback sets up the camera view and initializes data about the cameras. It also reads any previous data about the active camera that may have been written by `onSaveInstanceState`. Here are the implementations of the two methods:

```
// Suppress backward incompatibility errors because we provide
// backward-compatible fallbacks.
@SuppressLint("NewApi")
@Override
protected void onCreate(final Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);

    final Window window = getWindow();
    window.addFlags(
      WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

    if (savedInstanceState != null) {
      mCameraIndex = savedInstanceState.getInt(
        STATE_CAMERA_INDEX, 0);
      mImageSizeIndex = savedInstanceState.getInt(
        STATE_IMAGE_SIZE_INDEX, 0);
    } else {
      mCameraIndex = 0;
      mImageSizeIndex = 0;
    }

    final Camera camera;
    if (Build.VERSION.SDK_INT >=
      Build.VERSION_CODES.GINGERBREAD) {
      CameraInfo cameraInfo = new CameraInfo();
      Camera.getCameraInfo(mCameraIndex, cameraInfo);
      mIsCameraFrontFacing =
        (cameraInfo.facing ==
          CameraInfo.CAMERA_FACING_FRONT);
      mNumCameras = Camera.getNumberOfCameras();
      camera = Camera.open(mCameraIndex);
    } else { // pre-Gingerbread
      // Assume there is only 1 camera and it is rear-facing.
      mIsCameraFrontFacing = false;
      mNumCameras = 1;
      camera = Camera.open();
    }
    final Parameters parameters = camera.getParameters();
    camera.release();
    mSupportedImageSizes =
      parameters.getSupportedPreviewSizes();
    final Size size = mSupportedImageSizes.get(mImageSizeIndex);

    mCameraView = new JavaCameraView(this, mCameraIndex);
    mCameraView.setMaxFrameSize(size.width, size.height);
    mCameraView.setCvCameraViewListener(this);
    setContentView(mCameraView);
  }
```

```
public void onSaveInstanceState(Bundle savedInstanceState) {
  // Save the current camera index.
  savedInstanceState.putInt(STATE_CAMERA_INDEX, mCameraIndex);

  // Save the current image size index.
  savedInstanceState.putInt(STATE_IMAGE_SIZE_INDEX,
    mImageSizeIndex);

  super.onSaveInstanceState(savedInstanceState);
}
```

Note that certain data regarding the device's cameras are unavailable on Froyo (the oldest Android version that we support). To avoid runtime errors, we check `Build.VERSION.SDK_INT` before using the new APIs. Furthermore, to avoid seeing errors during static analysis (that is, before compilation), we add the `@SuppressLint("NewApi")` annotation to the declaration of `onCreate`.

Also note that every call to `Camera.open` must be paired with a call to the `Camera` instance's `release` method in order to make the camera available later. Otherwise, our app and other apps may subsequently encounter a `RuntimeException` when calling `Camera.open`.

For more details about the `Camera` class, see the official documentation at `http://developer.android.com/reference/android/hardware/Camera.html`.

When we switch to a different camera or image size, it will be most convenient to recreate the activity so that `onCreate` will run again. On Honeycomb and newer Android versions, a `recreate` method is available, but for backward compatibility, we should write our own alternative implementation, as follows:

```
// Suppress backward incompatibility errors because we provide
// backward-compatible fallbacks.
@SuppressLint("NewApi")
@Override
public void recreate() {
  if (Build.VERSION.SDK_INT >=
    Build.VERSION_CODES.HONEYCOMB) {
    super.recreate();
  } else {
    finish();
    startActivity(getIntent());
  }
}
```

An `Intent` (such as the `startActivity` argument) is the means for one activity to create or communicate with another activity. The `getIntent` method simply obtains the `Intent` that was used to launch the current activity in the first place. Thus, this `Intent` is appropriate to recreate the activity. We will discuss intents fully in the section *Deleting, editing, and sharing photos in LabActivity*, later in this chapter.

Several other activity lifecycle callbacks are also relevant to OpenCV. When the activity goes into the background (the `onPause` callback) or finishes (the `onDestroy` callback), the camera view should be disabled. When the activity comes into the foreground (the `onResume` callback), the `OpenCVLoader` should attempt to initialize the library. (Remember that the camera view is enabled once the library is successfully initialized.) Here are the implementations of the relevant callbacks:

```
@Override
public void onPause() {
  if (mCameraView != null) {
    mCameraView.disableView();
  }
  super.onPause();
}

@Override
public void onResume() {
  super.onResume();
  OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_3_0_0,
    this, mLoaderCallback);
  mIsMenuLocked = false;
}

@Override
public void onDestroy() {
  if (mCameraView != null) {
    mCameraView.disableView();
  }
  super.onDestroy();
}
```

Adapting the code to OpenCV 2.x

Replace `OpenCVLoader.OPENCV_VERSION_3_0_0` with an earlier version such as `OpenCVLoader.OPENCV_VERSION_2_4_9`.

Note that, in `onResume`, we re-enable the menu interaction. We do this in case it was previously disabled while pushing a child activity onto the stack.

At this point, our activity has the necessary code to set up a camera view and get data about the device's cameras. Next, we should implement the menu actions that enable the user to switch the camera on, change the image size, and request that a photo be taken. Again, there are relevant activity lifecycle callbacks such as `onCreateOptionsMenu` and `onOptionsItemSelected`. In `onCreateOptionsMenu`, we load our menu from its resource file. Then, if the device has only one camera, we remove the **Next Cam** menu item. If the active camera supports more than one image size, we will create a set of menu options for all the supported sizes. In `onOptionsItemSelected`, we will handle any image size menu item by recreating the activity with the specified image size. (Remember that the image size index is saved in `onSaveInstanceState` and restored in `onCreate`, where it is used to construct the camera view.) Similarly, we handle the **Next Cam** menu item by cycling to the next camera index and then recreating the activity. (Remember that the camera index is saved in `onSaveInstanceState` and restored in `onCreate`, where it is used to construct the camera view.) We handle the **Take Photo** menu item by setting a Boolean value, which we check in an OpenCV callback later. In either case, we block any further handling of menu options until the current handling is complete (for example, until `onResume`). Here is the implementation of the two menu-related callbacks:

```
@Override
public boolean onCreateOptionsMenu(final Menu menu) {
  getMenuInflater().inflate(R.menu.activity_camera, menu);
  if (mNumCameras < 2) {
    // Remove the option to switch cameras, since there is
    // only 1.
    menu.removeItem(R.id.menu_next_camera);
  }
  int numSupportedImageSizes = mSupportedImageSizes.size();
  if (numSupportedImageSizes > 1) {
    final SubMenu sizeSubMenu = menu.addSubMenu(
      R.string.menu_image_size);
    for (int i = 0; i < numSupportedImageSizes; i++) {
      final Size size = mSupportedImageSizes.get(i);
      sizeSubMenu.add(MENU_GROUP_ID_SIZE, i, Menu.NONE,
        String.format("%dx%d", size.width, size.height));
    }
  }
  return true;
}
```

```
// Suppress backward incompatibility errors because we provide
// backward-compatible fallbacks (for recreate).
@SuppressLint("NewApi")
@Override
public boolean onOptionsItemSelected(final MenuItem item) {
  if (mIsMenuLocked) {
    return true;
  }
  if (item.getGroupId() == MENU_GROUP_ID_SIZE) {
    mImageSizeIndex = item.getItemId();
    recreate();

    return true;
  }
  switch (item.getItemId()) {
  case R.id.menu_next_camera:
    mIsMenuLocked = true;

    // With another camera index, recreate the activity.
    mCameraIndex++;
    if (mCameraIndex == mNumCameras) {
      mCameraIndex = 0;
    }
    mImageSizeIndex = 0;
    recreate();

    return true;
  case R.id.menu_take_photo:
    mIsMenuLocked = true;

    // Next frame, take the photo.
    mIsPhotoPending = true;

    return true;
  default:
    return super.onOptionsItemSelected(item);
  }
}
```

Next, let's look at the callbacks that are required by the `CvCameraViewListener2` interface. `CameraActivity` does not need to do anything when the camera feed starts (the `onCameraViewStarted` callback) or stops (the `onCameraViewStopped` callback), but it may need to perform some operations whenever a new frame arrives (the `onCameraFrame` callback). First, if the user has requested a photo, one should be taken. (The photo capture functionality is actually quite complex, so we put it in a helper method, `takePhoto`, which we will examine later in this section.)

Second, if the active camera is front-facing (that is, user-facing), the camera view should be mirrored (horizontally flipped), since people are accustomed to looking at themselves in a mirror, rather than from a camera's true perspective. OpenCV's `Core.flip` method can be used to mirror the image. The arguments to `Core.flip` are a source `Mat`, a destination `Mat` (which may be the same as the source), and an integer indicating whether the flip should be vertical (`0`), horizontal (`1`), or both (`-1`). Here is the implementation of the `CvCameraViewListener2` callbacks:

```
@Override
public void onCameraViewStarted(final int width,
  final int height) {
}

@Override
public void onCameraViewStopped() {
}

@Override
public Mat onCameraFrame(final CvCameraViewFrame inputFrame) {
  final Mat rgba = inputFrame.rgba();

  if (mIsPhotoPending) {
    mIsPhotoPending = false;
    takePhoto(rgba);
  }

  if (mIsCameraFrontFacing) {
    // Mirror (horizontally flip) the preview.
    Core.flip(rgba, rgba, 1);
  }

  return rgba;
}
```

Now, finally, we are arriving at the function that will capture users' hearts and minds, or at least, their photos. As an argument, `takePhoto` receives an RGBA color `Mat` that was read from the camera. We want to write this image to a disk, using an OpenCV method called `Imgcodecs.imwrite`. This method requires an image in BGR or BGRA color format, so first we must convert the RGBA image, using the `Imgproc.cvtColor` method. Besides saving the image to a disk, we also want to enable other apps to find it via Android's `MediaStore`. To do so, we generate some metadata about the photo and then, using a `ContentResolver` object, we insert this metadata into `MediaStore` and get back a URI.

If we encounter a failure to save or insert the photo, we give up and call a helper method, `onTakePhotoFailed`, which unlocks the menu interaction and shows an error message to the user. (For example, a failure would arise if we omitted `WRITE_EXTERNAL_STORAGE` from `AndroidManifest.xml`, if the user ran out of disk space, or if a filename collided with a previous entry in `MediaStore`.) On the other hand, if everything succeeds, we start `LabActivity` and pass it the data it needs to locate the saved photo. Here is the implementation of `takePhoto` and `onTakePhotoFailed`:

```
private void takePhoto(final Mat rgba) {

  // Determine the path and metadata for the photo.
  final long currentTimeMillis = System.currentTimeMillis();
  final String appName = getString(R.string.app_name);
  final String galleryPath =
    Environment.getExternalStoragePublicDirectory(
      Environment.DIRECTORY_PICTURES).toString();
  final String albumPath = galleryPath + File.separator +
    appName;
  final String photoPath = albumPath + File.separator +
    currentTimeMillis + LabActivity.PHOTO_FILE_EXTENSION;
  final ContentValues values = new ContentValues();
  values.put(MediaStore.MediaColumns.DATA, photoPath);
  values.put(Images.Media.MIME_TYPE,
    LabActivity.PHOTO_MIME_TYPE);
  values.put(Images.Media.TITLE, appName);
  values.put(Images.Media.DESCRIPTION, appName);
  values.put(Images.Media.DATE_TAKEN, currentTimeMillis);

  // Ensure that the album directory exists.
  File album = new File(albumPath);
  if (!album.isDirectory() && !album.mkdirs()) {
    Log.e(TAG, "Failed to create album directory at " +
      albumPath);
    onTakePhotoFailed();
    return;
  }

  // Try to create the photo.
  Imgproc.cvtColor(rgba, mBgr, Imgproc.COLOR_RGBA2BGR, 3);
  if (!Imgcodecs.imwrite(photoPath, mBgr)) {
    Log.e(TAG, "Failed to save photo to " + photoPath);
    onTakePhotoFailed();
  }
```

```
      Log.d(TAG, "Photo saved successfully to " + photoPath);

      // Try to insert the photo into the MediaStore.
      Uri uri;
      try {
        uri = getContentResolver().insert(
          Images.Media.EXTERNAL_CONTENT_URI, values);
      } catch (final Exception e) {
        Log.e(TAG, "Failed to insert photo into MediaStore");
        e.printStackTrace();

        // Since the insertion failed, delete the photo.
        File photo = new File(photoPath);
        if (!photo.delete()) {
          Log.e(TAG, "Failed to delete non-inserted photo");
        }

        onTakePhotoFailed();
        return;
      }

      // Open the photo in LabActivity.
      final Intent intent = new Intent(this, LabActivity.class);
      intent.putExtra(LabActivity.EXTRA_PHOTO_URI, uri);
      intent.putExtra(LabActivity.EXTRA_PHOTO_DATA_PATH,
        photoPath);
      startActivity(intent);
    }

    private void onTakePhotoFailed() {
      mIsMenuLocked = false;

      // Show an error message.
      final String errorMessage =
        getString(R.string.photo_error_message);
      runOnUiThread(new Runnable() {
        @Override
        public void run() {
          Toast.makeText(CameraActivity.this, errorMessage,
            Toast.LENGTH_SHORT).show();
        }
      });
    }
  }
```

Adapting the code to OpenCV 2.x

Replace `Imgcodecs.imwrite` with `Highgui.imwrite`.

For now, that's everything we want `CameraActivity` to do. We will expand this class in the following chapters, by adding more menu actions and handling them in the `onCameraFrame` callback.

Deleting, editing, and sharing photos in LabActivity

Our second activity, `LabActivity`, needs to do the following:

- From the previous activity, receive a URI and file path for a PNG file.
- Display the image that is contained in the PNG file.
- Provide the following menu actions:
  - **Delete**: Show a confirmation dialog. On confirmation, delete the PNG file and finish the activity.
  - **Edit**: Show an intent chooser so that the user may select an app to edit the PNG file. (The URI is passed with the `EDIT` intent.)
  - **Share**: Show a chooser so that the user may select an app to share or send the PNG file. (The URI is passed with the `SEND` intent.)

All of this functionality relies on the standard Android library classes, notably the `Intent` class. **Intents** are the means by which activities communicate with each other. An activity receives an intent from its parent (the activity that created it) and may receive intents from its children (activities it created) as they finish. The communicating activities may be in different applications. An intent may contain key-value pairs called **extras**.

For details about intents, see the official documentation at `http://developer.android.com/guide/components/intents-filters.html`.

`LabActivity` declares several public constants that are used by it and `CameraActivity`. These constants relate to the image's file type and the extra keys that are used when `CameraActivity` and `LabActivity` communicate via intents. `LabActivity` also has member variables that are used to store the URI and path values, which are extracted from the extras. The `onCreate` method does the work of extracting these values and setting up an image view that shows the PNG file. The implementation is as follows:

```
public class LabActivity extends ActionBarActivityActionBarActivity {

  public static final String PHOTO_FILE_EXTENSION = ".png";
  public static final String PHOTO_MIME_TYPE = "image/png";

  public static final String EXTRA_PHOTO_URI =
    "com.nummist.secondsight.LabActivity.extra.PHOTO_URI";
  public static final String EXTRA_PHOTO_DATA_PATH =
    "com.nummist.secondsight.LabActivity.extra.PHOTO_DATA_PATH";

  private Uri mUri;
  private String mDataPath;

  @Override
  protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    final Intent intent = getIntent();
    mUri = intent.getParcelableExtra(EXTRA_PHOTO_URI);
    mDataPath = intent.getStringExtra(EXTRA_PHOTO_DATA_PATH);

    final ImageView imageView = new ImageView(this);
    imageView.setImageURI(mUri);

    setContentView(imageView);
  }
```

Again, we are creating the activity's layout in Java code (rather than the alternative of loading an XML layout). Our layout is simple, but we need to configure it dynamically based on the image URI, so it is sensible to just use Java in this case.

The menu logic is simpler in `LabActivity` than in `CameraActivity`. All the menu actions of `LabActivity` result in a dialog or chooser being shown, and since a dialog or chooser blocks the rest of the user interface, we do not have to worry about blocking conflicting input ourselves. We just load the menu's resource file in `onCreateOptionsMenu` and call a helper method for each possible action in `onOptionsItemSelected`. The implementation is as follows:

```
    @Override
    public boolean onCreateOptionsMenu(final Menu menu) {
      getMenuInflater().inflate(R.menu.activity_lab, menu);
      return true;
    }

    @Override
    public boolean onOptionsItemSelected(final MenuItem item) {
      switch (item.getItemId()) {
      case R.id.menu_delete:
        deletePhoto();
        return true;
      case R.id.menu_edit:
        editPhoto();
        return true;
      case R.id.menu_share:
        sharePhoto();
        return true;
      default:
        return super.onOptionsItemSelected(item);
      }
    }
  }
```

Let's examine the menu action helper methods one by one, starting with `deletePhoto`. Most of this method's implementation is boilerplate code to set up a confirmation dialog. The dialog's confirmation button has an `onClick` callback that deletes the image from the `MediaStore` and finishes the activity. The implementation of `deletePhoto` is as follows:

```
    /*
     * Show a confirmation dialog. On confirmation, the photo is
     * deleted and the activity finishes.
     */
    private void deletePhoto() {
      final AlertDialog.Builder alert = new AlertDialog.Builder(
        LabActivity.this);
```

```
      alert.setTitle(R.string.photo_delete_prompt_title);
      alert.setMessage(R.string.photo_delete_prompt_message);
      alert.setCancelable(false);
      alert.setPositiveButton(R.string.delete,
    new DialogInterface.OnClickListener() {
    @Override
    public void onClick(final DialogInterface dialog,
      final int which) {
      getContentResolver().delete(
        Images.Media.EXTERNAL_CONTENT_URI,
        MediaStore.MediaColumns.DATA + "=?",
        new String[] { mDataPath });
      finish();
    }
  });
      alert.setNegativeButton(android.R.string.cancel, null);
      alert.show();
    }
```

The next helper method, `editPhoto`, sets up an intent and starts a chooser for this intent, using the `Intent.createChooser` method. The user may cancel this chooser or use it to select an activity. If an activity is selected, `editPhoto` starts it. The implementation is as follows:

```
    /*
     * Show a chooser so that the user may pick an app for editing
     * the photo.
     */
    private void editPhoto() {
      final Intent intent = new Intent(Intent.ACTION_EDIT);
      intent.setDataAndType(mUri, PHOTO_MIME_TYPE);
      startActivity(Intent.createChooser(intent,
        getString(R.string.photo_edit_chooser_title)));
    }
```

The last helper method, `sharePhoto`, is similar to `editPhoto`, though the intent is configured differently. The implementation is as follows:

```
    /*
     * Show a chooser so that the user may pick an app for sending
     * the photo.
     */
    private void sharePhoto() {
      final Intent intent = new Intent(Intent.ACTION_SEND);
      intent.setType(PHOTO_MIME_TYPE);
```

```
        intent.putExtra(Intent.EXTRA_STREAM, mUri);
        intent.putExtra(Intent.EXTRA_SUBJECT,
          getString(R.string.photo_send_extra_subject));
        intent.putExtra(Intent.EXTRA_TEXT,
          getString(R.string.photo_send_extra_text));
        startActivity(Intent.createChooser(intent,
          getString(R.string.photo_send_chooser_title)));
      }
    }
```

This is the last functionality we need to be able to capture a basic photo and share an application. Now, we should be able to build and run Second Sight. (Remember to watch out for any runtime errors in the **LogCat** pane!)

Summary

We used OpenCV to create and show a live camera feed and save the still images from this feed. We also saw how to integrate the camera feed's lifecycle into the Android activity lifecycle and share the saved images across the boundaries of activities and applications.

The next chapter will expand our Second Sight app by adding various image filtering options to the menus of `CameraActivity` and `LabActivity`.

3
Applying Image Effects

For this chapter, our goal is to add several image filters to Second Sight. These filters rely on various OpenCV functions to manipulate matrices through splitting, merging, arithmetic operations, or applying lookup tables for complex functions. Certain filters also rely on a mathematics library called **Apache Commons Math**.

The complete Eclipse project for this chapter can be downloaded from the author's website. The project has two versions:

A version for OpenCV 3.x is located at `http://nummist.com/opencv/4598_03.zip`.

A version for OpenCV 2.x is located at `http://nummist.com/opencv/5206_03.zip`.

Adding files to the project

We need to add several files to our Eclipse project in order to create new types (interfaces and classes) and link to a new library, Apache Commons Math. The following are the new types that we want to create:

- `com.nummist.secondsight.filters.Filter`: This is an interface that represents a filter that can be applied to an image.
- `com.nummist.secondsight.filters.NoneFilter`: This is a class that represents a filter that does nothing. It implements the `Filter` interface.
- `com.nummist.secondsight.filters.convolution.StrokeEdgesFilter`: This is a class that represents a filter that draws heavy, black lines on top of edge regions. It implements the `Filter` interface.

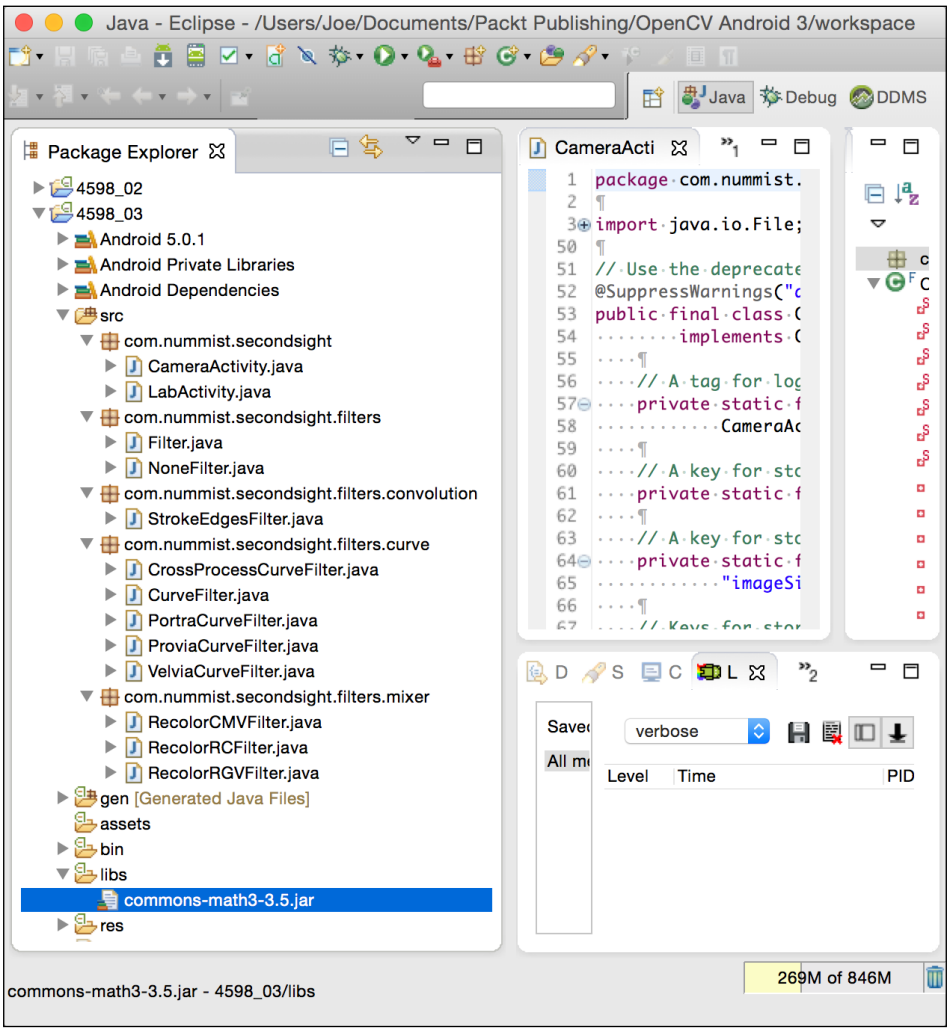
- `com.nummist.secondsight.filters.curve.CurveFilter`: This is a class that represents a filter that may apply a separate curvilinear transformation to each color channel in an image. (It is like **Curves** in Photoshop or GIMP.) It implements the `Filter` interface.
- `com.nummist.secondsight.filters.curve.CrossProcessCurveFilter`: This is a subclass of `CurveFilter`. It emulates a photo film processing technique called **cross-processing**.
- `com.nummist.secondsight.filters.curve.PortraCurveFilter`: This is a subclass of `CurveFilter`. It emulates a brand of photo film called Kodak Portra.
- `com.nummist.secondsight.filters.curve.ProviaCurveFilter`: This is a subclass of `CurveFilter`. It emulates a brand of photo film called Fuji Provia.
- `com.nummist.secondsight.filters.curve.VelviaCurveFilter`: This is a subclass of `CurveFilter`. It emulates a brand of photo film called Fuji Velvia.
- `com.nummist.secondsight.filters.mixer.RecolorCMVFilter`: This is a class that represents a filter that linearly combines color channels such that the image appears to be mixed from a limited palette of cyan, magenta, and white. (It is like a specialization of **Channel Mixer** in Photoshop or GIMP.) It implements the `Filter` interface.
- `com.nummist.secondsight.filters.mixer.RecolorRCFilter`: This is a class that represents a filter that linearly combines color channels such that the image appears to be mixed from a limited palette of red and cyan. (It is like a specialization of **Channel Mixer** in Photoshop or GIMP.) It implements the `Filter` interface.
- `com.nummist.secondsight.filters.mixer.RecolorRGVFilter`: This is a class that represents a filter that linearly combines color channels such that the image appears to be mixed from a limited palette of red, green, and white. (It is like a specialization of **Channel Mixer** in Photoshop or GIMP.) It implements the `Filter` interface.

