Community Experience Distilled

OpenCV Computer Vision with Python

Learn to capture videos, manipulate images, and track objects with Python using the OpenCV Library

Joseph Howse

OpenCV Computer Vision with Python

Table of Contents

OpenCV Computer Vision with Python

OpenCV Computer Vision with Python

Copyright © 2013 Packt Publishing

Credits

Author

Joseph Howse

Reviewer(s)

David Millán Escrivá

Abid K.

Acquisition Editor

Erol Staveley

Commissioning Editor

Neha Nagvekar

Technical Editors

Ankita Meshram

Veena Pagare

Project Coordinator

Sneha Modi

Proofreader

Lauren Tobon

Indexer

Indexer

Rekha Nair

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Joseph Howse (Joe) is fanciful. So to him, the virtual world always seemed to reach out into reality. One of his earliest memories is of watching an animated time-bomb on the screen of a Tandy Color Computer. The animation was programmed in BASIC by Joe's older brother, Sam, who explained, "I'm making a bomb. Get ready!" The bomb exploded in a rain of dots and a rumble of beeps as Joe and Sam ran to hide from the fallout.

Today, Joe still fancies that a computer program can blast a tunnel into reality. As a hobby, he likes looking at reality through the tunnel of a digital camera's lens. As a career, he develops augmented reality software, which uses cameras and other sensors to composite real and virtual scenes interactively in real time.

Joe holds a Master of Computer Science degree from Dalhousie University. He does research on software architecture as applied to augmented reality.

Joe works at Ad-Dispatch, an augmented reality company, where he develops applications for mobile devices, kiosks, and the Web.

Joe likes cats, kittens, oceans, and seas. Felines and saline water sustain him. He lives with his multi-species family in Halifax, on Canada's Atlantic coast.

About the Reviewers

David Millán Escrivá was eight years old when he wrote his first program on an 8086 PC with Basic language, which enabled the 2D plotting of basic equations; he started with his computer development relationship and created many applications and games.

In 2005, he finished his studies in IT from the Universitat Politécnica de Valencia with

honors in human-computer interaction supported by computer vision with OpenCV (v0.96). He had a final project based on this subject and published it on HCI Spanish congress.

He participated in Blender source code, an open source and 3D-software project, and worked in his first commercial movie *Plumíferos—Aventuras voladoras* as a Computer Graphics Software Developer.

David now has more than 10 years of experience in IT, with more than seven years experience in computer vision, computer graphics, and pattern recognition working on different projects and startups, applying his knowledge of computer vision, optical character recognition, and augmented reality.

He is the author of the *DamilesBlog* (http://blog.damiles.com), where he publishes research articles and tutorials about OpenCV, computer vision in general, and Optical Character Recognition algorithms. He is the co-author of *Mastering OpenCV with Practical Computer Vision Projects* , Daniel Lélis Baggio, Shervin Emami, David Millán Escrivá, Khvedchenia Ievgen, Naureen Mahmood, Jasonl Saragih, and Roy Shilkrot, Packt Publishing. He is also a reviewer of *GnuPlot Cookbook, Lee Phillips, Packt Publishing*.

Abid K. is a student from India pursuing M.Tech in VLSI Design at National Institute of Technology (Suratkal). He finished his B.Tech in Electronics & Communication. He is particularly interested in developing hardware architectures for image processing and speech processing algorithms.

He started using OpenCV Python in his college days as a hobby. The lack of learning resources on OpenCV Python at that time made him to create his own blog, www.opencvpython.blogspot.com, and he still maintains it. In his free time, he used to answer questions related to OpenCV Python at stackoverflow.com, and those discussions are reflected in his blog articles. He also works as a freelancer during college holidays and even helps school students grow their interest in OpenCV Python and computer vision.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Preface

This book will show you how to use OpenCV's Python bindings to capture video, manipulate images, and track objects with either a normal webcam or a specialized depth sensor, such as the Microsoft Kinect. OpenCV is an open source, cross-platform library that provides building blocks for computer vision experiments and applications. It provides high-level interfaces for capturing, processing, and presenting image data. For example, it abstracts details about camera hardware and array allocation. OpenCV is widely used in both academia and industry.

Today, computer vision can reach consumers in many contexts via webcams, camera phones, and gaming sensors such as the Kinect. For better or worse, people love to be on camera, and as developers, we face a demand for applications that capture images, change their appearance, and extract information from them. OpenCV's Python bindings can help us explore solutions to these requirements in a high-level language and in a standardized data format that is interoperable with scientific libraries such as NumPy and SciPy.

Although OpenCV is high-level and interoperable, it is not necessarily easy for new users. Depending on your needs, OpenCV's versatility may come at the cost of a complicated setup process and some uncertainty about how to translate the available functionality into organized and optimized application code. To help you with these problems, I have endeavored to deliver a concise book with an emphasis on clean setup, clean application design, and a simple understanding of each function's purpose. I hope you will learn from this book's project, outgrow it, and still be able to reuse the development environment and parts of the modular code that we have created together.

Specifically, by the end of this book's first chapter, you can have a development environment that links Python, OpenCV, depth camera libraries (OpenNI, SensorKinect), and general-purpose scientific libraries (NumPy, SciPy). After five chapters, you can have several variations of an entertaining application that manipulates users' faces in a live camera feed. Behind this application, you will have a small library of reusable functions and classes that you can apply in your future computer vision projects. Let's look at the book's progression in more detail.

What this book covers

Chapter 1, *Setting up OpenCV*, lets us examine the steps to setting up Python, OpenCV, and related libraries on Windows, Mac, and Ubuntu. We also discuss OpenCV's community, documentation, and official code samples.

Chapter 2, *Handling Files, Cameras, and GUIs*, helps us discuss OpenCV's I/O functionality. Then, using an object-oriented design, we write an application that displays a live camera feed, handles keyboard input, and writes video and still image files.

Chapter 3, *Filtering Images*, helps us to write image filters using OpenCV, NumPy, and SciPy. The filter effects include linear color manipulations, curve color manipulations,

blurring, sharpening, and outlining edges. We modify our application to apply some of these filters to the live camera feed.

Chapter 4, *Tracking Faces with Haar Cascades*, allows us to write a hierarchical face tracker that uses OpenCV to locate faces, eyes, noses, and mouths in an image. We also write functions for copying and resizing regions of an image. We modify our application so that it finds and manipulates faces in the camera feed.

Chapter 5, *Detecting Foreground/Background Regions and Depth*, helps us learn about the types of data that OpenCV can capture from depth cameras (with the support of OpenNI and SensorKinect). Then, we write functions that use such data to limit an effect to a foreground region. We incorporate this functionality in our application so that we can further refine the face regions before manipulating them.

Appendix A, *Integrating with Pygame*, lets us modify our application to use Pygame instead of OpenCV for handling certain I/O events. (Pygame offers more diverse event handling functionality.)

Appendix B, *Generating Haar Cascades for Custom Targets*, allows us to examine a set of OpenCV tools that enable us to build trackers for any type of object or pattern, not necessarily faces.

What you need for this book

This book provides setup instructions for all the relevant software, including package managers, build tools, Python, NumPy, SciPy, OpenCV, OpenNI, and SensorKinect. The setup instructions are tailored for Windows XP or later versions, Mac OS 10.6 (Snow Leopard) or later versions, and Ubuntu 12.04 or later versions. Other Unix-like operating systems should work too, if you are willing to do your own tailoring of the setup steps. You need a webcam for the core project described in the book. For additional features, some variants of the project use a second webcam or even an OpenNI-compatible depth camera, such as Microsoft Kinect or Asus Xtion PRO.

Who this book is for

This book is great for Python developers who are new to computer vision and who like to learn through application development. It is assumed that you have some previous experience in Python and the command line. A basic understanding of image data (for example, pixels and color channels) would also be helpful.

Conventions

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

Code words in text are shown as follows: "The two executables on Windows are called `ONopencv_createsamples.exe` and `ONopencv_traincascade.exe`."

A block of code is set as follows:

```
negative
    desc.txt
    images
        negative 0.png
        negative 1.png
```

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

```
> cd negative
> forfiles /m images\*.png /c "cmd /c echo @relpath" > desc.txt
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via 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 on 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 to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. 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. The example code for this book is also available from the author's website at http://nummist.com/opencv/.

Errata

Although we have taken 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 the code—we would be grateful if you would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright 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

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it. 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 this book.

Chapter 1. Setting up OpenCV

This chapter is a quick guide to setting up Python 2.7, OpenCV, and related libraries. After setup, we also look at OpenCV's Python sample scripts and documentation.

The following related libraries are covered:

- **NumPy**: This is a dependency of OpenCV's Python bindings. It provides numeric computing functionality, including efficient arrays.

- **SciPy**: This is a scientific computing library that is closely related to NumPy. It is not required by OpenCV but it is useful for manipulating the data in OpenCV images.

- **OpenNI**: This is an optional dependency of OpenCV. It adds support for certain depth cameras, such as Asus XtionPRO.

- **SensorKinect**: This is an OpenNI plugin and optional dependency of OpenCV. It adds support for the Microsoft Kinect depth camera.

For this book's purposes, OpenNI and SensorKinect can be considered optional. They are used throughout Chapter 5, *Separating Foreground/Background Regions Depth*, but are not used in the other chapters or appendices.

At the time of writing, OpenCV 2.4.3 is the latest version. On some operating systems, it is easier to set up an earlier version (2.3.1). The differences between these versions should not affect the project that we are going to build in this book.

Some additional information, particularly about OpenCV's build options and their dependencies, is available in the OpenCV wiki at http://opencv.willowgarage.com/wiki/InstallGuide. However, at the time of writing, the wiki is not up-to-date with OpenCV 2.4.3.

Choosing and using the right setup tools

We are free to choose among various setup tools, depending on our operating system and how much configuration we want to do. Let's take an overview of the tools for Windows, Mac, Ubuntu, and other Unix-like systems.

Making the choice on Windows XP, Windows Vista, Windows 7, or Windows 8

Windows does not come with Python preinstalled. However, installation wizards are available for precompiled Python, NumPy, SciPy, and OpenCV. Alternatively, we can build from source. OpenCV's build system uses CMake for configuration and either Visual Studio or MinGW for compilation.

If we want support for depth cameras including Kinect, we should first install OpenNI and SensorKinect, which are available as precompiled binaries with installation wizards. Then, we must build OpenCV from source.

Note

The precompiled version of OpenCV does not offer support for depth cameras.

On Windows, OpenCV offers better support for 32-bit Python than 64-bit Python. Even if we are building from source, I recommend using 32-bit Python. Fortunately, 32-bit Python works fine on either 32-bit or 64-bit editions of Windows.

Note

Some of the following steps refer to editing the system's Path variable. This task can be done in the **Environment Variables** window of **Control Panel**.

On Windows Vista/Windows 7/Windows 8, open the **Start** menu and launch **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**. Select the **Advanced** tab. Click on the **Environment Variables** button.

Now, under **System variables**, select **Path** and click on the **Edit** button. Make changes as directed. To apply the changes, click on all the **OK** buttons (until we are back in the main window of **Control Panel**). Then, log out and log back in (alternatively, reboot).

Using binary installers (no support for depth cameras)

Here are the steps to set up 32-bit Python 2.7, NumPy, and OpenCV:

1. Download and install 32-bit Python 2.7.3 from http://www.python.org/ftp/python/2.7.3/python-2.7.3.msi.

2. Download and install NumPy 1.6.2 from http://sourceforge.net/projects/numpy/files/NumPy/1.6.2/numpy-1.6.2-win32-superpack-python2.7.exe/download.

3. Download and install SciPy 11.0 from http://sourceforge.net/projects/scipy/files/scipy/0.11.0/scipy-0.11.0-win32-superpack-python2.7.exe/download.

4. Download the self-extracting ZIP of OpenCV 2.4.3 from http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.3/OpenCV-2.4.3.exe/download. Run the self-extracting ZIP and, when prompted, enter any destination folder, which we will refer to as <unzip_destination>. A subfolder, <unzip_destination>\opencv, is created.

5. C o p y `<unzip_destination>\opencv\build\python\2.7\cv2.pyd` to `C:\Python2.7\Lib\site-packages` (assuming we installed Python 2.7 to the default location). Now, the new Python installation can find OpenCV.

6. A final step is necessary if we want Python scripts to run using the new Python installation by default. Edit the system's `Path` variable and append `;C:\Python2.7` (assuming we installed Python 2.7 to the default location). Remove any previous Python paths, such as `;C:\Python2.6`. Log out and log back in (alternatively, reboot).

Using CMake and compilers

Windows does not come with any compilers or CMake. We need to install them. If we want support for depth cameras, including Kinect, we also need to install OpenNI and SensorKinect.

Let's assume that we have already installed 32-bit Python 2.7, NumPy, and SciPy either from binaries (as described previously) or from source. Now, we can proceed with installing compilers and CMake, optionally installing OpenNI and SensorKinect, and then building OpenCV from source:

1. Download and install CMake 2.8.9 from http://www.cmake.org/files/v2.8/cmake-2.8.9-win32-x86.exe. When running the installer, select either **Add CMake to the system PATH for all users** or **Add CMake to the system PATH for current user**.

2. Download and install Microsoft Visual Studio 2010, Microsoft Visual C++ Express 2010, or MinGW. Note that OpenCV 2.4.3 cannot be compiled with the more recent versions (Microsoft Visual Studio 2012 and Microsoft Visual Studio Express 2012).

   For Microsoft Visual Studio 2010, use any installation media you purchased. During installation, include any optional C++ components. Reboot after installation is complete.

   For Microsoft Visual C++ Express 2010, get the installer from http://www.microsoft.com/visualstudio/eng/downloads. Reboot after installation is complete.

   For MinGW get the installer from http://sourceforge.net/projects/mingw/files/Installer/mingw-get-inst/mingw-get-inst-20120426/mingw-get-inst-20120426.exe/download. When running the installer, make sure that the destination path does not contain spaces and that the optional C++ compiler is included. Edit the system's `Path` variable and append `;C:\MinGW\bin` (assuming MinGW is installed to the default location.) Reboot the system.

3. Optionally, download and install OpenNI 1.5.4.0 from http://www.openni.org/wp-content/uploads/2012/12/OpenNI-Win32-1.5.4.0-Dev1.zip (32 bit). Alternatively, for 64-bit Python, use http://www.openni.org/wp-content/uploads/2012/12/OpenNI-Win64-1.5.4.0-Dev.zip (64 bit).

4. Optionally, download and install SensorKinect 0.93 from https://github.com/avin2/SensorKinect/blob/unstable/Bin/SensorKinect093-Bin-Win32-v5.1.2.1.msi?raw=true (32 bit). Alternatively, for 64-bit Python, use https://github.com/avin2/SensorKinect/blob/unstable/Bin/SensorKinect093-Bin-Win64-v5.1.2.1.msi?raw=true (64 bit).

5. Download the self-extracting ZIP of OpenCV 2.4.3 from http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.3/OpenCV-2.4.3.exe/download. Run the self-extracting ZIP and, when prompted, enter a destination folder, which we will refer to as <unzip_destination>. A subfolder, <unzip_destination>\opencv, is created.

6. Open Command Prompt and make another folder where our build will go:

   ```
   > mkdir<build_folder>
   ```

   Change directory to the build folder:

   ```
   > cd <build_folder>
   ```

7. Now, we are ready to configure our build. To understand all the options, we could read the code in <unzip_destination>\opencv\CMakeLists.txt. However, for this book's purposes, we only need to use the options that will give us a release build with Python bindings and, optionally, depth camera support via OpenNI and SensorKinect.

   For Visual Studio 2010 or Visual C++ Express 2010, run:

   ```
   > cmake -D:CMAKE_BUILD_TYPE=RELEASE -D:WITH_OPENNI=ON -G "Visual
   Studio 10" <unzip_destination>\opencv
   ```

   Alternatively, for MinGW, run:

   ```
   > cmake -D:CMAKE_BUILD_TYPE=RELEASE -D:WITH_OPENNI=ON -G
   "MinGWMakefiles" <unzip_destination>\opencv
   ```

   If OpenNI is not installed, omit -D:WITH_OPENNI=ON. (In this case, depth cameras will not be supported.) If OpenNI and SensorKinect are installed to non-default locations, modify the command to include -D:OPENNI_LIB_DIR=<openni_install_destination>\Lib -D:OPENNI_INCLUDE_DIR=<openni_install_destination>\Include -D:OPENNI_PRIME_SENSOR_MODULE_BIN_DIR=<sensorkinect_install_destination>\Sensor\Bin.

CMake might report that it has failed to find some dependencies. Many of OpenCV's dependencies are optional; so, do not be too concerned yet. If the build fails to complete or you run into problems later, try installing missing dependencies (often available as prebuilt binaries) and then rebuild OpenCV from this step.

8. Having configured our build system, we are ready to compile.

   For Visual Studio or Visual C++ Express, open `<build_folder>/OpenCV.sln`. Select `Release` configuration and build. If you get build errors, double-check that `Release` configuration is selected.

   Alternatively, for MinGW, run:

   ```
   > mingw32-make.
   ```

9. Copy `<build_folder>\lib\Release\cv2.pyd` (from a Visual Studio build) or `<build_folder>\lib\cv2.pyd` (from a MinGW build) to `C:\Python2.7\Lib\site-packages` (assuming Python 2.7 is installed to the default location). Now, the Python installation can find part of OpenCV.

10. Finally, we need to make sure that Python and other processes can find the rest of OpenCV. Edit the system's `Path` variable and append `;<build_folder>/bin/Release` (for a Visual Studio build) or `;<build_folder>/bin` (for a MinGW build). Reboot your system.

Making the choice on Mac OS X Snow Leopard, Mac OS X Lion, or Mac OS X Mountain Lion

Some versions of Mac come with Python 2.7 preinstalled. However, the preinstalled Python is customized by Apple for the system's internal needs. Normally, we should not install any libraries atop Apple's Python. If we do, our libraries might break during system updates or, worse, might conflict with preinstalled libraries that the system requires. Instead, we should install standard Python 2.7 and then install our libraries atop it.

For Mac, there are several possible approaches to obtaining standard Python 2.7, NumPy, SciPy, and OpenCV. All approaches ultimately require OpenCV to be compiled from source using Xcode Developer Tools. However, depending on the approach, this task is automated for us by third-party tools in various ways. We will look at approaches using MacPorts or Homebrew. These tools can potentially do everything that CMake can do, plus they help us resolve dependencies and separate our development libraries from the system libraries.

Tip

I recommend MacPorts, especially if you want to compile OpenCV with depth camera support via OpenNI and SensorKinect. Relevant patches and build scripts, including some that I maintain, are ready-made for MacPorts. By contrast, Homebrew does not currently provide a ready-made solution for compiling OpenCV with depth camera support.

Before proceeding, let's make sure that the Xcode Developer Tools are properly set up:

1. Download and install Xcode from the Mac App Store or http://connect.apple.com/. During installation, if there is an option to install **Command Line Tools**, select it.

2. Open Xcode and accept the license agreement.

3. A final step is necessary if the installer did not give us the option to install **Command Line Tools**. Go to **Xcode | Preferences | Downloads** and click on the **Install** button next to **Command Line Tools**. Wait for the installation to finish and quit Xcode.

Now we have the required compilers for any approach.

Using MacPorts with ready-made packages

We can use the MacPorts package manager to help us set up Python 2.7, NumPy, and OpenCV. MacPorts provides Terminal commands that automate the process of downloading, compiling, and installing various pieces of **open source software** (**OSS**). MacPorts also installs dependencies as needed. For each piece of software, the dependencies and build recipe are defined in a configuration file called a **Portfile** . A MacPorts **repository** is a collection of Portfiles.

Starting from a system where Xcode and its Command Line Tools are already set up, the following steps will give us an OpenCV installation via MacPorts:

1. Download and install MacPorts from http://www.macports.org/install.php.

2. If we want support for the Kinect depth camera, we need to tell MacPorts where to download some custom Portfiles that I have written. To do so, edit /opt/local/etc/macports/sources.conf (assuming MacPorts is installed to the default location). Just above the line rsync://rsync.macports.org/release/ports/ [default], add the following line:
   http://nummist.com/opencv/ports.tar.gz

   Save the file. Now, MacPorts knows to search for Portfiles in my online repository first and, then, the default online repository.

3. Open Terminal and run the following command to update MacPorts:

   ```
   $ sudo port selfupdate
   ```

   When prompted, enter your password.

4. Now (if we are using my repository), run the following command to install OpenCV with Python 2.7 bindings and support for depth cameras including Kinect:

   ```
   $ sudo port install opencv +python27 +openni_sensorkinect
   ```

   Alternatively (with or without my repository), run the following command to install OpenCV with Python 2.7 bindings and support for depth cameras excluding Kinect:

   ```
   $ sudo port install opencv +python27 +openni
   ```

   Dependencies, including Python 2.7, NumPy, OpenNI, and (in the first example) SensorKinect, are automatically installed as well.

   By adding +python27 to the command, we are specifying that we want the opencv variant (build configuration) with Python 2.7 bindings. Similarly, +openni_sensorkinect specifies the variant with the broadest possible support for depth cameras via OpenNI and SensorKinect. You may omit +openni_sensorkinect if you do not intend to use depth cameras or you may replace it with +openni if you do intend to use OpenNI-compatible depth cameras but just not Kinect. To see the full list of available variants before installing, we can enter:

   ```
   $ port variants opencv
   ```

   Depending on our customization needs, we can add other variants to the install command. For even more flexibility, we can write our own variants (as described in the next section).

5. Also, run the following command to install SciPy:

   ```
   $ sudo port install py27-scipy
   ```

6. The Python installation's executable is named python2.7. If we want to link the default python executable to python2.7, let's also run:

   ```
   $ sudo port install python_select
   $ sudo port select python python27
   ```

Using MacPorts with your own custom packages

With a few extra steps, we can change the way that MacPorts compiles OpenCV or any other piece of software. As previously mentioned, MacPorts' build recipes are defined in configuration files called Portfiles. By creating or editing Portfiles, we can access highly configurable build tools, such as CMake, while also benefitting from MacPorts' features, such as dependency resolution.

Let's assume that we already have MacPorts installed. Now, we can configure MacPorts to use custom Portfiles that we write:

1. Create a folder somewhere to hold our custom Portfiles. We will refer to this folder as `<local_repository>`.

2. Edit the file `/opt/local/etc/macports/sources.conf` (assuming MacPorts is installed to the default location). Just above the line `rsync://rsync.macports.org/release/ports/ [default]`, add this line:
   `file://<local_repository>`

   For example, if `<local_repository>` is `/Users/Joe/Portfiles`, add:

   `file:///Users/Joe/Portfiles`

   Note the triple slashes.

   Save the file. Now, MacPorts knows to search for Portfiles in `<local_repository>` first and, then, its default online repository.

3. Open Terminal and update MacPorts to ensure that we have the latest Portfiles from the default repository:

   ```
   $ sudo port selfupdate
   ```

4. Let's copy the default repository's `opencv` Portfile as an example. We should also copy the directory structure, which determines how the package is categorized by MacPorts.
   ```
   $ mkdir <local_repository>/graphics/
   $ cp
   /opt/local/var/macports/sources/rsync.macports.org/release/ports/graphics/opencv
   <local_repository>/graphics
   ```

   Alternatively, for an example that includes Kinect support, we could download my online repository from http://nummist.com/opencv/ports.tar.gz, unzip it and copy its entire `graphics` folder into `<local_repository>`:

   ```
   $ cp <unzip_destination>/graphics <local_repository>
   ```

5. Edit `<local_repository>/graphics/opencv/Portfile`. Note that this file specifies CMake configuration flags, dependencies, and variants. For details on

Portfile editing, go to http://guide.macports.org/#development.

To see which CMake configuration flags are relevant to OpenCV, we need to look at its source code. Download the source code archive from http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download, unzip it to any location, and read <unzip_destination>/OpenCV-2.4.3/CMakeLists.txt.

After making any edits to the Portfile, save it.

6. Now, we need to generate an index file in our local repository so that MacPorts can find the new Portfile:

```
$ cd <local_repository>
$ portindex
```

7. From now on, we can treat our custom opencv just like any other MacPorts package. For example, we can install it as follows:

```
$ sudo port install opencv +python27 +openni_sensorkinect
```

Note that our local repository's Portfile takes precedence over the default repository's Portfile because of the order in which they are listed in /opt/local/etc/macports/sources.conf.

Using Homebrew with ready-made packages (no support for depth cameras)

Homebrew is another package manager that can help us. Normally, MacPorts and Homebrew should not be installed on the same machine.

Starting from a system where Xcode and its Command Line Tools are already set up, the following steps will give us an OpenCV installation via Homebrew:

1. Open Terminal and run the following command to install Homebrew:

```
$ ruby -e "$(curl -fsSkLraw.github.com/mxcl/homebrew/go)"
```

2. Unlike MacPorts, Homebrew does not automatically put its executables in PATH. To do so, create or edit the file ~/.profile and add this line at the top:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Save the file and run this command to refresh PATH:

```
$ source ~/.profile
```

Note that executables installed by Homebrew now take precedence over

executables installed by the system.

3. For Homebrew's self-diagnostic report, run:

   ```
   $ brew doctor
   ```

   Follow any troubleshooting advice it gives.

4. Now, update Homebrew:

   ```
   $ brew update
   ```

5. Run the following command to install Python 2.7:

   ```
   $ brew install python
   ```

6. Now, we can install NumPy. Homebrew's selection of Python library packages is limited so we use a separate package management tool called **pip**, which comes with Homebrew's Python:

   ```
   $ pip install numpy
   ```

7. SciPy contains some Fortran code, so we need an appropriate compiler. We can use Homebrew to install the gfortran compiler:

   ```
   $ brew install gfortran
   ```

   Now, we can install SciPy:

   ```
   $ pip install scipy
   ```

8. To install OpenCV on a 64-bit system (all new Mac hardware since late 2006), run:

   ```
   $ brew install opencv
   ```

   Alternatively, to install OpenCV on a 32-bit system, run:

   ```
   $ brew install opencv --build32
   ```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. 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.

Using Homebrew with your own custom packages

Homebrew makes it easy to edit existing package definitions:

```
$ brew edit opencv
```

The package definitions are actually scripts in the Ruby programming language. Tips on editing them can be found in the Homebrew wiki at https://github.com/mxcl/homebrew/wiki/Formula-Cookbook. A script may specify Make or CMake configuration flags, among other things.

To see which CMake configuration flags are relevant to OpenCV, we need to look at its source code. Download the source code archive from http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download, unzip it to any location, and read <unzip_destination>/OpenCV-2.4.3/CMakeLists.txt.

After making any edits to the Ruby script, save it.

The customized package can be treated as normal. For example, it can be installed as follows:

```
$ brew install opencv
```

Making the choice on Ubuntu 12.04 LTS or Ubuntu 12.10

Ubuntu comes with Python 2.7 preinstalled. The standard Ubuntu repository contains OpenCV 2.3.1 packages without support for depth cameras. Alternatively, OpenCV 2.4.3 can be built from source using CMake and GCC. When built from source, OpenCV can support depth cameras via OpenNI and SensorKinect, which are available as precompiled binaries with installation scripts.

Using the Ubuntu repository (no support for depth cameras)

We can install OpenCV 2.3.1 and its dependencies using the Apt package manager:

1. Open Terminal and run this command to update Apt:

   ```
   $ sudo apt-get update
   ```

2. Now, run these commands to install NumPy, SciPy, and OpenCV with Python bindings:

```
$ sudo apt-get install python-numpy
$ sudo apt-get install python-scipy
$ sudo apt-get install libopencv-*
$ sudo apt-get install python-opencv
```

Enter Y whenever prompted about package installation.

Equivalently, we could have used Ubuntu Software Center, which is Apt's graphical frontend.

Using CMake via a ready-made script that you may customize

Ubuntu comes with the GCC compilers preinstalled. However, we need to install the CMake build system. We also need to install or reinstall various other libraries, some of which need to be specially configured for compatibility with OpenCV. Because the dependencies are complex, I have written a script that downloads, configures, and builds OpenCV and related libraries so that the resulting OpenCV installation has support for depth cameras including Kinect:

1. Download my installation script from http://nummist.com/opencv/install_opencv_ubuntu.sh and put it in any destination, say `<script_folder>`.

2. Optionally, edit the script to customize OpenCV's build configuration. To see which CMake configuration flags are relevant to OpenCV, we need to look at its source code. Download the source code archive from http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download, unzip it to any location, and read `<unzip_destination>/OpenCV-2.4.3/CMakeLists.txt`.
After making any edits to the script, save it.

3. Open Terminal and run this command to update Apt:

   ```
   $ sudo apt-get update
   ```

4. Change directory to `<script_folder>`:

   ```
   $ cd <script_folder>
   ```

   Set the script's permissions so that it is executable:

   ```
   $ chmod +x install_opencv_ubuntu.sh
   ```

   Execute the script:

   ```
   $ ./install_opencv_ubuntu.sh
   ```

When prompted, enter your password. Enter Y whenever prompted about package installation.

5. The installation script creates a folder, `<script_folder>/opencv`, which contains downloads and built files that are temporarily used by the script. After the installation script terminates, `<script_folder>/opencv` may safely be deleted; although, first, you might want to look at OpenCV's Python samples in `<script_folder>/opencv/samples/python` and `<script_folder>/opencv/samples/python2`.

Making the choice on other Unix-like systems

The approaches for Ubuntu (as described previously) are likely to work on any Linux distribution derived from Ubuntu 12.04 LTS or Ubuntu 12.10, such as:

- Kubuntu 12.04 LTS or Kubuntu 12.10
- Xubuntu 12.04 LTS or Xubuntu 12.10
- Linux Mint 13 or Linux Mint 14

On Debian Linux and its derivatives, the Apt package manager works the same as on Ubuntu, though the available packages may differ.

On Gentoo Linux and its derivatives, the Portage package manager is similar to MacPorts (as described previously), though the available packages may differ.

On other Unix-like systems, the package manager and available packages may differ. Consult your package manager's documentation and search for any packages with `opencv` in their names. Remember that OpenCV and its Python bindings might be split into multiple packages.

Also, look for any installation notes published by the system provider, the repository maintainer, or the community. Because OpenCV uses camera drivers and media codecs, getting all of its functionality to work can be tricky on systems with poor multimedia support. Under some circumstances, system packages might need to be reconfigured or reinstalled for compatibility.

If packages are available for OpenCV, check their version number. OpenCV 2.3.1 or greater is recommended for this book's purposes. Also check whether the packages offer Python bindings and whether they offer depth camera support via OpenNI and SensorKinect. Finally, check whether anyone in the developer community has reported success or failure in using the packages.

If instead we want to do a custom build of OpenCV from source, it might be helpful to refer to the installation script for Ubuntu (discussed previously) and adapt it to the

package manager and packages that are present on another system.

Running samples

Running a few sample scripts is a good way to test that OpenCV is correctly set up. The samples are included in OpenCV's source code archive.

On Windows, we should have already downloaded and unzipped OpenCV's self-extracting ZIP. Find the samples in `<unzip_destination>/opencv/samples`.

On Unix-like systems, including Mac, download the source code archive from http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download and unzip it to any location (if we have not already done so). Find the samples in `<unzip_destination>/OpenCV-2.4.3/samples`.

Some of the sample scripts require command-line arguments. However, the following scripts (among others) should work without any arguments:

- `python/camera.py`: This displays a webcam feed (assuming a webcam is plugged in).
- `python/drawing.py`: This draws a series of shapes, like a screensaver.
- `python2/hist.py`: This displays a photo. Press *A*, *B*, *C*, *D*, or *E* to see variations of the photo, along with a corresponding histogram of color or grayscale values.
- `python2/opt_flow.py` (missing from the Ubuntu package): This displays a webcam feed with a superimposed visualization of optical flow (direction of motion). For example, slowly wave your hand at the webcam to see the effect. Press *1* or *2* for alternative visualizations.

To exit a script, press *Esc* (not the window's close button).

If we encounter the message, `ImportError: No module named cv2.cv`, then we are running the script from a Python installation that does not know anything about OpenCV. There are two possible explanations:

- Some steps in the OpenCV installation might have failed or been missed. Go back and review the steps.
- If we have multiple Python installations on the machine, we might be using the wrong Python to launch the script. For example, on Mac, it might be the case that OpenCV is installed for MacPorts Python but we are running the script with the system's Python. Go back and review the installation steps about editing the system path. Also, try launching the script manually from the command line using commands such as:

```
$ python python/camera.py
```

You can also use the following command:

```
$ python2.7 python/camera.py
```

As another possible means of selecting a different Python installation, try editing the sample script to remove #! lines. These lines might explicitly associate the script with the wrong Python installation (for our particular setup).

Finding documentation, help, and updates

OpenCV's documentation is online at http://docs.opencv.org/. The documentation includes a combined API reference for OpenCV's new C++ API, its new Python API (which is based on the C++ API), its old C API, and its old Python API (which is based on the C API). When looking up a class or function, be sure to read the section about the new Python API (cv2 module), not the old Python API (cv module).

Note

The documentation entitled **OpenCV 2.1 Python Reference** (http://opencv.willowgarage.com/documentation/python/) might show up in Google searches for OpenCV Python API. Avoid this documentation, since it is out-of-date and covers only the old (C-like) Python API.

The documentation is also available as several downloadable PDF files:

- **API reference**: http://docs.opencv.org/opencv2refman
- **Tutorials**: http://docs.opencv.org/opencv_tutorials (These tutorials use C++ code. For a Python port of the tutorials' code, see Abid Rahman K.'s repository at http://goo.gl/EPsD1.)
- **User guide (incomplete)**: http://docs.opencv.org/opencv_user

If you write code on airplanes or other places without Internet access, you will definitely want to keep offline copies of the documentation.

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/
- **Blog of David Millán Escrivá (one of this book's reviewers)**: http://blog.damiles.com/

http://blog.damiles.com/

- **Blog of Abid Rahman K. (one of this book's reviewers)**: http://www.opencvpython.blogspot.com/

- **My site for this book**: http://nummist.com/opencv/

Last, if you are an advanced user who wants to try new features, bug-fixes, and sample scripts from the latest (unstable) OpenCV source code, have a look at the project's repository at https://github.com/Itseez/opencv/.

Summary

By now, we should have an OpenCV installation that can do everything we need for the project described in this book. Depending on which approach we took, we might also have a set of tools and scripts that are usable to reconfigure and rebuild OpenCV for our future needs.

We know where to find OpenCV's Python samples. These samples cover a different range of functionality than this book's project, but they are useful as additional learning aids.

Chapter 2. Handling Files, Cameras, and GUIs

This chapter introduces OpenCV's I/O functionality. We also discuss a project concept and the beginnings of an object-oriented design for this project, which we will flesh out in subsequent chapters.

By starting with a look at I/O capabilities and design patterns, we are building our project in the same way we would make a sandwich: from the outside in. Bread slices and spread or endpoints and glue, come before fillings or algorithms. We choose this approach because computer vision is extroverted—it contemplates the real world outside our computer—and we want to apply all our subsequent, algorithmic work to the real world through a common interface.

Note

All the finished code for this chapter can be downloaded from my website: http://nummist.com/opencv/3923_02.zip.

Basic I/O scripts

All CV applications need to get images as input. Most also need to produce images as output. An interactive CV application might require a camera as an input source and a window as a output destination. However, other possible sources and destinations

window as a output destination. However, other possible sources and destinations include image files, video files, and raw bytes. For example, raw bytes might be received/sent via a network connection or might be generated by an algorithm if we are incorporating procedural graphics into our application. Let's look at each of these possibilities.

Reading/Writing an image file

OpenCV provides the `imread()` and `imwrite()` functions that support various file formats for still images. The supported formats vary by system but should always include the BMP format. Typically, PNG, JPEG, and TIFF should be among the supported formats too. Images can be loaded from one file format and saved to another. For example, let's convert an image from PNG to JPEG:

```
import cv2
```

```
image = cv2.imread('MyPic.png')
cv2.imwrite('MyPic.jpg', image)
```

Note

Most of the OpenCV functionality that we use is in the `cv2` module. You might come across other OpenCV guides that instead rely on the `cv` or `cv2.cv` modules, which are legacy versions. We do use `cv2.cv` for certain constants that are not yet redefined in `cv2`.

By default, `imread()` returns an image in BGR color format, even if the file uses a grayscale format. **BGR** (**blue-green-red**) represents the same color space as **RGB** (**red-green-blue**) but the byte order is reversed.

Optionally, we may specify the mode of `imread()` to be `CV_LOAD_IMAGE_COLOR` (BGR), `CV_LOAD_IMAGE_GRAYSCALE` (grayscale), or `CV_LOAD_IMAGE_UNCHANGED` (either BGR or grayscale, depending on the file's color space). For example, let's load a PNG as a grayscale image (losing any color information in the process) and, then, save it as a grayscale PNG image:

```
import cv2
```

```
grayImage = cv2.imread('MyPic.png', cv2.CV_LOAD_IMAGE_GRAYSCALE)
cv2.imwrite('MyPicGray.png', grayImage)
```

Regardless of the mode, `imread()` discards any alpha channel (transparency). The `imwrite()` function requires an image to be in BGR or grayscale format with a number of bits per channel that the output format can support. For example, `bmp` requires 8 bits per channel while PNG allows either 8 or 16 bits per channel.

Converting between an image and raw bytes

Conceptually, a byte is an integer ranging from 0 to 255. Throughout real-time graphics applications today, a pixel is typically represented by one byte per channel, though other representations are also possible.

An OpenCV image is a 2D or 3D array of type `numpy.array`. An 8-bit grayscale image is a 2D array containing byte values. A 24-bit BGR image is a 3D array, also containing byte values. We may access these values by using an expression like `image[0, 0]` or `image[0, 0, 0]`. The first index is the pixel's y coordinate, or row, 0 being the top. The second index is the pixel's x coordinate, or column, 0 being the leftmost. The third index (if applicable) represents a color channel.

For example, in an 8-bit grayscale image with a white pixel in the upper-left corner, `image[0, 0]` is 255. For a 24-bit BGR image with a blue pixel in the upper-left corner, `image[0, 0]` is [255, 0, 0].

Note

As an alternative to using an expression like `image[0, 0]` or `image[0, 0] = 128`, we may use an expression like `image.item((0, 0))` or `image.setitem((0, 0), 128)`. The latter expressions are more efficient for single-pixel operations. However, as we will see in subsequent chapters, we usually want to perform operations on large slices of an image rather than single pixels.

Provided that an image has 8 bits per channel, we can cast it to a standard Python `bytearray`, which is one-dimensional:

```
byteArray = bytearray(image)
```

Conversely, provided that `bytearray` contains bytes in an appropriate order, we can cast and then reshape it to get a `numpy.array` type that is an image:

```
grayImage = numpy.array(grayByteArray).reshape(height, width)
bgrImage = numpy.array(bgrByteArray).reshape(height, width, 3)
```

As a more complete example, let's convert `bytearray` containing random bytes to a grayscale image and a BGR image:

```
import cv2
import numpy
import os

# Make an array of 120,000 random bytes.
randomByteArray = bytearray(os.urandom(120000))
flatNumpyArray = numpy.array(randomByteArray)

# Convert the array to make a 400x300 grayscale image.
grayImage = flatNumpyArray.reshape(300, 400)
cv2.imwrite('RandomGray.png', grayImage)

# Convert the array to make a 400x100 color image.
bgrImage = flatNumpyArray.reshape(100, 400, 3)
```

```
cv2.imwrite('RandomColor.png', bgrImage)
```

After running this script, we should have a pair of randomly generated images, `RandomGray.png` and `RandomColor.png`, in the script's directory.

Note

Here, we use Python's standard `os.urandom()` function to generate random raw bytes, which we then convert to a Numpy array. Note that it is also possible to generate a random Numpy array directly (and more efficiently) using a statement such as `numpy.random.randint(0, 256, 120000).reshape(300, 400)`. The only reason we are using `os.urandom()` is to help demonstrate conversion from raw bytes.

Reading/Writing a video file

OpenCV provides the `VideoCapture` and `VideoWriter` classes that support various video file formats. The supported formats vary by system but should always include AVI. Via its `read()` method, a `VideoCapture` class may be polled for new frames until reaching the end of its video file. Each frame is an image in BGR format. Conversely, an image may be passed to the `write()` method of the `VideoWriter` class, which appends the image to the file in `VideoWriter`. Let's look at an example that reads frames from one AVI file and writes them to another AVI file with YUV encoding:

```
import cv2

videoCapture = cv2.VideoCapture('MyInputVid.avi')
fps = videoCapture.get(cv2.cv.CV_CAP_PROP_FPS)
size = (int(videoCapture.get(cv2.cv.CV_CAP_PROP_FRAME_WIDTH)),
        int(videoCapture.get(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)))
videoWriter = cv2.VideoWriter(
    'MyOutputVid.avi', cv2.cv.CV_FOURCC('I','4','2','0'), fps, size)

success, frame = videoCapture.read()
while success: # Loop until there are no more frames.
    videoWriter.write(frame)
    success, frame = videoCapture.read()
```

The arguments to `VideoWriter` class' constructor deserve special attention. The video's filename must be specified. Any preexisting file with that name is overwritten. A video codec must also be specified. The available codecs may vary from system to system. Options include:

- `cv2.cv.CV_FOURCC('I','4','2','0')`: This is an uncompressed YUV, 4:2:0 chroma subsampled. This encoding is widely compatible but produces large files. The file extension should be avi.

- `cv2.cv.CV_FOURCC('P','I','M','1')`: This is MPEG-1. The file extension should be avi.

should be avi.

- `cv2.cv.CV_FOURCC('M','J','P','G')`: This is motion-JPEG. The file extension should be avi.

- `cv2.cv.CV_FOURCC('T','H','E','O')`: This is Ogg-Vorbis. The file extension should be ogv.

- `cv2.cv.CV_FOURCC('F','L','V','1')`: This is Flash video. The file extension should be flv.

A frame rate and frame size must be specified, too. Since we are copying from another video, these properties can be read from our get() method of the VideoCapture class.

Capturing camera frames

A stream of camera frames is represented by the VideoCapture class, too. However, for a camera, we construct a VideoCapture class by passing the camera's device index instead of a video's filename. Let's consider an example that captures 10 seconds of video from a camera and writes it into an AVI file:

```
import cv2

cameraCapture = cv2.VideoCapture(0)
fps = 30 # an assumption
size = (int(cameraCapture.get(cv2.cv.CV_CAP_PROP_FRAME_WIDTH)),
        int(cameraCapture.get(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)))
videoWriter = cv2.VideoWriter(
    'MyOutputVid.avi', cv2.cv.CV_FOURCC('I','4','2','0'), fps, size)

success, frame = cameraCapture.read()
numFramesRemaining = 10 * fps - 1
while success and numFramesRemaining > 0:
    videoWriter.write(frame)
    success, frame = cameraCapture.read()
    numFramesRemaining -= 1
```

Unfortunately, the get() method of a VideoCapture class does not return an accurate value for the camera's frame rate; it always returns 0. For the purpose of creating an appropriate VideoWriter class for the camera, we have to either make an assumption about the frame rate (as we did in the code previously) or measure it using a timer. The latter approach is better and we will cover it later in this chapter.

The number of cameras and their ordering is of course system-dependent. Unfortunately, OpenCV does not provide any means of querying the number of cameras or their properties. If an invalid index is used to construct a VideoCapture class, the VideoCapture class will not yield any frames; its read() method will return (false, None).

The read() method is inappropriate when we need to synchronize a set of cameras or a multi-head camera (such as a stereo camera or a Kinect). Then, we use the grab() and

retrieve() methods instead. For a set of cameras:

```
success0 = cameraCapture0.grab()
success1 = cameraCapture1.grab()
if success0 and success1:
    frame0 = cameraCapture0.retrieve()
    frame1 = cameraCapture1.retrieve()
```

For a multi-head camera, we must specify a head's index as an argument to retrieve():

```
success = multiHeadCameraCapture.grab()
if success:
    frame0 = multiHeadCameraCapture.retrieve(channel = 0)
    frame1 = multiHeadCameraCapture.retrieve(channel = 1)
```

We will study multi-head cameras in more detail in Chapter 5, *Detecting Foreground/Background Regions and Depth*.

Displaying camera frames in a window

OpenCV allows named windows to be created, redrawn, and destroyed using the namedWindow(), imshow(), and destroyWindow() functions. Also, any window may capture keyboard input via the waitKey() function and mouse input via the setMouseCallback() function. Let's look at an example where we show frames of live camera input:

```
import cv2

clicked = False
def onMouse(event, x, y, flags, param):
    global clicked
    if event == cv2.cv.CV_EVENT_LBUTTONUP:
        clicked = True

cameraCapture = cv2.VideoCapture(0)
cv2.namedWindow('MyWindow')
cv2.setMouseCallback('MyWindow', onMouse)

print 'Showing camera feed. Click window or press any key to stop.'
success, frame = cameraCapture.read()
while success and cv2.waitKey(1) == -1 and not clicked:
    cv2.imshow('MyWindow', frame)
    success, frame = cameraCapture.read()

cv2.destroyWindow('MyWindow')
```

The argument to waitKey() is a number of milliseconds to wait for keyboard input. The return value is either -1 (meaning no key has been pressed) or an ASCII keycode, such as 27 for *Esc*. For a list of ASCII keycodes, see http://www.asciitable.com/. Also, note that Python provides a standard function, ord(), which can convert a character to its ASCII keycode. For example, ord('a') returns 97.

Tip

On some systems, `waitKey()` may return a value that encodes more than just the ASCII keycode. (A bug is known to occur on Linux when OpenCV uses GTK as its backend GUI library.) On all systems, we can ensure that we extract just the ASCII keycode by reading the last byte from the return value, like this:

```
keycode = cv2.waitKey(1)
if keycode != -1:
    keycode &= 0xFF
```

OpenCV's window functions and `waitKey()` are interdependent. OpenCV windows are only updated when `waitKey()` is called, and `waitKey()` only captures input when an OpenCV window has focus.

The mouse callback passed to `setMouseCallback()` should take five arguments, as seen in our code sample. The callback's `param` argument is set as an optional third argument to `setMouseCallback()`. By default, it is 0. The callback's `event` argument is one of the following:

- `cv2.cv.CV_EVENT_MOUSEMOVE`: Mouse movement
- `cv2.cv.CV_EVENT_LBUTTONDOWN`: Left button down
- `cv2.cv.CV_EVENT_RBUTTONDOWN`: Right button down
- `cv2.cv.CV_EVENT_MBUTTONDOWN`: Middle button down
- `cv2.cv.CV_EVENT_LBUTTONUP`: Left button up
- `cv2.cv.CV_EVENT_RBUTTONUP`: Right button up
- `cv2.cv.CV_EVENT_MBUTTONUP`: Middle button up
- `cv2.cv.CV_EVENT_LBUTTONDBLCLK`: Left button double-click
- `cv2.cv.CV_EVENT_RBUTTONDBLCLK`: Right button double-click
- `cv2.cv.CV_EVENT_MBUTTONDBLCLK`: Middle button double-click

The mouse callback's `flags` argument may be some bitwise combination of the following:

- `cv2.cv.CV_EVENT_FLAG_LBUTTON`: The left button pressed
- `cv2.cv.CV_EVENT_FLAG_RBUTTON`: The right button pressed
- `cv2.cv.CV_EVENT_FLAG_MBUTTON`: The middle button pressed
- `cv2.cv.CV_EVENT_FLAG_CTRLKEY`: The *Ctrl* key pressed

- `cv2.cv.CV_EVENT_FLAG_SHIFTKEY`: The *Shift* key pressed
- `cv2.cv.CV_EVENT_FLAG_ALTKEY`: The *Alt* key pressed

Unfortunately, OpenCV does not provide any means of handling window events. For example, we cannot stop our application when the window's close button is clicked. Due to OpenCV's limited event handling and GUI capabilities, many developers prefer to integrate it with another application framework. Later in this chapter, we will design an abstraction layer to help integrate OpenCV into any application framework.

Project concept

OpenCV is often studied through a cookbook approach that covers a lot of algorithms but nothing about high-level application development. To an extent, this approach is understandable because OpenCV's potential applications are so diverse. For example, we could use it in a photo/video editor, a motion-controlled game, a robot's AI, or a psychology experiment where we log participants' eye movements. Across such different use cases, can we truly study a useful set of abstractions?

I believe we can and the sooner we start creating abstractions, the better. We will structure our study of OpenCV around a single application, but, at each step, we will design a component of this application to be extensible and reusable.

We will develop an interactive application that performs face tracking and image manipulations on camera input in real time. This type of application covers a broad range of OpenCV's functionality and challenges us to create an efficient, effective implementation. Users would immediately notice flaws, such as a low frame rate or inaccurate tracking. To get the best results, we will try several approaches using conventional imaging and depth imaging.

Specifically, our application will perform real-time facial merging. Given two streams of camera input (or, optionally, prerecorded video input), the application will superimpose faces from one stream atop faces in the other. Filters and distortions will be applied to give the blended scene a unified look and feel. Users should have the experience of being engaged in a live performance where they enter another environment and another persona. This type of user experience is popular in amusement parks such as Disneyland.

We will call our application Cameo. A **cameo** is (in jewelry) a small portrait of a person or (in film) a very brief role played by a celebrity.

An object-oriented design

Python applications can be written in a purely procedural style. This is often done with small applications like our basic I/O scripts, discussed previously. However, from now on, we will use an object-oriented style because it promotes modularity and extensibility.

on, we will use an object-oriented style because it promotes modularity and extensibility.

From our overview of OpenCV's I/O functionality, we know that all images are similar, regardless of their source or destination. No matter how we obtain a stream of images or where we send it as output, we can apply the same application-specific logic to each frame in this stream. Separation of I/O code and application code becomes especially convenient in an application like Cameo, which uses multiple I/O streams.

We will create classes called `CaptureManager` and `WindowManager` as high-level interfaces to I/O streams. Our application code may use a `CaptureManager` to read new frames and, optionally, to dispatch each frame to one or more outputs, including a still image file, a video file, and a window (via a `WindowManager` class). A `WindowManager` class lets our application code handle a window and events in an object-oriented style.

Both `CaptureManager` and `WindowManager` are extensible. We could make implementations that did not rely on OpenCV for I/O. Indeed, <u>Appendix A</u>, *Integrating with Pygame* uses a `WindowManager` subclass.

Abstracting a video stream – managers.CaptureManager

As we have seen, OpenCV can capture, show, and record a stream of images from either a video file or a camera, but there are some special considerations in each case. Our `CaptureManager` class abstracts some of the differences and provides a higher-level interface for dispatching images from the capture stream to one or more outputs—a still image file, a video file, or a window.

A `CaptureManager` class is initialized with a `VideoCapture` class and has the `enterFrame()` and `exitFrame()` methods that should typically be called on every iteration of an application's main loop. Between a call to `enterFrame()` and a call to `exitFrame()`, the application may (any number of times) set a `channel` property and get a `frame` property. The `channel` property is initially 0 and only multi-head cameras use other values. The `frame` property is an image corresponding to the current channel's state when `enterFrame()` was called.

A `CaptureManager` class also has `writeImage()`, `startWritingVideo()`, and `stopWritingVideo()` methods that may be called at any time. Actual file writing is postponed until `exitFrame()`. Also during the `exitFrame()` method, the `frame` property may be shown in a window, depending on whether the application code provides a `WindowManager` class either as an argument to the constructor of `CaptureManager` or by setting a property, `previewWindowManager`.

If the application code manipulates `frame`, the manipulations are reflected in any recorded files and in the window. A `CaptureManager` class has a constructor argument and a property called `shouldMirrorPreview`, which should be `True` if we want `frame` to be mirrored (horizontally flipped) in the window but not in recorded files. Typically, when facing a camera, users prefer the live camera feed to be mirrored.

Recall that a `VideoWriter` class needs a frame rate, but OpenCV does not provide any

way to get an accurate frame rate for a camera. The `CaptureManager` class works around this limitation by using a frame counter and Python's standard `time.time()` function to estimate the frame rate if necessary. This approach is not foolproof. Depending on frame rate fluctuations and the system-dependent implementation of `time.time()`, the accuracy of the estimate might still be poor in some cases. However, if we are deploying to unknown hardware, it is better than just assuming that the user's camera has a particular frame rate.

Let's create a file called `managers.py`, which will contain our implementation of `CaptureManager`. The implementation turns out to be quite long. So, we will look at it in several pieces. First, let's add imports, a constructor, and properties, as follows:

```python
import cv2
import numpy
import time

class CaptureManager(object):

    def __init__(self, capture, previewWindowManager = None,
                 shouldMirrorPreview = False):

        self.previewWindowManager = previewWindowManager
        self.shouldMirrorPreview = shouldMirrorPreview

        self._capture = capture
        self._channel = 0
        self._enteredFrame = False
        self._frame = None
        self._imageFilename = None
        self._videoFilename = None
        self._videoEncoding = None
        self._videoWriter = None

        self._startTime = None
        self._framesElapsed = long(0)
        self._fpsEstimate = None

    @property
    def channel(self):
        return self._channel

    @channel.setter
    def channel(self, value):
        if self._channel != value:
            self._channel = value
            self._frame = None

    @property
    def frame(self):
        if self._enteredFrame and self._frame is None:
            _, self._frame = self._capture.retrieve(channel =
self.channel)
```

```
self.channel)
        return self._frame

    @property
    def isWritingImage (self):

        return self._imageFilename is not None

    @property
    def isWritingVideo(self):
        return self._videoFilename is not None
```

Note that most of the member variables are non-public, as denoted by the underscore
prefix in variable names, such as self._enteredFrame. These non-public variables
relate to the state of the current frame and any file writing operations. As previously
discussed, application code only needs to configure a few things, which are
implemented as constructor arguments and settable public properties: the camera
channel, the window manager, and the option to mirror the camera preview.

Note

By convention, in Python, variables that are prefixed with a single underscore should be
treated as **protected** (accessed only within the class and its subclasses), while variables
that are prefixed with a double underscore should be treated as **private** (accessed only
within the class).

Continuing with our implementation, let's add the enterFrame() and exitFrame()
methods to managers.py:

```
    def enterFrame(self):
        """Capture the next frame, if any."""

        # But first, check that any previous frame was exited.
        assert not self._enteredFrame, \
            'previous enterFrame() had no matching exitFrame()'

        if self._capture is not None:
            self._enteredFrame = self._capture.grab()

    def exitFrame (self):
        """Draw to the window. Write to files. Release the frame."""

