Joseph Howse , Prateek Joshi
Michael Beyeler

OpenCV: Computer Vision Projects with Python

Learning Path

Get savvy with OpenCV and actualize cool computer vision applications

Packt

OpenCV: Computer Vision Projects with Python

Get savvy with OpenCV and actualize cool computer vision applications

A course in three modules

Packt

BIRMINGHAM - MUMBAI

OpenCV: Computer Vision Projects with Python

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: October 2016

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

ISBN 978-1-78712-549-0

www.packtpub.com

Credits

Authors

Joseph Howse

Prateek Joshi

Michael Beyeler

Content Development Editor

Mayur Pawanikar

Production Coordinator

Nilesh Mohite

Reviewers

David Millán Escrivá

Abid K.

Will Brennan

Gabriel Garrido Calvo

Pavan Kumar Pavagada Nagaraja

Marvin Smith

Jia-Shen Boon

Florian LE BOURDAIS

Steve Goldsmith

Rahul Kavi

Scott Lobdell

Vipul Sharma

Preface

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.

Computer vision is found everywhere in modern technology. OpenCV for Python enables us to run computer vision algorithms in real time. With the advent of powerful machines, we are getting more processing power to work with. Using this technology, we can seamlessly integrate our computer vision applications into the cloud. Web developers can develop complex applications without having to reinvent the wheel.

This course is specifically designed to teach the following topics. First, we will learn how to get started with OpenCV and OpenCV 3's Python API, and develop a computer vision application that tracks body parts. Then, we will build amazing intermediate-level computer vision applications such as making an object disappear from an image, identifying different shapes, reconstructing a 3D map from images, and building an augmented reality application. Finally, we'll move to more advanced projects such as hand gesture recognition, tracking visually salient objects, as well as recognizing traffic signs and emotions on faces using support vector machines and multi-layer perceptron respectively.

What this learning path covers

Module 1, OpenCV Computer Vision with Python, in this module you can have a development environment that links Python, OpenCV, depth camera libraries (OpenNI, SensorKinect), and general-purpose scientific libraries (NumPy, SciPy).

Module 2, OpenCV with Python By Example, this module covers various examples at different levels, teaching you about the different functions of OpenCV, and their actual implementations.

Module 3, OpenCV with Python Blueprints, this module intends to give the tools, knowledge, and skills you need to be OpenCV experts and this newly gained experience will allow you to develop your own advanced computer vision applications.

What you need for this learning path

This course 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 projects described in the course. 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.

The hardware requirement being a webcam (or camera device), except for Chapter 2, Hand Gesture Recognition Using a Kinect Depth Sensor , of the 3rd Module which instead requires access to a Microsoft Kinect 3D Sensor or an Asus Xtion.

The course contains projects with the following requirements.

All projects can run on any of Windows, Mac, or Linux, and they require the following software packages:

- OpenCV 2.4.9 or later: Recent 32-bit and 64-bit versions as well as installation instructions are available at `http://opencv.org/downloads.html`. Platform-specific installation instructions can be found at `http://docs.opencv.org/doc/tutorials/introduction/table_of_content_introduction/table_of_content_introduction.html`.
- Python 2.7 or later: Recent 32-bit and 64-bit installers are available at `https://www.python.org/downloads`. The installation instructions can be found at `https://wiki.python.org/moin/BeginnersGuide/Download`.

- NumPy 1.9.2 or later: This package for scientific computing officially comes in 32-bit format only, and can be obtained from `http://www.scipy.org/scipylib/download.html`. The installation instructions can be found at `http://www.scipy.org/scipylib/building/index.html#building`.

wxPython 2.8 or later: This GUI programming toolkit can be obtained from `http://www.wxpython.org/download.php`. Its installation instructions are given at `http://wxpython.org/builddoc.php`.

In addition, some chapters require the following free Python modules:

- SciPy 0.16.0 or later: This scientific Python library officially comes in 32-bit only, and can be obtained from `http://www.scipy.org/scipylib/download.html`. The installation instructions can be found at `http://www.scipy.org/scipylib/building/index.html#building`.
- matplotlib 1.4.3 or later: This 2D plotting library can be obtained from `http://matplotlib.org/downloads.html`. Its installation instructions can be found by going `http://matplotlib.org/faq/installing_faq.html#how-to-install`.
- libfreenect 0.5.2 or later: The libfreenect module by the OpenKinect project (`http://www.openkinect.org`) provides drivers and libraries for the Microsoft Kinect hardware, and can be obtained from `https://github.com/OpenKinect/libfreenect`. Its installation instructions can be found at `http://openkinect.org/wiki/Getting_Started`.

Furthermore, the use of iPython (`http://ipython.org/install.html`) is highly recommended as it provides a flexible, interactive console interface.

Finally, if you are looking for help or get stuck along the way, you can go for several websites that provide excellent help, documentation, and tutorials:

- The official OpenCV API reference, user guide, and tutorials: `http://docs.opencv.org`

The official OpenCV forum: `http://www.answers.opencv.org/questions`

OpenCV-Python tutorials by Alexander Mordvintsev and Abid Rahman K: `http://opencv-python-tutroals.readthedocs.org/en/latest`

Who this learning path is for

This Learning Path is for someone who has a working knowledge of Python and wants to try out OpenCV. This Learning Path will take you from a beginner to an expert in computer vision applications using OpenCV.

OpenCV's applications are humongous and this Learning Path is the best resource to get yourself acquainted thoroughly with OpenCV.

Reader feedback

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

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

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

Customer support

Now that you are the proud owner of a Packt course, 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 this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.

6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/OpenCV-Computer-Vision-Projects-with-Python`. We also have other code bundles from our rich catalog of books, videos, and courses available at `https://github.com/PacktPublishing/`. Check them out!

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

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

Piracy

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

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

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

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1

OpenCV Computer Vision with Python

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

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.

> 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.

> 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. Copy `<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 any 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.

> 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
   ```

> **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).

> 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/`
- **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.

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.

> 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 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)
```

> 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]`.

> 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.

> 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.
- 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 to 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`.

> 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.

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, *Appendix A, 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:

```
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)
        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.

> 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:
            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 with 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()` 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.

Appendix A, *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`:

```
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.

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.

[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.

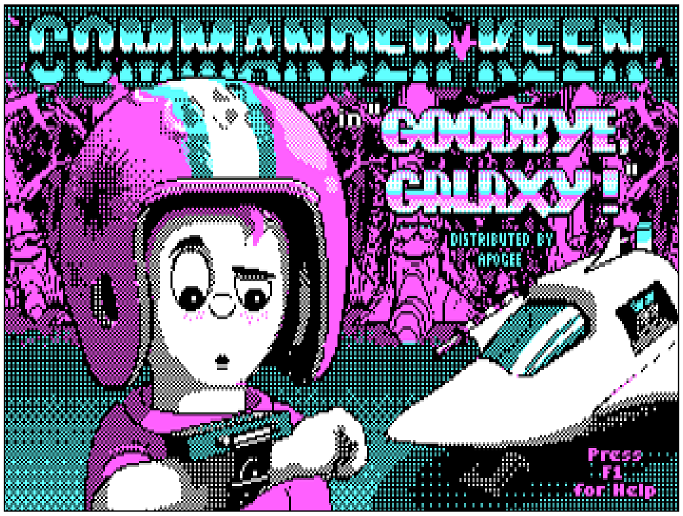
One use of channel mixing is to simulate some other, smaller color space inside RGB or BGR. By assigning equal values to any two channels, we can collapse part of the color space and create the impression that our palette is based on just two colors of light (blended additively) or two inks (blended 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 maximum 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`:

```
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:

```
    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)
            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`:

```
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 take 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:

```
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), `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.

> 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 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.

> 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 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.

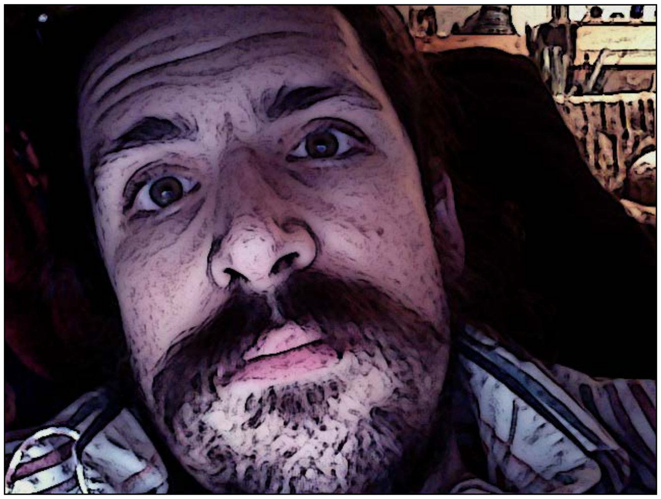
Modifying the application

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.

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!

> All the finished code for this chapter can be downloaded from my website: `http://nummist.com/opencv/3923_04.zip`.

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:

- **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

> 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
```

> 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 |of 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
- `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 we 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.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 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 method 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.

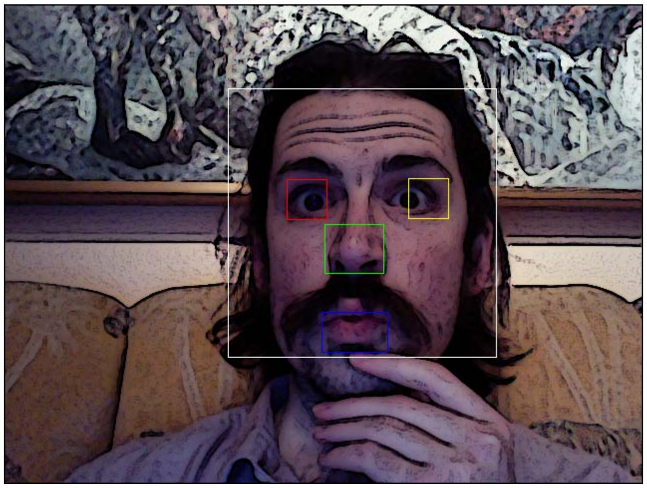
For now, we will limit ourselves to manipulating faces as a whole and not subelements such as eyes. However, you could modify the code to swap only eyes, for example. If you try this, be careful to check that the relevant subrectangles of the face are not `None`.

Swapping faces in one camera feed

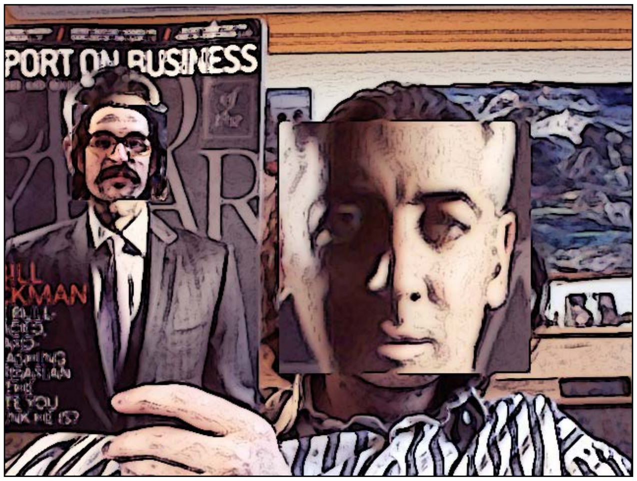
For the single-camera version, the modifications are quite straightforward. On initialization of `Cameo`, we create a `FaceTracker` and a Boolean variable indicating whether debug rectangles should be drawn for the `FaceTracker`. The Boolean is toggled in `onKeypress()` in response to the *X* key. As part of the main loop in `run()`, we update our `FaceTracker` with the current frame. 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:

> 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.

```
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.

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`:

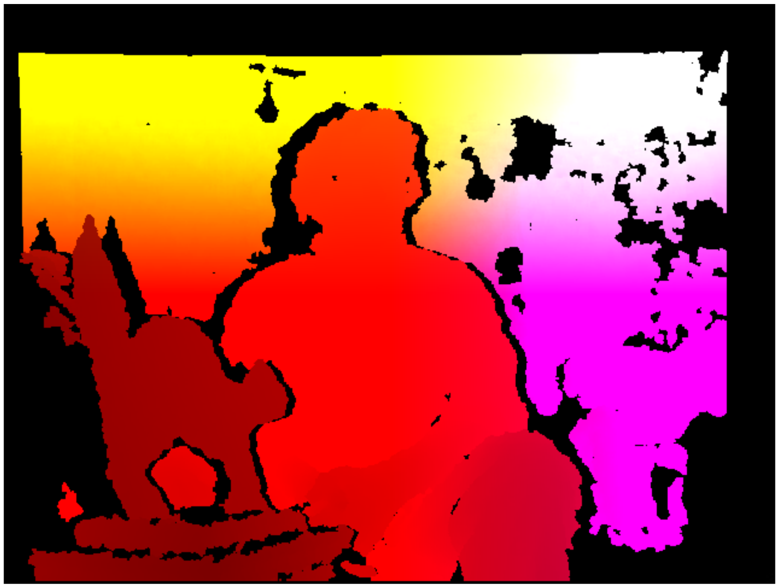
```
# Devices.
CV_CAP_OPENNI = 900 # OpenNI (for Microsoft Kinect)
CV_CAP_OPENNI_ASUS = 910 # OpenNI (for Asus Xtion)
# Channels of an OpenNI-compatible depth generator.
CV_CAP_OPENNI_DEPTH_MAP = 0 # Depth values in mm (CV_16UC1)
CV_CAP_OPENNI_POINT_CLOUD_MAP = 1 # XYZ in meters (CV_32FC3)
CV_CAP_OPENNI_DISPARITY_MAP = 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:

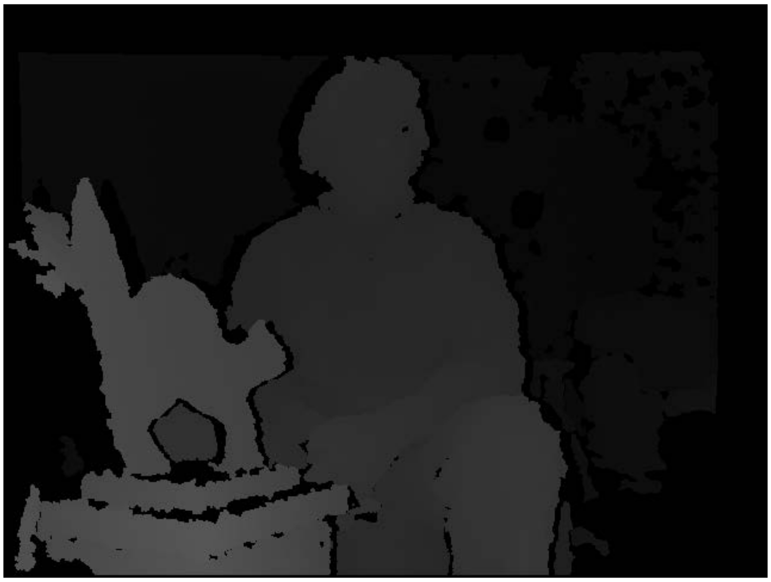
- A **depth map** is a grayscale image in which each pixel value is the estimated distance from the camera to a surface. Specifically, an image from the `CV_CAP_OPENNI_DEPTH_MAP` channel gives the distance as a floating-point number of millimeters.

- 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.

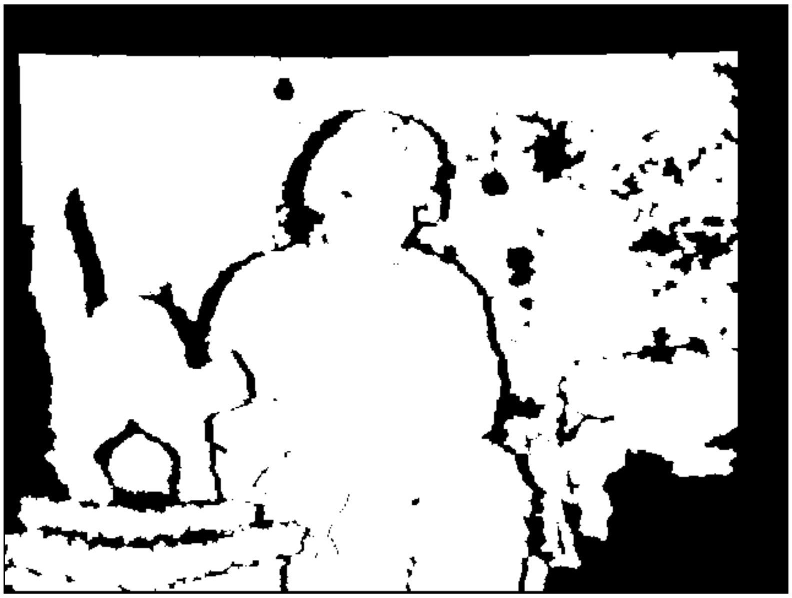
The following screenshot shows a point-cloud map of a man sitting behind a sculpture of a cat:



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.

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

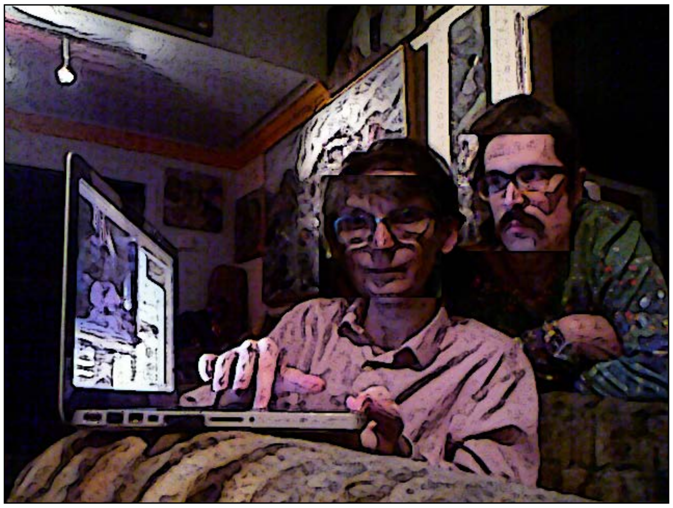
For the depth-camera version of Cameo, let's create a new class, CameoDepth, as a subclass of Cameo. On initialization, a CameoDepth class creates a CaptureManager class that uses a depth camera device (either CV_CAP_OPENNI for Microsoft Kinect or CV_CAP_OPENNI_ASUS for Asus Xtion, depending on our setup). 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!

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.

> 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 has 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 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.

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 *Chapter 1*, *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 *Chapter 1*, *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 *Chapter 1*, *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 *Chapter 1*, *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 file as `<negative_description>`.
2. Manually create a text file that describes the set of positive training images. We will refer to this file 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 `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 those 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`.

> **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!

Module 2

OpenCV with Python By Example

Build real-world computer vision applications and develop cool demos using OpenCV for Python

1
Detecting Edges and Applying Image Filters

In this chapter, we are going to see how to apply cool visual effects to images. We will learn how to use fundamental image processing operators. We are going to discuss edge detection and how we can use image filters to apply various effects on photos.

By the end of this chapter, you will know:

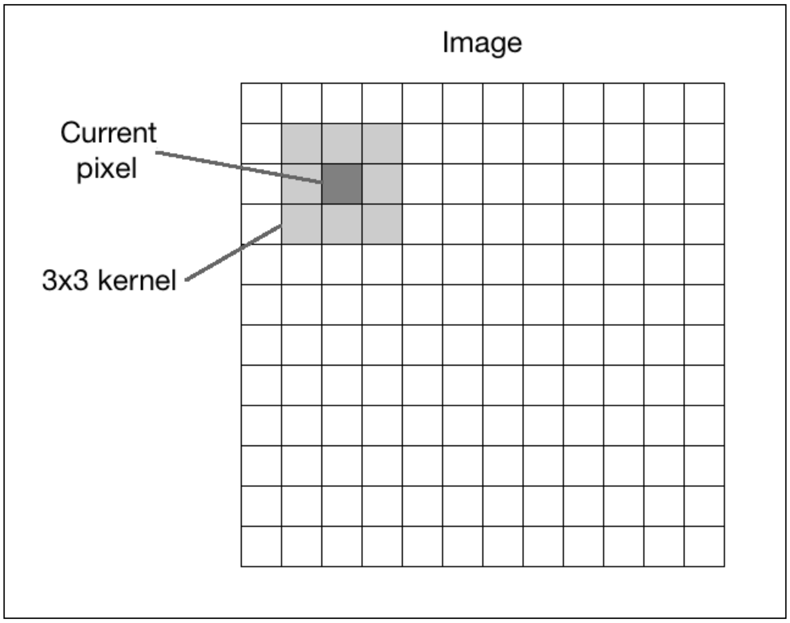
- What is 2D convolution and how to use it
- How to blur an image
- How to detect edges in an image
- How to apply motion blur to an image
- How to sharpen and emboss an image
- How to erode and dilate an image
- How to create a vignette filter
- How to enhance image contrast

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

2D convolution

Convolution is a fundamental operation in image processing. We basically apply a mathematical operator to each pixel and change its value in some way. To apply this mathematical operator, we use another matrix called a **kernel**. The kernel is usually much smaller in size than the input image. For each pixel in the image, we take the kernel and place it on top such that the center of the kernel coincides with the pixel under consideration. We then multiply each value in the kernel matrix with the corresponding values in the image, and then sum it up. This is the new value that will be substituted in this position in the output image.

Here, the kernel is called the "image filter" and the process of applying this kernel to the given image is called "image filtering". The output obtained after applying the kernel to the image is called the filtered image. Depending on the values in the kernel, it performs different functions like blurring, detecting edges, and so on. The following figure should help you visualize the image filtering operation:



Let's start with the simplest case which is identity kernel. This kernel doesn't really change the input image. If we consider a 3x3 identity kernel, it looks something like the following:

$$I = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Blurring

Blurring refers to averaging the pixel values within a neighborhood. This is also called a **low pass filter**. A low pass filter is a filter that allows low frequencies and blocks higher frequencies. Now, the next question that comes to our mind is—What does "frequency" mean in an image? Well, in this context, frequency refers to the rate of change of pixel values. So we can say that the sharp edges would be high frequency content because the pixel values change rapidly in that region. Going by that logic, plain areas would be low frequency content. Going by this definition, a low pass filter would try to smoothen the edges.

A simple way to build a low pass filter is by uniformly averaging the values in the neighborhood of a pixel. We can choose the size of the kernel depending on how much we want to smoothen the image, and it will correspondingly have different effects. If you choose a bigger size, then you will be averaging over a larger area. This tends to increase the smoothening effect. Let's see what a 3x3 low pass filter kernel looks like:

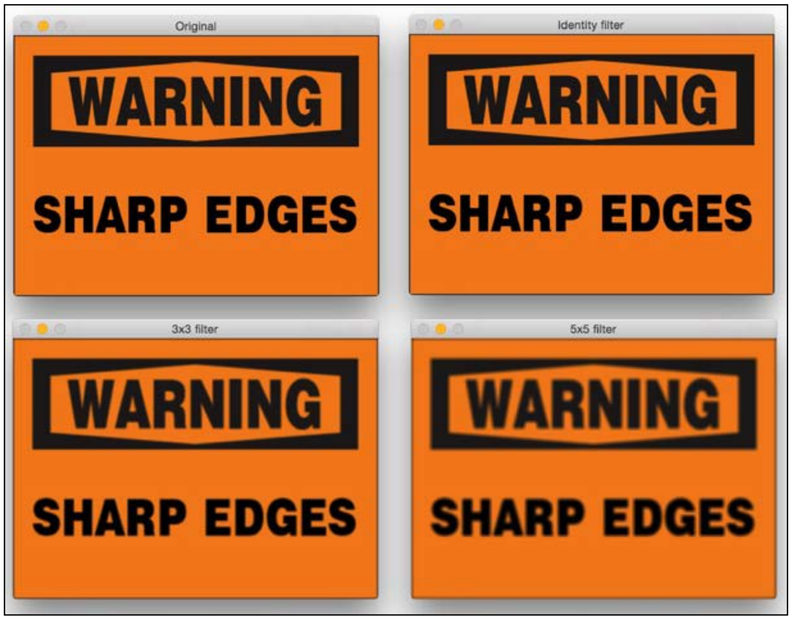
$$L = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

We are dividing the matrix by 9 because we want the values to sum up to 1. This is called **normalization**, and it's important because we don't want to artificially increase the intensity value at that pixel's location. So you should normalize the kernel before applying it to an image. Normalization is a really important concept, and it is used in a variety of scenarios, so you should read a couple of tutorials online to get a good grasp on it.

Here is the code to apply this low pass filter to an image:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
rows, cols = img.shape[:2]

kernel_identity = np.array([[0,0,0], [0,1,0], [0,0,0]])
kernel_3x3 = np.ones((3,3), np.float32) / 9.0
kernel_5x5 = np.ones((5,5), np.float32) / 25.0

cv2.imshow('Original', img)

output = cv2.filter2D(img, -1, kernel_identity)
cv2.imshow('Identity filter', output)

output = cv2.filter2D(img, -1, kernel_3x3)
cv2.imshow('3x3 filter', output)

output = cv2.filter2D(img, -1, kernel_5x5)
cv2.imshow('5x5 filter', output)

cv2.waitKey(0)
```

If you run the preceding code, you will see something like this:



The size of the kernel versus the blurriness

In the preceding code, we are generating different kernels in the code which are `kernel_identity`, `kernel_3x3`, and `kernel_5x5`. We use the function, `filter2D`, to apply these kernels to the input image. If you look at the images carefully, you can see that they keep getting blurrier as we increase the kernel size. The reason for this is because when we increase the kernel size, we are averaging over a larger area. This tends to have a larger blurring effect.

An alternative way of doing this would be by using the OpenCV function, `blur`. If you don't want to generate the kernels yourself, you can just use this function directly. We can call it using the following line of code:

```
    output = cv2.blur(img, (3,3))
```

This will apply the 3x3 kernel to the input and give you the output directly.

Edge detection

The process of edge detection involves detecting sharp edges in the image and producing a binary image as the output. Typically, we draw white lines on a black background to indicate those edges. We can think of edge detection as a high pass filtering operation. A high pass filter allows high frequency content to pass through and blocks the low frequency content. As we discussed earlier, edges are high frequency content. In edge detection, we want to retain these edges and discard everything else. Hence, we should build a kernel that is the equivalent of a high pass filter.

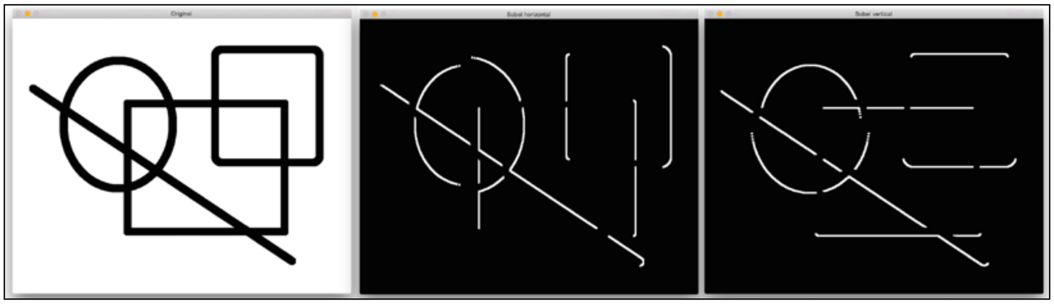
Let's start with a simple edge detection filter known as the `Sobel` filter. Since edges can occur in both horizontal and vertical directions, the `Sobel` filter is composed of the following two kernels:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The kernel on the left detects horizontal edges and the kernel on the right detects vertical edges. OpenCV provides a function to directly apply the `Sobel` filter to a given image. Here is the code to use Sobel filters to detect edges:

```
import cv2
import numpy as np

img = cv2.imread('input_shapes.png', cv2.IMREAD_GRAYSCALE)
rows, cols = img.shape

sobel_horizontal = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
sobel_vertical = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

cv2.imshow('Original', img)
cv2.imshow('Sobel horizontal', sobel_horizontal)
cv2.imshow('Sobel vertical', sobel_vertical)

cv2.waitKey(0)
```

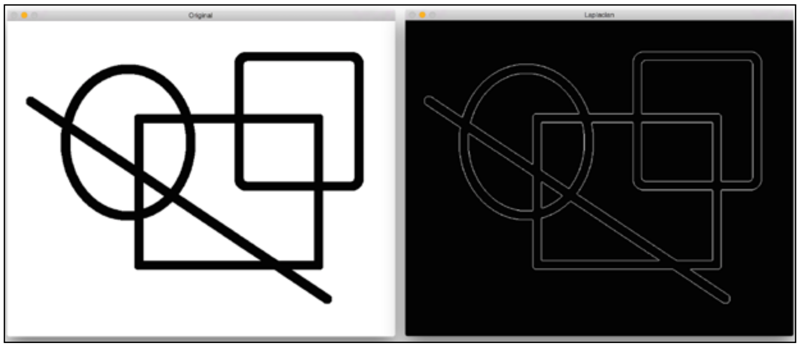
The output will look something like the following:



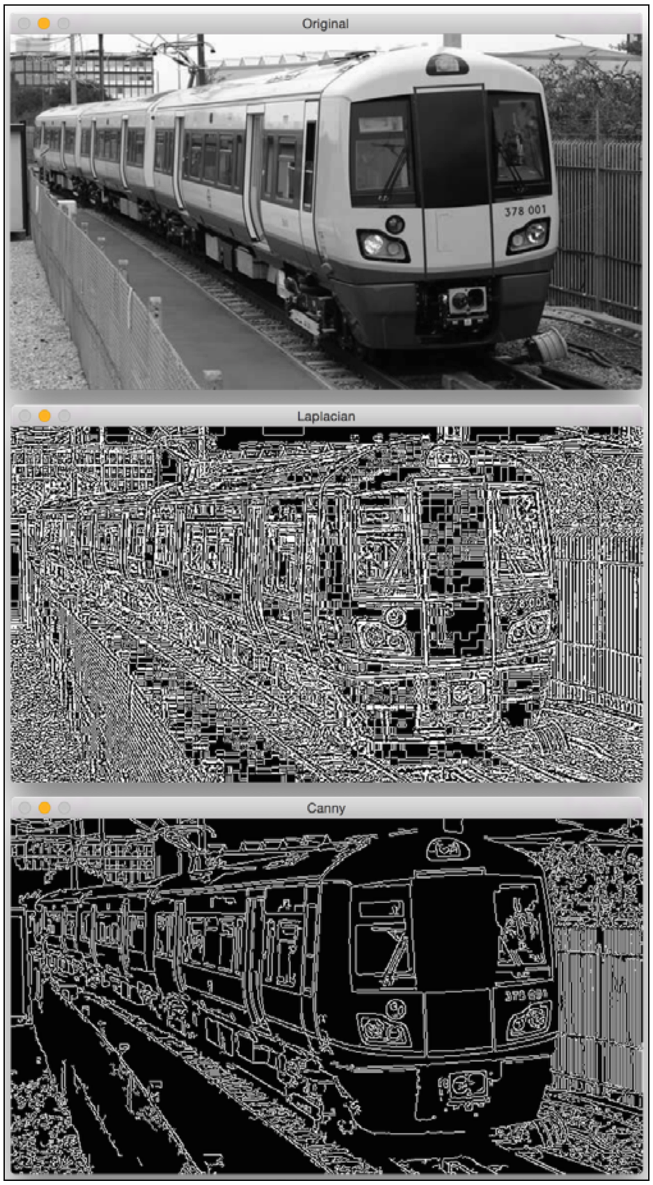
In the preceding figure, the image in the middle is the output of horizontal edge detector, and the image on the right is the vertical edge detector. As we can see here, the `Sobel` filter detects edges in either a horizontal or vertical direction and it doesn't give us a holistic view of all the edges. To overcome this, we can use the `Laplacian` filter. The advantage of using this filter is that it uses double derivative in both directions. You can call the function using the following line:

```
laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

The output will look something like the following screenshot:



Even though the `Laplacian` kernel worked great in this case, it doesn't always work well. It gives rise to a lot of noise in the output, as shown in the screenshot that follows. This is where the `Canny edge` detector comes in handy:



As we can see in the above images, the `Laplacian` kernel gives rise to a noisy output and this is not exactly useful. To overcome this problem, we use the `Canny edge` detector. To use the `Canny edge` detector, we can use the following function:

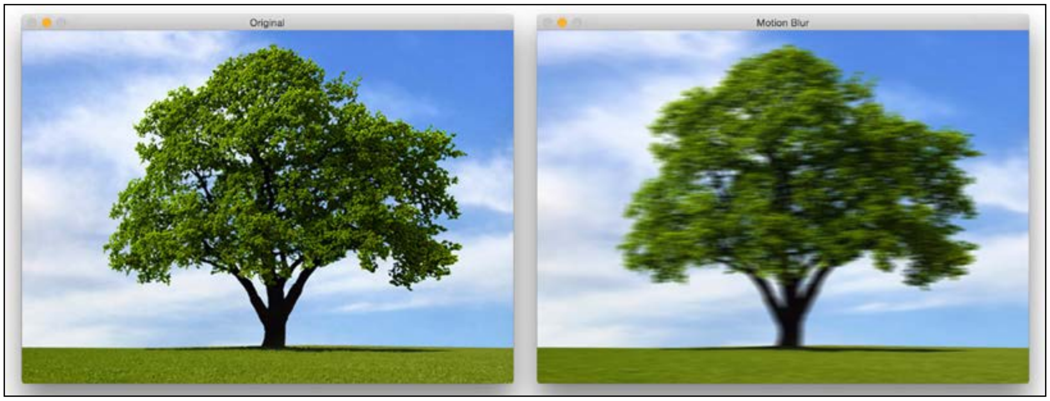
```
canny = cv2.Canny(img, 50, 240)
```

As we can see, the quality of the Canny edge detector is much better. It takes two numbers as arguments to indicate the thresholds. The second argument is called the low threshold value, and the third argument is called the high threshold value. If the gradient value is above the high threshold value, it is marked as a strong edge. The Canny Edge Detector starts tracking the edge from this point and continues the process until the gradient value falls below the low threshold value. As you increase these thresholds, the weaker edges will be ignored. The output image will be cleaner and sparser. You can play around with the thresholds and see what happens as you increase or decrease their values. The overall formulation is quite deep. You can learn more about it at `http://www.intelligence.tuc.gr/~petrakis/courses/computervision/canny.pdf`

Motion blur

When we apply the motion blurring effect, it will look like you captured the picture while moving in a particular direction. For example, you can make an image look like it was captured from a moving car.

The input and output images will look like the following ones:



Following is the code to achieve this motion blurring effect:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
cv2.imshow('Original', img)

size = 15

# generating the kernel
kernel_motion_blur = np.zeros((size, size))
kernel_motion_blur[int((size-1)/2), :] = np.ones(size)
kernel_motion_blur = kernel_motion_blur / size

# applying the kernel to the input image
output = cv2.filter2D(img, -1, kernel_motion_blur)

cv2.imshow('Motion Blur', output)
cv2.waitKey(0)
```

Under the hood

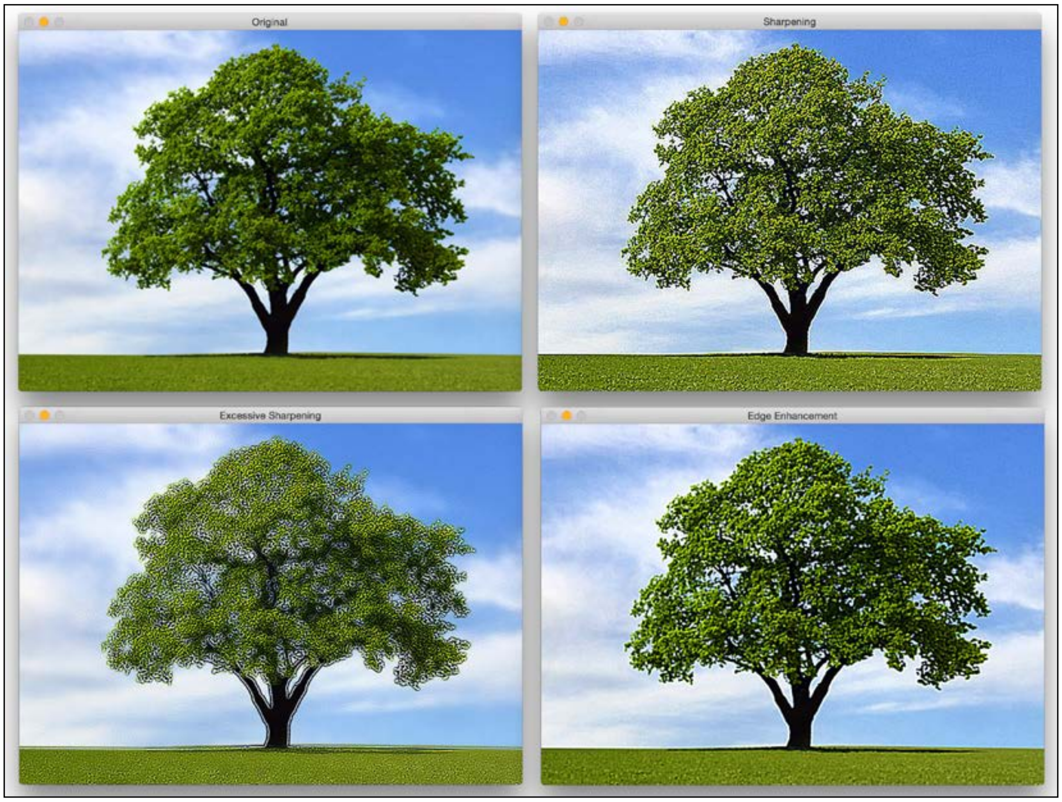
We are reading the image as usual. We are then constructing a motion `blur` kernel. A motion blur kernel averages the pixel values in a particular direction. It's like a directional low pass filter. A 3x3 horizontal motion-blurring kernel would look this:

$$M = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

This will blur the image in a horizontal direction. You can pick any direction and it will work accordingly. The amount of blurring will depend on the size of the kernel. So, if you want to make the image blurrier, just pick a bigger size for the kernel. To see the full effect, we have taken a 15x15 kernel in the preceding code. We then use `filter2D` to apply this kernel to the input image, to obtain the motion-blurred output.

Sharpening

Applying the `sharpening` filter will sharpen the edges in the image. This filter is very useful when we want to enhance the edges in an image that's not crisp. Here are some images to give you an idea of what the image `sharpening` process looks like:



As you can see in the preceding figure, the level of sharpening depends on the type of kernel we use. We have a lot of freedom to customize the kernel here, and each kernel will give you a different kind of sharpening. To just sharpen an image, like we are doing in the top right image in the preceding picture, we would use a kernel like this:

$$M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

If we want to do excessive sharpening, like in the bottom left image, we would use the following kernel:

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -7 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

But the problem with these two kernels is that the output image looks artificially enhanced. If we want our images to look more natural, we would use an `Edge Enhancement` filter. The underlying concept remains the same, but we use an approximate Gaussian kernel to build this filter. It will help us smoothen the image when we enhance the edges, thus making the image look more natural.

Here is the code to achieve the effects applied in the preceding screenshot:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
cv2.imshow('Original', img)

# generating the kernels
kernel_sharpen_1 = np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])
kernel_sharpen_2 = np.array([[1,1,1], [1,-7,1], [1,1,1]])
kernel_sharpen_3 = np.array([[-1,-1,-1,-1,-1],
                             [-1,2,2,2,-1],
                             [-1,2,8,2,-1],
                             [-1,2,2,2,-1],
                             [-1,-1,-1,-1,-1]]) / 8.0

# applying different kernels to the input image
output_1 = cv2.filter2D(img, -1, kernel_sharpen_1)
output_2 = cv2.filter2D(img, -1, kernel_sharpen_2)
output_3 = cv2.filter2D(img, -1, kernel_sharpen_3)

cv2.imshow('Sharpening', output_1)
cv2.imshow('Excessive Sharpening', output_2)
cv2.imshow('Edge Enhancement', output_3)
cv2.waitKey(0)
```

If you noticed, in the preceding code, we didn't divide the first two kernels by a normalizing factor. The reason is because the values inside the kernel already sum up to 1, so we are implicitly dividing the matrices by 1.

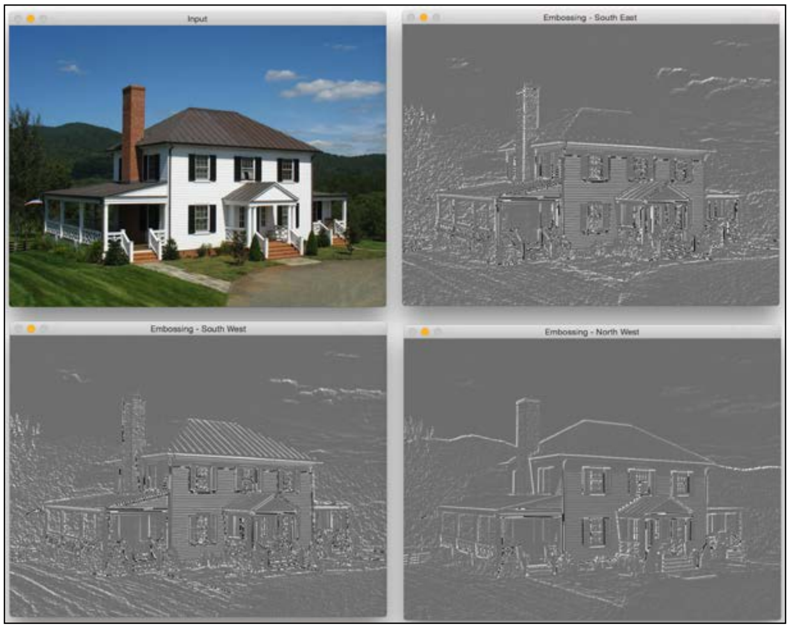
Understanding the pattern

You must have noticed a common pattern in image filtering code examples. We build a kernel and then use `filter2D` to get the desired output. That's exactly what's happening in this code example as well! You can play around with the values inside the kernel and see if you can get different visual effects. Make sure that you normalize the kernel before applying it, or else the image will look too bright because you are artificially increasing the pixel values in the image.

Embossing

An embossing filter will take an image and convert it into an embossed image. We basically take each pixel and replace it with a shadow or a highlight. Let's say we are dealing with a relatively plain region in the image. Here, we need to replace it with plain gray color because there's not much information there. If there is a lot of contrast in a particular region, we will replace it with a white pixel (highlight), or a dark pixel (shadow), depending on the direction in which we are embossing.

This is what it will look like:



Let's take a look at the code and see how to do this:

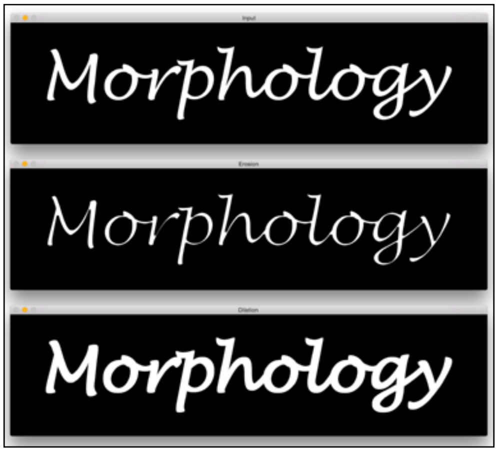
```
import cv2
import numpy as np

img_emboss_input = cv2.imread('input.jpg')

# generating the kernels
kernel_emboss_1 = np.array([[0,-1,-1],
                            [1,0,-1],
                            [1,1,0]])
kernel_emboss_2 = np.array([[-1,-1,0],
                            [-1,0,1],
                            [0,1,1]])
kernel_emboss_3 = np.array([[1,0,0],
                            [0,0,0],
                            [0,0,-1]])

# converting the image to grayscale
gray_img = cv2.cvtColor(img_emboss_input,cv2.COLOR_BGR2GRAY)

# applying the kernels to the grayscale image and adding the offset
output_1 = cv2.filter2D(gray_img, -1, kernel_emboss_1) + 128
output_2 = cv2.filter2D(gray_img, -1, kernel_emboss_2) + 128
output_3 = cv2.filter2D(gray_img, -1, kernel_emboss_3) + 128

cv2.imshow('Input', img_emboss_input)
cv2.imshow('Embossing - South West', output_1)
cv2.imshow('Embossing - South East', output_2)
cv2.imshow('Embossing - North West', output_3)
cv2.waitKey(0)
```

If you run the preceding code, you will see that the output images are embossed. As we can see from the kernels above, we are just replacing the current pixel value with the difference of the neighboring pixel values in a particular direction. The embossing effect is achieved by offsetting all the pixel values in the image by `128`. This operation adds the highlight/shadow effect to the picture.

Erosion and dilation

Erosion and **dilation** are morphological image processing operations. Morphological image processing basically deals with modifying geometric structures in the image. These operations are primarily defined for binary images, but we can also use them on grayscale images. Erosion basically strips out the outermost layer of pixels in a structure, where as dilation adds an extra layer of pixels on a structure.

Let's see what these operations look like:



Following is the code to achieve this:

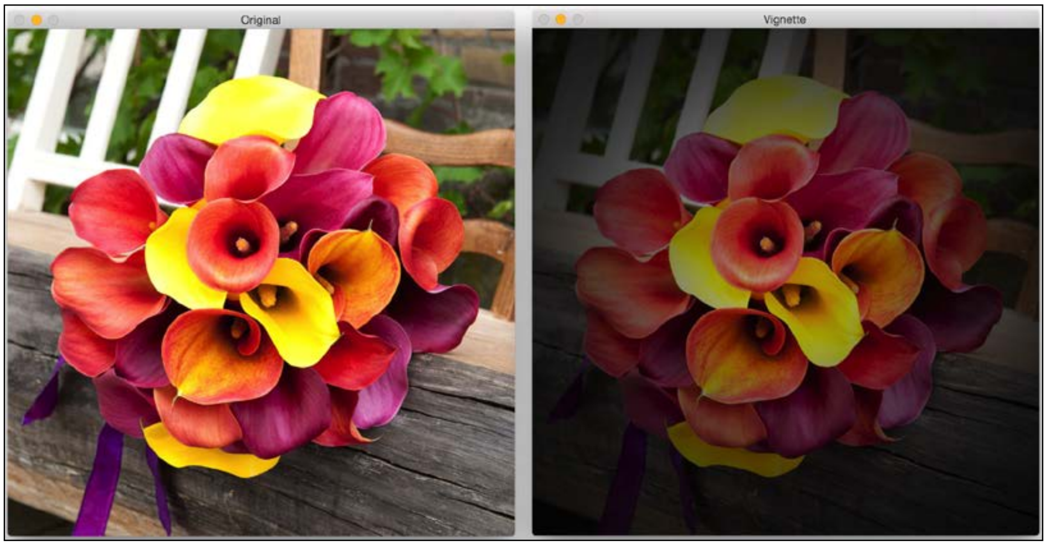
```
import cv2
import numpy as np

img = cv2.imread('input.png', 0)

kernel = np.ones((5,5), np.uint8)

img_erosion = cv2.erode(img, kernel, iterations=1)
img_dilation = cv2.dilate(img, kernel, iterations=1)

cv2.imshow('Input', img)
cv2.imshow('Erosion', img_erosion)
cv2.imshow('Dilation', img_dilation)

cv2.waitKey(0)
```

Afterthought

OpenCV provides functions to directly erode and dilate an image. They are called erode and dilate, respectively. The interesting thing to note is the third argument in these two functions. The number of iterations will determine how much you want to erode/dilate a given image. It basically applies the operation successively to the resultant image. You can take a sample image and play around with this parameter to see what the results look like.

Creating a vignette filter

Using all the information we have, let's see if we can create a nice `vignette` filter. The output will look something like the following:



Here is the code to achieve this effect:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
rows, cols = img.shape[:2]

# generating vignette mask using Gaussian kernels
kernel_x = cv2.getGaussianKernel(cols,200)
kernel_y = cv2.getGaussianKernel(rows,200)
kernel = kernel_y * kernel_x.T
```

```
mask = 255 * kernel / np.linalg.norm(kernel)
output = np.copy(img)

# applying the mask to each channel in the input image
for i in range(3):
    output[:,:,i] = output[:,:,i] * mask

cv2.imshow('Original', img)
cv2.imshow('Vignette', output)
cv2.waitKey(0)
```

What's happening underneath?

The `Vignette` filter basically focuses the brightness on a particular part of the image and the other parts look faded. In order to achieve this, we need to filter out each channel in the image using a Gaussian kernel. OpenCV provides a function to do this, which is called `getGaussianKernel`. We need to build a 2D kernel whose size matches the size of the image. The second parameter of the function, `getGaussianKernel`, is interesting. It is the standard deviation of the Gaussian and it controls the radius of the bright central region. You can play around with this parameter and see how it affects the output.

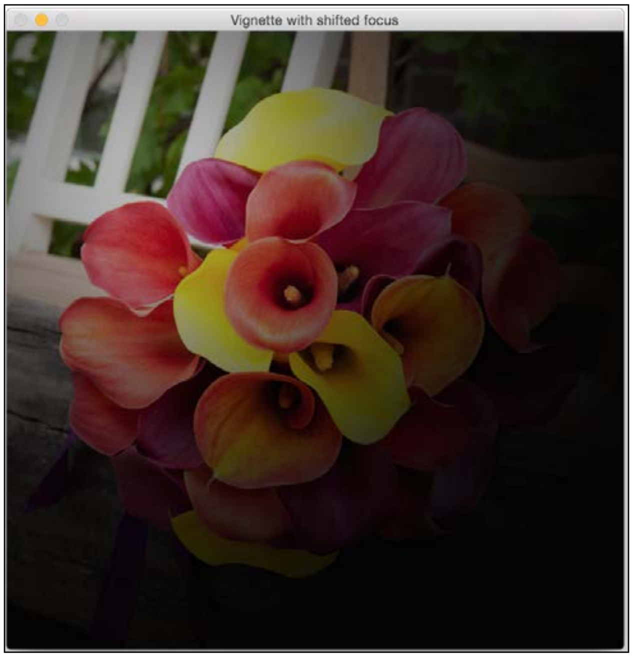
Once we build the 2D kernel, we need to build a mask by normalizing this kernel and scaling it up, as shown in the following line:

```
mask = 255 * kernel / np.linalg.norm(kernel)
```

This is an important step because if you don't scale it up, the image will look black. This happens because all the pixel values will be close to `0` after you superimpose the mask on the input image. After this, we iterate through all the color channels and apply the mask to each channel.

How do we move the focus around?