Photoshop and **GNU Image Manipulation Program** (**GIMP**) are popular image editing applications. They are not required for this book, but, optionally, you might want to try this kind of program so that you can experiment with image processing effects before implementing them in the code. Photoshop is a commercial application. GIMP is an open-source program, which is available for free at `http://www.gimp.org/downloads/`.

Create the appropriate packages and Java files under the `src` directory in the **Package Explorer** pane. (Right-click on the `src` directory and then navigate to **New** | **Package**, **New** | **Interface**, or **New** | **Class** from the context menu.)

Now, let's get the Apache Commons Math library. Download the latest version from `http://commons.apache.org/proper/commons-math/download_math.cgi`. Unzip the downloaded file. Inside the unzipped folder, find a file with a name such as `commons-math3-3.5.jar`. (The version numbers may differ.) Copy this file into the `libs` folder of the Eclipse project.

After all the necessary files are added, your **Package Explorer** pane should look similar to the one in the following screenshot:



Defining the Filter interface

For our purposes, a filter is any transformation that can be applied to a source image and a destination image. (The source and destination may be the same image or different images.) Our application needs to treat filters interchangeably, so it is a good idea to formalize this definition of a filter's interface. Let's edit `Filter.java` so that the `Filter` interface is defined as follows:

```
public interface Filter {
  public abstract void apply(final Mat src, final Mat dst);
}
```

As far as our app is concerned, the `apply` method is the only thing that our filters must have in common. Everything else is the implementation details.

The most basic implementation of the `Filter` interface is the `NoneFilter` class. As the name suggests, `NoneFilter` does no filtering at all. Let's implement it as follows:

```
public class NoneFilter implements Filter {
  @Override
  public void apply(final Mat src, final Mat dst) {
    // Do nothing.
  }
}
```

`NoneFilter` is just a convenient stand-in for the other filters. We use it when we want to turn off filtering but still have an object that conforms to the `Filter` interface.

Mixing color channels

As we saw in *Chapter 2*, *Working with Camera Frames*, OpenCV stores image data in a matrix of type `Mat`, which is like a multidimensional array. The rows and columns (specified by the first and second indices, respectively) correspond to the y and x pixel coordinates in the image. The elements are pixel values. A pixel value may be represented by one number (in the case of a grayscale image) or multiple numbers (in the case of a color image). Each of these numbers is said to belong to a channel. An opaque, grayscale image has just one channel, value (brightness), which is abbreviated as V. A color image may have as many as four channels—for example, red, green, blue, and alpha (transparency), which constitute the **RGBA** color model. Other useful models for color images include **RGB** (red, green, blue), **HSV** (hue, saturation, value), and **YUV** (brightness, greenness versus blueness, greenness versus redness). In this book, we focus on RGB and RGBA images, but OpenCV supports other color models and various numeric formats for each model. As we saw in the previous chapter, we can convert between color formats with the `Imgproc.cvtColor` static method.

If we separated the channels of an RGB image matrix, we could make three different grayscale image matrices, each having one channel. We could then apply some matrix arithmetic to these single-channel matrices and merge the results to get another RGB image matrix. The resulting RGB image would look as if it were mixed from a different color palette to the original image. This technique is called **channel mixing**. For an RGB image, we may define channel mixing as follows, in pseudocode:

```
dst.r = funcR(src.r, src.g, src.b)
dst.g = funcG(src.r, src.g, src.b)
dst.b = funcB(src.r, src.g, src.b)
```

This means that each channel in the destination image is a function of any or all channels in the source image. We will not restrict our definition to any particular kind of function. However, let's note the visual effects of the following operations, which I find useful when working with RGB images:

- An average or weighted average appears to tint the output channel. For example, in pseudocode, if `dst.b = 0.5 * src.r + 0.5 * src.b`, the image regions that were originally bluish become reddish or purplish.
- A `min` operation appears to desaturate the output channel. For example, in pseudocode, if `dst.b = min(src.r, src.g, src.b)`, blues become gray.
- A `max` operation appears to desaturate the output channel's complementary color. For example, in pseudocode, if `dst.b = max(src.r, src.g, src.b)`, yellows become gray. (Yellow is blue's complement or, in other words, white minus blue is yellow, when we are dealing with the RGB color model.)

With these effects in mind, let's look at the OpenCV functionality that we would use to produce them. OpenCV's `Core` class provides all the relevant functionality as static methods. The `Core.split(Mat m, List<Mat> mv)` method is responsible for channel splitting. It takes a source matrix and a list of destination matrices as arguments. Each channel from the source is copied into a single-channel matrix in the destination list. If necessary, the destination list is populated with new matrices.

After using the `Core.split` method, we can apply matrix operations to the individual channels. The `Core.addWeighted(Mat src1, double alpha, Mat src2, double beta, double gamma, Mat dst)` method can be used to take a weighted average of two channels. The first four arguments are weights and source matrices. The fifth argument is a constant that is added to the result. The last argument is the destination matrix. In pseudocode, `dst = alpha * src1 + beta * src2 + gamma`.

Generally, with methods in OpenCV, it is safe to pass a destination matrix that is also a source matrix. Of course, in this case, the values in the source matrix are overwritten. This is called an **in-place operation**.

Each of the `Core.min(Mat src1, Mat src2, Mat dst)` and `Core.max(Mat src1, Mat src2, Mat dst)` methods take a pair of source matrices and a destination matrix. These methods perform a per-element min or max.

Finally, the inverse of `Core.split` is `Core.merge(List<Mat> mv, Mat m)`. We can use it to recreate a multichannel image from the split channels.

To study a practical example of channel mixing, let's open `RecolorRCFilter.java` and write the following implementation of the class:

```
public class RecolorRCFilter implements Filter {
  private final ArrayList<Mat> mChannels = new ArrayList<Mat>(4);
  @Override
  public void apply(final Mat src, final Mat dst) {
    Core.split(src, mChannels);
    final Mat g = mChannels.get(1);
    final Mat b = mChannels.get(2);
    // dst.g = 0.5 * src.g + 0.5 * src.b
    Core.addWeighted(g, 0.5, b, 0.5, 0.0, g);
    // dst.b = dst.g
    mChannels.set(2, g);
    Core.merge(mChannels, dst);
  }
}
```

The effect of this filter is to turn greens and blues to cyan, leaving a limited color palette of red and cyan. It resembles the color palette of certain old movies and old computer games.

As a member variable, `RecolorRCFilter` has a list of four matrices. Whenever the `apply` method is called, this list is populated with the four channels of the source matrix. (We assume that the source and destination matrices each have four channels, in RGBA order.) We get the green and blue channels (at indices `1` and `2` in the list), take their average, and assign the result back to the same channels. Finally, we merge the four channels into the destination matrix, which may be the same as the source matrix.

The code for our other two channel mixing filters is similar, so, to save time, we will omit most of it. Just note that `RecolorRGVFilter` relies on the following operations:

```
// dst.b = min(dst.r, dst.g, dst.b)
Core.min(b, r, b);
Core.min(b, g, b);
```

The effect of this filter is to desaturate blues, leaving a limited color palette of red, green, and white. It, too, resembles the color palette of certain old movies and old computer games.

Similarly, `RecolorCMVFilter` relies on the following operations:

```
// dst.b = max(dst.r, dst.g, dst.b)
Core.max(b, r, b);
Core.max(b, g, b);
```

The effect of this filter is to desaturate yellows, leaving a limited color palette of cyan, magenta, and white. Nobody has made a movie in this color palette (yet!), but it will be a familiar sight to gamers of the 1980s.

The following strip of screenshots is a comparison of our channel mixing filters. From left to right, we see an unfiltered image and then, images filtered with `RecolorRCFilter`, `RecolorRGFilter`, and `RecolorCMVFilter`:

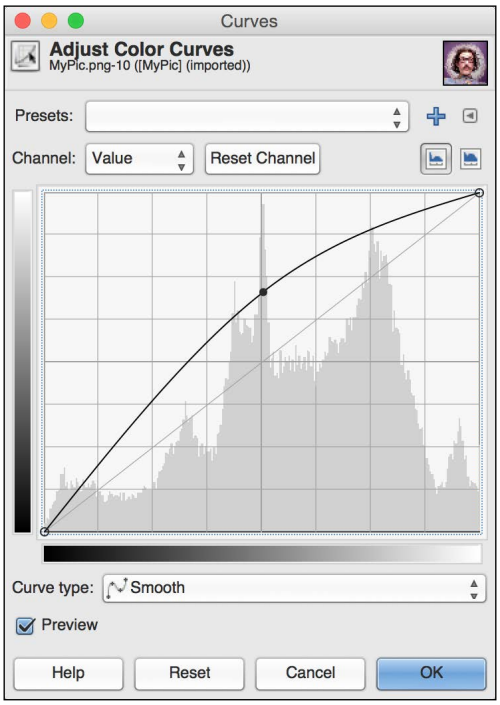
The differences among these channel mixing filters are not obvious in black-and-white prints. For color images, please see the eBook.

Arbitrary channel mixing functions, in RGB, tend to produce effects that are bold and stylized, not subtle. This is true of our examples here. Next, let's look at a family of filters that are easier to parameterize for subtle, natural-looking results.

Making subtle color shifts with curves

When looking at a scene, we may pick up subtle cues from the way colors shift between different image regions. For example, on a clear day outside, shadows have a slightly blue tint due to the ambient light reflected from the blue sky, while highlights have a slightly yellow tint because they are in direct sunlight. When we see bluish shadows and yellowish highlights in a photograph, we may get a "warm and sunny" feeling. This effect may be natural, or it may be exaggerated by a filter.

Curve filters are useful for this type of manipulation. A curve filter is parameterized by sets of control points. For example, there might be one set of control points for each color channel. Each control point is a pair of numbers that represents the input and output values of the given channel. For example, the pair `(128, 180)` means that a value of `128` in the given color channel is brightened to become a value of `180`. Values between the control points are interpolated along a curve (hence, the name, curve filter). In GIMP, a curve with the control points `(0, 0)`, `(128, 180)`, and `(255, 255)` is visualized, as shown in the following screenshot:



The x axis shows the input values ranging from 0 to 255, while the y axis shows the output values over the same range. Besides showing the curve, the graph shows the line `y = x` (no change) for comparison.

Curvilinear interpolation helps to ensure that color transitions are smooth, not abrupt. Thus, a curve filter makes it relatively easy to create subtle, natural-looking effects. We may define an RGB curve filter as follows, in pseudocode:

```
dst.r = funcR(src.r) where funcR interpolates pointsR
dst.g = funcG(src.g) where funcG interpolates pointsG
dst.b = funcB(src.b) where funcB interpolates pointsB
```

For now, we will work with RGB and RGBA curve filters and with channel values that range from 0 to 255. If we want such a curve filter to produce natural-looking results, we should use the following rules of thumb:

- Every set of control points should include `(0, 0)` and `(255, 255)`. This way, black remains black, white remains white, and the image does not appear to have an overall tint.
- As the input value increases, the output value should always increase too. (Their relationship should be monotonically increasing.) This way, shadows remain shadows, highlights remain highlights, and the image does not appear to have inconsistent lighting or contrast.

OpenCV does not provide curvilinear interpolation functions, but the Apache Commons Math library does. (See *Adding files to the project*, earlier in this chapter, for instructions on setting up Apache Commons Math.) This library provides interfaces called `UnivariateInterpolator` and `UnivariateFunction`, which have implementations including `LinearInterpolator`, `SplineInterpolator`, `LinearFunction`, and `PolynomialSplineFunction`. (Splines are a type of curve.) `UnivariateInterpolator` has an instance method, `interpolate(double[] xval, double[] yval)`, which takes arrays of input and output values for the control points and returns a `UnivariateFunction` object. The `UnivariateFunction` object can provide interpolated values via the method `value(double x)`.

API documentation for Apache Commons Math is available at `http://commons.apache.org/proper/commons-math/apidocs/`.

These interpolation functions are computationally expensive. We do not want to run them again and again for every channel of every pixel in every frame. Fortunately, we do not have to. There are only 256 possible input values per channel, so it is practical to precompute all possible output values and store them in a **lookup table**. For OpenCV's purposes, a lookup table is a `Mat` object whose indices represent input values and whose elements represent output values. The lookup can be performed using the static method `Core.LUT(Mat src, Mat lut, Mat dst)`. In pseudocode, `dst = lut[src]`. The number of elements in `lut` should match the range of values in `src`, and the number of channels in `lut` should match the number of channels in `src`.

Now, using Apache Commons Math and OpenCV, let's implement a curve filter for RGBA images with channel values ranging from `0` to `255`. Open `CurveFilter.java` and write the following code:

```
public class CurveFilter implements Filter {
  // The lookup table.
  private final Mat mLUT = new MatOfInt();
  public CurveFilter(
    final double[] vValIn, final double[] vValOut,
    final double[] rValIn, final double[] rValOut,
    final double[] gValIn, final double[] gValOut,
    final double[] bValIn, final double[] bValOut) {
    // Create the interpolation functions.
    UnivariateFunction vFunc = newFunc(vValIn, vValOut);
    UnivariateFunction rFunc = newFunc(rValIn, rValOut);
    UnivariateFunction gFunc = newFunc(gValIn, gValOut);
    UnivariateFunction bFunc = newFunc(bValIn, bValOut);
    // Create and populate the lookup table.
    mLUT.create(256, 1, CvType.CV_8UC4);
    for (int i = 0; i < 256; i++) {
      final double v = vFunc.value(i);
      final double r = rFunc.value(v);
      final double g = gFunc.value(v);
      final double b = bFunc.value(v);
      mLUT.put(i, 0, r, g, b, i); // alpha is unchanged
    }
  }
  @Override
  public void apply(final Mat src, final Mat dst) {
    // Apply the lookup table.
    Core.LUT(src, mLUT, dst);
  }
  private UnivariateFunction newFunc(final double[] valIn,
```

```
      final double[] valOut) {
      UnivariateInterpolator interpolator;
      if (valIn.length > 2) {
        interpolator = new SplineInterpolator();
      } else {
        interpolator = new LinearInterpolator();
      }
      return interpolator.interpolate(valIn, valOut);
    }
  }
```

`CurveFilter` stores the lookup table in a member variable. The constructor method populates the lookup table based on the four sets of control points that are taken as arguments. In addition to a set of control points for each of the RGB channels, the constructor also takes a set of control points for the image's overall brightness, just for convenience. A helper method, `newFunc`, creates an appropriate interpolation function (linear or spline) for each set of control points. Then, we iterate over the possible input values and populate the lookup table.

The `apply` method is a one-liner. It simply uses the precomputed lookup table with the given source and destination matrices.

`CurveFilter` can be extended in a subclass to define a filter with a specific set of control points. For example, let's open `PortraCurveFilter.java` and write the following code:

```
public class PortraCurveFilter extends CurveFilter {
  public PortraCurveFilter() {
    super(
      new double[] { 0, 23, 157, 255 }, // vValIn
      new double[] { 0, 20, 173, 255 }, // vValOut
      new double[] { 0, 69, 213, 255 }, // rValIn
      new double[] { 0, 69, 218, 255 }, // rValOut
      new double[] { 0, 52, 189, 255 }, // gValIn
      new double[] { 0, 47, 196, 255 }, // gValOut
      new double[] { 0, 41, 231, 255 }, // bValIn
      new double[] { 0, 46, 228, 255 }); // bValOut
  }
}
```

This filter brightens the image, makes shadows cooler (more blue), and makes highlights warmer (more yellow). It produces flattering skin tones and tends to make things look sunnier and cleaner. It resembles the color characteristics of a brand of photo film called Kodak Portra, which was often used for portraits.

The code for our other three channel mixing filters is similar. The `ProviaCurveFilter` class uses the following arguments for its control points:

```
new double[] { 0, 255 }, // vValIn
new double[] { 0, 255 }, // vValOut
new double[] { 0, 59, 202, 255 }, // rValIn
new double[] { 0, 54, 210, 255 }, // rValOut
new double[] { 0, 27, 196, 255 }, // gValIn
new double[] { 0, 21, 207, 255 }, // gValOut
new double[] { 0, 35, 205, 255 }, // bValIn
new double[] { 0, 25, 227, 255 }); // bValOut
```

The effect of this filter is to increase the contrast between shadows and highlights and make the image slightly cooler (bluish) throughout most of the tones. Sky, water, and shade are accentuated more than the sun. It resembles a brand of photo film called Fuji Provia, which was often used for landscapes. For example, the following photo has been taken on Provia film, which accentuates the bluish, misty background in an otherwise sunny scene:

The `VelviaCurveFilter` class uses the following arguments for its control points:

```
new double[] { 0, 128, 221, 255 }, // vValIn
new double[] { 0, 118, 215, 255 }, // vValOut
new double[] { 0, 25, 122, 165, 255 }, // rValIn
```

```
new double[] { 0, 21, 153, 206, 255 }, // rValOut
new double[] { 0, 25, 95, 181, 255 }, // gValIn
new double[] { 0, 21, 102, 208, 255 }, // gValOut
new double[] { 0, 35, 205, 255 }, // bValIn
new double[] { 0, 25, 227, 255 }); // bValOut
```

The effect of this filter is to produce deep shadows and vivid colors. It resembles a brand of photo film called Fuji Velvia, which was often used to depict landscapes, with azure skies in daytime or crimson clouds at sunset. The next photo has been taken on Velvia film and, in this sunny scene, we can see Velvia's distinctive deep shadows and azure skies (or slate gray skies in the black-and-white print edition):

Finally, the `CrossProcessCurveFilter` class uses the following arguments for its control points:

```
new double[] { 0, 255 }, // vValIn
new double[] { 0, 255 }, // vValOut
new double[] { 0, 56, 211, 255 }, // rValIn
new double[] { 0, 22, 255, 255 }, // rValOut
new double[] { 0, 56, 208, 255 }, // gValIn
new double[] { 0, 39, 226, 255 }, // gValOut
new double[] { 0, 255 }, // bValIn
new double[] { 20, 235 }); // bValOut
```

The effect is a strong, blue or greenish-blue tint in shadows and a strong, yellow or greenish-yellow tint in highlights. It resembles a film processing technique called cross-processing, which was sometimes used to produce grungy-looking photos of fashion models, pop stars, and so on.

For a good discussion of how to emulate various brands of photo film, see Petteri Sulonen's blog at `http://www.prime-junta.net/pont/How_to/100_Curves_and_Films/_Curves_and_films.html`. The control points that we use are based on the examples given in this article.

The following strip of screenshots is a showcase of our curve filters. Some of the differences are subtle. From left to right, we see an unfiltered image, followed by images filtered with `PortraCurveFilter`, `ProviaCurveFilter`, `VelviaCurveFilter`, and `CrossProcessCurveFilter`:

The differences between these curve filters are not obvious in black-and-white prints. For color images, please see the eBook.

Curve filters are a convenient tool for manipulating color and contrast, but they are limited insofar as each destination pixel is affected by only a single input pixel. Next, we will examine a more flexible family of filters, which enable each destination pixel to be affected by a neighborhood of input pixels.

Mixing pixels with convolution filters

For a convolution filter, the channel values at each output pixel are a weighted average of the corresponding channel values in a neighborhood of input pixels. We can put the weights in a matrix called a **convolution matrix** or **kernel**. For example, consider the following kernel:

```
{{ 0, -1,  0},
 {-1,  4, -1},
 { 0, -1,  0}}
```

The central element is the weight of the source pixel that has the same indices as the destination pixel. Other elements represent weights for the rest of the neighborhood of input pixels. Here, we are considering a 3 x 3 neighborhood. However, OpenCV supports kernels with any square and odd-numbered dimensions. This particular kernel is a type of edge-finding filter called a **Laplacian** filter. For a neighborhood of flat color (no contrast), it yields a black output pixel. For a neighborhood of high contrast, it may yield a bright output pixel.

Let's consider another kernel where the central element is greater by 1:

```
{{ 0, -1,  0},
 {-1,  5, -1},
 { 0, -1,  0}}
```

This is equivalent to taking the result of a Laplacian filter and then adding it to the original image. Instead of edge-finding, we get edge-sharpening. That is, edge regions get brighter, while the rest of the image remains unchanged.

Beware big kernels

The bigger the kernel, the more expensive the computation. Kernels larger than 5 x 5 (25 input pixels per output pixel) are probably not practical for processing live, HD video on typical Android devices today.

OpenCV provides many static methods for convolution filters that use certain popular kernels. The following are some examples:

- `Imgproc.blur(Mat src, Mat dst, Size ksize)`: This blurs the image by taking a simple average of a neighborhood of size `ksize`. For example, if `ksize` is `new Size(5, 5)`, the kernel is the following:

  ```
  {{0.04, 0.04, 0.04, 0.04, 0.04},
   {0.04, 0.04, 0.04, 0.04, 0.04},
   {0.04, 0.04, 0.04, 0.04, 0.04},
  ```

```
{0.04, 0.04, 0.04, 0.04, 0.04},
{0.04, 0.04, 0.04, 0.04, 0.04}}
```

- `Imgproc.Laplacian(Mat src, Mat dst, int ddepth, int ksize, double scale, double delta)`: This is a Laplacian edge-finding filter, as described previously. The results are multiplied by a constant (the `scale` argument) and added to another constant (the `delta` argument).
- `Imgproc.Scharr(Mat src, Mat dst, int ddepth, int dx, int dy, double scale, double delta)`: This is a Scharr edge-finding filter. Unlike the Laplacian filter, the Scharr filter only finds edges that run in a particular direction, either vertical (if `dx` is 1 and `dy` is 0) or horizontal (if `dx` is 0 and `dy` is 1). For vertical edge detection, the kernel is the following:

  ```
  {{ -3,  0,   3},
   {-10,  0,  10},
   { -3,  0,   3}}
  ```

  Similarly, for horizontal edge detection, the Scharr kernel is the following:

  ```
  {{ -3, -10, -3},
   {  0,   0,  0},
   {  3,  10,  3}}
  ```

Moreover, OpenCV provides a static method, `Imgproc.filter2D(Mat src, Mat dst, int ddepth, Mat kernel)`, which enables us to specify our own kernels. For learning purposes, we will take this approach. The `ddepth` argument determines the numeric type of the destination's data. This argument may be any of the following:

- `-1`: This means the same numeric type as the source.
- `CvType.CV_16S`: This represents 16-bit signed integers. The source must be 8-bit signed integers.
- `CvType.CV_32F`: This represents 32-bit floats.
- `CvType.CV_64F`: This represents 64-bit floats. The source must also be 64-bit floats.

For more information about all the preceding filter functions, see the main documentation of OpenCV's Imgproc module at `http://docs.opencv.org/java/org/opencv/imgproc/Imgproc.html`. This document includes mathematical descriptions of the filters. Also see Imgproc's official tutorials at `http://docs.opencv.org/doc/tutorials/imgproc/table_of_content_imgproc/table_of_content_imgproc.html` for examples covering a wide range of filters.

For more information on CvType's members, see the relevant Javadoc at `http://docs.opencv.org/java/org/opencv/core/CvType.html`.

Let's use a convolution filter as part of a more complex filter that draws heavy, black lines on top of edge regions in the image. To achieve this effect, we also rely on two more static methods from OpenCV:

- `Core.bitwise_not(Mat src, Mat dst)`: This method inverts the image's brightness and colors so that white becomes black, red becomes cyan, and so on. It is useful to us because our convolution filter will produce white edges on a black field, whereas we want the opposite, black edges on a white field.
- `Core.multiply(Mat s, Mat src2, Mat dst, double scale)`: This method blends a pair of images by multiplying their values together. The resulting values are scaled by a constant (the `scale` argument). For example, `scale` can be used to normalize the product to the [0, 255] range. For our purposes, `Core.multiply` can serve to superimpose the black edges on the original image.

The following is the implementation of the blackened edge effect in `StrokeEdgesFilter`:

```
public class StrokeEdgesFilter implements Filter {
  private final Mat mKernel = new MatOfInt(
    0, 0,   1, 0, 0,
    0, 1,   2, 1, 0,
    1, 2, -16, 2, 1,
    0, 1,   2, 1, 0,
    0, 0,   1, 0, 0
  );
  private final Mat mEdges = new Mat();
  @Override
  public void apply(final Mat src, final Mat dst) {
    Imgproc.filter2D(src, mEdges, -1, mKernel);
    Core.bitwise_not(mEdges, mEdges);
    Core.multiply(src, mEdges, dst, 1.0/255.0);
  }
}
```

For more information about `Mat` and its subclasses, such as `MatOfInt`, see `Mat's` Javadoc at `http://docs.opencv.org/java/org/opencv/core/Mat.html`.

The following pair of screenshots is a comparison between an unfiltered image (left) and an image filtered with `StrokeEdgesFilter` (right):

Next, let's add a user interface for enabling and disabling all our filters.

Adding the filters to CameraActivity

We will let the user have up to one channel mixing filter, one curve filter, and one convolution filter active at any time. For each filter category, we will provide a menu button that lets the user cycle through the available filters or no filter.