        # Check whether any grabbed frame is retrievable.
        # The getter may retrieve and cache the frame.
        if self.frame is None:
            self._enteredFrame = False
            return

        # Update the FPS estimate and related variables.
        if self._framesElapsed == 0:
            self._startTime = time.time()
        else:
            timeElapsed = time.time() - self._startTime
```

```
                self._fpsEstimate =  self._framesElapsed / timeElapsed
            self._framesElapsed += 1

            # Draw to the window, if any.
            if self.previewWindowManager is not None:
                if self.shouldMirrorPreview:
                    mirroredFrame = numpy.fliplr(self._frame).copy()
                    self.previewWindowManager.show(mirroredFrame)
                else:
                    self.previewWindowManager.show(self._frame)

            # Write to the image file, if any.
            if self.isWritingImage:
                cv2.imwrite(self._imageFilename, self._frame)
                self._imageFilename = None

            # Write to the video file, if any.
            self._writeVideoFrame()

            # Release the frame.
            self._frame = None
            self._enteredFrame = False
```

Note that the implementation of `enterFrame()` only grabs (synchronizes) a frame, whereas actual retrieval from a channel is postponed to a subsequent reading of the frame variable. The implementation of `exitFrame()` takes the image from the current channel, estimates a frame rate, shows the image via the window manager (if any), and fulfills any pending requests to write the image to files.

Several other methods also pertain to file writing. To finish our class implementation, let's add the remaining file-writing methods to managers.py:

```
    def writeImage(self, filename):
        """Write the next exited frame to an image file."""
        self._imageFilename = filename

    def startWritingVideo(
            self, filename,
            encoding = cv2.cv.CV_FOURCC('I','4','2','0')):
        """Start writing exited frames to a video file."""
        self._videoFilename = filename
        self._videoEncoding = encoding

    def stopWritingVideo (self):
        """Stop writing exited frames to a video file."""
        self._videoFilename = None
        self._videoEncoding = None
        self._videoWriter = None


def _writeVideoFrame(self):

        if not self.isWritingVideo:
            return

        if self._videoWriter is None:
```

```
    if self._videoWriter is None:
        fps = self._capture.get(cv2.cv.CV_CAP_PROP_FPS)
        if fps == 0.0:
            # The capture's FPS is unknown so use an estimate.
            if self._framesElapsed < 20:
                # Wait until more frames elapse so that the
                # estimate is more stable.
                return
            else:
                fps = self._fpsEstimate
        size = (int(self._capture.get(
                    cv2.cv.CV_CAP_PROP_FRAME_WIDTH)),
                int(self._capture.get(
                    cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)))
        self._videoWriter = cv2.VideoWriter(
            self._videoFilename, self._videoEncoding,
            fps, size)

    self._videoWriter.write(self._frame)
```

The public methods, `writeImage()`, `startWritingVideo()`, and `stopWritingVideo()`, simply record the parameters for file writing operations, whereas the actual writing operations are postponed to the next call of `exitFrame()`. The non-public method, `_writeVideoFrame()`, creates or appends to a video file in a manner that should be familiar from our earlier scripts. (See the *Reading/Writing a video file* section.) However, in situations where the frame rate is unknown, we skip some frames at the start of the capture session so that we have time to build up an estimate of the frame rate.

Although our current implementation of `CaptureManager` relies on `VideoCapture`, we could make other implementations that do not use OpenCV for input. For example, we could make a subclass that was instantiated with a socket connection, whose byte stream could be parsed as a stream of images. Also, we could make a subclass that used a third-party camera library whose different hardware support than what OpenCV provides. However, for Cameo, our current implementation is sufficient.

Abstracting a window and keyboard – managers.WindowManager

As we have seen, OpenCV provides functions that cause a window to be created, be destroyed, show an image, and process events. Rather than being methods of a window class, these functions require a window's name to pass as an argument. Since this interface is not object-oriented, it is inconsistent with OpenCV's general style. Also, it is unlikely to be compatible with other window or event handling interfaces that we might eventually want to use instead of OpenCV's.

For the sake of object-orientation and adaptability, we abstract this functionality into a `WindowManager` class with the `createWindow()`, `destroyWindow()`, `show()`, and `processEvents()` methods. As a property, a `WindowManager` class has a function object called `keypressCallback`, which (if not `None`) is called from `processEvents()`

object called `keypressCallback`, which (if not `None`) is called from `processEvents()` in response to any key press. The `keypressCallback` object must take a single argument, an ASCII keycode.

Let's add the following implementation of `WindowManager` to `managers.py`:

```
class WindowManager(object):

    def __init__(self, windowName, keypressCallback = None):
        self.keypressCallback = keypressCallback

        self._windowName = windowName
        self._isWindowCreated = False

    @property
    def isWindowCreated(self):
        return self._isWindowCreated

    def createWindow (self):
        cv2.namedWindow(self._windowName)
        self._isWindowCreated = True

    def show(self, frame):
        cv2.imshow(self._windowName, frame)

    def destroyWindow (self):
        cv2.destroyWindow(self._windowName)
        self._isWindowCreated = False

    def processEvents (self):
        keycode = cv2.waitKey(1)
        if self.keypressCallback is not None and keycode != -1:
            # Discard any non-ASCII info encoded by GTK.
            keycode &= 0xFF
            self.keypressCallback(keycode)
```

Our current implementation only supports keyboard events, which will be sufficient for Cameo. However, we could modify `WindowManager` to support mouse events too. For example, the class's interface could be expanded to include a `mouseCallback` property (and optional constructor argument) but could otherwise remain the same. With some event framework other than OpenCV's, we could support additional event types in the same way, by adding callback properties.

<u>Appendix A</u>, *Integrating with Pygame*, shows a `WindowManager` subclass that is implemented with Pygame's window handling and event framework instead of OpenCV's. This implementation improves on the base `WindowManager` class by properly handling quit events—for example, when the user clicks on the window's close button. Potentially, many other event types can be handled via Pygame too.

Applying everything – cameo.Cameo

Our application is represented by a class, Cameo, with two methods: run() and onKeypress(). On initialization, a Cameo class creates a WindowManager class with onKeypress() as a callback, as well as a CaptureManager class using a camera and the WindowManager class. When run() is called, the application executes a main loop in which frames and events are processed. As a result of event processing, onKeypress() may be called. The Space bar causes a screenshot to be taken, Tab causes a screencast (a video recording) to start/stop, and Esc causes the application to quit.

In the same directory as managers.py, let's create a file called cameo.py containing the following implementation of Cameo:

```python
import cv2
from managers import WindowManager, CaptureManager

class Cameo(object):

    def __init__(self):
        self._windowManager = WindowManager('Cameo',
                                            self.onKeypress)
        self._captureManager = CaptureManager(
            cv2.VideoCapture(0), self._windowManager, True)

    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            frame = self._captureManager.frame

            # TODO: Filter the frame (Chapter 3).

            self._captureManager.exitFrame()
            self._windowManager.processEvents()

    def onKeypress (self, keycode):
        """Handle a keypress.

        space   -> Take a screenshot.
        tab     -> Start/stop recording a screencast.
        escape  -> Quit.

        """
        if keycode == 32: # space
            self._captureManager.writeImage('screenshot.png')
        elif keycode == 9: # tab
            if not self._captureManager.isWritingVideo:
                self._captureManager.startWritingVideo(
                    'screencast.avi')
            else:
                self._captureManager.stopWritingVideo()
        elif keycode == 27: # escape
            self._windowManager.destroyWindow()

if __name__=="__main__":
    Cameo().run()
```

When running the application, note that the live camera feed is mirrored, while screenshots and screencasts are not. This is the intended behavior, as we pass `True` for `shouldMirrorPreview` when initializing the `CaptureManager` class.

So far, we do not manipulate the frames in any way except to mirror them for preview. We will start to add more interesting effects in Chapter 3, *Filtering Images*.

Summary

By now, we should have an application that displays a camera feed, listens for keyboard input, and (on command) records a screenshot or screencast. We are ready to extend the application by inserting some image-filtering code (Chapter 3, *Filtering Images*) between the start and end of each frame. Optionally, we are also ready to integrate other camera drivers or other application frameworks (Appendix A, *Integrating with Pygame*), besides the ones supported by OpenCV.

Chapter 3. Filtering Images

This chapter presents some techniques for altering images. Our goal is to achieve artistic effects, similar to the filters that can be found in image editing applications, such as Photoshop or Gimp.

As we proceed with implementing filters, you can try applying them to any BGR image and then saving or displaying the result. To fully appreciate each effect, try it with various lighting conditions and subjects. By the end of this chapter, we will integrate filters into the Cameo application.

Note

All the finished code for this chapter can be downloaded from my website: http://nummist.com/opencv/3923_03.zip.

Creating modules

Like our `CaptureManager` and `WindowManager` classes, our filters should be reusable outside Cameo. Thus, we should separate the filters into their own Python module or file.

Let's create a file called `filters.py` in the same directory as `cameo.py`. We need the following import statements in `filters.py`:

```
import cv2
import numpy
import utils
```

Let's also create a file called `utils.py` in the same directory. It should contain the following import statements:

```
import cv2
import numpy
import scipy.interpolate
```

We will be adding filter functions and classes to `filters.py`, while more general-purpose math functions will go in `utils.py`.

Channel mixing – seeing in Technicolor

Channel mixing is a simple technique for remapping colors. The color at a destination pixel is a function of the color at the corresponding source pixel (only). More specifically, each channel's value at the destination pixel is a function of any or all channels' values at the source pixel. In pseudocode, for a BGR image:

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

We may define these functions however we please. Potentially, we can map a scene's colors much differently than a camera normally does or our eyes normally do.

One use of channel mixing is to simulate some other, smaller color space inside RGB or BGR. By assigning equal values to any two channels, we can collapse part of the color space and create the impression that our palette is based on just two colors of light (blended additively) or two inks (blended subtractively). This type of effect can offer nostalgic value because early color films and early digital graphics had more limited palettes than digital graphics today.

As examples, let's invent some notional color spaces that are reminiscent of Technicolor movies of the 1920s and CGA graphics of the 1980s. All of these notional color spaces can represent grays but none can represent the full color range of RGB:

- **RC (red, cyan)**: Note that red and cyan can mix to make grays. This color space resembles Technicolor Process 2 and CGA Palette 3.

- **RGV (red, green, value)**: Note that red and green cannot mix to make grays. So we need to specify value or whiteness as well. This color space resembles Technicolor Process 1.

- **CMV (cyan, magenta, value)**: Note that cyan and magenta cannot mix to make grays. So we need to specify value or whiteness as well. This color space resembles CGA Palette 1.

The following is a screenshot from *The Toll of the Sea* (1922), a movie shot in Technicolor Process 2:

The following image is from *Commander Keen: Goodbye Galaxy* (1991), a game that supports CGA Palette 1. (For color images, see the electronic edition of this book.):
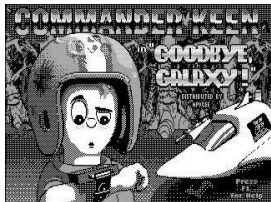
Simulating RC color space

RC color space is easy to simulate in BGR. Blue and green can mix to make cyan. By averaging the B and G channels and storing the result in both B and G, we effectively collapse these two channels into one, C. To support this effect, let's add the following function to filters.py:

```
def recolorRC(src, dst):
    """Simulate conversion from BGR to RC (red, cyan).

    The source and destination images must both be in BGR format.

    Blues and greens are replaced with cyans.
```

```
Pseudocode:
dst.b = dst.g = 0.5 * (src.b + src.g)
dst.r = src.r

"""
b, g, r = cv2.split(src)
cv2.addWeighted(b, 0.5, g, 0.5, 0, b)
cv2.merge((b, b, r), dst)
```

Three things are happening in this function:

1. Using `split()`, we extract our source image's channels as one-dimensional arrays. Having put the data in this format, we can write clear, simple channel mixing code.

2. Using `addWeighted()`, we replace the B channel's values with an average of B and G. The arguments to `addWeighted()` are (in order) the first source array, a weight applied to the first source array, the second source array, a weight applied to the second source array, a constant added to the result, and a destination array.

3. Using `merge()`, we replace the values in our destination image with the modified channels. Note that we use `b` twice as an argument because we want the destination's B and G channels to be equal.

Similar steps—splitting, modifying, and merging channels—can be applied to our other color space simulations as well.

Simulating RGV color space

RGV color space is just slightly more difficult to simulate in BGR. Our intuition might say that we should set all B-channel values to `0` because RGV cannot represent blue. However, this change would be wrong because it would discard the blue component of lightness and, thus, turn grays and pale blues into yellows. Instead, we want grays to remain gray while pale blues become gray. To achieve this result, we should reduce B values to the per-pixel minimum of B, G, and R. Let's implement this effect in `filters.py` as the following function:

```
def recolorRGV(src, dst):
    """Simulate conversion from BGR to RGV (red, green, value).

    The source and destination images must both be in BGR format.

    Blues are desaturated.

    Pseudocode:
    dst.b = min(src.b, src.g, src.r)
    dst.g = src.g
    dst.r = src.r

    """
```

```
    b, g, r = cv2.split(src)
    cv2.min(b, g, b)
    cv2.min(b, r, b)
    cv2.merge((b, g, r), dst)
```

The `min()` function computes the per-element minimums of the first two arguments and writes them to the third argument.

Simulating CMV color space

Simulating CMV color space is quite similar to simulating RGV, except that the desaturated part of the spectrum is yellow instead of blue. To desaturate yellows, we should increase B values to the per-pixel maximums of B, G, and R. Here is an implementation that we can add to `filters.py`:

```
def recolorCMV(src, dst):
    """Simulate conversion from BGR to CMV (cyan, magenta, value).

    The source and destination images must both be in BGR format.

    Yellows are desaturated.

    Pseudocode:
    dst.b = max(src.b, src.g, src.r)
    dst.g = src.g
    dst.r = src.r

    """
    b, g, r = cv2.split(src)
    cv2.max(b, g, b)
    cv2.max(b, r, b)
    cv2.merge((b, g, r), dst)
```

The `max()` function computes the per-element maximums of the first two arguments and writes them to the third argument.

By design, the three preceding effects tend to produce major color distortions, especially when the source image is colorful in the first place. If we want to craft subtle effects, channel mixing with arbitrary functions is probably not the best approach.

Curves – bending color space

Curves are another technique for remapping colors. Channel mixing and curves are similar insofar as the color at a destination pixel is a function of the color at the corresponding source pixel (only). However, in the specifics, channel mixing and curves are dissimilar approaches. With curves, a channel's value at a destination pixel is a function of (only) the same channel's value at the source pixel. Moreover, we do not define the functions directly; instead, for each function, we define a set of control points from which the function is interpolated. In pseudocode, for a BGR image:

dst.b = funcB(src.b) where funcB interpolates pointsB

dst.g = funcG(src.g) where funcG interpolates pointsG
dst.r = funcR(src.r) where funcR interpolates pointsR

The type of interpolation may vary between implementations, though it should avoid discontinuous slopes at control points and, instead, produce curves. We will use **cubic spline interpolation** whenever the number of control points is sufficient.

Formulating a curve

Our first step toward curve-based filters is to convert control points to a function. Most of this work is done for us by a SciPy function called interp1d(), which takes two arrays (x and y coordinates) and returns a function that interpolates the points. As an optional argument to interp1d(), we may specify a kind of interpolation, which, in principle, may be linear, nearest, zero, slinear (spherical linear), quadratic, or cubic, though not all options are implemented in the current version of SciPy. Another optional argument, bounds_error, may be set to False to permit extrapolation as well as interpolation.

Let's edit utils.py and add a function that wraps interp1d() with a slightly simpler interface:

```
def createCurveFunc(points):
    """Return a function derived from control points."""
    if points is None:
        return None
    numPoints = len(points)
    if numPoints < 2:
        return None
    xs, ys = zip(*points)
    if numPoints < 4:
        kind = 'linear'
        # 'quadratic' is not implemented.
    else:
        kind = 'cubic'
    return scipy.interpolate.interp1d(xs, ys, kind,
                                      bounds_error = False)
```

Rather than two separate arrays of coordinates, our function takes an array of (x, y) pairs, which is probably a more readable way of specifying control points. The array must be ordered such that x increases from one index to the next. Typically, for natural-looking effects, the y values should increase too, and the first and last control points should be (0, 0) and (255, 255) in order to preserve black and white. Note that we will treat x as a channel's input value and y as the corresponding output value. For example, (128, 160) would brighten a channel's midtones.

Note that cubic interpolation requires at least four control points. If there are only two or three control points, we fall back to linear interpolation but, for natural-looking effects, this case should be avoided.

Caching and applying a curve

Now we can get the function of a curve that interpolates arbitrary control points. However, this function might be expensive. We do not want to run it once per channel, per pixel (for example, 921,600 times per frame if applied to three channels of 640 x 480 video). Fortunately, we are typically dealing with just 256 possible input values (in 8 bits per channel) and we can cheaply precompute and store that many output values. Then, our per-channel, per-pixel cost is just a lookup of the cached output value.

Let's edit utils.py and add functions to create a lookup array for a given function and to apply the lookup array to another array (for example, an image):

```
def createLookupArray(func, length = 256):
    """Return a lookup for whole-number inputs to a function.

    The lookup values are clamped to [0, length - 1].