We now know how to create a `vignette` filter that focuses on the center of the image. Let's say we want to achieve the same `vignette` effect, but we want to focus on a different region in the image, as shown in the following figure:



All we need to do is build a bigger Gaussian kernel and make sure that the peak coincides with the region of interest. Following is the code to achieve this:

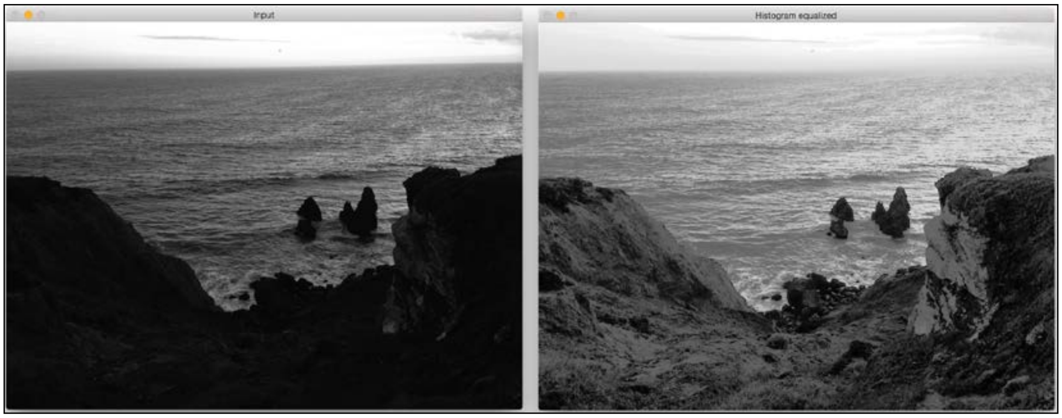
```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
rows, cols = img.shape[:2]

# generating vignette mask using Gaussian kernels
kernel_x = cv2.getGaussianKernel(int(1.5*cols),200)
kernel_y = cv2.getGaussianKernel(int(1.5*rows),200)
kernel = kernel_y * kernel_x.T
mask = 255 * kernel / np.linalg.norm(kernel)
mask = mask[int(0.5*rows):, int(0.5*cols):]
output = np.copy(img)
```

```
# applying the mask to each channel in the input image
for i in range(3):
    output[:,:,i] = output[:,:,i] * mask

cv2.imshow('Input', img)
cv2.imshow('Vignette with shifted focus', output)

cv2.waitKey(0)
```

Enhancing the contrast in an image

Whenever we capture images in low-light conditions, the images turn out to be dark. This typically happens when you capture images in the evening or in a dimly lit room. You must have seen this happen many times! The reason this happens is because the pixel values tend to concentrate near 0 when we capture the images under such conditions. When this happens, a lot of details in the image are not clearly visible to the human eye. The human eye likes contrast, and so we need to adjust the contrast to make the image look nice and pleasant. A lot of cameras and photo applications implicitly do this already. We use a process called **Histogram Equalization** to achieve this.

To give an example, this is what it looks like before and after contrast enhancement:



As we can see here, the input image on the left is really dark. To rectify this, we need to adjust the pixel values so that they are spread across the entire spectrum of values, that is, between 0 and 255.

Following is the code for adjusting the pixel values:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg', 0)

# equalize the histogram of the input image
histeq = cv2.equalizeHist(img)

cv2.imshow('Input', img)
cv2.imshow('Histogram equalized', histeq)
cv2.waitKey(0)
```

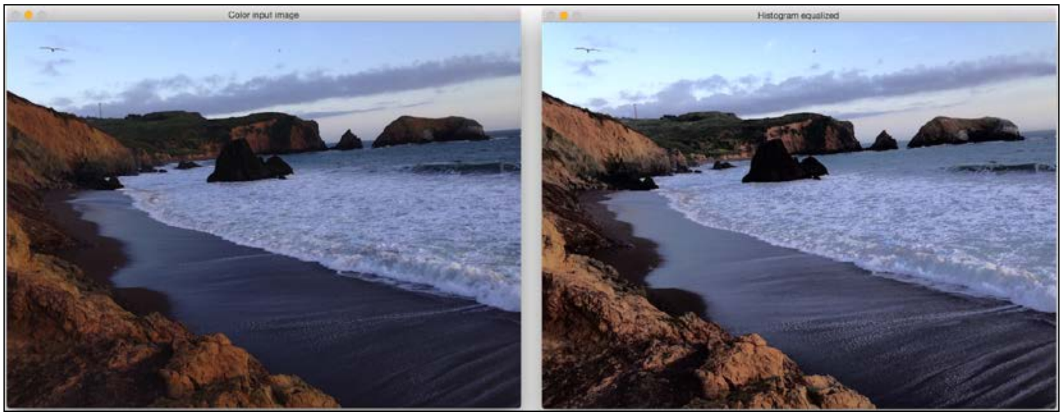
Histogram equalization is applicable to grayscale images. OpenCV provides a function, `equalizeHist`, to achieve this effect. As we can see here, the code is pretty straightforward, where we read the image and equalize its histogram to adjust the contrast of the image.

How do we handle color images?

Now that we know how to equalize the histogram of a grayscale image, you might be wondering how to handle color images. The thing about histogram equalization is that it's a nonlinear process. So, we cannot just separate out the three channels in an RGB image, equalize the histogram separately, and combine them later to form the output image. The concept of histogram equalization is only applicable to the intensity values in the image. So, we have to make sure not to modify the color information when we do this.

In order to handle the histogram equalization of color images, we need to convert it to a color space where intensity is separated from the color information. YUV is a good example of such a color space. Once we convert it to YUV, we just need to equalize the Y-channel and combine it with the other two channels to get the output image.

Following is an example of what it looks like:



Here is the code to achieve histogram equalization for color images:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')

img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)

# equalize the histogram of the Y channel
img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])

# convert the YUV image back to RGB format
img_output = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)

cv2.imshow('Color input image', img)
cv2.imshow('Histogram equalized', img_output)

cv2.waitKey(0)
```

Summary

In this chapter, we learned how to use image filters to apply cool visual effects to images. We discussed the fundamental image processing operators and how we can use them to build various things. We learnt how to detect edges using various methods. We understood the importance of 2D convolution and how we can use it in different scenarios. We discussed how to smoothen, motion-blur, sharpen, emboss, erode, and dilate an image. We learned how to create a vignette filter, and how we can change the region of focus as well. We discussed contrast enhancement and how we can use histogram equalization to achieve it. In the next chapter, we will discuss how to cartoonize a given image.

2
Cartoonizing an Image

In this chapter, we are going to learn how to convert an image into a cartoon-like image. We will learn how to access the webcam and take keyboard/mouse inputs during a live video stream. We will also learn about some advanced image filters and see how we can use them to cartoonize an image.

By the end of this chapter, you will know:

- How to access the webcam
- How to take keyboard and mouse inputs during a live video stream
- How to create an interactive application
- How to use advanced image filters
- How to cartoonize an image

Accessing the webcam

We can build very interesting applications using the live video stream from the webcam. OpenCV provides a video capture object which handles everything related to opening and closing of the webcam. All we need to do is create that object and keep reading frames from it.

The following code will open the webcam, capture the frames, scale them down by a factor of 2, and then display them in a window. You can press the *Esc* key to exit.

```
import cv2

cap = cv2.VideoCapture(0)

# Check if the webcam is opened correctly
if not cap.isOpened():
    raise IOError("Cannot open webcam")

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=0.5, fy=0.5, interpolation=cv2.
INTER_AREA)
    cv2.imshow('Input', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Under the hood

As we can see in the preceding code, we use OpenCV's `VideoCapture` function to create the video capture object cap. Once it's created, we start an infinite loop and keep reading frames from the webcam until we encounter a keyboard interrupt. In the first line within the while loop, we have the following line:

```
ret, frame = cap.read()
```

Here, `ret` is a Boolean value returned by the `read` function, and it indicates whether or not the frame was captured successfully. If the frame is captured correctly, it's stored in the variable `frame`. This loop will keep running until we press the *Esc* key. So we keep checking for a keyboard interrupt in the following line:

```
if c == 27:
```

As we know, the ASCII value of *Esc* is 27. Once we encounter it, we break the loop and release the video capture object. The line `cap.release()` is important because it gracefully closes the webcam.

Keyboard inputs

Now that we know how to capture a live video stream from the webcam, let's see how to use the keyboard to interact with the window displaying the video stream.

```
import argparse

import cv2

def argument_parser():
    parser = argparse.ArgumentParser(description="Change color space
of the \
            input video stream using keyboard controls. The control
keys are: \
            Grayscale - 'g', YUV - 'y', HSV - 'h'")
    return parser

if __name__=='__main__':
    args = argument_parser().parse_args()

    cap = cv2.VideoCapture(0)

    # Check if the webcam is opened correctly
    if not cap.isOpened():
        raise IOError("Cannot open webcam")

    cur_char = -1
    prev_char = -1

    while True:
        # Read the current frame from webcam
        ret, frame = cap.read()

        # Resize the captured image
        frame = cv2.resize(frame, None, fx=0.5, fy=0.5,
interpolation=cv2.INTER_AREA)

        c = cv2.waitKey(1)

        if c == 27:
            break

        if c > -1 and c != prev_char:
            cur_char = c
        prev_char = c

        if cur_char == ord('g'):
            output = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        elif cur_char == ord('y'):
            output = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV)

        elif cur_char == ord('h'):
            output = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        else:
            output = frame

        cv2.imshow('Webcam', output)

    cap.release()
    cv2.destroyAllWindows()
```

Interacting with the application

This program will display the input video stream and wait for the keyboard input to change the color space. If you run the previous program, you will see the window displaying the input video stream from the webcam. If you press *G*, you will see that the color space of the input stream gets converted to grayscale. If you press *Y*, the input stream will be converted to YUV color space. Similarly, if you press *H*, you will see the image being converted to HSV color space.

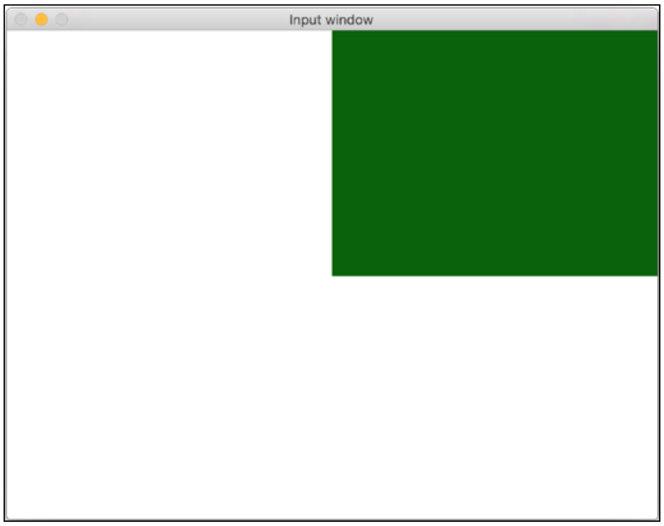
As we know, we use the function `waitKey()` to listen to the keyboard events. As and when we encounter different keystrokes, we take appropriate actions. The reason we are using the function `ord()` is because `waitKey()` returns the ASCII value of the keyboard input; thus, we need to convert the characters into their ASCII form before checking their values.

Mouse inputs

In this section, we will see how to use the mouse to interact with the display window. Let's start with something simple. We will write a program that will detect the quadrant in which the mouse click was detected. Once we detect it, we will highlight that quadrant.

```
import cv2
import numpy as np

def detect_quadrant(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        if x > width/2:
            if y > height/2:
                point_top_left = (int(width/2), int(height/2))
                point_bottom_right = (width-1, height-1)
            else:
                point_top_left = (int(width/2), 0)
                point_bottom_right = (width-1, int(height/2))
        else:
            if y > height/2:
                point_top_left = (0, int(height/2))
                point_bottom_right = (int(width/2), height-1)
            else:
                point_top_left = (0, 0)
                point_bottom_right = (int(width/2), int(height/2))

        cv2.rectangle(img, (0,0), (width-1,height-1), (255,255,255),
-1)
        cv2.rectangle(img, point_top_left, point_bottom_right,
(0,100,0), -1)

if __name__=='__main__':
    width, height = 640, 480
    img = 255 * np.ones((height, width, 3), dtype=np.uint8)
    cv2.namedWindow('Input window')
    cv2.setMouseCallback('Input window', detect_quadrant)

    while True:
        cv2.imshow('Input window', img)
        c = cv2.waitKey(10)
        if c == 27:
            break

    cv2.destroyAllWindows()
```

The output will look something like the following image:



What's happening underneath?

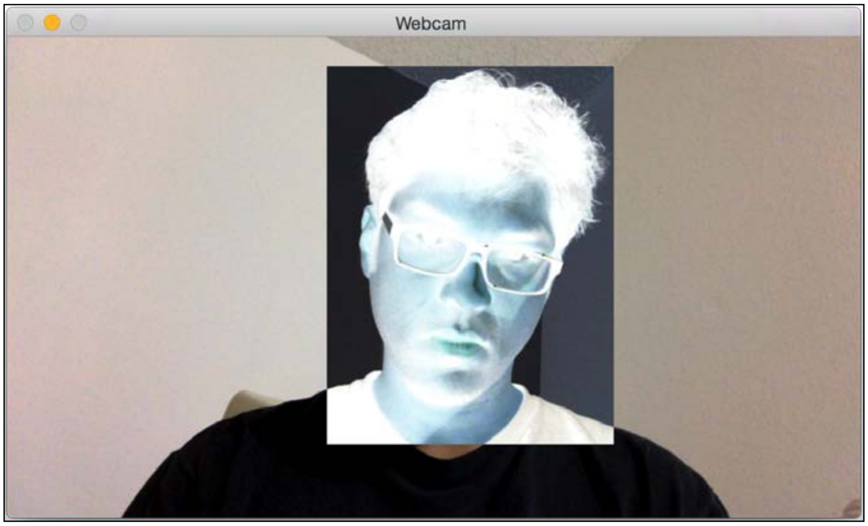
Let's start with the main function in this program. We create a white image on which we are going to click using the mouse. We then create a named window and bind the mouse callback function to this window. Mouse callback function is basically the function that will be called when a mouse event is detected. There are many kinds of mouse events such as clicking, double-clicking, dragging, and so on. In our case, we just want to detect a mouse click. In the function `detect_quadrant`, we check the first input argument event to see what action was performed. OpenCV provides a set of predefined events, and we can call them using specific keywords. If you want to see a list of all the mouse events, you can go to the Python shell and type the following:

```
>>> import cv2
>>> print [x for x in dir(cv2) if x.startswith('EVENT')]
```

The second and third arguments in the function `detect_quadrant` provide the X and Y coordinates of the mouse click event. Once we know these coordinates, it's pretty straightforward to determine what quadrant it's in. With this information, we just go ahead and draw a rectangle with the specified color, using `cv2.rectangle()`. This is a very handy function that takes the top left point and the bottom right point to draw a rectangle on an image with the specified color.

Interacting with a live video stream

Let's see how we can use the mouse to interact with live video stream from the webcam. We can use the mouse to select a region and then apply the "negative film" effect on that region, as shown next:



In the following program, we will capture the video stream from the webcam, select a region of interest with the mouse, and then apply the effect:

```
import cv2
import numpy as np

def draw_rectangle(event, x, y, flags, params):
    global x_init, y_init, drawing, top_left_pt, bottom_right_pt

    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        x_init, y_init = x, y

    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing:
            top_left_pt = (min(x_init, x), min(y_init, y))
            bottom_right_pt = (max(x_init, x), max(y_init, y))
            img[y_init:y, x_init:x] = 255 - img[y_init:y, x_init:x]

    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        top_left_pt = (min(x_init, x), min(y_init, y))
        bottom_right_pt = (max(x_init, x), max(y_init, y))
        img[y_init:y, x_init:x] = 255 - img[y_init:y, x_init:x]

if __name__=='__main__':
    drawing = False
    top_left_pt, bottom_right_pt = (-1,-1), (-1,-1)

    cap = cv2.VideoCapture(0)

    # Check if the webcam is opened correctly
    if not cap.isOpened():
        raise IOError("Cannot open webcam")

    cv2.namedWindow('Webcam')
    cv2.setMouseCallback('Webcam', draw_rectangle)

    while True:
        ret, frame = cap.read()
        img = cv2.resize(frame, None, fx=0.5, fy=0.5,
interpolation=cv2.INTER_AREA)
        (x0,y0), (x1,y1) = top_left_pt, bottom_right_pt
        img[y0:y1, x0:x1] = 255 - img[y0:y1, x0:x1]
```

```
            cv2.imshow('Webcam', img)

            c = cv2.waitKey(1)
            if c == 27:
                break

        cap.release()
        cv2.destroyAllWindows()
```

If you run the preceding program, you will see a window displaying the video stream. You can just draw a rectangle on the window using your mouse and you will see that region being converted to its "negative".

How did we do it?

As we can see in the main function of the program, we initialize a video capture object. We then bind the function `draw_rectangle` with the mouse callback in the following line:

```
cv2.setMouseCallback('Webcam', draw_rectangle)
```

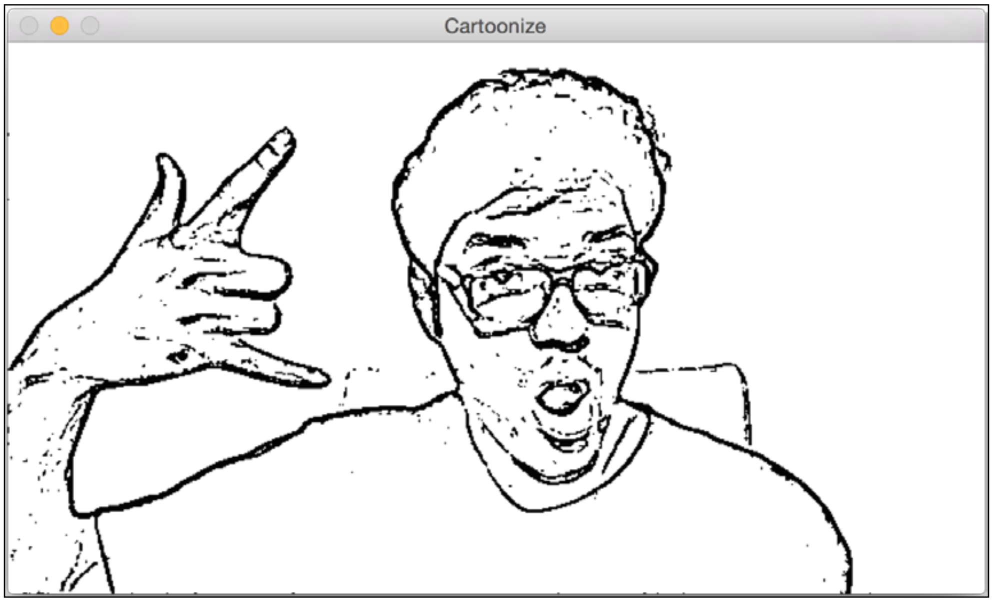
We then start an infinite loop and start capturing the video stream. Let's see what is happening in the function `draw_rectangle`. Whenever we draw a rectangle using the mouse, we basically have to detect three types of mouse events: mouse click, mouse movement, and mouse button release. This is exactly what we do in this function. Whenever we detect a mouse click event, we initialize the top left point of the rectangle. As we move the mouse, we select the region of interest by keeping the current position as the bottom right point of the rectangle.

Once we have the region of interest, we just invert the pixels to apply the "negative film" effect. We subtract the current pixel value from 255 and this gives us the desired effect. When the mouse movement stops and button-up event is detected, we stop updating the bottom right position of the rectangle. We just keep displaying this image until another mouse click event is detected.

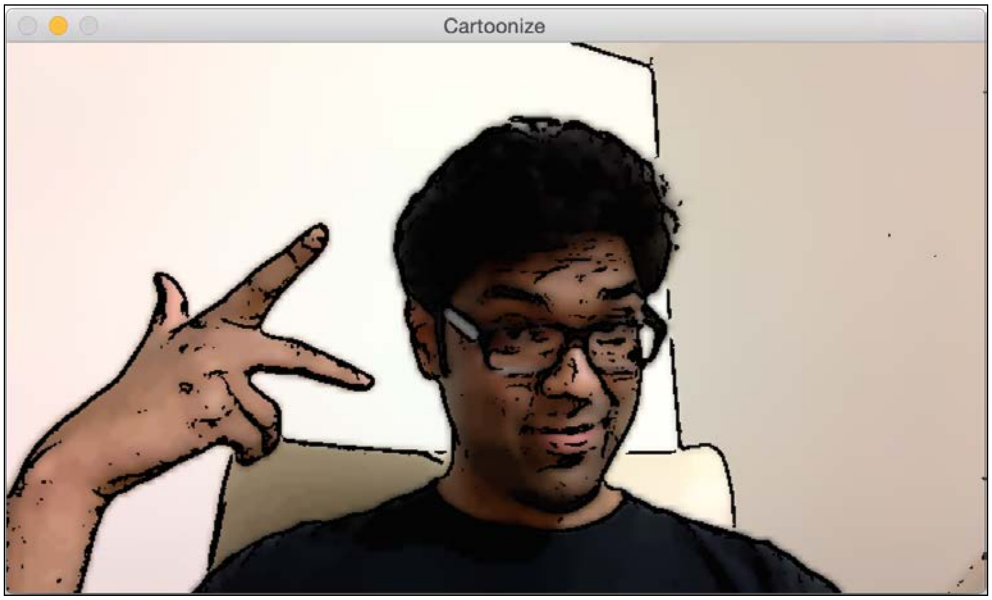
Cartoonizing an image

Now that we know how to handle the webcam and keyboard/mouse inputs, let's go ahead and see how to convert a picture into a cartoon-like image. We can either convert an image into a sketch or a colored cartoon image.

Following is an example of what a sketch will look like:



If you apply the cartoonizing effect to the color image, it will look something like this next image:



Let's see how to achieve this:

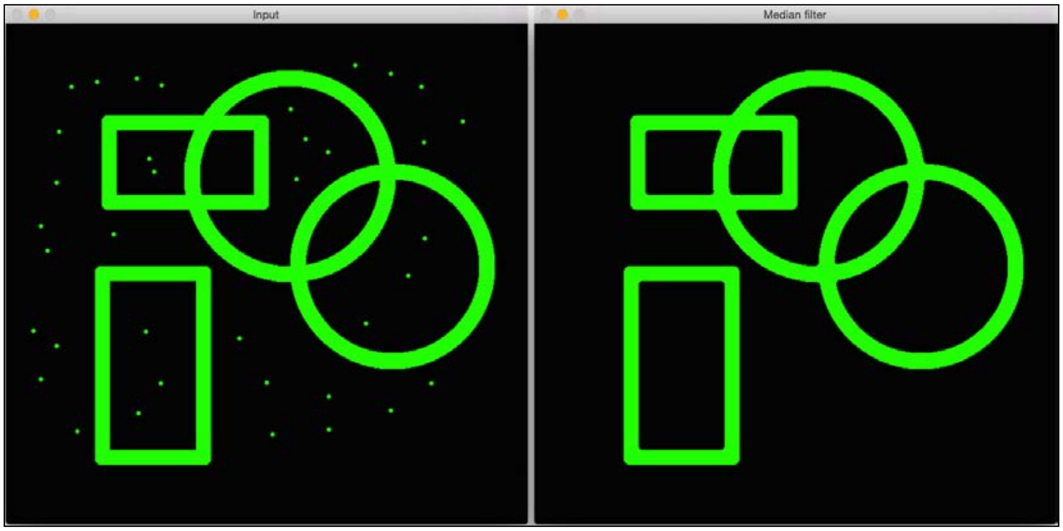
```
import cv2
import numpy as np

def cartoonize_image(img, ds_factor=4, sketch_mode=False):
    # Convert image to grayscale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply median filter to the grayscale image
    img_gray = cv2.medianBlur(img_gray, 7)

    # Detect edges in the image and threshold it
    edges = cv2.Laplacian(img_gray, cv2.CV_8U, ksize=5)
    ret, mask = cv2.threshold(edges, 100, 255, cv2.THRESH_BINARY_INV)

    # 'mask' is the sketch of the image
    if sketch_mode:
        return cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)

    # Resize the image to a smaller size for faster computation
    img_small = cv2.resize(img, None, fx=1.0/ds_factor, fy=1.0/ds_
factor, interpolation=cv2.INTER_AREA)
    num_repetitions = 10
    sigma_color = 5
    sigma_space = 7
    size = 5

    # Apply bilateral filter the image multiple times
    for i in range(num_repetitions):
        img_small = cv2.bilateralFilter(img_small, size, sigma_color,
sigma_space)

    img_output = cv2.resize(img_small, None, fx=ds_factor, fy=ds_
factor, interpolation=cv2.INTER_LINEAR)

    dst = np.zeros(img_gray.shape)

    # Add the thick boundary lines to the image using 'AND' operator
    dst = cv2.bitwise_and(img_output, img_output, mask=mask)
    return dst
```

```
if __name__=='__main__':
    cap = cv2.VideoCapture(0)

    cur_char = -1
    prev_char = -1

    while True:
        ret, frame = cap.read()
        frame = cv2.resize(frame, None, fx=0.5, fy=0.5, 
interpolation=cv2.INTER_AREA)

        c = cv2.waitKey(1)
        if c == 27:
            break

        if c > -1 and c != prev_char:
            cur_char = c
        prev_char = c

        if cur_char == ord('s'):
            cv2.imshow('Cartoonize', cartoonize_image(frame, sketch_
mode=True))
        elif cur_char == ord('c'):
            cv2.imshow('Cartoonize', cartoonize_image(frame, sketch_
mode=False))
        else:
            cv2.imshow('Cartoonize', frame)

    cap.release()
    cv2.destroyAllWindows()
```

Deconstructing the code

When you run the preceding program, you will see a window with a video stream from the webcam. If you press *S*, the video stream will change to sketch mode and you will see its pencil-like outline. If you press *C*, you will see the color-cartoonized version of the input stream. If you press any other key, it will return to the normal mode.

Let's look at the function `cartoonize_image` and see how we did it. We first convert the image to a grayscale image and run it through a median filter. Median filters are very good at removing salt and pepper noise. This is the kind of noise where you see isolated black or white pixels in the image. It is common in webcams and mobile cameras, so we need to filter it out before we proceed further. To give an example, look at the following images:



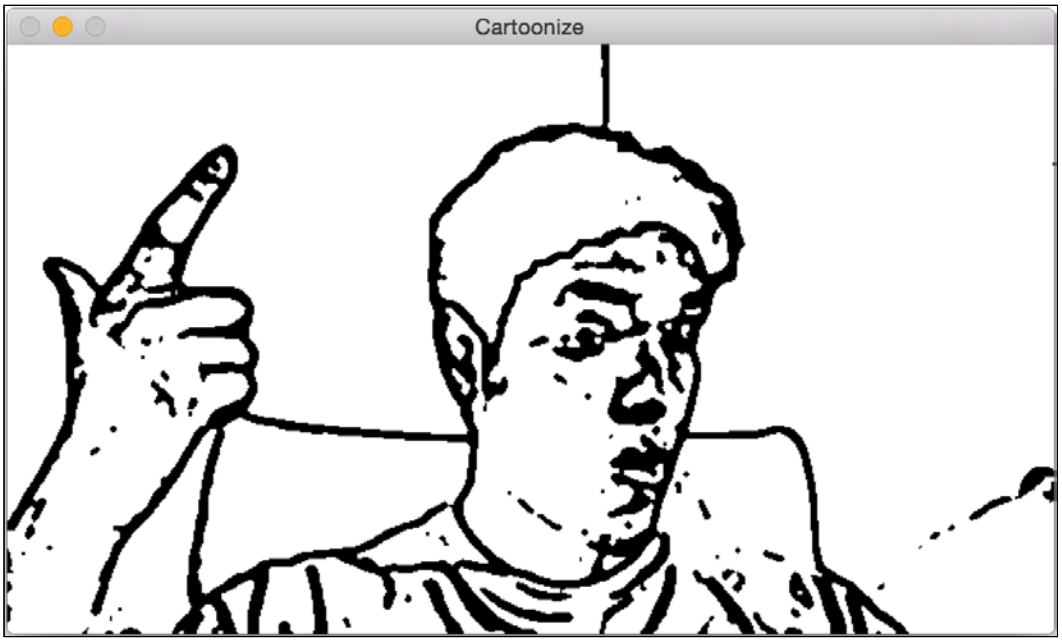
As we see in the input image, there are a lot of isolated green pixels. They are lowering the quality of the image and we need to get rid of them. This is where the median filter comes in handy. We just look at the NxN neighborhood around each pixel and pick the median value of those numbers. Since the isolated pixels in this case have high values, taking the median value will get rid of these values and also smoothen the image. As you can see in the output image, the median filter got rid of all those isolated pixels and the image looks clean. Following is the code to do it:

```
import cv2
import numpy as np

img = cv2.imread('input.png')
output = cv2.medianBlur(img, 7)
cv2.imshow('Input', img)
cv2.imshow('Median filter', output)
cv2.waitKey()
```

The code is pretty straightforward. We just use the function medianBlur to apply the median filter to the input image. The second argument in this function specifies the size of the kernel we are using. The size of the kernel is related to the neighborhood size that we need to consider. You can play around with this parameter and see how it affects the output.

Coming back to cartoonize_image, we proceed to detect the edges on the grayscale image. We need to know where the edges are so that we can create the pencil-line effect. Once we detect the edges, we threshold them so that things become black and white, both literally and metaphorically!

In the next step, we check if the sketch mode is enabled. If it is, then we just convert it into a color image and return it. What if we want the lines to be thicker? Let's say we want to see something like the following image:

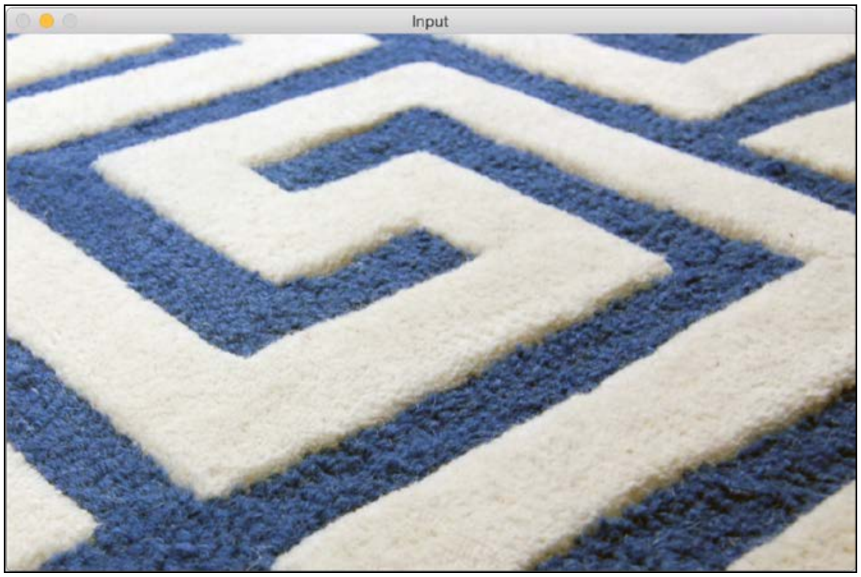


As you can see, the lines are thicker than before. To achieve this, replace the if code block with the following piece of code:

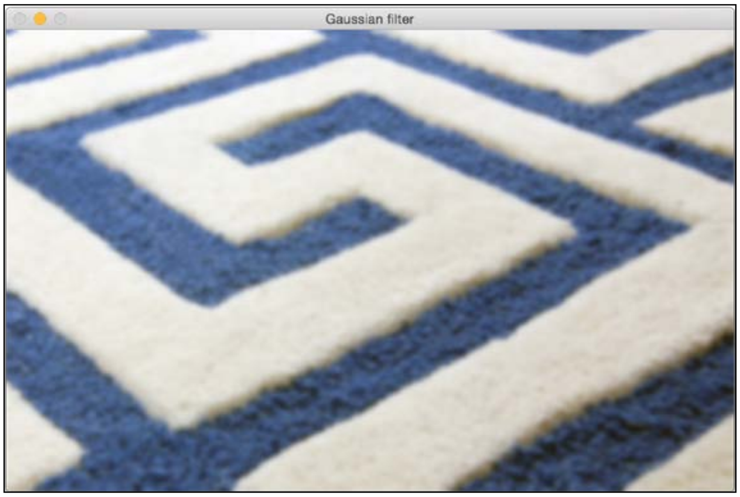
```
if sketch_mode:
    img_sketch = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
    kernel = np.ones((3,3), np.uint8)
    img_eroded = cv2.erode(img_sketch, kernel, iterations=1)
    return cv2.medianBlur(img_eroded, 5)
```

We are using the erode function with a 3x3 kernel here. The reason we have this in place is because it gives us a chance to play with the thickness of the line drawing. Now you might ask that if we want to increase the thickness of something, shouldn't we be using dilation? Well, the reasoning is right, but there is a small twist here. Note that the foreground is black and the background is white. Erosion and dilation treat white pixels as foreground and black pixels as background. So if we want to increase the thickness of the black foreground, we need to use erosion. After we apply erosion, we just use the median filter to clear out the noise and get the final output.

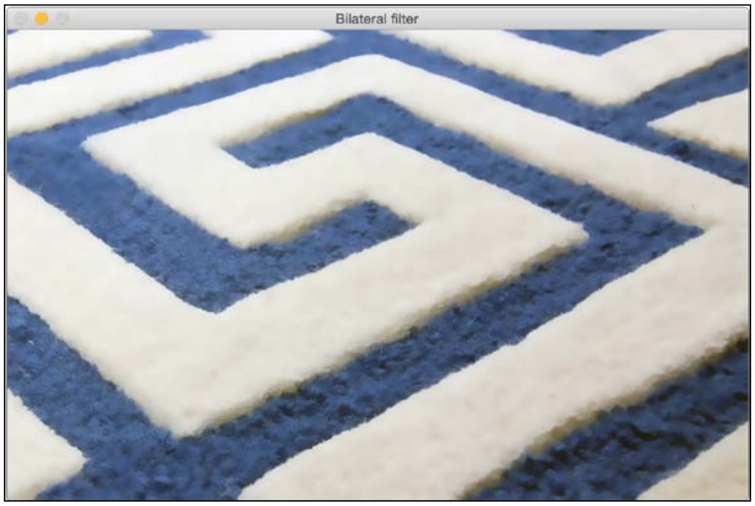
In the next step, we use bilateral filtering to smoothen the image. Bilateral filtering is an interesting concept and its performance is much better than a Gaussian filter. The good thing about bilateral filtering is that it preserves the edges, whereas the Gaussian filter smoothens everything out equally. To compare and contrast, let's look at the following input image:



Let's apply the Gaussian filter to the previous image:



Now, let's apply the bilateral filter to the input image:



As you can see, the quality is better if we use the bilateral filter. The image looks smooth and the edges look nice and sharp! The code to achieve this is given next:

```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
```

```
img_gaussian = cv2.GaussianBlur(img, (13,13), 0)
img_bilateral = cv2.bilateralFilter(img, 13, 70, 50)

cv2.imshow('Input', img)
cv2.imshow('Gaussian filter', img_gaussian)
cv2.imshow('Bilateral filter', img_bilateral)
cv2.waitKey()
```

If you closely observe the two outputs, you can see that the edges in the Gaussian filtered image look blurred. Usually, we just want to smoothen the rough areas in the image and keep the edges intact. This is where the bilateral filter comes in handy. The Gaussian filter just looks at the immediate neighborhood and averages the pixel values using a Gaussian kernel. The bilateral filter takes this concept to the next level by averaging only those pixels that are similar to each other in intensity. It also takes a color neighborhood metric to see if it can replace the current pixel that is similar in intensity as well. If you look the function call:

```
img_small = cv2.bilateralFilter(img_small, size, sigma_color,
sigma_space)
```

The last two arguments here specify the color and space neighborhood. This is the reason the edges look crisp in the output of the bilateral filter. We run this filter multiple times on the image to smoothen it out, to make it look like a cartoon. We then superimpose the pencil-like mask on top of this color image to create a cartoon-like effect.

Summary

In this chapter, we learnt how to access the webcam. We discussed how to take the keyboard and mouse inputs during live video stream. We used this knowledge to create an interactive application. We discussed the median and bilateral filters, and talked about the advantages of the bilateral filter over the Gaussian filter. We used all these principles to convert the input image into a sketch-like image, and then cartoonized it.

In the next chapter, we will learn how to detect different body parts in static images as well as in live videos.

3

Detecting and Tracking Different Body Parts

In this chapter, we are going to learn how to detect and track different body parts in a live video stream. We will start by discussing the face detection pipeline and how it's built from the ground up. We will learn how to use this framework to detect and track other body parts, such as eyes, ears, mouth, and nose.

By the end of this chapter, you will know:

- How to use Haar cascades
- What are integral images
- What is adaptive boosting
- How to detect and track faces in a live video stream
- How to detect and track eyes in a live video stream
- How to automatically overlay sunglasses on top of a person's face
- How to detect ears, nose, and mouth
- How to detect pupils using shape analysis

Using Haar cascades to detect things

When we say Haar cascades, we are actually talking about cascade classifiers based on Haar features. To understand what this means, we need to take a step back and understand why we need this in the first place. Back in 2001, Paul Viola and Michael Jones came up with a very effective object detection method in their seminal paper. It has become one of the major landmarks in the field of machine learning.

In their paper, they have described a machine learning technique where a boosted cascade of simple classifiers is used to get an overall classifier that performs really well. This way, we can circumvent the process of building a single complex classifier that performs with high accuracy. The reason this is so amazing is because building a robust single-step classifier is a computationally intensive process. Besides, we need a lot of training data to build such a classifier. The model ends up becoming complex and the performance might not be up to the mark.

Let's say we want to detect an object like, say, a pineapple. To solve this, we need to build a machine learning system that will learn what a pineapple looks like. It should be able to tell us if an unknown image contains a pineapple or not. To achieve something like this, we need to train our system. In the realm of machine learning, we have a lot of methods available to train a system. It's a lot like training a dog, except that it won't fetch the ball for you! To train our system, we take a lot of pineapple and non-pineapple images, and then feed them into the system. Here, pineapple images are called positive images and the non-pineapple images are called negative images.

As far as the training is concerned, there are a lot of routes available. But all the traditional techniques are computationally intensive and result in complex models. We cannot use these models to build a real time system. Hence, we need to keep the classifier simple. But if we keep the classifier simple, it will not be accurate. The trade off between speed and accuracy is common in machine learning. We overcome this problem by building a set of simple classifiers and then cascading them together to form a unified classifier that's robust. To make sure that the overall classifier works well, we need to get creative in the cascading step. This is one of the main reasons why the **Viola-Jones** method is so effective.

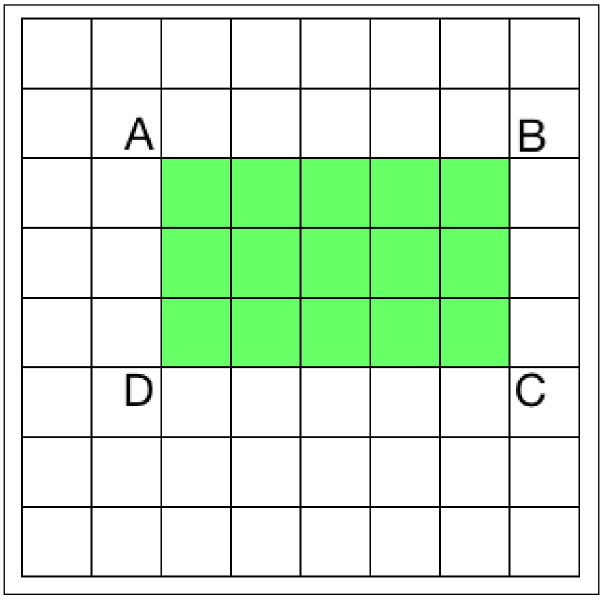
Coming to the topic of face detection, let's see how to train a system to detect faces. If we want to build a machine learning system, we first need to extract features from all the images. In our case, the machine learning algorithms will use these features to learn what a face looks like. We use Haar features to build our feature vectors. Haar features are simple summations and differences of patches across the image. We do this at multiple image sizes to make sure our system is scale invariant.

> If you are curious, you can learn more about the formulation at `http://www.cs.ubc.ca/~lowe/425/slides/13-ViolaJones.pdf`

Once we extract these features, we pass it through a cascade of classifiers. We just check all the different rectangular sub-regions and keep discarding the ones that don't have faces in them. This way, we arrive at the final answer quickly to see if a given rectangle contains a face or not.

What are integral images?

If we want to compute Haar features, we will have to compute the summations of many different rectangular regions within the image. If we want to effectively build the feature set, we need to compute these summations at multiple scales. This is a very expensive process! If we want to build a real time system, we cannot spend so many cycles in computing these sums. So we use something called integral images.



To compute the sum of any rectangle in the image, we don't need to go through all the elements in that rectangular area. Let's say AP indicates the sum of all the elements in the rectangle formed by the top left point and the point P in the image as the two diagonally opposite corners. So now, if we want to compute the area of the rectangle ABCD, we can use the following formula:

Area of the rectangle ABCD = AC – (AB + AD - AA)

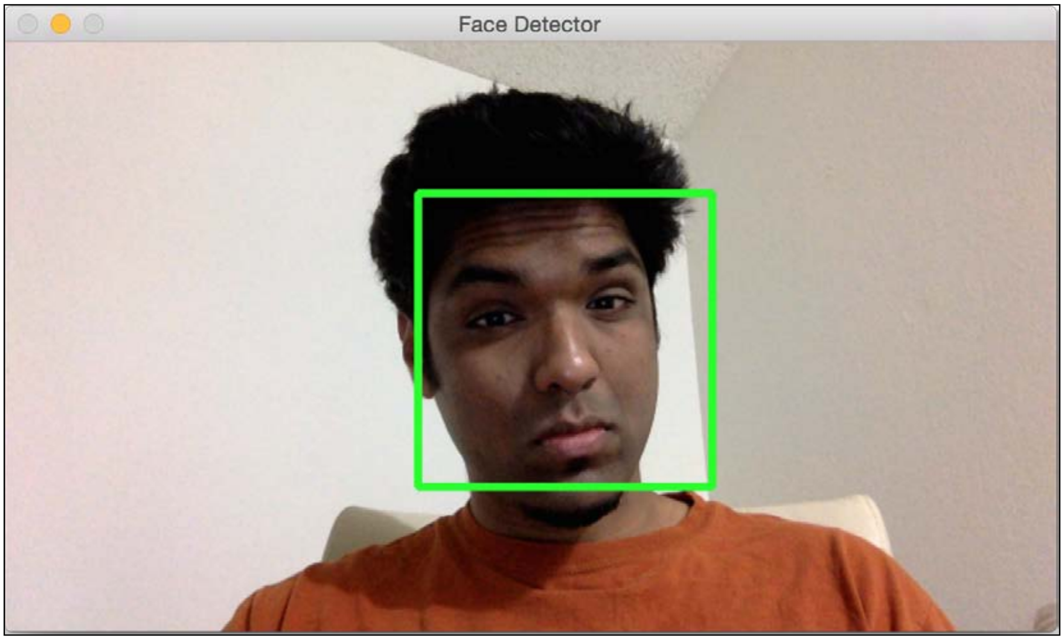
Why do we care about this particular formula? As we discussed earlier, extracting Haar features includes computing the areas of a large number of rectangles in the image at multiple scales. A lot of those computations are repetitive and the overall process is very slow. In fact, it is so slow that we cannot afford to run anything in real time. That's the reason we use this formulation! The good thing about this approach is that we don't have to recalculate anything. All the values for the areas on the right hand side of this equation are already available. So we just use them to compute the area of any given rectangle and extract the features.

Detecting and tracking faces

OpenCV provides a nice face detection framework. We just need to load the cascade file and use it to detect the faces in an image. Let's see how to do it:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in face_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)

    cv2.imshow('Face Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

If you run the above code, it will look something like the following image:

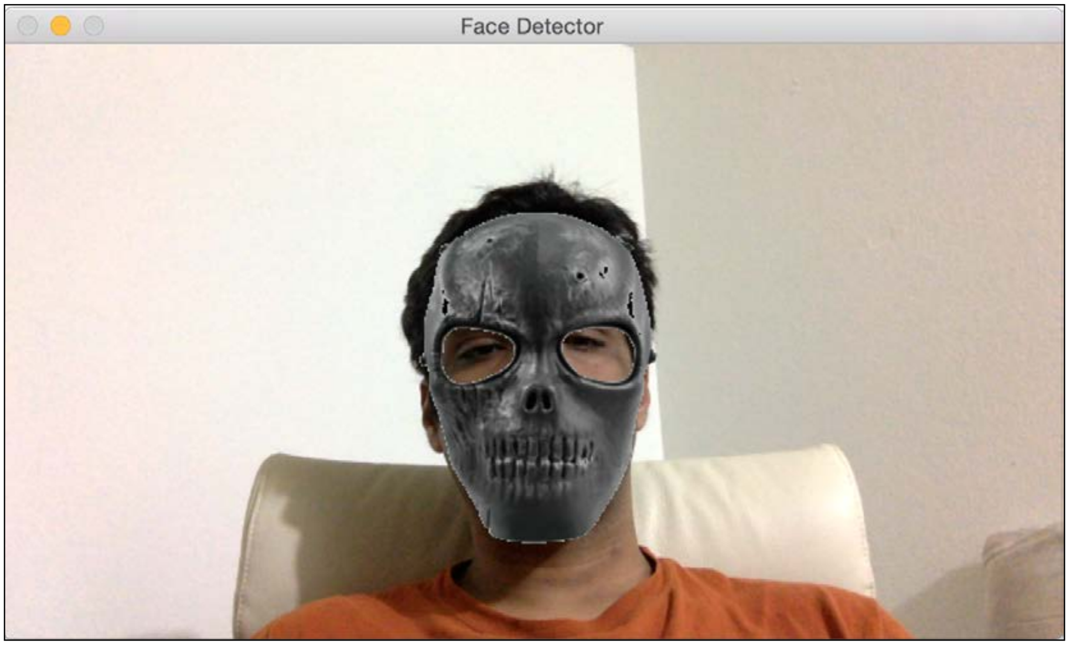


Understanding it better

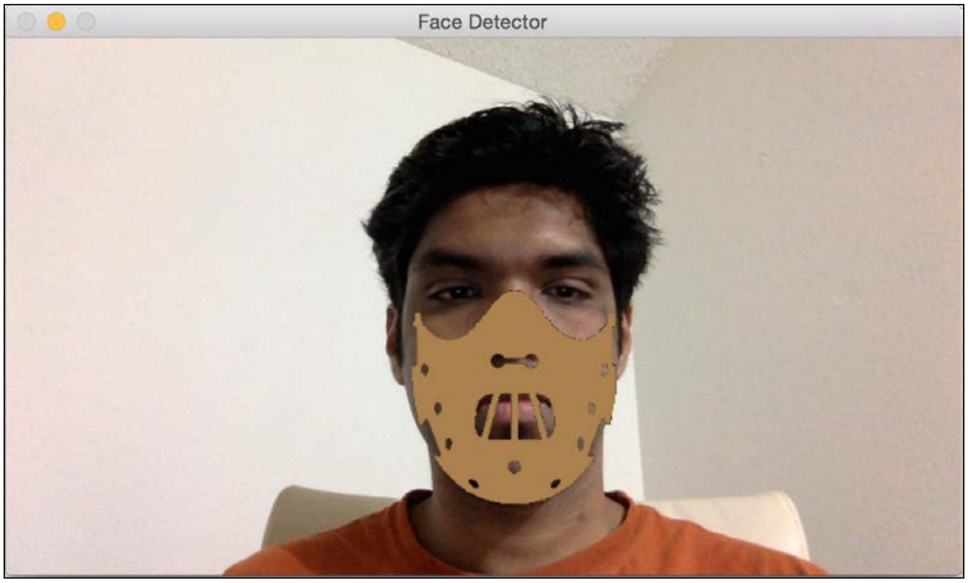
We need a classifier model that can be used to detect the faces in an image. OpenCV provides an xml file that can be used for this purpose. We use the function `CascadeClassifier` to load the xml file. Once we start capturing the input frames from the webcam, we convert it to grayscale and use the function `detectMultiScale` to get the bounding boxes for all the faces in the current image. The second argument in this function specifies the jump in the scaling factor. As in, if we don't find an image in the current scale, the next size to check will be, in our case, 1.3 times bigger than the current size. The last parameter is a threshold that specifies the number of adjacent rectangles needed to keep the current rectangle. It can be used to increase the robustness of the face detector.

Fun with faces

Now that we know how to detect and track faces, let's have some fun with it. When we capture a video stream from the webcam, we can overlay funny masks on top of our faces. It will look something like this next image:



If you are a fan of Hannibal, you can try this next one:



Let's look at the code to see how to overlay the skull mask on top of the face in the input video stream:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')

face_mask = cv2.imread('mask_hannibal.png')
h_mask, w_mask = face_mask.shape[:2]

if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier
xml file')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in face_rects:
        if h > 0 and w > 0:
            # Adjust the height and weight parameters depending
on the sizes and the locations. You need to play around with
these to make sure you get it right.
            h, w = int(1.4*h), int(1.0*w)
            y -= 0.1*h

            # Extract the region of interest from the image
            frame_roi = frame[y:y+h, x:x+w]
            face_mask_small = cv2.resize(face_mask, (w, h),
interpolation=cv2.INTER_AREA)

            # Convert color image to grayscale and threshold it
            gray_mask = cv2.cvtColor(face_mask_small, cv2.COLOR_
BGR2GRAY)
            ret, mask = cv2.threshold(gray_mask, 180, 255,
cv2.THRESH_BINARY_INV)

            # Create an inverse mask
            mask_inv = cv2.bitwise_not(mask)

            # Use the mask to extract the face mask region of
interest
            masked_face = cv2.bitwise_and(face_mask_small, face_mask_
small, mask=mask)

            # Use the inverse mask to get the remaining part of
the image
            masked_frame = cv2.bitwise_and(frame_roi,
frame_roi, mask=mask_inv)

            # add the two images to get the final output
            frame[y:y+h, x:x+w] = cv2.add(masked_face,
masked_frame)

    cv2.imshow('Face Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Under the hood

Just like before, we first load the face cascade classifier xml file. The face detection steps work as usual. We start the infinite loop and keep detecting the face in every frame. Once we know where the face is, we need to modify the coordinates a bit to make sure the mask fits properly. This manipulation process is subjective and depends on the mask in question. Different masks require different levels of adjustments to make it look more natural. We extract the region-of-interest from the input frame in the following line:

```
frame_roi = frame[y:y+h, x:x+w]
```

Now that we have the required region-of-interest, we need to overlay the mask on top of this. So we resize the input mask to make sure it fits in this region-of-interest. The input mask has a white background. So if we just overlay this on top of the region-of-interest, it will look unnatural because of the white background. We need to overlay only the skull-mask pixels and the remaining area should be transparent.

So in the next step, we create a mask by thresholding the skull image. Since the background is white, we threshold the image so that any pixel with an intensity value greater than 180 becomes 0, and everything else becomes 255. As far as the frame region-of-interest is concerned, we need to black out everything in this mask region. We can do that by simply using the inverse of the mask we just created. Once we have the masked versions of the skull image and the input region-of-interest, we just add them up to get the final image.

Detecting eyes

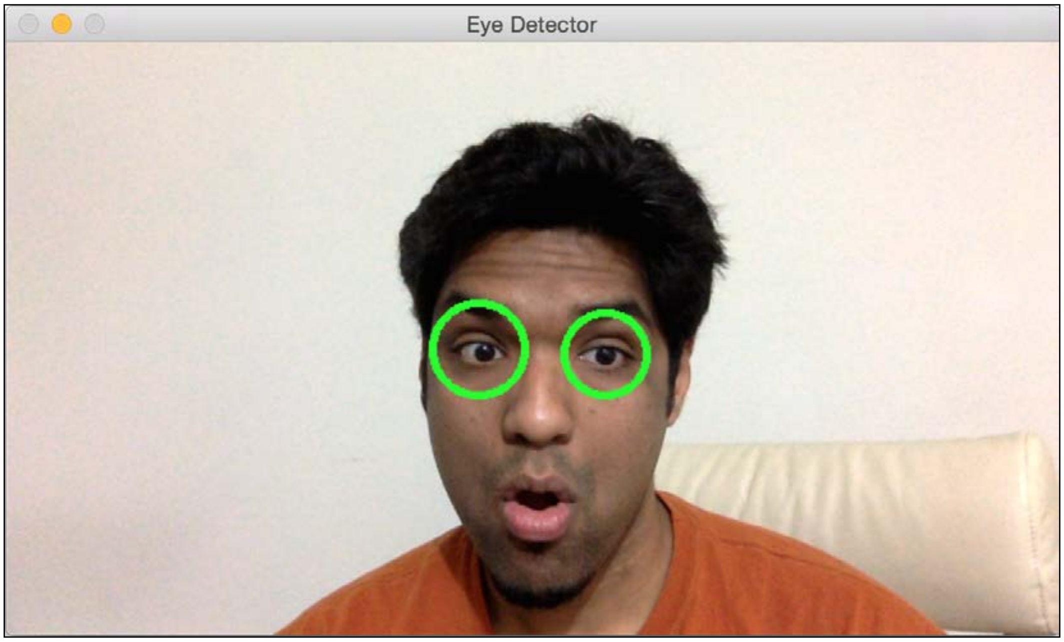
Now that we understand how to detect faces, we can generalize the concept to detect other body parts too. It's important to understand that Viola-Jones framework can be applied to any object. The accuracy and robustness will depend on the uniqueness of the object. For example, a human face has very unique characteristics, so it's easy to train our system to be robust. On the other hand, an object like towel is too generic, and there are no distinguishing characteristics as such; so it's more difficult to build a robust towel detector.

Let's see how to build an eye detector:

```
import cv2
import numpy as np

face_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_
frontalface_alt.xml')
eye_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_eye.
xml')
```

```
if face_cascade.empty():
  raise IOError('Unable to load the face cascade classifier xml file')

if eye_cascade.empty():
  raise IOError('Unable to load the eye cascade classifier xml file')

cap = cv2.VideoCapture(0)
ds_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in faces:
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = frame[y:y+h, x:x+w]
        eyes = eye_cascade.detectMultiScale(roi_gray)
        for (x_eye,y_eye,w_eye,h_eye) in eyes:
            center = (int(x_eye + 0.5*w_eye), int(y_eye + 0.5*h_eye))
            radius = int(0.3 * (w_eye + h_eye))
            color = (0, 255, 0)
            thickness = 3
            cv2.circle(roi_color, center, radius, color, thickness)

    cv2.imshow('Eye Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

If you run this program, the output will look something like the following image:

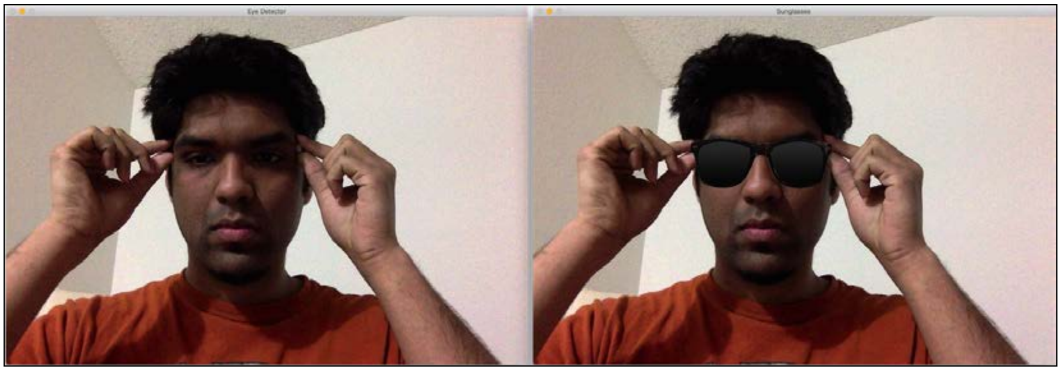


Afterthought

If you notice, the program looks very similar to the face detection program. Along with loading the face detection cascade classifier, we load the eye detection cascade classifier as well. Technically, we don't need to use the face detector. But we know that eyes are always on somebody's face. We use this information and search for eyes only in the relevant region of interest, that is the face. We first detect the face, and then run the eye detector on this sub-image. This way, it's faster and more efficient.