Let's start editing the relevant resource files to define the menu buttons and their text. We should add the following strings to `res/values/strings.xml`:

```
<string name="menu_next_curve_filter">Next Curve</string>
<string name="menu_next_mixer_filter">Next Mixer</string>
<string name="menu_next_convolution_filter">Next Kernel</string>
```

Then, we should edit `res/menu/activity_camera.xml` as follows:

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
  <item
```

```
      android:id="@+id/menu_next_curve_filter"
      app:showAsAction="ifRoom|withText"
      android:title="@string/menu_next_curve_filter" />
    <item
      android:id="@+id/menu_next_mixer_filter"
      app:showAsAction="ifRoom|withText"
      android:title="@string/menu_next_mixer_filter" />
    <item
      android:id="@+id/menu_next_convolution_filter"
      app:showAsAction="ifRoom|withText"
      android:title="@string/menu_next_convolution_filter" />
    <item
      android:id="@+id/menu_next_camera"
      app:showAsAction="ifRoom|withText"
      android:title="@string/menu_next_camera" />
    <item
      android:id="@+id/menu_take_photo"
      app:showAsAction="always|withText"
      android:title="@string/menu_take_photo" />
</menu>
```

To store information about the available and selected filters, we need several new variables in `CameraActivity`. The available filters are just `Filter[]` arrays. The indices of the selected filters are stored in the same way as the index of the selected camera device, by serializing and deserializing (saving and restoring) an integer to/from an Android `Bundle` object. The following are the variable declarations that we must add to `CameraActivity`:

```
// Keys for storing the indices of the active filters.
private static final String STATE_CURVE_FILTER_INDEX =
  "curveFilterIndex";
private static final String STATE_MIXER_FILTER_INDEX =
  "mixerFilterIndex";
private static final String STATE_CONVOLUTION_FILTER_INDEX =
  "convolutionFilterIndex";
// The filters.
private Filter[] mCurveFilters;
private Filter[] mMixerFilters;
private Filter[] mConvolutionFilters;
// The indices of the active filters.
private int mCurveFilterIndex;
private int mMixerFilterIndex;
private int mConvolutionFilterIndex;
```

Since our `Filter` implementations rely on classes in OpenCV, they cannot be instantiated until the OpenCV library is loaded. Thus, our `BaseLoaderCallback` object is responsible for initializing the `Filter[]` arrays. We should edit it as follows:

```
private BaseLoaderCallback mLoaderCallback =
  new BaseLoaderCallback(this) {
  @Override
  public void onManagerConnected(final int status) {
    switch (status) {
    case LoaderCallbackInterface.SUCCESS:
      Log.d(TAG, "OpenCV loaded successfully");
      mCameraView.enableView();
      mBgr = new Mat();
      mCurveFilters = new Filter[] {
        new NoneFilter(),
        new PortraCurveFilter(),
        new ProviaCurveFilter(),
        new VelviaCurveFilter(),
        new CrossProcessCurveFilter()
      };
      mMixerFilters = new Filter[] {
        new NoneFilter(),
        new RecolorRCFilter(),
        new RecolorRGVFilter(),
        new RecolorCMVFilter()
      };
      mConvolutionFilters = new Filter[] {
        new NoneFilter(),
        new StrokeEdgesFilter()
      };
      break;
    default:
      super.onManagerConnected(status);
      break;
    }
  }
};
```

The `onCreate` method can initialize the selected filter indices or load them from the `savedInstanceState` argument. Let's edit the method as follows:

```
protected void onCreate(final Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  final Window window = getWindow();
```

```
    window.addFlags(
      WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    if (savedInstanceState != null) {
      mCameraIndex = savedInstanceState.getInt(
        STATE_CAMERA_INDEX, 0);
      mImageSizeIndex = savedInstanceState.getInt(
        STATE_IMAGE_SIZE_INDEX, 0);
      mCurveFilterIndex = savedInstanceState.getInt(
        STATE_CURVE_FILTER_INDEX, 0);
      mMixerFilterIndex = savedInstanceState.getInt(
        STATE_MIXER_FILTER_INDEX, 0);
      mConvolutionFilterIndex = savedInstanceState.getInt(
        STATE_CONVOLUTION_FILTER_INDEX, 0);
    } else {
      mCameraIndex = 0;
      mImageSizeIndex = 0;
      mCurveFilterIndex = 0;
      mMixerFilterIndex = 0;
      mConvolutionFilterIndex = 0;
    }
    // ...
  }
```

Similarly, the `onSaveInstanceState` method should save the selected filter indices to the `savedInstanceState` argument. Let's edit the method as follows:

```
  public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the current camera index.
    savedInstanceState.putInt(STATE_CAMERA_INDEX, mCameraIndex);
    // Save the current image size index.
    savedInstanceState.putInt(STATE_IMAGE_SIZE_INDEX,
      mImageSizeIndex);
    // Save the current filter indices.
    savedInstanceState.putInt(STATE_CURVE_FILTER_INDEX,
      mCurveFilterIndex);
    savedInstanceState.putInt(STATE_MIXER_FILTER_INDEX,
      mMixerFilterIndex);
    savedInstanceState.putInt(STATE_CONVOLUTION_FILTER_INDEX,
      mConvolutionFilterIndex);
    super.onSaveInstanceState(savedInstanceState);
  }
```

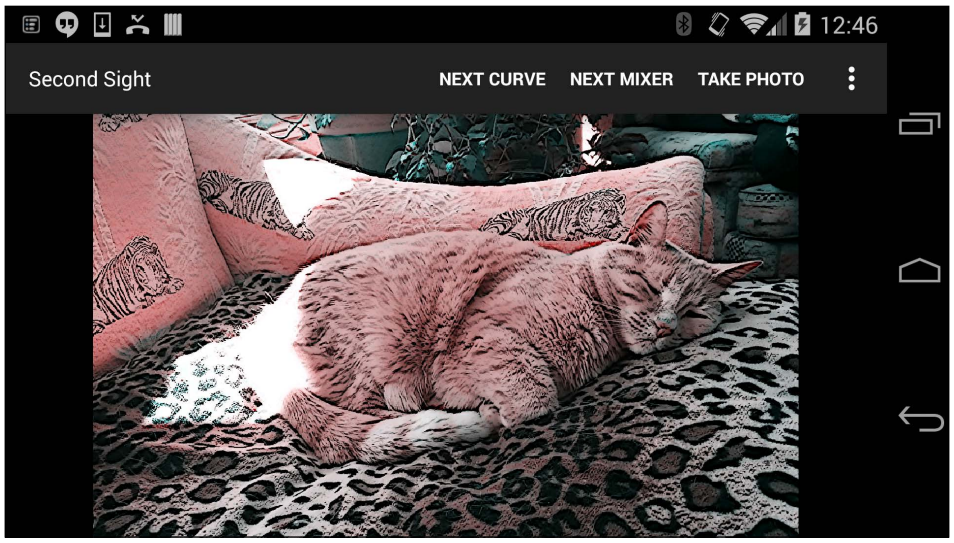
To make each of the new menu items functional, we just need to add some boilerplate code that updates the relevant filter index. Let's edit the `onOptionsItemSelected` method as follows:

```
public boolean onOptionsItemSelected(final MenuItem item) {
  if (mIsMenuLocked) {
    return true;
  }
  if (item.getGroupId() == MENU_GROUP_ID_SIZE) {
    mImageSizeIndex = item.getItemId();
    recreate();
    return true;
  }
  switch (item.getItemId()) {
  case R.id.menu_next_curve_filter:
    mCurveFilterIndex++;
    if (mCurveFilterIndex == mCurveFilters.length) {
      mCurveFilterIndex = 0;
    }
    return true;
  case R.id.menu_next_mixer_filter:
    mMixerFilterIndex++;
    if (mMixerFilterIndex == mMixerFilters.length) {
      mMixerFilterIndex = 0;
    }
    return true;
  case R.id.menu_next_convolution_filter:
    mConvolutionFilterIndex++;
    if (mConvolutionFilterIndex ==
      mConvolutionFilters.length) {
      mConvolutionFilterIndex = 0;
    }
    return true;
  // ...
  default:
    return super.onOptionsItemSelected(item);
  }
}
```

Now, in the `onCameraFrame` callback method, we should apply each selected filter to the image. The following is the new implementation:

```
public Mat onCameraFrame(final CvCameraViewFrame inputFrame) {
  final Mat rgba = inputFrame.rgba();
  // Apply the active filters.
  mCurveFilters[mCurveFilterIndex].apply(rgba, rgba);
  mMixerFilters[mMixerFilterIndex].apply(rgba, rgba);
  mConvolutionFilters[mConvolutionFilterIndex].apply(
    rgba, rgba);
  if (mIsPhotoPending) {
    mIsPhotoPending = false;
    takePhoto(rgba);
  }
  if (mIsCameraFrontFacing) {
    // Mirror (horizontally flip) the preview.
    Core.flip(rgba, rgba, 1);
  }
  return rgba;
}
```

That's all! Run the app, select the filters, take some photos, and share them. As an example of how the app should look, here is a screenshot with `PortraCurveFilter`, `RecolorRCFilter`, and `StrokeEdgesFilter` enabled:



Remember to look under the **...** menu for more options, including **Next Kernel** and **Size**.

Note that the creases in the cat's fur appear very dark due to the use of `StrokeEdgesFilter`. Also, if you are viewing the image in color (in the eBook), note that the ginger cat and yellow sofa both have a reddish tint, while the leaves in the background have a cyan tint due to the use of `RecolorRCFilter`.

Summary

Second Sight now has some functionality that is more interesting than just reading and sharing camera data. Several filters can be selected and combined to give a stylized or vintage look to our photos. These filters are efficient enough to apply to live video too, so we use them in preview mode and on saved photos.

Although photo filters are fun, they are only the most basic use of OpenCV. Before we can truly say we have made a computer vision application, we need to make the app respond differently depending on what it is seeing. This goal will be the focus of the next chapter.

4
Recognizing and Tracking Images

Our goal in this chapter is to add image tracking to Second Sight. We will train the app to recognize certain arbitrary, rectangular images—for example, paintings—and determine their pose in a 2D projection. The app will draw an outline around a tracked image when it appears in the camera feed. All of the tracking and drawing is done using OpenCV, rather than other Android libraries.

The complete Eclipse project for this chapter can be downloaded from the author's website. The project has two versions:

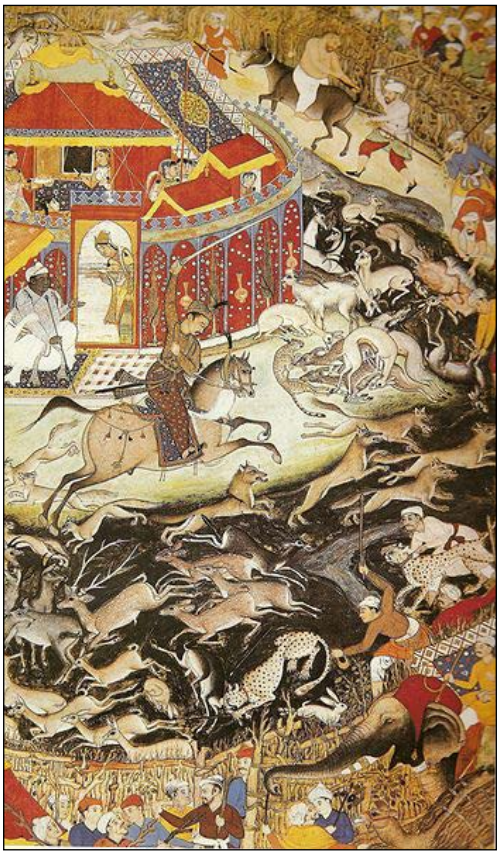
A version for OpenCV 3.x is located at `http://nummist.com/opencv/4598_04.zip`.

A version for OpenCV 2.x version is located at `http://nummist.com/opencv/5206_04.zip`.

Adding files to the project

For this chapter, we need to add one new class, `com.nummist.secondsight.filters.ar.ImageDetectionFilter`. We also need to add some resource files, the images that we want to track. Download the images from `http://nummist.com/opencv/5206_04_images.zip`, unzip them, and put them in the project's `res/drawable-nodpi` folder.

These images are famous paintings by a 19th-century Dutch artist named Vincent van Gogh (`http://en.wikipedia.org/wiki/Vincent_van_Gogh`) and a 16th-century Indian artist named Basawan (`http://en.wikipedia.org/wiki/Basawan`). Our tracker will work well with these images because they contain many high-contrast details, without much repetition of patterns. Thus, there is something distinctive to track in most parts of each image. For example, here is one of Basawan's paintings, *Akbar Hunting with Cheetahs*:



Here is one of Vincent van Gogh's paintings, *The Starry Night*:

Understanding image tracking

Imagine the following conversation:

Person A: I can't find my print of *The Starry Night*. Do you know where it is?

Person B: What does it look like?

For a computer, or for someone who is naive about Western art, Person B's question is quite reasonable. Before we can use our sense of sight (or other senses) to track something, we need to have sensed that thing before. Failing that, we at least need a good description of what we will sense. For computer vision, we must provide a **reference image** that will be compared with the live camera image or **scene**. If the target has complex geometry or moving parts, we might need to provide many reference images to account for different perspectives and poses. However, for our examples using famous paintings, we will assume that the target is rectangular and rigid.

Google's **Search by image** is an example of a popular tool that requires the user to provide a reference image. If you have not used it before, go to `https://images.google.com/`, click the camera icon, and provide a reference image by entering a URL or uploading a file. The search results should include matching or similar images. This tool may help a researcher who wants to verify the original source of an image.

For this chapter's purposes, let's say that the goal of tracking is to determine how our rectangular target is posed in 3D. With this information, we can draw an outline around our target. In the final 2D image, the outline will be a quadrilateral. (It will not necessarily be a rectangle, since the target could be skewed away from the camera.)

There are four major steps involved in this type of tracking:

1. Find **features** in the reference image and scene. A feature is a point that is likely to maintain a similar appearance when viewed from different distances or angles. For example, corners often have this characteristic. See `http://en.wikipedia.org/wiki/Feature_%28computer_vision%29` and `http://en.wikipedia.org/wiki/Feature_extraction`.
2. Find **descriptors** for each set of features (the reference and scene features). A descriptor is a vector of data about a feature. Some features are not suitable for generating a descriptor, so an image has fewer descriptors than features. See `http://en.wikipedia.org/wiki/Visual_descriptors`.
3. Find **matches** between the two sets of descriptors (the reference and scene descriptors). If we imagine the descriptors as points in a multidimensional space, a match is defined in terms of a measure of distance between the points. Descriptors that are close enough to each other are considered a match. When a pair of descriptors is a match, we may also say that the underlying pair of features is a match.
4. Find the **homography** between a reference image and a matching image in the scene. A homography is a 3D transformation that would be necessary to line up the two projected 2D images (or come as close as possible to lining them up). It is calculated based on the two images' matching feature points. By applying the homography to a rectangle, we can get an outline of the tracked object. See `http://en.wikipedia.org/wiki/Homography_%28computer_vision%29`.

There are many different techniques to perform each of the first three steps. OpenCV provides relevant classes called `FeatureDetector` (`http://docs.opencv.org/java/org/opencv/features2d/FeatureDetector.html`), `DescriptorExtractor` (`http://docs.opencv.org/java/org/opencv/features2d/DescriptorExtractor.html`), and `DescriptorMatcher` (`http://docs.opencv.org/java/org/opencv/features2d/DescriptorMatcher.html`), and each supports several techniques. We will use a combination of techniques that OpenCV calls `FeatureDetector.ORB`, `DescriptorExtractor.ORB`, and `DescriptorMatcher.BRUTEFORCE_HAMMINGLUT`. This combination is relatively fast and robust. Unlike some alternatives, it is **scale-invariant** and **rotation-invariant**, which means that the target can be tracked at various magnifications, resolutions, or distances, and at various angles of view. Also, unlike some other alternatives, it is not patented, so it is free to use even in commercial applications.

For a description of **ORB**, other techniques, and their relative merits, see the multipart *Tutorial on Binary Descriptors* on Gil Levi's blog, starting at `https://gilscvblog.wordpress.com/2013/08/26/tutorial-on-binary-descriptors-part-1/`. As a precursor to this topic, see also his *A Short introduction to descriptors* at `https://gilscvblog.wordpress.com/2013/08/18/a-short-introduction-to-descriptors/`.

Another good source on ORB and the alternatives is the paper *A Comparative Evaluation of Well-known Feature Detectors and Descriptors* by Şahin Işık and Kemal Özkan. An electronic version of this paper is available at `http://www.atscience.org/IJAMEC/article/download/135/83`.

Finally, for benchmarks of ORB and alternatives as implemented in OpenCV, see Ievgen Khvedchenya's blog posts on feature detectors at `http://computer-vision-talks.com/articles/2011-07-13-comparison-of-the-opencv-feature-detection-algorithms/` and on descriptor extractors at `http://computer-vision-talks.com/articles/2011-01-28-comparison-of-feature-descriptors/`.

At the time of writing, OpenCV's feature detectors support only grayscale images. For our purposes, this limitation is not so bad because algorithms for grayscale feature detection are faster than algorithms for color feature detection. On a mobile device, we must economize in order to maintain a reasonable speed for live video, and running computer vision functions on a grayscale version of each frame is a way to economize. However, for the best results, we should make sure that we choose target images that still have strong contrast when converted to grayscale.

The following pseudocode represents the standard grayscale conversion formula used in OpenCV:

```
grayValue(r, g, b) = (0.299 * r) + (0.587 * g) + (0.114 * b)
```

The preceding formula is most effective at preserving blue-yellow contrast (where yellow is red plus green) and of course black-white contrast. For example, this standard grayscale conversion does a great job of preserving the contrast in *The Starry Night*, which has a palette composed primarily of blue, yellow, black, and white.

Writing an image tracking filter

We will write our tracker as an implementation of the `Filter` interface, which we created in the previous chapter. The tracker's class name will be `ImageDetectionFilter`. As member variables, this class has instances of `FeatureDetector`, `DescriptorExtractor`, and `DescriptorMatcher`, as well as several `Mat` instances that store image data and the intermediate or final results of tracking calculations. Some of these results are stored because they do not change from frame to frame. Others are stored simply because it is more efficient than recreating the `Mat` instance for each frame. The declarations of the class and member variables are as follows:

```
public class ImageDetectionFilter implements Filter {

  // The reference image (this detector's target).
  private final Mat mReferenceImage;
  // Features of the reference image.
  private final MatOfKeyPoint mReferenceKeypoints =
    new MatOfKeyPoint();
  // Descriptors of the reference image's features.
  private final Mat mReferenceDescriptors = new Mat();
  // The corner coordinates of the reference image, in pixels.
  // CvType defines the color depth, number of channels, and
  // channel layout in the image. Here, each point is represented
  // by two 32-bit floats.
  private final Mat mReferenceCorners =
    new Mat(4, 1, CvType.CV_32FC2);

  // Features of the scene (the current frame).
  private final MatOfKeyPoint mSceneKeypoints =
    new MatOfKeyPoint();
  // Descriptors of the scene's features.
  private final Mat mSceneDescriptors = new Mat();
  // Tentative corner coordinates detected in the scene, in
```

```
// pixels.
private final Mat mCandidateSceneCorners =
  new Mat(4, 1, CvType.CV_32FC2);
// Good corner coordinates detected in the scene, in pixels.
private final Mat mSceneCorners = new Mat(4, 1,
  CvType.CV_32FC2);
// The good detected corner coordinates, in pixels, as integers.
private final MatOfPoint mIntSceneCorners = new MatOfPoint();

// A grayscale version of the scene.
private final Mat mGraySrc = new Mat();
// Tentative matches of scene features and reference features.
private final MatOfDMatch mMatches = new MatOfDMatch();

// A feature detector, which finds features in images.
private final FeatureDetector mFeatureDetector =
  FeatureDetector.create(FeatureDetector.ORB);
// A descriptor extractor, which creates descriptors of
// features.
private final DescriptorExtractor mDescriptorExtractor =
  DescriptorExtractor.create(DescriptorExtractor.ORB);
// A descriptor matcher, which matches features based on their
// descriptors.
private final DescriptorMatcher mDescriptorMatcher =
  DescriptorMatcher.create(
    DescriptorMatcher.BRUTEFORCE_HAMMINGLUT);

// The color of the outline drawn around the detected image.
private final Scalar mLineColor = new Scalar(0, 255, 0);
```

We want a convenient way to make an image tracker for any arbitrary image. We can package images with our app as **drawable** resources, which can be loaded by any Android `Context` subclass such as `Activity`. Thus, we provide a constructor, `ImageDetectionFilter(final Context context, final int referenceImageResourceID)`, which loads the reference image with the given `Context` and resource identifier. RGBA and grayscale versions of the image are stored in the member variables. The image's corner points are also stored, and so are its features and descriptors. The constructor's code is as follows:

```
public ImageDetectionFilter(final Context context,
  final int referenceImageResourceID) throws IOException {

  // Load the reference image from the app's resources.
  // It is loaded in BGR (blue, green, red) format.
```

```
    mReferenceImage = Utils.loadResource(context,
      referenceImageResourceID,
      Imgcodecs.CV_LOAD_IMAGE_COLOR);

    // Create grayscale and RGBA versions of the reference image.
    final Mat referenceImageGray = new Mat();
    Imgproc.cvtColor(mReferenceImage, referenceImageGray,
      Imgproc.COLOR_BGR2GRAY);
    Imgproc.cvtColor(mReferenceImage, mReferenceImage,
      Imgproc.COLOR_BGR2RGBA);

    // Store the reference image's corner coordinates, in pixels.
    mReferenceCorners.put(0, 0,
      new double[] {0.0, 0.0});
    mReferenceCorners.put(1, 0,
      new double[] {referenceImageGray.cols(), 0.0});
    mReferenceCorners.put(2, 0,
      new double[] {referenceImageGray.cols(),
      referenceImageGray.rows()});
    mReferenceCorners.put(3, 0,
      new double[] {0.0, referenceImageGray.rows()});

    // Detect the reference features and compute their
    // descriptors.
    mFeatureDetector.detect(referenceImageGray,
      mReferenceKeypoints);
    mDescriptorExtractor.compute(referenceImageGray,
      mReferenceKeypoints, mReferenceDescriptors);
  }
```

Adapting the code to OpenCV 2.x

Replace `Imgcodecs.CV_LOAD_IMAGE_COLOR` with `Highgui.CV_LOAD_IMAGE_COLOR`.

Recall that the `Filter` interface declares a method, `apply(final Mat src, final Mat dst)`. Our implementation of this method applies the feature detector, descriptor extractor, and descriptor matcher to a grayscale version of the source image. Then, we call helper functions that find the four corners of the tracked target (if any), and draw the quadrilateral outline. The code is as follows:

```
  @Override
  public void apply(final Mat src, final Mat dst) {
```

```
    // Convert the scene to grayscale.
    Imgproc.cvtColor(src, mGraySrc, Imgproc.COLOR_RGBA2GRAY);

    // Detect the scene features, compute their descriptors,
    // and match the scene descriptors to reference descriptors.
    mFeatureDetector.detect(mGraySrc, mSceneKeypoints);
    mDescriptorExtractor.compute(mGraySrc, mSceneKeypoints,
      mSceneDescriptors);
    mDescriptorMatcher.match(mSceneDescriptors,
      mReferenceDescriptors, mMatches);

    // Attempt to find the target image's corners in the scene.
    findSceneCorners();

    // If the corners have been found, draw an outline around the
    // target image.
    // Else, draw a thumbnail of the target image.    draw(src, dst);
  }
```

The `findSceneCorners()` helper method is a bigger block of code, but a lot of it simply iterates through the matches to assemble a list of the best ones. If all the matches are really bad (as indicated by a large distance value), we assume that the target is not in the scene and we clear any previous estimate of its corner locations. If the matches are neither bad nor good, we assume that the target is somewhere in the scene, but we keep our previous estimate of its corner locations. This policy helps us stabilize the estimate of the corner locations. Finally, if the matches are good and there are at least four of them, we find the homography and use it to update the estimated corner locations.

For a mathematical description of finding the homography, see the official OpenCV documentation at `http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findhomography`.

The implementation of `findSceneCorners()` is as follows:

```
  private void findSceneCorners() {

    List<DMatch> matchesList = mMatches.toList();
    if (matchesList.size() < 4) {
      // There are too few matches to find the homography.
      return;
    }
```

```
List<KeyPoint> referenceKeypointsList =
  mReferenceKeypoints.toList();
List<KeyPoint> sceneKeypointsList =
  mSceneKeypoints.toList();

// Calculate the max and min distances between keypoints.
double maxDist = 0.0;
double minDist = Double.MAX_VALUE;
for(DMatch match : matchesList) {
  double dist = match.distance;
  if (dist < minDist) {
    minDist = dist;
  }
  if (dist > maxDist) {
    maxDist = dist;
  }
}

// The thresholds for minDist are chosen subjectively
// based on testing. The unit is not related to pixel
// distances; it is related to the number of failed tests
// for similarity between the matched descriptors.
if (minDist > 50.0) {
  // The target is completely lost.
  // Discard any previously found corners.
  mSceneCorners.create(0, 0, mSceneCorners.type());
  return;
} else if (minDist > 25.0) {
  // The target is lost but maybe it is still close.
  // Keep any previously found corners.
  return;
}

// Identify "good" keypoints based on match distance.
ArrayList<Point> goodReferencePointsList =
  new ArrayList<Point>();
ArrayList<Point> goodScenePointsList =
  new ArrayList<Point>();
double maxGoodMatchDist = 1.75 * minDist;
for(DMatch match : matchesList) {
  if (match.distance < maxGoodMatchDist) {
    goodReferencePointsList.add(
      referenceKeypointsList.get(match.trainIdx).pt);
    goodScenePointsList.add(
```

```
            sceneKeypointsList.get(match.queryIdx).pt);
        }
      }

      if (goodReferencePointsList.size() < 4 ||
          goodScenePointsList.size() < 4) {
        // There are too few good points to find the homography.
        return;
      }

      // There are enough good points to find the homography.
      // (Otherwise, the method would have already returned.)