    """
    if func is None:
        return None
    lookupArray = numpy.empty(length)
    i = 0
    while i < length:
        func_i = func(i)
        lookupArray[i] = min(max(0, func_i), length - 1)
        i += 1
    return lookupArray

def applyLookupArray(lookupArray, src, dst):
    """Map a source to a destination using a lookup."""
    if lookupArray is None:
        return
    dst[:] = lookupArray[src]
```

Note that the approach in createLookupArray() is limited to whole-number input values, as the input value is used as an index into an array. The applyLookupArray() function works by using a source array's values as indices into the lookup array. Python's slice notation ([:]) is used to copy the looked-up values into a destination array.

Let's consider another optimization. What if we always want to apply two or more curves in succession? Performing multiple lookups is inefficient and may cause loss of precision. We can avoid this problem by combining two curve functions into one function before creating a lookup array. Let's edit utils.py again and add the following function that returns a composite of two given functions:

```
def createCompositeFunc(func0, func1):
    """Return a composite of two functions."""
    if func0 is None:
        return func1
    if func1 is None:
        return func0
    return lambda x: func0(func1(x))
```

The approach in `createCompositeFunc()` is limited to input functions that each take a single argument. The arguments must be of compatible types. Note the use of Python's `lambda` keyword to create an anonymous function.

Here is a final optimization issue. What if we want to apply the same curve to all channels of an image? Splitting and remerging channels is wasteful, in this case, because we do not need to distinguish between channels. We just need one-dimensional indexing, as used by `applyLookupArray()`. Let's edit `utils.py` to add a function that returns a one-dimensional interface to a preexisting, given array that may be multidimensional:

```
def createFlatView(array):
    """Return a 1D view of an array of any dimensionality."""
    flatView = array.view()
    flatView.shape = array.size
    return flatView
```

The return type is `numpy.view`, which has much the same interface as `numpy.array`, but `numpy.view` only owns a reference to the data, not a copy.

The approach in `createFlatView()` works for images with any number of channels. Thus, it allows us to abstract the difference between grayscale and color images in cases when we wish to treat all channels the same.

Designing object-oriented curve filters

Since we cache a lookup array for each curve, our curve-based filters have data associated with them. Thus, they need to be classes, not just functions. Let's make a pair of curve filter classes, along with corresponding higher-level classes that can apply any function, not just a curve function:

- `VFuncFilter`: This is a class that is instantiated with a function, which it can later apply to an image using `apply()`. The function is applied to the **V (value) channel** of a grayscale image or to all channels of a color image.

- `VcurveFilter`: This is a subclass of `VFuncFilter`. Instead of being instantiated with a function, it is instantiated with a set of control points, which it uses internally to create a curve function.

- `BGRFuncFilter`: This is a class that is instantiated with up to four functions, which it can later apply to a BGR image using `apply()`. One of the functions is applied to all channels and the other three functions are each applied to a single channel. The overall function is applied first and then the per-channel functions.

- `BGRCurveFilter`: this is a subclass of `BGRFuncFilter`. Instead of being instantiated with four functions, it is instantiated with four sets of control points, which it uses internally to create curve functions.

Additionally, all these classes accept a constructor argument that is a numeric type, such as `numpy.uint8` for 8 bits per channel. This type is used to determine how many entries should be in the lookup array.

Let's first look at the implementations of `VFuncFilter` and `VcurveFilter`, which may both be added to `filters.py`:

```python
class VFuncFilter(object):
    """A filter that applies a function to V (or all of BGR)."""

    def __init__(self, vFunc = None, dtype = numpy.uint8):
        length = numpy.iinfo(dtype).max + 1
        self._vLookupArray = utils.createLookupArray(vFunc, length)

    def apply(self, src, dst):
        """Apply the filter with a BGR or gray source/destination."""
        srcFlatView = utils.flatView(src)
        dstFlatView = utils.flatView(dst)
        utils.applyLookupArray(self._vLookupArray, srcFlatView,
                               dstFlatView)


class VCurveFilter(VFuncFilter):
    """A filter that applies a curve to V (or all of BGR)."""

    def __init__(self, vPoints, dtype = numpy.uint8):
        VFuncFilter.__init__(self, utils.createCurveFunc(vPoints),
                             dtype)
```

Here, we are internalizing the use of several of our previous functions: `createCurveFunc()`, `createLookupArray()`, `flatView()`, and `applyLookupArray()`. We are also using `numpy.iinfo()` to determine the relevant range of lookup values, based on the given numeric type.

Now, let's look at the implementations of `BGRFuncFilter` and `BGRCurveFilter`, which may both be added to `filters.py` as well:

```python
class BGRFuncFilter(object):
    """A filter that applies different functions to each of BGR."""

    def __init__(self, vFunc = None, bFunc = None, gFunc = None,
                 rFunc = None, dtype = numpy.uint8):
        length = numpy.iinfo(dtype).max + 1
        self._bLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(bFunc, vFunc), length)
        self._gLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(gFunc, vFunc), length)
        self._rLookupArray = utils.createLookupArray(
            utils.createCompositeFunc(rFunc, vFunc), length)

    def apply(self, src, dst):
        """Apply the filter with a BGR source/destination."""
        b, g, r = cv2.split(src)
        utils.applyLookupArray(self._bLookupArray, b, b)
        utils.applyLookupArray(self._gLookupArray, g, g)
        utils.applyLookupArray(self._rLookupArray, r, r)
```

```
        utils.applyLookupArray(self._rLookupArray, r, r)
        cv2.merge([b, g, r], dst)

class BGRCurveFilter(BGRFuncFilter):
    """A filter that applies different curves to each of BGR."""

    def __init__(self, vPoints = None, bPoints = None,
                 gPoints = None, rPoints = None, dtype =
numpy.uint8):
        BGRFuncFilter.__init__(self,
                               utils.createCurveFunc(vPoints),
                               utils.createCurveFunc(bPoints),
                               utils.createCurveFunc(gPoints),
                               utils.createCurveFunc(rPoints), dtype)
```

Again, we are internalizing the use of several of our previous functions: `createCurveFunc()`, `createCompositeFunc()`, `createLookupArray()`, and `applyLookupArray()`. We are also using `iinfo()`, `split()`, and `merge()`.

These four classes can be used as is, with custom functions or control points being passed as arguments at instantiation. Alternatively, we can make further subclasses that hard-code certain functions or control points. Such subclasses could be instantiated without any arguments.

Emulating photo films

A common use of curves is to emulate the palettes that were common in pre-digital photography. Every type of photo film has its own, unique rendition of color (or grays) but we can generalize about some of the differences from digital sensors. Film tends to suffer loss of detail and saturation in shadows, whereas digital tends to suffer these failings in highlights. Also, film tends to have uneven saturation across different parts of the spectrum. So each film has certain colors that *pop* or jump out.

Thus, when we think of good-looking film photos, we may think of scenes (or renditions) that are bright and that have certain dominant colors. At the other extreme, we may remember the murky look of underexposed film that could not be improved much by the efforts of the lab technician.

We are going to create four different film-like filters using curves. They are inspired by three kinds of film and a processing technique:

- Kodak Portra, a family of films that are optimized for portraits and weddings
- Fuji Provia, a family of general-purpose films
- Fuji Velvia, a family of films that are optimized for landscapes
- Cross-processing, a nonstandard film processing technique, sometimes used to produce a grungy look in fashion and band photography

Each film emulation effect is a very simple subclass of `BGRCurveFilter`. We just override the constructor to specify a set of control points for each channel. The choice of control points is based on recommendations by photographer Petteri Sulonen. See his article on film-like curves at http://www.prime-junta.net/pont/How_to/100_Curves_and_Films/_Curves_and_films.html.

The Portra, Provia, and Velvia effects should produce *normal-looking* images. The effect should not be obvious except in before-and-after comparisons.

Emulating Kodak Portra

Portra has a broad highlight range that tends toward warm (amber) colors, while shadows are cooler (more blue). As a portrait film, it tends to make people's complexions fairer. Also, it exaggerates certain common clothing colors, such as milky white (for example, a wedding dress) and dark blue (for example, a suit or jeans). Let's add this implementation of a Portra filter to `filters.py`:

```
class BGRPortraCurveFilter(BGRCurveFilter):
    """A filter that applies Portra-like curves to BGR."""

    def __init__(self, dtype = numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            vPoints = [(0,0),(23,20),(157,173),(255,255)],
            bPoints = [(0,0),(41,46),(231,228),(255,255)],
            gPoints = [(0,0),(52,47),(189,196),(255,255)],
            rPoints = [(0,0),(69,69),(213,218),(255,255)],
            dtype = dtype)
```

Emulating Fuji Provia

Provia has strong contrast and is slightly cool (blue) throughout most tones. Sky, water, and shade are enhanced more than sun. Let's add this implementation of a Provia filter to `filters.py`:

```
class BGRProviaCurveFilter(BGRCurveFilter):
    """A filter that applies Provia-like curves to BGR."""

    def __init__(self, dtype = numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            bPoints = [(0,0),(35,25),(205,227),(255,255)],
            gPoints = [(0,0),(27,21),(196,207),(255,255)],
            rPoints = [(0,0),(59,54),(202,210),(255,255)],
            dtype = dtype)
```

Emulating Fuji Velvia

Velvia has deep shadows and vivid colors. It can often produce azure skies in daytime and crimson clouds at sunset. The effect is difficult to emulate but here is an attempt that

we can add to filters.py:

```python
class BGRVelviaCurveFilter(BGRCurveFilter):
    """A filter that applies Velvia-like curves to BGR."""

    def __init__(self, dtype = numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            vPoints = [(0,0),(128,118),(221,215),(255,255)],
            bPoints = [(0,0),(25,21),(122,153),(165,206),(255,255)],
            gPoints = [(0,0),(25,21),(95,102),(181,208),(255,255)],
            rPoints = [(0,0),(41,28),(183,209),(255,255)],
            dtype = dtype)
```

Emulating cross-processing

Cross-processing produces a strong, blue or greenish-blue tint in shadows and a strong, yellow or greenish-yellow in highlights. Black and white are not necessarily preserved. Also, contrast is very high. Cross-processed photos have on a sickly appearance. People look jaundiced, while inanimate objects look stained. Let's edit filters.py to add the following implementation of a cross-processing filter:

```python
class BGRCrossProcessCurveFilter(BGRCurveFilter):
    """A filter that applies cross-process-like curves to BGR."""

    def __init__(self, dtype = numpy.uint8):
        BGRCurveFilter.__init__(
            self,
            bPoints = [(0,20),(255,235)],
            gPoints = [(0,0),(56,39),(208,226),(255,255)],
            rPoints = [(0,0),(56,22),(211,255),(255,255)],
            dtype = dtype)
```

Highlighting edges

Edges play a major role in both human and computer vision. We, as humans, can easily recognize many object types and their pose just by seeing a backlit silhouette or a rough sketch. Indeed, when art emphasizes edges and pose, it often seems to convey the idea of an archetype, like Rodin's *The Thinker* or Joe Shuster's *Superman*. Software, too, can reason about edges, poses, and archetypes. We will discuss these kinds of reasoning in later chapters.

For the moment, we are interested in a simple use of edges for artistic effect. We are going to trace an image's edges with bold, black lines. The effect should be reminiscent of a comic book or other illustration, drawn with a felt pen.

OpenCV provides many edge-finding filters, including Laplacian(), Sobel(), and Scharr(). These filters are supposed to turn non-edge regions to black while turning edge regions to white or saturated colors. However, they are prone to misidentifying noise as edges. This flaw can be mitigated by blurring an image before trying to find its edges. OpenCV also provides many blurring filters, including blur() (simple average),

edges. OpenCV also provides many blurring filters, including `blur()` (simple average), `medianBlur()`, and `GaussianBlur()`. The arguments to the edge-finding and blurring filters vary but always include `ksize`, an odd whole number that represents the width and height (in pixels) of the filter's kernel.

Note

A **kernel** is a set of weights that are applied to a region in the source image to generate a single pixel in the destination image. For example, a `ksize` of 7 implies that 49 (7 x 7) source pixels are considered in generating each destination pixel. We can think of a kernel as a piece of frosted glass moving over the source image and letting through a diffused blend of the source's light.

For blurring, let's use `medianBlur()`, which is effective in removing digital video noise, especially in color images. For edge-finding, let's use `Laplacian()`, which produces bold edge lines, especially in grayscale images. After applying `medianBlur()`, but before applying `Laplacian()`, we should convert from BGR to grayscale.

Once we have the result of `Laplacian()`, we can invert it to get black edges on a white background. Then, we can normalize it (so that its values range from 0 to 1) and multiply it with the source image to darken the edges. Let's implement this approach in `filters.py`:

```
def strokeEdges(src, dst, blurKsize = 7, edgeKsize = 5):
    if blurKsize >= 3:
        blurredSrc = cv2.medianBlur(src, blurKsize)
        graySrc = cv2.cvtColor(blurredSrc, cv2.COLOR_BGR2GRAY)
    else:
        graySrc = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
    cv2.Laplacian(graySrc, cv2.cv.CV_8U, graySrc, ksize = edgeKsize)
    normalizedInverseAlpha = (1.0 / 255) * (255 - graySrc)
    channels = cv2.split(src)
    for channel in channels:
        channel[:] = channel * normalizedInverseAlpha
    cv2.merge(channels, dst)
```

Note that we allow kernel sizes to be specified as arguments to `strokeEdges()`. The `blurKsize` argument is used as `ksize` for `medianBlur()`, while `edgeKsize` is used as `ksize` for `Laplacian()`. With my webcams, I find that a `blurKsize` value of 7 and `edgeKsize` value of 5 look best. Unfortunately, `medianBlur()` is expensive with a large `ksize` like 7. If you encounter performance problems when running `strokeEdges()`, try decreasing the `blurKsize` value. To turn off blur, set it to a value less than 3.

Custom kernels – getting convoluted

As we have just seen, many of OpenCV's predefined filters use a kernel. Remember that a kernel is a set of weights, which determine how each output pixel is calculated from a neighborhood of input pixels. Another term for a kernel is a **convolution matrix**. It

neighborhood of input pixels. Another term for a kernel is a **convolution matrix**. It mixes up or *convolutes* the pixels in a region. Similarly, a kernel-based filter may be called a convolution filter.

OpenCV provides a very versatile function, filter2D(), which applies any kernel or convolution matrix that we specify. To understand how to use this function, let's first learn the format of a convolution matrix. It is a 2D array with an odd number of rows and columns. The central element corresponds to a pixel of interest and the other elements correspond to that pixel's neighbors. Each element contains an integer or floating point value, which is a weight that gets applied to an input pixel's value. Consider this example:

```
kernel = numpy.array([[-1, -1, -1],
                      [-1,  9, -1],
                      [-1, -1, -1]])
```

Here, the pixel of interest has a weight of 9 and its immediate neighbors each have a weight of -1. For the pixel of interest, the output color will be nine times its input color, minus the input colors of all eight adjacent pixels. If the pixel of interest was already a bit different from its neighbors, this difference becomes intensified. The effect is that the image looks *sharper* as the contrast between neighbors is increased.

Continuing our example, we can apply this convolution matrix to a source image and destination image as follows:

```
cv2.filter2D(src, -1, kernel, dst)
```

The second argument specifies the per-channel depth of the destination image (such as cv2.CV_8U for 8 bits per channel). A negative value (as used here) means that the destination image has the same depth as the source image.

Note

For color images, note that filter2D() applies the kernel equally to each channel. To use different kernels on different channels, we would also have to use the split() and merge() functions, as we did in our earlier channel mixing functions. (See the section *Simulating RC color space*.)

Based on this simple example, let's add two classes to filters.py. One class, VConvolutionFilter, will represent a convolution filter in general. A subclass, SharpenFilter, will represent our sharpening filter specifically. Let's edit filters.py to implement these two new classes as follows:

```
class VConvolutionFilter(object):
    """A filter that applies a convolution to V (or all of BGR)."""

    def __init__(self, kernel):
        self._kernel = kernel

    def apply(self, src, dst):
        """Apply the filter with a BGR or grey source/destination."""
```

```
        """Apply the filter with a BGR or gray source/destination."""
        cv2.filter2D(src, -1, self._kernel, dst)

class SharpenFilter(VConvolutionFilter):
    """A sharpen filter with a 1-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[-1, -1, -1],
                              [-1,  9, -1],
                              [-1, -1, -1]])
        VConvolutionFilter.__init__(self, kernel)
```

The pattern is very similar to the VCurveFilter class and its subclasses. (See the section *Designing object-oriented curve filters*.)

Note that the weights sum to 1. This should be the case whenever we want to leave the image's overall brightness unchanged. If we modify a sharpening kernel slightly, so that its weights sum to 0 instead, then we have an edge detection kernel that turns edges white and non-edges black. For example, let's add the following edge detection filter to filters.py:

```
class FindEdgesFilter(VConvolutionFilter):
    """An edge-finding filter with a 1-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[-1, -1, -1],
                              [-1,  8, -1],
                              [-1, -1, -1]])
        VConvolutionFilter.__init__(self, kernel)
```

Next, let's make a blur filter. Generally, for a blur effect, the weights should sum to 1 and should be positive throughout the neighborhood. For example, we can take a simple average of the neighborhood, as follows:

```
class BlurFilter(VConvolutionFilter):
    """A blur filter with a 2-pixel radius."""

    def __init__(self):
        kernel = numpy.array([[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]])
        VConvolutionFilter.__init__(self, kernel)
```

Our sharpening, edge detection, and blur filters use kernels that are highly symmetric. Sometimes, though, kernels with less symmetry produce an interesting effect. Let's consider a kernel that blurs on one side (with positive weights) and sharpens on the other (with negative weights). It will produce a ridged or *embossed* effect. Here is an implementation that we can add to filters.py:

```
class EmbossFilter(VConvolutionFilter):
    """An emboss filter with a 1-pixel radius."""

    def __init__(self):
```

```
        kernel = numpy.array([[-2, -1, 0],
                              [-1,  1, 1],
                              [ 0,  1, 2]])
        VConvolutionFilter.__init__(self, kernel)
```

This set of custom convolution filters is very basic. Indeed, it is more basic than OpenCV's ready-made set of filters. However, with a bit of experimentation, you should be able to write your own kernels that produce a unique look.

Modifying the application

Now that we have high-level functions and classes for several filters, it is trivial to apply any of them to the captured frames in Cameo. Let's edit `cameo.py` and add the lines that appear in bold face in the following excerpt:

```
import cv2
import filters
from managers import WindowManager, CaptureManager

class Cameo(object):

    def __init__(self):
        self._windowManager = WindowManager('Cameo',
                                            self.onKeypress)
        self._captureManager = CaptureManager(
            cv2.VideoCapture(0), self._windowManager, True)
        self._curveFilter = filters.BGRPortraCurveFilter()

    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            frame = self._captureManager.frame

            # TODO: Track faces (Chapter 3).

            filters.strokeEdges(frame, frame)
            self._curveFilter.apply(frame, frame)

            self._captureManager.exitFrame()
            self._windowManager.processEvents()