Fun with eyes

Now that we know how to detect eyes in an image, let's see if we can do something fun with it. We can do something like what is shown in the following screenshot:



Let's look at the code to see how to do something like this:

```
import cv2
import numpy as np

face_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.
xml')
eye_cascade = cv2.CascadeClassifier('./cascade_files/haarcascade_eye.
xml')

if face_cascade.empty():
  raise IOError('Unable to load the face cascade classifier
xml file')

if eye_cascade.empty():
  raise IOError('Unable to load the eye cascade classifier xml
file')

img = cv2.imread('input.jpg')
sunglasses_img = cv2.imread('sunglasses.jpg')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

centers = []
faces = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in faces:
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (x_eye,y_eye,w_eye,h_eye) in eyes:
        centers.append((x + int(x_eye + 0.5*w_eye), y +
int(y_eye + 0.5*h_eye)))

if len(centers) > 0:
    # Overlay sunglasses; the factor 2.12 is customizable
depending on the size of the face
    sunglasses_width = 2.12 * abs(centers[1][0] -
centers[0][0])
    overlay_img = np.ones(img.shape, np.uint8) * 255
    h, w = sunglasses_img.shape[:2]
    scaling_factor = sunglasses_width / w
    overlay_sunglasses = cv2.resize(sunglasses_img, None,
fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    x = centers[0][0] if centers[0][0] < centers[1][0] else
centers[1][0]

    # customizable X and Y locations; depends on the size of
the face
    x -= 0.26*overlay_sunglasses.shape[1]
    y += 0.85*overlay_sunglasses.shape[0]

    h, w = overlay_sunglasses.shape[:2]
    overlay_img[y:y+h, x:x+w] = overlay_sunglasses

    # Create mask
    gray_sunglasses = cv2.cvtColor(overlay_img,
cv2.COLOR_BGR2GRAY)
    ret, mask = cv2.threshold(gray_sunglasses, 110, 255,
cv2.THRESH_BINARY)
    mask_inv = cv2.bitwise_not(mask)
    temp = cv2.bitwise_and(img, img, mask=mask)
    temp2 = cv2.bitwise_and(overlay_img, overlay_img,
mask=mask_inv)
    final_img = cv2.add(temp, temp2)

    cv2.imshow('Eye Detector', img)
    cv2.imshow('Sunglasses', final_img)
    cv2.waitKey()
    cv2.destroyAllWindows()
```

Positioning the sunglasses

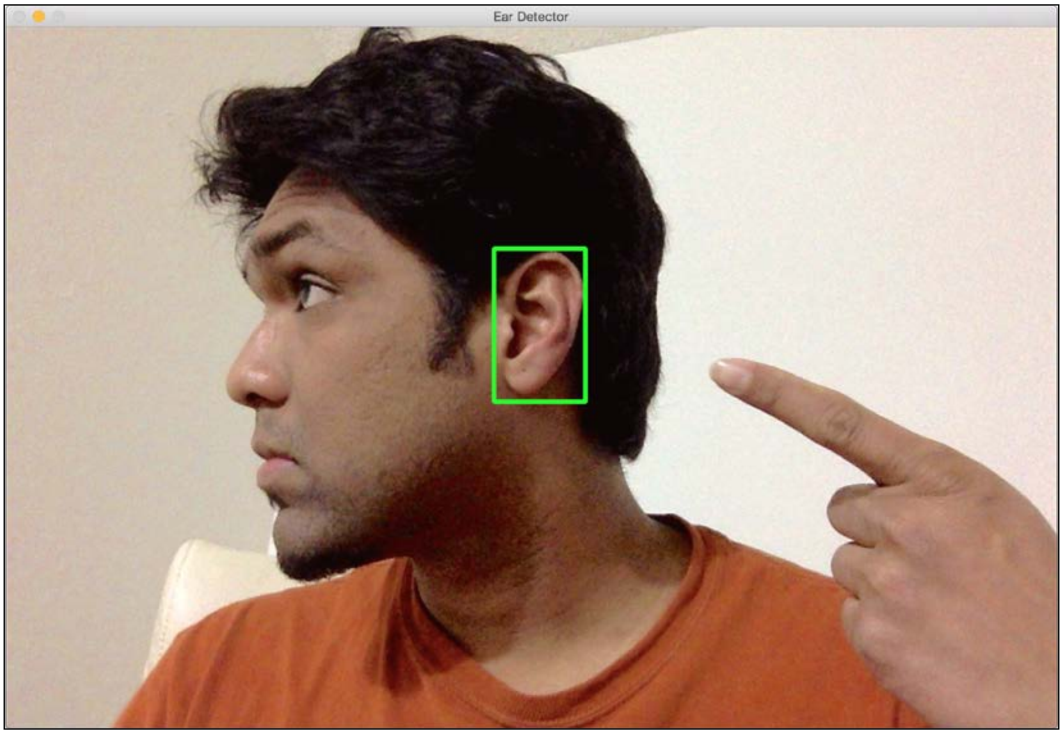
Just like we did earlier, we load the image and detect the eyes. Once we detect the eyes, we resize the sunglasses image to fit the current region of interest. To create the region of interest, we consider the distance between the eyes. We resize the image accordingly and then go ahead to create a mask. This is similar to what we did with the skull mask earlier. The positioning of the sunglasses on the face is subjective. So you will have to tinker with the weights if you want to use a different pair of sunglasses.

Detecting ears

Since we know how the pipeline works, let's just jump into the code:

```
import cv2
import numpy as np

left_ear_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_leftear.xml')
right_ear_cascade = cv2.CascadeClassifier('./cascade_files/
haarcascade_mcs_rightear.xml')

if left_ear_cascade.empty():
  raise IOError('Unable to load the left ear cascade
classifier xml file')

if right_ear_cascade.empty():
  raise IOError('Unable to load the right ear cascade classifier xml
file')

img = cv2.imread('input.jpg')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

left_ear = left_ear_cascade.detectMultiScale(gray, 1.3, 5)
right_ear = right_ear_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in left_ear:
    cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 3)

for (x,y,w,h) in right_ear:
    cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 3)

cv2.imshow('Ear Detector', img)
cv2.waitKey()
cv2.destroyAllWindows()
```

If you run the above code on an image, you should see something like the following screenshot:



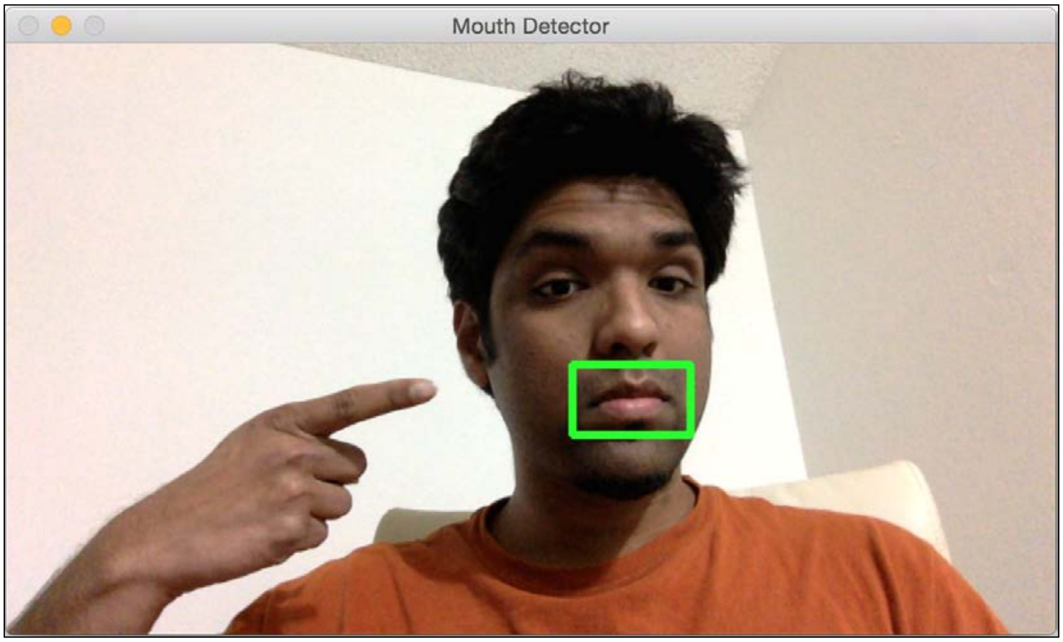
Detecting a mouth

Following is the code:

```
import cv2
import numpy as np

mouth_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_mouth.xml')

if mouth_cascade.empty():
  raise IOError('Unable to load the mouth cascade classifier
xml file')

cap = cv2.VideoCapture(0)
ds_factor = 0.5
```

```
while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    mouth_rects = mouth_cascade.detectMultiScale(gray, 1.7, 11)
    for (x,y,w,h) in mouth_rects:
        y = int(y - 0.15*h)
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
        break

    cv2.imshow('Mouth Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Following is what the output looks like:



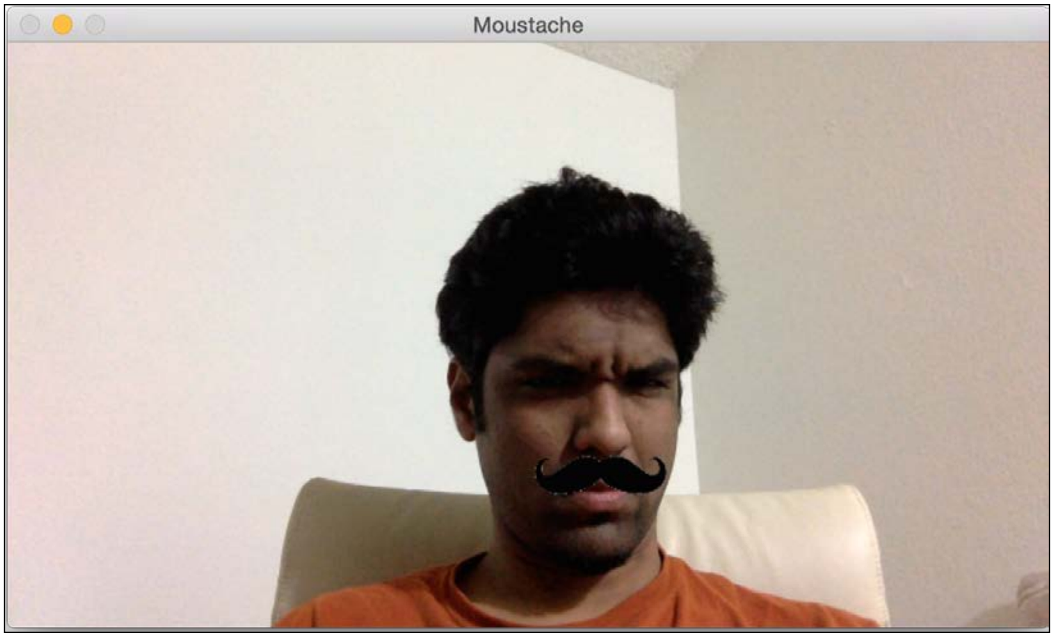
It's time for a moustache

Let's overlay a moustache on top:

```
import cv2
import numpy as np

mouth_cascade =
cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_mouth.xml')

moustache_mask = cv2.imread('../images/moustache.png')
h_mask, w_mask = moustache_mask.shape[:2]

if mouth_cascade.empty():
  raise IOError('Unable to load the mouth cascade classifier
xml file')

cap = cv2.VideoCapture(0)
scaling_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    mouth_rects = mouth_cascade.detectMultiScale(gray, 1.3, 5)
    if len(mouth_rects) > 0:
        (x,y,w,h) = mouth_rects[0]
        h, w = int(0.6*h), int(1.2*w)
        x -= 0.05*w
        y -= 0.55*h
        frame_roi = frame[y:y+h, x:x+w]
        moustache_mask_small = cv2.resize(moustache_mask, (w,
h), interpolation=cv2.INTER_AREA)

        gray_mask = cv2.cvtColor(moustache_mask_small,
cv2.COLOR_BGR2GRAY)
        ret, mask = cv2.threshold(gray_mask, 50, 255,
cv2.THRESH_BINARY_INV)
        mask_inv = cv2.bitwise_not(mask)
        masked_mouth = cv2.bitwise_and(moustache_mask_small,
moustache_mask_small, mask=mask)
        masked_frame = cv2.bitwise_and(frame_roi, frame_roi,
mask=mask_inv)
        frame[y:y+h, x:x+w] = cv2.add(masked_mouth,
masked_frame)
```

```
        cv2.imshow('Moustache', frame)

        c = cv2.waitKey(1)
        if c == 27:
            break

    cap.release()
    cv2.destroyAllWindows()
```

Here's what it looks like:



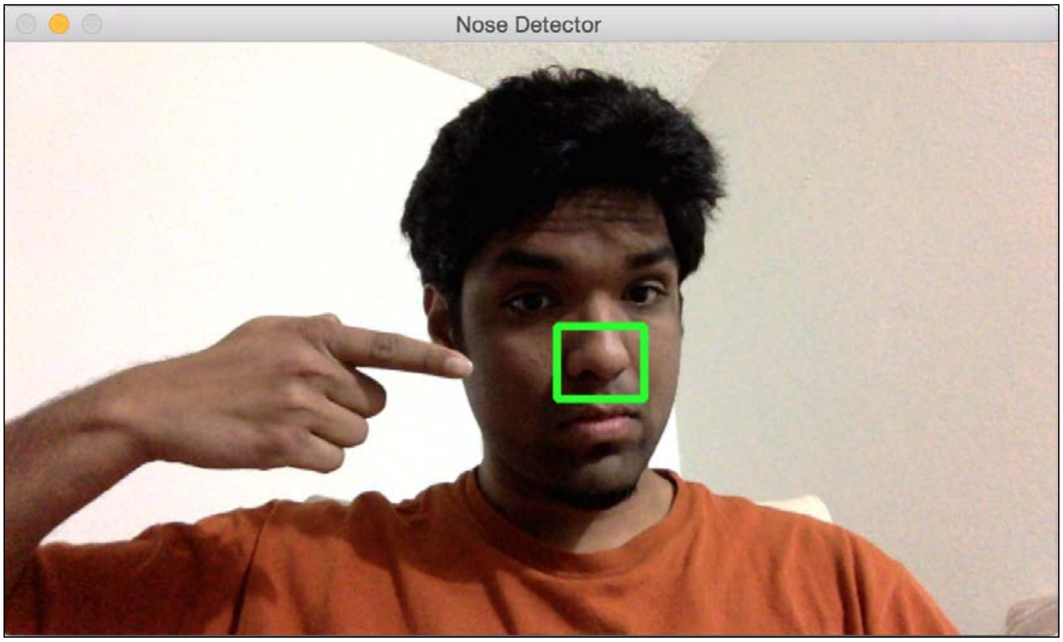
Detecting a nose

The following program shows how you detect a nose:

```
    import cv2
    import numpy as np

    nose_cascade =
    cv2.CascadeClassifier('./cascade_files/haarcascade_mcs_nose.xml')

    if nose_cascade.empty():
      raise IOError('Unable to load the nose cascade classifier
    xml file')
```

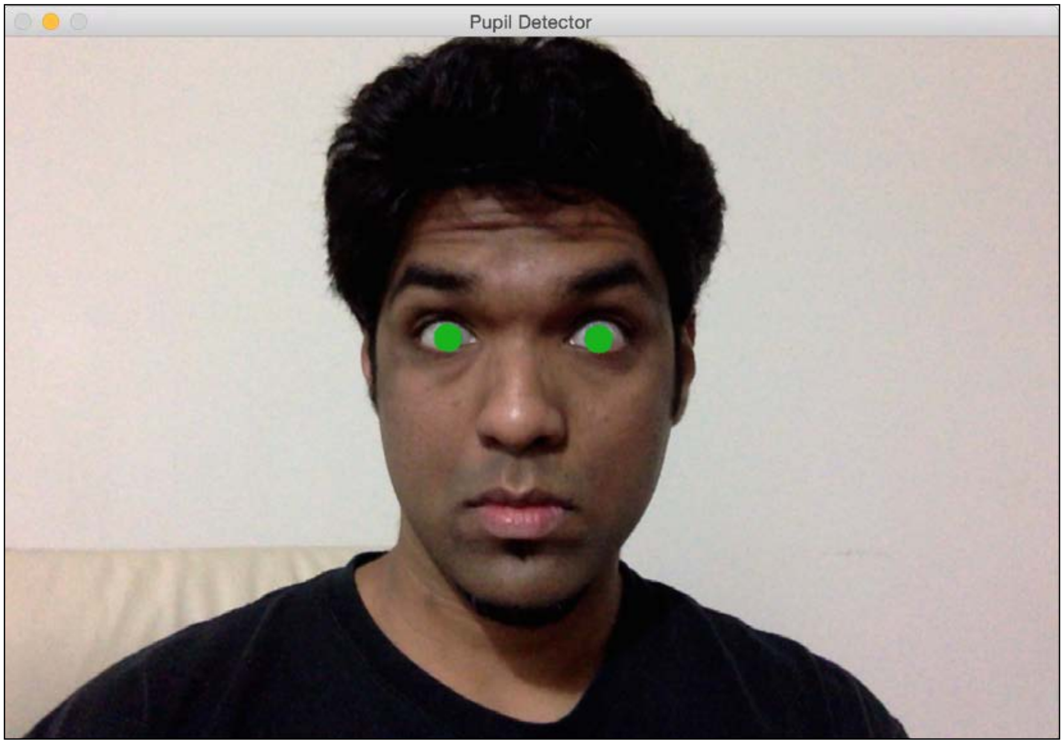
```
cap = cv2.VideoCapture(0)
ds_factor = 0.5

while True:
    ret, frame = cap.read()
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    nose_rects = nose_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in nose_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
        break

    cv2.imshow('Nose Detector', frame)

    c = cv2.waitKey(1)
    if c == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

The output looks something like the following image:



Detecting pupils

We are going to take a different approach here. Pupils are too generic to take the Haar cascade approach. We will also get a sense of how to detect things based on their shape. Following is what the output will look like:



Let's see how to build the pupil detector:

```
import math

import cv2
import numpy as np

img = cv2.imread('input.jpg')
scaling_factor = 0.7

img = cv2.resize(img, None, fx=scaling_factor,
fy=scaling_factor, interpolation=cv2.INTER_AREA)
cv2.imshow('Input', img)
gray = cv2.cvtColor(~img, cv2.COLOR_BGR2GRAY)
```

```
ret, thresh_gray = cv2.threshold(gray, 220, 255,
cv2.THRESH_BINARY)
contours, hierarchy = cv2.findContours(thresh_gray,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

for contour in contours:
    area = cv2.contourArea(contour)
    rect = cv2.boundingRect(contour)
    x, y, width, height = rect
    radius = 0.25 * (width + height)

    area_condition = (100 <= area <= 200)
    symmetry_condition = (abs(1 - float(width)/float(height))
<= 0.2)
    fill_condition = (abs(1 - (area / (math.pi * math.pow(radius,
2.0)))) <= 0.3)

    if area_condition and symmetry_condition and fill_condition:
        cv2.circle(img, (int(x + radius), int(y + radius)),
int(1.3*radius), (0,180,0), -1)

cv2.imshow('Pupil Detector', img)

c = cv2.waitKey()
cv2.destroyAllWindows()
```

If you run this program, you will see the output as shown earlier.

Deconstructing the code

As we discussed earlier, we are not going to use Haar cascade to detect pupils. If we can't use a pre-trained classifier, then how are we going to detect the pupils? Well, we can use shape analysis to detect the pupils. We know that pupils are circular, so we can use this information to detect them in the image. We invert the input image and then convert it into grayscale image as shown in the following line:

```
gray = cv2.cvtColor(~img, cv2.COLOR_BGR2GRAY)
```

As we can see here, we can invert an image using the tilde operator. Inverting the image is helpful in our case because the pupil is black in color, and black corresponds to a low pixel value. We then threshold the image to make sure that there are only black and white pixels. Now, we have to find out the boundaries of all the shapes. OpenCV provides a nice function to achieve this, that is `findContours`. We will discuss more about this in the upcoming chapters. But for now, all we need to know is that this function returns the set of boundaries of all the shapes that are found in the image.

The next step is to identify the shape of the pupil and discard the rest. We will use certain properties of the circle to zero-in on this shape. Let's consider the ratio of width to height of the bounding rectangle. If the shape is a circle, this ratio will be 1. We can use the function `boundingRect` to obtain the coordinates of the bounding rectangle. Let's consider the area of this shape. If we roughly compute the radius of this shape and use the formula for the area of the circle, then it should be close to the area of this contour. We can use the function `contourArea` to compute the area of any contour in the image. So we can use these conditions and filter out the shapes. After we do that, we are left with two pupils in the image. We can refine it further by limiting the search region to the face or the eyes. Since you know how to detect faces and eyes, you can give it a try and see if you can get it working for a live video stream.

Summary

In this chapter, we discussed Haar cascades and integral images. We understood how the face detection pipeline is built. We learnt how to detect and track faces in a live video stream. We discussed how to use the face detection pipeline to detect various body parts like eyes, ears, nose, and mouth. We learnt how to overlay masks on top on the input image using the results of body parts detection. We used the principles of shape analysis to detect the pupils.

In the next chapter, we are going to discuss feature detection and how it can be used to understand the image content.

4
Extracting Features from an Image

In this chapter, we are going to learn how to detect salient points, also known as keypoints, in an image. We will discuss why these keypoints are important and how we can use them to understand the image content. We will talk about different techniques that can be used to detect these keypoints, and understand how we can extract features from a given image.

By the end of this chapter, you will know:

- What are keypoints and why do we care about them
- How to detect keypoints
- How to use keypoints for image content analysis
- The different techniques to detect keypoints
- How to build a feature extractor

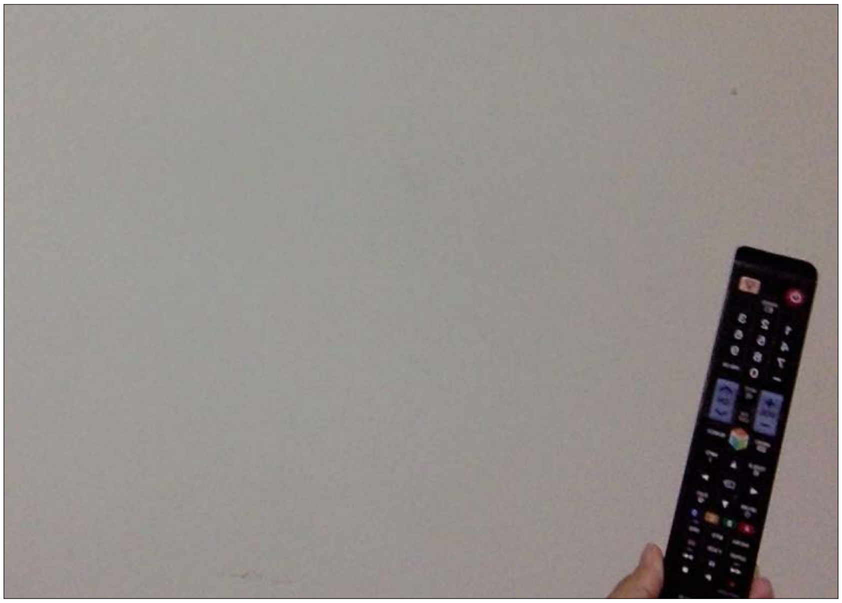
Why do we care about keypoints?

Image content analysis refers to the process of understanding the content of an image so that we can take some action based on that. Let's take a step back and talk about how humans do it. Our brain is an extremely powerful machine that can do complicated things very quickly. When we look at something, our brain automatically creates a footprint based on the "interesting" aspects of that image. We will discuss what interesting means as we move along this chapter.

For now, an interesting aspect is something that's distinct in that region. If we call a point interesting, then there shouldn't be another point in its neighborhood that satisfies the constraints. Let's consider the following image:

Now close your eyes and try to visualize this image. Do you see something specific? Can you recollect what's in the left half of the image? Not really! The reason for this is that the image doesn't have any interesting information. When our brain looks at something like this, there's nothing to make note of. So it tends to wander around! Let's take a look at the following image:

Now close your eyes and try to visualize this image. You will see that the recollection is vivid and you remember a lot of details about this image. The reason for this is that there are a lot of interesting regions in the image. The human eye is more sensitive to high frequency content as compared to low frequency content. This is the reason we tend to recollect the second image better than the first one. To further demonstrate this, let's look at the following image:

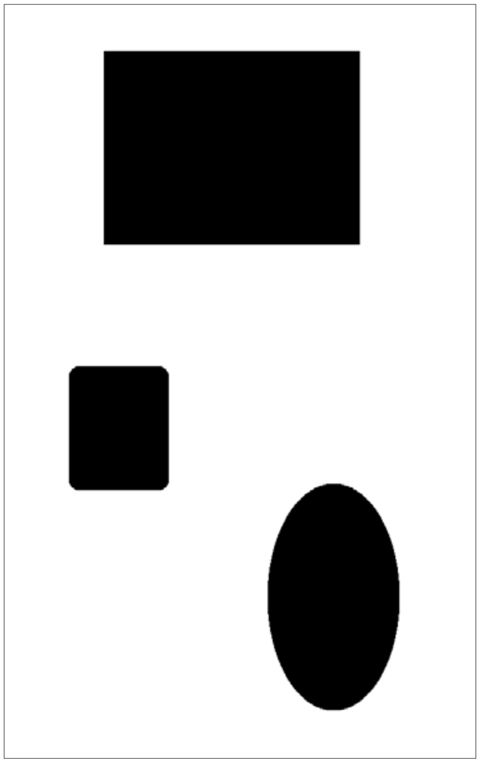


If you notice, your eye immediately went to the TV remote, even though it's not at the center of the image. We automatically tend to gravitate towards the interesting regions in the image because that is where all the information is. This is what our brain needs to store in order to recollect it later.

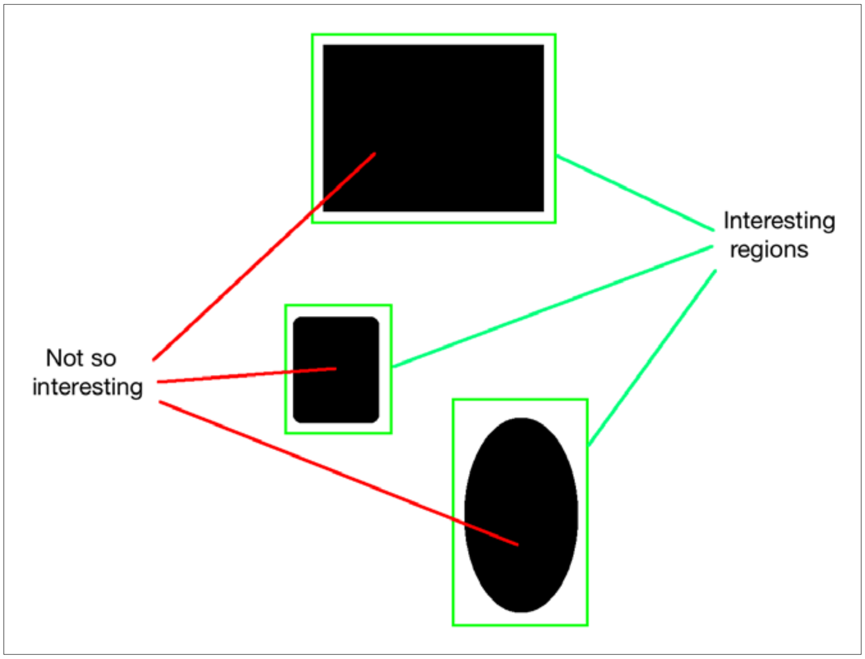
When we build object recognition systems, we need to detect these "interesting" regions to create a signature for the image. These interesting regions are characterized by keypoints. This is why keypoint detection is critical in many modern computer vision systems.

What are keypoints?

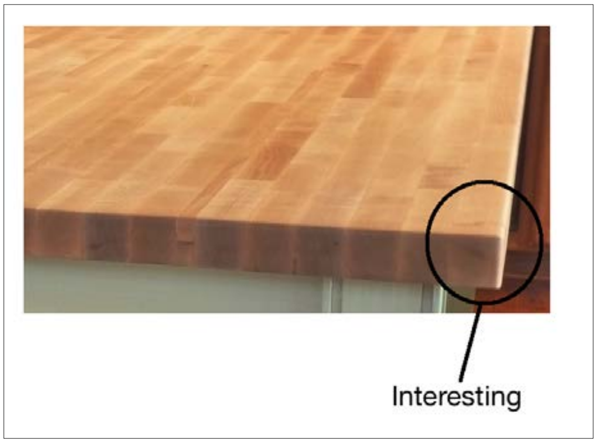
Now that we know that keypoints refer to the interesting regions in the image, let's dig a little deeper. What are keypoints made of? Where are these points? When we say "interesting", it means that something is happening in that region. If the region is just uniform, then it's not very interesting. For example, corners are interesting because there is sharp change in intensity in two different directions. Each corner is a unique point where two edges meet. If you look at the preceding images, you will see that the interesting regions are not completely made up of "interesting" content. If you look closely, we can still see plain regions within busy regions. For example, consider the following image:



If you look at the preceding object, the interior parts of the interesting regions are "uninteresting".



So, if we were to characterize this object, we would need to make sure that we picked the interesting points. Now, how do we define "interesting points"? Can we just say that anything that's not uninteresting can be an interesting point? Let's consider the following example:



Now, we can see that there is a lot of high frequency content in this image along the edge. But we cannot call the whole edge "interesting". It is important to understand that "interesting" doesn't necessarily refer to color or intensity values. It can be anything, as long as it is distinct. We need to isolate the points that are unique in their neighborhood. The points along the edge are not unique with respect to their neighbors. So, now that we know what we are looking for, how do we pick an interesting point?

What about the corner of the table? That's pretty interesting, right? It's unique with respect to its neighbors and we don't have anything like that in its vicinity. Now this point can be chosen as one of our keypoints. We take a bunch of these keypoints to characterize a particular image.

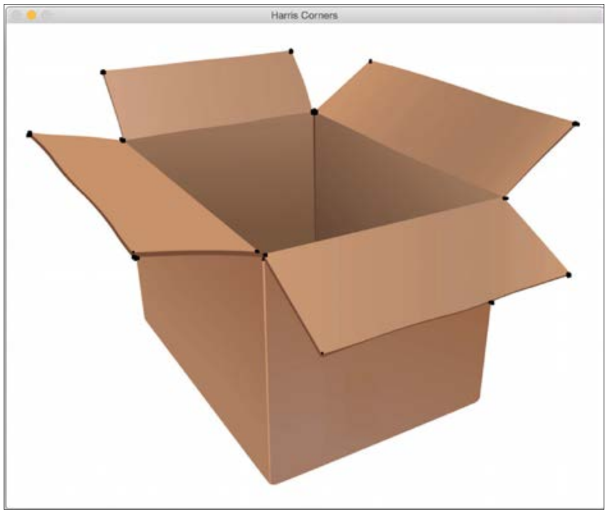
When we do image analysis, we need to convert it into a numerical form before we deduce something. These keypoints are represented using a numerical form and a combination of these keypoints is then used to create the image signature. We want this image signature to represent a given image in the best possible way.

Detecting the corners

Since we know that the corners are "interesting", let's see how we can detect them. In computer vision, there is a popular corner detection technique called **Harris Corner Detector**. We basically construct a 2x2 matrix based on partial derivatives of the grayscale image, and then analyze the eigenvalues. This is actually an oversimplification of the actual algorithm, but it covers the gist. So, if you want to understand the underlying mathematical details, you can look into the original paper by Harris and Stephens at `http://www.bmva.org/bmvc/1988/avc-88-023.pdf`. A corner point is a point where both the eigenvalues would have large values.

Let's consider the following image:

If you run the Harris corner detector on this image, you will see something like this:



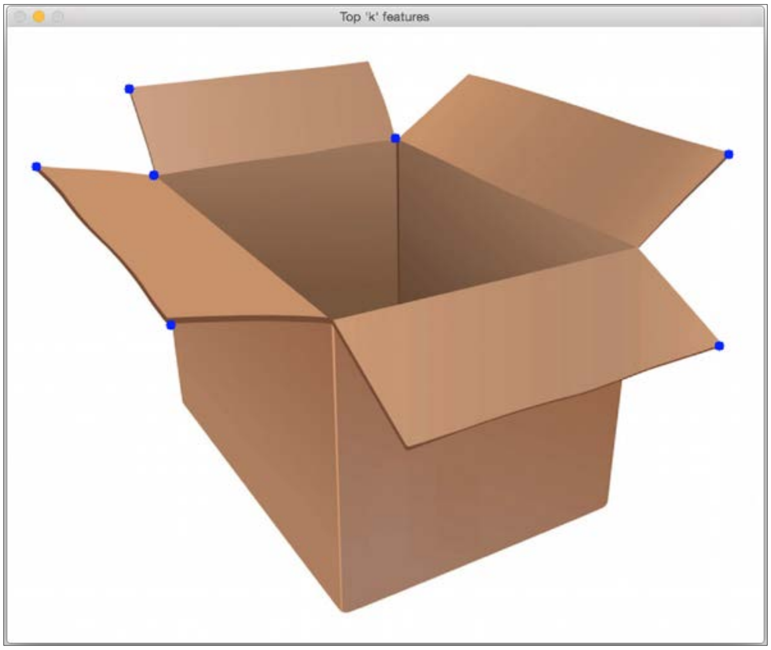
As you can see, all the black dots correspond to the corners in the image. If you notice, the corners at the bottom of the box are not detected. The reason for this is that the corners are not sharp enough. You can adjust the thresholds in the corner detector to identify these corners. The code to do this is as follows:

```
import cv2
import numpy as np

img = cv2.imread('box.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

gray = np.float32(gray)

dst = cv2.cornerHarris(gray, 4,5, 0.04)      # to detect only
sharp corners
#dst = cv2.cornerHarris(gray, 14, 5, 0.04)    # to detect soft
corners

# Result is dilated for marking the corners
dst = cv2.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the
image.
img[dst > 0.01*dst.max()] = [0,0,0]

cv2.imshow('Harris Corners',img)
cv2.waitKey()
```

Good Features To Track

Harris corner detector performs well in many cases, but it misses out on a few things. Around six years after the original paper by Harris and Stephens, Shi-Tomasi came up with a better corner detector. You can read the original paper at `http://www.ai.mit.edu/courses/6.891/handouts/shi94good.pdf`. They used a different scoring function to improve the overall quality. Using this method, we can find the 'N' strongest corners in the given image. This is very useful when we don't want to use every single corner to extract information from the image.

If you apply the Shi-Tomasi corner detector to the image shown earlier, you will see something like this:



Following is the code:

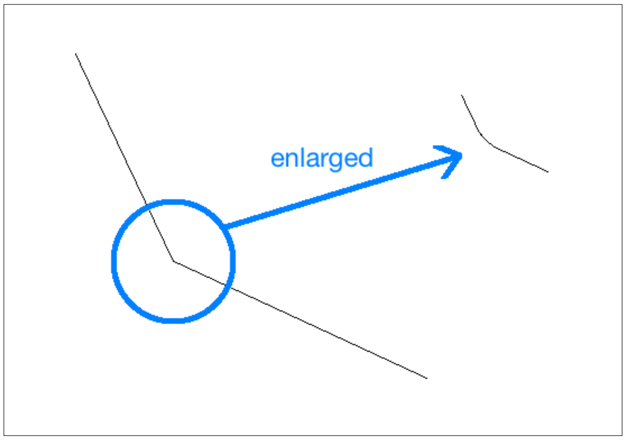
```
import cv2
import numpy as np

img = cv2.imread('box.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray, 7, 0.05, 25)
corners = np.float32(corners)
```

```
for item in corners:
    x, y = item[0]
    cv2.circle(img, (x,y), 5, 255, -1)

cv2.imshow("Top 'k' features", img)
cv2.waitKey()
```

Scale Invariant Feature Transform (SIFT)

Even though corner features are "interesting", they are not good enough to characterize the truly interesting parts. When we talk about image content analysis, we want the image signature to be invariant to things such as scale, rotation, illumination, and so on. Humans are very good at these things. Even if I show you an image of an apple upside down that's dimmed, you will still recognize it. If I show you a really enlarged version of that image, you will still recognize it. We want our image recognition systems to be able to do the same.

Let's consider the corner features. If you enlarge an image, a corner might stop being a corner as shown below.



In the second case, the detector will not pick up this corner. And, since it was picked up in the original image, the second image will not be matched with the first one. It's basically the same image, but the corner features based method will totally miss it. This means that corner detector is not exactly scale invariant. This is why we need a better method to characterize an image.

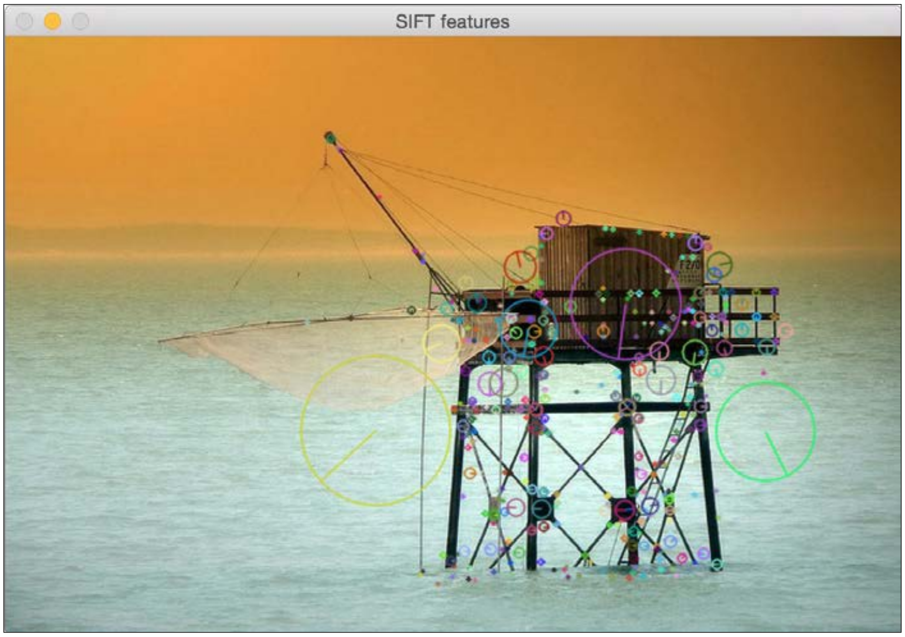
SIFT is one of the most popular algorithms in all of computer vision. You can read David Lowe's original paper at `http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf`. We can use this algorithm to extract keypoints and build the corresponding feature descriptors. There is a lot of good documentation available online, so we will keep our discussion brief. To identify a potential keypoint, SIFT builds a pyramid by downsampling an image and taking the difference of Gaussian. This means that we run a Gaussian filter at each level and take the difference to build the successive levels in the pyramid. In order to see if the current point is a keypoint, it looks at the neighbors as well as the pixels at the same location in neighboring levels of the pyramid. If it's a maxima, then the current point is picked up as a keypoint. This ensures that we keep the keypoints scale invariant.

Now that we know how it achieves scale invariance, let's see how it achieves rotation invariance. Once we identify the keypoints, each keypoint is assigned an orientation. We take the neighborhood around each keypoint and compute the gradient magnitude and direction. This gives us a sense of the direction of that keypoint. If we have this information, we will be able to match this keypoint to the same point in another image even if it's rotated. Since we know the orientation, we will be able to normalize those keypoints before making the comparisons.

Once we have all this information, how do we quantify it? We need to convert it to a set of numbers so that we can do some kind of matching on it. To achieve this, we just take the 16x16 neighborhood around each keypoint, and divide it into 16 blocks of size 4x4. For each block, we compute the orientation histogram with 8 bins. So, we have a vector of length 8 associated with each block, which means that the neighborhood is represented by a vector of size 128 (8x16). This is the final keypoint descriptor that will be used. If we extract `N` keypoints from an image, then we will have `N` descriptors of length 128 each. This array of `N` descriptors characterizes the given image.

Consider the following image:

If you extract the keypoint locations using SIFT, you will see something like the following, where the size of the circle indicates the strength of the keypoints, and the line inside the circle indicates the orientation:



Before we look at the code, it is important to know that SIFT is patented and it's not freely available for commercial use. Following is the code to do it:

```
import cv2
import numpy as np

input_image = cv2.imread('input.jpg')
gray_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)

sift = cv2.SIFT()
keypoints = sift.detect(gray_image, None)

input_image = cv2.drawKeypoints(input_image, keypoints,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('SIFT features', input_image)
cv2.waitKey()
```

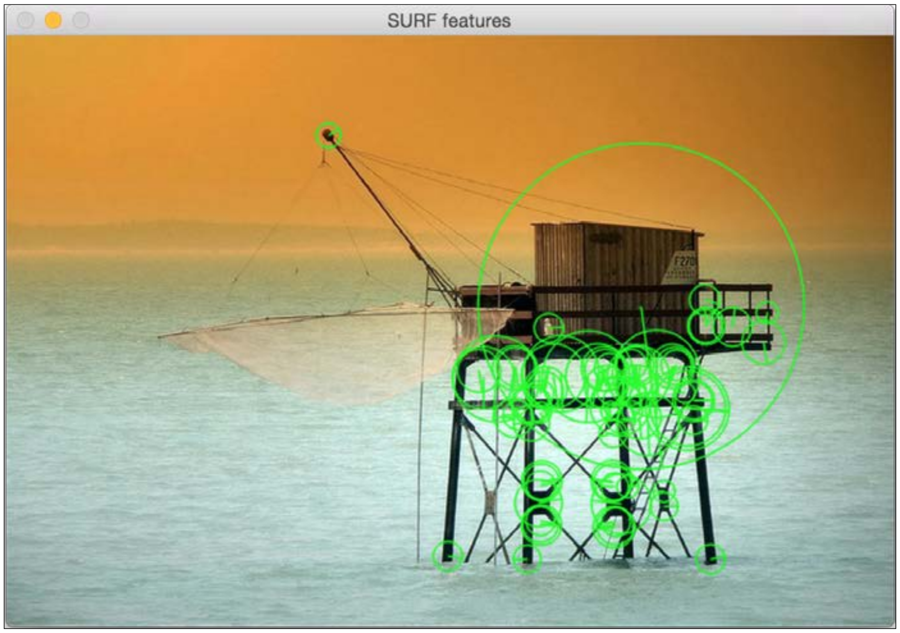
We can also compute the descriptors. OpenCV lets us do it separately or we can combine the detection and computation parts in the same step by using the following:

```
keypoints, descriptors = sift.detectAndCompute(gray_image,
None)
```

Speeded Up Robust Features (SURF)

Even though SIFT is nice and useful, it's computationally intensive. This means that it's slow and we will have a hard time implementing a real-time system if it uses SIFT. We need a system that's fast and has all the advantages of SIFT. If you remember, SIFT uses the difference of Gaussian to build the pyramid and this process is slow. So, to overcome this, SURF uses a simple box filter to approximate the Gaussian. The good thing is that this is really easy to compute and it's reasonably fast. There's a lot of documentation available online on SURF at `http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html?highlight=surf`. So, you can go through it to see how they construct a descriptor. You can refer to the original paper at `http://www.vision.ee.ethz.ch/~surf/eccv06.pdf`. It is important to know that SURF is also patented and it is not freely available for commercial use.

If you run the SURF keypoint detector on the earlier image, you will see something like the following one:



Here is the code:

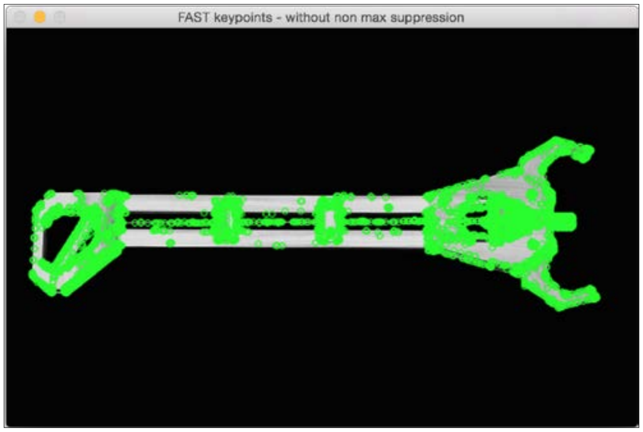
```
import cv2
import numpy as np

img = cv2.imread('input.jpg')
gray= cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

surf = cv2.SURF()

# This threshold controls the number of keypoints
surf.hessianThreshold = 15000

kp, des = surf.detectAndCompute(gray, None)

img = cv2.drawKeypoints(img, kp, None, (0,255,0), 4)

cv2.imshow('SURF features', img)
cv2.waitKey()
```

Features from Accelerated Segment Test (FAST)

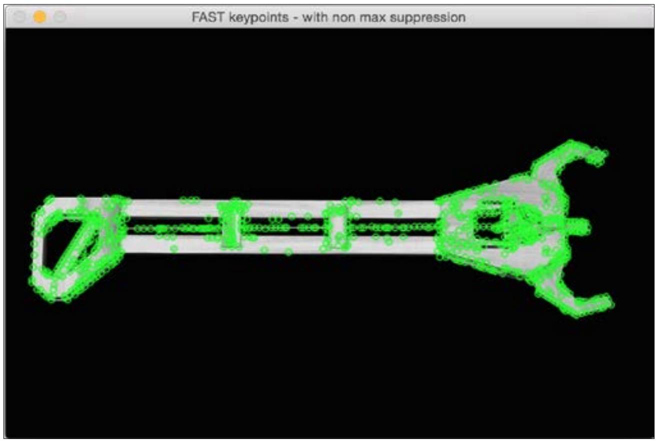
Even though SURF is faster than SIFT, it's just not fast enough for a real-time system, especially when there are resource constraints. When you are building a real-time application on a mobile device, you won't have the luxury of using SURF to do computations in real time. We need something that's really fast and computationally inexpensive. Hence, Rosten and Drummond came up with FAST. As the name indicates, it's really fast!