      // Convert the matched points to MatOfPoint2f format, as
      // required by the Calib3d.findHomography function.
      MatOfPoint2f goodReferencePoints = new MatOfPoint2f();
      goodReferencePoints.fromList(goodReferencePointsList);
      MatOfPoint2f goodScenePoints = new MatOfPoint2f();
      goodScenePoints.fromList(goodScenePointsList);

      // Find the homography.
      Mat homography = Calib3d.findHomography(
        goodReferencePoints, goodScenePoints);

      // Use the homography to project the reference corner
      // coordinates into scene coordinates.
      Core.perspectiveTransform(mReferenceCorners,
        mCandidateSceneCorners, homography);

      // Convert the scene corners to integer format, as required
      // by the Imgproc.isContourConvex function.
      mCandidateSceneCorners.convertTo(mIntSceneCorners,
        CvType.CV_32S);

      // Check whether the corners form a convex polygon. If not,
      // (that is, if the corners form a concave polygon), the
      // detection result is invalid because no real perspective can
      // make the corners of a rectangular image look like a concave
      // polygon!
      if (Imgproc.isContourConvex(mIntSceneCorners)) {
        // The corners form a convex polygon, so record them as
        // valid scene corners.
        mCandidateSceneCorners.copyTo(mSceneCorners);
      }
    }
```

Our other helper method, `draw(Mat src, Mat dst)`, starts by copying the source image to the destination. Then, if the target is not being tracked, we draw a thumbnail of it in a corner of the image, so that the user knows what to look for. If the target is being tracked, we draw an outline around it. The following pair of screenshots shows the result of the `draw` method when the target is not being tracked (left-hand side) and when it is being tracked (right-hand side):

The following code implements the `draw` helper method:

```
protected void draw(Mat src, Mat dst) {

  if (dst != src) {
    src.copyTo(dst);
  }

  if (mSceneCorners.height() < 4) {
    // The target has not been found.

    // Draw a thumbnail of the target in the upper-left
    // corner so that the user knows what it is.

    // Compute the thumbnail's larger dimension as half the
    // video frame's smaller dimension.
    int height = mReferenceImage.height();
    int width = mReferenceImage.width();
    int maxDimension = Math.min(dst.width(),
      dst.height()) / 2;
    double aspectRatio = width / (double)height;
    if (height > width) {
      height = maxDimension;
```

```
            width = (int)(height * aspectRatio);
          } else {
            width = maxDimension;
            height = (int)(width / aspectRatio);
          }

          // Select the region of interest (ROI) where the thumbnail
          // will be drawn.
          Mat dstROI = dst.submat(0, height, 0, width);
          // Copy a resized reference image into the ROI.
          Imgproc.resize(mReferenceImage, dstROI, dstROI.size(),
            0.0, 0.0, Imgproc.INTER_AREA);

          return;
        }

        // Outline the found target in green.
        Imgproc.line(dst, new Point(mSceneCorners.get(0, 0)),
          new Point(mSceneCorners.get(1, 0)), mLineColor, 4);
        Imgproc.line(dst, new Point(mSceneCorners.get(1, 0)),
          new Point(mSceneCorners.get(2, 0)), mLineColor, 4);
        Imgproc.line(dst, new Point(mSceneCorners.get(2, 0)),
          new Point(mSceneCorners.get(3, 0)), mLineColor, 4);
        Imgproc.line(dst, new Point(mSceneCorners.get(3,0)),
          new Point(mSceneCorners.get(0, 0)), mLineColor, 4);
      }
    }
```

Adapting the code to OpenCV 2.x

Replace `Imgproc.line` with `Core.line`.

Although `ImageDetectionFilter` has a more complicated implementation than our previous filters, it still has a simple interface. You have to instantiate it with a drawable resource and then apply the filter to source and destination images as required.

Adding the tracker filters to CameraActivity

To use instances of `ImageDetectionFilter`, we make the same kind of modifications to `CameraActivity` that we did to the other filters in the previous chapter. Recall that all our filter classes implement the `Filter` interface so that `CameraActivity` can use them all in similar ways.

First, we need to define some text (for the menu button) in `res/values/strings.xml`:

```
<string name="menu_next_image_detection_filter">Next
  Tracker</string>
```

Next, we need to define the menu button itself in `res/menu/activity_camera.xml`:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <item
    android:id="@+id/menu_next_image_detection_filter"
    app:showAsAction="ifRoom|withText"
    android:title="@string/menu_next_image_detection_filter" />
  <!-- ... -->
</menu>
```

The rest of our modifications pertain to `CameraActivity.java`. We need to add new member variables to keep track of the selected image detection filter:

```
// Keys for storing the indices of the active filters.
private static final String STATE_IMAGE_DETECTION_FILTER_INDEX =
  "imageDetectionFilterIndex";
private static final String STATE_CURVE_FILTER_INDEX =
  "curveFilterIndex";
private static final String STATE_MIXER_FILTER_INDEX =
  "mixerFilterIndex";
private static final String STATE_CONVOLUTION_FILTER_INDEX =
  "convolutionFilterIndex";

// The filters.
private Filter[] mImageDetectionFilters;
private Filter[] mCurveFilters;
private Filter[] mMixerFilters;
private Filter[] mConvolutionFilters;

// The indices of the active filters.
private int mImageDetectionFilterIndex;
```

```
private int mCurveFilterIndex;
private int mMixerFilterIndex;
private int mConvolutionFilterIndex;
```

Once OpenCV is initialized, we need to instantiate all the image detection filters and put them in an array. For brevity, I added two image detection filters as examples, but you can easily modify the following code to support the tracking of more images, or different images:

```
public void onManagerConnected(final int status) {
  switch (status) {
    case LoaderCallbackInterface.SUCCESS:
      Log.d(TAG, "OpenCV loaded successfully");
      mCameraView.enableView();
      mBgr = new Mat();

      final Filter starryNight;
      try {
        starryNight = new ImageDetectionFilter(
          CameraActivity.this,
          R.drawable.starry_night);
      } catch (IOException e) {
        Log.e(TAG, "Failed to load drawable: " +
          "starry_night");
        e.printStackTrace();
        break;
      }

      final Filter akbarHunting;
      try {
        akbarHunting = new ImageDetectionFilter(
          CameraActivity.this,
          R.drawable.akbar_hunting_with_cheetahs);
      } catch (IOException e) {
        Log.e(TAG, "Failed to load drawable: " +
          "akbar_hunting_with_cheetahs");
        e.printStackTrace();
        break;
      }

      mImageDetectionFilters = new Filter[] {
        new NoneFilter(),
        starryNight,
        akbarHunting
      };
```

```
            // ...
        }
      }
    };
```

When the activity is created, we need to load any saved data about the selected image detection filter:

```
    protected void onCreate(final Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      final Window window = getWindow();
      window.addFlags(
        WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

      if (savedInstanceState != null) {
        mCameraIndex = savedInstanceState.getInt(
          STATE_CAMERA_INDEX, 0);
        mImageSizeIndex = savedInstanceState.getInt(
          STATE_IMAGE_SIZE_INDEX, 0);
        mImageDetectionFilterIndex =
    savedInstanceState.getInt(
      STATE_IMAGE_DETECTION_FILTER_INDEX, 0);
        mCurveFilterIndex = savedInstanceState.getInt(
          STATE_CURVE_FILTER_INDEX, 0);
        mMixerFilterIndex = savedInstanceState.getInt(
          STATE_MIXER_FILTER_INDEX, 0);
        mConvolutionFilterIndex = savedInstanceState.getInt(
          STATE_CONVOLUTION_FILTER_INDEX, 0);
      } else {
        mCameraIndex = 0;
        mImageSizeIndex = 0;
        mImageDetectionFilterIndex = 0;
        mCurveFilterIndex = 0;
        mMixerFilterIndex = 0;
        mConvolutionFilterIndex = 0;
      }

      // ...
    }
```

Conversely, before the activity is destroyed, we need to save data about the selected image detection filter:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
  // Save the current camera index.
  savedInstanceState.putInt(STATE_CAMERA_INDEX, mCameraIndex);

  // Save the current image size index.
  savedInstanceState.putInt(STATE_IMAGE_SIZE_INDEX,
    mImageSizeIndex);

  // Save the current filter indices.
  savedInstanceState.putInt(STATE_IMAGE_DETECTION_FILTER_INDEX,
    mImageDetectionFilterIndex);
  savedInstanceState.putInt(STATE_CURVE_FILTER_INDEX,
    mCurveFilterIndex);
  savedInstanceState.putInt(STATE_MIXER_FILTER_INDEX,
    mMixerFilterIndex);
  savedInstanceState.putInt(STATE_CONVOLUTION_FILTER_INDEX,
    mConvolutionFilterIndex);

  super.onSaveInstanceState(savedInstanceState);
}
```

When the **Next Tracker** menu button is pressed, the selected image detection filter needs to be updated:

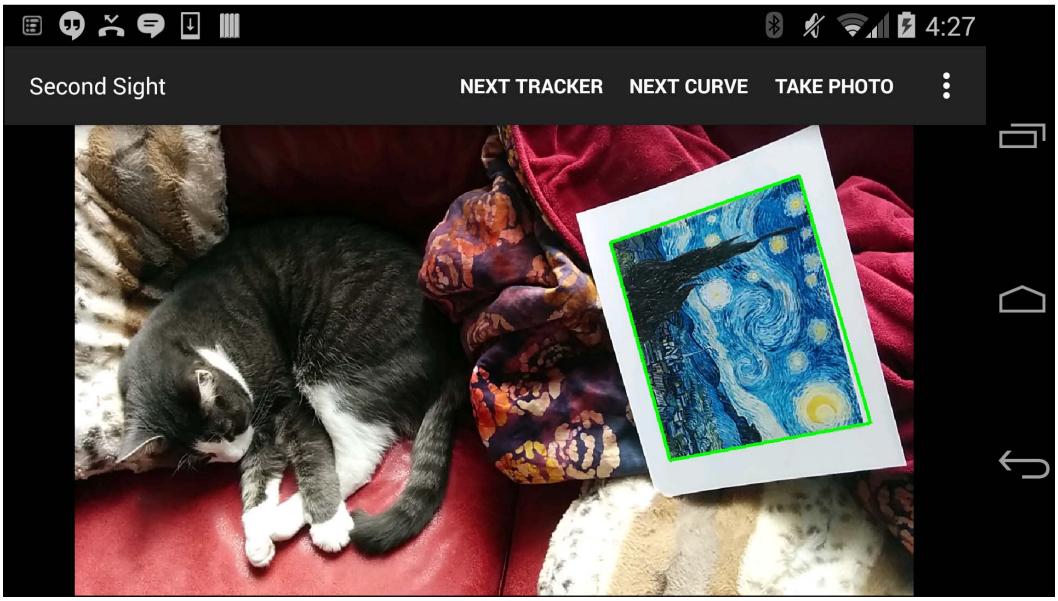
```
public boolean onOptionsItemSelected(final MenuItem item) {
  if (mIsMenuLocked) {
    return true;
  }
  if (item.getGroupId() == MENU_GROUP_ID_SIZE) {
    mImageSizeIndex = item.getItemId();
    recreate();
    return true;
  }
  switch (item.getItemId()) {
  case R.id.menu_next_image_detection_filter:
    mImageDetectionFilterIndex++;
    if (mImageDetectionFilterIndex ==
      mImageDetectionFilters.length) {
      mImageDetectionFilterIndex = 0;
    }
    return true;
```

```
    // ...
    default:
      return super.onOptionsItemSelected(item);
    }
  }
```

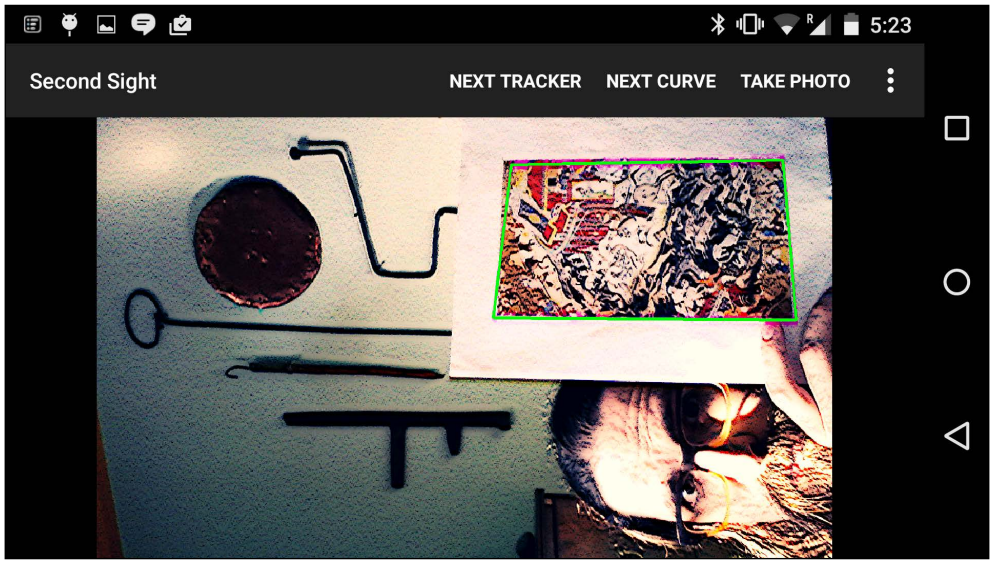
Finally, when the camera captures a frame, the selected image detection filter needs to be applied to the frame. To ensure that other filters do not interfere with image detection, it is important to apply the image detection filter first:

```
  public Mat onCameraFrame(final CvCameraViewFrame inputFrame) {
    final Mat rgba = inputFrame.rgba();

    // Apply the active filters.
    if (mImageDetectionFilters != null) {
      mImageDetectionFilters[mImageDetectionFilterIndex].apply(
        rgba, rgba);
    }
    if (mCurveFilters != null) {
      mCurveFilters[mCurveFilterIndex].apply(rgba, rgba);
    }
    if (mMixerFilters != null) {
      mMixerFilters[mMixerFilterIndex].apply(rgba, rgba);
    }
    if (mConvolutionFilters != null) {
      mConvolutionFilters[mConvolutionFilterIndex].apply(
        rgba, rgba);
    }

    if (mIsPhotoPending) {
      mIsPhotoPending = false;
      takePhoto(rgba);
    }

    if (mIsCameraFrontFacing) {
      // Mirror (horizontally flip) the preview.
      Core.flip(rgba, rgba, 1);
    }

    return rgba;
  }
```

That's all! Print the target images or display them onscreen. Then, run the app, select an appropriate image detection filter, and point the camera at the target. If the video is too choppy (due to the slow processing of the image), select a lower camera resolution and try again. Also, you might need to hold your Android device still for a second or two in order for the camera to autofocus on the target. Then, you should see the target outlined in green. For example, see the outline around *The Starry Night* in the following screenshot:



Note that the image detection filter works even when the target image is rotated or skewed. Now, try combining the image detection filter with other filters. Because we perform image detection before other filtering, the target image still looks the same to the detector and should be detected. For example, in the following screenshot, note that *Akbar Hunting with Cheetahs* is detected (and outlined in green) even while `RecolorRGVFilter`, `CrossProcessCurveFilter`, and `StrokeEdgesFilter` are also running:



Summary

The Second Sight app can see now! At least, it can recognize any image from a predefined set and draw a quadrilateral around that image. To a certain extent, this feature is robust with respect to scale, rotation, and skew. For example, the image can be tracked at various magnifications, resolutions, distances, and angles of view.

Although we added only a single class in this chapter, we covered a lot of OpenCV functionality. Next, we will step back and consider how to integrate this OpenCV functionality with other types of graphics that will respond to camera input in real time. We will build a small 3D-rendered scene atop the image recognition filters of Second Sight.

5
Combining Image Tracking with 3D Rendering

Our goal in this chapter is to combine image tracking with 3D rendering. We will modify our existing image tracker so that it fully determines the target's position and rotation in 3D. Then, using Android SDK's implementation of OpenGL ES, we will draw a colorful 3D cube sitting in front of the tracked image. This is a case of **augmented reality** (**AR**), which means that we are superimposing a virtual object (the cube) onto a specific part of a real scene.

OpenGL (Open Graphics Library) is a standardized, language-independent, platform-independent API for rendering 2D and 3D graphics, normally using the GPU. OpenGL helps developers define the appearance of shapes (geometry), surfaces (materials), and lights from a virtual camera's perspective. Like OpenCV, OpenGL performs computations on matrices. For example, these matrices may store color data or spatial data (including position and rotation). **OpenGL ES** (OpenGL for Embedded Systems) is an OpenGL subset designed for devices with relatively constrained resources, such as smartphones and tablets.

The complete Eclipse project for this chapter can be downloaded from the author's website. The project has two versions:

A version for OpenCV 3.x is located at `http://nummist.com/opencv/4598_05.zip`.

A version for OpenCV 2.x version is located at `http://nummist.com/opencv/5206_05.zip`.

Adding files to the project

For this chapter, we will modify our existing `ImageDetectionFilter` class. We will also add files for the following new classes and interfaces:

- `com.nummist.secondsight.ARCubeRenderer`: This class represents the rendering logic for a colorful cube that sits in front of a tracked, real-world object. The class implements the `GLSurfaceView.Renderer` interface from the Android standard library. The projection matrix is determined by a `CameraProjectionAdapter` instance, and the cube's pose matrix is determined by an `ARFilter` instance, as described later.
- `com.nummist.secondsight.adapters.CameraProjectionAdapter`: This class represents the relationship between a physical camera and a projection matrix. Other classes may fetch the projection matrix in either OpenCV or OpenGL format.
- `com.nummist.secondsight.filters.ar.ARFilter`: This interface represents a filter that captures the position and rotation of a real-world object as an OpenGL matrix. We will modify `ImageDetectionFilter` to implement this interface.
- `com.nummist.secondsight.filters.ar.NoneARFilter`: This class represents a filter that does nothing. It extends the `NoneFilter` class and implements the `ARFilter` interface. We use `NoneARFilter` when we want to turn off filtering but still have an object that conforms to the `ARFilter` interface.

Together, these types support the rendering of a virtual 3D environment that is consistent with certain properties of the real video camera and scene.

Defining the ARFilter interface

Given a source image, our previous filters just produced a destination image. Now, we also want to produce data about the pose (position and rotation) of something that may be visible in the source image. For OpenGL's purposes, a pose is expressed as an array of 16 floating point numbers, representing a 4 x 4 transformation matrix. Thus, we may define the `ARFilter` interface as follows:

If you are unfamiliar with the use of vector and matrix algebra in 3D geometry, you might find parts of this chapter hard to follow. Roughly speaking, you can imagine a transformation matrix as a table containing values that are based on the three coordinates of a 3D position and on trigonometric functions of the three angles of a 3D rotation. Two transformations can be applied consecutively by matrix multiplication. For a primer on these topics, see the online tutorial book *Vector Math for 3D Computer Graphics* at `http://chortle.ccsu.edu/vectorlessons/vectorindex.html`.

```
public interface ARFilter extends Filter {
  public float[] getGLPose();
}
```

When the pose matrix is unknown, `getGLPose()` should return `null`.

The most basic implementation of the `ARFilter` interface is the `NoneARFilter` class. `NoneARFilter` does not actually find the pose matrix. Instead, the `getGLPose()` method always returns `null`, as we can see in the following code:

```
public class NoneARFilter extends NoneFilter implements ARFilter {
  @Override
  public float[] getGLPose() {
    return null;
  }
}
```

The `NoneARFilter` class, similar to its parent class `NoneFilter`, is just a convenient stand-in for other filters. We use `NoneARFilter` when we want to turn off filtering but still have an object that conforms to the `ARFilter` interface.

Building projection matrices in CameraProjectionAdapter

Here is an exercise for sightseers. Choose a famous photo that was taken at a recognizable location, somewhere that should still look similar today. Travel to that site and explore it until you know how the photographer set up the shot. Where was the camera positioned and how was it rotated?

If you found an answer, and if you are sure of it, you must have already known which lens (and, for a zoom lens, which zoom setting) the photographer used. Without this information, you could not have narrowed down the feasible camera poses to the one true pose.

We face a similar problem when trying to determine the pose of a photographed object relative to a monocular (single-lens) camera. To find a unique solution, we first need to know the camera's horizontal and vertical field of view, and horizontal and vertical resolution in pixels.

Fortunately, we can get this data via the `android.hardware.Camera.Parameters` class. Our `CameraProjectionAdapter` class will allow client code to provide a `Camera.Parameters` object and then get a projection matrix in either the OpenCV or OpenGL format.

Unfortunately, on some devices, the data provided by `Camera.Parameters` are misleading or just plain wrong.

On a device with a zoom lens, the horizontal and vertical fields of view may be based on the lens's widest (1x) zoom setting. For advice on finding fields of view based on the current zoom setting, see the StackOverflow post at `http://stackoverflow.com/a/12118760`.

On some devices, the field of view is reported as 360 degrees or another incorrect value. For example, the Sony Xperia Arc may report a 360 degree field of view.

As an alternative to relying on `Camera.Parameters`, we could ask the user to calibrate the camera at runtime. OpenCV provides calibration functions that require the user to take a series of pictures of a chessboard. We do not cover these functions in this book, but you can read about them in the official documentation at `http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html` or in other OpenCV books such as *OpenCV 2 Computer Vision Application Programming Cookbook* (Packt Publishing) by Robert Laganière.

As a member variable, `CameraProjectionAdapter` stores all the data that it needs to construct the projection matrices. It also stores the matrices themselves and the `boolean` flags to indicate whether the matrices are dirty (whether they need to be reconstructed the next time that client code fetches them). Let's write the following declaration of the class and member variables:

```
// Use the deprecated Camera class.
@SuppressWarnings("deprecation")
public class CameraProjectionAdapter {
  float mFOVY = 45f; // equivalent in 35mm photography: 28mm lens
  float mFOVX = 60f; // equivalent in 35mm photography: 28mm lens
  int mHeightPx = 480;
  int mWidthPx = 640;
```

```
float mNear = 0.1f;
float mFar = 10f;
final float[] mProjectionGL = new float[16];
boolean mProjectionDirtyGL = true;

MatOfDouble mProjectionCV;
boolean mProjectionDirtyCV = true;
```

Note that we assume some default values in case the client code fails to provide a `Camera.Parameters` instance. Also note that the `mNear` and `mFar` variables store the **near and far clipping distances**, which means that the OpenGL camera will not render anything nearer or farther than these respective distances. We can set some of the variables based on the video camera's specifications and the current image size, as seen in the following method:

```
public void setCameraParameters(
  final Parameters cameraParameters, final Size imageSize) {
  mFOVY = cameraParameters.getVerticalViewAngle();
  mFOVX = cameraParameters.getHorizontalViewAngle();

  mHeightPx = imageSizeimageSize.height;
  mWidthPx = imageSizeimageSize.width;

  mProjectionDirtyGL = true;
  mProjectionDirtyCV = true;
}
```

As a convenient way of getting the image's aspect ratio, let's provide the following method:

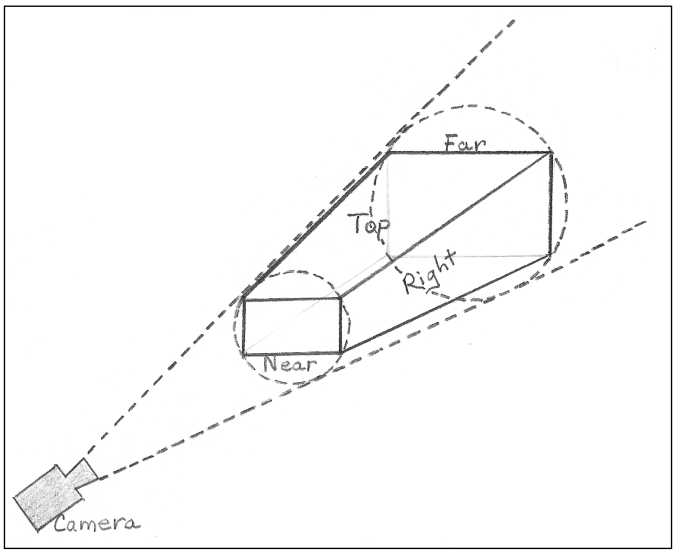
```
public float getAspectRatio() {
  return (float)mWidthPx / (float)mHeightPx;
}
```

For the near and far clipping distances, we just need a simple setter, which we can implement as follows:

```
public void setClipDistances(float near, float far) {
  mNear = near;
  mFar = far;
  mProjectionDirtyGL = true;
}
```

Since the clipping distances are only relevant to OpenGL, we only set the dirty flag for the OpenGL matrix.

Next, let's consider the getter for the OpenGL projection matrix. If the matrix is dirty, we reconstruct it. To construct a projection matrix, OpenGL provides a function called `frustumM(float[] m, int offset, float left, float right, float bottom, float top, float near, float far)`. The first two arguments are an array and offset where the matrix data should be stored. The rest of the arguments describe the edges of the **view frustum**, which is the region of space that the camera can see. Although you might be tempted to think that this region is conical, it is actually a truncated pyramid. The cone's extremities are lost due to near clipping, far clipping, and the rectangular shape of the user's screen. Here is a visualization of the view frustum inside the view cone:



Based on the clipping distances and fields of view, we can find the view frustum's other measurements by using simple trigonometry, as seen in the following implementation:

```
public float[] getProjectionGL() {
  if (mProjectionDirtyGL) {
    final float right =
      (float)Math.tan(0.5f * mFOVX * Math.PI / 180f) * mNear;
    // Calculate vertical bounds based on horizontal bounds
    // and the image's aspect ratio. Some aspect ratios will
    // be crop modes that do not use the full vertical FOV
    // reported by Camera.Paremeters.
    final float top = right / getAspectRatio();
    Matrix.frustumM(mProjectionGL, 0,
```

```
            -right, right, -top, top, mNear, mFar);
        mProjectionDirtyGL = false;
      }
      return mProjectionGL;
    }
```

The getter for the OpenCV projection matrix is slightly more complicated because the library does not offer a similar helper function to construct the matrix. Thus, we must understand the contents of the OpenCV projection matrix and construct it ourselves. It has the following 3 x 3 format:

focalLengthXInPixels	0	centerXInPixels
0	focalLengthYInPixels	centerYInPixels
0	0	1

We will assume a symmetrical lens system and a sensor with square pixels. Subject to these constraints, the matrix simplifies to the following format:

focalLengthInPixels	0	(0.5 * widthInPixels)
0	focalLengthInPixels	(0.5 * heightInPixels)
0	0	1

The focal length is the distance between the camera's sensor and the lens system's optical center when the lens is focused at an infinite distance. For OpenCV's purposes, the focal length is expressed in **pixel-related units**. Notionally, we could attribute a physical size to a pixel. We would do this by dividing the camera sensor's width or height by its horizontal or vertical resolution. However, since we do not know any of the physical measurements of the sensor, we instead use trigonometry to determine the pixel-related focal length. The implementation is as follows:

```
    public MatOfDouble getProjectionCV() {
      if (mProjectionDirtyCV) {
        if (mProjectionCV == null) {
          mProjectionCV = new MatOfDouble();
          mProjectionCV.create(3, 3, CvType.CV_64FC1);
        }

        // Calculate focal length using the aspect ratio of the
        // FOV values reported by Camera.Parameters. This is not
        // necessarily the same as the image's current aspect
        // ratio, which might be a crop mode.
        final float fovAspectRatio = mFOVX / mFOVY;
```

```
        double diagonalPx = Math.sqrt(
          (Math.pow(mWidthPx, 2.0) +
            Math.pow(mWidthPx / fovAspectRaio, 2.0)));
        double diagonalFOV = Math.sqrt(
          (Math.pow(mFOVX, 2.0) +
            Math.pow(mFOVY, 2.0)));
        double focalLengthPx = diagonalPx /
          (2.0 * Math.tan(0.5 * diagonalFOV * Math.PI / 180f));

        mProjectionCV.put(0, 0, focalLengthPx);
        mProjectionCV.put(0, 1, 0.0);
        mProjectionCV.put(0, 2, 0.5 * mWidthPx);
        mProjectionCV.put(1, 0, 0.0);
        mProjectionCV.put(1, 1, focalLengthPx);
        mProjectionCV.put(1, 2, 0.5 * mHeightPx);
        mProjectionCV.put(2, 0, 0.0);
        mProjectionCV.put(2, 1, 0.0);
        mProjectionCV.put(2, 2, 1.0);
      }
      return mProjectionCV;
    }
  }
```

Client code can use `CameraProjectionAdapter` by instantiating it, calling `setCameraParameters` whenever the active camera or image size changes, and calling `getProjectionGL` and `getProjectionCV` whenever a projection matrix is needed for OpenGL or OpenCV computations.

Modifying ImageDetectionFilter for 3D tracking

For 3D tracking, `ImageDetectionFilter` needs all the same member variables as before, except the `mSceneCorners` variable. We also need several new variables to store computations about the target's pose. Moreover, the class needs to implement the `ARFilter` interface. Let's modify `ImageDetectionFilter` as follows:

```
public class ImageDetectionFilter implements ARFilter {

  // ...

  // The reference image's corner coordinates, in 3D, in real
  // units.
```

```
private final MatOfPoint3f mReferenceCorners3D =
  new MatOfPoint3f();
// Good corner coordinates detected in the scene, in
// pixels.
private final MatOfPoint2f mSceneCorners2D =
  new MatOfPoint2f();

// Distortion coefficients of the camera's lens.
// Assume no distortion.
private final MatOfDouble mDistCoeffs =
new MatOfDouble(0.0, 0.0, 0.0, 0.0);

// An adaptor that provides the camera's projection matrix.
private final CameraProjectionAdapter
  mCameraProjectionAdapter;
// The Euler angles of the detected target.
private final MatOfDouble mRVec = new MatOfDouble();
// The XYZ coordinates of the detected target.
private final MatOfDouble mTVec = new MatOfDouble();
// The rotation matrix of the detected target.
private final MatOfDouble mRotation =
  new MatOfDouble();
// The OpenGL pose matrix of the detected target.
private final float[] mGLPose = new float[16];

// Whether the target is currently detected.
private boolean mTargetFound = false;
```

The constructor requires two new arguments. One of these is an instance of `CameraProjectionAdapter`, which we store in a member variable. The other is a number that represents the size of the printed image's smaller dimension (height for a landscape image or width for a portrait image), which we use to calculate the 3D bounds of the object that we are tracking. We may choose any arbitrary unit of measurement, but elsewhere we must use the same unit when specifying the near clip distance, far clip distance, and the cube's scale. For example, if we specify that the printed image's size is 1.0 and the cube's scale is 0.5, the cube will be half as big as the printed image's smaller dimension. The new arguments and their usage are seen in the following code:

```
public ImageDetectionFilter(final Context context,
  final int referenceImageResourceID,
  final CameraProjectionAdapter
    cameraProjectionAdapter,
```

```
    final double realSize)
        throws IOException {

    // ...

    // Compute the image's width and height in real units, based
    // on the specified real size of the image's smaller
    // dimension.
    final double aspectRatio = (double)referenceImageGray.cols()

      /(double)referenceImageGray.rows();
    final double halfRealWidth;
    final double halfRealHeight;
    if (referenceImageGray.cols() > referenceImageGray.rows()) {
      halfRealHeight = 0.5f * realSize;
      halfRealWidth = halfRealHeight * aspectRatio;
    } else {
      halfRealWidth = 0.5f * realSize;
      halfRealHeight = halfRealWidth / aspectRatio;
    }

    // Define the printed image so that it normally lies in the
    // xy plane (like a painting or poster on a wall).
    // That is, +z normally points out of the page toward the
    // viewer.
    mReferenceCorners3D.fromArray(
      new Point3(-halfRealWidth, -halfRealHeight, 0.0),
      new Point3( halfRealWidth, -halfRealHeight, 0.0),
      new Point3( halfRealWidth,  halfRealHeight, 0.0),
      new Point3(-halfRealWidth,  halfRealHeight, 0.0));

    mCameraProjectionAdapter = cameraProjectionAdapter;
  }
```

To satisfy the `ARFilter` interface, we need to implement a getter for the OpenGL pose matrix. When the target is lost, this getter should return `null` because we have no valid data about the pose. We can implement the getter as follows:

```
@Override
public float[] getGLPose() {
  return (mTargetFound ? mGLPose : null);
}
```

Let's rename our `findSceneCorners` method to `findPose`. To reflect this name change, the implementation of the `apply` method changes as follows:

```
@Override
public void apply(final Mat src, final Mat dst) {

  // Convert the scene to grayscale.
  Imgproc.cvtColor(src, mGraySrc, Imgproc.COLOR_RGBA2GRAY);

  // Detect the scene features, compute their descriptors,
  // and match the scene descriptors to reference descriptors.
  mFeatureDetector.detect(mGraySrc, mSceneKeypoints);
  mDescriptorExtractor.compute(mGraySrc, mSceneKeypoints,
    mSceneDescriptors);
  mDescriptorMatcher.match(mSceneDescriptors,
    mReferenceDescriptors, mMatches);

  // Attempt to find the target image's 3D pose in the
  // scene.
  findPose();

  // If the pose has not been found, draw a thumbnail of the
  // target image.
  draw(src, dst);
}
```

The implementation of `findPose` covers some additional steps beyond the old `findSceneCorners` method. After finding corners, we get an OpenCV projection matrix from our instance of `CameraProjectionAdapter`. Next, we solve the target's position and rotation, based on the matching corners and the projection. Most of the calculations are done by an OpenCV function called `Calib3d.solvePnP(MatOfPoint3f objectPoints, MatOfPoint2f imagePoints, Mat cameraMatrix, MatOfDouble distCoeffs, Mat rvec, Mat tvec)`. This function puts the position and rotation results in two separate vectors. The y and z axes in OpenCV are inverted compared to OpenGL. The direction of the angles is also inverted. Thus, we need to multiply some components of the vectors by -1. We convert the rotation vector into a matrix using another OpenCV function called `Calib3d.Rodrigues(Mat src, Mat dst)`. Finally, we manually convert the resulting rotation matrix and position the vector into a `float[16]` array that is appropriate for OpenGL. The code is as follows:

```
private void findPose() {

  // ...
```

```
if (minDist > 50.0) {
  // The target is completely lost.
  mTargetFound = false;
  return;
} else if (minDist > 25.0) {
  // The target is lost but maybe it is still close.
  // Keep using any previously found pose.
  return;
}

// ...

// Convert the scene corners to integer format, as required
   // by the Imgproc.isContourConvex function.
mCandidateSceneCorners.convertTo(mIntSceneCorners,
  CvType.CV_32S);

// Check whether the corners form a convex polygon. If not,
// (that is, if the corners form a concave polygon), the
// detection result is invalid because no real perspective
// can make the corners of a rectangular image look like a
// concave polygon!
if (!Imgproc.isContourConvex(mIntSceneCorners)) {
  return;
}

double[] sceneCorner0 =
  mCandidateSceneCorners.get(0, 0);
double[] sceneCorner1 =
  mCandidateSceneCorners.get(1, 0);
double[] sceneCorner2 =
  mCandidateSceneCorners.get(2, 0);
double[] sceneCorner3 =
  mCandidateSceneCorners.get(3, 0);
mSceneCorners2D.fromArray(
  new Point(sceneCorner0[0], sceneCorner0[1]),
  new Point(sceneCorner1[0], sceneCorner1[1]),
  new Point(sceneCorner2[0], sceneCorner2[1]),
  new Point(sceneCorner3[0], sceneCorner3[1]));

MatOfDouble projection =
  mCameraProjectionAdapter.getProjectionCV();
```

```
    // Find the target's Euler angles and XYZ coordinates.
    Calib3d.solvePnP(mReferenceCorners3D mSceneCorners2D,
      projection, mDistCoeffs, mRVec, mTVec);

    // Positive y is up in OpenGL, down in OpenCV.
    // Positive z is backward in OpenGL, forward in OpenCV.
    // Positive angles are counter-clockwise in OpenGL,
    // clockwise in OpenCV.
    // Thus, x angles are negated but y and z angles are
    // double-negated (that is, unchanged).
    // Meanwhile, y and z positions are negated.

    double[] rVecArray = mRVec.toArray();
    rVecArray[0] *= -1.0; // negate x angle
    mRVec.fromArray(rVecArray);

    // Convert the Euler angles to a 3x3 rotation matrix.
    Calib3d.Rodrigues(mRVec, mRotation);

    double[] tVecArray = mTVec.toArray();

    // OpenCV's matrix format is transposed, relative to
    // OpenGL's matrix format.
    mGLPose[0]  =  (float)mRotation.get(0, 0)[0];
    mGLPose[1]  =  (float)mRotation.get(0, 1)[0];
    mGLPose[2]  =  (float)mRotation.get(0, 2)[0];
    mGLPose[3]  =  0f;
    mGLPose[4]  =  (float)mRotation.get(1, 0)[0];
    mGLPose[5]  =  (float)mRotation.get(1, 1)[0];
    mGLPose[6]  =  (float)mRotation.get(1, 2)[0];
    mGLPose[7]  =  0f;
    mGLPose[8]  =  (float)mRotation.get(2, 0)[0];
    mGLPose[9]  =  (float)mRotation.get(2, 1)[0];
    mGLPose[10] =  (float)mRotation.get(2, 2)[0];
    mGLPose[11] =  0f;
    mGLPose[12] =  (float)tVecArray[0];
    mGLPose[13] = -(float)tVecArray[1]; // negate y position
    mGLPose[14] = -(float)tVecArray[2]; // negate z position
    mGLPose[15] =  1f;

    mTargetFound = true;
  }
```

Finally, let's modify our `draw` method by removing the code that draws a green border around the tracked image. (Instead, the `ARCubeRenderer` class will be responsible for drawing a cube in front of the tracked image.) After removing the unwanted code, we are left with the following implementation of the `draw` method:

```
    protected void draw(Mat src, Mat dst) {

      if (dst != src) {
        src.copyTo(dst);
      }

      if (!mTargetFound) {
        // The target has not been found.

        // Draw a thumbnail of the target in the upper-left
        // corner so that the user knows what it is.

        // Compute the thumbnail's larger dimension as half the
        // video frame's smaller dimension.
        int height = mReferenceImage.height();
        int width = mReferenceImage.width();
        int maxDimension = Math.min(dst.width(),
          dst.height()) / 2;
        double aspectRatio = width / (double)height;
        if (height > width) {
          height = maxDimension;
          width = (int)(height * aspectRatio);
        } else {
          width = maxDimension;
          height = (int)(width / aspectRatio);
        }

        // Select the region of interest (ROI) where the thumbnail
        // will be drawn.
        Mat dstROI = dst.submat(0, height, 0, width);

        // Copy a resized reference image into the ROI.
        Imgproc.resize(mReferenceImage, dstROI, dstROI.size(),
          0.0, 0.0, Imgproc.INTER_AREA);
      }
    }
  }
```

Next, we look at how to render the cube with OpenGL.

Rendering the cube in ARCubeRenderer

Android provides a class called `GLSurfaceView`, which is a widget that is drawn by OpenGL. The drawing logic is encapsulated via an interface called `GLSurfaceView.Renderer`, which we will implement in `ARCubeRenderer`. The interface requires the following methods:

- `onDrawFrame(GL10 gl)`: This is called to draw the current frame. Here, we will also configure the OpenGL perspective and **viewport** (its drawing area on the screen) because the interfaces of `ARCubeRenderer` and `CameraProjectionAdapter` potentially allow the perspective to change on a frame-to-frame basis.
- `onSurfaceChanged(GL10 gl, int width, int height)`: This is called when the surface size changes. For our purposes, this method just needs to store the width and height in member variables.
- `onSurfaceCreated(GL10 gl, EGLConfig config)`: This is called when the surface is created or recreated. Typically, this method configures any OpenGL settings that we will not subsequently change. In other words, these settings are independent of the perspective and of the content being drawn.

The `GL10` instance, which is passed as an argument, provides access to the standard OpenGL ES 1.0 functionality. Basically, we are interested in two kinds of OpenGL functionality: applying matrix transformations to 3D vertices and then drawing triangles based on the transformed vertices. Our cube will have eight vertices and 12 triangles (6 square faces * 2 triangles per square face). We will specify a color for each vertex and describe the triangles as a sequence of vertex indices (three vertex indices per triangle).

Vertices, vertex colors, and triangles are all stored in `ByteBuffer` instances. Since we only support one style of cube, we will use static instances of `ByteBuffer` so that multiple `ARCubeRenderer` instances may share them. As member variables, we also want `ARFilter` to provide the cube's pose matrix, a `CameraProjectionAdapter` to provide the projection matrix, and a scale to allow client code to resize the cube. The declarations of `ARCubeRenderer` and its variables are as follows:

```
public class ARCubeRenderer implements GLSurfaceView.Renderer {

  public ARFilter filter;
  public CameraProjectionAdapter cameraProjectionAdapter;
  public float scale = 1f;

  private int mSurfaceWidth;
  private int mSurfaceHeight;
```

```
private static final ByteBuffer VERTICES;
private static final ByteBuffer COLORS;
private static final ByteBuffer TRIANGLES;
```

Since the vertices, colors, and triangles are `static` variables, we initialize them in a `static` block. For each buffer, we must specify the required number of bytes. The vertices take up `96` bytes (8 vertices * 3 floats per vertex * 4 bytes per float). We specify the vertices for a cube that is 2 units wide. After populating the buffer, we rewind its pointer to the first index. The code is as follows:

```
static {
  VERTICES = ByteBuffer.allocateDirect(96);
  VERTICES.order(ByteOrder.nativeOrder());
  VERTICES.asFloatBuffer().put(new float[] {
    // Front.
    -0.5f, -0.5f,  0.5f,
     0.5f, -0.5f,  0.5f,
     0.5f,  0.5f,  0.5f,
    -0.5f,  0.5f,  0.5f,
    // Back.
    -0.5f, -0.5f, -0.5f,
     0.5f, -0.5f, -0.5f,
     0.5f,  0.5f, -0.5f,
    -0.5f,  0.5f, -0.5f
  });
  VERTICES.position(0);
```

The vertex colors take up 32 bytes (8 vertices * 4 bytes of RGBA color per vertex). We specify a different color for each vertex, as seen in the following code:

```
  COLORS = ByteBuffer.allocateDirect(32);
  final byte maxColor = (byte)255;
  COLORS.put(new byte[] {
    // Front.
    maxColor, 0, 0, maxColor,        // red
    maxColor, maxColor, 0, maxColor, // yellow
    maxColor, maxColor, 0, maxColor, // yellow
    maxColor, 0, 0, maxColor,        // red
    // Back.
    0, maxColor, 0, maxColor,        // green
    0, 0, maxColor, maxColor,        // blue
    0, 0, maxColor, maxColor,        // blue
    0, maxColor, 0, maxColor         // green
  });
  COLORS.position(0);
```

The triangles take up 36 bytes (12 triangles * 3 vertex indices per triangle). OpenGL requires us to specify each triangle's vertex indices in a counter-clockwise order, when the triangle is viewed from its rear face. This ordering is called **counter-clocking winding**, and it is seen in the following code:

```
    TRIANGLES = ByteBuffer.allocateDirect(36);
    TRIANGLES.put(new byte[] {
      // Front.
      0,  1,  2,  2,  3,  0,
      3,  2,  6,  6,  7,  3,
      7,  6,  5,  5,  4,  7,
      // Back.
      4,  5,  1,  1,  0,  4,
      4,  0,  3,  3,  7,  4,
      1,  5,  6,  6,  2,  1
    });
    TRIANGLES.position(0);

  }
```

When our instance of `GLSurfaceView` is created, we must configure it to use a fully transparent background color so that the underlying video will be visible. We must also specify that we want OpenGL to perform **backface culling**, which means that the triangles will only be drawn when they are facing the viewer. Since our cube is opaque, we do not want OpenGL to draw the inside of it! This initial configuration is implemented in the `onSurfaceCreated` method, as follows:

```
  @Override
  public void onSurfaceCreated(final GL10 gl,
    final EGLConfig config) {
    gl.glClearColor(0f, 0f, 0f, 0f); // transparent
    gl.glEnable(GL10.GL_CULL_FACE);
  }
```

When our instance of the `GLSurfaceView` is resized, we record the new size, as seen in the following code:

```
  @Override
  public void onSurfaceChanged(final GL10 gl, final int width,
    final int height) {
    mSurfaceWidth = width;
    mSurfaceHeight = height;
  }
}
```

When drawing to an instance of `GLSurfaceView`, we first clear any previous content (replacing it with the fully transparent background color that we specified earlier). Then, we check whether the projection and pose matrices are available. If they are available, we tell OpenGL to size the viewport, use the projection and pose matrices, and finally move and scale the cube so that we have an appropriately sized cube sitting in front of the tracked image. Then, we supply the vertices and vertex colors to OpenGL and tell it to draw the triangles. The implementation is as follows:

```
@Override
public void onDrawFrame(final GL10 gl) {

  gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
    GL10.GL_DEPTH_BUFFER_BIT);


  if (filter == null) {
    return;
  }

  if (cameraProjectionAdapter == null) {
    return;
  }

  float[] pose = filter.getGLPose();
  if (pose == null) {
    return;
  }

  final int adjustedWidth = (int)(mSurfaceHeight *
    cameraProjectionAdapter.getAspectRatio());
  final int marginX = (mSurfaceWidth - adjustedWidth) / 2;
  gl.glViewport(marginX, 0, adjustedWidth, mSurfaceHeight);

  gl.glMatrixMode(GL10.GL_PROJECTION);
  float[] projection =
    cameraProjectionAdapter.getProjectionGL();
  gl.glLoadMatrixf(projection, 0);

  gl.glMatrixMode(GL10.GL_MODELVIEW);
  gl.glLoadMatrixf(pose, 0);

  gl.glScalef(scale, scale, scale);
  // Move the cube forward so that it is not halfway inside
  // the image.
```

```
    gl.glTranslatef(0f, 0f, 0.5f);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    gl.glVertexPointer(3, GL11.GL_FLOAT, 0, VERTICES);
    gl.glColorPointer(4, GL11.GL_UNSIGNED_BYTE, 0, COLORS);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    gl.glDrawElements(GL10.GL_TRIANGLES, 36,
      GL10.GL_UNSIGNED_BYTE, TRIANGLES);

    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
  }
```

Now, we are ready to integrate 3D tracking and rendering into our application.

Adding 3D tracking and rendering to CameraActivity

We need to make a few changes to `CameraActivity` so that it conforms with our changes to `ImageDetectionFilter` and with the new interface provided by `ARFilter`. We also need to modify the activity's layout so that it includes `GLSurfaceView`. The adapter for this `GLSurfaceView` will be `ARCubeRenderer`. The `ImageDetectionFilter` and the `ARCubeRenderer` methods will use `CameraProjectionAdapter` to coordinate their projection matrices.

First, let's make the following changes to the member variables of `CameraActivity`:

```
// The filters.
private ARFilter[] mImageDetectionFilters;
private Filter[] mCurveFilters;
private Filter[] mMixerFilters;
private Filter[] mConvolutionFilters;

// ...