    # ... The rest is the same as in Chapter 2.
```

Here, I have chosen to apply two effects: stroking the edges and emulating Portra film colors. Feel free to modify the code to apply any filters you like.

Here is a screenshot from Cameo, with stroked edges and Portra-like colors:
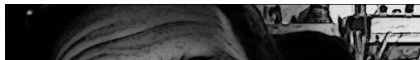
Summary

At this point, we should have an application that displays a filtered camera feed. We should also have several more filter implementations that are easily swappable with the ones we are currently using. Now, we are ready to proceed with analyzing each frame for the sake of finding faces to manipulate in the next chapter.

Chapter 4. Tracking Faces with Haar Cascades

This chapter introduces some of OpenCV's tracking functionality, along with the data files that define particular types of trackable objects. Specifically, we look at Haar cascade classifiers, which analyze contrast between adjacent image regions to determine whether or not a given image or subimage matches a known type. We consider how to combine multiple Haar cascade classifiers in a hierarchy, such that one classifier identifies a parent region (for our purposes, a face) and other classifiers identify child regions (eyes, nose, and mouth).

We also take a detour into the humble but important subject of rectangles. By drawing, copying, and resizing rectangular image regions, we can perform simple manipulations on image regions that we are tracking.

By the end of this chapter, we will integrate face tracking and rectangle manipulations into Cameo. Finally, we'll have some face-to-face interaction!

Note

All the finished code for this chapter can be downloaded from my website:

Conceptualizing Haar cascades

When we talk about classifying objects and tracking their location, what exactly are we hoping to pinpoint? What constitutes a recognizable part of an object?

Photographic images, even from a webcam, may contain a lot of detail for our (human) viewing pleasure. However, image detail tends to be unstable with respect to variations in lighting, viewing angle, viewing distance, camera shake, and digital noise. Moreover, even real differences in physical detail might not interest us for the purpose of classification. I was taught in school, that no two snowflakes look alike under a microscope. Fortunately, as a Canadian child, I had already learned how to recognize snowflakes without a microscope, as the similarities are more obvious in bulk.

Thus, some means of abstracting image detail is useful in producing stable classification and tracking results. The abstractions are called **features**, which are said to be **extracted** from the image data. There should be far fewer features than pixels, though any pixel might influence multiple features. The level of similarity between two images can be evaluated based on distances between the images' corresponding features. For example, distance might be defined in terms of spatial coordinates or color coordinates. Haar-like features are one type of feature that is often applied to real-time face tracking. They were first used for this purpose by Paul Viola and Michael Jones in 2001. Each Haar-like feature describes the pattern of contrast among adjacent image regions. For example, edges, vertices, and thin lines each generate distinctive features. For any given image, the features may vary depending on the regions' size, which may be called the **window size**. Two images that differ only in scale should be capable of yielding similar features, albeit for different window sizes. Thus, it is useful to generate features for multiple window sizes. Such a collection of features is called a **cascade**. We may say a Haar cascade is scale-invariant or, in other words, robust to changes in scale. OpenCV provides a classifier and tracker for scale-invariant Haar cascades, which it expects to be in a certain file format. Haar cascades, as implemented in OpenCV, are not robust to changes in rotation. For example, an upside-down face is not considered similar to an upright face and a face viewed in profile is not considered similar to a face viewed from the front. A more complex and more resource-intensive implementation could improve Haar cascades' robustness to rotation by considering multiple transformations of images as well as multiple window sizes. However, we will confine ourselves to the implementation in OpenCV.

Getting Haar cascade data

As part of your OpenCV setup, you probably have a directory called `haarcascades`. It contains cascades that are trained for certain subjects using tools that come with OpenCV. The directory's full path depends on your system and method of setting up OpenCV, as follows:

OpenCV, as follows:

- **Build from source archive**: `<unzip_destination>/data/haarcascades`
- **Windows with self-extracting ZIP**: `<unzip_destination>/data/haarcascades`
- **Mac with MacPorts**: `/opt/local/share/OpenCV/haarcascades`
- **Mac with Homebrew**: The `haarcascades` file is not included; to get it, download the source archive
- **Ubuntu with apt or Software Center**: The `haarcascades` file is not included; to get it, download the source archive

---

Tip

If you cannot find `haarcascades`, then download the source archive from http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download (or the Windows self-extracting ZIP from http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.3/OpenCV-2.4.3.exe/download), unzip it, and look for `<unzip_destination>/data/haarcascades`.

Once you find `haarcascades`, create a directory called `cascades` in the same folder as `cameo.py` and copy the following files from `haarcascades` into `cascades`:

```
haarcascade_frontalface_alt.xml
haarcascade_eye.xml
haarcascade_mcs_nose.xml
haarcascade_mcs_mouth.xml
```

As their names suggest, these cascades are for tracking faces, eyes, noses, and mouths. They require a frontal, upright view of the subject. We will use them later when building a high-level tracker. If you are curious about how these data sets are generated, refer to *Appendix B*, *Generating Haar Cascades for Custom Targets*. With a lot of patience and a powerful computer, you can make your own cascades, trained for various types of objects.

Creating modules

We should continue to maintain good separation between application-specific code and reusable code. Let's make new modules for tracking classes and their helpers.

A file called `trackers.py` should be created in the same directory as `cameo.py` (and, equivalently, in the parent directory of `cascades`). Let's put the following import statements at the start of `trackers.py`:

```
import cv2
import rects
import utils
```

Alongside `trackers.py` and `cameo.py`, let's make another file called `rects.py` containing the following import statement:

```
import cv2
```

Our face tracker and a definition of a face will go in `trackers.py`, while various helpers will go in `rects.py` and our preexisting `utils.py` file.

Defining a face as a hierarchy of rectangles

Before we start implementing a high-level tracker, we should define the type of tracking result that we want to get. For many applications, it is important to estimate how objects are posed in real, 3D space. However, our application is about image manipulation. So we care more about 2D image space. An upright, frontal view of a face should occupy a roughly rectangular region in the image. Within such a region, eyes, a nose, and a mouth should occupy rough rectangular subregions. Let's open `trackers.py` and add a class containing the relevant data:

```
class Face(object):
    """Data on facial features: face, eyes, nose, mouth."""

    def __init__(self):
        self.faceRect = None
        self.leftEyeRect = None
        self.rightEyeRect = None
        self.noseRect = None
        self.mouthRect = None
```

Note

Whenever our code contains a rectangle as a property or a function argument, we will assume it is in the format (x, y, w, h) where the unit is pixels, the upper-left corner is at (x, y), and the lower-right corner at (x+w, y+h). OpenCV sometimes uses a compatible representation but not always. So we must be careful when sending/receiving rectangles to/from OpenCV. For example, sometimes OpenCV requires the upper-left and lower-right corners as coordinate pairs.

Tracing, cutting, and pasting rectangles

When I was in primary school, I was poor at crafts. I often had to take my unfinished craft projects home, where my mother volunteered to finish them for me so that I could spend more time on the computer instead. I shall never cut and paste a sheet lof paper, nor an array of bytes, without thinking of those days.

Just as in crafts, mistakes in our graphics program are easier to see if we first draw outlines. For debugging purposes, Cameo will include an option to draw lines around any rectangles represented by a Face. OpenCV provides a rectangle() function for drawing. However, its arguments represent a rectangle differently than Face does. For convenience, let's add the following wrapper of rectangle() to rects.py:

```
def outlineRect(image, rect, color):
    if rect is None:
        return
    x, y, w, h = rect
    cv2.rectangle(image, (x, y), (x+w, y+h), color)
```

Here, color should normally be either a BGR triplet (of values ranging from 0 to 255) or a grayscale value (ranging from 0 to 255), depending on the image's format.

Next, Cameo must support copying one rectangle's contents into another rectangle. We can read or write a rectangle within an image by using Python's slice notation. Remembering that an image's first index is the y coordinate or row, we can specify a rectangle as image[y:y+h, x:x+w]. For copying, a complication arises if the source and destination of rectangles are of different sizes. Certainly, we expect two faces to appear at different sizes, so we must address this case. OpenCV provides a resize() function that allows us to specify a destination size and an interpolation method. Combining slicing and resizing, we can add the following implementation of a copy function to rects.py:

```
def copyRect(src, dst, srcRect, dstRect,
             interpolation = cv2.INTER_LINEAR):
    """Copy part of the source to part of the destination."""

    x0, y0, w0, h0 = srcRect
    x1, y1, w1, h1 = dstRect

    # Resize the contents of the source sub-rectangle.
    # Put the result in the destination sub-rectangle.
    dst[y1:y1+h1, x1:x1+w1] = \
        cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                   interpolation = interpolation)
```

OpenCV supports the following options for interpolation:

- cv2.INTER_NEAREST: This is nearest-neighbor interpolation, which is cheap but produces blocky results

- cv2.INTER_LINEAR: This is bilinear interpolation (the default), which offers a good compromise between cost and quality in real-time applications

- cv2.INTER_AREA: This is pixel area relation, which may offer a better compromise between cost and quality when downscaling but produces blocky results when upscaling

- cv2.INTER_CUBIC: This is bicubic interpolation over a 4 x 4 pixel neighborhood, a high-cost, high-quality approach

a high-cost, high-quality approach

- `cv2.INTER_LANCZOS4`: This is Lanczos interpolation over an 8 x 8 pixel neighborhood, the highest-cost, highest-quality approach

Copying becomes more complicated if we want to support swapping of two or more rectangles' contents. Consider the following approach, which is wrong:

```
copyRect(image, image, rect0, rect1) # overwrite rect1
copyRect(image, image, rect1, rect0) # copy from rect1
# Oops! rect1 was already overwritten by the time we copied from it!
```

Instead, we need to copy one of the rectangles to a temporary array before overwriting anything. Let's edit `rects.py` to add the following function, which swaps the contents of two or more rectangles in a single source image:

```
def swapRects(src, dst, rects,
              interpolation = cv2.INTER_LINEAR):
    """Copy the source with two or more sub-rectangles swapped."""

    if dst is not src:
        dst[:] = src

    numRects = len(rects)
    if numRects < 2:
        return

    # Copy the contents of the last rectangle into temporary storage.
    x, y, w, h = rects[numRects - 1]
    temp = src[y:y+h, x:x+w].copy()

    # Copy the contents of each rectangle into the next.
    i = numRects - 2
    while i >= 0:
        copyRect(src, dst, rects[i], rects[i+1], interpolation)
        i -= 1

    # Copy the temporarily stored content into the first rectangle.
    copyRect(temp, dst, (0, 0, w, h), rects[0], interpolation)
```

The swap is circular, such that it can support any number of rectangles. Each rectangle's content is destined for the next rectangle, except that the last rectangle's content is destined for the first rectangle.

This approach should serve us well enough for Cameo, but it is still not entirely foolproof. Intuition might tell us that the following code should leave `image` unchanged:

```
swapRects(image, image, rect0, rect1)
swapRects(image, image, rect1, rect0)
```

However, if `rect0` and `rect1` overlap, our intuition may be incorrect. If you see strange-looking results, then investigate the possibility that you are swapping overlapping rectangles.

Adding more utility functions

Last chapter, we created a module called `utils` for some miscellaneous helper functions. A couple of extra helper functions will make it easier for us to write a tracker.

First, it may be useful to know whether an image is in grayscale or color. We can tell based on the dimensionality of the image. Color images are 3D arrays, while grayscale images have fewer dimensions. Let's add the following function to `utils.py` to test whether an image is in grayscale:

```
def isGray(image):
    """Return True if the image has one channel per pixel."""
    return image.ndim < 3
```

Second, it may be useful to know an image's dimensions and to divide these dimensions by a given factor. An image's (or other array's) height and width, respectively, are the first two entries in its `shape` property. Let's add the following function to `utils.py` to get an image's dimensions, divided by a value:

```
def widthHeightDividedBy(image, divisor):
    """Return an image's dimensions, divided by a value."""
    h, w = image.shape[:2]
    return (w/divisor, h/divisor)
```

Now, let's get back on track with this chapter's main subject, tracking.

Tracking faces

The challenge in using OpenCV's Haar cascade classifiers is not just getting a tracking result; it is getting a series of sensible tracking results at a high frame rate. One kind of common sense that we can enforce is that certain tracked objects should have a hierarchical relationship, one being located relative to the other. For example, a nose should be in the middle of a face. By attempting to track both a whole face and parts of a face, we can enable application code to do more detailed manipulations and to check how good a given tracking result is. A face with a nose is a better result than one without. At the same time, we can support some optimizations, such as only looking for faces of a certain size and noses in certain places.

We are going to implement an optimized, hierarchical tracker in a class called `FaceTracker`, which offers a simple interface. A `FaceTracker` may be initialized with certain optional configuration arguments that are relevant to the tradeoff between tracking accuracy and performance. At any given time, the latest tracking results of `FaceTracker` are stored in a property called `faces`, which is a list of `Face` instances. Initially, this list is empty. It is refreshed via an `update()` method that accepts an image for the tracker to analyze. Finally, for debugging purposes, the rectangles of `faces` may be drawn via a `drawDebugRects()` method, which accepts an image as a drawing surface. Every frame, a real-time face-tracking application would call `update()`, read `faces`, and perhaps call `drawDebugRects()`.

Internally, FaceTracker uses an OpenCV class called `CascadeClassifier`. A `CascadeClassifier` is initialized with a cascade data file, such as the ones that were found and copied earlier. For our purposes, the important method of `CascadeClassifier` is `detectMultiScale()`, which performs tracking that may be robust to variations in scale. The possible arguments to `detectMultiScale()` are:

- `image`: This is an image to be analyzed. It must have 8 bits per channel.

- `scaleFactor`: This scaling factor separates the window sizes in two successive passes. A higher value improves performance but diminishes robustness with respect to variations in scale.

- `minNeighbors`: This value is one less than the minimum number of regions that are required in a match. (A match may merge multiple neighboring regions.)

- `flags`: There are several flags but not all combinations are valid. The valid standalone flags and valid combinations include:
    - `cv2.cv.CV_HAAR_SCALE_IMAGE`: Scales each windowed image region to match the feature data. (The default approach is the opposite: scale the feature data to match the window.) Scaling the image allows for certain optimizations on modern hardware. This flag must not be combined with others.

    - `cv2.cv.CV_HAAR_DO_CANNY_PRUNING`: Eagerly rejects regions that contain too many or too few edges to match the object type. This flag should not be combined with `cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT`.

    - `cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT`: Accepts, at most, one match (the biggest).

    - `cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT | cv2.cv.CV_HAAR_DO_ROUGH SEARCH`: Accepts, at most, one match (the biggest) and skips some steps that would refine (shrink) the region of this match. The `minNeighbors` argument should be greater than 0.

- `minSize`: A pair of pixel dimensions representing the minimum object size being sought. A higher value improves performance.

- `maxSize`: A pair of pixel dimensions representing the maximum object size being sought. A lower value improves performance.

The return value of `detectMultiScale()` is a list of matches, each expressed as a rectangle in the format [x, y, w, h].

Similarly, the initializer of FaceTracker accepts `scaleFactor`, `minNeighbors`, and `flags` as arguments. The given values are passed to all `detectMultiScale()` calls that a FaceTracker makes internally. Also during initialization, a FaceTracker creates CascadeClassifiers using face, eye, nose, and mouth data. Let's add the following

cascadeclassifier's using face, eye, nose, and mouth data. Let's add the following
implementation of the initializer and the faces property to trackers.py:

```
class FaceTracker(object):
    """A tracker for facial features: face, eyes, nose, mouth."""

    def __init__(self, scaleFactor = 1.2, minNeighbors = 2,
                 flags = cv2.cv.CV_HAAR_SCALE_IMAGE):

        self.scaleFactor = scaleFactor
        self.minNeighbors = minNeighbors
        self.flags = flags

        self._faces = []

        self._faceClassifier = cv2.CascadeClassifier(
            'cascades/haarcascade_frontalface_alt.xml')
        self._eyeClassifier = cv2.CascadeClassifier(
            'cascades/haarcascade_eye.xml')
        self._noseClassifier = cv2.CascadeClassifier(
            'cascades/haarcascade_mcs_nose.xml')
        self._mouthClassifier = cv2.CascadeClassifier(
            'cascades/haarcascade_mcs_mouth.xml')

    @property
    def faces(self):
        """The tracked facial features."""
        return self._faces
```

The update() method of FaceTracker first creates an equalized, grayscale variant of
the given image. Equalization, as implemented in OpenCV's equalizeHist() function,
normalizes an image's brightness and increases its contrast. Equalization as a
preprocessing step makes our tracker more robust to variations in lighting, while
conversion to grayscale improves performance. Next, we feed the preprocessed image to
our face classifier. For each matching rectangle, we search certain subregions for a left
and right eye, nose, and mouth. Ultimately, the matching rectangles and subrectangles
are stored in Face instances in faces. For each type of tracking, we specify a minimum
object size that is proportional to the image size. Our implementation of FaceTracker
should continue with the following code for update():

```
    def update(self, image):
        """Update the tracked facial features."""

        self._faces = []

        if utils.isGray(image):
            image = cv2.equalizeHist(image)
        else:
            image = cv2.cvtColor(image, cv2.cv.CV_BGR2GRAY)
            cv2.equalizeHist(image, image)

        minSize = utils.widthHeightDividedBy(image, 8)

        faceRects = self._faceClassifier.detectMultiScale(
            image, self.scaleFactor, self.minNeighbors, self.flags,
```

```
            minSize)

        if faceRects is not None:
            for faceRect in faceRects:

                face = Face()
                face.faceRect = faceRect

                x, y, w, h = faceRect

                # Seek an eye in the upper-left part of the face.
                searchRect = (x+w/7, y, w*2/7, h/2)
                face.leftEyeRect = self._detectOneObject(
                    self._eyeClassifier, image, searchRect, 64)

                # Seek an eye in the upper-right part of the face.
                searchRect = (x+w*4/7, y, w*2/7, h/2)
                face.rightEyeRect = self._detectOneObject(
                    self._eyeClassifier, image, searchRect, 64)

                # Seek a nose in the middle part of the face.
                searchRect = (x+w/4, y+h/4, w/2, h/2)
                face.noseRect = self._detectOneObject(
                    self._noseClassifier, image, searchRect, 32)

                # Seek a mouth in the lower-middle part of the face.
                searchRect = (x+w/6, y+h*2/3, w*2/3, h/3)
                face.mouthRect = self._detectOneObject(
                    self._mouthClassifier, image, searchRect, 16)

                self._faces.append(face)
```

Note that update() relies on utils.isGray() and utils.widthHeightDividedBy(),
both implemented earlier in this chapter. Also, it relies on a private helper method,
_detectOneObject(), which is called several times in order to handle the repetitious
work of tracking several subparts of the face. As arguments, _detectOneObject()
requires a classifier, image, rectangle, and minimum object size. The rectangle is the
image subregion that the given classifier should search. For example, the nose classifier
should search the middle of the face. Limiting the search area improves performance and
helps eliminate false positives. Internally, _detectOneObject() works by running the
classifier on a slice of the image and returning the first match (or None if there are no
matches). This approach works whether or not we are using the
cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT flag. Our implementation of FaceTracker
should continue with the following code for _detectOneObject():

```
    def _detectOneObject(self, classifier, image, rect,
                         imageSizeToMinSizeRatio):

        x, y, w, h = rect

        minSize = utils.widthHeightDividedBy(
            image, imageSizeToMinSizeRatio)

        subImage = image[y:y+h, x:x+w]
```

```
        subRects = classifier.detectMultiScale(
            subImage, self.scaleFactor, self.minNeighbors,
            self.flags, minSize)

        if len(subRects) == 0:
            return None

        subX, subY, subW, subH = subRects[0]
        return (x+subX, y+subY, subW, subH)
```

Lastly, `FaceTracker` should offer basic drawing functionality so that its tracking results can be displayed for debugging purposes. The following implementation simply defines colors, iterates over `Face` instances, and draws rectangles of each `Face` to a given image using our `rects.outlineRect()` function:

```
def drawDebugRects(self, image):
    """Draw rectangles around the tracked facial features."""

    if utils.isGray(image):
        faceColor = 255
        leftEyeColor = 255
        rightEyeColor = 255
        noseColor = 255
        mouthColor = 255
    else:
        faceColor = (255, 255, 255) # white
        leftEyeColor = (0, 0, 255) # red
        rightEyeColor = (0, 255, 255) # yellow
        noseColor = (0, 255, 0) # green
        mouthColor = (255, 0, 0) # blue

    for face in self.faces:
        rects.outlineRect(image, face.faceRect, faceColor)
        rects.outlineRect(image, face.leftEyeRect, leftEyeColor)
        rects.outlineRect(image, face.rightEyeRect,
                          rightEyeColor)
        rects.outlineRect(image, face.noseRect, noseColor)
        rects.outlineRect(image, face.mouthRect, mouthColor)
```

Now, we have a high-level tracker that hides the details of Haar cascade classifiers while allowing application code to supply new images, fetch data about tracking results, and ask for debug drawing.

Modifying the application

Let's look at two approaches to integrating face tracking and swapping into Cameo. The first approach uses a single camera feed and swaps face rectangles found within this camera feed. The second approach uses two camera feeds and copies face rectangles from one camera feed to the other.

For now, we will limit ourselves to manipulating faces as a whole and not subelements such as eyes. However, you could modify the code to swap only eyes, for example. If

you try this, be careful to check that the relevant subrectangles of the face are not `None`.

Swapping faces in one camera feed

For the single-camera version, the modifications are quite straightforward. On initialization of `Cameo`, we create a `FaceTracker` and a Boolean variable indicating whether debug rectangles should be drawn for the `FaceTracker`. The Boolean is toggled in `onKeypress()` in response to the *X* key. As part of the main loop in `run()`, we update our `FaceTracker` with the current that. Then, the resulting `FaceFace` objects (in the `faces` property) are fetched and their `faceRects` are swapped using `rects.swapRects()`. Also, depending on the Boolean value, we may draw debug rectangles that reflect the original positions of facial elements before any swap.