Instead of going through all the expensive calculations, they came up with a high-speed test to quickly determine if the current point is a potential keypoint. We need to note that FAST is just for keypoint detection. Once keypoints are detected, we need to use SIFT or SURF to compute the descriptors. Consider the following image:

If we run the FAST keypoint detector on this image, you will see something like this:



If we clean it up and suppress the unimportant keypoints, it will look like this:



Following is the code for this:

```
import cv2
import numpy as np

gray_image = cv2.imread('input.jpg', 0)

fast = cv2.FastFeatureDetector()

# Detect keypoints
keypoints = fast.detect(gray_image, None)
print "Number of keypoints with non max suppression:",
len(keypoints)

# Draw keypoints on top of the input image
img_keypoints_with_nonmax = cv2.drawKeypoints(gray_image,
keypoints, color=(0,255,0))
cv2.imshow('FAST keypoints - with non max suppression',
img_keypoints_with_nonmax)

# Disable nonmaxSuppression
fast.setBool('nonmaxSuppression', False)

# Detect keypoints again
keypoints = fast.detect(gray_image, None)
```

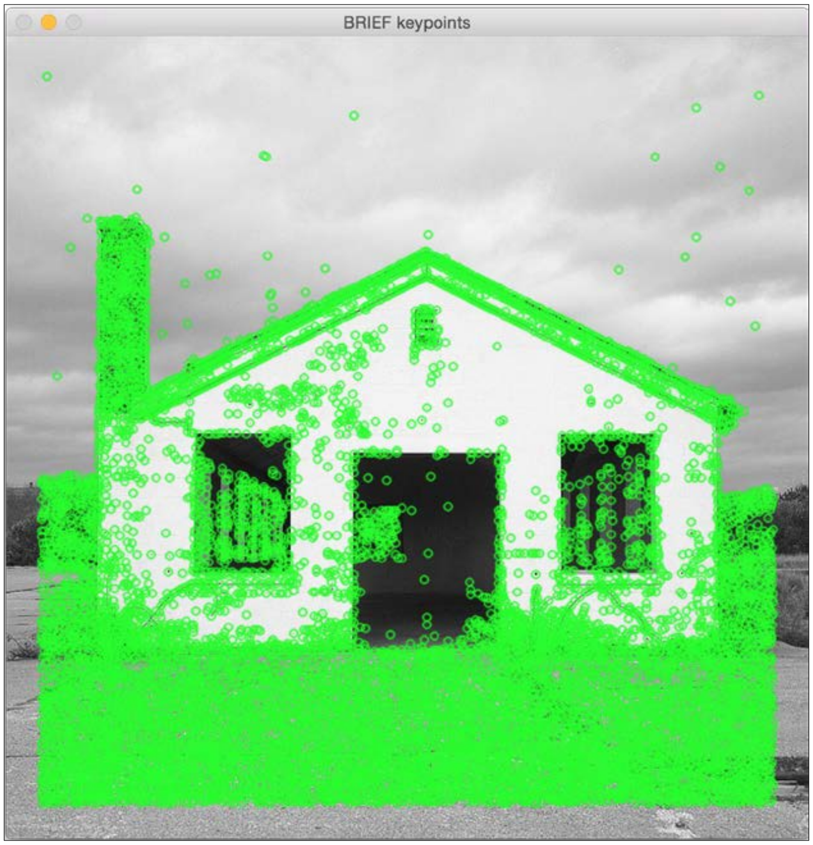
```
print "Total Keypoints without nonmaxSuppression:",
len(keypoints)

# Draw keypoints on top of the input image
img_keypoints_without_nonmax = cv2.drawKeypoints(gray_image,
keypoints, color=(0,255,0))
cv2.imshow('FAST keypoints - without non max suppression',
img_keypoints_without_nonmax)
cv2.waitKey()
```

Binary Robust Independent Elementary Features (BRIEF)

Even though we have FAST to quickly detect the keypoints, we still have to use SIFT or SURF to compute the descriptors. We need a way to quickly compute the descriptors as well. This is where BRIEF comes into the picture. BRIEF is a method for extracting feature descriptors. It cannot detect the keypoints by itself, so we need to use it in conjunction with a keypoint detector. The good thing about BRIEF is that it's compact and fast.

Consider the following image:

BRIEF takes the list of input keypoints and outputs an updated list. So if you run BRIEF on this image, you will see something like this:



Following is the code:

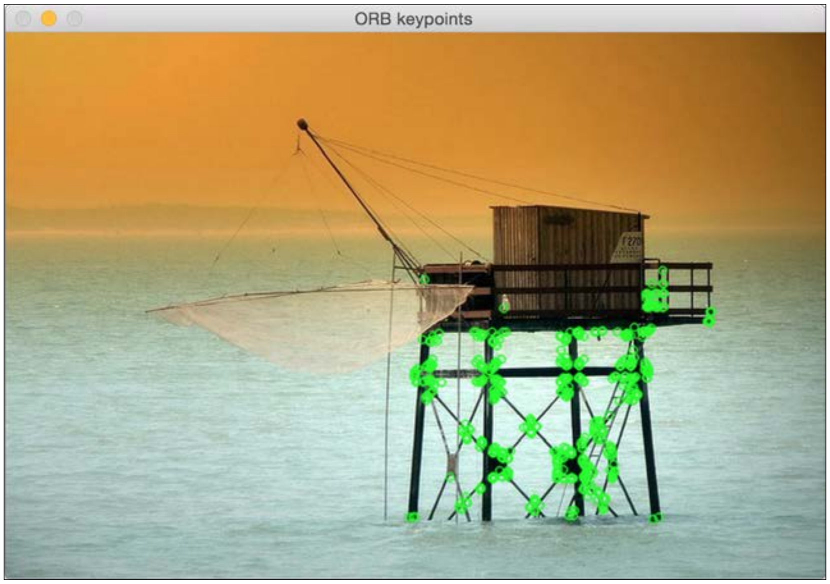
```
import cv2
import numpy as np

gray_image = cv2.imread('input.jpg', 0)

# Initiate FAST detector
fast = cv2.FastFeatureDetector()

# Initiate BRIEF extractor
brief = cv2.DescriptorExtractor_create("BRIEF")
```

```
# find the keypoints with STAR
keypoints = fast.detect(gray_image, None)

# compute the descriptors with BRIEF
keypoints, descriptors = brief.compute(gray_image, keypoints)

gray_keypoints = cv2.drawKeypoints(gray_image, keypoints,
color=(0,255,0))
cv2.imshow('BRIEF keypoints', gray_keypoints)
cv2.waitKey()
```

Oriented FAST and Rotated BRIEF (ORB)

So, now we have arrived at the best combination out of all the combinations that we have discussed so far. This algorithm came out of the OpenCV Labs. It's fast, robust, and open-source! Both SIFT and SURF algorithms are patented and you can't use them for commercial purposes. This is why ORB is good in many ways.

If you run the ORB keypoint extractor on one of the images shown earlier, you will see something like the following:



Here is the code:

```
import cv2
import numpy as np

input_image = cv2.imread('input.jpg')
gray_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)

# Initiate ORB object
orb = cv2.ORB()

# find the keypoints with ORB
keypoints = orb.detect(gray_image, None)

# compute the descriptors with ORB
keypoints, descriptors = orb.compute(gray_image, keypoints)

# draw only the location of the keypoints without size or
orientation
final_keypoints = cv2.drawKeypoints(input_image, keypoints,
color=(0,255,0), flags=0)

cv2.imshow('ORB keypoints', final_keypoints)
cv2.waitKey()
```

Summary

In this chapter, we learned about the importance of keypoints and why we need them. We discussed various algorithms to detect keypoints and compute feature descriptors. We will be using these algorithms in all the subsequent chapters in various different contexts. The concept of keypoints is central to computer vision, and plays an important role in many modern systems.

In the next chapter, we are going to discuss how to stitch multiple images of the same scene together to create a panoramic image.

5

Creating a Panoramic Image

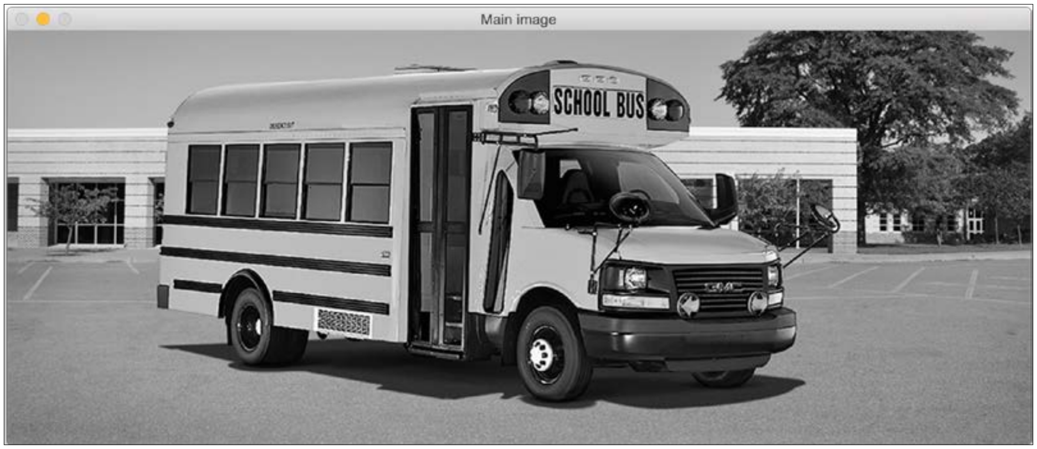
In this chapter, we are going to learn how to stitch multiple images of the same scene together to create a panoramic image.

By the end of this chapter, you will know:

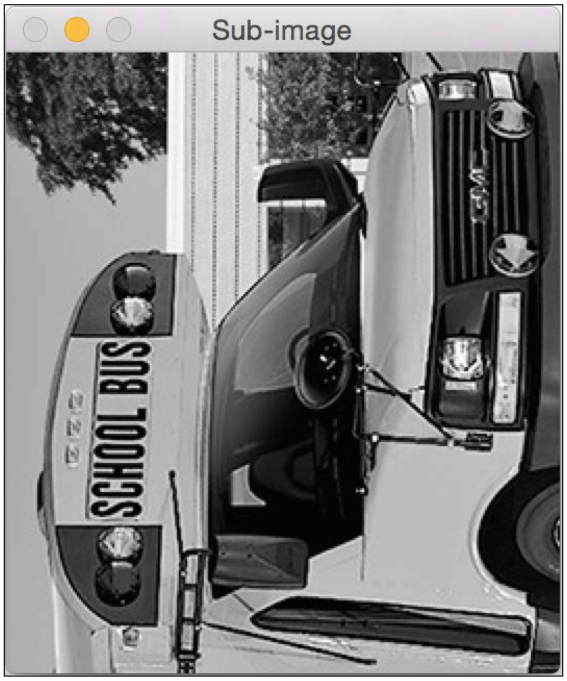
- How to match keypoint descriptors between multiple images
- How to find overlapping regions between images
- How to warp images based on the matching keypoints
- How to stitch multiple images to create a panoramic image

Matching keypoint descriptors

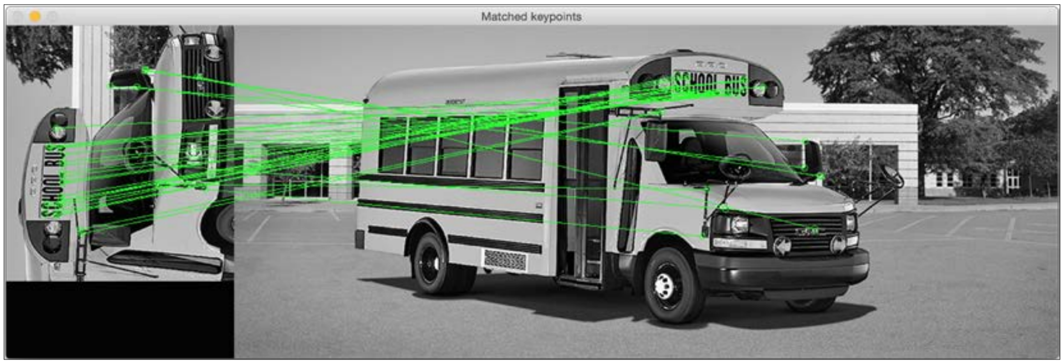
In the last chapter, we learned how to extract keypoints using various methods. The reason that we extract keypoints is because we can use them for image matching. Let's consider the following image:



As you can see, it's the picture of a school bus. Now, let's take a look at the following image:



The preceding image is a part of the school bus image and it's been rotated anticlockwise by 90 degrees. We could easily recognize this because our brain is invariant to scaling and rotation. Our goal here is to find the matching points between these two images. If you do that, it would look something like this:



Following is the code to do this:

```
import sys

import cv2
import numpy as np

def draw_matches(img1, keypoints1, img2, keypoints2, matches):
    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]

    # Create a new output image that concatenates the two images
together
    output_img = np.zeros((max([rows1,rows2]), cols1+cols2, 3),
dtype='uint8')
    output_img[:rows1, :cols1, :] = np.dstack([img1, img1, img1])
    output_img[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2,
img2])

    # Draw connecting lines between matching keypoints
    for match in matches:
        # Get the matching keypoints for each of the images
        img1_idx = match.queryIdx
        img2_idx = match.trainIdx

        (x1, y1) = keypoints1[img1_idx].pt
        (x2, y2) = keypoints2[img2_idx].pt

        # Draw a small circle at both co-ordinates and then draw a
line
        radius = 4
        colour = (0,255,0)   # green
        thickness = 1
        cv2.circle(output_img, (int(x1),int(y1)), radius, colour,
thickness)
        cv2.circle(output_img, (int(x2)+cols1,int(y2)), radius,
colour, thickness)
        cv2.line(output_img, (int(x1),int(y1)),
(int(x2)+cols1,int(y2)), colour, thickness)

    return output_img

if __name__=='__main__':
    img1 = cv2.imread(sys.argv[1], 0)   # query image (rotated
subregion)
```

```
    img2 = cv2.imread(sys.argv[2], 0)   # train image (full image)

    # Initialize ORB detector
    orb = cv2.ORB()

    # Extract keypoints and descriptors
    keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
    keypoints2, descriptors2 = orb.detectAndCompute(img2, None)

    # Create Brute Force matcher object
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    # Match descriptors
    matches = bf.match(descriptors1, descriptors2)

    # Sort them in the order of their distance
    matches = sorted(matches, key = lambda x:x.distance)

    # Draw first 'n' matches
    img3 = draw_matches(img1, keypoints1, img2, keypoints2,
matches[:30])

    cv2.imshow('Matched keypoints', img3)
    cv2.waitKey()
```

How did we match the keypoints?

In the preceding code, we used the ORB detector to extract the keypoints. Once we extracted the keypoints, we used the Brute Force matcher to match the descriptors. Brute Force matching is pretty straightforward! For every descriptor in the first image, we match it with every descriptor in the second image and take the closest one. To compute the closest descriptor, we use the Hamming distance as the metric, as shown in the following line:

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

You can read more about the Hamming distance at `https://en.wikipedia.org/wiki/Hamming_distance`. The second argument in the preceding line is a Boolean variable. If this is true, then the matcher returns only those keypoints that are closest to each other in both directions. This means that if we get (i, j) as a match, then we can be sure that the i-th descriptor in the first image has the j-th descriptor in the second image as its closest match and vice versa. This increases the consistency and robustness of descriptor matching.

Understanding the matcher object

Let's consider the following line again:

```
matches = bf.match(descriptors1, descriptors2)
```

Here, the variable matches is a list of DMatch objects. You can read more about it in the OpenCV documentation. We just need to quickly understand what it means because it will become increasingly relevant in the upcoming chapters. If we are iterating over this list of DMatch objects, then each item will have the following attributes:

- **item.distance**: This attribute gives us the distance between the descriptors. A lower distance indicates a better match.
- **item.trainIdx**: This attribute gives us the index of the descriptor in the list of train descriptors (in our case, it's the list of descriptors in the full image).
- **item.queryIdx**: This attribute gives us the index of the descriptor in the list of query descriptors (in our case, it's the list of descriptors in the rotated subimage).
- **item.imgIdx**: This attribute gives us the index of the train image.

Drawing the matching keypoints

Now that we know how to access different attributes of the matcher object, let's see how we can use them to draw the matching keypoints. OpenCV 3.0 provides a direct function to draw the matching keypoints, but we will not be using that. It's better to take a peek inside to see what's happening underneath.

We need to create a big output image that can fit both the images side by side. So, we do that in the following line:

```
output_img = np.zeros((max([rows1,rows2]), cols1+cols2, 3),
dtype='uint8')
```

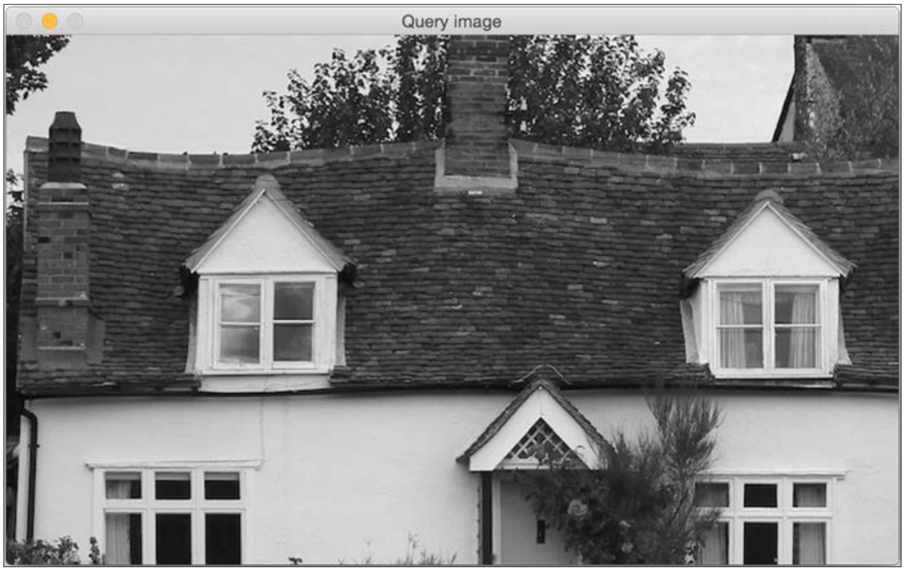
As we can see here, the number of rows is set to the bigger of the two values and the number of columns is simply the sum of both the values. For each item in the list of matches, we extract the locations of the matching keypoints, as we can see in the following lines:

```
(x1, y1) = keypoints1[img1_idx].pt
(x2, y2) = keypoints2[img2_idx].pt
```

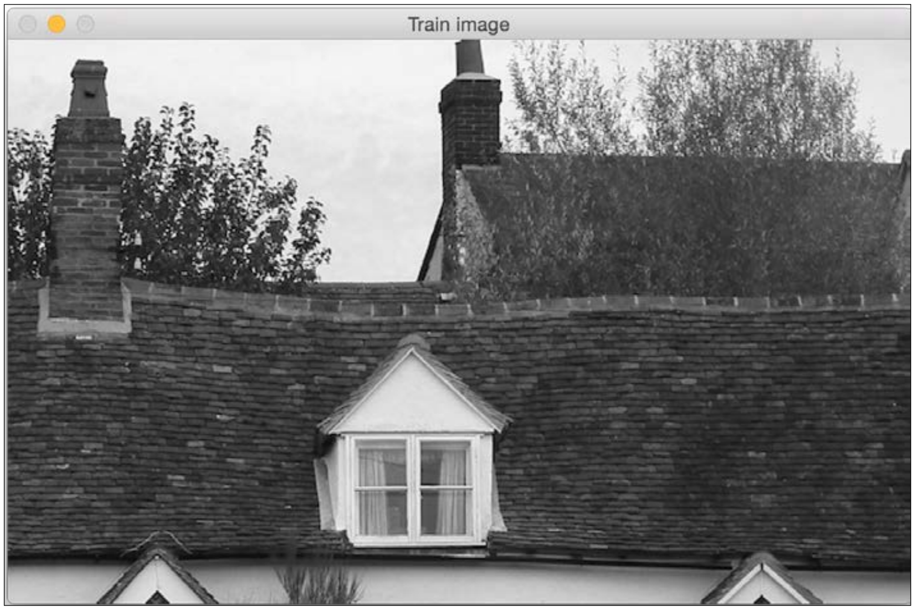
Once we do that, we just draw circles on those points to indicate their locations and then draw a line connecting the two points.

Creating the panoramic image

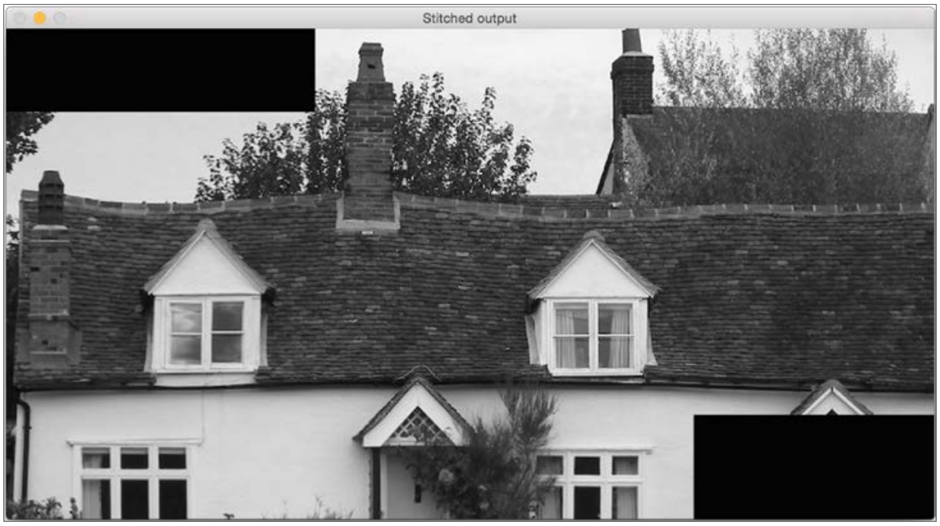
Now that we know how to match keypoints, let's go ahead and see how we can stitch multiple images together. Consider the following image:



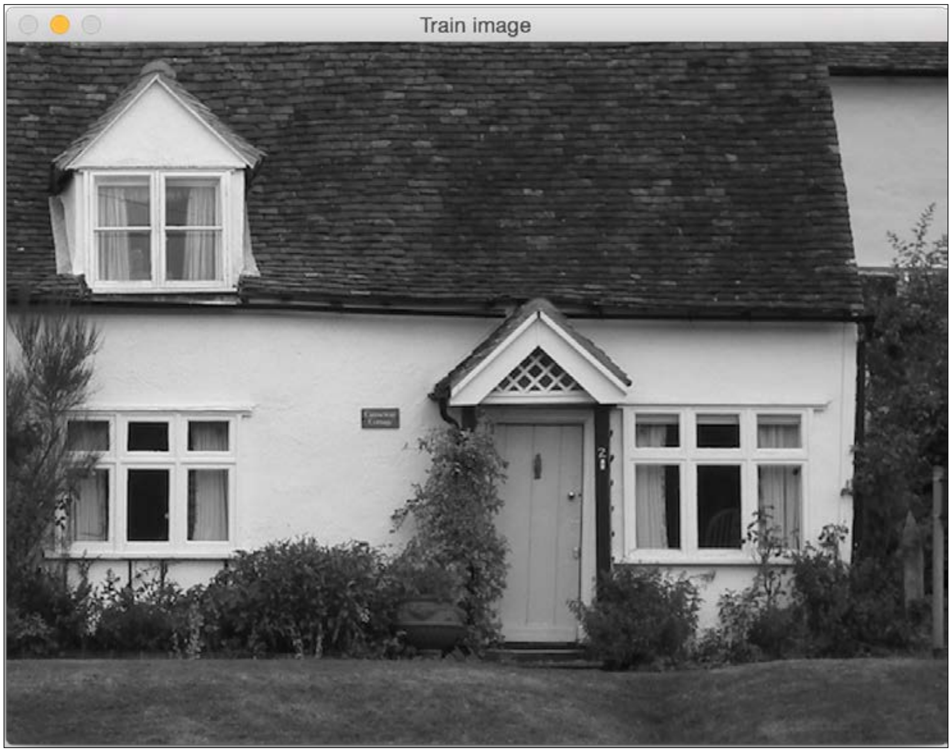
Let's say we want to stitch the following image with the preceding image:



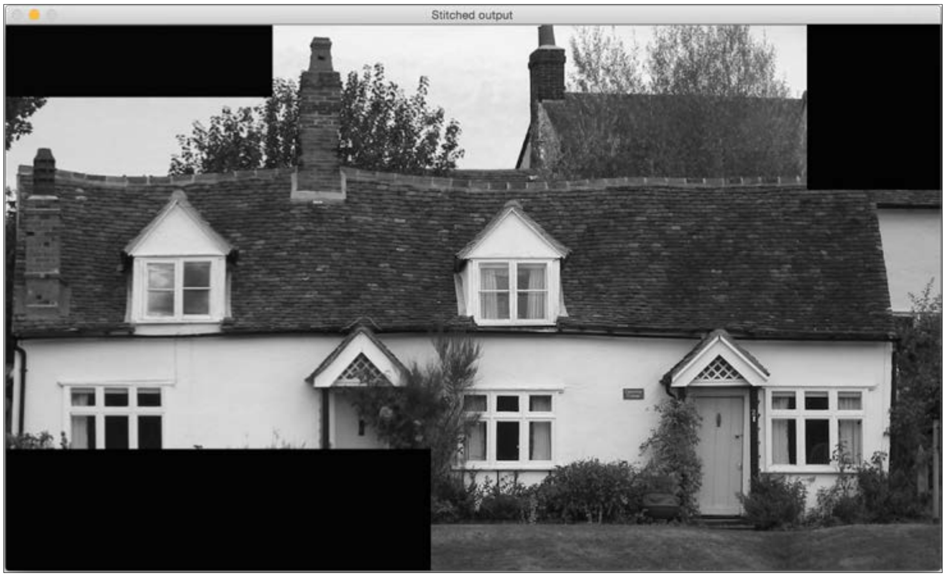
If we stitch these images, it will look something like the following one:



Now let's say we captured another part of this house, as seen in the following image:



If we stitch the preceding image with the stitched image we saw earlier, it will look something like this:



We can keep stitching images together to create a nice panoramic image. Let's take a look at the code:

```
import sys
import argparse

import cv2
import numpy as np

def argument_parser():
    parser = argparse.ArgumentParser(description='Stitch two
images together')
    parser.add_argument("--query-image", dest="query_image",
required=True,
            help="First image that needs to be stitched")
    parser.add_argument("--train-image", dest="train_image",
required=True,
            help="Second image that needs to be stitched")
    parser.add_argument("--min-match-count",
dest="min_match_count", type=int,
            required=False, default=10, help="Minimum number of
matches required")
    return parser
```

```
# Warp img2 to img1 using the homography matrix H
def warpImages(img1, img2, H):
    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]

    list_of_points_1 = np.float32([[0,0], [0,rows1],
[cols1,rows1], [cols1,0]]).reshape(-1,1,2)
    temp_points = np.float32([[0,0], [0,rows2], [cols2,rows2],
[cols2,0]]).reshape(-1,1,2)
    list_of_points_2 = cv2.perspectiveTransform(temp_points, H)
    list_of_points = np.concatenate((list_of_points_1,
list_of_points_2), axis=0)

    [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() -
0.5)
    [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() +
0.5)
    translation_dist = [-x_min,-y_min]
    H_translation = np.array([[1, 0, translation_dist[0]], [0, 1,
translation_dist[1]], [0,0,1]])

    output_img = cv2.warpPerspective(img2, H_translation.dot(H),
(x_max-x_min, y_max-y_min))
    output_img[translation_dist[1]:rows1+translation_dist[1],
translation_dist[0]:cols1+translation_dist[0]] = img1

    return output_img

if __name__=='__main__':
    args = argument_parser().parse_args()
    img1 = cv2.imread(args.query_image, 0)
    img2 = cv2.imread(args.train_image, 0)
    min_match_count = args.min_match_count

    cv2.imshow('Query image', img1)
    cv2.imshow('Train image', img2)

    # Initialize the SIFT detector
    sift = cv2.SIFT()

    # Extract the keypoints and descriptors
    keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
    # Initialize parameters for Flann based matcher
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)

    # Initialize the Flann based matcher object
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # Compute the matches
    matches = flann.knnMatch(descriptors1, descriptors2, k=2)

    # Store all the good matches as per Lowe's ratio test
    good_matches = []
    for m1,m2 in matches:
        if m1.distance < 0.7*m2.distance:
            good_matches.append(m1)

    if len(good_matches) > min_match_count:
        src_pts = np.float32([ keypoints1[good_match.queryIdx].pt
for good_match in good_matches ]).reshape(-1,1,2)
        dst_pts = np.float32([ keypoints2[good_match.trainIdx].pt
for good_match in good_matches ]).reshape(-1,1,2)

        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,
5.0)
        result = warpImages(img2, img1, M)
        cv2.imshow('Stitched output', result)

        cv2.waitKey()

    else:
        print "We don't have enough number of matches between the
two images."
        print "Found only %d matches. We need at least %d
matches." % (len(good_matches), min_match_count)
```

Finding the overlapping regions

The goal here is to find the matching keypoints so that we can stitch the images together. So, the first step is to get these matching keypoints. As discussed in the previous section, we use a keypoint detector to extract the keypoints, and then use a Flann based matcher to match the keypoints.

You can learn more about Flann at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.192.5378&rep=rep1&type=pdf.

The Flann based matcher is faster than Brute Force matching because it doesn't compare each point with every single point on the other list. It only considers the neighborhood of the current point to get the matching keypoint, thereby making it more efficient.

Once we get a list of matching keypoints, we use Lowe's ratio test to keep only the strong matches. David Lowe proposed this ratio test in order to increase the robustness of SIFT.

You can read more about this at http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf.

Basically, when we match the keypoints, we reject the matches in which the ratio of the distances to the nearest neighbor and the second nearest neighbor is greater than a certain threshold. This helps us in discarding the points that are not distinct enough. So, we use that concept here to keep only the good matches and discard the rest. If we don't have sufficient matches, we don't proceed further. In our case, the default value is 10. You can play around with this input parameter to see how it affects the output.

If we have a sufficient number of matches, then we extract the list of keypoints in both the images and extract the homography matrix. If you remember, we have already discussed homography in the first chapter. So if you have forgotten about it, you may want to take a quick look. We basically take a bunch of points from both the images and extract the transformation matrix.

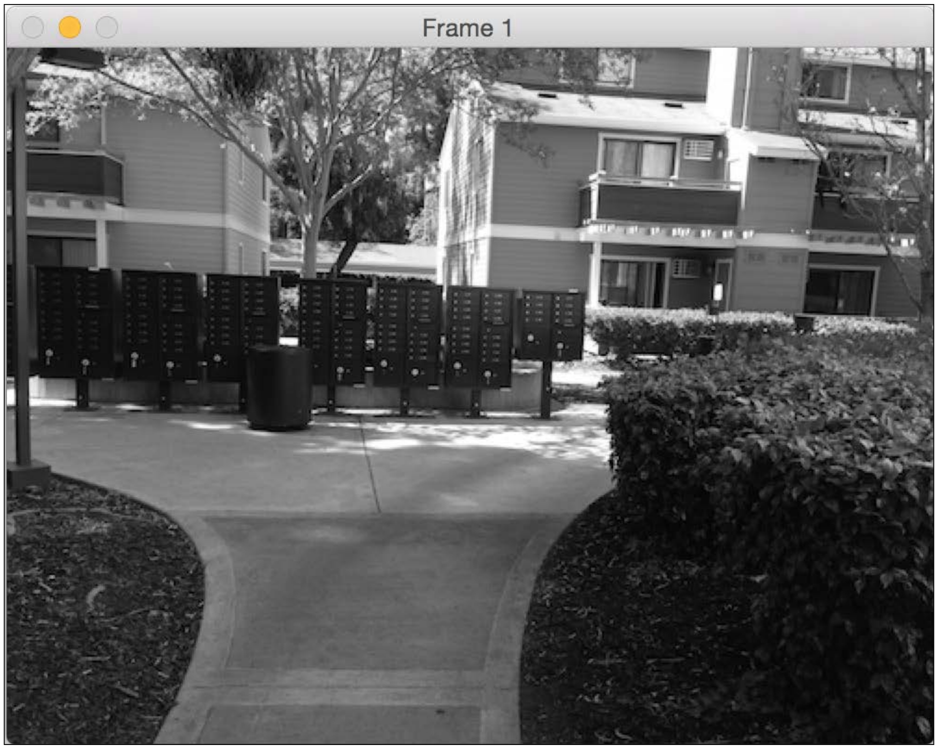
Stitching the images

Now that we have the transformation, we can go ahead and stitch the images. We will use the transformation matrix to transform the second list of points. We keep the first image as the frame of reference and create an output image that's big enough to hold both the images. We need to extract information about the transformation of the second image. We need to move it into this frame of reference to make sure it aligns with the first image. So, we have to extract the translation information and then warp it. We then add the first image into this and construct the final output. It is worth mentioning that this works for images with different aspect ratios as well. So, if you get a chance, try it out and see what the output looks like.

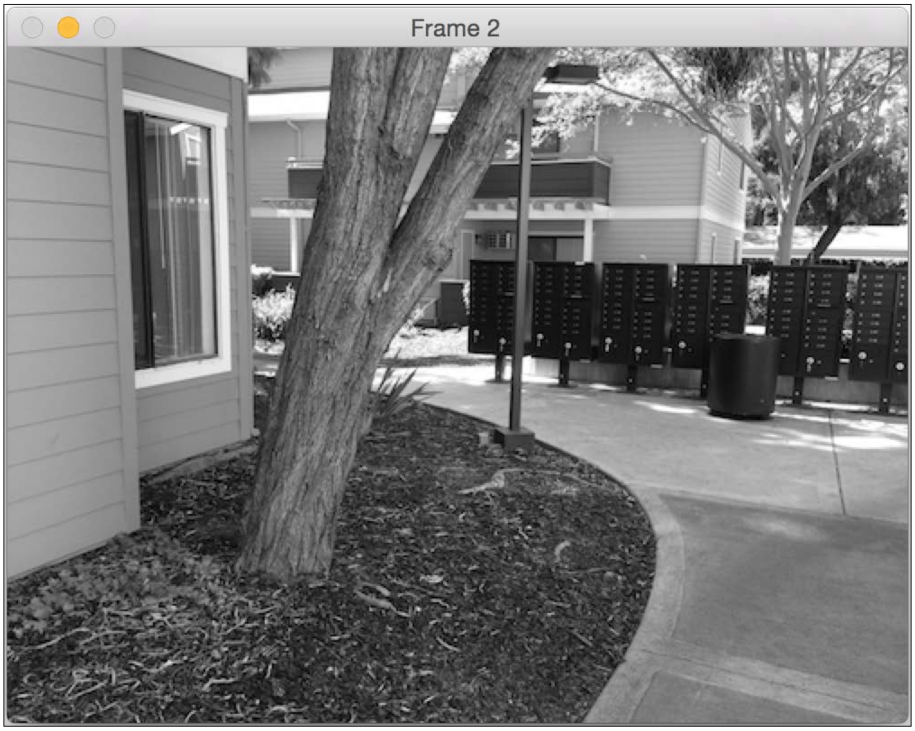
What if the images are at an angle to each other?

Until now, we were looking at images that were on the same plane. Stitching those images was straightforward and we didn't have to deal with any artifacts. In real life, you cannot capture multiple images on exactly the same plane. When you are capturing multiple images of the same scene, you are bound to tilt your camera and change the plane. So the question is, will our algorithm work in that scenario? As it turns out, it can handle those cases as well.

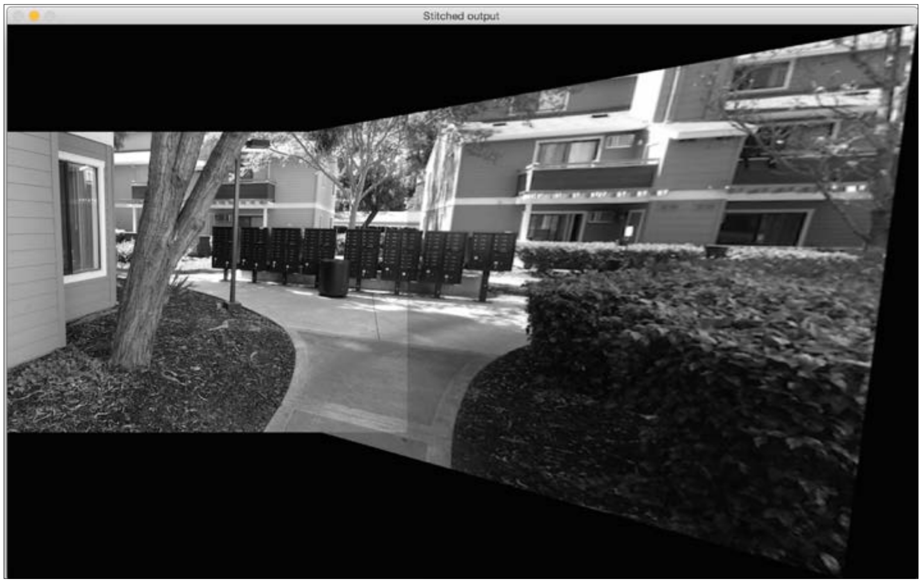
Let's consider the following image:



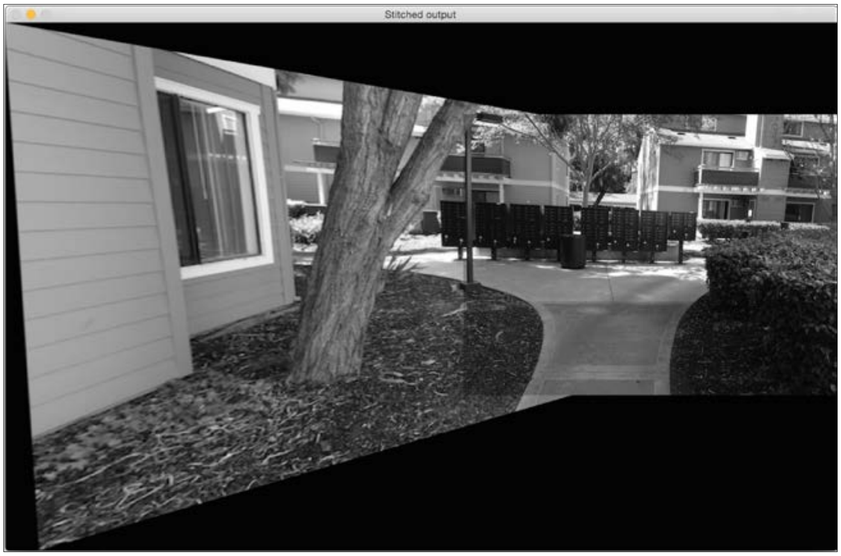
Now, let's consider another image of the same scene. It's at an angle with respect to the first image, and it's partially overlapping as well:



Let's consider the first image as our reference. If we stitch these images using our algorithm, it will look something like this:



If we keep the second image as our reference, it will look something like this:



Why does it look stretched?

If you observe, a portion of the output image corresponding to the query image looks stretched. It's because the query image is transformed and adjusted to fit into our frame of reference. The reason it looks stretched is because of the following lines in our code:

```
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
result = warpImages(img2, img1, M)
```

Since the images are at an angle with respect to each other, the query image will have to undergo a perspective transformation in order to fit into the frame of reference. So, we transform the query image first, and then stitch it into our main image to form the panoramic image.

Summary

In this chapter, we learned how to match keypoints among multiple images. We discussed how to stitch multiple images together to create a panoramic image. We learned how to deal with images that are not on the same plane.

In the next chapter, we are going to discuss how to do content-aware image resizing by detecting "interesting" regions in the image.

6
Seam Carving

In this chapter, we are going to learn about content-aware image resizing, which is also known as seam carving. We will discuss how to detect "interesting" parts in an image and how to use that information to resize a given image without deteriorating those interesting parts.

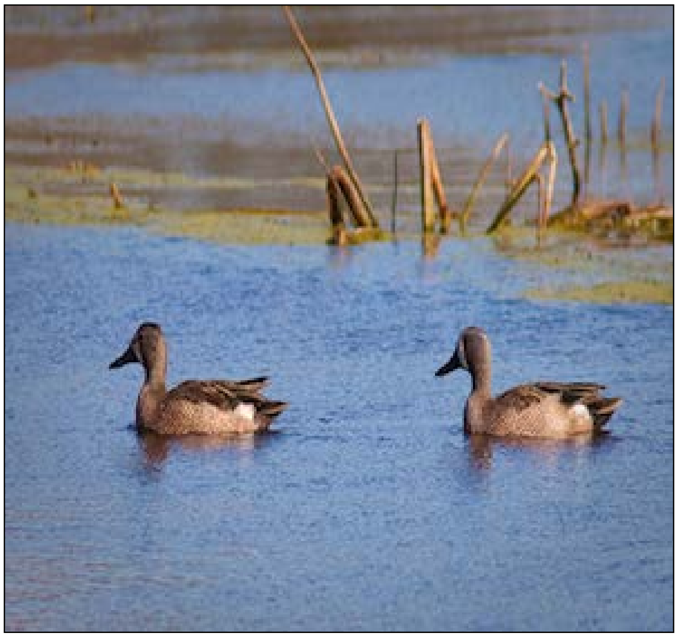
By the end of this chapter, you will know:

- What is content awareness
- How to quantify "interesting" parts in an image
- How to use dynamic programming for image content analysis
- How to increase and decrease the width of an image without deteriorating the interesting regions while keeping the height constant
- How to make an object disappear from an image

Why do we care about seam carving?

Before we start our discussion about seam carving, we need to understand why it is needed in the first place. Why should we care about the image content? Why can't we just resize the given image and move on with our lives? Well, to answer that question, let's consider the following image:

Now, let's say we want to reduce the width of this image while keeping the height constant. If you do that, it will look something like this:

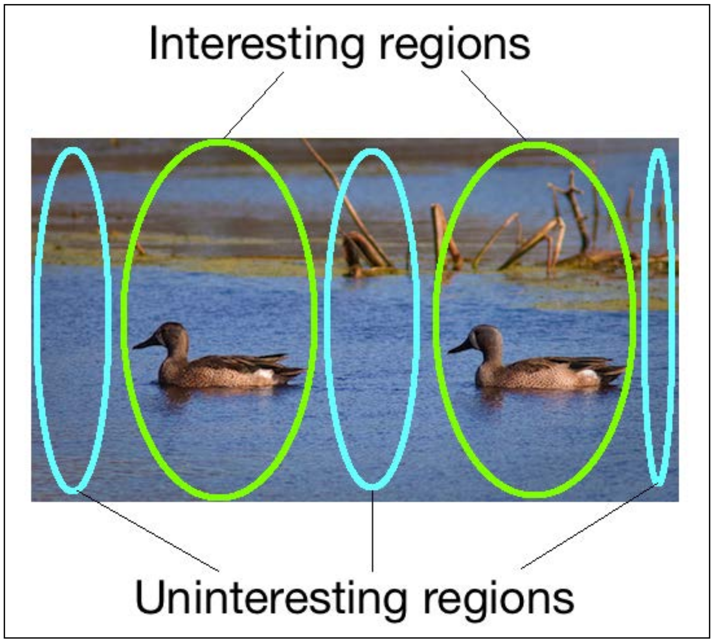


As you can see, the ducks in the image look skewed, and there's degradation in the overall quality of the image. Intuitively speaking, we can say that the ducks are the "interesting" parts in the image. So when we resize it, we want the ducks to be intact. This is where seam carving comes into the picture. Using seam carving, we can detect these interesting regions and make sure they don't get degraded.

How does it work?

We have been talking about image resizing and how we should consider the image's content when we resize it. So, why on earth is it called seam carving? It should just be called content-aware image resizing, right? Well, there are many different terms that are used to describe this process, such as image retargeting, liquid scaling, seam carving, and so on. The reason it's called seam carving is because of the way we resize the image. The algorithm was proposed by Shai Avidan and Ariel Shamir. You can refer to the original paper at `http://dl.acm.org/citation.cfm?id=1276390`.

We know that the goal is to resize the given image and keep the interesting content intact. So, we do that by finding the paths of least importance in that image. These paths are called seams. Once we find these seams, we remove them from the image to obtain a rescaled image. This process of removing, or "carving", will eventually result in a resized image. This is the reason we call it "seam carving". Consider the image that follows:



In the preceding image, we can see how we can roughly divide the image into interesting and uninteresting parts. We need to make sure that our algorithm detects these uninteresting parts and removes them. Let's consider the ducks image and the constraints we have to work with. We need to keep the height constant. This means that we need to find vertical seams in the image and remove them. These seams start at the top and end at the bottom (or vice versa). If we were dealing with vertical resizing, then the seams would start on the left-hand side and end on the right. A vertical seam is just a bunch of connected pixels starting at the top row and ending at the last row in the image.

How do we define "interesting"?

Before we start computing the seams, we need to find out what metric we will be using to compute these seams. We need a way to assign "importance" to each pixel so that we can find out the paths that are least important. In computer vision terminology, we say that we need to assign an energy value to each pixel so that we can find the path of minimum energy. Coming up with a good way to assign the energy value is very important because it will affect the quality of the output.

One of the metrics that we can use is the value of the derivative at each point. This is a good indicator of the level of activity in that neighborhood. If there is some activity, then the pixel values will change rapidly. Hence the value of the derivative at that point would be high. On the other hand, if the region were plain and uninteresting, then the pixel values wouldn't change as rapidly. So, the value of the derivative at that point in the grayscale image would be low.

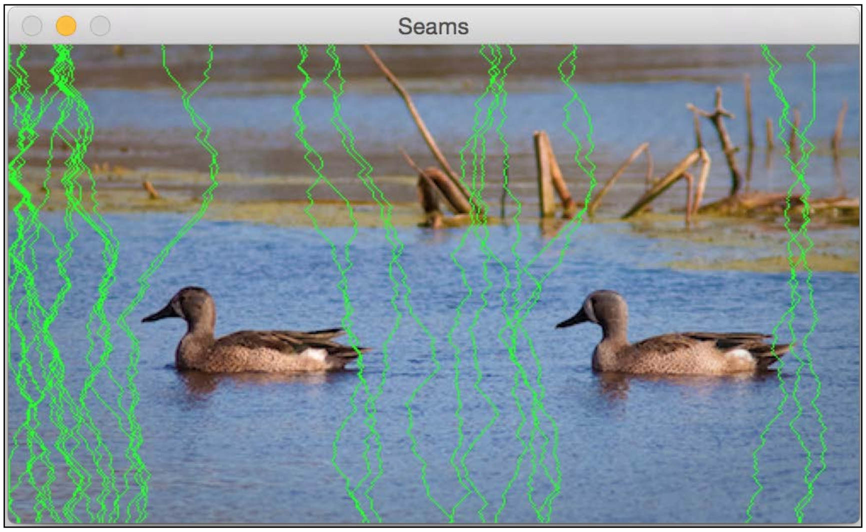
For each pixel location, we compute the energy by summing up the X and Y derivatives at that point. We compute the derivatives by taking the difference between the current pixel and its neighbors. If you recall, we did something similar to this when we were doing edge detection using **Sobel Filter** in *Chapter 1*, *Detecting Edges and Applying Image Filters*. Once we compute these values, we store them in a matrix called the energy matrix.

How do we compute the seams?

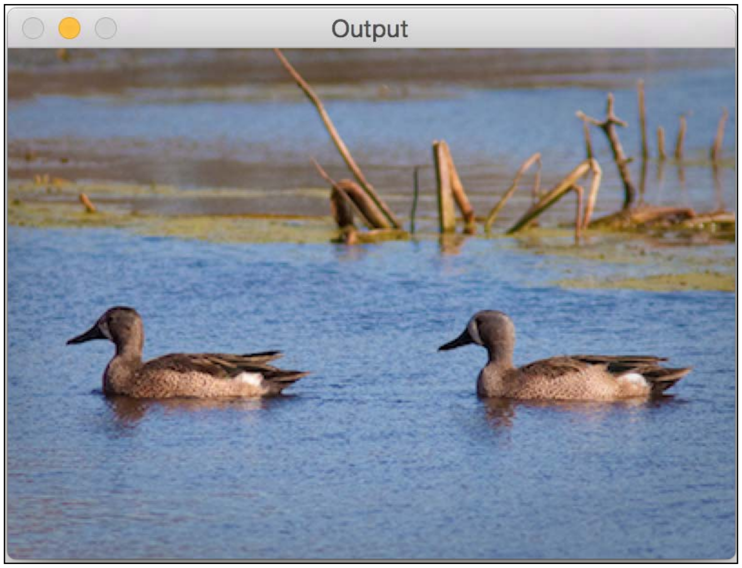
Now that we have the energy matrix, we are ready to compute the seams. We need to find the path through the image with the least energy. Computing all the possible paths is prohibitively expensive, so we need to find a smarter way to do this. This is where dynamic programming comes into the picture. In fact, seam carving is a direct application of dynamic programming. We need to start with each pixel in the first row and find our way to the last row. In order to find the path of least energy, we compute and store the best paths to each pixel in a table. Once we've construct this table, the path to a particular pixel can be found by backtracking through the rows in that table.

For each pixel in the current row, we calculate the energy of three possible pixel locations in the next row that we can move to, that is, bottom left, bottom, and bottom right. We keep repeating this process until we reach the bottom. Once we reach the bottom, we take the one with the least cumulative value and backtrack our way to the top. This will give us the path of least energy. Every time we remove a seam, the width of the image decreases by `1`. So we need to keep removing these seams until we arrive at the required image size.

Let's consider our ducks image again. If you compute the first 30 seams, it will look something like this:



These green lines indicate the paths of least importance. As we can see here, they carefully go around the ducks to make sure that the interesting regions are not touched. In the upper half of the image, the seams go around the twigs so that the quality is preserved. Technically speaking, the twigs are also "interesting". If you continue and remove the first 100 seams, it will look something like this:



Now, compare this with the naively resized image. Doesn't it look much better? The ducks look nice in this image.