// The camera view.
private CameraBridgeViewBase mCameraView;
```

```
// An adapter between the video camera and projection
// matrix.
private CameraProjectionAdapter
  mCameraProjectionAdapter;

// The renderer for 3D augmentations.
private ARCubeRenderer mARRenderer;
```

As usual, once the OpenCV library is loaded, we need to create the filters. The only changes are that we need to pass an instance of `CameraProjectionAdapter` to each constructor of `ImageDetectionFilter` and that we need to use a `NoneARFilter` in place of a `NoneFilter`. The code is as follows:

```
public void onManagerConnected(final int status) {
  switch (status) {
  case LoaderCallbackInterface.SUCCESS:
    Log.d(TAG, "OpenCV loaded successfully");
    mCameraView.enableView();
    mBgr = new Mat();

    final ARFilter starryNight;
    try {
      // Define The Starry Night to be 1.0 units
      // tall
      starryNight = new ImageDetectionFilter(
        CameraActivity.this,
        R.drawable.starry_night,
        mCameraProjectionAdapter, 1.0);
    } catch (IOException e) {
      Log.e(TAG, "Failed to load drawable: " +
        "starry_night");
      e.printStackTrace();
      break;
    }

    final ARFilter akbarHunting;
    try {
      // Define Akbar Hunting with Cheetahs to be 1.0
      // units wide.
      akbarHunting = new ImageDetectionFilter(
        CameraActivity.this,
        R.drawable.akbar_hunting_with_cheetahs,
        mCameraProjectionAdapter, 1.0);
    } catch (IOException e) {
```

```
            Log.e(TAG, "Failed to load drawable: " +
              "akbar_hunting_with_cheetahs");
            e.printStackTrace();
            break;
          }

          mImageDetectionFilters = new ARFilter[] {
            new NoneARFilter(),
            starryNight,
            akbarHunting
          };

          // ...
        }
      }
```

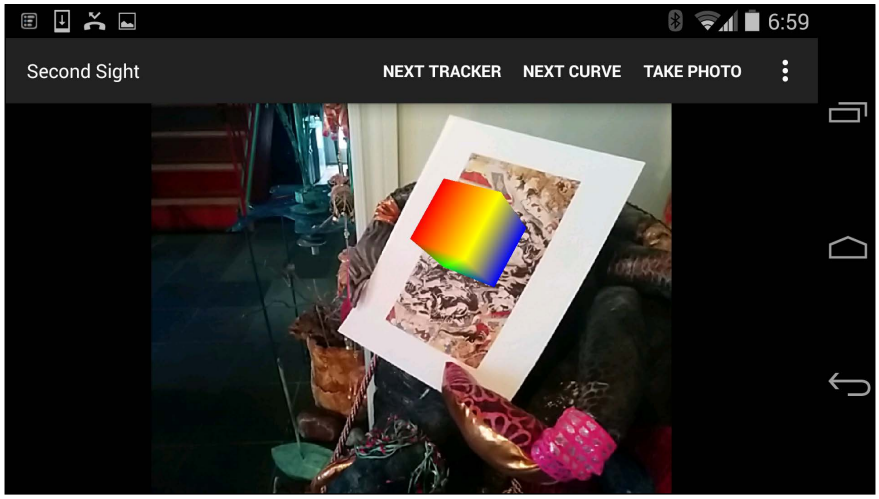
The remaining changes belong to the `onCreate` method, where we should create and configure the instances of `GLSurfaceView`, `ARCubeRenderer`, and `CameraProjectionAdapter`. The implementation includes some boilerplate code to overlay an instance of `GLSurfaceView` on top of an instance of `JavaCameraView`. These two views are contained inside a standard Android layout widget called `FrameLayout`. After setting up the layout, we need a `Camera` instance and a `Camera.Parameters` instance in order to do our remaining configuration. As in previous chapters, the `Camera` instance is obtained via a static method, `Camera.open`. The code is as follows:

```
protected void onCreate(final Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // ...

  final FrameLayout layout = new FrameLayout(this);
  layout.setLayoutParams(new FrameLayout.LayoutParams(
    FrameLayout.LayoutParams.MATCH_PARENT,
    FrameLayout.LayoutParams.MATCH_PARENT));
  setContentView(layout);

  mCameraView = new JavaCameraView(this,
    mCameraIndex);
  mCameraView.setCvCameraViewListener(this);
  mCameraView.setLayoutParams(
new FrameLayout.LayoutParams(
  FrameLayout.LayoutParams.MATCH_PARENT,
  FrameLayout.LayoutParams.MATCH_PARENT));
```

```
        layout.addView(mCameraView);

        GLSurfaceView glSurfaceView =
      new GLSurfaceView(this);
        glSurfaceView.getHolder().setFormat(
          PixelFormat.TRANSPARENT);
        glSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 0, 0);
        glSurfaceView.setZOrderOnTop(true);
        glSurfaceView.setLayoutParams(new
    FrameLayout.LayoutParams(
    FrameLayout.LayoutParams.MATCH_PARENT,
    FrameLayout.LayoutParams.MATCH_PARENT));
        layout.addView(glSurfaceView);

        mCameraProjectionAdapter =
      new CameraProjectionAdapter();

        mARRenderer = new ARCubeRenderer();
        mARRenderer.cameraProjectionAdapter =
          mCameraProjectionAdapter;
        // Earlier, we defined the printed image's size as
        // 1.0 unit.
        // Define the cube to be half this size.
        mARRenderer.scale = 0.5f;
        glSurfaceView.setRenderer(mARRenderer);

        final Camera camera;
        if (Build.VERSION.SDK_INT >=
          Build.VERSION_CODES.GINGERBREAD) {
          CameraInfo cameraInfo = new CameraInfo();
          Camera.getCameraInfo(mCameraIndex, cameraInfo);
          mIsCameraFrontFacing = (cameraInfo.facing ==
            CameraInfo.CAMERA_FACING_FRONT);
          mNumCameras = Camera.getNumberOfCameras();
          camera = Camera.open(mCameraIndex);
        } else { // pre-Gingerbread
          // Assume there is only 1 camera and it is rear-facing.
          mIsCameraFrontFacing = false;
          mNumCameras = 1;
          camera = Camera.open();
        }
        final Parameters parameters = camera.getParameters();
        camera.release();
```

```
        mSupportedImageSizes = parameters.getSupportedPreviewSizes();
        final Size size = mSupportedImageSizes.get(mImageSizeIndex);
        mCameraProjectionAdapter.setCameraParameters(
          parameters, size);
        // Earlier, we defined the printed image's size as
        // 1.0 unit.
        // Leave the near and far clip distances at their default
        // values, which are 0.1 (one-tenth the image size)
        // and 10.0 (ten times the image size).

        mCameraView.setMaxFrameSize(size.width, size.height);
        mCameraView.setCvCameraViewListener(this);
    }
```

That's all! Run and test Second Sight. When you activate one of the instances of `ImageDetectionFilter` and hold the appropriate printed image in front of the camera, you should see a colorful cube rendered in front of the image. For example, see the following screenshot:



If the video is too choppy (due to the slow processing of the image) or if the cube does not stick closely to the center of the tracked image, select a lower camera resolution and try again. Our reference images have a low resolution (approximately 600 pixels in the larger dimension), which should be optimal for a low camera resolution such as 800x600 or 640x480. Finally, remember that you might need to hold your Android device still for a second or two in order for the camera to autofocus on the target.

Learning more about 3D graphics on Android

Of course, in the world of 3D graphics, drawing a cube is similar to printing "Hello World"; it is just a basic demo. Although we have introduced meshes, transformations, and perspectives, there are many other topics that we have not touched at all, such as lighting, materials (realistic-looking surfaces), and importing an artist's work from 3D art packages. For a deeper understanding of 3D graphics on Android, have a look at these books:

- *Pro OpenGL ES for Android* (Apress) by Mike Smithwick and Mayank Verma. This book covers Android's Java API for OpenGL ES.
- *OpenGL ES 2.0 Programming Guide* (Addison-Wesley) by Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. This book covers the cross-platform C++ API for OpenGL ES.
- *Augmented Reality for Android Application Development* (Packt Publishing) by Jens Grubert and Dr. Raphael Grasset. This book shows how to use **jMonkeyEngine**, a cross-platform Java game engine, to superimpose 3D graphics on tracked, real-world images.

There are also many books on Android game development that may include a good introduction to 3D graphics.

Summary

We are now at the end of our introduction to OpenCV's Java interface and Android SDK. We covered several major uses of OpenCV, including capturing camera input, applying effects to images, tracking images in 2D and 3D, and integrating with OpenGL for augmented reality rendering.

Taking the knowledge you have gained so far, you can go on to develop other OpenCV applications in Java, whether targeted at Android or other platforms. You might also wish to explore OpenCV's C++ version, which is also cross-platform and can interface with Android NDK. To give you a flavor of this alternative, in our next and final chapter, we convert part of Second Sight to C++, and we call this C++ code via an interoperability layer called the Java Native Interface (JNI).

6
Mixing Java and C++ via JNI

Our goal in this chapter is to rewrite some of our Java classes so that they become thin wrappers around C++ classes. We will use an intermediary framework, **Java Native Interface** (**JNI**), which can expose Java and C++ code to each other. Along the way, we will gain a greater understanding of OpenCV's Java and C++ interfaces.

The complete Eclipse project for this chapter can be downloaded from the author's website. The project has the following two versions:

A version for OpenCV 3.x is located at `http://nummist.com/opencv/4598_06.zip`.

A version for OpenCV 2.x is located at `http://nummist.com/opencv/5206_06.zip`.

Understanding the role of JNI

JNI enables Java code to call C or C++ code (and vice versa). OpenCV4Android, Android SDK, and the Java standard libraries all rely on JNI. That is to say, these major Java libraries are partly built atop C++ or C libraries.

OpenCV is largely written in C++. Although the library provides a Java interface (OpenCV4Android) and a Python interface as well, most parts of these interfaces are thin layers atop the C++ implementation. For example, an `org.opencv.core.Mat` object (in the Java interface) or a NumPy array (in the Python interface) is backed by a `cv::Mat` object (in the C++ implementation), and they share a reference to the same data; there is no duplication of data.

When OpenCV's Java or Python interface forwards a function call to the C++ implementation, it does incur a small overhead cost. If our code makes thousands of OpenCV function calls per frame (for example, one or more calls per pixel per frame), we might begin to worry about this overhead. Typically, though, a frugal programmer does not make thousands of OpenCV function calls per frame, and the choice of OpenCV interface (Java, Python, or C++) has no appreciable effect on the frame rate! For comparison, consider Google's following remarks about the use of Android NDK (a C++ interface) versus Android SDK (a Java interface):

> *Before downloading the NDK, you should understand that the NDK will not benefit most apps. As a developer, you need to balance its benefits against its drawbacks. Notably, using native code on Android generally does not result in a noticeable performance improvement, but it always increases your app complexity. In general, you should only use the NDK if it is essential to your app – never because you simply prefer to program in C/C++.*

On the other hand, OpenCV's C++ interface does offer several features that are missing from the Java interface:

- **Manual memory management**: Whereas OpenCV's Java interface frees memory on the garbage collector's schedule, OpenCV's C++ interface frees memory on command. This manual control may be useful in situations where we face tight constraints on resources (either memory or the CPU cycles required to allocate and free memory).
- **Interoperability with other libraries**: OpenCV's C++ interface provides access to image data as raw bytes, which can be interpreted and used by many other C++ or C libraries directly, without copying or modification.
- **Cross-platform compatibility without Java**: OpenCV's C++ interface can be used on platforms where the Java runtime is unavailable or not installed. We can reuse a single C++ code base across Windows, Mac, Linux, Android, iOS, WinRT, and Windows Phone 8.

These features make the C++ interface an important topic in our tour of OpenCV. By leveraging JNI and writing our own C++ code, we will learn a different and versatile way of using OpenCV.

Measuring performance

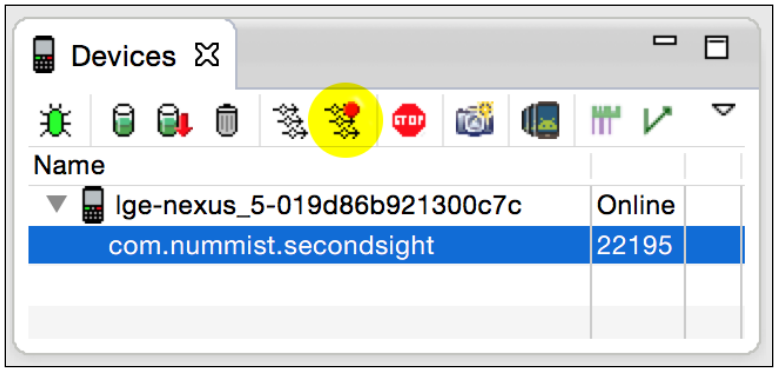
To convince yourself that OpenCV's Java and C++ interfaces offer similar speed, you might want to measure the app's performance before and after the modifications that we will make in this chapter. A good overall measure of performance is the number of **frames per second** (**FPS**) that are processed by the `onCameraFrame` callback. Optionally, OpenCV's `CameraView` class can compute and display this FPS metric. When we enable `CameraView` (in our `onManagerConnected` callback), we can also enable the FPS meter, as seen in the following code:

```
@Override
public void onManagerConnected(final int status) {
  switch (status) {
  case LoaderCallbackInterface.SUCCESS:
    Log.d(TAG, "OpenCV loaded successfully");
    mCameraView.enableView();
    mCameraView.enableFpsMeter();
    //...
```

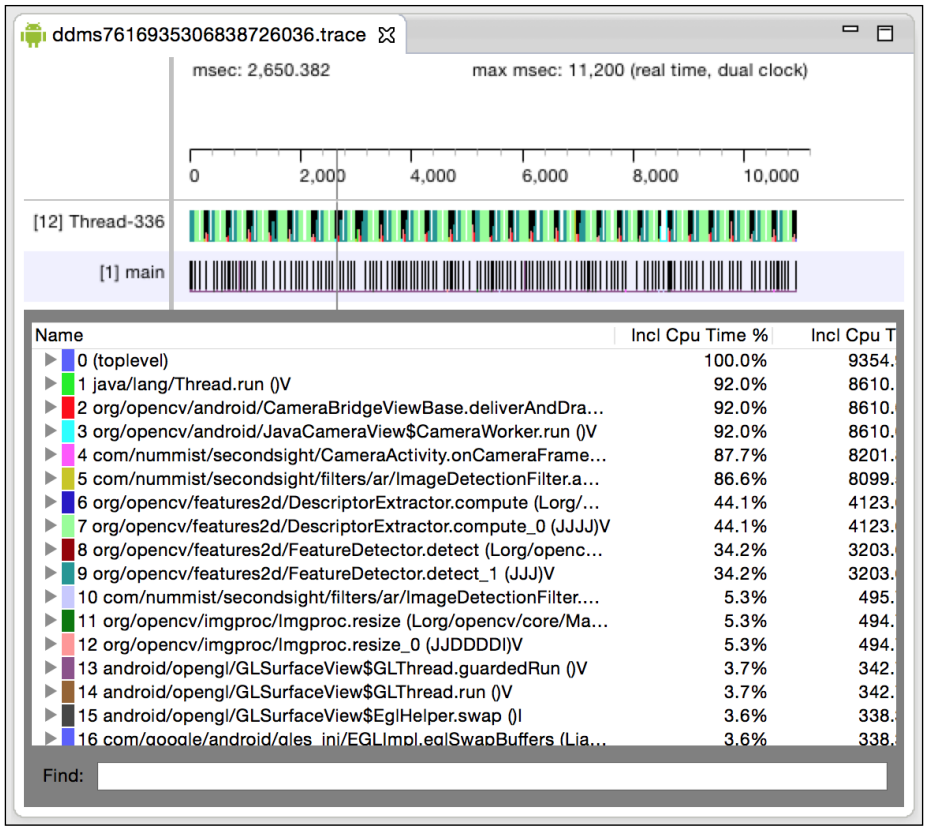
Although FPS is an important statistic, it does not tell us exactly how the app is using system resources. For example, we might want to know how much CPU time the app is spending on each method. To determine this, we can use the **Dalvik Debug Monitor Server** (**DDMS**), which is an Android debugging tool with integration into Eclipse. Here are the steps for profiling CPU usage via DDMS:

1. Open Eclipse. Open the DDMS perspective by clicking on the **DDMS** button (normally in the upper-right corner of the window) or navigating to the menu option **Window** | **Open Perspective** | **Other…** | **DDMS**.
2. Ensure that your Android device is connected via USB and that the Second Sight app is open on the device.
3. In the **Devices** pane of the DDMS perspective, you should see your Android device. Expand it. Select the **com.nummist.sightsight** process, which should be listed under your device.
4. On the Android device, configure Second Sight to use whichever filters and camera settings you wish to profile.

5. To start profiling the app's CPU usage, click on the **Start Method Profiling** button at the top of the **Devices** pane. The button looks like three arrows with a red dot. (The next screenshot marks the button's location with a highlighted, semitransparent circle.) Even after clicking on the button, you will not see a report on the CPU usage. The report is generated after step 7.



6. Continue using the app until you complete any actions (such as image recognition) that you wish to profile.
7. To stop profiling the app's CPU usage, click on the **Stop Method Profiling** button. It looks like three arrows with a black square, and it is located in the place where the **Start Method Profiling** button was. Now, you should see a report on each method's CPU usage. Each method name is listed along with its percentage share of the app's total CPU usage. See the following screenshot:



Some of the other buttons in the **Devices** pane also provide access to profiling reports, such as breakdowns of memory usage and OpenGL calls. Gather as many reports as you like! Once you profile the existing app, we will proceed with our modifications.

Adding files to the project

The C++ files for an Android project should always be located in a subdirectory named `jni`. This subdirectory should also contain our C++ code's dependencies and a type of compilation instruction named **Makefiles**. Specifically, we need to create the following new folders and files inside our Second Sight project folder:

- `jni/include/opencv2/`: This folder contains the **header files** (the definitions of types and functions) for OpenCV's C++ interface. Into this folder, copy the contents of `<opencv>/sdk/native/jni/include/opencv2/`. (Replace `<opencv>` with the path to OpenCV4Android on your system.)

- `jni/libs/`: This folder contains the library files (the compiled implementations) for OpenCV's C++ interface and OpenCV's dependencies. Into this folder, copy the contents of `<opencv>/sdk/native/libs/` and `<opencv>/sdk/native/3rdparty/libs/`.
- `jni/Application.mk`: This Makefile describes the type of environment in which our C++ library will be used. The environment's features include the hardware architecture, the Android version, and the application's usage of C++ language features and standard libraries.
- `jni/Android.mk`: This Makefile describes our C++ library's dependencies on other libraries such as OpenCV.
- `ImageDetectionFilter.hpp`: This C++ header file contains the definition of the `ImageDetectionFilter` class.
- `ImageDetectionFilter.cpp`: This C++ **source file** contains the implementation details of the `ImageDetectionFilter` class.
- `RecolorCMVFilter.hpp`: This is the header file for the `RecolorCMVFilter` class.
- `RecolorCMVFilter.cpp`: This is the source file for the `RecolorCMVFilter` class.
- `RecolorRCFilter.hpp`: This is the header file for the `RecolorRCFilter` class.
- `RecolorRCFilter.cpp`: This is the source file for the `RecolorRCFilter` class.
- `RecolorRGVFilter.hpp`: This is the header file for the `RecolorRGVFilter` class.
- `RecolorRGVFilter.cpp`: This is the source file for the `RecolorRGVFilter` class.
- `StrokeEdgesFilter.hpp`: This is the header file for the `StrokeEdgesFilter` class.
- `StrokeEdgesFilter.cpp`: This is the source file for the `StrokeEdgesFilter` class.
- `SecondSightJNI.cpp`: This file defines and implements JNI functions, which are C++ functions that are callable from Java. Our JNI functions wrap the public methods of our C++ classes.

As noted in the preceding list, we must copy the OpenCV header files and library files into our project. All the other files will contain our own code, which we will discuss in detail throughout the rest of this chapter. First, let's turn our attention to the Makefiles, which will enable us to build our C++ library.

Building the native library

C++ is a compiled language. This means that C++ code is not intended as an instruction set for machines (such as ARMv7-A chips) or virtual machines (such as the **Java Virtual Machine** or **JVM**). Rather, we use a tool named a **compiler** to translate C++ code into instructions that a given platform can understand. Each C++ source file is compiled to produce one **object file** (not to be confused with the term "object" in object-oriented programming). Then, we use a tool named a **linker** to combine multiple object files into either a library or an executable. The linker can also combine libraries to produce other libraries or executables. The linkage of libraries may be either static (meaning that a given version of the input library is "baked into" the output) or dynamic (meaning that any available version of the library may be loaded at runtime). Android NDK provides a tool named `ndk-build`, which abstracts the use of the compiler and linker, but we must still specify certain compiler flags and linker flags in the Makefiles. Our library will statically link to OpenCV (because dynamic linking is more complex in the case of OpenCV). Thus, our compiler and linker flags must satisfy the needs of OpenCV, as well as our own code's needs.

OpenCV relies on advanced features of the C++ language. Particularly, **Run-Time Type Information** (**RTTI**) and runtime exceptions are language features that enable OpenCV to robustly handle the various (valid or invalid) data types that users might try to process. OpenCV also relies on the C++ **Standard Template Library** (**STL**), which provides patterns to store and process data. Some STL implementations do not support RTTI. A safe choice is the GNU STL implementation, which does support RTTI. Our `Application.mk` file must tell the compiler to use RTTI, exceptions, and GNU STL. Furthermore, this Makefile must specify the hardware architectures and Android versions that we are targeting. To stipulate these requirements, open `Application.mk` and write the following code:

```
APP_STL := gnustl_static # GNU STL
APP_CPPFLAGS := -frtti -fexceptions # RTTI, exceptions
APP_ABI := arm64-v8a armeabi armeabi-v7a mips mips64 x86 x86_64
APP_PLATFORM := android-8
```

Save the changes to `Application.mk`. These changes affect the way that all of the app's C++ modules are built. Next, in `Android.mk`, we need to provide instructions to build the specific C++ module that we will write in this chapter. This Makefile begins by specifying our module's name, its dependencies on other libraries, and its source files, as seen in the following code:

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

include $(CLEAR_VARS)
LOCAL_MODULE := SecondSight
```

```
LOCAL_C_INCLUDES += $(LOCAL_PATH)/include
LOCAL_LDLIBS := -lstdc++ # C++ standard library
LOCAL_LDLIBS += -llog # Android logging
LOCAL_LDLIBS += -lz # zlib
LOCAL_STATIC_LIBRARIES += opencv_calib3d
LOCAL_STATIC_LIBRARIES += opencv_features2d
LOCAL_STATIC_LIBRARIES += opencv_flann
LOCAL_STATIC_LIBRARIES += opencv_imgproc
LOCAL_STATIC_LIBRARIES += opencv_core
LOCAL_STATIC_LIBRARIES += opencv_hal
ifneq ($(TARGET_ARCH_ABI), arm64-v8a)
  ifneq ($(TARGET_ARCH_ABI), armeabi)
    LOCAL_STATIC_LIBRARIES += tbb
  endif
endif
LOCAL_SRC_FILES := SecondSightJNI.cpp
LOCAL_SRC_FILES += RecolorCMVFilter.cpp
LOCAL_SRC_FILES += RecolorRCFilter.cpp
LOCAL_SRC_FILES += RecolorRGVFilter.cpp
LOCAL_SRC_FILES += StrokeEdgesFilter.cpp
LOCAL_SRC_FILES += ImageDetectionFilter.cpp
include $(BUILD_SHARED_LIBRARY)
```

Adapting the code to OpenCV 2.x

Remove the line `LOCAL_STATIC_LIBRARIES += opencv_hal`, as OpenCV 2.x has no `hal` (hardware abstraction layer) module.

Let's give further thought to the preceding block of code. The `LOCAL_MODULE` variable is our module's name. The `LOCAL_C_INCLUDES` variable is a path that the compiler will search for header files. The `LOCAL_LDLIBS` variable is a list of dynamic libraries that will be available on Android to load at runtime, while the `LOCAL_STATIC_LIBRARIES` variable is a list of static libraries that will be built into our module. The latter list comprises six OpenCV modules (`calib3d`, `features2d`, `flann`, `imgproc`, `core`, and `hal`), plus one of OpenCV's optional dependencies, **Intel Thread Building Blocks** (**TBB**). Last, `LOCAL_SRC_FILES` is a list of our source files.

TBB is a multiprocessing library that enables OpenCV to optimize many operations. However, TBB is not relevant to the armeabi architecture (that is, ARMv5 and ARMv6) because this architecture uses only single-core processors. TBB is also not yet available for arm64-v8a (that is, 64-bit ARMv8-A), a relatively new architecture.

To complete `Android.mk`, we must specify the path to each of the libraries that we have named in `LOCAL_STATIC_LIBRARIES`. Here is the relevant code:

```
include $(CLEAR_VARS)
LOCAL_MODULE := opencv_calib3d
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_calib3d.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := opencv_features2d
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_features2d.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := opencv_flann
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_flann.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := opencv_imgproc
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_imgproc.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := opencv_core
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_core.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := opencv_hal
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libopencv_hal.a
include $(PREBUILT_STATIC_LIBRARY)


ifneq ($(TARGET_ARCH_ABI), arm64-v8a)
  ifneq ($(TARGET_ARCH_ABI), armeabi)
    include $(CLEAR_VARS)
    LOCAL_MODULE := tbb
    LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/libtbb.a
    include $(PREBUILT_STATIC_LIBRARY)
  endif
endif
```

Adapting the code to OpenCV 2.x

Remove the following lines, as OpenCV 2.x has no `hal` (hardware abstraction layer) module:

```
include $(CLEAR_VARS)
LOCAL_MODULE := opencv_hal
LOCAL_SRC_FILES := libs/$(TARGET_ARCH_ABI)/
libopencv_hal.a
include $(PREBUILT_STATIC_LIBRARY)
```

Note that we are not using all the library files that come with OpenCV. Optionally, we could delete the unused library files (and the related headers) from our project. However, when we build Second Sight, the app's size will ultimately be unaffected by the presence of the unused libraries and headers inside the project's `jni` folder.

Save the changes to `Android.mk`. We have finished writing the Makefiles. Now, let's do a simple test of the build process. Open Terminal (in Mac or Linux) or Command Prompt (in Windows) and run the following commands:

```
$ cd <secondsight_project_path>
$ ndk-build
```

Note that `<secondsight_project_path>` refers to the project's root folder, not the `jni` subfolder. If the `ndk-build` command is not found, refer to *Chapter 1*, *Setting Up OpenCV*, and ensure that the Android NDK directory is correctly added to your system's `PATH` variable (in Unix) or `Path` variable (in Windows).

The `ndk-build` command should print the results of the attempted build (including any warnings and errors). If the build succeeds, the following library files should be created inside the Second Sight project folder:

- `libs/arm64-v8a/libSecondSight.so`
- `libs/armeabi/libSecondSight.so`
- `libs/armeabi-v7a/libSecondSight.so`
- `libs/mips/libSecondSight.so`
- `libs/mips64/libSecondSight.so`
- `libs/x86/libSecondSight.so`
- `libs/x86_64/libSecondSight.so`

The `.so` extension stands for **shared object**, which is a standard name (in Unix systems, such as Android) for a compiled library that can be loaded at runtime by one or more applications. This is analogous to a dynamically linked library or `.dll` (in Windows systems).

Note that these four compiled libraries correspond to the four target architectures that we specified in `Application.mk`. So far, we have not compiled any useful content into these library files because we have not yet written any code in our header and source files! Let's proceed to modify the `Filter` interface and then write our first C++ class, its JNI bindings, and the Java code that uses it.