```
import cv2
import filters
from managers import WindowManager, CaptureManager
import rects
from trackers import FaceTracker

class Cameo(object):

    def __init__(self):
        self._windowManager = WindowManager('Cameo',
                                             self.onKeypress)
        self._captureManager = CaptureManager(
            cv2.VideoCapture(0), self._windowManager, True)
        self._faceTracker = FaceTracker()
        self._shouldDrawDebugRects = False
        self._curveFilter = filters.BGRPortraCurveFilter()

    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            frame = self._captureManager.frame

            self._faceTracker.update(frame)
            faces = self._faceTracker.faces
            rects.swapRects(frame, frame,
                            [face.faceRect for face in faces])

            filters.strokeEdges(frame, frame)
            self._curveFilter.apply(frame, frame)

            if self._shouldDrawDebugRects:
                self._faceTracker.drawDebugRects(frame)

            self._captureManager.exitFrame()
            self._windowManager.processEvents()

    def onKeypress(self, keycode):
        """Handle a keypress.
```

```
        space -> Take a screenshot.
        tab   -> Start/stop recording a screencast.
        x     -> Start/stop drawing debug rectangles around faces.
        escape -> Quit.

        """
        if keycode == 32: # space
            self._captureManager.writeImage('screenshot.png')
        elif keycode == 9: # tab
            if not self._captureManager.isWritingVideo:
                self._captureManager.startWritingVideo(
                    'screencast.avi')
            else:
                self._captureManager.stopWritingVideo()
        elif keycode == 120: # x
            self._shouldDrawDebugRects = \
                not self._shouldDrawDebugRects
        elif keycode == 27: # escape
            self._windowManager.destroyWindow()

if __name__=="__main__":
    Cameo().run()
```

The following screenshot is from Cameo. Face regions are outlined after the user presses *X*:

The following screenshot is from Cameo. American businessman Bill Ackman performs a takeover of the author's face:
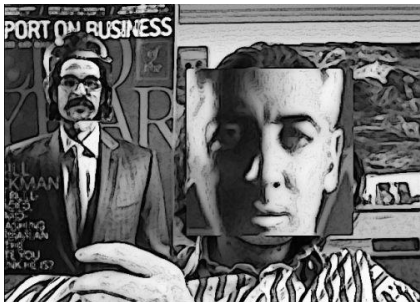
Copying faces between camera feeds

For the two-camera version, let's create a new class, `CameoDouble`, which is a subclass of `Cameo`. On initialization, a `CameoDouble` invokes the constructor of `Cameo` and also creates a second `CaptureManager`. During the main loop in `run()`, a `CameoDouble` gets new frames from both cameras and then gets face tracking results for both frames. Faces are copied from one frame to the other using `copyRect()`. Then, the destination frame is displayed, optionally with debug rectangles drawn overtop it. We can implement `CameoDouble` in `cameo.py` as follows:

Note

For some models of MacBook, OpenCV has problems using the built-in camera when an external webcam is plugged in. Specifically, the application may become deadlocked while waiting for the built-in camera to supply a frame. If you encounter this issue, use two external cameras and do not use the built-in camera.

```python
class CameoDouble(Cameo):

    def __init__(self):
        Cameo.__init__(self)
        self._hiddenCaptureManager = CaptureManager(
            cv2.VideoCapture(1))

    def run(self):
```

```
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            self._hiddenCaptureManager.enterFrame()
            frame = self._captureManager.frame
            hiddenFrame = self._hiddenCaptureManager.frame

            self._faceTracker.update(hiddenFrame)
            hiddenFaces = self._faceTracker.faces
            self._faceTracker.update(frame)
            faces = self._faceTracker.faces

            i = 0
            while i < len(faces) and i < len(hiddenFaces):
                rects.copyRect(
                    hiddenFrame, frame, hiddenFaces[i].faceRect,
                    faces[i].faceRect)
                i += 1

            filters.strokeEdges(frame, frame)
            self._curveFilter.apply(frame, frame)

            if self._shouldDrawDebugRects:
                self._faceTracker.drawDebugRects(frame)

            self._captureManager.exitFrame()
            self._hiddenCaptureManager.exitFrame()
            self._windowManager.processEvents()
```

To run a `CameoDouble` instead of a `Cameo`, we just need to modify our `if __name__=="__main__"` block, as follows:

```
if __name__=="__main__":
    #Cameo().run() # uncomment for single camera
    CameoDouble().run() # uncomment for double camera
```

Summary

We now have two versions of Cameo. One version tracks faces in a single camera feed and, when faces are found, swaps them by copying and resizing. The other version tracks faces in two camera feeds and, when faces are found in each, copies and resizes faces from one feed to replace faces in the other. Additionally, in both versions, one camera feed is made visible and effects are applied to it.

These versions of Cameo demonstrate the basic functionality that we proposed two chapters ago. The user can displace his or her face onto another body, and the result can be stylized to give it a more unified feel. However, the transplanted faces are still just rectangular cutouts. So far, no effort is made to cut away non-face parts of the rectangle or to align superimposed and underlying components such as eyes. The next chapter examines some more sophisticated techniques for facial blending, particularly using depth vision.

Chapter 5. Detecting Foreground/Background Regions and Depth

This chapter shows how to use data from a depth camera to identify foreground and background regions, such that we can limit an effect to only the foreground or only the background. As prerequisites, we need a depth camera, such as Microsoft Kinect, and we need to build OpenCV with support for our depth camera. For build instructions, see Chapter 1, *Setting up OpenCV*.

Creating modules

Our code for capturing and manipulating depth-camera data will be reusable outside `Cameo.py`. So we should separate it into a new module. Let's create a file called `depth.py` in the same directory as `Cameo.py`. We need the following import statement in `depth.py`:

```
import numpy
```

We will also need to modify our preexisting `rects.py` file so that our copy operations can be limited to a non-rectangular sub region of a rectangle. To support the changes we are going to make, let's add the following import statements to `rects.py`:

```
import numpy
import utils
```

Finally, the new version of our application will use depth-related functionality. So, let's add the following `import` statement to `Cameo.py`:

```
import depth
```

Now, let's get deeper into the subject of depth.

Capturing frames from a depth camera

Back in Chapter 2, *Handling Files, Cameras, and GUIs*, we discussed the concept that a computer can have multiple video capture devices and each device can have multiple channels. Suppose a given device is a stereo camera. Each channel might correspond to a different lens and sensor. Also, each channel might correspond to a different kind of data, such as a normal color image versus a depth map. When working with OpenCV's `VideoCapture` class or our wrapper `CaptureManager`, we can choose a device on initialization and we can read one or more channels from each frame of that device. Each device and channel is identified by an integer. Unfortunately, the numbering of devices and channels is unintuitive. The C++ version of OpenCV defines some constants for the identifiers of certain devices and channels. However, these constants are not defined in the Python version. To remedy this situation, let's add the following definitions in `depth.py`:

```
# Devices.
CV_CAP_OPENNI = 900 # OpenNI (for Microsoft Kinect)
CV_CAP_OPENNI_ASUS = 910 # OpenNI (for Asus Xtion)
# Channels of an OpenNI-compatible depth generator.
CV_CAP_OPENNI_DEPTH_MAP = 0 # Depth values in mm (CV_16UC1)
CV_CAP_OPENNI_POINT_CLOUD_MAP = 1 # XYZ in meters (CV_32FC3)
CV_CAP_OPENNI_DISPARITY_MAP = 2 # Disparity in pixels (CV_8UC1)
CV_CAP_OPENNI_DISPARITY_MAP_32F = 3 # Disparity in pixels (CV_32FC1)
CV_CAP_OPENNI_VALID_DEPTH_MASK = 4 # CV_8UC1
# Channels of an OpenNI-compatible RGB image generator.
CV_CAP_OPENNI_BGR_IMAGE = 5
CV_CAP_OPENNI_GRAY_IMAGE = 6
```

The depth-related channels require some explanation, as given in the following list:

- A **depth map** is a grayscale image in which each pixel value is the estimated distance from the camera to a surface. Specifically, an image from the CV_CAP_OPENNI_DEPTH_MAP channel gives the distance as a floating-point number of millimeters.

- A **point cloud map** is a color image in which each color corresponds to a spatial dimension (x, y, or z). Specifically, the CV_CAP_OPENNI_POINT_CLOUD_MAP channel yields a BGR image where B is x (blue is right), G is y (green is up), and R is z (red is deep), from the camera's perspective. The values are in meters.

- A **disparity map** is a grayscale image in which each pixel value is the **stereo disparity** of a surface. To conceptualize stereo disparity, let's suppose we overlay two images of a scene, shot from different viewpoints. The result would be like seeing double images. For points on any pair of twin objects in the scene, we can measure the distance in pixels. This measurement is the stereo disparity. Nearby objects exhibit greater stereo disparity than far-off objects. Thus, nearby objects appear brighter in a disparity map.

- A **valid depth mask** shows whether the depth information at a given pixel is believed to be valid (shown by a non-zero value) or invalid (shown by a value of zero). For example, if the depth camera depends on an infrared illuminator (an infrared flash), then depth information is invalid in regions that are occluded (shadowed) from this light.

The following screenshot shows a point-cloud map of a man sitting behind a sculpture of a cat:
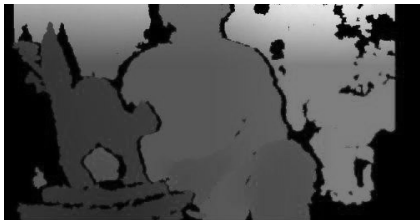
The following screenshot has a disparity map of a man sitting behind a sculpture of a cat:

A valid depth mask of a man sitting behind a sculpture of a cat is shown in the following screenshot:

Creating a mask from a disparity map

For the purposes of Cameo, we are interested in disparity maps and valid depth masks.
They can help us refine our estimates of facial regions.

Using our FaceTracker function and a normal color image, we can obtain rectangular
estimates of facial regions. By analyzing such a rectangular region in the corresponding
disparity map, we can tell that some pixels within the rectangle are outliers—too near or
too far to really be a part of the face. We can refine the facial region to exclude these
outliers. However, we should only apply this test where the data are valid, as indicated
by the valid depth mask.

Let's write a function to generate a mask whose values are 0 for rejected regions of the
facial rectangle and 1 for accepted regions. This function should take a disparity map, a
valid depth mask, and a rectangle as arguments. We can implement it in depth.py as
follows:

```
def createMedianMask(disparityMap, validDepthMask, rect = None):
    """Return a mask selecting the median layer, plus shadows."""
    if rect is not None:
        x, y, w, h = rect
        disparityMap = disparityMap[y:y+h, x:x+w]
        validDepthMask = validDepthMask[y:y+h, x:x+w]
    median = numpy.median(disparityMap)
    return numpy.where((validDepthMask == 0) | \
                       (abs(disparityMap - median) < 12),
                       1.0, 0.0)
```

To identify outliers in the disparity map, we first find the median using
numpy.median(), which takes an array as an argument. If the array is of odd length,
median() returns the value that would lie in the middle of the array if the array were
sorted. If the array is of even length, median() returns the average of the two values that
would be sorted nearest to the middle of the array.

would be sorted nearest to the middle of the array.

To generate the mask based on per-pixel Boolean operations, we use `numpy.where()` with three arguments. As the first argument, `where()` takes an array whose elements are evaluated for truth or falsity. An output array of like dimensions is returned. Wherever an element in the input array is `true`, the `where()` function's second argument is assigned to the corresponding element in the output array. Conversely, wherever an element in the input array is `false`, the `where()` function's third argument is assigned to the corresponding element in the output array.

Our implementation treats a pixel as an outlier when it has a valid disparity value that deviates from the median disparity value by 12 or more. I chose the value 12 just by experimentation. Feel free to tweak this value later based on the results you encounter when running `Cameo` with your particular camera setup.

Masking a copy operation

As part of the previous chapter's work, we wrote `copyRect()` as a copy operation that limits itself to given rectangles of the source and destination images. Now, we want to apply further limits to this copy operation. We want to use a given mask that has the same dimensions as the source rectangle. We shall copy only those pixels in the source rectangle where the mask's value is not zero. Other pixels shall retain their old values from the destination image. This logic, with an array of conditions and two arrays of possible output values, can be expressed concisely with the `numpy.where()` function that we have recently learned.

Let's open `rects.py` and edit `copyRect()` to add a new argument, `mask`. This argument may be `None`, in which case we fall back to our old implementation of the copy operation. Otherwise, we next ensure that `mask` and the images have the same number of channels. We assume that `mask` has one channel but the images may have three channels (BGR). We can add duplicate channels to `mask` using the `repeat()` and `reshape()` methods of `numpy.array`. Finally, we perform the copy operation using `where()`. The complete implementation is as follows:

```
def copyRect(src, dst, srcRect, dstRect, mask = None,
             interpolation = cv2.INTER_LINEAR):
    """Copy part of the source to part of the destination."""

    x0, y0, w0, h0 = srcRect
    x1, y1, w1, h1 = dstRect

    # Resize the contents of the source sub-rectangle.
    # Put the result in the destination sub-rectangle.
    if mask is None:
        dst[y1:y1+h1, x1:x1+w1] = \
            cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                       interpolation = interpolation)
    else:
        if not utils.isGray(src):
            # Convert the mask to 3 channels, like the image.
```

```
            mask = mask.repeat(3).reshape(h0, w0, 3)
        # Perform the copy, with the mask applied.
        dst[y1:y1+h1, x1:x1+w1] = \
            numpy.where(cv2.resize(mask, (w1, h1),
                                    interpolation = \
                                        cv2.INTER_NEAREST),
                        cv2.resize(src[y0:y0+h0, x0:x0+w0], (w1, h1),
                                    interpolation = interpolation),
                        dst[y1:y1+h1, x1:x1+w1])
```

We also need to modify our swapRects() function, which uses copyRect() to perform a circular swap of a list of rectangular regions. The modifications to swapRects() are quite simple. We just need to add a new argument, masks, which is a list of masks whose elements are passed to the respective copyRect() calls. If the given masks is None, we pass None to every copyRect() call. The following is the full implementation:

```
def swapRects(src, dst, rects, masks = None,
              interpolation = cv2.INTER_LINEAR):
    """Copy the source with two or more sub-rectangles swapped."""

    if dst is not src:
        dst[:] = src

    numRects = len(rects)
    if numRects < 2:
        return

    if masks is None:
        masks = [None] * numRects

    # Copy the contents of the last rectangle into temporary storage.
    x, y, w, h = rects[numRects - 1]
    temp = src[y:y+h, x:x+w].copy()

    # Copy the contents of each rectangle into the next.
    i = numRects - 2
    while i >= 0:
        copyRect(src, dst, rects[i], rects[i+1], masks[i],
                 interpolation)
        i -= 1

    # Copy the temporarily stored content into the first rectangle.
    copyRect(temp, dst, (0, 0, w, h), rects[0], masks[numRects - 1],
             interpolation)
```

Note that the mask in copyRect() and masks in swapRects() both default to None. Thus, our new versions of these functions are backward-compatible with our previous versions of Cameo.

Modifying the application

For the depth-camera version of Cameo, let's create a new class, CameoDepth, as a subclass of Cameo. On initialization, a CameoDepth class creates a CaptureManager

class that uses a depth camera device (either CV_CAP_OPENNI for Microsoft Kinect or CV_CAP_OPENNI_ASUS for Asus Xtion, depending on our setup). During the main loop in run(), a CameoDepth function gets a disparity map, a valid depth mask, and a normal color image in each frame. The normal color image is used to estimate facial rectangles, while the disparity map and valid depth mask are used to refine the estimate of the facial region using createMedianMask(). Faces in the normal color image are swapped using copyRect(), with the faces' respective masks applied. Then, the destination frame is displayed, optionally with debug rectangles drawn overtop it. We can implement CameoDepth in cameo.py as follows:

```
class CameoDepth(Cameo):
    def __init__(self):
        self._windowManager = WindowManager('Cameo',
                                             self.onKeypress)
        device = depth.CV_CAP_OPENNI # uncomment for Microsoft Kinect
        #device = depth.CV_CAP_OPENNI_ASUS # uncomment for Asus Xtion
        self._captureManager = CaptureManager(
            cv2.VideoCapture(device), self._windowManager, True)
        self._faceTracker = FaceTracker()
        self._shouldDrawDebugRects = False
        self._curveFilter = filters.BGRPortraCurveFilter()
    def run(self):
        """Run the main loop."""
        self._windowManager.createWindow()
        while self._windowManager.isWindowCreated:
            self._captureManager.enterFrame()
            self._captureManager.channel = \
                depth.CV_CAP_OPENNI_DISPARITY_MAP
            disparityMap = self._captureManager.frame
            self._captureManager.channel = \
                depth.CV_CAP_OPENNI_VALID_DEPTH_MASK
            validDepthMask = self._captureManager.frame
            self._captureManager.channel = \
                depth.CV_CAP_OPENNI_BGR_IMAGE
            frame = self._captureManager.frame
            self._faceTracker.update(frame)
            faces = self._faceTracker.faces
            masks = [
                depth.createMedianMask(
                    disparityMap, validDepthMask, face.faceRect) \
                for face in faces
            ]
            rects.swapRects(frame, frame,
                            [face.faceRect for face in faces], masks)
            filters.strokeEdges(frame, frame)
            self._curveFilter.apply(frame, frame)
            if self._shouldDrawDebugRects:
                self._faceTracker.drawDebugRects(frame)
            self._captureManager.exitFrame()
            self._windowManager.processEvents()
```

To run a CameoDepth function instead of a Cameo or CameoDouble function, we just need to modify our if __name__=="__main__" block, as follows:

```
if __name__=="__main__":
```

```
#Cameo().run() # uncomment for single camera
#CameoDouble().run() # uncomment for double camera
CameoDepth().run() # uncomment for depth camera
```

The following is a screenshot showing the `CameoDepth` class in action. Note that our mask gives the copied regions some irregular edges, as intended. The effect is more successful on the left and right sides of the faces (where they meet the background) than on the top and bottom (where they meet hair and neck regions of similar depth):

Summary

We now have an application that uses a depth camera, facial tracking, copy operations, masks, and image filters. By developing this application, we have gained practice in leveraging the functionality of OpenCV, NumPy, and other libraries. We have also practiced wrapping this functionality in a high-level, reusable, and object-oriented design.

Congratulations! You now have the skill to develop computer vision applications in Python using OpenCV. Still, there is always more to learn and do! If you liked working with NumPy and OpenCV, please check out these other titles from Packt Publishing:

- *NumPy Cookbook, Ivan Idris*
- *OpenCV 2 Computer Vision Application Programming Cookbook, Robert Laganière*, which uses OpenCV's C++ API for desktops

- *Mastering OpenCV with Practical Computer Vision Projects, (by multiple authors)*, which uses OpenCV's C++ API for multiple platforms

- The upcoming book, *OpenCV for iOS How-to*, which uses OpenCV's C++ API for iPhone and iPad

- *OpenCV Android Application Programming*, my upcoming book, which uses OpenCV's Java API for Android

Here ends of our tour of OpenCV's Python bindings. I hope you are able to use this book and its codebase as a starting point for rewarding work in computer vision. Let me know what you are studying or developing next!

Appendix A. Integrating with Pygame

This appendix shows how to set up the Pygame library and how to use Pygame for window management in an OpenCV application. Also, the appendix gives an overview of Pygame's other functionality and some resources for learning Pygame.

Note

All the finished code for this chapter can be downloaded from my website: http://nummist.com/opencv/3923_06.zip.

Installing Pygame

Let's assume that we already have Python set up according to one of the approaches described in Chapter 1, *Setting up OpenCV*. Depending on our existing setup, we can install Pygame in one of the following ways:

- Windows with 32-bit Python: Download and install Pygame 1.9.1 from http://pygame.org/ftp/pygame-1.9.1.win32-py2.7.msi.

- Windows with 64-bit Python: Download and install Pygame 1.9.2 preview from http://www.lfd.uci.edu/~gohlke/pythonlibs/2k2kdosm/pygame-1.9.2pre.win-amd64-py2.7.exe.

- Mac with Macports: Open **Terminal** and run the following command:

  ```
  $ sudo port install py27-game
  ```

- Mac with Homebrew: Open **Terminal** and run the following commands to install Pygame's dependencies and, then, Pygame itself:

  ```
  $ brew install sdl sdl_image sdl_mixer sdl_ttf smpeg portmidi
  ```

```
$ /usr/local/share/python/pip install \
> hg+http://bitbucket.org/pygame/pygame
```

- Ubuntu and its derivatives: Open **Terminal** and run the following command:

```
$ sudo apt-get install python-pygame
```

- Other Unix-like systems: Pygame is available in the standard repositories of many systems. Typical package names include pygame, pygame27, py-game, py27-game, python-pygame, and python27-pygame.

Now, Pygame should be ready for use.

Documentation and tutorials

Pygame's API documentation and some tutorials can be found online at http://www.pygame.org/docs/.

Al Sweigart's *Making Games With Python and Pygame* is a cookbook for recreating several classic games in Pygame 1.9.1. The free electronic version is available online at http://inventwithpython.com/pygame/chapters/ or as a downloadable PDF file at http://inventwithpython.com/makinggames.pdf.

Subclassing managers.WindowManager

As discussed in Chapter 2, *Handling Cameras, Files and GUIs*, our object-oriented design allows us to easily swap OpenCV's HighGUI window manager for another window manager, such as Pygame. To do so, we just need to subclass our managers.WindowManager class and override four methods: createWindow(), show(), destroyWindow(), and processEvents(). Also, we need to import some new dependencies.

To proceed, we need the managers.py file from Chapter 2, *Handling Cameras, Files, and GUIs* and the utils.py file from Chapter 4, *Tracking Faces with Haar Cascades*. From utils.py, we only need one function, isGray(), which we implemented in Chapter 4, *Tracking Faces with Haar Cascades*. Let's edit managers.py to add the following imports:

```
import pygame
import utils
```

Also in managers.py, somewhere after our WindowManager implementation, we want to add our new subclass called PygameWindowManager:

```
class PygameWindowManager(WindowManager):
    def createWindow(self):
        pygame.display.init()
        pygame.display.set_caption(self._windowName)
```

```
            self._isWindowCreated = True
    def show(self, frame):
        # Find the frame's dimensions in (w, h) format.
        frameSize = frame.shape[1::-1]
        # Convert the frame to RGB, which Pygame requires.
        if utils.isGray(frame):
            conversionType = cv2.COLOR_GRAY2RGB
        else:
            conversionType = cv2.COLOR_BGR2RGB
        rgbFrame = cv2.cvtColor(frame, conversionType)
        # Convert the frame to Pygame's Surface type.
        pygameFrame = pygame.image.frombuffer(
            rgbFrame.tostring(), frameSize, 'RGB')
        # Resize the window to match the frame.
        displaySurface = pygame.display.set_mode(frameSize)
        # Blit and display the frame.
        displaySurface.blit(pygameFrame, (0, 0))
        pygame.display.flip()
    def destroyWindow(self):
        pygame.display.quit()
        self._isWindowCreated = False
    def processEvents(self):
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN and \
                    self.keypressCallback is not None:
                self.keypressCallback(event.key)
            elif event.type == pygame.QUIT:
                self.destroyWindow()
                return
```

Note that we are using two Pygame modules: pygame.display and pygame.event.

A window is created by calling pygame.display.init() and destroyed by calling pygame.display.quit(). Repeated calls to display.init() have no effect, as Pygame is intended for single-window applications only. The Pygame window is a drawing surface of type pygame.Surface. To get a reference to this Surface, we can call pygame.display.get_surface() or pygame.display.set_mode(). The latter function modifies the Surface entity's properties before returning it. A Surface entity has a blit() method, which takes, as arguments, another Surface and a coordinate pair where the latter Surface should be "blitted" (drawn) onto the first. When we are done updating the window's Surface for the current frame, we should display it by calling pygame.display.flip().

Events, such as keypresses, are polled by calling pygame.event.get(), which returns the list of all events that have occurred since the last call. Each event is of type pygame.event.Event and has the property type, which indicates the category of an event such as pygame.KEYDOWN for keypresses and pygame.QUIT for the window's **Close** button being clicked. Depending on the value of type, an Event entity may have other properties, such as key (an ASCII key code) for the KEYDOWN events.

Relative to the base WindowManager that uses HighGUI, PygameWindowManager incurs some overhead cost by converting between OpenCV's image format and Pygame's Surface format of each frame. However, PygameWindowManager offers normal

surface format of each frame. However, `PygameWindowManager` offers normal window closing behavior, whereas the base `WindowManager` does not.

Modifying the application

Let's modify the `cameo.py` file to use `PygameWindowManager` instead of `WindowManager`. Find the following line in `cameo.py`:

```
from managers import WindowManager, CaptureManager
```

Replace it with:

```
from managers import PygameWindowManager as WindowManager, \
                      CaptureManager
```

That's all! Now `cameo.py` uses a Pygame window that should close when the standard **Close** button is clicked.

Further uses of Pygame

We have used only some basic functions of the `pygame.display` and `pygame.event` modules. Pygame provides much more functionality, including:

- Drawing 2D geometry
- Drawing text
- Managing groups of drawable AI entities (sprites)
- Capturing various input events relating to the window, keyboard, mouse, and joysticks/gamepads
- Creating custom events
- Playback and synthesis of sounds and music

For example, Pygame might be a suitable backend for a game that uses computer vision, whereas HighGUI would not be.

Summary

By now, we should have an application that uses OpenCV for capturing (and possibly manipulating) images, while using Pygame for displaying the images and catching events. Starting from this basic integration example, you might want to expand `PygameWindowManager` to wrap additional Pygame functionality or you might want to create another `WindowManager` subclass to wrap another library.

Appendix B. Generating Haar Cascades for Custom

Targets

This appendix shows how to generate Haar cascade XML files like the ones used in Chapter 4, *Tracking Faces with Haar Cascades*. By generating our own cascade files, we can potentially track any pattern or object, not just faces. However, good results might not come quickly. We must carefully gather images, configure script parameters, perform real-world tests, and iterate. A lot of human time and processing time might be involved.

Gathering positive and negative training images

Do you know the flashcard pedagogy? It is a method of teaching words and recognition skills to small children. The teacher shows the class a series of pictures and says the following:

> *"This is a cow. Moo! This is a horse. Neigh!"*

The way that cascade files are generated is analogous to the flashcard pedagogy. To learn how to recognize cows, the computer needs **positive training images** that are pre-identified as cows and **negative training images** that are pre-identified as *non-cows*. Our first step, as trainers, is to gather these two sets of images.

When deciding how many positive training images to use, we need to consider the various ways in which our users might view the target. The ideal, simplest case is that the target is a 2D pattern that is always on a flat surface. In this case, one positive training image might be enough. However, in other cases, hundreds or even thousands of training images might be required. Suppose that the target is your country's flag. When printed on a document, the flag might have a predictable appearance but when printed on a piece of fabric that is blowing in the wind, the flag's appearance is highly variable. A natural, 3D target, such as a human face, might range even more widely in appearance. Ideally, our set of positive training images should be representative of the many variations our camera may capture. Optionally, any of our positive training images may contain multiple instances of the target.

For our negative training set, we want a large number of images that do not contain any instances of the target but do contain other things that our camera is likely to capture. For example, if a flag is our target, our negative training set might include photos of the sky in various weather conditions. (The sky is not a flag but is often seen behind a flag.) Do not assume too much though. If the camera's environment is unpredictable and the target occurs in many settings, use a wide variety of negative training images. Consider building a set of generic environmental images that you can reuse across multiple training scenarios.

Finding the training executables

To automate cascade training as much as possible, OpenCV provides two executables. Their names and locations depend on the operating system and the particular setup of OpenCV, as described in the following two sections.

On Windows

The two executables on Windows are called `ONopencv_createsamples.exe` and `ONopencv_traincascade.exe`. They are not prebuilt. Rather, they are present only if you compiled OpenCV from source. Their parent folder is one of the following, depending on the compilation approach you chose in <u>Chapter 1</u>, *Setting up OpenCV*:

- MinGW: `<unzip_destination>\bin`
- Visual Studio or Visual C++ Express: `<unzip_destination>\bin\Release`

If you want to add the executables' folder to the system's `Path` variable, refer back to the instructions in the information box in the *Making the choice on Windows XP, Windows Vista, Windows 7, and Windows 8* section of <u>Chapter 1</u>, *Setting up OpenCV*. Otherwise, take note of the executables' full path because we will need to use it in running them.

On Mac, Ubuntu, and other Unix-like systems

The two executables on Mac, Ubuntu, and other Unix-like systems are called `opencv_createsamples` and `opencv_traincascade`. Their parent folder is one of the following, depending on your system and the approach that you chose in <u>Chapter 1</u>, *Setting up OpenCV*:

- Mac with MacPorts: `/opt/local/bin`
- Mac with Homebrew: `/opt/local/bin` or `/opt/local/sbin`
- Ubuntu with Apt: `/usr/bin`
- Ubuntu with my custom installation script: `/usr/local/bin`
- Other Unix-like systems: `/usr/bin` and `/usr/local/bin`

Except in the case of Mac with Homebrew, the executables' folder should be in `PATH` by default. For Homebrew, if you want to add the relevant folders to `PATH`, see the instructions in the second step of the *Using Homebrew with ready-made packages (no support for depth cameras)* section of <u>Chapter 1</u>, *Setting up OpenCV*. Otherwise, note the executables' full path because we will need to use it in running them.

Creating the training sets and cascade

Hereafter, we will refer to the two executables as `<opencv_createsamples>` and `<opencv_traincascade>`. Remember to substitute the path and filename that are appropriate to your system and setup.

These executables have certain data files as inputs and outputs. Following is a typical approach to generating these data files:

1. Manually create a text file that describes the set of negative training images. We will refer to this as `<negative_description>`.

2. Manually create a text file that describes the set of positive training images. We will refer to this as `<positive_description>`.

3. Run `<opencv_createsamples>` with `<negative_description>` and `<positive_description>` as arguments. The executable creates a binary file describing the training data. We will refer to the latter file as `<binary_description>`.

4. Run `<opencv_traincascade>` with `<binary_description>` as an argument. The executable creates the binary cascade file, which we will refer to as `<cascade>`.

The actual names and paths of `<negative_description>`, `<positive_description>`, `<binary_description>`, and `<cascade>` may be anything we choose.

Now, let's look at each of the three steps in detail.

Creating <negative_description>

`<negative_description>` is a text file listing the relative paths to all negative training images. The paths should be separated by line breaks. For example, suppose we have the following directory structure, where `<negative_description>` is `negative/desc.txt`:

```
negative
    desc.txt
    images
        negative 0.png
        negative 1.png
```

Then, the contents of `negative/desc.txt` could be as follows:

```
"images/negative 0.png"
"images/negative 1.png"
```

For a small number of images, we can write such a file by hand. For a large number of images, we should instead use the command line to find relative paths matching a certain pattern and to output these matches to a file. Continuing our example, we could generate

pattern and to output these matches to a file. Continuing our example, we could generate negative/desc.txt by running the following commands on Windows in Command Prompt:

```
> cd negative
> forfiles /m images\*.png /c "cmd /c echo @relpath" > desc.txt
```

Note that in this case, relative paths are formatted as .\images\negative 0.png, which is acceptable.

Alternatively, in a Unix-like shell, such as Terminal on Mac or Ubuntu, we could run the following commands:

```
$ cd negative
$ find images/*.png | sed -e "s/^/\"/g;s/$/\"/g" > desc.txt
```

Creating <positive_description>

<positive_description> is needed if we have more than one positive training image. Otherwise, proceed to the next section. <positive_description> is a text file listing the relative paths to all positive training images. After each path, <positive_description> also contains a series of numbers indicating how many instances of the target are found in the image and which sub-rectangles contain instances of the target. For each sub-rectangle, the numbers are in this order: x, y, width, and height. Consider the following example:

```
"images/positive 0.png"  1  120 160 40 40
"images/positive 1.png"  2  200 120 40 60  80 60 20 20
```

Here, images/positive 0.png contains one instance of the target in a sub-rectangle whose upper-left corner is at (120, 160) and whose lower-right corner is at (160, 200). Meanwhile, images/positive 1.png contains two instances of the target. One instance is in a sub-rectangle whose upper-left corner is at (200, 120) and whose lower-right corner is at (240, 180). The other instance is in a sub-rectangle whose upper-left corner is at (80, 60) and whose lower-right corner is at (100, 80).

To create such a file, we can start by generating the list of image paths in the same manner as for <negative_description>. Then, we must manually add data about target instances based on an expert (human) analysis of the images.

Creating <binary_description> by running <opencv_createsamples>

Assuming we have multiple positive training images and, thus, we created <positive_description>, we can now generate <binary_description> by running the following command:

```
$ <opencv_createsamples> -vec <binary_description> -info
<positive_description> -bg <negative_description>
```

Alternatively, if we have a single positive training image, which we will refer to as `<positive_image>`, we should run the following command instead:

```
$ <opencv_createsamples> -vec <binary_description> -image
<positive_image> -bg <negative_description>
```

For other (optional) flags of `<opencv_createsamples>`, see the official documentation at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

Creating <cascade> by running <opencv_traincascade>

Finally, we can generate `<cascade>` by running the following command:

```
$ <opencv_traincascade> -data <cascade> -vec <binary_description> -bg
<negative_description>
```

For other (optional) flags of `<opencv_traincascade>`, see the official documentation at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

Tip

Vocalizations

For good luck, make an imitative sound when running `<opencv_traincascade>`. For example, say "Moo!" if the positive training images are cows.

Testing and improving <cascade>

`<cascade>` is an XML file that is compatible with the constructor for OpenCV's `CascadeClassifier` class. For an example of how to use `CascadeClassifier`, refer back to our implementation of `FaceTracker` in Chapter 4, *Tracking Faces with Haar Cascades*. By copying and modifying `FaceTracker` and `Cameo`, you should be able to create a simple test application that draws rectangles around tracked instances of your custom target.

Perhaps in your first attempts at cascade training, you will not get reliable tracking results. To improve your training results, do the following:

- Consider making your classification problem more specific. For example, a `bald, shaven, male face without glasses` cascade might be easier to train than a general face cascade. Later, as your results improve, you can try to expand your problem again.

- Gather more training images, many more!

- Ensure that `<negative_description>` contains *all* the negative training images and *only* the negative training images.

- Ensure that `<positive_description>` contains *all* the positive training images and *only* the positive training images.

- Ensure that the sub-rectangles specified in `<positive_description>` are accurate.

- Review and experiment with the optional flags to `<opencv_createsamples>` and `<opencv_traincascade>`. The flags are described in the official documentation at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

Good luck and good image-hunting!

Summary

We have discussed the data and executables that are used in generating cascade files that are compatible with OpenCV's `CascadeClassifier`. Now, you can start gathering images of your favorite things and training classifiers for them!

Index

windows 7, or windows 8

T

U

V

W