Let's take a look at the code and see how to do it:

```
import sys

import cv2
import numpy as np

# Draw vertical seam on top of the image
def overlay_vertical_seam(img, seam):
    img_seam_overlay = np.copy(img) x

    # Extract the list of points from the seam
    x_coords, y_coords = np.transpose([(i,int(j)) for i,j in
enumerate(seam)])

    # Draw a green line on the image using the list of points
    img_seam_overlay[x_coords, y_coords] = (0,255,0)
    return img_seam_overlay

# Compute the energy matrix from the input image
def compute_energy_matrix(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Compute X derivative of the image
    sobel_x = cv2.Sobel(gray,cv2.CV_64F,1,0,ksize=3)

    # Compute Y derivative of the image
    sobel_y = cv2.Sobel(gray,cv2.CV_64F,0,1,ksize=3)

    abs_sobel_x = cv2.convertScaleAbs(sobel_x)
    abs_sobel_y = cv2.convertScaleAbs(sobel_y)

    # Return weighted summation of the two images i.e. 0.5*X +
0.5*Y
    return cv2.addWeighted(abs_sobel_x, 0.5, abs_sobel_y, 0.5, 0)

# Find vertical seam in the input image
def find_vertical_seam(img, energy):
    rows, cols = img.shape[:2]

    # Initialize the seam vector with 0 for each element
    seam = np.zeros(img.shape[0])

    # Initialize distance and edge matrices
    dist_to = np.zeros(img.shape[:2]) + sys.maxint
    dist_to[0,:] = np.zeros(img.shape[1])
    edge_to = np.zeros(img.shape[:2])

    # Dynamic programming; iterate using double loop and compute
the paths efficiently
    for row in xrange(rows-1):
        for col in xrange(cols):
            if col != 0:
                if dist_to[row+1, col-1] > dist_to[row, col] +
energy[row+1, col-1]:
                    dist_to[row+1, col-1] = dist_to[row, col] +
energy[row+1, col-1]
                    edge_to[row+1, col-1] = 1

            if dist_to[row+1, col] > dist_to[row, col] +
energy[row+1, col]:
```

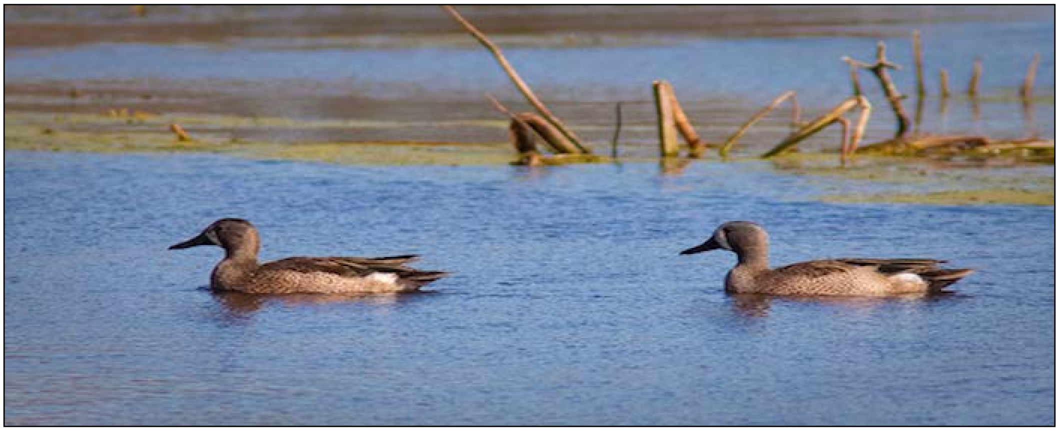
```
                dist_to[row+1, col] = dist_to[row, col] +
energy[row+1, col]
                edge_to[row+1, col] = 0

            if col != cols-1:
                if dist_to[row+1, col+1] > dist_to[row, col] +
energy[row+1, col+1]:
                    dist_to[row+1, col+1] = dist_to[row, col] +
energy[row+1, col+1]
                    edge_to[row+1, col+1] = -1

    # Retracing the path
    seam[rows-1] = np.argmin(dist_to[rows-1, :])
    for i in (x for x in reversed(xrange(rows)) if x > 0):
        seam[i-1] = seam[i] + edge_to[i, int(seam[i])]

    return seam

# Remove the input vertical seam from the image
def remove_vertical_seam(img, seam):
    rows, cols = img.shape[:2]

    # To delete a point, move every point after it one step
towards the left
    for row in xrange(rows):
        for col in xrange(int(seam[row]), cols-1):
            img[row, col] = img[row, col+1]

    # Discard the last column to create the final output image
    img = img[:, 0:cols-1]
    return img

if __name__=='__main__':
    # Make sure the size of the input image is reasonable.
    # Large images take a lot of time to be processed.
    # Recommended size is 640x480.
    img_input = cv2.imread(sys.argv[1])

    # Use a small number to get started. Once you get an
    # idea of the processing time, you can use a bigger number.
    # To get started, you can set it to 20.
    num_seams = int(sys.argv[2])
```

```
        img = np.copy(img_input)
        img_overlay_seam = np.copy(img_input)
        energy = compute_energy_matrix(img)

        for i in xrange(num_seams):
            seam = find_vertical_seam(img, energy)
            img_overlay_seam = overlay_vertical_seam(img_overlay_seam, 
    seam)
            img = remove_vertical_seam(img, seam)
            energy = compute_energy_matrix(img)
            print 'Number of seams removed =', i+1

        cv2.imshow('Input', img_input)
        cv2.imshow('Seams', img_overlay_seam)
        cv2.imshow('Output', img)
        cv2.waitKey()
```

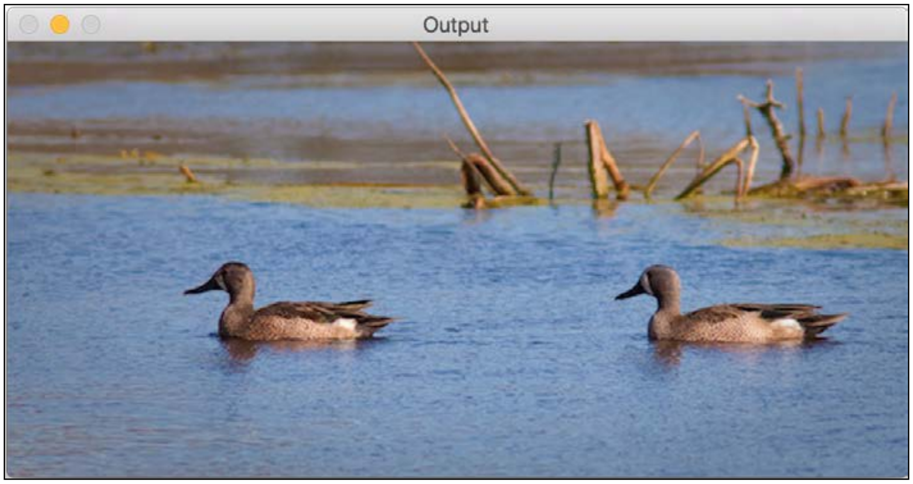
Can we expand an image?

We know that we can use seam carving to reduce the width of an image without deteriorating the interesting regions. So naturally, we need to ask ourselves if we can expand an image without deteriorating the interesting regions? As it turns out, we can do it using the same logic. When we compute the seams, we just need to add an extra column instead of deleting it.

If you expand the ducks image naively, it will look something like this:



If you do it in a smarter way, that is, by using seam carving, it will look something like this:



As you can see here, the width of the image has increased and the ducks don't look stretched. Following is the code to do it:

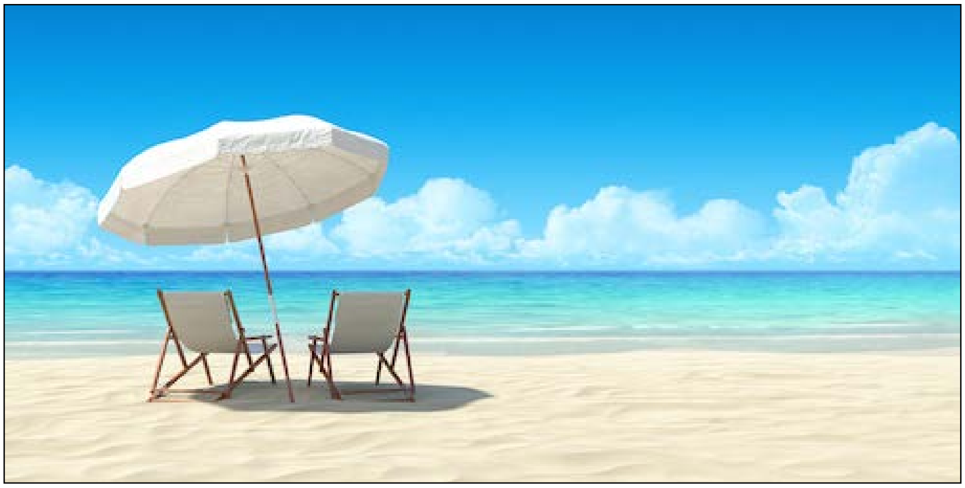
```
import sys

import cv2
import numpy as np

# Compute the energy matrix from the input image
def compute_energy_matrix(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sobel_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    abs_sobel_x = cv2.convertScaleAbs(sobel_x)
    abs_sobel_y = cv2.convertScaleAbs(sobel_y)
    return cv2.addWeighted(abs_sobel_x, 0.5, abs_sobel_y, 0.5, 0)

# Find the vertical seam
def find_vertical_seam(img, energy):
    rows, cols = img.shape[:2]

    # Initialize the seam vector with 0 for each element
    seam = np.zeros(img.shape[0])
```

```
        # Initialize distance and edge matrices
        dist_to = np.zeros(img.shape[:2]) + sys.maxint
        dist_to[0,:] = np.zeros(img.shape[1])
        edge_to = np.zeros(img.shape[:2])

        # Dynamic programming; iterate using double loop and compute
        #the paths efficiently
        for row in xrange(rows-1):
            for col in xrange(cols):
                if col != 0:
                    if dist_to[row+1, col-1] > dist_to[row, col] +
                    energy[row+1, col-1]:
                        dist_to[row+1, col-1] = dist_to[row, col] +
                        energy[row+1, col-1]
                        edge_to[row+1, col-1] = 1

                if dist_to[row+1, col] > dist_to[row, col] +
                energy[row+1, col]:
                    dist_to[row+1, col] = dist_to[row, col] +
                    energy[row+1, col]
                    edge_to[row+1, col] = 0

                if col != cols-1:
                    if dist_to[row+1, col+1] > dist_to[row, col] +
                    energy[row+1, col+1]:
                        dist_to[row+1, col+1] = dist_to[row, col] +
                        energy[row+1, col+1]
                        edge_to[row+1, col+1] = -1

        # Retracing the path
        seam[rows-1] = np.argmin(dist_to[rows-1, :])
        for i in (x for x in reversed(xrange(rows)) if x > 0):
            seam[i-1] = seam[i] + edge_to[i, int(seam[i])]

        return seam

    # Add a vertical seam to the image
    def add_vertical_seam(img, seam, num_iter):
        seam = seam + num_iter
        rows, cols = img.shape[:2]
        zero_col_mat = np.zeros((rows,1,3), dtype=np.uint8)
        img_extended = np.hstack((img, zero_col_mat))
        for row in xrange(rows):
            for col in xrange(cols, int(seam[row]), -1):
                img_extended[row, col] = img[row, col-1]

            # To insert a value between two columns, take the average
            # value of the neighbors. It looks smooth this way and we
            # can avoid unwanted artifacts.
            for i in range(3):
                v1 = img_extended[row, int(seam[row])-1, i]
                v2 = img_extended[row, int(seam[row])+1, i]
                img_extended[row, int(seam[row]), i] =
                (int(v1)+int(v2))/2

        return img_extended

    # Remove vertical seam from the image
    def remove_vertical_seam(img, seam):
        rows, cols = img.shape[:2]
        for row in xrange(rows):
            for col in xrange(int(seam[row]), cols-1):
                img[row, col] = img[row, col+1]

        img = img[:, 0:cols-1]
        return img

    if __name__=='__main__':
        img_input = cv2.imread(sys.argv[1])
        num_seams = int(sys.argv[2])
        img = np.copy(img_input)
        img_output = np.copy(img_input)
        energy = compute_energy_matrix(img)

        for i in xrange(num_seams):
            seam = find_vertical_seam(img, energy)
            img = remove_vertical_seam(img, seam)
            img_output = add_vertical_seam(img_output, seam, i)
            energy = compute_energy_matrix(img)
            print 'Number of seams added =', i+1

        cv2.imshow('Input', img_input)
        cv2.imshow('Output', img_output)
        cv2.waitKey()
```

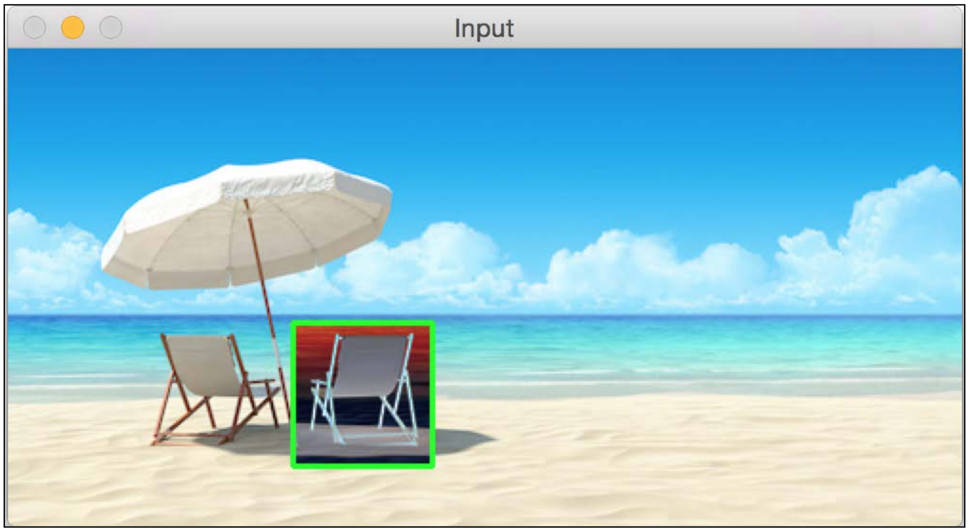
We added an extra function, `add_vertical_seam`, in this code. We use it to add vertical seams so that the image looks natural.

Can we remove an object completely?

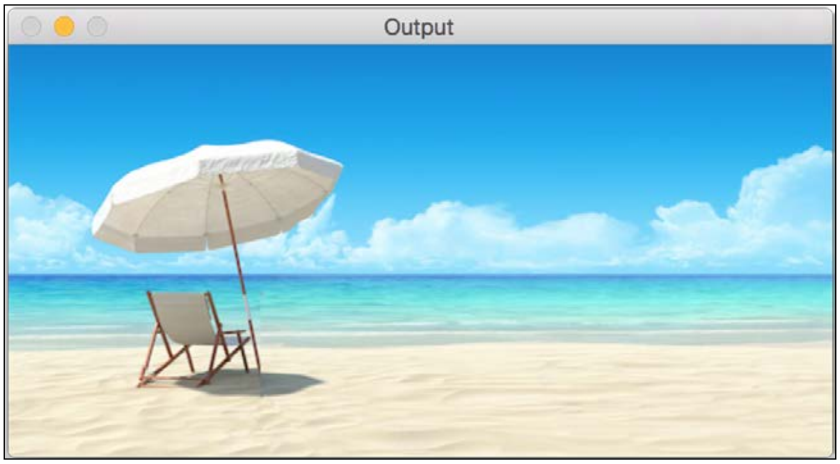
This is perhaps the most interesting application of seam carving. We can make an object completely disappear from an image. Let's consider the following image:



Let's select the region of interest:



After you remove the chair on the right, it will look something like this:



It's as if the chair never existed! Before we look at the code, it's important to know that this takes a while to run. So, just wait for a couple of minutes to get an idea of the processing time. You can adjust the input image size accordingly! Let's take a look at the code:

```
import sys

import cv2
import numpy as np

# Draw rectangle on top of the input image
def draw_rectangle(event, x, y, flags, params):
    global x_init, y_init, drawing, top_left_pt, bottom_right_pt,
    img_orig

    # Detecting a mouse click
    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        x_init, y_init = x, y

    # Detecting mouse movement
    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing:
            top_left_pt, bottom_right_pt = (x_init,y_init), (x,y)
            img[y_init:y, x_init:x] = 255 - img_orig[y_init:y,
            x_init:x]
            cv2.rectangle(img, top_left_pt, bottom_right_pt,
            (0,255,0), 2)

    # Detecting the mouse button up event
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        top_left_pt, bottom_right_pt = (x_init,y_init), (x,y)

        # Create the "negative" film effect for the selected
        # region
        img[y_init:y, x_init:x] = 255 - img[y_init:y, x_init:x]

        # Draw rectangle around the selected region
        cv2.rectangle(img, top_left_pt, bottom_right_pt,
        (0,255,0), 2)
        rect_final = (x_init, y_init, x-x_init, y-y_init)

        # Remove the object in the selected region
        remove_object(img_orig, rect_final)

# Computing the energy matrix using modified algorithm
def compute_energy_matrix_modified(img, rect_roi):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Compute the X derivative
    sobel_x = cv2.Sobel(gray,cv2.CV_64F,1,0,ksize=3)

    # Compute the Y derivative
    sobel_y = cv2.Sobel(gray,cv2.CV_64F,0,1,ksize=3)
    abs_sobel_x = cv2.convertScaleAbs(sobel_x)
    abs_sobel_y = cv2.convertScaleAbs(sobel_y)

    # Compute weighted summation i.e. 0.5*X + 0.5*Y
    energy_matrix = cv2.addWeighted(abs_sobel_x, 0.5, abs_sobel_y,
    0.5, 0)
    x,y,w,h = rect_roi

    # We want the seams to pass through this region, so make sure the
energy values in this region are set to 0
    energy_matrix[y:y+h, x:x+w] = 0

    return energy_matrix
```

```
# Compute energy matrix
def compute_energy_matrix(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Compute X derivative
    sobel_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)

    # Compute Y derivative
    sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    abs_sobel_x = cv2.convertScaleAbs(sobel_x)
    abs_sobel_y = cv2.convertScaleAbs(sobel_y)

    # Return weighted summation i.e. 0.5*X + 0.5*Y
    return cv2.addWeighted(abs_sobel_x, 0.5, abs_sobel_y, 0.5, 0)

# Find the vertical seam
def find_vertical_seam(img, energy):
    rows, cols = img.shape[:2]

    # Initialize the seam vector
    seam = np.zeros(img.shape[0])

    # Initialize the distance and edge matrices
    dist_to = np.zeros(img.shape[:2]) + sys.maxint
    dist_to[0,:] = np.zeros(img.shape[1])
    edge_to = np.zeros(img.shape[:2])

    # Dynamic programming; using double loop to compute the paths
    for row in xrange(rows-1):
        for col in xrange(cols):
            if col != 0:
                if dist_to[row+1, col-1] > dist_to[row, col] +
                energy[row+1, col-1]:
                    dist_to[row+1, col-1] = dist_to[row, col] +
                    energy[row+1, col-1]
                    edge_to[row+1, col-1] = 1

            if dist_to[row+1, col] > dist_to[row, col] +
            energy[row+1, col]:
                dist_to[row+1, col] = dist_to[row, col] +
                energy[row+1, col]
                edge_to[row+1, col] = 0
            if col != cols-1:
                if dist_to[row+1, col+1] > dist_to[row, col] +
                energy[row+1, col+1]:
                    dist_to[row+1, col+1] = dist_to[row, col] +
                    energy[row+1, col+1]
                    edge_to[row+1, col+1] = -1

    # Retracing the path
    seam[rows-1] = np.argmin(dist_to[rows-1, :])
    for i in (x for x in reversed(xrange(rows)) if x > 0):
        seam[i-1] = seam[i] + edge_to[i, int(seam[i])]

    return seam

# Add vertical seam to the input image
def add_vertical_seam(img, seam, num_iter):
    seam = seam + num_iter
    rows, cols = img.shape[:2]
    zero_col_mat = np.zeros((rows,1,3), dtype=np.uint8)
    img_extended = np.hstack((img, zero_col_mat))

    for row in xrange(rows):
        for col in xrange(cols, int(seam[row]), -1):
            img_extended[row, col] = img[row, col-1]

        # To insert a value between two columns, take the average
        # value of the neighbors. It looks smooth this way and we
        # can avoid unwanted artifacts.
        for i in range(3):
            v1 = img_extended[row, int(seam[row])-1, i]
            v2 = img_extended[row, int(seam[row])+1, i]
            img_extended[row, int(seam[row]), i] = (int(v1)+int(v2))/2

    return img_extended

# Remove vertical seam
def remove_vertical_seam(img, seam):
    rows, cols = img.shape[:2]
    for row in xrange(rows):
        for col in xrange(int(seam[row]), cols-1):
            img[row, col] = img[row, col+1]

    img = img[:, 0:cols-1]
    return img
```

```
# Remove the object from the input region of interest
def remove_object(img, rect_roi):
    num_seams = rect_roi[2] + 10
    energy = compute_energy_matrix_modified(img, rect_roi)

    # Start a loop and remove one seam at a time
    for i in xrange(num_seams):
        # Find the vertical seam that can be removed
        seam = find_vertical_seam(img, energy)

        # Remove that vertical seam
        img = remove_vertical_seam(img, seam)
        x,y,w,h = rect_roi

        # Compute energy matrix after removing the seam
        energy = compute_energy_matrix_modified(img, (x,y,w-i,h))
        print 'Number of seams removed =', i+1

    img_output = np.copy(img)

    # Fill up the region with surrounding values so that the size
    # of the image remains unchanged
    for i in xrange(num_seams):
        seam = find_vertical_seam(img, energy)
        img = remove_vertical_seam(img, seam)
        img_output = add_vertical_seam(img_output, seam, i)
        energy = compute_energy_matrix(img)
        print 'Number of seams added =', i+1

    cv2.imshow('Input', img_input)
    cv2.imshow('Output', img_output)
    cv2.waitKey()

if __name__=='__main__':
    img_input = cv2.imread(sys.argv[1])

    drawing = False
    img = np.copy(img_input)
    img_orig = np.copy(img_input)

    cv2.namedWindow('Input')
    cv2.setMouseCallback('Input', draw_rectangle)
```

```
while True:
    cv2.imshow('Input', img)
    c = cv2.waitKey(10)
    if c == 27:
        break

cv2.destroyAllWindows()
```

How did we do it?

The basic logic remains the same here. We are using seam carving to remove an object. Once we select the region of interest, we make all the seams pass through this region. We do this by manipulating the energy matrix after every iteration. We have added a new function called `compute_energy_matrix_modified` to achieve this. Once we compute the energy matrix, we assign a value of 0 to this region of interest. This way, we force all the seams to pass through this area. After we remove all the seams related to this region, we keep adding the seams until we expand the image to its original width.

Summary

In this chapter, we learned about content-aware image resizing. We discussed how to quantify interesting and uninteresting regions in an image. We learned how to compute seams in an image and how to use dynamic programming to do it efficiently. We discussed how to use seam carving to reduce the width of an image, and how we can use the same logic to expand an image. We also learned how to remove an object from an image completely.

In the next chapter, we are going to discuss how to do shape analysis and image segmentation. We will see how to use those principles to find the exact boundaries of an object of interest in the image.

7

Detecting Shapes and Segmenting an Image

In this chapter, we are going to learn about shape analysis and image segmentation. We will learn how to recognize shapes and estimate the exact boundaries. We will discuss how to segment an image into its constituent parts using various methods. We will learn how to separate the foreground from the background as well.

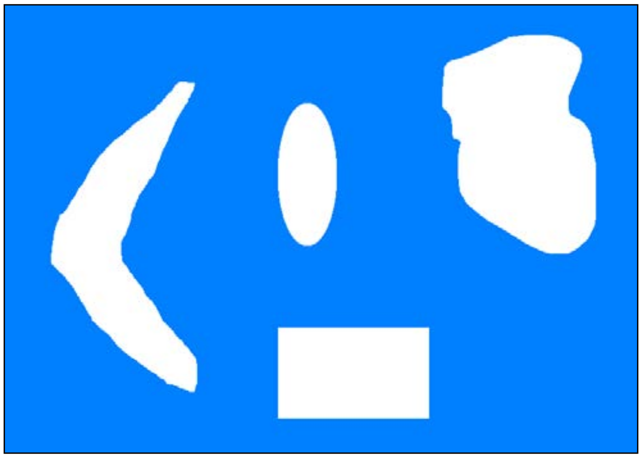
By the end of this chapter, you will know:

- What is contour analysis and shape matching
- How to match shapes
- What is image segmentation
- How to segment an image into its constituent parts
- How to separate the foreground from the background
- How to use various techniques to segment an image

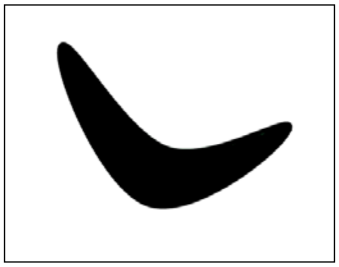
Contour analysis and shape matching

Contour analysis is a very useful tool in the field of computer vision. We deal with a lot of shapes in the real world and contour analysis helps in analyzing those shapes using various algorithms. When we convert an image to grayscale and threshold it, we are left with a bunch of lines and contours. Once we understand the properties of different shapes, we will be able to extract detailed information from an image.

Let's say we want to identify the boomerang shape in the following image:



In order to do that, we first need to know what a regular boomerang looks like:



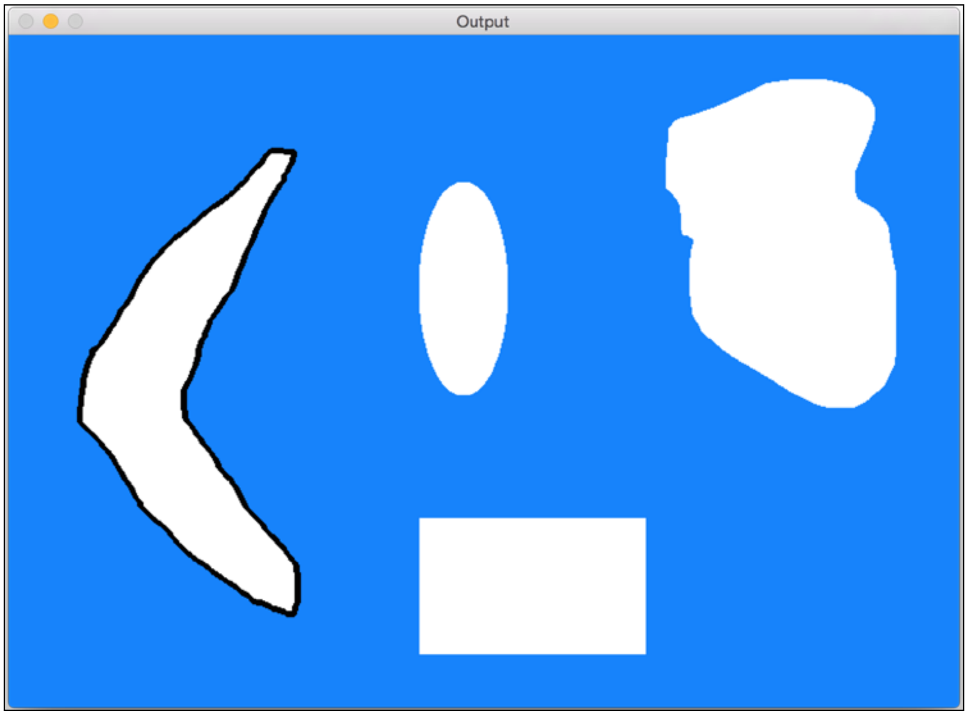
Now using the above image as a reference, can we identify what shape in our original image corresponds to a boomerang? If you notice, we cannot use a simple correlation based approach because the shapes are all distorted. This means that an approach where we look for an exact match won't work! We need to understand the properties of this shape and match the corresponding properties to identify the boomerang shape. OpenCV provides a nice shape matcher function that we can use to achieve this. The matching is based on the concept of Hu moment, which in turn is related to image moments. You can refer to the following paper to learn more about moments: `http://zoi.utia.cas.cz/files/chapter_moments_color1.pdf`. The concept of "image moments" basically refers to the weighted and power-raised summation of the pixels within a shape.

$$I = \sum_{i=0}^{N} w_i p_i^k$$

In the above equation, **p** refers to the pixels inside the contour, **w** refers to the weights, **N** refers to the number of points inside the contour, **k** refers to the power, and **I** refers to the moment. Depending on the values we choose for w and k, we can extract different characteristics from that contour.

Perhaps the simplest example is to compute the area of the contour. To do this, we need to count the number of pixels within that region. So mathematically speaking, in the weighted and power raised summation form, we just need to set w to 1 and k to 0. This will give us the area of the contour. Depending on how we compute these moments, they will help us in understanding these different shapes. This also gives rise to some interesting properties that help us in determining the shape similarity metric.

If we match the shapes, you will see something like this:



Let's take a look at the code to do this:

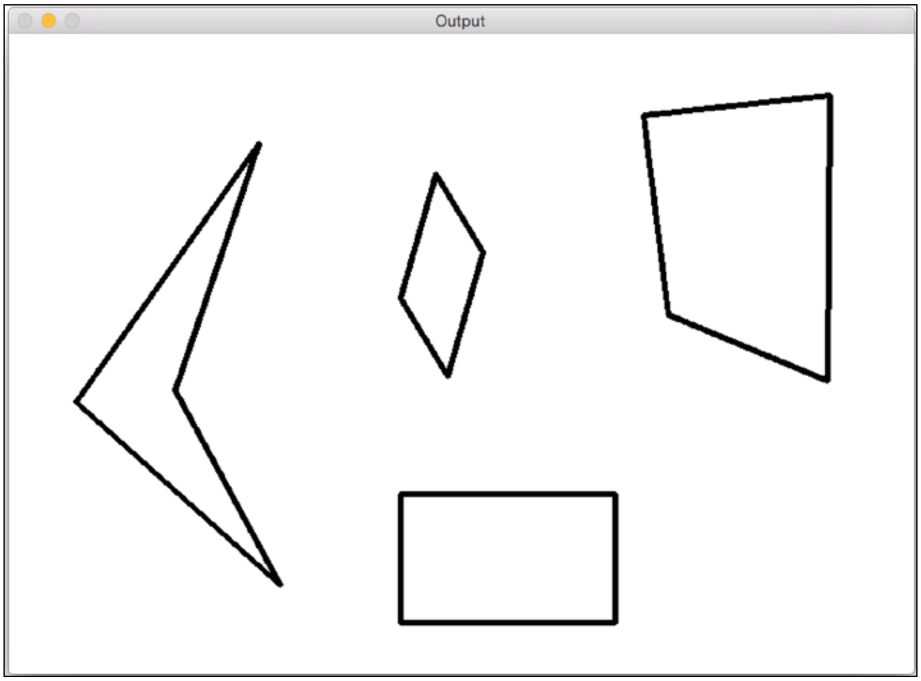
```
import sys

import cv2
import numpy as np
```

```
# Extract reference contour from the image
def get_ref_contour(img):
    ref_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(ref_gray, 127, 255, 0)

    # Find all the contours in the thresholded image. The values
    # for the second and third parameters are restricted to a
    # certain number of possible values. You can learn more
    # 'findContours' function here: http://docs.opencv.org/modules/
imgproc/doc/structural_analysis_and_shape_descriptors.html
    contours, hierarchy = cv2.findContours(thresh, 1, 2)

    # Extract the relevant contour based on area ratio. We use the
    # area ratio because the main image boundary contour is
    # extracted as well and we don't want that. This area ratio
    # threshold will ensure that we only take the contour inside
    # the image.
    for contour in contours:
        area = cv2.contourArea(contour)
        img_area = img.shape[0] * img.shape[1]
        if 0.05 < area/float(img_area) < 0.8:
            return contour

# Extract all the contours from the image
def get_all_contours(img):
    ref_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(ref_gray, 127, 255, 0)
    contours, hierarchy = cv2.findContours(thresh, 1, 2)
    return contours

if __name__=='__main__':
    # Boomerang reference image
    img1 = cv2.imread(sys.argv[1])

    # Input image containing all the different shapes
    img2 = cv2.imread(sys.argv[2])

    # Extract the reference contour
    ref_contour = get_ref_contour(img1)

    # Extract all the contours from the input image
    input_contours = get_all_contours(img2)

    closest_contour = input_contours[0]
    min_dist = sys.maxint
    # Finding the closest contour
    for contour in input_contours:
        # Matching the shapes and taking the closest one
        ret = cv2.matchShapes(ref_contour, contour, 1, 0.0)
        if ret < min_dist:
            min_dist = ret
            closest_contour = contour

    cv2.drawContours(img2, [closest_contour], -1, (0,0,0), 3)
    cv2.imshow('Output', img2)
    cv2.waitKey()
```

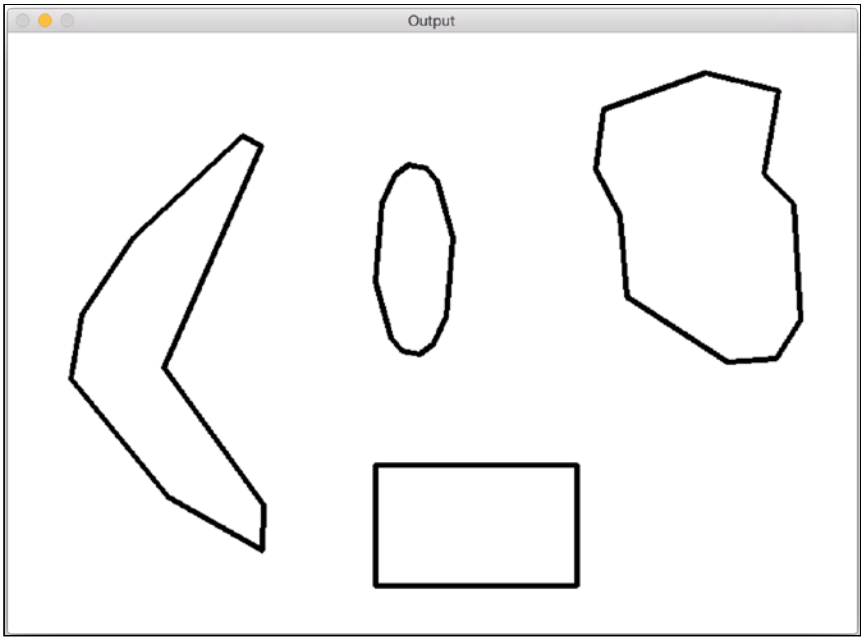
Approximating a contour

A lot of contours that we encounter in real life are noisy. This means that the contours don't look smooth, and hence our analysis takes a hit. So how do we deal with this? One way to go about this would be to get all the points on the contour and then approximate it with a smooth polygon.

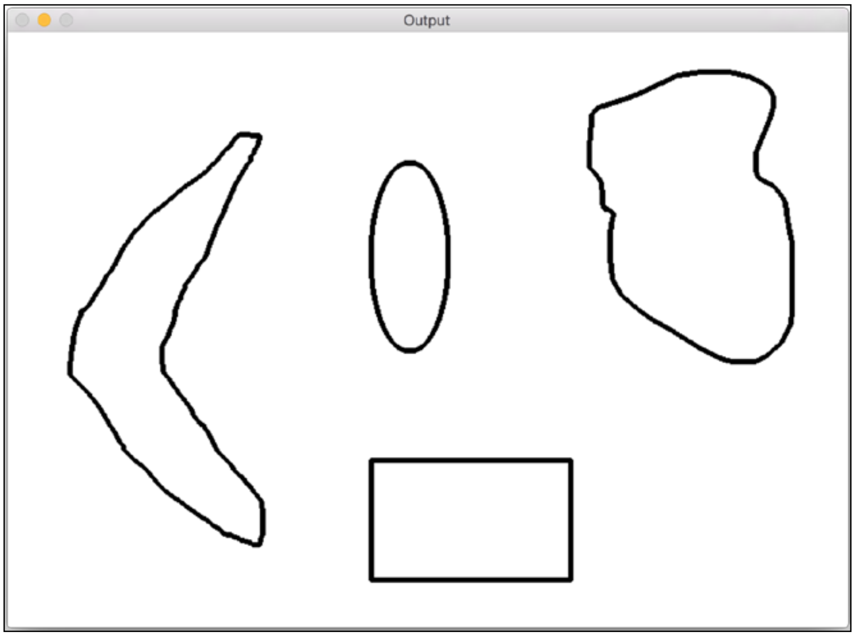
Let's consider the boomerang image again. If you approximate the contours using various thresholds, you will see the contours changing their shapes. Let's start with a factor of 0.05:



If you reduce this factor, the contours will get smoother. Let's make it 0.01:



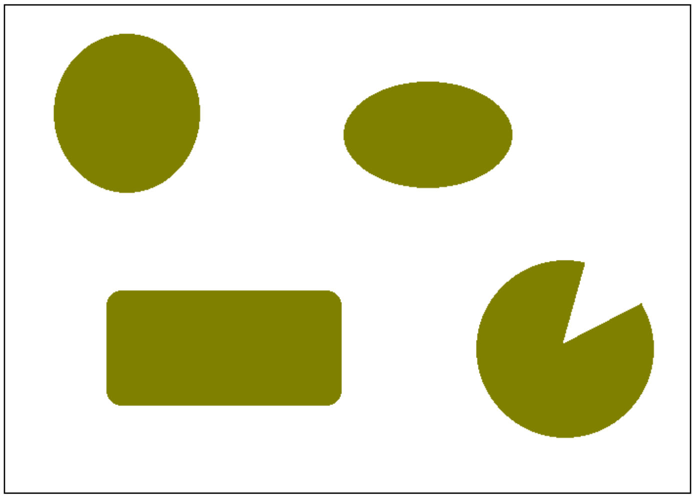
If you make it really small, say 0.00001, then it will look like the original image:



Identifying the pizza with the slice taken out

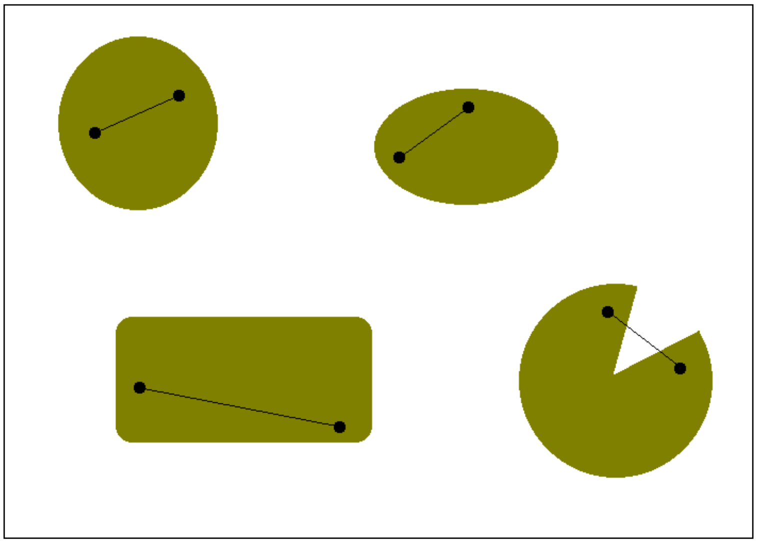
The title might be slightly misleading, because we will not be talking about pizza slices. But let's say you are in a situation where you have an image containing different types of pizzas with different shapes. Now, somebody has taken a slice out of one of those pizzas. How would we automatically identify this?

We cannot take the approach we took earlier because we don't know what the shape looks like. So we don't have any template. We are not even sure what shape we are looking for, so we cannot build a template based on any prior information. All we know is the fact that a slice has been taken from one of the pizzas. Let's consider the following image:



It's not exactly a real image, but you get the idea. You know what shape we are talking about. Since we don't know what we are looking for, we need to use some of the properties of these shapes to identify the sliced pizza. If you notice, all the other shapes are nicely closed. As in, you can take any two points within those shapes and draw a line between them, and that line will always lie within that shape. These kinds of shapes are called **convex shapes**.

If you look at the sliced pizza shape, we can choose two points such that the line between them goes outside the shape as shown in the figure that follows:

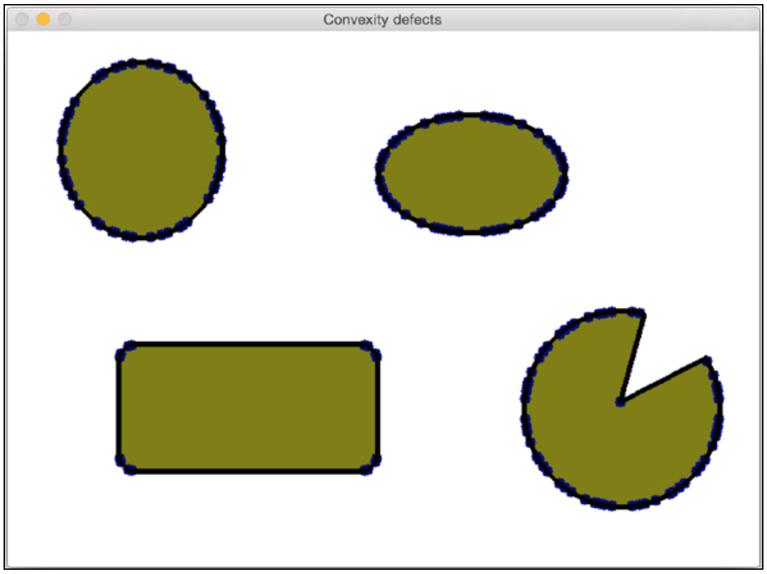


So, all we need to do is detect the non-convex shape in the image and we'll be done. Let's go ahead and do that:

```
import sys

import cv2
import numpy as np

# Input is a color image
def get_contours(img):
    # Convert the image to grayscale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Threshold the input image
    ret, thresh = cv2.threshold(img_gray, 127, 255, 0)

    # Find the contours in the above image
    contours, hierarchy = cv2.findContours(thresh, 2, 1)

    return contours

if __name__=='__main__':
    img = cv2.imread(sys.argv[1])
```

```
# Iterate over the extracted contours
for contour in get_contours(img):
    # Extract convex hull from the contour
    hull = cv2.convexHull(contour, returnPoints=False)

    # Extract convexity defects from the above hull
    defects = cv2.convexityDefects(contour, hull)

    if defects is None:
        continue

    # Draw lines and circles to show the defects
    for i in range(defects.shape[0]):
        start_defect, end_defect, far_defect, _ = defects[i,0]
        start = tuple(contour[start_defect][0])
        end = tuple(contour[end_defect][0])
        far = tuple(contour[far_defect][0])
        cv2.circle(img, far, 5, [128,0,0], -1)
        cv2.drawContours(img, [contour], -1, (0,0,0), 3)

cv2.imshow('Convexity defects',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

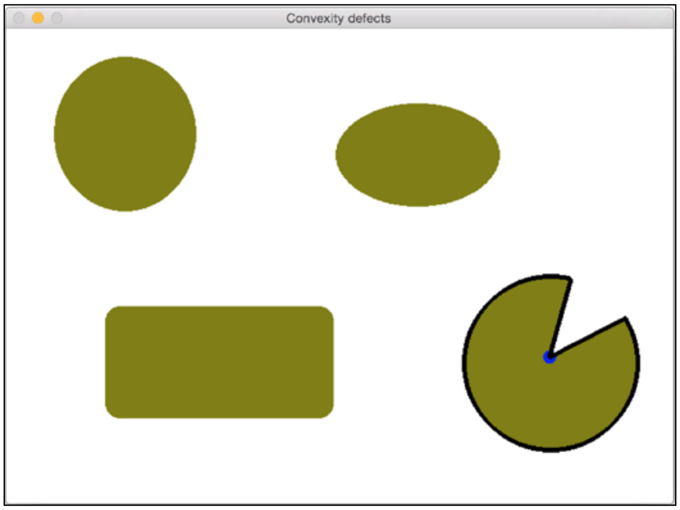
If you run the above code, you will see something like this:



Wait a minute, what happened here? It looks so cluttered. Did we do something wrong? As it turns out, the curves are not really smooth. If you observe closely, there are tiny ridges everywhere along the curves. So, if you just run your convexity detector, it's not going to work. This is where contour approximation comes in really handy. Once we've detected the contours, we need to smoothen them so that the ridges do not affect them. Let's go ahead and do that:

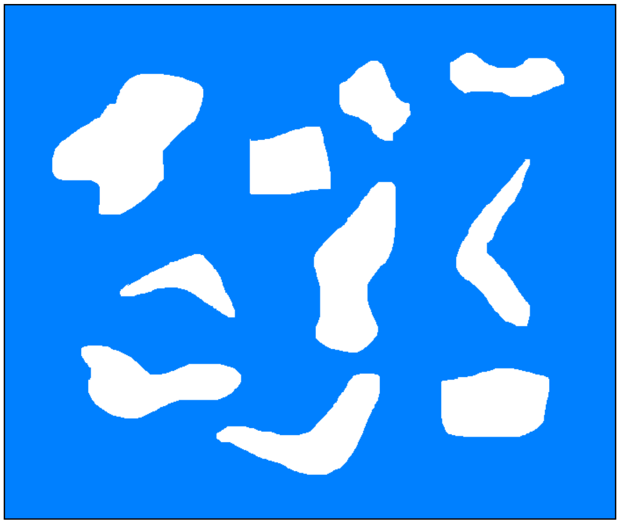
```
import sys

import cv2
import numpy as np

# Input is a color image
def get_contours(img):
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(img_gray, 127, 255, 0)
    contours, hierarchy = cv2.findContours(thresh, 2, 1)
    return contours

if __name__=='__main__':
    img = cv2.imread(sys.argv[1])

    # Iterate over the extracted contours
    for contour in get_contours(img):
        orig_contour = contour
        epsilon = 0.01 * cv2.arcLength(contour, True)
        contour = cv2.approxPolyDP(contour, epsilon, True)

        # Extract convex hull and the convexity defects
        hull = cv2.convexHull(contour, returnPoints=False)
        defects = cv2.convexityDefects(contour,hull)

        if defects is None:
            continue

        # Draw lines and circles to show the defects
        for i in range(defects.shape[0]):
            start_defect, end_defect, far_defect, _ = defects[i,0]
            start = tuple(contour[start_defect][0])
            end = tuple(contour[end_defect][0])
            far = tuple(contour[far_defect][0])
            cv2.circle(img, far, 7, [255,0,0], -1)
            cv2.drawContours(img, [orig_contour], -1, (0,0,0), 3)

    cv2.imshow('Convexity defects',img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

If you run the preceding code, the output will look like the following:

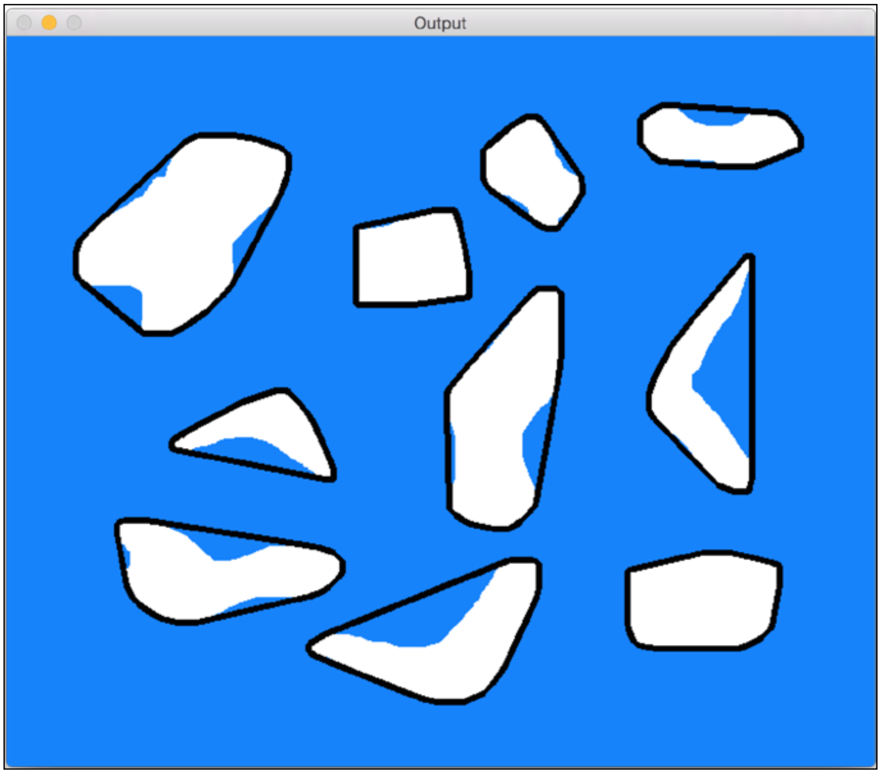


How to censor a shape?

Let's say you are dealing with images and you want to block out a particular shape. Now, you might say that you will use shape matching to identify the shape and then just block it out, right? But the problem here is that we don't have any template available. So, how do we go about doing this? Shape analysis comes in various forms, and we need to build our algorithm depending on the situation. Let's consider the following figure:

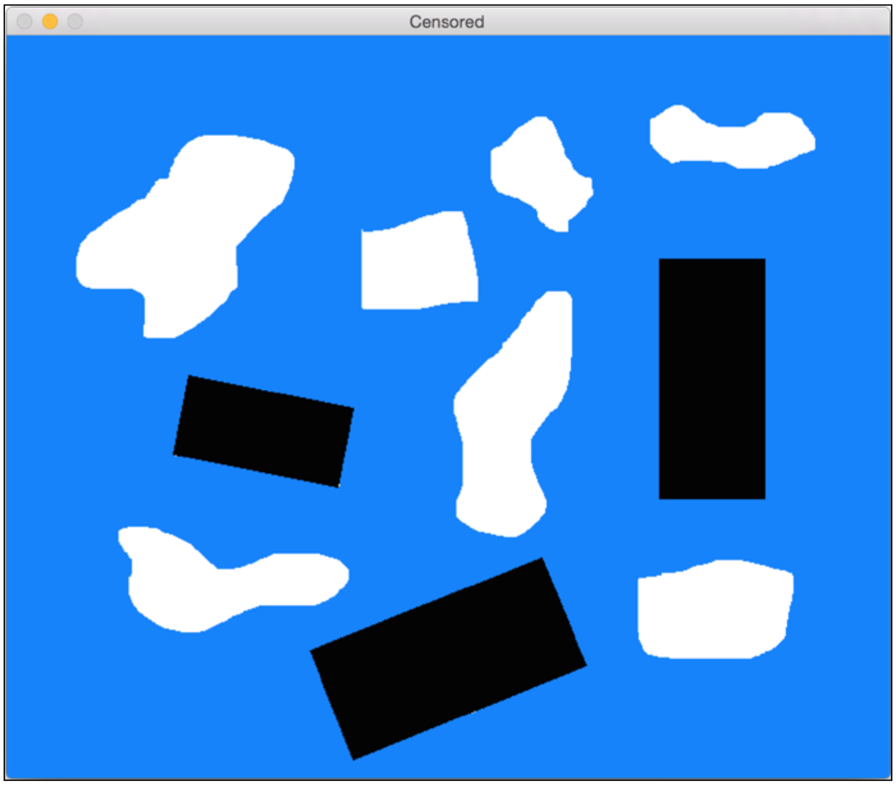


Let's say we want to identify all the boomerang shapes and then block them out without using any template images. As you can see, there are various other weird shapes in that image and the boomerang shapes are not really smooth. We need to identify the property that's going to differentiate the boomerang shape from the other shapes present. Let's consider the convex hull. If you take the ratio of the area of each shape to the area of the convex hull, we can see that this can be a distinguishing metric. This metric is called **solidity factor** in shape analysis. This metric will have a lower value for the boomerang shapes because of the empty area that will be left out, as shown in the following figure:



The black boundaries represent the convex hulls. Once we compute these values for all the shapes, how do separate them out? Can we just use a fixed threshold to detect the boomerang shapes? Not really! We cannot have a fixed threshold value because you never know what kind of shape you might encounter later. So, a better approach would be to use **K-Means clustering**. K-Means is an unsupervised learning technique that can be used to separate out the input data into K classes. You can quickly brush up on K-Means before proceeding further at `http://docs.opencv.org/master/de/d4d/tutorial_py_kmeans_understanding.html`.