Modifying the filter interface

As we discussed in *Understanding the role of JNI*, C++ is designed for manual memory management. It frees memory on command, not on a garbage collector's schedule. If we want to provide a degree of manual memory management to Java users as well, we can write a public method that is responsible for freeing any C++ resources (or other unmanaged resources) associated with a Java class. Conventionally, such a method is named `dispose`. Since several of our `Filter` implementations will own C++ resources, let's add a `dispose` method to `Filter.java`:

```
public interface Filter {
  public abstract void dispose();
  public abstract void apply(final Mat src, final Mat dst);
}
```

Let's modify `NoneFilter.java` to provide an empty implementation of `dispose`, as seen in the following code:

```
public class NoneFilter implements Filter {

  @Override
  public void dispose() {
    // Do nothing.
  }

  @Override
  public void apply(final Mat src, final Mat dst) {
    // Do nothing.
  }
}
```

Similarly, as a placeholder, we can add an empty `dispose` method to all our other concrete implementations of `Filter`. Later, in classes that use JNI, we will add some more code to the implementation of `dispose`.

OpenCV4Android does not provide `dispose` methods, but it does override the `finalize` method to free C++ resources when the associated Java objects are garbage-collected. (The `finalize` method is inherited from `java.lang.Object` and is called when the object is garbage-collected.) When we write nonempty implementations of `dispose`, we will also override `finalize` to call `dispose`. Thus, our classes will provide automatic memory management as a fallback if there is no manual call to `dispose`.

An activity's `onDestroy` method is a good place to free resources. Let's modify the `onDestroy` method in `CameraActivity.java` to ensure that each `Filter` object's `dispose` method is called:

```
@Override
public void onDestroy() {
  if (mCameraView != null) {
    mCameraView.disableView();
  }
  // Dispose of native resources.
  disposeFilters(mImageDetectionFilters);
  disposeFilters(mCurveFilters);
  disposeFilters(mMixerFilters);
  disposeFilters(mConvolutionFilters);
  super.onDestroy();
}

private void disposeFilters(Filter[] filters) {
  if (filters != null) {
    for (Filter filter : filters) {
      filter.dispose();
    }
  }
}
```

Note that we are using a helper method, `disposeFilters`, to iterate over a `Filter[]` array and call `dispose` on each member.

To find a wealth of information about memory management and garbage collection in various systems and languages, browse the Memory Management Reference at `http://www.memorymanagement.org/`.

Having redefined the `Filter` interface, let's next examine an implementation that builds atop a C++ class.

Porting the channel-mixing filters to C++

C++ (like its ancestor, C) separates definitions from implementations. Whereas a Java class is defined and implemented in one file, a C++ class or function is typically defined in a header file (with the extension `.h` or `.hpp`) and implemented in a source file (with the extension `.cpp`). The header contains a full interface specification that other headers and source files might need in order to implement or use the class. Often, multiple files depend on a given header and, thus, the header will be imported multiple times from multiple files. We must manually ensure that the class is defined just once, regardless of how many times its header is imported. To achieve this, we wrap our class definition (or other definitions) inside a set of compile-time directives named an **include guard**, which looks like this:

```
#ifndef MY_CLASS
#define MY_CLASS

// ... TODO: Define MyClass here.

#endif
```

Other concepts that are relevant to C++ header files—such as imports, namespaces, classes, methods, instance variables, and protection levels—should be familiar to Java users. C++ namespaces are optional, but, like Java namespaces, they serve the important purpose of qualifying the names of a group of classes (and functions). Thus, if we happen to import two classes of the same name, we can disambiguate between them, as long as they belong to different namespaces. C++ protection levels, like Java protection levels, include `public` (which makes a method or variable visible even outside its class), `protected` (visible inside the class and its subclasses), and `private` (visible only inside the class). C++ protection levels may also have the modifier `friend` (visible to a specified other class or function). This is somewhat comparable to Java's protection level `internal` (visible to other classes in the same namespace).

Let's move on to an example. We will port our `RecolorRCFilter` class from Java to C++. Remember that this class has an `apply` method and, as an instance variable, it has a collection of matrices (where it stores the intermediate results of the color channel manipulations). The C++ type of these matrices will be `cv::Mat` (the `Mat` class in the `cv` namespace). This type is defined in OpenCV's core module, whose header file we will import. Let's open `RecolorRCFilter.hpp` and write the following header for the `RecolorRCFilter` class:

```
#ifndef RECOLOR_RC_FILTER
#define RECOLOR_RC_FILTER

#include <opencv2/core/core.hpp>

namespace secondsight {

class RecolorRCFilter
{
public: // All subsequent methods or variables are public.
  void apply(cv::Mat &src, cv::Mat &dst);

private: // All subsequent methods or variables are private.
  cv::Mat mChannels[4];
};

} // namespace secondsight

#endif // RECOLOR_RC_FILTER
```

Note that the `apply` method's argument names are preceded by an ampersand (`&`), which means that these arguments are passed by reference (not by value). However, even if we did pass `cv::Mat` objects by value, the underlying image data would be shared between the caller's and callee's copies of the `cv::Mat` (unless we perform some operation that causes either of the copies to reassign or recreate its data buffer). This is a notable feature of the `cv::Mat` design: a `cv::Mat` is not the exclusive owner of its data, but rather, the `cv::Mat` just holds a **pointer** to (an address of) the data.

For more information about `cv::Mat`, see the official documentation at `http://docs.opencv.org/modules/core/doc/basic_structures.html#mat`. Also, look out for Packt Publishing's forthcoming book *OpenCV Blueprints*. Among other topics, this book addresses the pitfalls and best practices of sharing data between different stages (and potentially different libraries) of an image-processing pipeline. Such a discussion fosters a deeper understanding of how and why OpenCV (and `cv::Mat` particularly) views raw image data.

Now, let's write the corresponding source file, `RecolorRCFilter.cpp`. Here, we must implement the bodies of all the methods declared in the header—in this case, just the `apply` method. The implementation must refer to methods by their fully qualified names, such as `secondsight::RecolorRCFilter::apply` (the `apply` method in the `RecolorRCFilter` class in the `secondsight` namespace), except that we may omit the namespace if we have previously made a statement such as `using namespace secondsight;`. Here is the complete implementation:

```
#include "RecolorRCFilter.hpp"

using namespace secondsight;

void RecolorRCFilter::apply(cv::Mat &src, cv::Mat &dst)
{
  cv::split(src, mChannels);

  cv::Mat g = mChannels[1];
  cv::Mat b = mChannels[2];

  // dst.g = 0.5 * src.g + 0.5 * src.b
  cv::addWeighted(g, 0.5, b,  0.5, 0.0, g);

  // dst.b = dst.g
  g.copyTo(b);

  cv::merge(mChannels, 4, dst);
}
```

Note that the relevant OpenCV functions have the same names and nearly the same arguments in the Java and C++ versions. This is often the case.

Our next task is to write JNI functions that expose the functionality of the C++ class to Java callers. JNI functions must be wrapped in a block that looks like this:

```
extern "C" {
  // TODO: Put JNI functions here.
}
```

When a C++ compiler encounters an `extern "C"` block, it compiles the contents in such a way that either a C or C++ linker will be able to understand them. This is crucial because JNI (like many other interoperability layers) understands C, but not C++. Thus, the implementation of `SecondSightJNI.cpp` starts with an `extern "C"` block, as seen in the following code:

```
#include <jni.h>

#include "RecolorRCFilter.hpp"

using namespace secondsight;

extern "C" {
```

JNI functions tend to have long names because the function name corresponds to a fully qualified Java method name. Also, JNI defines many special types that represent Java primitives, classes, and objects. Consider the following signature and body of a JNI function, whose fully qualified Java name will be `com.nummist.secondsight.filters.mixer.RecolorRCFilter.newSelf` (a method called `newSelf` in the `RecolorRCFilter` class in the `com.nummist.secondsight.filters.mixer` namespace):

```
JNIEXPORT jlong JNICALL
Java_com_nummist_secondsight_filters_mixer_RecolorRCFilter_newSelf(
    JNIEnv *env, jclass clazz)
{
  RecolorRCFilter *self = new RecolorRCFilter();
  return (jlong)self;
}
```

The first two arguments represent the Java environment and Java class that own the method. (In this case, the Java class is RecolorRCFilter.) These two arguments are passed implicitly anytime we make a JNI function call from the Java side. Our `newSelf` function's role is to create a C++ object and provide the Java side with a reference to this object. The return type, `jlong`, is a Java `long`, which we can use to represent the memory address of the C++ object. Note that we use C++'s `new` operator to create an object and get its memory address. C++ understands that this address stores an instance of a particular type (in this case, `RecolorRCFilter`), and we say that the address is a `RecolorRCFilter` pointer or `RecolorRCFilter *` (where `*` means pointer). Java does not understand this C++ type information, so we just cast the pointer to `jlong` before returning it to the Java side. Our other JNI functions will accept a `selfAddr` argument, which will enable the Java side to specify an instance of the `RecolorRCFilter` class. Thus, the address becomes a key that maps a Java object to a corresponding C++ object.

Whenever we create a C++ object using the `new` operator, we become responsible for the object's lifecycle. Specifically, when the object is no longer needed, we must destroy it (thereby freeing its memory) using the `delete` operator. After freeing the memory at an address, it is a good policy to set the pointer's value to 0 (also named `NULL` in C++), which means that the pointer does not point to anything anymore. The following JNI function allows a Java caller to delete the `RecolorRCFilter` C++ object at a given address, if the address is not already 0 or `NULL`:

```
JNIEXPORT void JNICALL
Java_com_nummist_secondsight_filters_mixer_RecolorRCFilter_deleteSelf(
    JNIEnv *env, jclass clazz, jlong selfAddr)
{
  if (selfAddr != 0)
  {
    RecolorRCFilter *self = (RecolorRCFilter *)selfAddr;
    delete self;
  }
}
```

Note that we cast the address to a `RecolorRCFilter *`. This step is crucial. The `delete` operator needs the pointer's type information.

To expose the `apply` method to a Java caller, we will again use objects' addresses as JNI arguments. This time, we need the address of not only a `RecolorRCFilter` object, but also the source and destination matrices. Consider the following implementation:

```
JNIEXPORT void JNICALL
Java_com_nummist_secondsight_filters_mixer_RecolorRCFilter_apply(
    JNIEnv *env, jclass clazz, jlong selfAddr, jlong srcAddr,
```

```
        jlong dstAddr)
    {
      if (selfAddr != 0)
      {
        RecolorRCFilter *self = (RecolorRCFilter *)selfAddr;
        cv::Mat &src = *(cv::Mat *)srcAddr;
        cv::Mat &dst = *(cv::Mat *)dstAddr;
        self->apply(src, dst);
      }
    }

    } // extern "C"
```

The syntax of a line such as `cv::Mat &src = *(cv::Mat *)srcAddr;` might be a bit baffling, even if you have used C++ before. Let's read the statement from right to left. It means: cast the address to a matrix pointer, **dereference** it (get the matrix at the address), and then store a reference to the matrix (not a copy by value). The left-hand asterisk (`*`) is the dereference operator, while the right-hand asterisk is a part of the casting operator. Meanwhile, the line `self->apply(src, dst);` uses the arrow (`->`) operator, which means that we are dereferencing the left-hand side and then accessing one of its methods or fields. An equivalent but more verbose expression would be `(*self).apply(src, dst);`.

Our work in C++ is done for this filter. Now, we are going to rewrite the contents of `RecolorRCFilter.java` from scratch so that our Java class becomes a thin wrapper around the C++ class. The wrapper class will have just one instance variable, `long mSelfAddr`, which will store the memory address of the corresponding C++ object. The Java implementation of `RecolorRCFilter` begins like this:

```
public final class RecolorRCFilter implements Filter {

  private long mSelfAddr;
```

Before using any of the functionality in `libSecondSight.so` (our built C++ library), we must call a static method, `System.loadLibrary("SecondSight");`. Note that the argument is a shortened version of the library's filename, omitting `lib` and `.so`. A `static` initialization block, such as the following, is a good place to load the library:

```
  static {
    // Load the native library if it is not already loaded.
    System.loadLibrary("SecondSight");
  }
```

We can safely put the `loadLibrary` call in multiple classes' static initialization blocks. Despite multiple calls, a given library is loaded only once and is shared across the entire application.

The Java class's constructor calls the C++ class's constructor via the `newSelf` JNI function. This function returns the C++ object's memory address, which the Java object stores in the `mSelfAddr` instance variable. The implementation is a one-liner:

```
public RecolorRCFilter() {
  mSelfAddr = newSelf();
}
```

Conversely, the `dispose` method (on the Java side) invokes the `delete` operator (on the C++ object) via the `deleteSelf` JNI function. Then, the Java object sets its `mSelfAddr` variable to 0 (which is equivalent to the C++ `NULL` pointer) so that we cannot attempt to access the deleted memory anymore. Thanks to NULL checking in the implementation of `deleteSelf`, it is safe to call `dispose` multiple times. The implementation is a two-liner:

```
@Override
public void dispose() {
  deleteSelf(mSelfAddr);
  mSelfAddr = 0;
}
```

We also override the `finalize` method, as follows, to ensure that our `dispose` method is always called at least once before the object is garbage-collected:

```
@Override
protected void finalize() throws Throwable {
  dispose();
}
```

The Java class's `apply` method is another simple wrapper around a JNI function. Remember that the JNI function requires the memory addresses of the `cv::Mat` objects that are the source and destination matrices. Conveniently, `org.opencv.core.Mat` provides a method, `getNativeObjectAddr`, which returns the memory address of the associated `cv::Mat` object. Thus, note that OpenCV4Android offers the same kind of wrapper as we use: a Java object holds the address of a C++ object. Observe the use of the `RecolorRCFilter` object's address (`mSelfAddr`) and the source and destination matrices' addresses in the following implementation of `apply`:

```
@Override
public void apply(final Mat src, final Mat dst) {
```

```
        apply(mSelfAddr, src.getNativeObjAddr(),
            dst.getNativeObjAddr());
    }
```

Finally, we declare the JNI functions using the `native` keyword. The function names, return types, and argument types must match the ones in `SecondSightJNI.cpp`. (Otherwise, we would get an error at runtime.) Here is the relevant code:

```
    private static native long newSelf();
    private static native void deleteSelf(long selfAddr);
    private static native void apply(long selfAddr, long srcAddr,
        long dstAddr);
}
```

We are now ready to test our first set of C++ and JNI code! Ensure that the changes to all C++ files are saved, run `ndk-build`, rebuild the project in Eclipse, and then try the new build of Second Sight on your Android device. The red-cyan filter should behave exactly the same way as it did in our previous builds.

For the C++ and JNI versions of `RecolorCMVFilter` and `RecolorRGVFilter`, download this chapter's code bundle. As these classes are very similar to `RecolorRCFilter`, we will omit their code here to save time. Next, let's look at porting `StrokeEdgesFilter`, which uses a convolution filter to enhance edges.

Porting the edge-enhancing filter to C++

We have only one edge-enhancing filter class to port to C++. Let's start by writing its header. `StrokeEdgesFilter.hpp` will be a simple header file, which specifies that the `StrokeEdgesFilter` class has a constructor, an `apply` method, and two matrices (where it stores the edge-finding kernel and intermediate computations). This definition of the class has just one dependency, OpenCV's `core` module. Here is the header file's complete code:

```
#ifndef STROKE_EDGES_FILTER
#define STROKE_EDGES_FILTER

#include <opencv2/core/core.hpp>

namespace secondsight {

class StrokeEdgesFilter
{
public:
  StrokeEdgesFilter();
```

```
  void apply(cv::Mat &src, cv::Mat &dst);

private:
  cv::Mat mKernel;
  cv::Mat mEdges;
};

} // namespace secondsight

#endif // STROKE_EDGES_FILTER
```

Moving on to implementation, we can begin `StrokeEdgesFilter.cpp` by importing the header file and an additional dependency, OpenCV's `imgproc` module:

```
#include <opencv2/imgproc/imgproc.hpp>

#include "StrokeEdgesFilter.hpp"

using namespace secondsight;
```

Now, we can consider the bodies of methods. The constructor needs to create the 5 x 5 kernel matrix and set its values. This is a case where we need to explicitly create a matrix of a specified size and type. (By contrast, many OpenCV functions may implicitly create or recreate the matrices that are given as arguments, depending on the sizes and types that the function implies.) The kernel contains small numbers that may be positive or negative, so let's use 8-bit unsigned integers as the data type. Here is the constructor's implementation, which shows the use of the `create` method of `cv::Mat` to set the size and type, as well as the `at` method to set the elements' values:

```
StrokeEdgesFilter::StrokeEdgesFilter()
{
  mKernel.create(5, 5, CV_8S);
  mKernel.at<char>(0, 0) = 0;
  mKernel.at<char>(0, 1) = 0;
  mKernel.at<char>(0, 2) = 1;
  mKernel.at<char>(0, 3) = 0;
  mKernel.at<char>(0, 4) = 0;
  mKernel.at<char>(1, 0) = 0;
  mKernel.at<char>(1, 1) = 1;
  mKernel.at<char>(1, 2) = 2;
  mKernel.at<char>(1, 3) = 1;
  mKernel.at<char>(1, 4) = 0;
  mKernel.at<char>(2, 0) = 1;
  mKernel.at<char>(2, 1) = 2;
  mKernel.at<char>(2, 2) = -16;
```

```
    mKernel.at<char>(2, 3) = 2;
    mKernel.at<char>(2, 4) = 1;
    mKernel.at<char>(3, 0) = 0;
    mKernel.at<char>(3, 1) = 1;
    mKernel.at<char>(3, 2) = 2;
    mKernel.at<char>(3, 3) = 1;
    mKernel.at<char>(3, 4) = 0;
    mKernel.at<char>(4, 0) = 0;
    mKernel.at<char>(4, 1) = 0;
    mKernel.at<char>(4, 2) = 1;
    mKernel.at<char>(4, 3) = 0;
    mKernel.at<char>(4, 4) = 0;
}
```

The `apply` method's C++ implementation uses the same sequence of OpenCV functions as we saw in its old Java implementation. By filtering with the edge-finding kernel and inverting the result, we get an image of black edges on a white background. We then multiply this intermediate result with the source image to produce a destination image with black edges on a normal, colored background. Here is the relevant code:

```
void StrokeEdgesFilter::apply(cv::Mat &src, cv::Mat &dst)
{
  cv::filter2D(src, mEdges, -1, mKernel);
  cv::bitwise_not(mEdges, mEdges);
  cv::multiply(src, mEdges, dst, 1.0/255.0);
}
```

This completes the C++ version of the `StrokeEdgesFilter` class. We also need to modify `SecondSightJNI.cpp` and `StrokeEdgesFilter.java`. These modifications are very similar to the code we wrote for the `RecolorRC` class (in the previous section), so to save time, they are omitted here. You can find them in this chapter's downloadable code bundle.

With the necessary modifications in place, you might want to try building and running Second Sight again. Ensure that the changes to all C++ files are saved, run `ndk-build`, rebuild the project in Eclipse, and then test the app on your Android device. All filters should function just as they did in the previous chapters. If you encounter any problems, try running ndk-build clean && ndk-build to ensure that our C++ library is rebuilt from scratch.

Our final (and largest) set of modifications pertains to the image-tracking filter. Let's see how this complex class looks in C++!

Porting the ARFilter to C++

We will port the `ImageDetectionFilter` class to C++. However, for the sake of limiting this project's scope, we will leave `CameraProjectionAdapter` and `ARCubeRenderer` as pure Java classes.

Once again, let's start with a header file, `ImageDetectionFilter.hpp`. This header uses types from OpenCV's `core` and `features2d` modules, plus the Standard Template Library's `std::vector` class. (The latter class is analogous to Java's `ArrayList`.) Accordingly, we can begin the header with the following code:

```
#ifndef IMAGE_DETECTION_FILTER
#define IMAGE_DETECTION_FILTER

#include <vector>

#include <opencv2/core/core.hpp>
#include <opencv2/features2d/features2d.hpp>

namespace secondsight {
```

The C++ version of `ImageDetectionFilter` has the same methods as the Java version. The arguments differ slightly, as we will still require the Java side to provide the reference image (for the constructor) and the projection matrix (for the `apply` method). Let's declare the C++ methods like this:

```
class ImageDetectionFilter
{
public:
  ImageDetectionFilter(cv::Mat &referenceImageBGR,
      double realSize);
  float *getGLPose();
  void apply(cv::Mat &src, cv::Mat &dst, cv::Mat &projection);

private:
  void findPose(cv::Mat &projection);
  void draw(cv::Mat &src, cv::Mat &dst);
```

The instance variables of `ImageDetectionFilter` include several `cv::Mat` objects and `std::vector` objects. Our C++ definition of `ImageDetectionFilter` also has several `cv::Ptr` (OpenCV pointer) objects, which store references to the feature detector, descriptor extractor, and descriptor matcher. Unlike the plain old pointers that we previously encountered while implementing JNI functions, `cv::Ptr` is a **smart pointer**, which means that it automatically manages the deletion of the referenced object after nothing references it anymore. One of the roles of smart pointers in OpenCV is to provide a way of delaying the creation of complex objects. (The pointer can just refer to `NULL` until we are ready to create a real object for it.) Here are the declarations of the instance variables:

```
// The reference image (this detector's target).
cv::Mat mReferenceImage;
// Features of the reference image.
std::vector<cv::KeyPoint> mReferenceKeypoints;
// Descriptors of the reference image's features.
cv::Mat mReferenceDescriptors;
// The corner coordinates of the reference image, in pixels.
cv::Mat mReferenceCorners;
// The reference image's corner coordinates, in 3D, in real
// units.
std::vector<cv::Point3f> mReferenceCorners3D;

// Features of the scene (the current frame).
std::vector<cv::KeyPoint> mSceneKeypoints;
// Descriptors of the scene's features.
cv::Mat mSceneDescriptors;
// Tentative corner coordinates detected in the scene, in
// pixels.
cv::Mat mCandidateSceneCorners;

// A grayscale version of the scene.
cv::Mat mGraySrc;
// Tentative matches of the scene features and reference features.
std::vector<cv::DMatch> mMatches;

// A feature detector, which finds features in images, and
// descriptor extractor, which creates descriptors of features.
cv::Ptr<cv::Feature2D> mFeatureDetectorAndDescriptorExtractor;
// A descriptor matcher, which matches features based on their
// descriptors.
cv::Ptr<cv::DescriptorMatcher> mDescriptorMatcher;

// Distortion coefficients of the camera's lens.
```

```
    cv::Mat mDistCoeffs;

    // The Euler angles of the detected target.
    cv::Mat mRVec;
    // The XYZ coordinates of the detected target.
    cv::Mat mTVec;
    // The rotation matrix of the detected target.
    cv::Mat mRotation;
    // The OpenGL pose matrix of the detected target.
    float mGLPose[16];

    // Whether the target is currently detected.
    bool mTargetFound;
};

} // namespace secondsight

#endif // IMAGE_DETECTION_FILTER
```

Adapting the code to OpenCV 2.x

OpenCV 3.x's C++ interface uses one class, `Feature2D`, for feature detection and descriptor extraction, whereas OpenCV 2.x uses two classes, `FeatureDetector` and `DescriptorExtractor`. Therefore, replace the following line:

```
cv::Ptr<cv::Feature2D>
mFeatureDetectorAndDescriptorExtractor;
```

Use the following lines instead:

```
cv::Ptr<cv::FeatureDetector> mFeatureDetector;
cv::Ptr<cv::DescriptorExtractor>
mDescriptorExtractor;
```

Having finished the header, let's open `ImageDetectionFilter.cpp` to write the implementation. Besides OpenCV's `core` and `features2d` modules (which we already imported in the header), we need to import the `imgproc` and `calib3d` modules. Also, we need the C standard library's `float` module because we need to initialize some local variables to the maximum value of `float`. Here are the import statements and our usual `namespace` declaration:

```
#include <float.h>

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>
```

```
#include "ImageDetectionFilter.hpp"

using namespace secondsight;
```

The constructor's implementation needs to process the reference image, which it receives as an argument (from the Java side). The first step is to convert the image from the BGR format to the two formats that we will use, gray and RGBA. These conversions are seen in the following code:

```
ImageDetectionFilter::ImageDetectionFilter(
    cv::Mat &referenceImageBGR, double realSize)
{
  // Create grayscale and RGBA versions of the reference image.
  cv::Mat referenceImageGray;
  cv::cvtColor(referenceImageBGR, referenceImageGray,
      cv::COLOR_BGR2GRAY);
  cv::cvtColor(referenceImageBGR, mReferenceImage,
      cv::COLOR_BGR2RGBA);
```