We know that we want to separate the shapes into two groups, that is, boomerang shapes and other shapes. So, we know what our *K* will be in K-Means. Once we use that and cluster the values, we pick the cluster with the lowest solidity factor and that will give us our boomerang shapes. Bear in mind that this approach works only in this particular case. If you are dealing with other kinds of shapes, then you will have to use some other metrics to make sure that the shape detection works. As we discussed earlier, it depends heavily on the situation. If you detect the shapes and block them out, it will look like this:



Following is the code to do it:

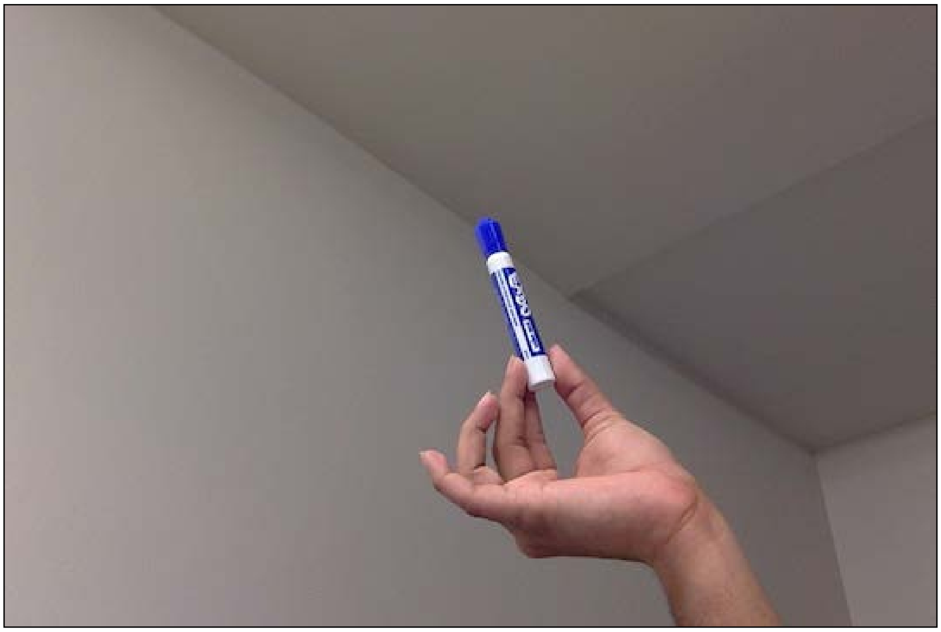
```
import sys

import cv2
import numpy as np

def get_all_contours(img):
    ref_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
        ret, thresh = cv2.threshold(ref_gray, 127, 255, 0)
        contours, hierarchy = cv2.findContours(thresh, 1, 2)
        return contours

    if __name__=='__main__':
        # Input image containing all the shapes
        img = cv2.imread(sys.argv[1])

        img_orig = np.copy(img)
        input_contours = get_all_contours(img)
        solidity_values = []

        # Compute solidity factors of all the contours
        for contour in input_contours:
            area_contour = cv2.contourArea(contour)
            convex_hull = cv2.convexHull(contour)
            area_hull = cv2.contourArea(convex_hull)
            solidity = float(area_contour)/area_hull
            solidity_values.append(solidity)

        # Clustering using KMeans
        criteria = (cv2.TERM_CRITERIA_EPS +
    cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
        flags = cv2.KMEANS_RANDOM_CENTERS
        solidity_values = np.array(solidity_values).reshape((len(solidity_
    values),1)).astype('float32')
        compactness, labels, centers = cv2.kmeans(solidity_values, 2,
    criteria, 10, flags)

        closest_class = np.argmin(centers)
        output_contours = []
        for i in solidity_values[labels==closest_class]:
            index = np.where(solidity_values==i)[0][0]
            output_contours.append(input_contours[index])

        cv2.drawContours(img, output_contours, -1, (0,0,0), 3)
        cv2.imshow('Output', img)

        # Censoring
        for contour in output_contours:
            rect = cv2.minAreaRect(contour)
            box = cv2.cv.BoxPoints(rect)
            box = np.int0(box)
            cv2.drawContours(img_orig,[box],0,(0,0,0),-1)

        cv2.imshow('Censored', img_orig)
        cv2.waitKey()
```

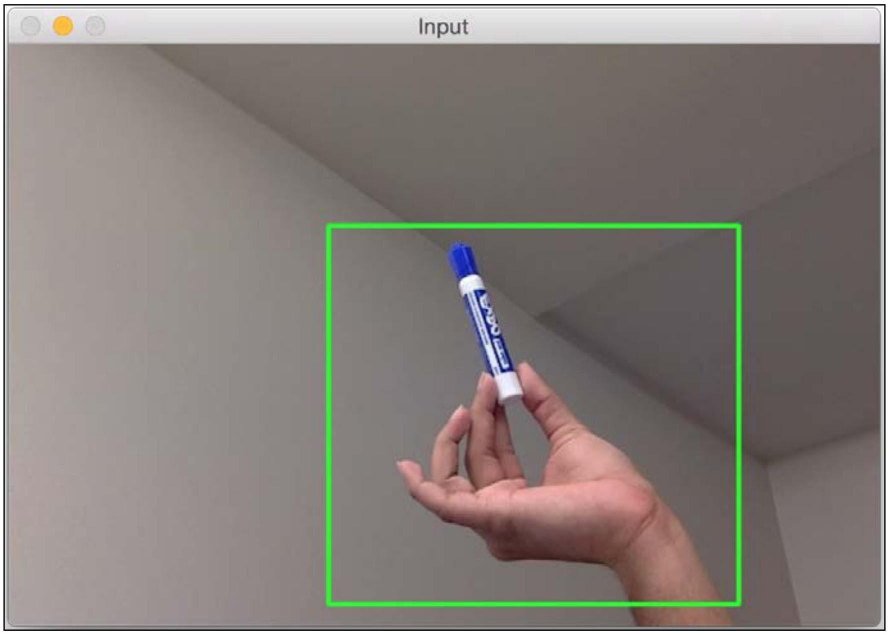
What is image segmentation?

Image segmentation is the process of separating an image into its constituent parts. It is an important step in many computer vision applications in the real world. There are many different ways of segmenting an image. When we segment an image, we separate the regions based on various metrics such as color, texture, location, and so on. All the pixels within each region have something in common, depending on the metric we are using. Let's take a look at some of the popular approaches here.

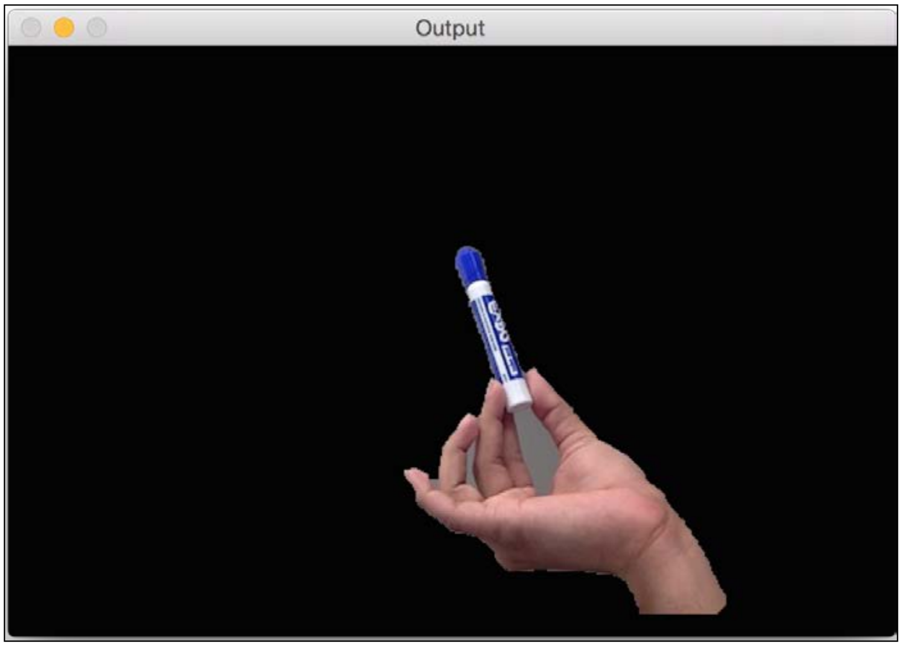
To start with, we will be looking at a technique called **GrabCut**. It is an image segmentation method based on a more generic approach called **graph-cuts**. In the graph-cuts method, we consider the entire image to be a graph, and then we segment the graph based on the strength of the edges in that graph. We construct the graph by considering each pixel to be a node and edges are constructed between the nodes, where edge weight is a function of the pixel values of those two nodes. Whenever there is a boundary, the pixel values are higher. Hence, the edge weights will also be higher. This graph is then segmented by minimizing the Gibss energy of the graph. This is analogous to finding the maximum entropy segmentation. You can refer to the original paper to learn more about it at `http://cvg.ethz.ch/teaching/cvl/2012/grabcut-siggraph04.pdf`. Let's consider the following image:



Let's select the region of interest:



Once the image has been segmented, it will look something like this:



Following is the code to do this:

```
import cv2
import numpy as np

# Draw rectangle based on the input selection
def draw_rectangle(event, x, y, flags, params):
    global x_init, y_init, drawing, top_left_pt, bottom_right_pt,
    img_orig

    # Detecting mouse button down event
    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        x_init, y_init = x, y

    # Detecting mouse movement
    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing:
            top_left_pt, bottom_right_pt = (x_init,y_init), (x,y)
            img[y_init:y, x_init:x] = 255 - img_orig[y_init:y,
            x_init:x]
            cv2.rectangle(img, top_left_pt, bottom_right_pt,
            (0,255,0), 2)

    # Detecting mouse button up event
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        top_left_pt, bottom_right_pt = (x_init,y_init), (x,y)
        img[y_init:y, x_init:x] = 255 - img[y_init:y, x_init:x]
        cv2.rectangle(img, top_left_pt, bottom_right_pt,
        (0,255,0), 2)
        rect_final = (x_init, y_init, x-x_init, y-y_init)

        # Run Grabcut on the region of interest
        run_grabcut(img_orig, rect_final)

# Grabcut algorithm
def run_grabcut(img_orig, rect_final):
    # Initialize the mask
    mask = np.zeros(img_orig.shape[:2],np.uint8)

    # Extract the rectangle and set the region of
    # interest in the above mask
    x,y,w,h = rect_final
    mask[y:y+h, x:x+w] = 1

    # Initialize background and foreground models
    bgdModel = np.zeros((1,65), np.float64)
    fgdModel = np.zeros((1,65), np.float64)

    # Run Grabcut algorithm
    cv2.grabCut(img_orig, mask, rect_final, bgdModel, fgdModel, 5,
    cv2.GC_INIT_WITH_RECT)

    # Extract new mask
    mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')

    # Apply the above mask to the image
    img_orig = img_orig*mask2[:,:,np.newaxis]

    # Display the image
    cv2.imshow('Output', img_orig)

if __name__=='__main__':
    drawing = False
    top_left_pt, bottom_right_pt = (-1,-1), (-1,-1)

    # Read the input image
    img_orig = cv2.imread(sys.argv[1])
    img = img_orig.copy()

    cv2.namedWindow('Input')
    cv2.setMouseCallback('Input', draw_rectangle)

    while True:
        cv2.imshow('Input', img)
        c = cv2.waitKey(1)
        if c == 27:
            break

    cv2.destroyAllWindows()
```

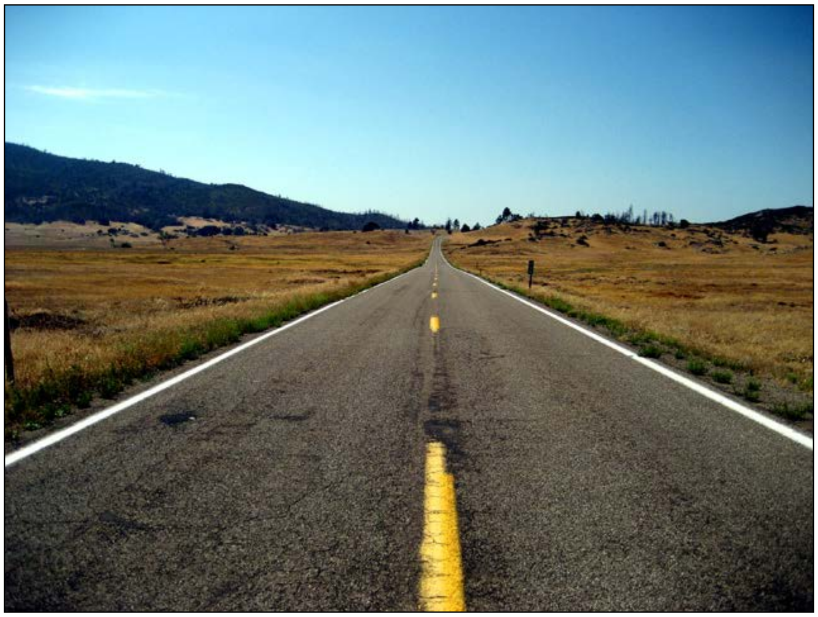
How does it work?

We start with the seed points specified by the user. This is the bounding box within which we have the object of interest. Underneath the surface, the algorithm estimates the color distribution of the object and the background. The algorithm represents the color distribution of the image as a **Gaussian Mixture Markov Random Field** (**GMMRF**). You can refer to the detailed paper to learn more about GMMRF at `http://research.microsoft.com/pubs/67898/eccv04-GMMRF.pdf`. We need the color distribution of both, the object and the background, because we will be using this knowledge to separate the object. This information is used to find the maximum entropy segmentation by applying the min-cut algorithm to the Markov Random Field. Once we have this, we use the graph cuts optimization method to infer the labels.

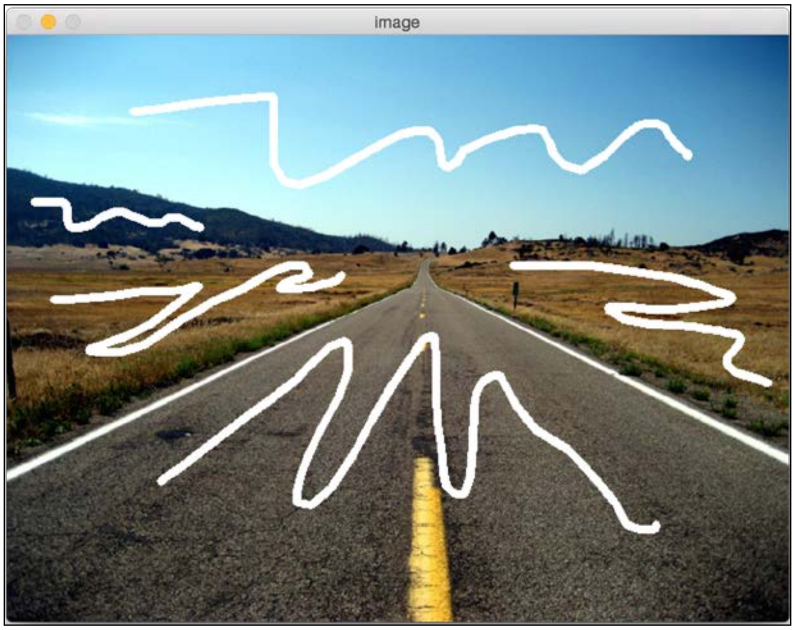
Watershed algorithm

OpenCV comes with a default implementation of the watershed algorithm. It's pretty famous and there are a lot of implementations available out there. You can read more about it at `http://docs.opencv.org/master/d3/db4/tutorial_py_watershed.html`. Since you already have access to the OpenCV source code, we will not be looking at the code here.

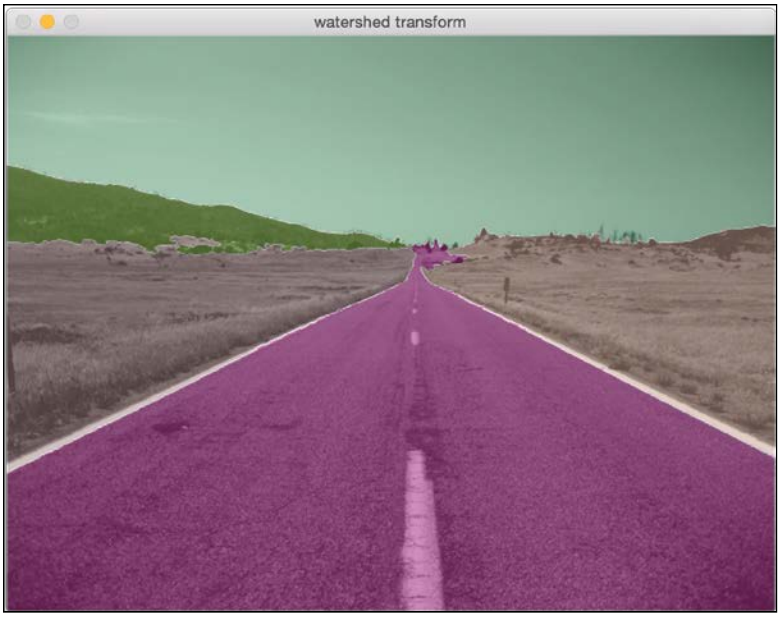
We will just see what the output looks like. Consider the following image:



Let's select the regions:



If you run the watershed algorithm on this, the output will look something like the following:



Summary

In this chapter, we learned about contour analysis and image segmentation. We learned how to match shapes based on a template. We learned about the various different properties of shapes and how we can use them to identify different kinds of shapes. We discussed image segmentation and how we can use graph-based methods to segment regions in an image. We briefly discussed watershed transformation as well.

In the next chapter, we are going to discuss how to track an object in a live video.

8
Object Tracking

In this chapter, we are going to learn about tracking an object in a live video. We will discuss the different characteristics that can be used to track an object. We will also learn about the different methods and techniques for object tracking.

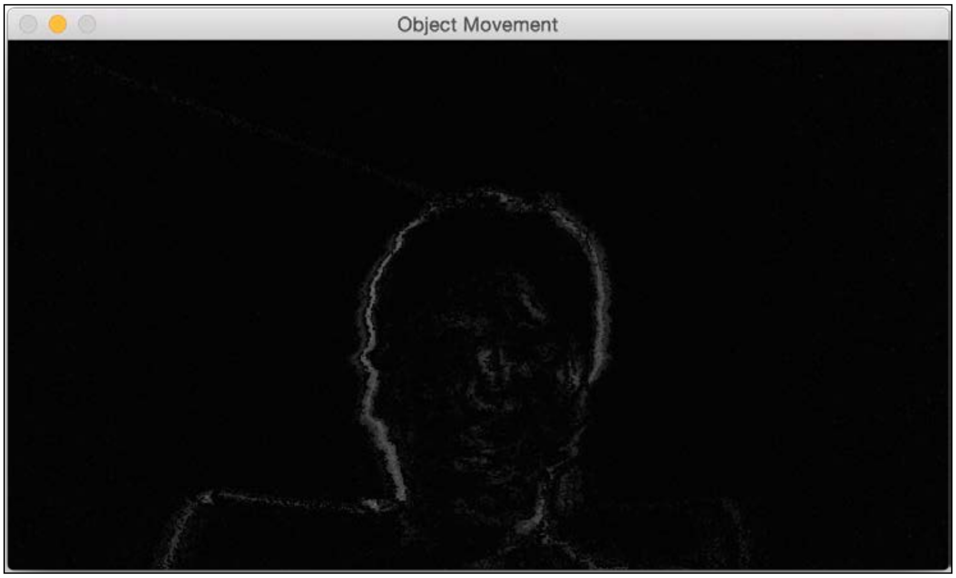
By the end of this chapter, you will know:

- How to use frame differencing
- How to use colorspaces to track colored objects
- How to build an interactive object tracker
- How to build a feature tracker
- How to build a video surveillance system

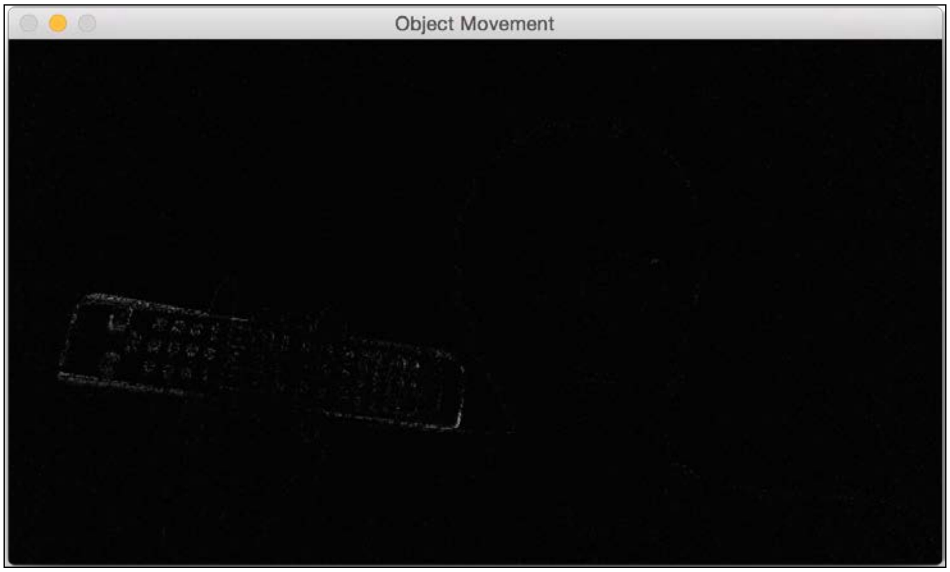
Frame differencing

This is, possibly, the simplest technique we can use to see what parts of the video are moving. When we consider a live video stream, the difference between successive frames gives us a lot of information. The concept is fairly straightforward! We just take the difference between successive frames and display the differences.

If I move my laptop rapidly from left to right, we will see something like this:



If I rapidly move the TV remote in my hand, it will look something like this:



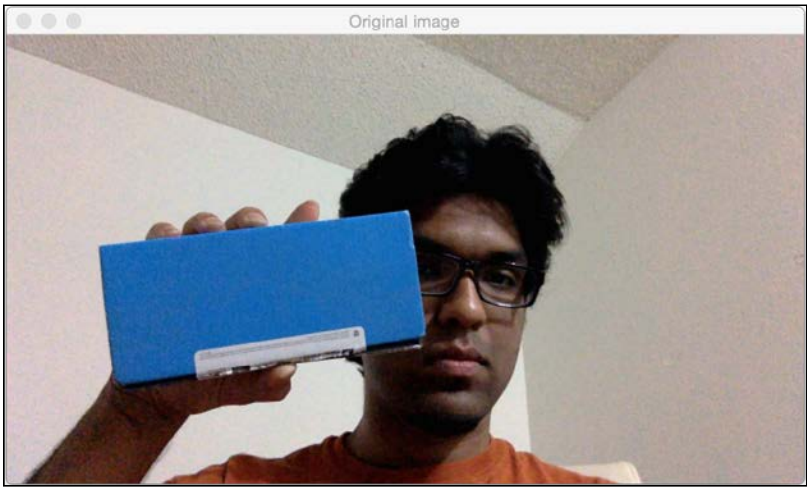
As you can see from the previous images, only the moving parts in the video get highlighted. This gives us a good starting point to see what areas are moving in the video. Here is the code to do this:

```
import cv2

# Compute the frame difference
def frame_diff(prev_frame, cur_frame, next_frame):
    # Absolute difference between current frame and next frame
    diff_frames1 = cv2.absdiff(next_frame, cur_frame)

    # Absolute difference between current frame and
    # previous frame
    diff_frames2 = cv2.absdiff(cur_frame, prev_frame)

    # Return the result of bitwise 'AND' between the
    # above two resultant images
    return cv2.bitwise_and(diff_frames1, diff_frames2)

# Capture the frame from webcam
def get_frame(cap):
    # Capture the frame
    ret, frame = cap.read()

    # Resize the image
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    # Return the grayscale image
    return cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

if __name__=='__main__':
    cap = cv2.VideoCapture(0)
    scaling_factor = 0.5

    prev_frame = get_frame(cap)
    cur_frame = get_frame(cap)
    next_frame = get_frame(cap)

    # Iterate until the user presses the ESC key
    while True:
        # Display the result of frame differencing
        cv2.imshow("Object Movement", frame_diff(prev_frame,
        cur_frame, next_frame))
        # Update the variables
        prev_frame = cur_frame
        cur_frame = next_frame
        next_frame = get_frame(cap)

        # Check if the user pressed ESC
        key = cv2.waitKey(10)
        if key == 27:
            break

    cv2.destroyAllWindows()
```

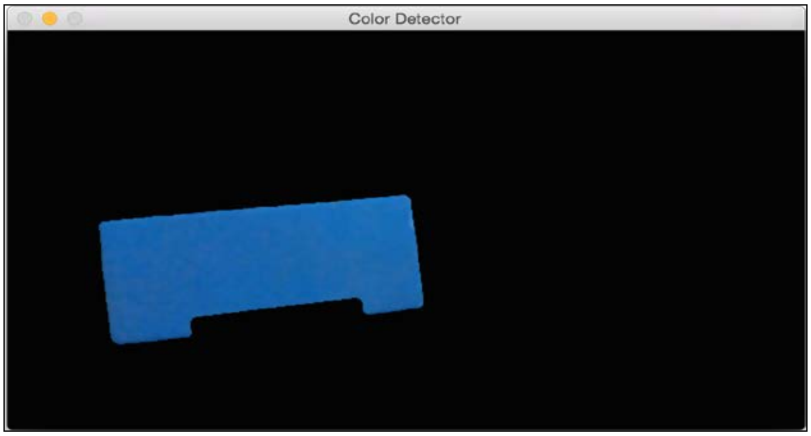
Colorspace based tracking

Frame differencing gives us some useful information, but we cannot use it to build anything meaningful. In order to build a good object tracker, we need to understand what characteristics can be used to make our tracking robust and accurate. So, let's take a step in that direction and see how we can use **colorspaces** to come up with a good tracker. As we have discussed in previous chapters, HSVcolorspace is very informative when it comes to human perception. We can convert an image to the HSV space, and then use `colorspacethresholding` to track a given object.

Consider the following frame in the video:



If you run it through the colorspace filter and track the object, you will see something like this:



As we can see here, our tracker recognizes a particular object in the video, based on the color characteristics. In order to use this tracker, we need to know the color distribution of our target object. Following is the code:

```
import cv2
import numpy as np

# Capture the input frame from webcam
def get_frame(cap, scaling_factor):
    # Capture the frame from video capture object
    ret, frame = cap.read()

    # Resize the input frame
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    return frame

if __name__=='__main__':
    cap = cv2.VideoCapture(0)
    scaling_factor = 0.5

    # Iterate until the user presses ESC key
    while True:
        frame = get_frame(cap, scaling_factor)
```

```
            # Convert the HSV colorspace
            hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

            # Define 'blue' range in HSV colorspace
            lower = np.array([60,100,100])
            upper = np.array([180,255,255])

            # Threshold the HSV image to get only blue color
            mask = cv2.inRange(hsv, lower, upper)

            # Bitwise-AND mask and original image
            res = cv2.bitwise_and(frame, frame, mask=mask)
            res = cv2.medianBlur(res, 5)

            cv2.imshow('Original image', frame)
            cv2.imshow('Color Detector', res)

            # Check if the user pressed ESC key
            c = cv2.waitKey(5)
            if c == 27:
                break

        cv2.destroyAllWindows()
```

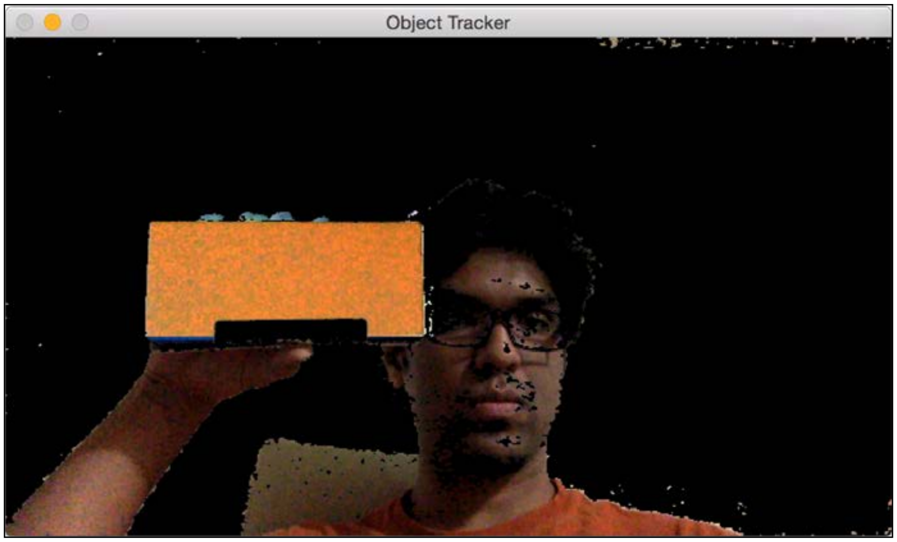
Building an interactive object tracker

Colorspace based tracker gives us the freedom to track a colored object, but we are also constrained to a predefined color. What if we just want to pick an object at random? How do we build an object tracker that can learn the characteristics of the selected object and just track it automatically? This is where the **CAMShift** algorithm, which stands for Continuously Adaptive Meanshift, comes into the picture. It's basically an improved version of the **Meanshift** algorithm.

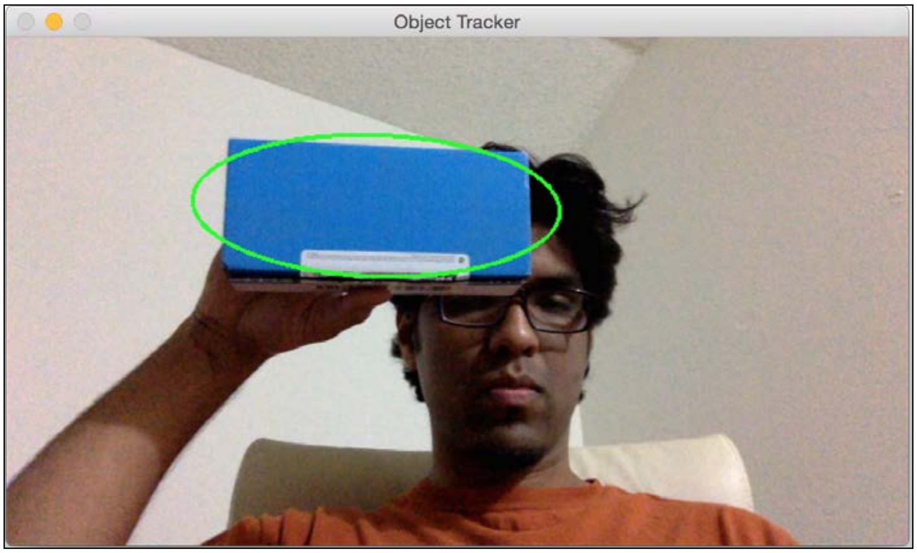
The concept of Meanshift is actually nice and simple. Let's say we select a region of interest and we want our object tracker to track that object. In that region, we select a bunch of points based on the color histogram and compute the centroid. If the centroid lies at the center of this region, we know that the object hasn't moved. But if the centroid is not at the center of this region, then we know that the object is moving in some direction. The movement of the centroid controls the direction in which the object is moving. So, we move our bounding box to a new location so that the new centroid becomes the center of this bounding box. Hence, this algorithm is called Meanshift, because the mean (i.e. the centroid) is shifting. This way, we keep ourselves updated with the current location of the object.

But the problem with Meanshift is that the size of the bounding box is not allowed to change. When you move the object away from the camera, the object will appear smaller to the human eye, but Meanshift will not take this into account. The size of the bounding box will remain the same throughout the tracking session. Hence, we need to use CAMShift. The advantage of CAMShift is that it can adapt the size of the bounding box to the size of the object. Along with that, it can also keep track of the orientation of the object.

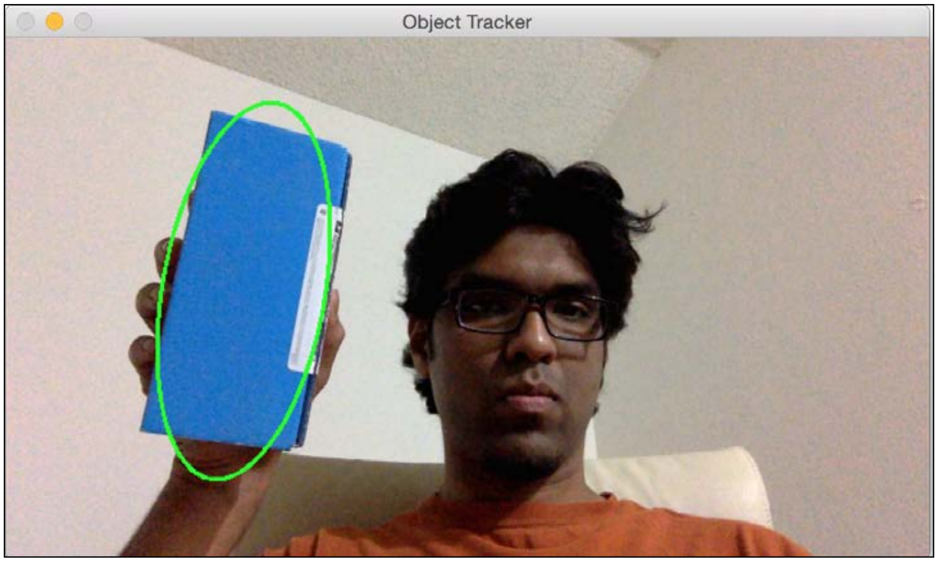
Let's consider the following frame in which the object is highlighted in orange (the box in my hand):



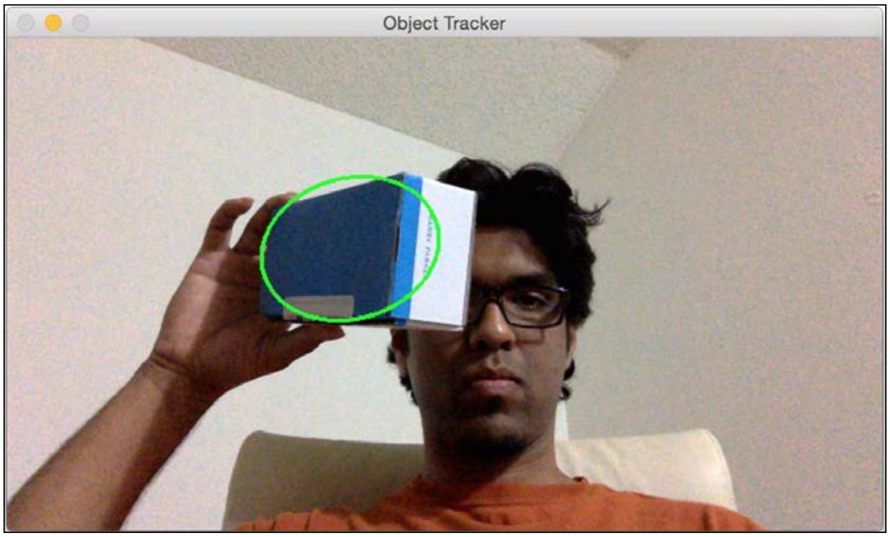
Now that we have selected the object, the algorithm computes the histogram `backprojection` and extracts all the information. Let's move the object and see how it's getting tracked:



Looks like the object is getting tracked fairly well. Let's change the orientation and see if the tracking is maintained:



As we can see, the bounding ellipse has changed its location as well as its orientation. Let's change the perspective of the object and see if it's still able to track it:



We are still good! The bounding ellipse has changed the aspect ratio to reflect the fact that the object looks skewed now (because of the perspective transformation).

Following is the code:

```
import sys

import cv2
import numpy as np

class ObjectTracker(object):
    def __init__(self):
        # Initialize the video capture object
        # 0 -> indicates that frame should be captured
        # from webcam
        self.cap = cv2.VideoCapture(0)

        # Capture the frame from the webcam
        ret, self.frame = self.cap.read()

        # Downsampling factor for the input frame
        self.scaling_factor = 0.5
        self.frame = cv2.resize(self.frame, None,
        fx=self.scaling_factor,
                  fy=self.scaling_factor,
                  interpolation=cv2.INTER_AREA)

        cv2.namedWindow('Object Tracker')
        cv2.setMouseCallback('Object Tracker',
        self.mouse_event)

        self.selection = None
        self.drag_start = None
        self.tracking_state = 0

    # Method to track mouse events
    def mouse_event(self, event, x, y, flags, param):
        x, y = np.int16([x, y])

        # Detecting the mouse button down event
        if event == cv2.EVENT_LBUTTONDOWN:
            self.drag_start = (x, y)
            self.tracking_state = 0

        if self.drag_start:
            if flags & cv2.EVENT_FLAG_LBUTTON:
                h, w = self.frame.shape[:2]
                xo, yo = self.drag_start
                x0, y0 = np.maximum(0, np.minimum([xo, yo],
                [x, y]))
                x1, y1 = np.minimum([w, h],
                np.maximum([xo, yo], [x, y]))
                self.selection = None

                if x1-x0 > 0 and y1-y0 > 0:
                    self.selection = (x0, y0, x1, y1)

            else:
                self.drag_start = None
                if self.selection is not None:
                    self.tracking_state = 1

    # Method to start tracking the object
    def start_tracking(self):
        # Iterate until the user presses the Esc key
        while True:
            # Capture the frame from webcam
            ret, self.frame = self.cap.read()
```

```
# Resize the input frame
self.frame = cv2.resize(self.frame, None,
fx=self.scaling_factor,
            fy=self.scaling_factor,
            interpolation=cv2.INTER_AREA)

vis = self.frame.copy()

# Convert to HSV colorspace
hsv = cv2.cvtColor(self.frame, cv2.COLOR_BGR2HSV)

# Create the mask based on predefined thresholds.
mask = cv2.inRange(hsv, np.array((0., 60., 32.)),
            np.array((180., 255., 255.)))

if self.selection:
    x0, y0, x1, y1 = self.selection
    self.track_window = (x0, y0, x1-x0, y1-y0)
    hsv_roi = hsv[y0:y1, x0:x1]
    mask_roi = mask[y0:y1, x0:x1]

    # Compute the histogram
    hist = cv2.calcHist( [hsv_roi], [0], mask_roi,
    [16], [0, 180] )

    # Normalize and reshape the histogram
    cv2.normalize(hist, hist, 0, 255,
    cv2.NORM_MINMAX);
    self.hist = hist.reshape(-1)

    vis_roi = vis[y0:y1, x0:x1]
    cv2.bitwise_not(vis_roi, vis_roi)
    vis[mask == 0] = 0

if self.tracking_state == 1:
    self.selection = None

    # Compute the histogram back projection
    prob = cv2.calcBackProject([hsv], [0],
    self.hist, [0, 180], 1)

    prob &= mask
    term_crit = ( cv2.TERM_CRITERIA_EPS |
    cv2.TERM_CRITERIA_COUNT, 10, 1 )
```

```
                # Apply CAMShift on 'prob'
                track_box, self.track_window = cv2.CamShift(prob,
                self.track_window, term_crit)

                # Draw an ellipse around the object
                cv2.ellipse(vis, track_box, (0, 255, 0), 2)

            cv2.imshow('Object Tracker', vis)

            c = cv2.waitKey(5)
            if c == 27:
                break

        cv2.destroyAllWindows()

if __name__ == '__main__':
    ObjectTracker().start_tracking()
```

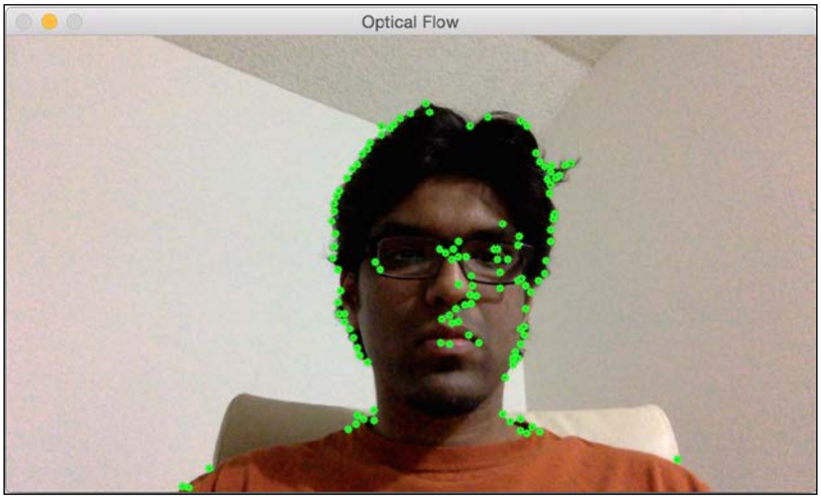
Feature based tracking

Feature based tracking refers to tracking individual feature points across successive frames in the video. We use a technique called **optical flow** to track these features. Optical flow is one of the most popular techniques in computer vision. We choose a bunch of feature points and track them through the video stream.

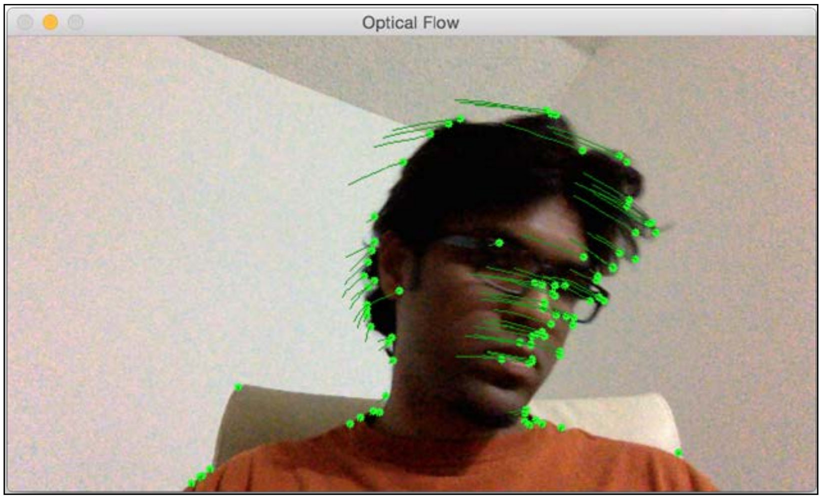
When we detect the feature points, we compute the displacement vectors and show the motion of those keypoints between consecutive frames. These vectors are called motion vectors. There are many ways to do this, but the Lucas-Kanade method is perhaps the most popular of all these techniques. You can refer to their original paper at `http://cseweb.ucsd.edu/classes/sp02/cse252/lucaskanade81.pdf`. We start the process by extracting the feature points. For each feature point, we create 3x3 patches with the feature point in the center. The assumption here is that all the points within each patch will have a similar motion. We can adjust the size of this window depending on the problem at hand.

For each feature point in the current frame, we take the surrounding 3x3 patch as our reference point. For this patch, we look in its neighborhood in the previous frame to get the best match. This neighborhood is usually bigger than 3x3 because we want to get the patch that's closest to the patch under consideration. Now, the path from the center pixel of the matched patch in the previous frame to the center pixel of the patch under consideration in the current frame will become the motion vector. We do that for all the feature points and extract all the motion vectors.

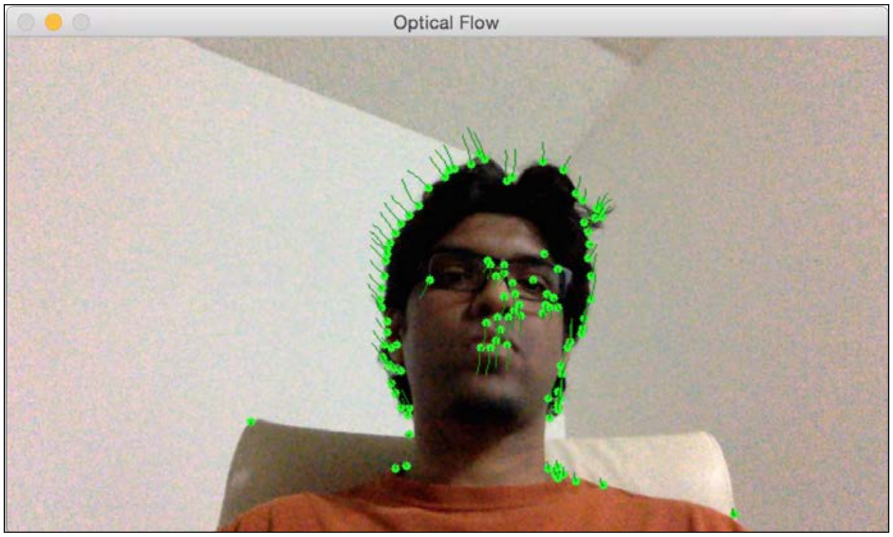
Let's consider the following frame:



If I move in a horizontal direction, you will see the motion vectors in a horizontal direction:



If I move away from the webcam, you will see something like this:



So, if you want to play around with it, you can let the user select a region of interest in the input video (like we did earlier). You can then extract feature points from this region of interest and track the object by drawing the bounding box. It will be a fun exercise!

Here is the code to perform optical flow based tracking:

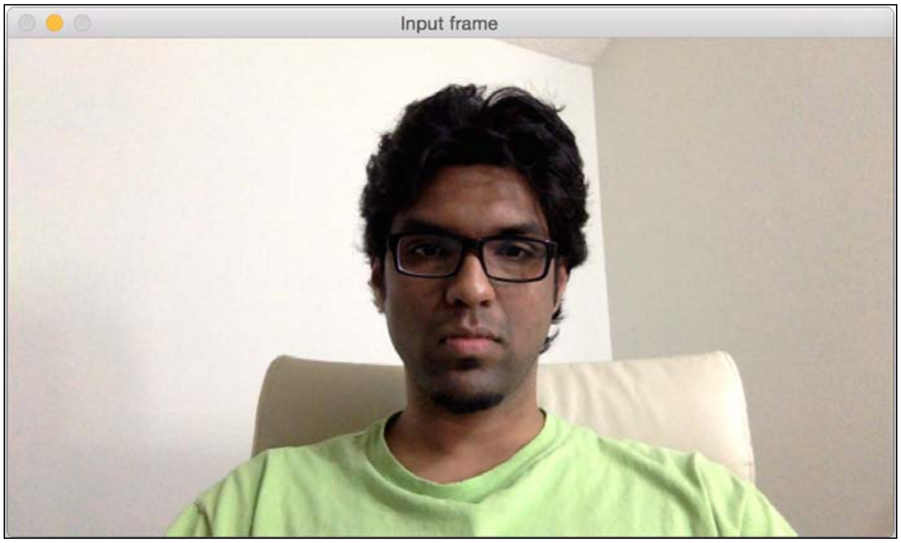
```
import cv2
import numpy as np

def start_tracking():
    # Capture the input frame
    cap = cv2.VideoCapture(0)

    # Downsampling factor for the image
    scaling_factor = 0.5

    # Number of frames to keep in the buffer when you
    # are tracking. If you increase this number,
    # feature points will have more "inertia"
    num_frames_to_track = 5

    # Skip every 'n' frames. This is just to increase the speed.
    num_frames_jump = 2
```

```
tracking_paths = []
frame_index = 0

# 'winSize' refers to the size of each patch. These patches
# are the smallest blocks on which we operate and track
# the feature points. You can read more about the parameters
# here: http://goo.gl/ulwqLk
tracking_params = dict(winSize  = (11, 11), maxLevel = 2,
        criteria = (cv2.TERM_CRITERIA_EPS |
        cv2.TERM_CRITERIA_COUNT, 10, 0.03))

# Iterate until the user presses the ESC key
while True:
    # read the input frame
    ret, frame = cap.read()

    # downsample the input frame
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    output_img = frame.copy()

    if len(tracking_paths) > 0:
        prev_img, current_img = prev_gray, frame_gray
        feature_points_0 = np.float32([tp[-1] for tp in
        tracking_paths]).reshape(-1, 1, 2)

        # Compute feature points using optical flow. You can
        # refer to the documentation to learn more about the
        # parameters here: http://goo.gl/t6P4SE
        feature_points_1, _, _ =
        cv2.calcOpticalFlowPyrLK(prev_img,
        current_img, feature_points_0,
                None, **tracking_params)
        feature_points_0_rev, _, _ =
        cv2.calcOpticalFlowPyrLK(current_img, prev_img,
        feature_points_1,
                None, **tracking_params)

        # Compute the difference of the feature points
        diff_feature_points = abs(feature_points_0-
        feature_points_0_rev).reshape(-1, 2).max(-1)
        # threshold and keep the good points
        good_points = diff_feature_points < 1

        new_tracking_paths = []

        for tp, (x, y), good_points_flag in
        zip(tracking_paths,
                    feature_points_1.reshape(-1, 2),
                    good_points):
            if not good_points_flag:
                continue

            tp.append((x, y))

            # Using the queue structure i.e. first in,
            # first out
            if len(tp) > num_frames_to_track:
                del tp[0]

            new_tracking_paths.append(tp)

            # draw green circles on top of the output image
            cv2.circle(output_img, (x, y), 3, (0, 255, 0), -1)

        tracking_paths = new_tracking_paths

        # draw green lines on top of the output image
        cv2.polylines(output_img, [np.int32(tp) for tp in
        tracking_paths], False, (0, 150, 0))

    # 'if' condition to skip every 'n'th frame
    if not frame_index % num_frames_jump:
        mask = np.zeros_like(frame_gray)
        mask[:] = 255
        for x, y in [np.int32(tp[-1]) for tp in
        tracking_paths]:
            cv2.circle(mask, (x, y), 6, 0, -1)

        # Extract good features to track. You can learn more
        # about the parameters here: http://goo.gl/BI2Kml
        feature_points = cv2.goodFeaturesToTrack(frame_gray,
                mask = mask, maxCorners = 500,
                qualityLevel = 0.3,
                minDistance = 7, blockSize = 7)
                if feature_points is not None:
                    for x, y in np.float32(feature_points).reshape
                    (-1, 2):
                        tracking_paths.append([(x, y)])

            frame_index += 1
            prev_gray = frame_gray

            cv2.imshow('Optical Flow', output_img)

            # Check if the user pressed the ESC key
            c = cv2.waitKey(1)
            if c == 27:
                break

if __name__ == '__main__':
    start_tracking()
    cv2.destroyAllWindows()
```

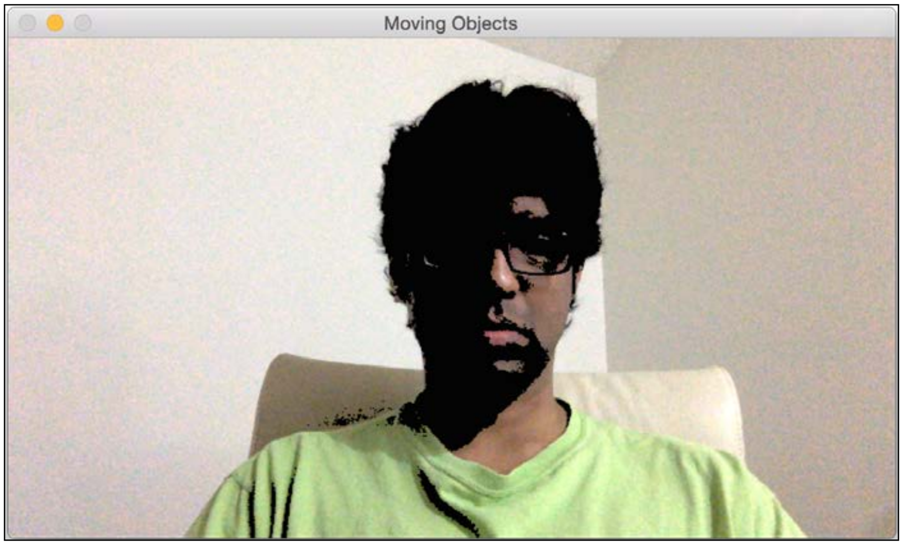
Background subtraction

Background subtraction is very useful in video surveillance. Basically, background subtraction technique performs really well for cases where we have to detect moving objects in a static scene. As the name indicates, this algorithm works by detecting the background and subtracting it from the current frame to obtain the foreground, that is, moving objects. In order to detect moving objects, we need to build a model of the background first. This is not the same as frame differencing because we are actually modeling the background and using this model to detect moving objects. So, this performs much better than the simple frame differencing technique. This technique tries to detect static parts in the scene and then include it in the background model. So, it's an adaptive technique that can adjust according to the scene.