Next, the constructor needs to define the coordinates of the reference image's corners in 2D and 3D. Looking at the following code, note the use of the `at` method of `cv::Mat` to set matrix elements and the `push_back` method of `std::vector` to append elements:

```
  int cols = referenceImageGray.cols;
  int rows = referenceImageGray.rows;

  // Store the reference image's corner coordinates, in pixels.
  mReferenceCorners.create(4, 1, CV_32FC2);
  mReferenceCorners.at<cv::Vec2f>(0, 0)[0] = 0.0f;
  mReferenceCorners.at<cv::Vec2f>(0, 0)[1] = 0.0f;
  mReferenceCorners.at<cv::Vec2f>(1, 0)[0] = cols;
  mReferenceCorners.at<cv::Vec2f>(1, 0)[1] = 0.0f;
  mReferenceCorners.at<cv::Vec2f>(2, 0)[0] = cols;
  mReferenceCorners.at<cv::Vec2f>(2, 0)[1] = rows;
  mReferenceCorners.at<cv::Vec2f>(3, 0)[0] = 0.0f;
  mReferenceCorners.at<cv::Vec2f>(3, 0)[1] = rows;

  // Compute the image's width and height in real units, based
  // on the specified real size of the image's smaller dimension.
  float aspectRatio = (float)cols /(float)rows;
  float halfRealWidth;
  float halfRealHeight;
  if (cols > rows) {
    halfRealHeight = 0.5f * realSize;
```

```
    halfRealWidth = halfRealHeight * aspectRatio;
  } else {
    halfRealWidth = 0.5f * realSize;
    halfRealHeight = halfRealWidth / aspectRatio;
  }

  // Define the real corner coordinates of the printed image
  // so that it normally lies in the xy plane (like a painting
  // or poster on a wall).
  // That is, +z normally points out of the page toward the
  // viewer.
  mReferenceCorners3D.push_back(
      cv::Point3f(-halfRealWidth, -halfRealHeight, 0.0f));
  mReferenceCorners3D.push_back(
      cv::Point3f( halfRealWidth, -halfRealHeight, 0.0f));
  mReferenceCorners3D.push_back(
      cv::Point3f( halfRealWidth,  halfRealHeight, 0.0f));
  mReferenceCorners3D.push_back(
      cv::Point3f(-halfRealWidth,  halfRealHeight, 0.0f));
```

To finish the constructor's implementation, we will create and use the feature detector, descriptor extractor, and descriptor matcher. Remember that our `ImageDetectionFilter` object stores smart pointers (`cv::Ptr` objects) that point to the detector, extractor, and matcher. The `cv::Ptr` class supports the asterisk and arrow operators, mimicking the way that these operators apply to plain old pointers. Thus, we can use a familiar syntax for dereferencing OpenCV's smart pointers. For a practical example, consider the following code, which completes the constructor:

```
  // Create the feature detector, descriptor extractor, and
  // descriptor matcher.
  mFeatureDetector = cv::FeatureDetector::create("ORB");
  mDescriptorExtractor = cv::DescriptorExtractor::create("ORB");
  mDescriptorMatcher = cv::DescriptorMatcher::create(
      "BruteForce-HammingLUT");

  // Detect the reference features and compute their descriptors.
  mFeatureDetector->detect(referenceImageGray,
      mReferenceKeypoints);
  mDescriptorExtractor->compute(referenceImageGray,
      mReferenceKeypoints, mReferenceDescriptors);

  mCandidateSceneCorners.create(4, 1, CV_32FC2);

  // Assume no distortion.
```

```
    mDistCoeffs.zeros(4, 1, CV_64F);

    mTargetFound = false;
}
```

Adapting the code to OpenCV 2.x

OpenCV 3.x's C++ interface uses one class, `Feature2D`, for feature detection and descriptor extraction, whereas OpenCV 2.x uses two classes, `FeatureDetector` and `DescriptorExtractor`. Therefore, replace the following lines:

```
mFeatureDetectorAndDescriptorExtractor =
     cv::ORB::create();
mDescriptorMatcher = cv::DescriptorMatcher::create(
    "BruteForce-HammingLUT");

mFeatureDetectorAndDescriptorExtractor->detect(
    referenceImageGray, mReferenceKeypoints);
mFeatureDetectorAndDescriptorExtractor->compute(
    referenceImageGray, mReferenceKeypoints,
    mReferenceDescriptors);
```

Use the following lines instead:

```
mFeatureDetector = cv::FeatureDetector::create("ORB");
mDescriptorExtractor =
    cv::DescriptorExtractor::create("ORB");
mDescriptorMatcher = cv::DescriptorMatcher::create(
    "BruteForce-HammingLUT");

mFeatureDetector->detect(referenceImageGray,
    mReferenceKeypoints);
mDescriptorExtractor->compute(referenceImageGray,
    mReferenceKeypoints, mReferenceDescriptors);
```

The `getGLPose` method simply returns the OpenGL pose matrix (if the target has been found) or `NULL` (if it has not been found):

```
float *ImageDetectionFilter::getGLPose()
{
  return (mTargetFound ? mGLPose : NULL);
}
```

The `apply` method finds keypoints, descriptors, and matches. Then, it calls helper methods to attempt to find the target's 3D pose and draw a preview of the target (in case the user has not yet found the target). The C++ implementation is similar to the old Java implementation, except that the projection matrix is now an argument (to be passed from the Java side to the C++ side). Here is the C++ code:

```
void ImageDetectionFilter::apply(cv::Mat &src, cv::Mat &dst,
    cv::Mat &projection)
{
  // Convert the scene to grayscale.
  cv::cvtColor(src, mGraySrc, cv::COLOR_RGBA2GRAY);

  // Detect the scene features, compute their descriptors,
  // and match the scene descriptors to reference descriptors.
  mFeatureDetectorAndDescriptorExtractor->detect(
      referenceImageGray, mReferenceKeypoints);
  mFeatureDetectorAndDescriptorExtractor->compute(
      referenceImageGray, mReferenceKeypoints,
      mReferenceDescriptors);
  mDescriptorMatcher->match(mSceneDescriptors,
      mReferenceDescriptors, mMatches);

  // Attempt to find the target image's 3D pose in the scene.
  findPose(projection);

  // If the pose has not been found, draw a thumbnail of the
  // target image.
  draw(src, dst);
}
```

Adapting the code to OpenCV 2.x

OpenCV 3.x's C++ interface uses one class, `Feature2D`, for feature detection and descriptor extraction, whereas OpenCV 2.x uses two classes, `FeatureDetector` and `DescriptorExtractor`. Therefore, consider the following lines:

```
mFeatureDetectorAndDescriptorExtractor->detect(
    referenceImageGray, mReferenceKeypoints);
mFeatureDetectorAndDescriptorExtractor->compute(
    referenceImageGray, mReferenceKeypoints,
    mReferenceDescriptors);
```

We must replace them with:

```
mFeatureDetector->detect(mGraySrc, mSceneKeypoints);
mDescriptorExtractor->compute(mGraySrc,
mSceneKeypoints,
    mSceneDescriptors);
```

The `findPose` helper method has a long implementation, which we will consider in several chunks. First, if we have enough matches to possibly find the pose, we will proceed to find the minimum and maximum (best and worst) distances among the matches. This is an opportunity to write a loop that iterates over the elements of `std::vector`, as seen in the following code:

```
void ImageDetectionFilter::findPose(cv::Mat &projection)
{
  if (mMatches.size() < 4) {
    // There are too few matches to find the pose.
    return;
  }

  // Calculate the max and min distances between keypoints.
  float maxDist = 0.0f;
  float minDist = FLT_MAX;
  for (int i = 0; i < mMatches.size(); i++) {
    cv::DMatch match = mMatches[i];
    float dist = match.distance;
    if (dist < minDist) {
      minDist = dist;
    }
    if (dist > maxDist) {
      maxDist = dist;
    }
  }
```

If the matches are plausible overall (based on the minimum distance), we will proceed to collect good matching keypoints in the `std::vector<cv::Point2f>>` objects. (These are analogous to the `List<Point>` objects that we used in the Java implementation.) Here is the relevant C++ code:

```
// The thresholds for minDist are chosen subjectively
// based on testing. The unit is not related to pixel
// distances; it is related to the number of failed tests
// for similarity between the matched descriptors.
if (minDist > 50.0) {
  // The target is completely lost.
  mTargetFound = false;
  return;
} else if (minDist > 25.0) {
  // The target is lost but maybe it is still close.
  // Keep using any previously found pose.
  return;
}

// Identify "good" keypoints based on match distance.
std::vector<cv::Point2f> goodReferencePoints;
std::vector<cv::Point2f> goodScenePoints;
double maxGoodMatchDist = 1.75 * minDist;
for(int i = 0; i < mMatches.size(); i++) {
  cv::DMatch match = mMatches[i];
  if (match.distance < maxGoodMatchDist) {
    goodReferencePoints.push_back(
        mReferenceKeypoints[match.trainIdx].pt);
    goodScenePoints.push_back(
        mSceneKeypoints[match.queryIdx].pt);
  }
}
```

If we have at least four good matching pairs of keypoints, we can find the homography, estimate the corner positions in 3D, and validate that the projected corners form a convex shape. To conform to the argument types of the relevant OpenCV functions, our old Java implementation needed to convert the `List<Point>` objects to `MatOfPoint2f` objects. OpenCV's C++ interface is less restrictive with respect to arguments' types. Most arguments can be either a `cv::Mat` or a `std::vector` object. To make these two types interchangeable, the OpenCV authors wrote a **proxy class** (a substitute that provides a common interface) named `InputArray`. It is able to implicitly construct itself from either a `cv::Mat` or `std::vector`. Thus, anytime you see `InputArray` in OpenCV's C++ API docs, you will know that you can provide either `cv::Mat` or `std::vector`.

The implicit construction of an `InputArray` is cheap and does not copy the image data. The following block of code shows our continued use of `std::vector` objects, without explicit conversion:

```
if (goodReferencePoints.size() < 4 ||
    goodScenePoints.size() < 4) {
  // There are too few good points to find the pose.
  return;
}

// There are enough good points to find the pose.
// (Otherwise, the method would have already returned.)

// Find the homography.
cv::Mat homography = cv::findHomography(
    goodReferencePoints, goodScenePoints);

// Use the homography to project the reference corner
// coordinates into scene coordinates.
cv::perspectiveTransform(mReferenceCorners,
    mCandidateSceneCorners, homography);

// Check whether the corners form a convex polygon. If not,
// (that is, if the corners form a concave polygon), the
// detection result is invalid because no real perspective can
// make the corners of a rectangular image look like a concave
// polygon!
if (!cv::isContourConvex(mCandidateSceneCorners)) {
  return;
}
```

The remainder of the method's implementation is responsible for finding the target's 3D pose (based on the corners) and converting this pose to OpenGL's format. Note the use of the `at` method of `cv::Mat` to get and set elements of the matrices:

```
// Find the target's Euler angles and XYZ coordinates.
cv::solvePnP(mReferenceCorners3D, mCandidateSceneCorners,
    projection, mDistCoeffs, mRVec, mTVec);

// Positive y is up in OpenGL, down in OpenCV.
// Positive z is backward in OpenGL, forward in OpenCV.
// Positive angles are counter-clockwise in OpenGL,
// clockwise in OpenCV.
// Thus, x angles are negated but y and z angles are
// double-negated (that is, unchanged).
```

```
    // Meanwhile, y and z positions are negated.

    mRVec.at<double>(0, 0) *= -1.0; // negate x angle

    // Convert the Euler angles to a 3x3 rotation matrix.
    cv::Rodrigues(mRVec, mRotation);

    // OpenCV's matrix format is transposed, relative to
    // OpenGL's matrix format.
    mGLPose[0]  =  (float)mRotation.at<double>(0, 0);
    mGLPose[1]  =  (float)mRotation.at<double>(0, 1);
    mGLPose[2]  =  (float)mRotation.at<double>(0, 2);
    mGLPose[3]  =  0.0f;
    mGLPose[4]  =  (float)mRotation.at<double>(1, 0);
    mGLPose[5]  =  (float)mRotation.at<double>(1, 1);
    mGLPose[6]  =  (float)mRotation.at<double>(1, 2);
    mGLPose[7]  =  0.0f;
    mGLPose[8]  =  (float)mRotation.at<double>(2, 0);
    mGLPose[9]  =  (float)mRotation.at<double>(2, 1);
    mGLPose[10] =  (float)mRotation.at<double>(2, 2);
    mGLPose[11] =  0.0f;
    mGLPose[12] =  (float)mTVec.at<double>(0, 0);
    mGLPose[13] = -(float)mTVec.at<double>(1, 0); // negate y position
    mGLPose[14] = -(float)mTVec.at<double>(2, 0); // negate z position
    mGLPose[15] =  1.0f;

    mTargetFound = true;
  }
```

For the `draw` helper method, the C++ implementation looks somewhat similar to the old Java implementation. However, `cv::Mat` (unlike `org.opencv.core.Mat` in the Java interface) has no `submat` method; instead, it has an `adjustROI` method whose arguments are offsets from the top, bottom, left, and right edges of `cv::Mat`. In our case, the offsets are negative because we are shrinking the region of interest (ROI). Afterward, we need to restore the ROI to its original state, so we use positive offsets. Here is the `draw` method's implementation, with the use of `adjustROI` highlighted:

```
  void ImageDetectionFilter::draw(cv::Mat &src, cv::Mat &dst)
  {
    if (&src != &dst) {
      src.copyTo(dst);
    }

    if (!mTargetFound) {
```

```
        // The target has not been found.

        // Draw a thumbnail of the target in the upper-left
        // corner so that the user knows what it is.

        // Compute the thumbnail's larger dimension as half the
        // video frame's smaller dimension.
        int height = mReferenceImage.rows;
        int width = mReferenceImage.cols;
        int maxDimension = MIN(dst.rows, dst.cols) / 2;
        double aspectRatio = width / (double)height;
        if (height > width) {
          height = maxDimension;
          width = (int)(height * aspectRatio);
        } else {
          width = maxDimension;
          height = (int)(width / aspectRatio);
        }

        // Select the region of interest (ROI) where the thumbnail
        // will be drawn.
        int offsetY = height - dst.rows;
        int offsetX = width - dst.cols;
        dst.adjustROI(0, offsetY, 0, offsetX);

        // Copy a resized reference image into the ROI.
        cv::resize(mReferenceImage, dst, dst.size(), 0.0, 0.0,
            cv::INTER_AREA);

        // Deselect the ROI.
        dst.adjustROI(0, -offsetY, 0, -offsetX);
      }
    }
```

Despite the long implementation of the `ImageDetectionFilter` C++ class, the associated JNI functions are only slightly more complicated than the ones we have previously written. First, we must modify `SecondSightJNI.cpp` to import the new header file, as seen in this code snippet:

```
#include <jni.h>

// ...
#include "ImageDetectionFilter.hpp"
```

```
using namespace secondsight;

extern "C" {

// ...
```

Later in the same file, we will add new JNI functions to expose all of the `ImageDetectionFilter`'s public methods, starting with its constructor. The Java side is responsible for providing the reference image's address and its real (printed) size in arbitrary units:

```
JNIEXPORT jlong JNICALL
Java_com_nummist_secondsight_filters_ar_ImageDetectionFilter_newSelf(
    JNIEnv *env, jclass clazz, jlong referenceImageBGRAddr,
    jdouble realSize)
{
  cv::Mat &referenceImageBGR =  *(cv::Mat *)referenceImageBGRAddr;
  ImageDetectionFilter *self = new ImageDetectionFilter(
      referenceImageBGR, realSize);
  return (jlong)self;
}
```

As usual, we expose the `delete` operation in the following manner:

```
JNIEXPORT void JNICALL
Java_com_nummist_secondsight_filters_ar_ImageDetectionFilter_
deleteSelf(
    JNIEnv *env, jclass clazz, jlong selfAddr)
{
  if (selfAddr != 0)
  {
    ImageDetectionFilter *self = (ImageDetectionFilter *)selfAddr;
    delete self;
  }
}
```

To expose the `getGLPose` method, we must create a new Java array that references the same data as the C++ array, as seen in the following code:

```
JNIEXPORT jfloatArray JNICALL
Java_com_nummist_secondsight_filters_ar_ImageDetectionFilter_
getGLPose(
    JNIEnv *env, jclass clazz, jlong selfAddr)
{
  if (selfAddr == 0)
  {
```

```
    return NULL;
  }

  ImageDetectionFilter *self = (ImageDetectionFilter *)selfAddr;
  float *glPoseNative = self->getGLPose();
  if (glPoseNative == NULL)
  {
    return NULL;
  }

  jfloatArray glPoseJava = env->NewFloatArray(16);
  if (glPoseJava != NULL)
  {
    env->SetFloatArrayRegion(glPoseJava, 0, 16, glPoseNative);
  }
  return glPoseJava;
}
```

To expose the `apply` method, we simply reinterpret the source, destination, and projection matrices' addresses, which come from the Java side as `long` integers:

```
JNIEXPORT void JNICALL
Java_com_nummist_secondsight_filters_ar_ImageDetectionFilter_apply(
    JNIEnv *env, jclass clazz, jlong selfAddr, jlong srcAddr,
    jlong dstAddr, jlong projectionAddr)
{
  if (selfAddr != 0)
  {
    ImageDetectionFilter *self = (ImageDetectionFilter *)selfAddr;
    cv::Mat &src = *(cv::Mat *)srcAddr;
    cv::Mat &dst = *(cv::Mat *)dstAddr;
    cv::Mat &projection = *(cv::Mat *)projectionAddr;
    self->apply(src, dst, projection);
  }
}

} // extern "C"
```

Now, let's turn our attention to the Java wrapper. As usual, it stores the address of the C++ object. It also has an instance of `CameraProjectionAdapter`, which is a pure Java class that we have not modified from the previous chapter. Our modified `ImageDetectionFilter.java` file will begin as follows:

```
public final class ImageDetectionFilter implements ARFilter {

  private long mSelfAddr;
```

```
private final CameraProjectionAdapter mCameraProjectionAdapter;

static {
  // Load the native library if it is not already loaded.
  System.loadLibrary("SecondSight");
}
```

The constructor has the same arguments as it did in its previous version. It loads the reference image from the app's resources and then it passes this image, along with the image's real (printed) size in arbitrary units, to the C++ class's constructor. Here is the revised implementation of the Java class's constructor:

```
public ImageDetectionFilter(final Context context,
    final int referenceImageResourceID,
    final CameraProjectionAdapter cameraProjectionAdapter,
    final double realSize)
        throws IOException {
  final Mat referenceImageBGR = Utils.loadResource(context,
      referenceImageResourceID,
      Imgcodecs.CV_LOAD_IMAGE_COLOR);
  mSelfAddr = newSelf(referenceImageBGR.getNativeObjAddr(),
      realSize);
  mCameraProjectionAdapter = cameraProjectionAdapter;
}
```

Adapting the code to OpenCV 2.x

Replace `Imgcodecs.CV_LOAD_IMAGE_COLOR` with `Highgui.CV_LOAD_IMAGE_COLOR`.

As in our other wrapper classes, we need to add the `dispose` and `finalize` methods with the following implementations:

```
@Override
public void dispose() {
  deleteSelf(mSelfAddr);
  mSelfAddr = 0;
}

@Override
protected void finalize() throws Throwable {
  dispose();
}
```

The `getGLPose` method needs to be modified so that it simply returns the result of the equivalent C++ method:

```
@Override
public float[] getGLPose() {
  return getGLPose(mSelfAddr);
}
```

Similarly, our new implementation of the `apply` method will simply pass the source, destination, and projection matrices to the C++ `apply` method. The projection matrix comes from the `CameraProjectionAdapter` object, as seen in the following code:

```
@Override
public void apply(final Mat src, final Mat dst) {
  final Mat projection =
      mCameraProjectionAdapter.getProjectionCV();
  apply(mSelfAddr, src.getNativeObjAddr(),
      dst.getNativeObjAddr(),
      projection.getNativeObjAddr());
}
```

To finish the wrapper, we will make the following declarations of the JNI methods:

```
  private static native long newSelf(long referenceImageBGRAddr,
      double realSize);
  private static native void deleteSelf(long selfAddr);
  private static native float[] getGLPose(long selfAddr);
  private static native void apply(long selfAddr, long srcAddr,
      long dstAddr, long projectionAddr);
}
```

That's all! Ensure that the changes to all the C++ files are saved, run `ndk-build`, rebuild the project in Eclipse, and then test Second Sight on your Android device. The app should behave the same way as the pure Java version that we completed in the previous chapter. We have learned how to achieve the same results using either OpenCV's Java interface or its C++ interface plus JNI!

Learning more about OpenCV and C++

Packt Publishing offers a big selection of OpenCV books, and many of them focus on the library's C++ interface. Here are some fine examples:

- *OpenCV Essentials*, by multiple authors: This is a concise introduction to OpenCV with C++.

- *Learning Image Processing with OpenCV*, by multiple authors: This book emphasizes photo and video enhancement techniques, from beginning to advanced levels.
- *OpenCV Computer Vision Application Programming Cookbook*, by Robert Laganière: This is a book of recipes showing how to efficiently solve common computer vision problems with OpenCV.
- *Mastering OpenCV with Practical Computer Vision Projects*, by multiple authors: This is a set of advanced projects. Each combines multiple features of OpenCV to solve a problem.
- *OpenCV Blueprints* (forthcoming), by multiple authors: This is a collection of advanced tutorials and articles, which is intended to clear up mysteries about building robust applications with OpenCV. A combination of theory, implementation, and benchmarking is covered.

If, instead, you are looking for a general reference guide to C++, check out Bjarne Stroustrup's books, such as *A Tour of C++* (published by Addison-Wesley). Stroustrup is the designer and original implementer of C++.

Summary

We have bridged the gap between Java and C++! More specifically, we have ported several of our Java classes to C++ and compiled these C++ classes, along with JNI functions that are callable from Java. Also, by passing C++ objects' addresses across the JNI boundary, we explored a technique to create Java classes that thinly wrap C++ classes. This technique is the foundation of OpenCV's Java interface, which wraps a C++ interface. Thus, without knowing it, we were already relying on JNI and a C++ library (OpenCV) in our previous chapters! With our new knowledge, we can have greater control over the way an app leverages C++ libraries. We are also in a better position to learn about using OpenCV on other platforms and with other libraries.

Having passed through the "escape hatch" from Java to C++, we have also finished our quick tour of Android and OpenCV. Congratulations on all you learned, and thank you for participating in the success of this second edition of *Android Application Programming with OpenCV*! If you have any questions about the book, or if you would like to tell me about your other OpenCV work, please stay in touch by writing to `josephhowse@nummist.com`. Also, remember to check `http://nummist.com/opencv` for updates and errata.

Until we meet again, I hope you enjoy clear vision and illuminating discoveries!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Thank you for buying
Android Application Programming with OpenCV 3

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

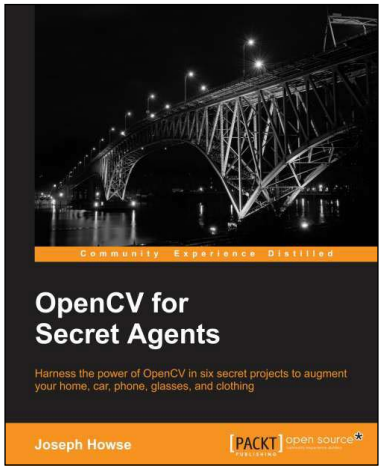
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



OpenCV for Secret Agents

ISBN: 978-1-78328-737-6 Paperback: 302 pages

Harness the power of OpenCV in six secret projects to augment your home, car, phone. glasses, and clothing

1. Build OpenCV apps for the desktop, the Raspberry Pi, Android, and the Unity game engine.
2. Learn real-time techniques that can be used to classify images, detecting and recognizing any person or animal, and studying motion and distance with superhuman precision.
3. Design hands-free interfaces that are practical in home automation, in cars, and in discrete surveillance.

OpenCV Computer Vision Application Programming Cookbook

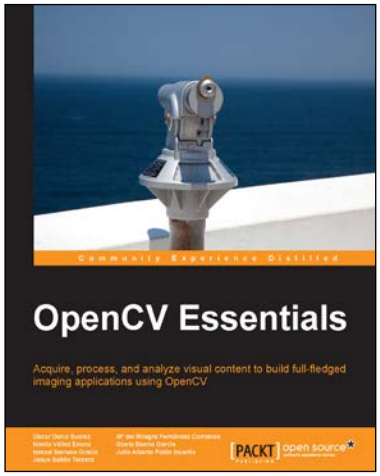
Second Edition

ISBN: 978-1-78216-148-6 Paperback: 374 pages

Over 50 recipes to help you build computer vision applications in C++ using the OpenCV library

1. Master OpenCV, the open source library of the computer vision community.
2. Master fundamental concepts in computer vision and image processing.
3. Learn the important classes and functions of OpenCV with complete working examples applied on real images.

[PACKT] open source
community experience distilled
PUBLISHING

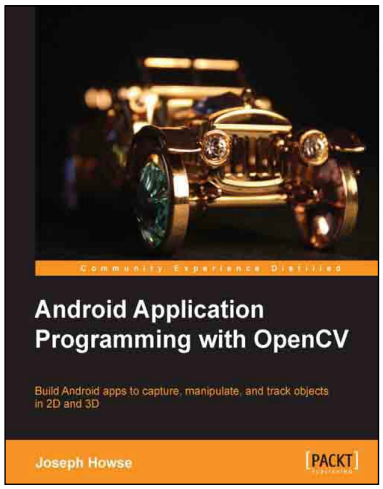


OpenCV Essentials

ISBN: 978-1-78398-424-4 Paperback: 214 pages

Acquire, process, and analyze visual content to build full-fledged imaging applications using OpenCV

1. Create OpenCV programs with a rich user interface.
2. Develop real-world imaging applications using free tools and libraries.
3. Understand the intricate details of OpenCV and its implementation using easy-to-follow examples.



Android Application Programming with OpenCV

ISBN: 978-1-84969-520-6 Paperback: 130 pages

Build Android apps to capture, manipulate, and track objects in 2D and 3D

1. Set up OpenCV and an Android development environment on Windows, Mac, or Linux.
2. Capture and display real-time videos and still images.
3. Manipulate image data using OpenCV and Apache Commons Math.

Please check **www.PacktPub.com** for information on our titles