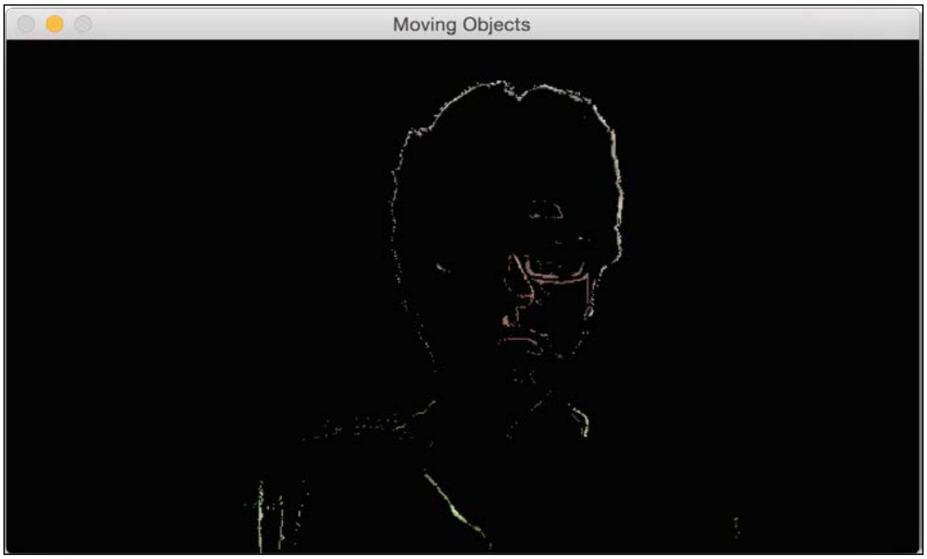
Let's consider the following image:



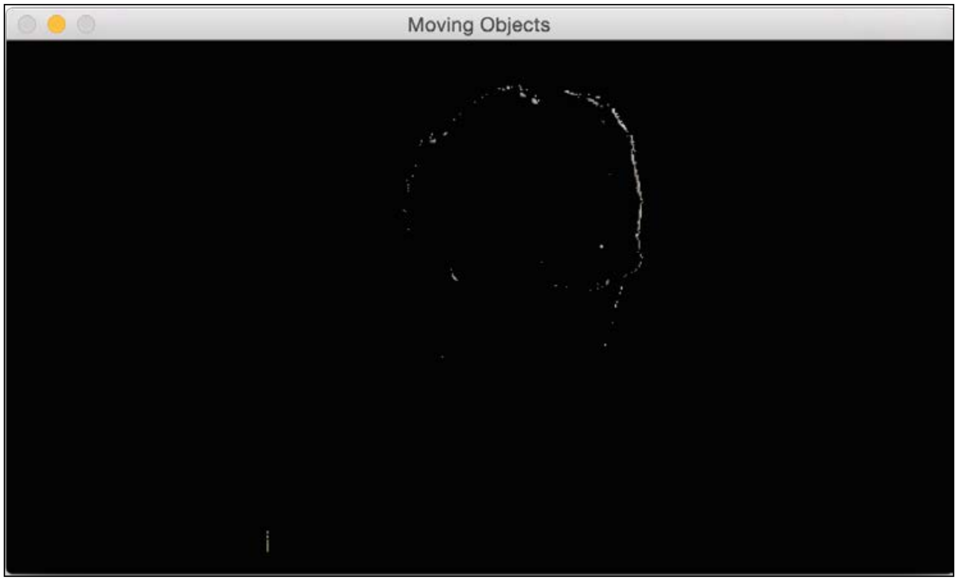
Now, as we gather more frames in this scene, every part of the image will gradually become a part of the background model. This is what we discussed earlier as well. If a scene is static, the model adapts itself to make sure the background model is updated. This is how it looks in the beginning:



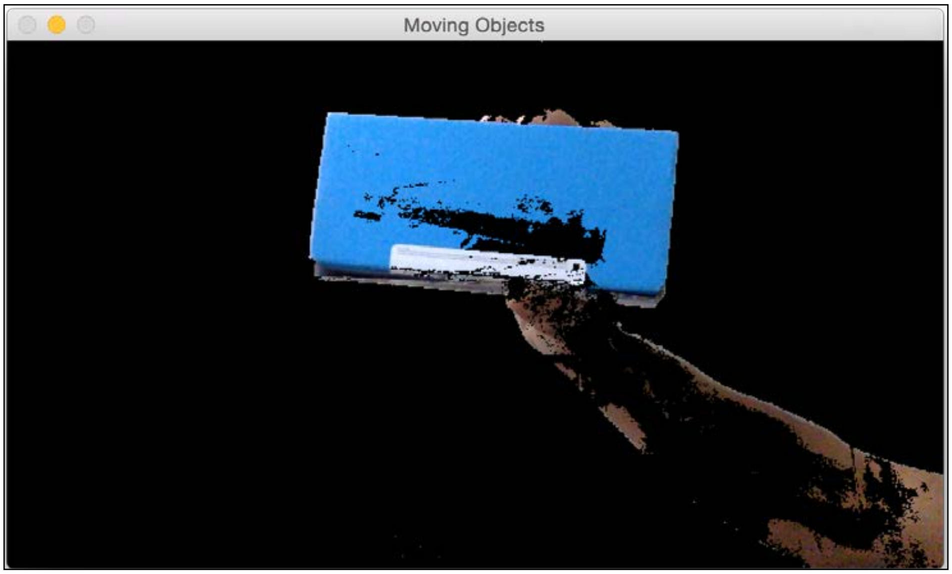
Notice how a part of my face has already become a part of the background model (the blackened region). The following screenshot shows what we'll see after a few seconds:



If we keep going, everything eventually becomes part of the background model:



Now, if we introduce a new moving object, it will be detected clearly, as shown next:



Here is the code to do this:

```
import cv2
import numpy as np

# Capture the input frame
def get_frame(cap, scaling_factor=0.5):
    ret, frame = cap.read()

    # Resize the frame
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    return frame

if __name__=='__main__':
    # Initialize the video capture object
    cap = cv2.VideoCapture(0)

    # Create the background subtractor object
    bgSubtractor = cv2.BackgroundSubtractorMOG()

    # This factor controls the learning rate of the algorithm.
    # The learning rate refers to the rate at which your model
    # will learn about the background. Higher value for
    # 'history' indicates a slower learning rate. You
    # can play with this parameter to see how it affects
    # the output.
    history = 100

    # Iterate until the user presses the ESC key
    while True:
        frame = get_frame(cap, 0.5)

        # Apply the background subtraction model to the
        # input frame
        mask = bgSubtractor.apply(frame,
        learningRate=1.0/history)

        # Convert from grayscale to 3-channel RGB
        mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)

        cv2.imshow('Input frame', frame)
        cv2.imshow('Moving Objects', mask & frame)

        # Check if the user pressed the ESC key
        c = cv2.waitKey(10)
        if c == 27:
            break

    cap.release()
    cv2.destroyAllWindows()
```

Summary

In this chapter, we learned about object tracking. We learned how to get motion information using frame differencing, and how it can be limiting when we want to track different types of objects. We learned about colorspacethresholding and how it can be used to track colored objects. We discussed clustering techniques for object tracking and how we can build an interactive object tracker using the CAMShift algorithm. We discussed how to track features in a video and how we can use optical flow to achieve the same. We learned about background subtraction and how it can be used for video surveillance.

In the next chapter, we are going to discuss object recognition, and how we can build a visual search engine.

9
Object Recognition

In this chapter, we are going to learn about object recognition and how we can use it to build a visual search engine. We will discuss feature detection, building feature vectors, and using machine learning to build a classifier. We will learn how to use these different blocks to build an object recognition system.

By the end of this chapter, you will know:

- What is the difference between object detection and object recognition
- What is a dense feature detector
- What is a visual dictionary
- How to build a feature vector
- What is supervised and unsupervised learning
- What are Support Vector Machines and how to use them to build a classifier
- How to recognize an object in an unknown image

Object detection versus object recognition

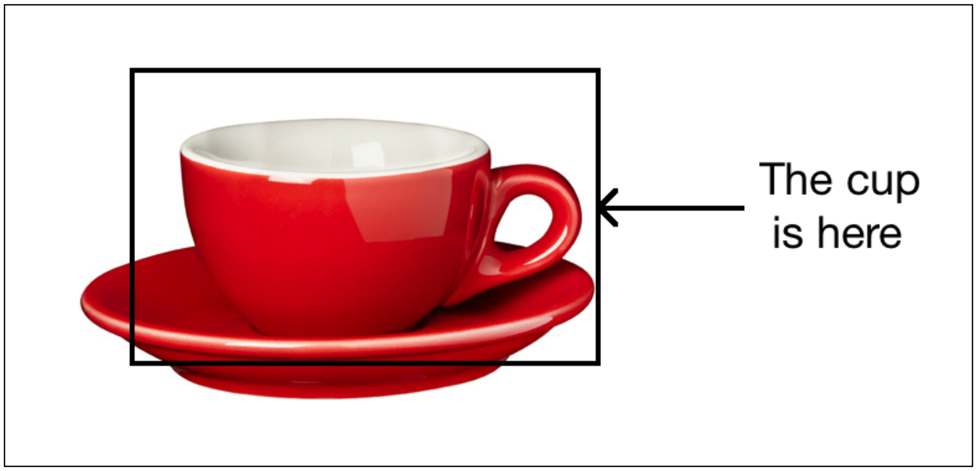
Before we proceed, we need to understand what we are going to discuss in this chapter. You must have frequently heard the terms "object detection" and "object recognition", and they are often mistaken to be the same thing. There is a very distinct difference between the two.

Object detection refers to detecting the presence of a particular object in a given scene. We don't know what the object might be. For instance, we discussed face detection in *Chapter 3, Detecting and Tracking Different Body Parts*. During the discussion, we only detected whether or not a face is present in the given image. We didn't recognize the person! The reason we didn't recognize the person is because we didn't care about that in our discussion. Our goal was to find the location of the face in the given image. Commercial face recognition systems employ both face detection and face recognition to identify a person. First, we need to locate the face, and then, run the face recognizer on the cropped face.

Object recognition is the process of identifying an object in a given image. For instance, an object recognition system can tell you if a given image contains a dress or a pair of shoes. In fact, we can train an object recognition system to identify many different objects. The problem is that object recognition is a really difficult problem to solve. It has eluded computer vision researchers for decades now, and has become the holy grail of computer vision. Humans can identify a wide variety of objects very easily. We do it everyday and we do it effortlessly, but computers are unable to do it with that kind of accuracy.

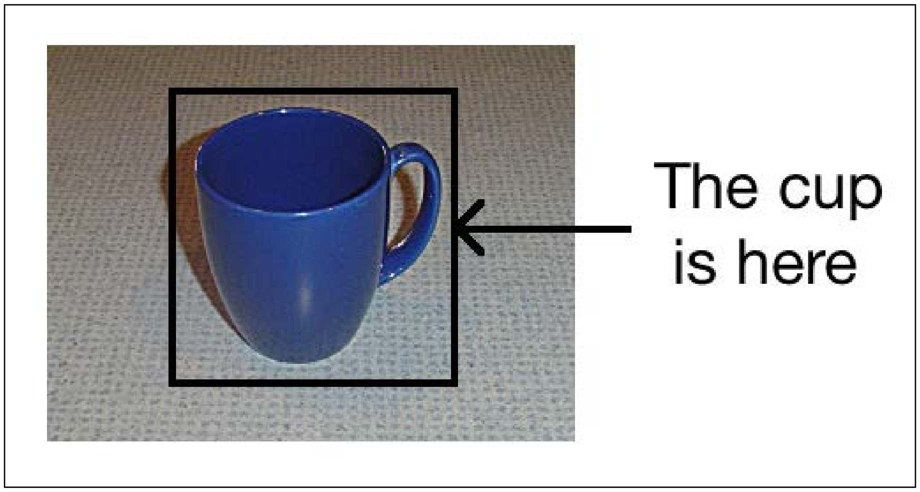
Let's consider the following image of a latte cup:

An object detector will give you the following information:

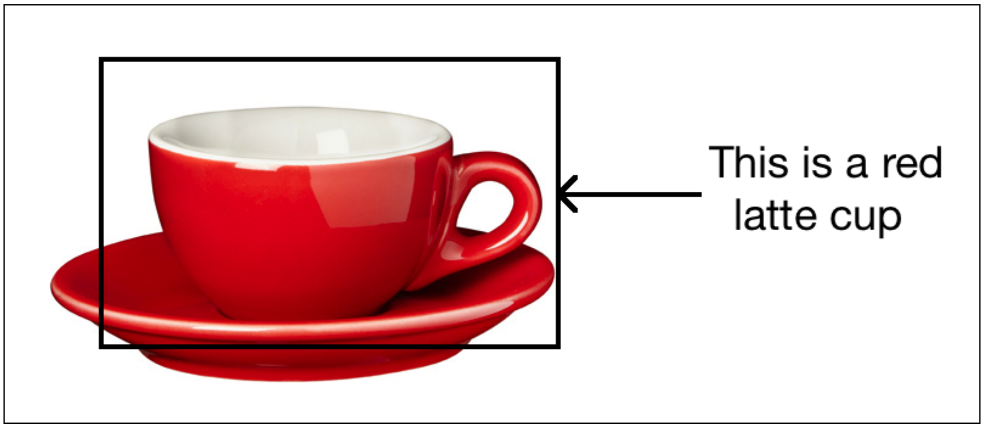


Now, consider the following image of a teacup:

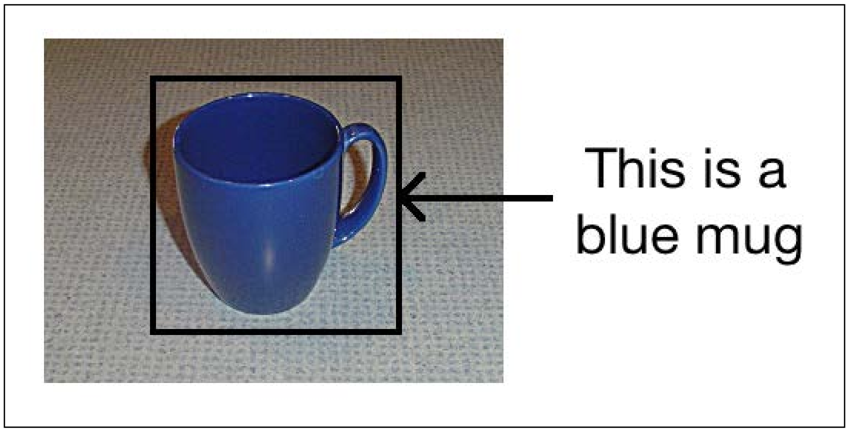
If you run it through an object detector, you will see the following result:



As you can see, the object detector detects the presence of the teacup, but nothing more than that. If you train an object recognizer, it will give you the following information, as shown in the image below:



If you consider the second image, it will give you the following information:

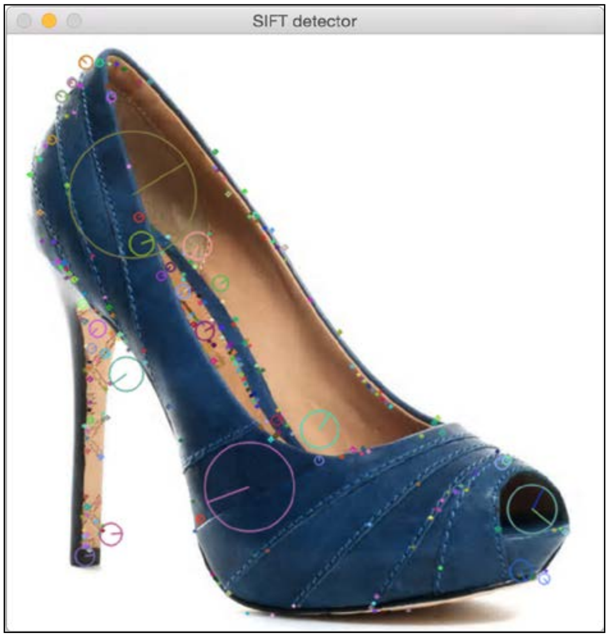


As you can see, a perfect object recognizer would give you all the information associated with that object. An object recognizer functions more accurately if it knows where the object is located. If you have a big image and the cup is a small part of it, then the object recognizer might not be able to recognize it. Hence, the first step is to detect the object and get the bounding box. Once we have that, we can run an object recognizer to extract more information.

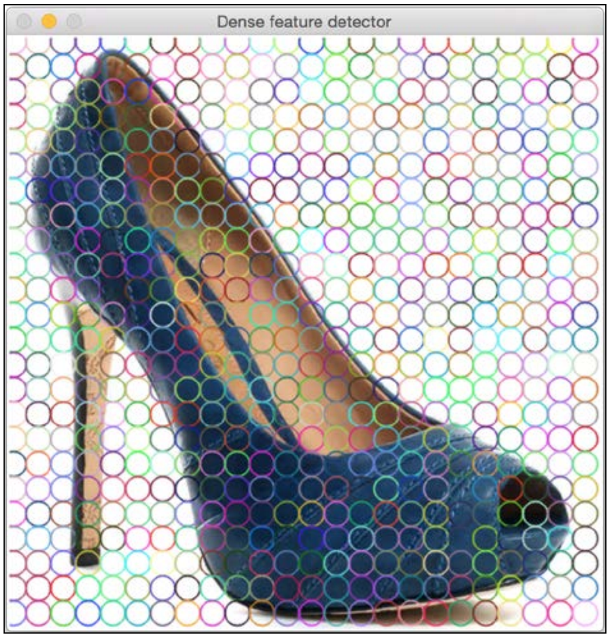
What is a dense feature detector?

In order to extract a meaningful amount of information from the images, we need to make sure our feature extractor extracts features from all the parts of a given image. Consider the following image:

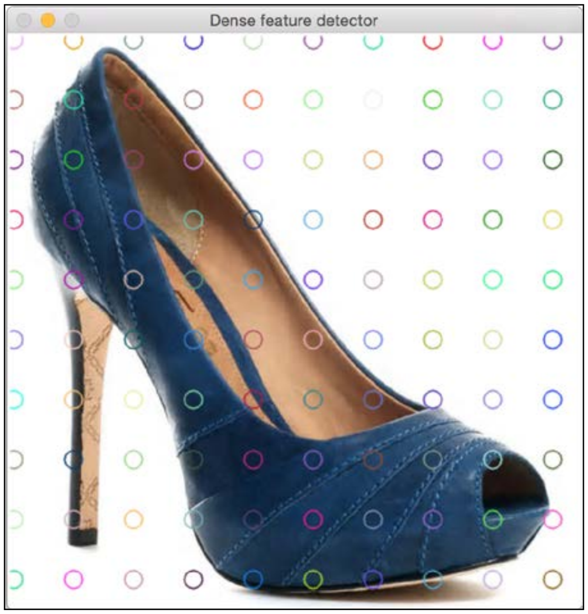
If you extract features using a feature extractor, it will look like this:



If you use `Dense` detector, it will look like this:



We can control the density as well. Let's make it sparse:



By doing this, we can make sure that every single part in the image is processed. Here is the code to do it:

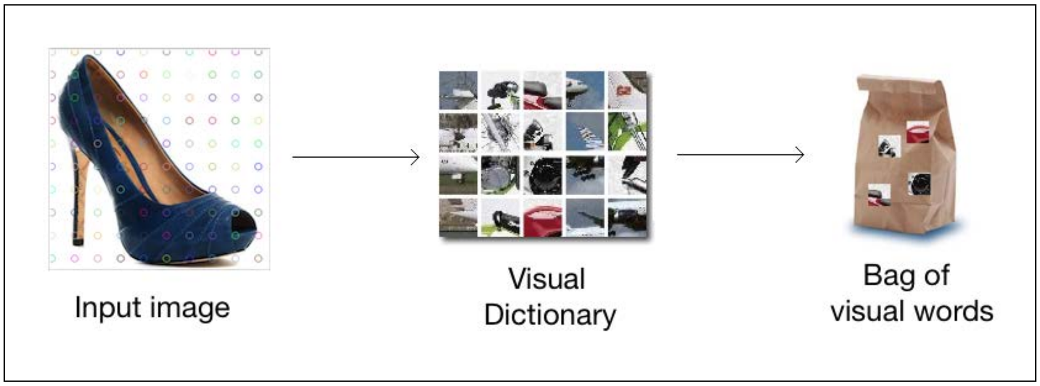
```
import cv2
import numpy as np

class DenseDetector(object):
    def __init__(self, step_size=20, feature_scale=40,
    img_bound=20):
        # Create a dense feature detector
        self.detector = cv2.FeatureDetector_create("Dense")

        # Initialize it with all the required parameters
        self.detector.setInt("initXyStep", step_size)
        self.detector.setInt("initFeatureScale", feature_scale)
        self.detector.setInt("initImgBound", img_bound)

    def detect(self, img):
        # Run feature detector on the input image
        return self.detector.detect(img)
```

```
if __name__=='__main__':
    input_image = cv2.imread(sys.argv[1])
    input_image_sift = np.copy(input_image)

    # Convert to grayscale
    gray_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)

    keypoints = DenseDetector(20,20,5).detect(input_image)

    # Draw keypoints on top of the input image
    input_image = cv2.drawKeypoints(input_image, keypoints,
            flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    # Display the output image
    cv2.imshow('Dense feature detector', input_image)

    # Initialize SIFT object
    sift = cv2.SIFT()

    # Detect keypoints using SIFT
    keypoints = sift.detect(gray_image, None)

    # Draw SIFT keypoints on the input image
    input_image_sift = cv2.drawKeypoints(input_image_sift,
            keypoints,
            flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    # Display the output image
    cv2.imshow('SIFT detector', input_image_sift)

    # Wait until user presses a key
    cv2.waitKey()
```

This gives us close control over the amount of information that gets extracted. When we use a SIFT detector, some parts of the image are neglected. This works well when we are dealing with the detection of prominent features, but when we are building an object recognizer, we need to evaluate all parts of the image. Hence, we use a dense detector and then extract features from those keypoints.

What is a visual dictionary?

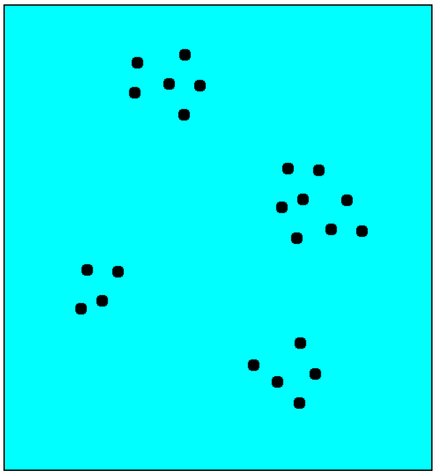
We will be using the **Bag of Words** model to build our object recognizer. Each image is represented as a histogram of visual words. These visual words are basically the **N** centroids built using all the keypoints extracted from training images. The pipeline is as shown in the image that follows:



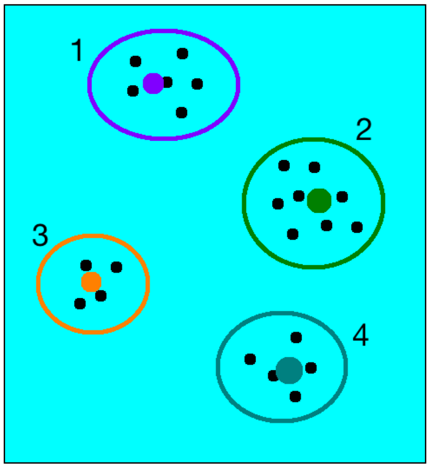
From each training image, we detect a set of keypoints and extract features for each of those keypoints. Every image will give rise to a different number of keypoints. In order to train a classifier, each image must be represented using a fixed length feature vector. This feature vector is nothing but a histogram, where each bin corresponds to a visual word.

When we extract all the features from all the keypoints in the training images, we perform K-Means clustering and extract N centroids. This N is the length of the feature vector of a given image. Each image will now be represented as a histogram, where each bin corresponds to one of the 'N' centroids. For simplicity, let's say that N is set to 4. Now, in a given image, we extract **K** keypoints. Out of these K keypoints, some of them will be closest to the first centroid, some of them will be closest to the second centroid, and so on. So, we build a histogram based on the closest centroid to each keypoint. This histogram becomes our feature vector. This process is called **vector quantization**.

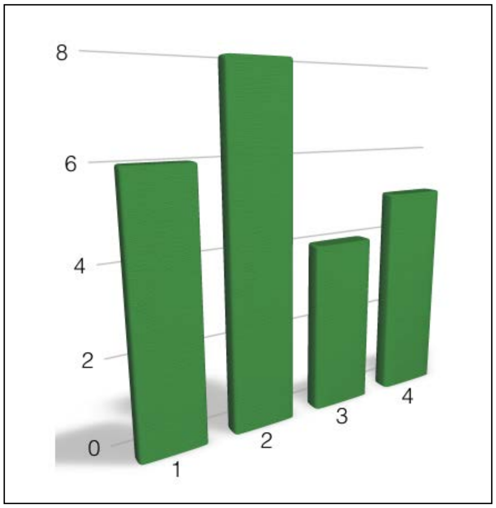
To understand vector quantization, let's consider an example. Assume we have an image and we've extracted a certain number of feature points from it. Now our goal is to represent this image in the form of a feature vector. Consider the following image:



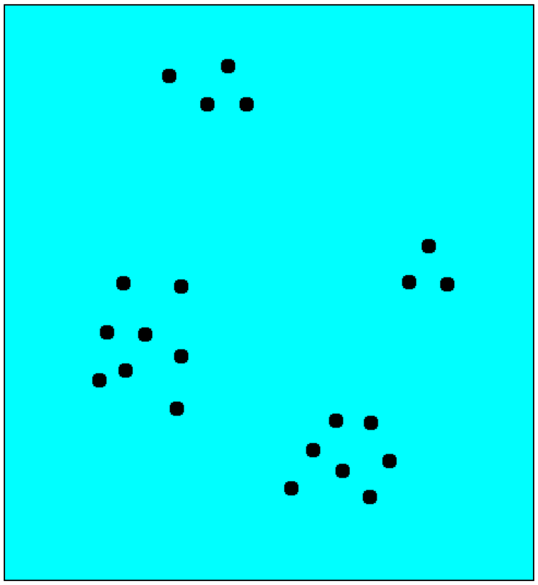
As you can see, we have 4 centroids. Bear in mind that the points shown in the figures represent the feature space and not the actual geometric locations of those feature points in the image. It is shown this way in the preceding figure so that it's easy to visualize. Points from many different geometric locations in an image can be close to each other in the feature space. Our goal is to represent this image as a histogram, where each bin corresponds to one of these centroids. This way, no matter how many feature points we extract from an image, it will always be converted to a fixed length feature vector. So, we "round off" each feature point to its nearest centroid, as shown in the next image:



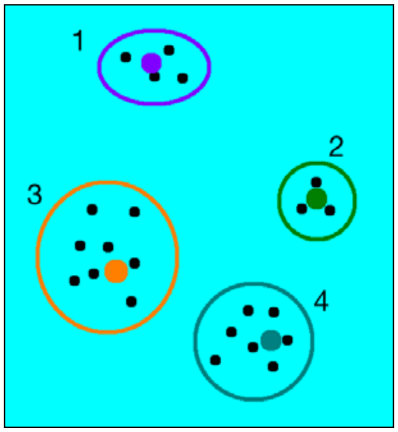
If you build a histogram for this image, it will look like this:



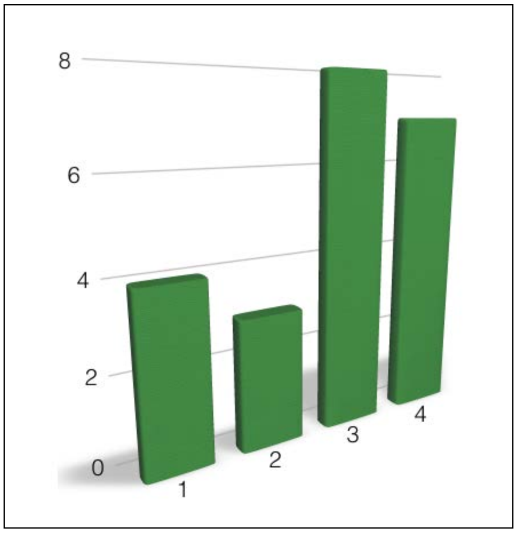
Now, if you consider a different image with a different distribution of feature points, it will look like this:



The clusters would look like the following:



The histogram would look like this:



As you can see, the histograms are very different for the two images even though the points seem to be randomly distributed. This is a very powerful technique and it's widely used in computer vision and signal processing. There are many different ways to do this and the accuracy depends on how fine-grained you want it to be. If you increase the number of centroids, you will be able to represent the image better, thereby increasing the uniqueness of your feature vector. Having said that, it's important to mention that you cannot just keep increasing the number of centroids indefinitely. If you do that, it will become too noisy and lose its power.

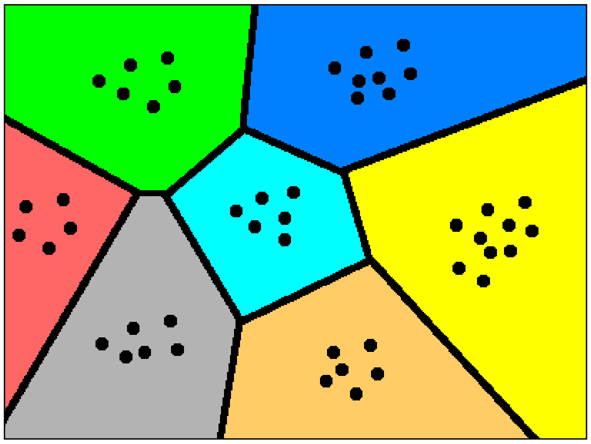
What is supervised and unsupervised learning?

If you are familiar with the basics of machine learning, you will certainly know what supervised and unsupervised learning is all about. To give a quick refresher, supervised learning refers to building a function based on labeled samples. For example, if we are building a system to separate dress images from footwear images, we first need to build a database and label it. We need to tell our algorithm what images correspond to dresses and what images correspond to footwear. Based on this data, the algorithm will learn how to identify dresses and footwear so that when an unknown image comes in, it can recognize what's inside that image.

Unsupervised learning is the opposite of what we just discussed. There is no labeled data available here. Let's say we have a bunch of images, and we just want to separate them into three groups. We don't know what the criteria will be. So, an unsupervised learning algorithm will try to separate the given set of data into 3 groups in the best possible way. The reason we are discussing this is because we will be using a combination of supervised and unsupervised learning to build our object recognition system.

What are Support Vector Machines?

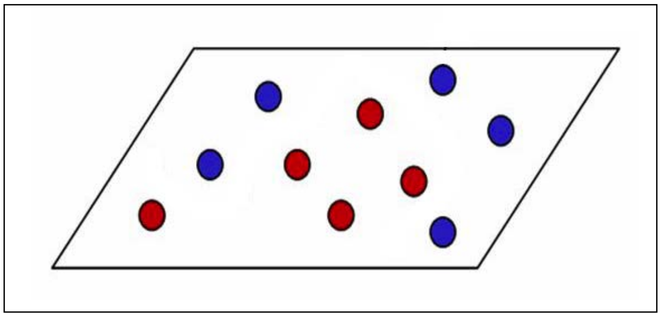
Support Vector Machines (**SVM**) are supervised learning models that are very popular in the realm of machine learning. SVMs are really good at analyzing labeled data and detecting patterns. Given a bunch of data points and the associated labels, SVMs will build the separating hyperplanes in the best possible way.
Wait a minute, what are "hyperplanes"? To understand that, let's consider the following figure:



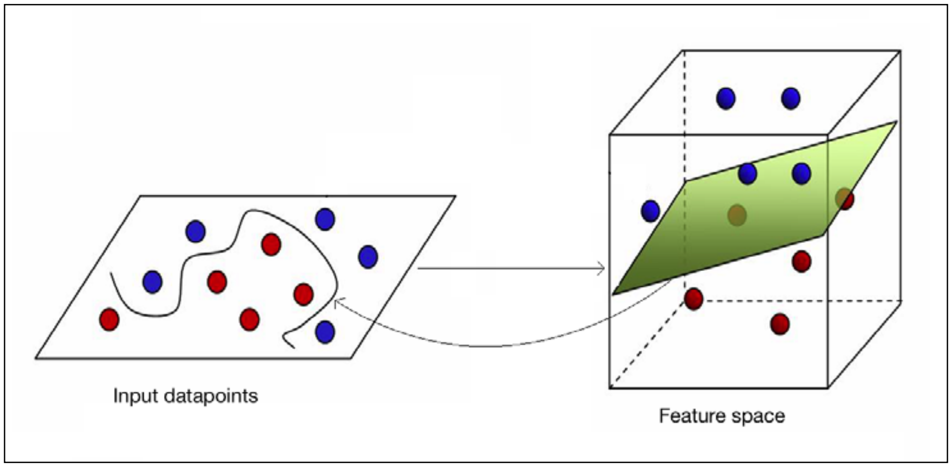
As you can see, the points are being separated by line boundaries that are equidistant from the points. This is easy to visualize in 2 dimensions. If it were in 3 dimensions, the separators would be planes. When we build features for images, the length of the feature vectors is usually in the six-digit range. So, when we go to such a high dimensional space, the equivalent of "lines" would be hyperplanes. Once the hyperplanes are formulated, we use this mathematical model to classify unknown data, based on where it falls on this map.

What if we cannot separate the data with simple straight lines?

There is something called the **kernel trick** that we use in SVMs. Consider the following image:



As we can see, we cannot draw a simple straight line to separate the red points from the blue points. Coming up with a nice curvy boundary that will satisfy all the points is prohibitively expensive. SVMs are really good at drawing "straight lines". So, what's our answer here? The good thing about SVMs is that they can draw these "straight lines" in any number of dimensions. So technically, if you project these points into a high dimensional space, where they can separated by a simple hyperplane, SVMs will come up with an exact boundary. Once we have that boundary, we can project it back to the original space. The projection of this hyperplane on our original lower dimensional space looks curvy, as we can see in the next figure:



The topic of SVMs is really deep and we will not be able to discuss it in detail here. If you are really interested, there is a ton of material available online. You can go through a simple tutorial to understand it better.

How do we actually implement this?

We have now arrived at the core. The discussion up until now was necessary because it gives you the background required to build an object recognition system. Now, let's build an object recognizer that can recognize whether the given image contains a dress, a pair of shoes, or a bag. We can easily extend this system to detect any number of items. We are starting with three distinct items so that you can start experimenting with it later.

Before we start, we need to make sure that we have a set of training images. There are many databases available online where the images are already arranged into groups. Caltech256 is perhaps one of the most popular databases for object recognition. You can download it from `http://www.vision.caltech.edu/Image_Datasets/Caltech256`. Create a folder called `images` and create three subfolders inside it, that is, `dress`, `footwear`, and `bag`. Inside each of those subfolders, add 20 images corresponding to that item. You can just download these images from the internet, but make sure those images have a clean background.

For example, a dress image would like this:

A footwear image would look like this:

A bag image would look like this:

Now that we have 60 training images, we are ready to start. As a side note, object recognition systems actually need tens of thousands of training images in order to perform well in the real world. Since we are building an object recognizer to detect 3 types of objects, we will take only 20 training images per object. Adding more training images will increase the accuracy and robustness of our system.

The first step here is to extract feature vectors from all the training images and build the visual dictionary (also known as codebook). Here is the code:

```
import os
import sys
import argparse
import cPickle as pickle
import json

import cv2
import numpy as np
from sklearn.cluster import KMeans
```

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Creates features
    for given images')
    parser.add_argument("--samples", dest="cls", nargs="+",
    action="append",
            required=True, help="Folders containing the training
            images. \
            The first element needs to be the class label.")
    parser.add_argument("--codebook-file", dest='codebook_file',
    required=True,
            help="Base file name to store the codebook")
    parser.add_argument("--feature-map-file",
    dest='feature_map_file', required=True,
            help="Base file name to store the feature map")

    return parser

# Loading the images from the input folder
def load_input_map(label, input_folder):
    combined_data = []

    if not os.path.isdir(input_folder):
        raise IOError("The folder " + input_folder + " doesn't
        exist")

    # Parse the input folder and assign the  labels
    for root, dirs, files in os.walk(input_folder):
        for filename in (x for x in files if x.endswith('.jpg')):
            combined_data.append({'label': label, 'image':
            os.path.join(root, filename)})

    return combined_data

class FeatureExtractor(object):
    def extract_image_features(self, img):
        # Dense feature detector
        kps = DenseDetector().detect(img)

        # SIFT feature extractor
        kps, fvs = SIFTExtractor().compute(img, kps)

        return fvs
    # Extract the centroids from the feature points
    def get_centroids(self, input_map, num_samples_to_fit=10):
        kps_all = []

        count = 0
        cur_label = ''
        for item in input_map:
            if count >= num_samples_to_fit:
                if cur_label != item['label']:
                    count = 0
                else:
                    continue

            count += 1

            if count == num_samples_to_fit:
                print "Built centroids for", item['label']

            cur_label = item['label']
            img = cv2.imread(item['image'])
            img = resize_to_size(img, 150)

            num_dims = 128
            fvs = self.extract_image_features(img)
            kps_all.extend(fvs)

        kmeans, centroids = Quantizer().quantize(kps_all)
        return kmeans, centroids

    def get_feature_vector(self, img, kmeans, centroids):
        return Quantizer().get_feature_vector(img, kmeans,
        centroids)

def extract_feature_map(input_map, kmeans, centroids):
    feature_map = []

    for item in input_map:
        temp_dict = {}
        temp_dict['label'] = item['label']

        print "Extracting features for", item['image']
        img = cv2.imread(item['image'])
        img = resize_to_size(img, 150)
```

```
            temp_dict['feature_vector'] =
            FeatureExtractor().get_feature_vector(
                        img, kmeans, centroids)

            if temp_dict['feature_vector'] is not None:
                feature_map.append(temp_dict)

        return feature_map

    # Vector quantization
    class Quantizer(object):
        def __init__(self, num_clusters=32):
            self.num_dims = 128
            self.extractor = SIFTExtractor()
            self.num_clusters = num_clusters
            self.num_retries = 10

        def quantize(self, datapoints):
            # Create KMeans object
            kmeans = KMeans(self.num_clusters,
                            n_init=max(self.num_retries, 1),
                            max_iter=10, tol=1.0)

            # Run KMeans on the datapoints
            res = kmeans.fit(datapoints)

            # Extract the centroids of those clusters
            centroids = res.cluster_centers_

            return kmeans, centroids

        def normalize(self, input_data):
            sum_input = np.sum(input_data)
            if sum_input > 0:
                return input_data / sum_input
            else:
                return input_data

        # Extract feature vector from the image
        def get_feature_vector(self, img, kmeans, centroids):
            kps = DenseDetector().detect(img)
            kps, fvs = self.extractor.compute(img, kps)
            labels = kmeans.predict(fvs)
            fv = np.zeros(self.num_clusters)

            for i, item in enumerate(fvs):
                fv[labels[i]] += 1

            fv_image = np.reshape(fv, ((1, fv.shape[0])))
            return self.normalize(fv_image)

    class DenseDetector(object):
        def __init__(self, step_size=20, feature_scale=40,
        img_bound=20):
            self.detector = cv2.FeatureDetector_create("Dense")
            self.detector.setInt("initXyStep", step_size)
            self.detector.setInt("initFeatureScale", feature_scale)
            self.detector.setInt("initImgBound", img_bound)

        def detect(self, img):
            return self.detector.detect(img)

    class SIFTExtractor(object):
        def compute(self, image, kps):
            if image is None:
                print "Not a valid image"
                raise TypeError

            gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            kps, des = cv2.SIFT().compute(gray_image, kps)
            return kps, des

    # Resize the shorter dimension to 'new_size'
    # while maintaining the aspect ratio
    def resize_to_size(input_image, new_size=150):
        h, w = input_image.shape[0], input_image.shape[1]
        ds_factor = new_size / float(h)

        if w < h:
            ds_factor = new_size / float(w)

        new_size = (int(w * ds_factor), int(h * ds_factor))
        return cv2.resize(input_image, new_size)

    if __name__=='__main__':
        args = build_arg_parser().parse_args()
```

```
        input_map = []
        for cls in args.cls:

            assert len(cls) >= 2, "Format for classes is `<label>
            file`"
            label = cls[0]
            input_map += load_input_map(label, cls[1])

        # Building the codebook
        print "===== Building codebook ====="
        kmeans, centroids = FeatureExtractor().get_centroids(input_map)
        if args.codebook_file:
            with open(args.codebook_file, 'w') as f:
                pickle.dump((kmeans, centroids), f)

        # Input data and labels
        print "===== Building feature map ====="
        feature_map = extract_feature_map(input_map, kmeans,
        centroids)
        if args.feature_map_file:
            with open(args.feature_map_file, 'w') as f:
                pickle.dump(feature_map, f)
```

What happened inside the code?

The first thing we need to do is extract the centroids. This is how we are going to build our visual dictionary. The `get_centroids` method in the `FeatureExtractor` class is designed to do this. We keep collecting the image features extracted from keypoints until we have a sufficient number of them. Since we are using a dense detector, 10 images should be sufficient. The reason we are just taking 10 images is because they will give rise to a large number of features. The centroids will not change much even if you add more feature points.

Once we've extracted the centroids, we are ready to move on to the next step of feature extraction. The set of centroids is our visual dictionary. The function, `extract_feature_map`, will extract a feature vector from each image and associate it with the corresponding label. The reason we do this is because we need this mapping to train our classifier. We need a set of datapoints, and each datapoint should be associated with a label. So, we start from an image, extract the feature vector, and then associate it with the corresponding label (like bag, dress, or footwear).

The `Quantizer` class is designed to achieve vector quantization and build the feature vector. For each keypoint extracted from the image, the `get_feature_vector` method finds the closest visual word in our dictionary. By doing this, we end up building a histogram based on our visual dictionary. Each image is now represented as a combination from a set of visual words. Hence the name, **Bag of Words**.

The next step is to train the classifier using these features. Here is the code:

```
import os
import sys
import argparse

import cPickle as pickle
import numpy as np
from sklearn.multiclass import OneVsOneClassifier
from sklearn.svm import LinearSVC
from sklearn import preprocessing

def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the
    classifier models')
    parser.add_argument("--feature-map-file",
    dest="feature_map_file", required=True,
            help="Input pickle file containing the feature map")
    parser.add_argument("--svm-file", dest="svm_file",
    required=False,
            help="Output file where the pickled SVM model will be
            stored")
    return parser

# To train the classifier
class ClassifierTrainer(object):
    def __init__(self, X, label_words):
        # Encoding the labels (words to numbers)
        self.le = preprocessing.LabelEncoder()

        # Initialize One vs One Classifier using a linear kernel
        self.clf = OneVsOneClassifier(LinearSVC(random_state=0))

        y = self._encodeLabels(label_words)
        X = np.asarray(X)
        self.clf.fit(X, y)
    # Predict the output class for the input datapoint
    def _fit(self, X):
        X = np.asarray(X)
        return self.clf.predict(X)

    # Encode the labels (convert words to numbers)
    def _encodeLabels(self, labels_words):
        self.le.fit(labels_words)
        return np.array(self.le.transform(labels_words),
        dtype=np.float32)

    # Classify the input datapoint
    def classify(self, X):
        labels_nums = self._fit(X)
        labels_words = self.le.inverse_transform([int(x) for x in
        labels_nums])
        return labels_words

if __name__=='__main__':
    args = build_arg_parser().parse_args()
    feature_map_file = args.feature_map_file
    svm_file = args.svm_file

    # Load the feature map
    with open(feature_map_file, 'r') as f:
        feature_map = pickle.load(f)

    # Extract feature vectors and the labels
    labels_words = [x['label'] for x in feature_map]

    # Here, 0 refers to the first element in the
    # feature_map, and 1 refers to the second
    # element in the shape vector of that element
    # (which gives us the size)
    dim_size = feature_map[0]['feature_vector'].shape[1]

    X = [np.reshape(x['feature_vector'], (dim_size,)) for x in
    feature_map]

    # Train the SVM
    svm = ClassifierTrainer(X, labels_words)
    if args.svm_file:
        with open(args.svm_file, 'w') as f:
            pickle.dump(svm, f)
```

How did we build the trainer?

We use the `scikit-learn` package to build the SVM model. You can install it, as shown next:

```
$ pip install scikit-learn
```

We start with labeled data and feed it to the `OneVsOneClassifier` method. We have a `classify` method that classifies an input image and associates a label with it.

Let's give this a trial run, shall we? Make sure you have a folder called `images`, where you have the training images for the three classes. Create a folder called `models`, where the learning models will be stored. Run the following commands on your terminal to create the features and train the classifier:

```
$ python create_features.py --samples bag images/bag/ --samples dress
images/dress/ --samples footwear images/footwear/ --codebook-file
models/codebook.pkl --feature-map-file models/feature_map.pkl
```

```
$ python training.py --feature-map-file models/feature_map.pkl
--svm-file models/svm.pkl
```

Now that the classifier has been trained, we just need a module to classify the input image and detect the object inside. Here is the code to do it:

```
import os
import sys
import argparse
import cPickle as pickle

import cv2
import numpy as np

import create_features as cf
from training import ClassifierTrainer

def build_arg_parser():
    parser = argparse.ArgumentParser(description='Extracts
    features \
            from each line and classifies the data')
    parser.add_argument("--input-image", dest="input_image",
    required=True,
            help="Input image to be classified")
    parser.add_argument("--svm-file", dest="svm_file",
    required=True,
            help="File containing the trained SVM model")
    parser.add_argument("--codebook-file", dest="codebook_file",
            required=True, help="File containing the codebook")
    return parser

# Classifying an image
class ImageClassifier(object):
    def __init__(self, svm_file, codebook_file):
        # Load the SVM classifier
        with open(svm_file, 'r') as f:
            self.svm = pickle.load(f)

        # Load the codebook
        with open(codebook_file, 'r') as f:
            self.kmeans, self.centroids = pickle.load(f)

    # Method to get the output image tag
    def getImageTag(self, img):
        # Resize the input image
        img = cf.resize_to_size(img)

        # Extract the feature vector
        feature_vector =
        cf.FeatureExtractor().get_feature_vector(img, self.kmeans,
        self.centroids)

        # Classify the feature vector and get the output tag
        image_tag = self.svm.classify(feature_vector)

        return image_tag

if __name__=='__main__':
    args = build_arg_parser().parse_args()
    svm_file = args.svm_file
    codebook_file = args.codebook_file
    input_image = cv2.imread(args.input_image)

    print "Output class:", ImageClassifier(svm_file,
    codebook_file).getImageTag(input_image)
```

We are all set! We just extract the `feature` vector from the input image and use it as the input argument to the classifier. Let's go ahead and see if this works. Download a random footwear image from the internet and make sure it has a clean background. Run the following command by replacing `new_image.jpg` with the right filename:

```
$ python classify_data.py --input-image new_image.jpg --svm-file
models/svm.pkl --codebook-file models/codebook.pkl
```

We can use the same technique to build a visual search engine. A visual search engine looks at the input image and shows a bunch of images that are similar to it. We can reuse the object recognition framework to build this. Extract the feature vector from the input image, and compare it with all the feature vectors in the training dataset. Pick out the top matches and display the results. This is a simple way of doing things!

In the real world, we have to deal with billions of images. So, you cannot afford to search through every single image before you display the output. There are a lot of algorithms that are used to make sure that this is efficient and fast in the real world. Deep Learning is being used extensively in this field and it has shown a lot of promise in recent years. It is a branch of machine learning that focuses on learning optimal representation of data, so that it becomes easier for the machines to *learn* new tasks. You can learn more about it at `http://deeplearning.net`.

Summary

In this chapter, we learned how to build an object recognition system. The differences between object detection and object recognition were discussed in detail. We learned about the dense feature detector, visual dictionary, vector quantization, and how to use these concepts to build a feature vector. The concepts of supervised and unsupervised learning were discussed. We talked about Support Vector Machines and how we can use them to build a classifier. We learned how to recognize an object in an unknown image, and how we can extend that concept to build a visual search engine.

In the next chapter, we are going to discuss stereo imaging and 3D reconstruction. We will talk about how we can build a depth map and extract the 3D information from a given scene.

10

Stereo Vision and 3D Reconstruction

In this chapter, we are going to learn about stereo vision and how we can reconstruct the 3D map of a scene. We will discuss epipolar geometry, depth maps, and 3D reconstruction. We will learn how to extract 3D information from stereo images and build a point cloud.

By the end of this chapter, you will know:

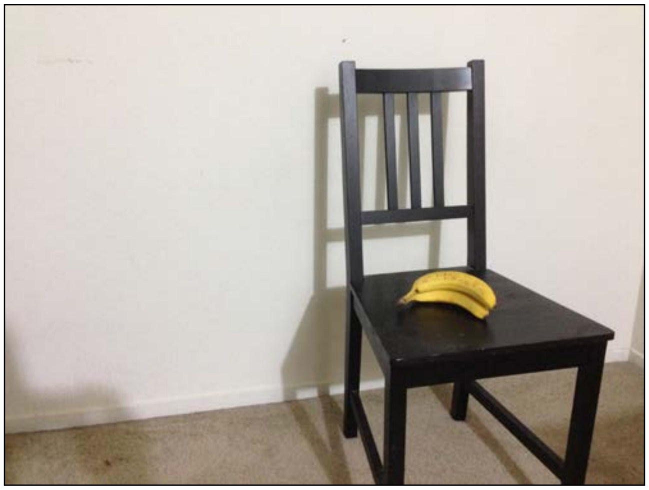
- What is stereo correspondence
- What is epipolar geometry
- What is a depth map
- How to extract 3D information
- How to build and visualize the 3D map of a given scene

What is stereo correspondence?

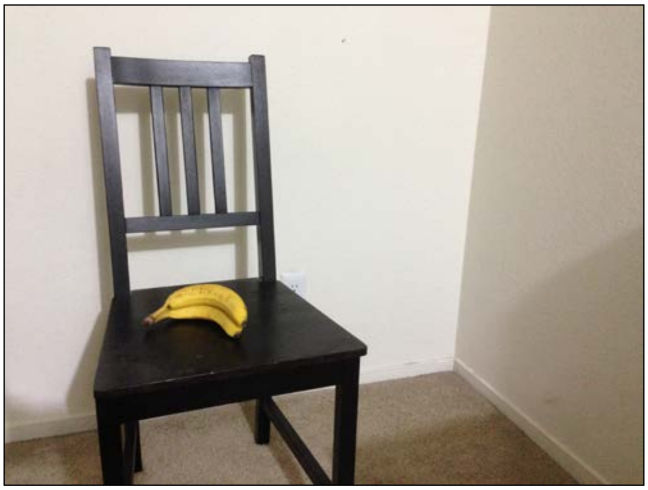
When we capture images, we project the 3D world around us on a 2D image plane. So technically, we only have 2D information when we capture those photos. Since all the objects in that scene are projected onto a flat 2D plane, the depth information is lost. We have no way of knowing how far an object is from the camera or how the objects are positioned with respect to each other in the 3D space. This is where stereo vision comes into the picture.

Humans are very good at inferring depth information from the real world. The reason is that we have two eyes positioned a couple of inches from each other. Each eye acts as a camera and we capture two images of the same scene from two different viewpoints, that is, one image each using the left and right eyes. So, our brain takes these two images and builds a 3D map using stereo vision. This is what we want to achieve using stereo vision algorithms. We can capture two photos of the same scene using different viewpoints, and then match the corresponding points to obtain the depth map of the scene.

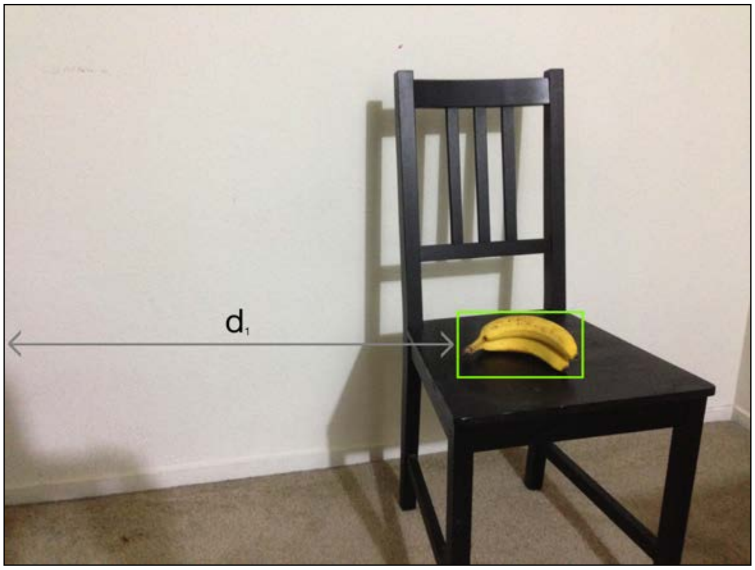
Let's consider the following image:



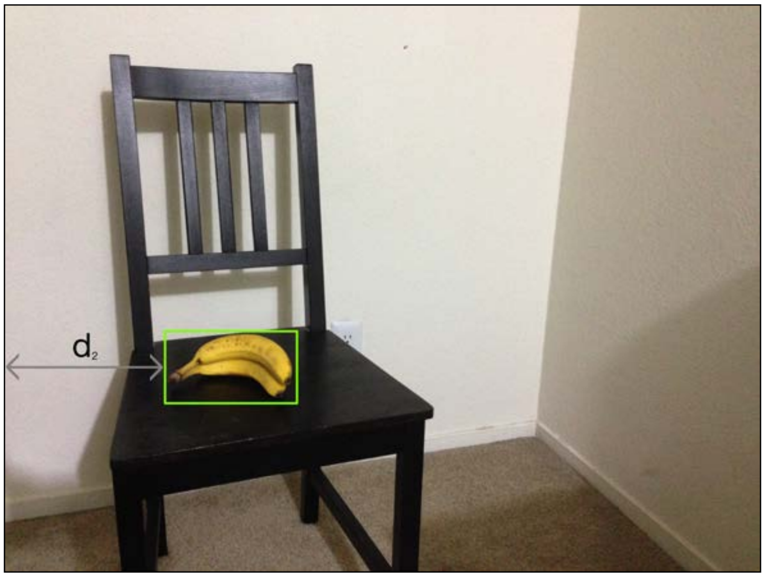
Now, if we capture the same scene from a different angle, it will look like this:



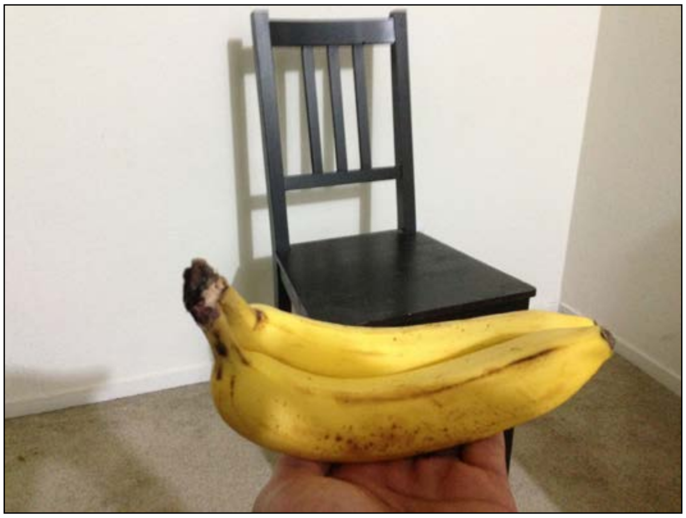
As you can see, there is a large amount of movement in the positions of the objects in the image. If you consider the pixel coordinates, the values of the initial position and final position will differ by a large amount in these two images. Consider the following image:



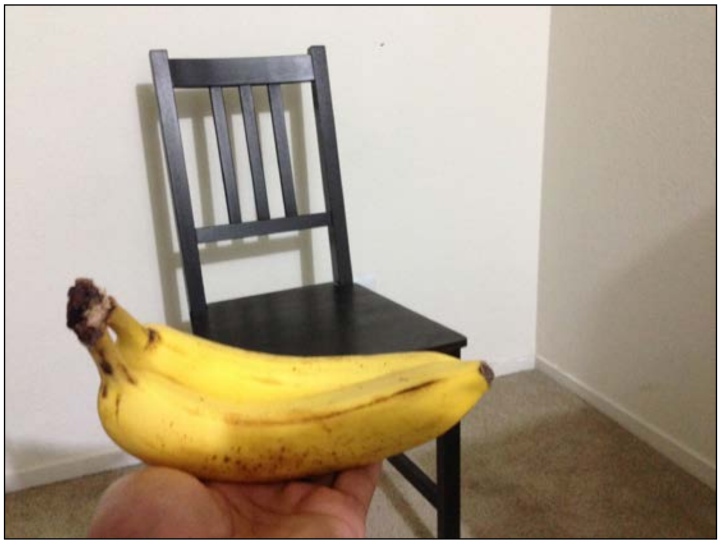
If we consider the same line of distance in the second image, it will look like this:



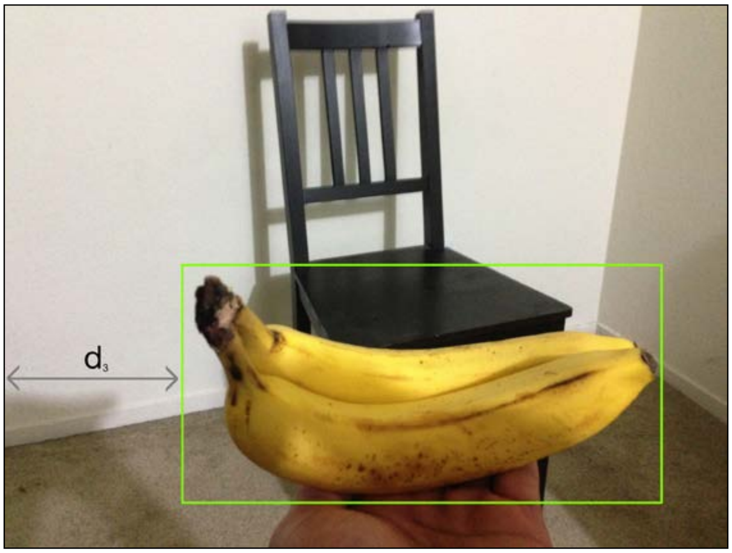
The difference between **d1** and **d2** is large. Now, let's bring the box closer to the camera:



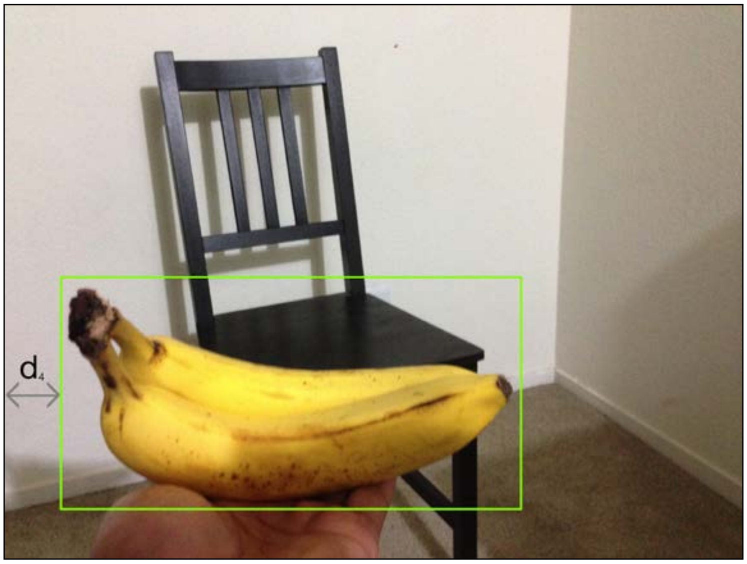
Now, let's move the camera by the same amount as we did earlier, and capture the same scene from this angle:



As you can see, the movement between the positions of the objects is not much. If you consider the pixel coordinates, you will see that the values are close to each other. The distance in the first image would be:



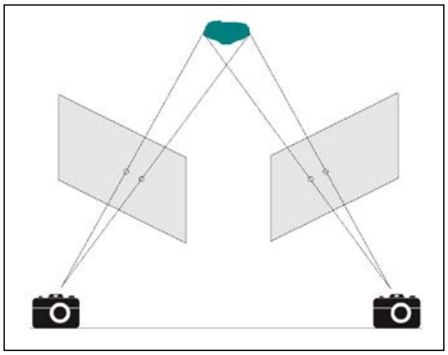
If we consider the same line of distance in the second image, it will be as shown in the following image:



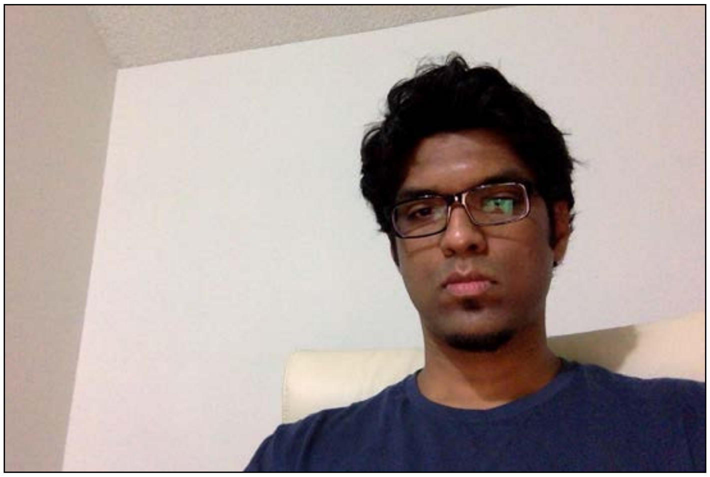
The difference between **d3** and **d4** is small. We can say that the absolute difference between **d1** and **d2** is greater than the absolute difference between **d3** and **d4**. Even though the camera moved by the same amount, there is a big difference between the apparent distances between the initial and final positions. This happens because we can bring the object closer to the camera; the apparent movement decreases when you capture two images from different angles. This is the concept behind stereo correspondence: we capture two images and use this knowledge to extract the depth information from a given scene.

What is epipolar geometry?

Before discussing epipolar geometry, let's discuss what happens when we capture two images of the same scene from two different viewpoints. Consider the following figure:



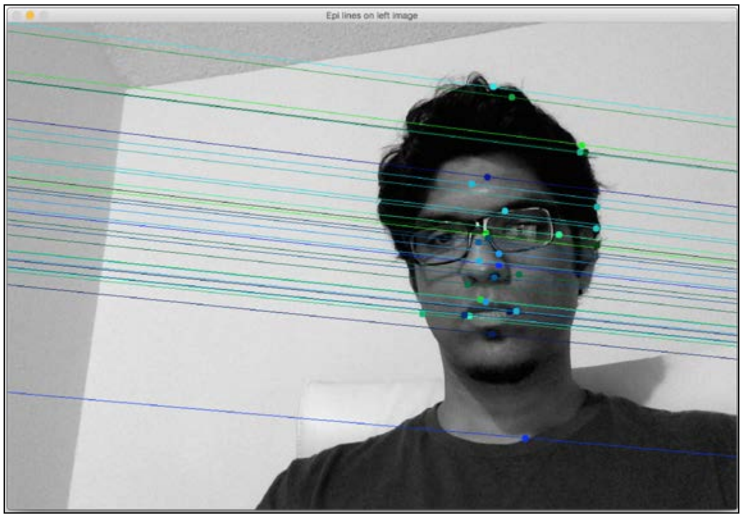
Let's see how it happens in real life. Consider the following image:



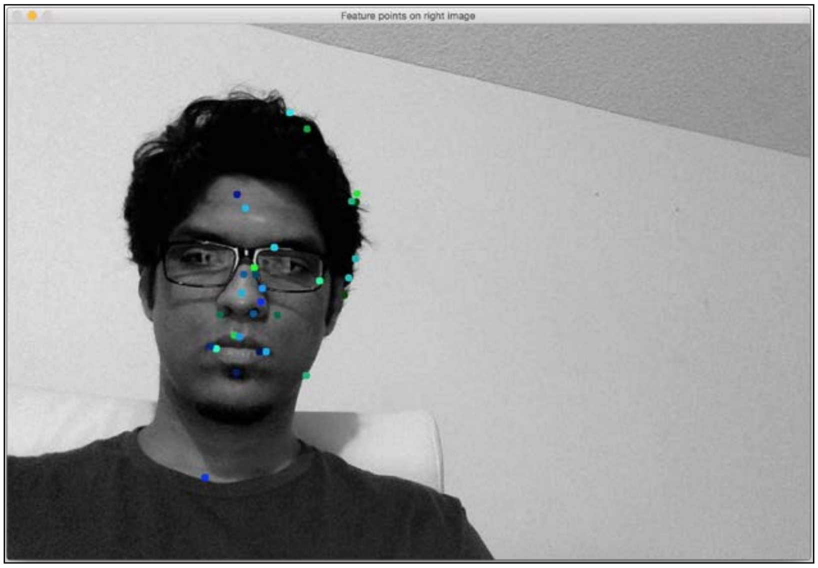
Now, let's capture the same scene from a different viewpoint:

Our goal is to match the keypoints in these two images to extract the scene information. The way we do this is by extracting a matrix that can associate the corresponding points between two stereo images. This is called the **fundamental matrix**.

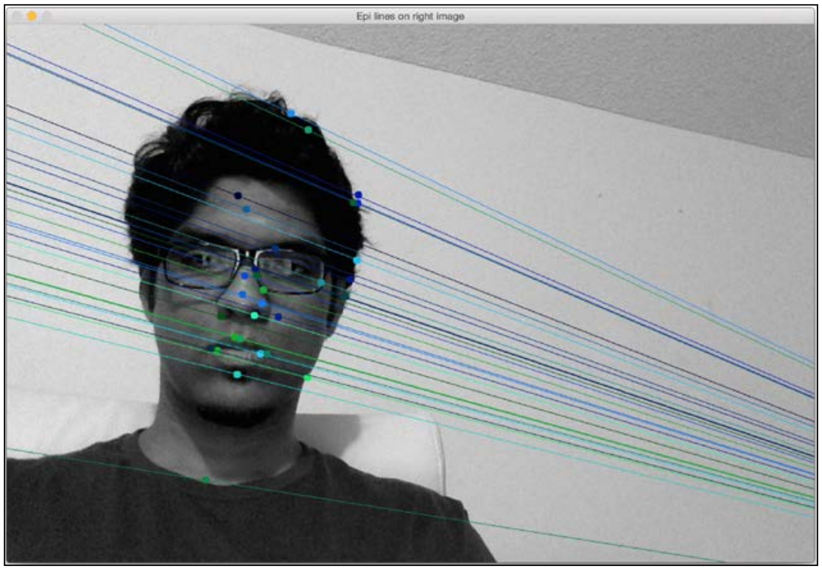
As we saw in the camera figure earlier, we can draw lines to see where they meet. These lines are called **epipolar lines**. The point at which the epipolar lines converge is called epipole. If you match the keypoints using SIFT, and draw the lines towards the meeting point on the left image, it will look like this:



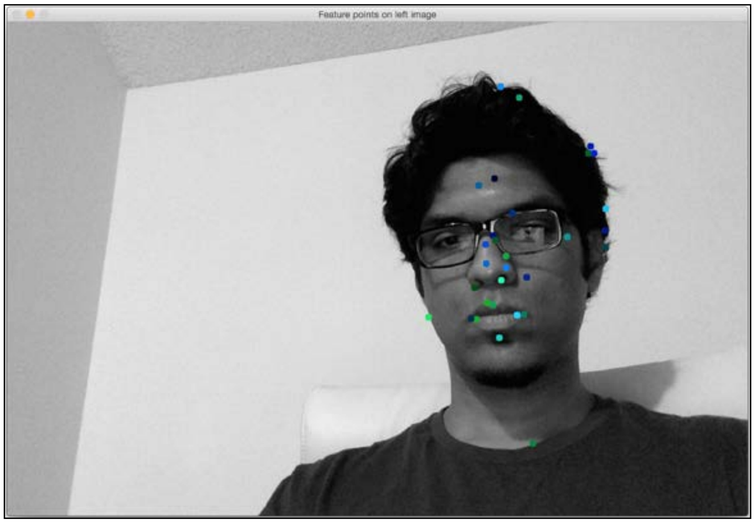
Following are the matching feature points in the right image:



The lines are epipolar lines. If you take the second image as the reference, they will appear as shown in the next image:



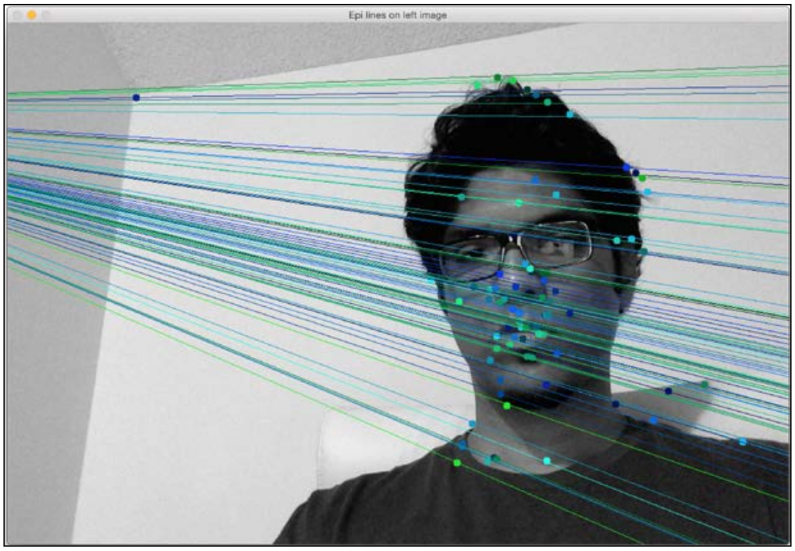
Following are the matching feature points in the first image:



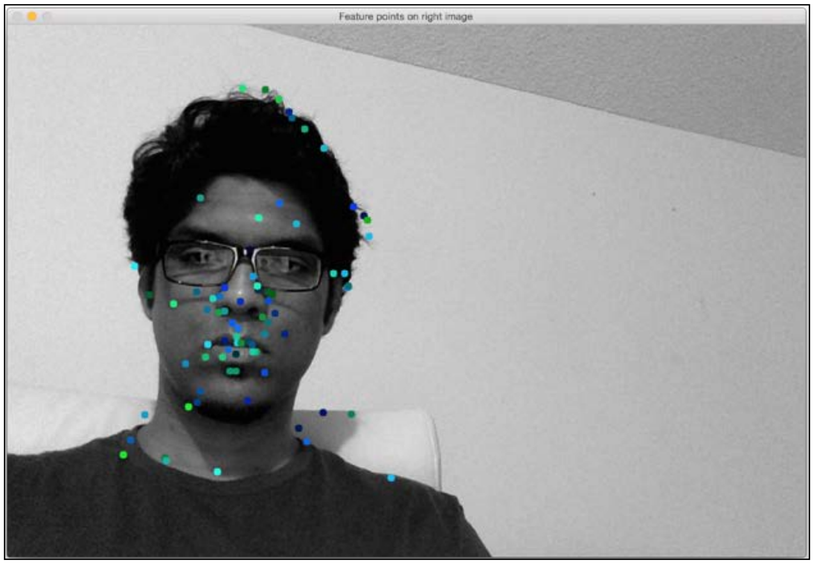
It's important to understand epipolar geometry and how we draw these lines. If two frames are positioned in 3D, then each epipolar line between the two frames must intersect the corresponding feature in each frame and each of the camera origins. This can be used to estimate the pose of the cameras with respect to the 3D environment. We will use this information later on, to extract 3D information from the scene. Let's take a look at the code:

```
import argparse

import cv2
import numpy as np

def build_arg_parser():
    parser = argparse.ArgumentParser(description='Find fundamental
    matrix \
            using the two input stereo images and draw epipolar
            lines')
    parser.add_argument("--img-left", dest="img_left",
    required=True,
            help="Image captured from the left view")
    parser.add_argument("--img-right", dest="img_right",
    required=True,
            help="Image captured from the right view")
    parser.add_argument("--feature-type", dest="feature_type",
            required=True, help="Feature extractor that will be
            used; can be either 'sift' or 'surf'")
    return parser

def draw_lines(img_left, img_right, lines, pts_left, pts_right):
    h,w = img_left.shape
    img_left = cv2.cvtColor(img_left, cv2.COLOR_GRAY2BGR)
    img_right = cv2.cvtColor(img_right, cv2.COLOR_GRAY2BGR)

    for line, pt_left, pt_right in zip(lines, pts_left,
    pts_right):
        x_start,y_start = map(int, [0, -line[2]/line[1] ])
        x_end,y_end = map(int, [w, -(line[2]+line[0]*w)/line[1] ])
        color = tuple(np.random.randint(0,255,2).tolist())
        cv2.line(img_left, (x_start,y_start), (x_end,y_end),
        color,1)
        cv2.circle(img_left, tuple(pt_left), 5, color, -1)
        cv2.circle(img_right, tuple(pt_right), 5, color, -1)

    return img_left, img_right

def get_descriptors(gray_image, feature_type):
    if feature_type == 'surf':
        feature_extractor = cv2.SURF()

    elif feature_type == 'sift':
        feature_extractor = cv2.SIFT()

    else:
        raise TypeError("Invalid feature type; should be either
        'surf' or 'sift'")

    keypoints, descriptors = feature_extractor.detectAndCompute(gray_
image, None)
    return keypoints, descriptors

if __name__=='__main__':
    args = build_arg_parser().parse_args()
    img_left = cv2.imread(args.img_left,0)  # left image
    img_right = cv2.imread(args.img_right,0)  # right image
    feature_type = args.feature_type

    if feature_type not in ['sift', 'surf']:
        raise TypeError("Invalid feature type; has to be either
        'sift' or 'surf'")

    scaling_factor = 1.0
    img_left = cv2.resize(img_left, None, fx=scaling_factor,
                fy=scaling_factor, interpolation=cv2.INTER_AREA)
    img_right = cv2.resize(img_right, None, fx=scaling_factor,
                fy=scaling_factor, interpolation=cv2.INTER_AREA)

    kps_left, des_left = get_descriptors(img_left, feature_type)
    kps_right, des_right = get_descriptors(img_right, feature_type)

    # FLANN parameters
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks=50)

    # Get the matches based on the descriptors
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des_left, des_right, k=2)

    pts_left_image = []
    pts_right_image = []

    # ratio test to retain only the good matches
    for i,(m,n) in enumerate(matches):
        if m.distance < 0.7*n.distance:
            pts_left_image.append(kps_left[m.queryIdx].pt)
            pts_right_image.append(kps_right[m.trainIdx].pt)

    pts_left_image = np.float32(pts_left_image)
    pts_right_image = np.float32(pts_right_image)
    F, mask = cv2.findFundamentalMat(pts_left_image,
    pts_right_image, cv2.FM_LMEDS)

    # Selecting only the inliers
    pts_left_image = pts_left_image[mask.ravel()==1]
    pts_right_image = pts_right_image[mask.ravel()==1]

    # Drawing the lines on left image and the corresponding feature
points on the right image
    lines1 = cv2.computeCorrespondEpilines
    (pts_right_image.reshape(-1,1,2), 2, F)
    lines1 = lines1.reshape(-1,3)
    img_left_lines, img_right_pts = draw_lines(img_left,
    img_right, lines1, pts_left_image, pts_right_image)

    # Drawing the lines on right image and the corresponding feature
points on the left image
    lines2 = cv2.computeCorrespondEpilines
    (pts_left_image.reshape(-1,1,2), 1,F)
    lines2 = lines2.reshape(-1,3)
    img_right_lines, img_left_pts = draw_lines(img_right,
    img_left, lines2, pts_right_image, pts_left_image)

    cv2.imshow('Epi lines on left image', img_left_lines)
    cv2.imshow('Feature points on right image', img_right_pts)
    cv2.imshow('Epi lines on right image', img_right_lines)
    cv2.imshow('Feature points on left image', img_left_pts)
    cv2.waitKey()
    cv2.destroyAllWindows()
```

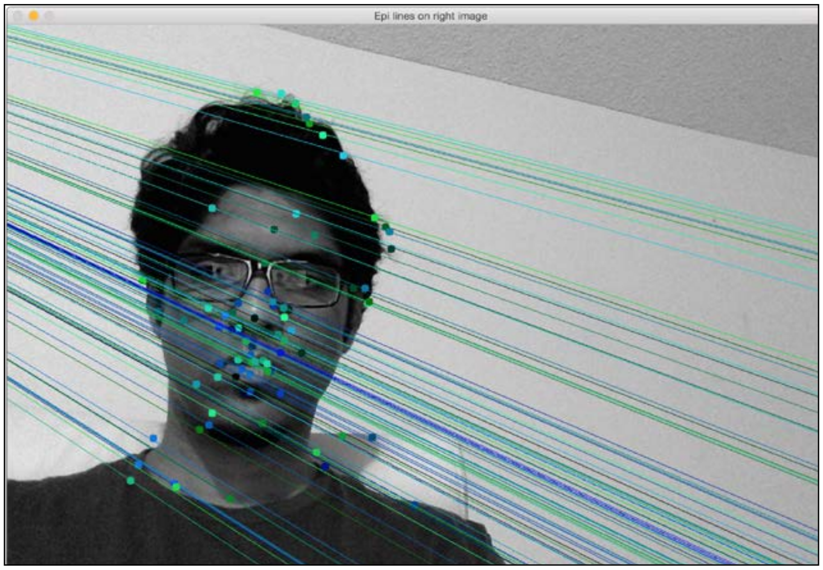
Let's see what happens if we use the **SURF** feature extractor. The lines in the left image will look like this:



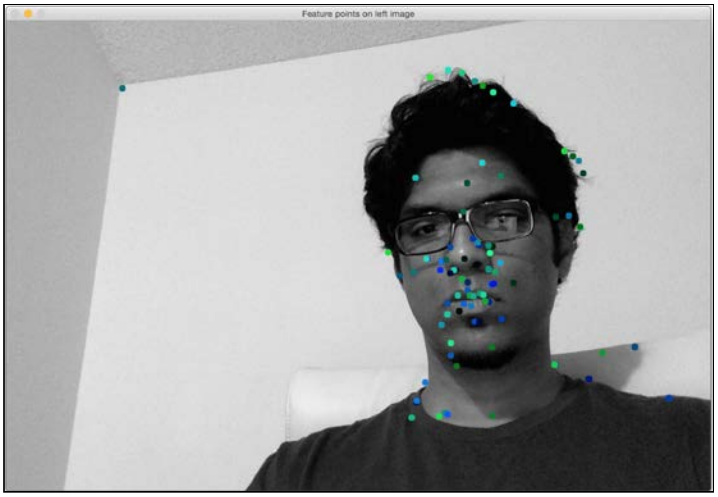
Following are the matching feature points in the right image:



If you take the second image as the reference, you will see something like the following image:



These are the matching feature points in the first image:

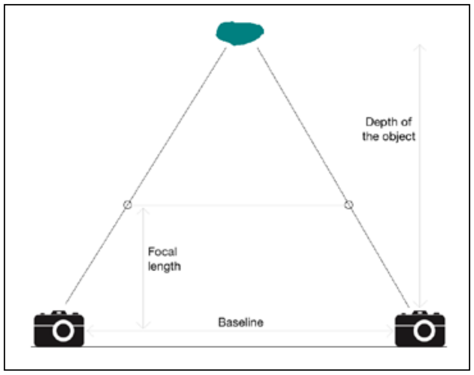


Why are the lines different as compared to SIFT?

SURF detects a different set of feature points, so the corresponding epipolar lines differ as well. As you can see in the images, there are more feature points detected when we use SURF. Since we have more information than before, the corresponding epipolar lines will also change accordingly.

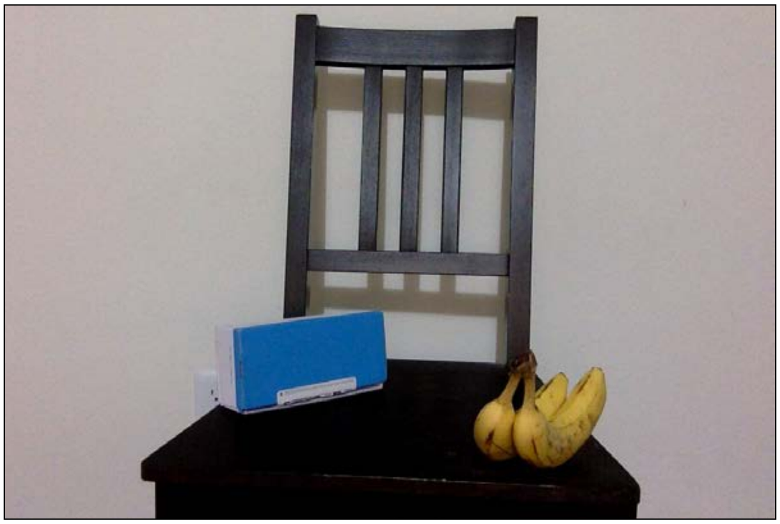
Building the 3D map

Now that we are familiar with epipolar geometry, let's see how to use it to build a 3D map based on stereo images. Let's consider the following figure:

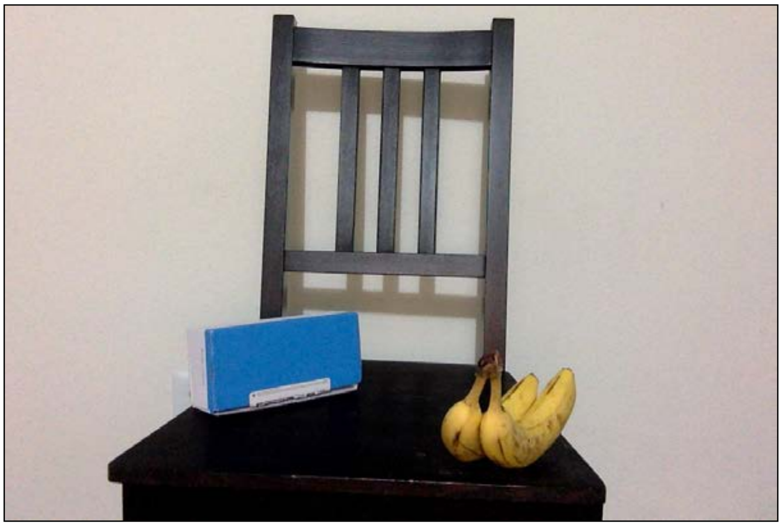


The first step is to extract the disparity map between the two images. If you look at the figure, as we go closer to the object from the cameras along the connecting lines, the distance decreases between the points. Using this information, we can infer the distance of each point from the camera. This is called a depth map. Once we find the matching points between the two images, we can find the disparity by using epipolar lines to impose epipolar constraints.

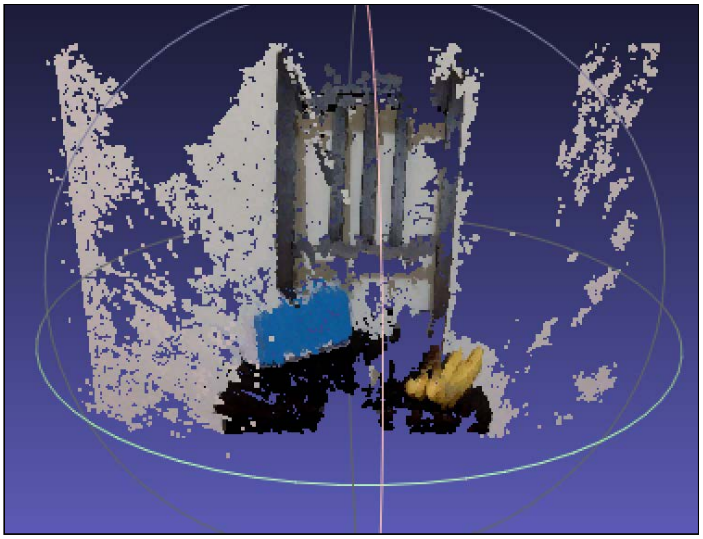
Let's consider the following image:



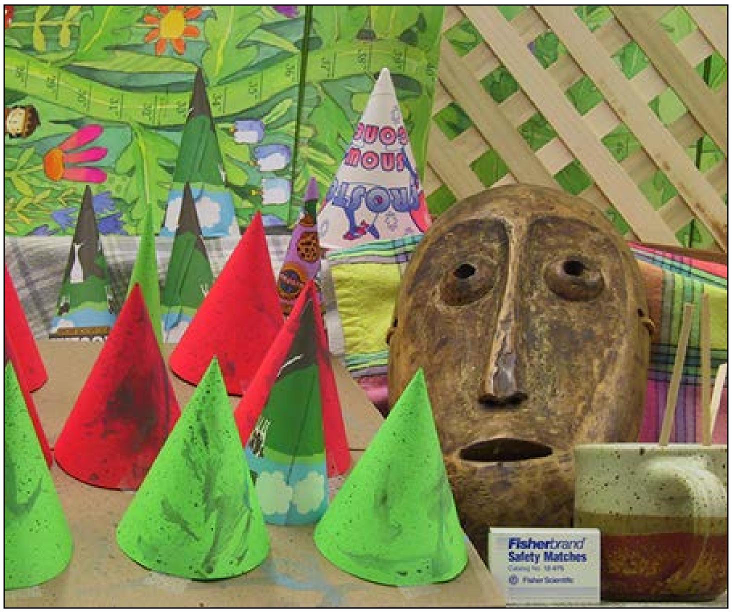
If we capture the same scene from a different position, we get the following image:



If we reconstruct the 3D map, it will look like this:

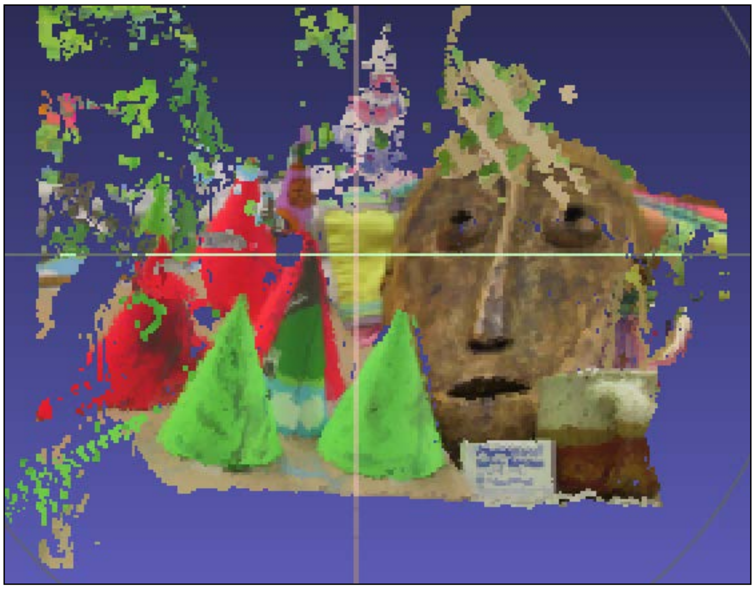


Bear in mind that these images were not captured using perfectly aligned stereo cameras. That's the reason the 3D map looks so noisy! This is just to demonstrate how we can reconstruct the real world using stereo images. Let's consider an image pair captured using stereo cameras that are properly aligned. Following is the left view image:

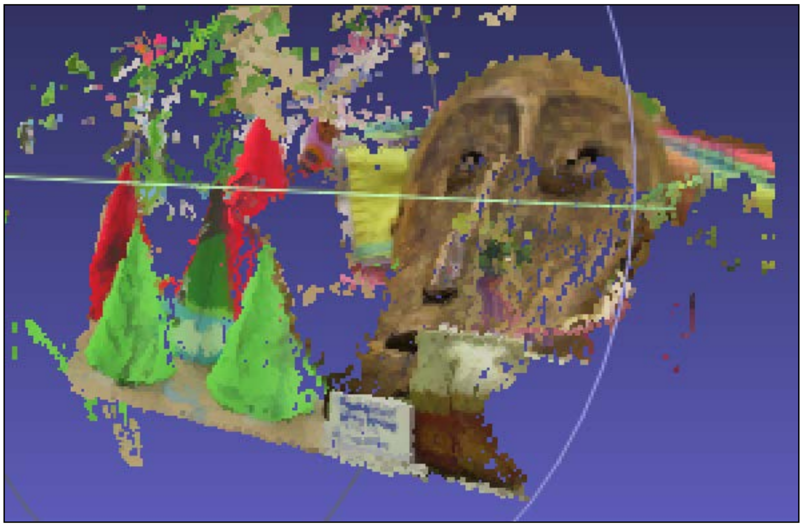


Next is the corresponding right view image:

If you extract the depth information and build the 3D map, it will look like this:



Let's rotate it to see if the depth is right for the different objects in the scene:



You need a software called **MeshLab** to visualize the 3D scene. We'll discuss about it soon. As we can see in the preceding images, the items are correctly aligned according to their distance from the camera. We can intuitively see that they are arranged in the right way, including the tilted position of the mask. We can use this technique to build many interesting things.

Let's see how to do it in OpenCV-Python:

```
import argparse

import cv2
import numpy as np

def build_arg_parser():
    parser = argparse.ArgumentParser(description='Reconstruct the
    3D map from \
            the two input stereo images. Output will be saved in
            \'output.ply\'')
    parser.add_argument("--image-left", dest="image_left",
    required=True,
            help="Input image captured from the left")
    parser.add_argument("--image-right", dest="image_right",
    required=True,
            help="Input image captured from the right")
    parser.add_argument("--output-file", dest="output_file",
    required=True,
            help="Output filename (without the extension) where
            the point cloud will be saved")
    return parser

def create_output(vertices, colors, filename):
    colors = colors.reshape(-1, 3)
    vertices = np.hstack([vertices.reshape(-1,3), colors])

    ply_header = '''ply
        format ascii 1.0
        element vertex %(vert_num)d
        property float x
        property float y
        property float z
        property uchar red
        property uchar green
        property uchar blue
        end_header
    '''

    with open(filename, 'w') as f:
        f.write(ply_header % dict(vert_num=len(vertices)))
        np.savetxt(f, vertices, '%f %f %f %d %d %d')

if __name__ == '__main__':
    args = build_arg_parser().parse_args()
    image_left = cv2.imread(args.image_left)
    image_right = cv2.imread(args.image_right)
    output_file = args.output_file + '.ply'

    if image_left.shape[0] != image_right.shape[0] or \
            image_left.shape[1] != image_right.shape[1]:
        raise TypeError("Input images must be of the same size")

    # downscale images for faster processing
    image_left = cv2.pyrDown(image_left)
    image_right = cv2.pyrDown(image_right)

    # disparity range is tuned for 'aloe' image pair
    win_size = 1
    min_disp = 16
    max_disp = min_disp * 9
    num_disp = max_disp - min_disp   # Needs to be divisible by 16
    stereo = cv2.StereoSGBM(minDisparity = min_disp,
        numDisparities = num_disp,
        SADWindowSize = win_size,
        uniquenessRatio = 10,
        speckleWindowSize = 100,
        speckleRange = 32,
        disp12MaxDiff = 1,
        P1 = 8*3*win_size**2,
        P2 = 32*3*win_size**2,
        fullDP = True
    )

    print "\nComputing the disparity map ..."
    disparity_map = stereo.compute(image_left,
image_right).astype(np.float32) / 16.0

    print "\nGenerating the 3D map ..."
    h, w = image_left.shape[:2]
    focal_length = 0.8*w

    # Perspective transformation matrix
    Q = np.float32([[1, 0, 0, -w/2.0],
                    [0,-1, 0,  h/2.0],
                    [0, 0, 0, -focal_length],
                    [0, 0, 1, 0]])

    points_3D = cv2.reprojectImageTo3D(disparity_map, Q)
    colors = cv2.cvtColor(image_left, cv2.COLOR_BGR2RGB)
    mask_map = disparity_map > disparity_map.min()
    output_points = points_3D[mask_map]
    output_colors = colors[mask_map]

    print "\nCreating the output file ...\n"
    create_output(output_points, output_colors, output_file)
```

To visualize the output, you need to download MeshLab from `http://meshlab.sourceforge.net`.

Just open the `output.ply` file using MeshLab and you'll see the 3D image. You can rotate it to get a complete 3D view of the reconstructed scene. Some of the alternatives to MeshLab are Sketchup on OS X and Windows, and Blender on Linux.

Summary

In this chapter, we learned about stereo vision and 3D reconstruction. We discussed how to extract the fundamental matrix using different feature extractors. We learned how to generate the disparity map between two images, and use it to reconstruct the 3D map of a given scene.

In the next chapter, we are going to discuss augmented reality, and how we can build a cool application where we overlay graphics on top of real world objects in a live video.

11
Augmented Reality

In this chapter, you are going to learn about augmented reality and how you can use it to build cool applications. We will discuss pose estimation and plane tracking. You will learn how to map the coordinates from 2D to 3D, and how we can overlay graphics on top of a live video.

By the end of this chapter, you will know:

- What is the premise of augmented reality
- What is pose estimation
- How to track a planar object
- How to map coordinates from 3D to 2D
- How to overlay graphics on top of a video in real time

What is the premise of augmented reality?

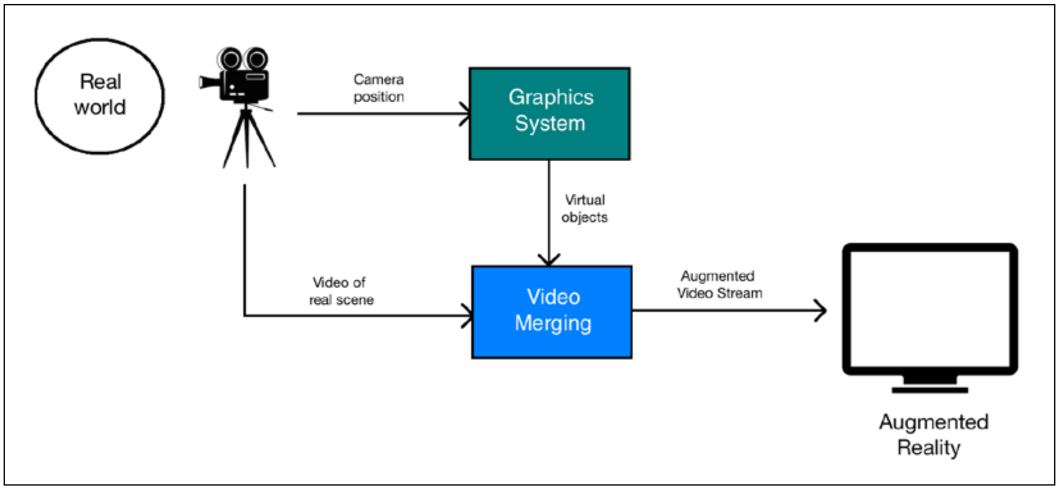
Before we jump into all the fun stuff, let's understand what augmented reality means. You would have probably seen the term "augmented reality" being used in a variety of contexts. So, we should understand the premise of augmented reality before we start discussing the implementation details. Augmented Reality refers to the superposition of computer-generated input such as imagery, sounds, graphics, and text on top of the real world.

Augmented reality tries to blur the line between what's real and what's computer-generated by seamlessly merging the information and enhancing what we see and feel. It is actually closely related to a concept called mediated reality where a computer modifies our view of the reality. As a result of this, the technology works by enhancing our current perception of reality. Now the challenge here is to make it look seamless to the user. It's easy to just overlay something on top of the input video, but we need to make it look like it is part of the video. The user should feel that the computer-generated input is closely following the real world. This is what we want to achieve when we build an augmented reality system.

Computer vision research in this context explores how we can apply computer-generated imagery to live video streams so that we can enhance the perception of the real world. Augmented reality technology has a wide variety of applications including, but not limited to, head-mounted displays, automobiles, data visualization, gaming, construction, and so on. Now that we have powerful smartphones and smarter machines, we can build high-end augmented reality applications with ease.

What does an augmented reality system look like?

Let's consider the following figure:



As we can see here, the camera captures the real world video to get the reference point. The graphics system generates the virtual objects that need to be overlaid on top of the video. Now the video-merging block is where all the magic happens. This block should be smart enough to understand how to overlay the virtual objects on top of the real world in the best way possible.

Geometric transformations for augmented reality

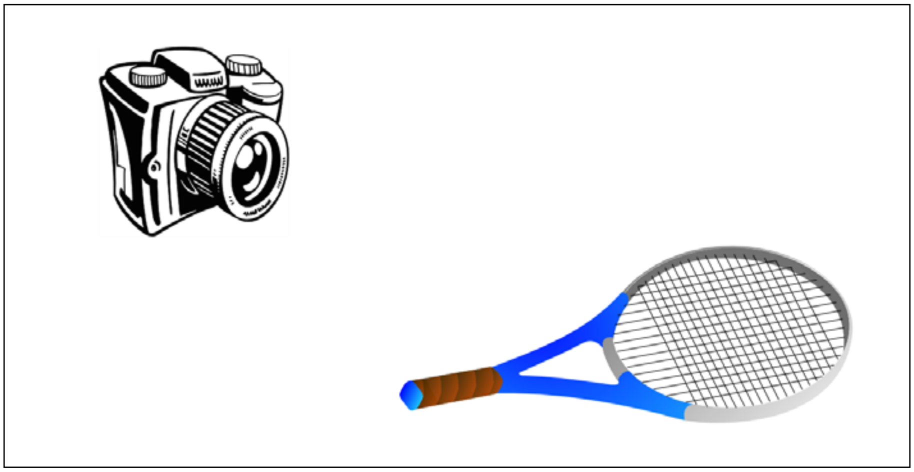
The outcome of augmented reality is amazing, but there are a lot of mathematical things going on underneath. Augmented reality utilizes a lot of geometric transformations and the associated mathematical functions to make sure everything looks seamless. When talking about a live video for augmented reality, we need to precisely register the virtual objects on top of the real world. To understand it better, let's think of it as an alignment of two cameras—the real one through which we see the world, and the virtual one that projects the computer generated graphical objects.

In order to build an augmented reality system, the following geometric transformations need to be established:

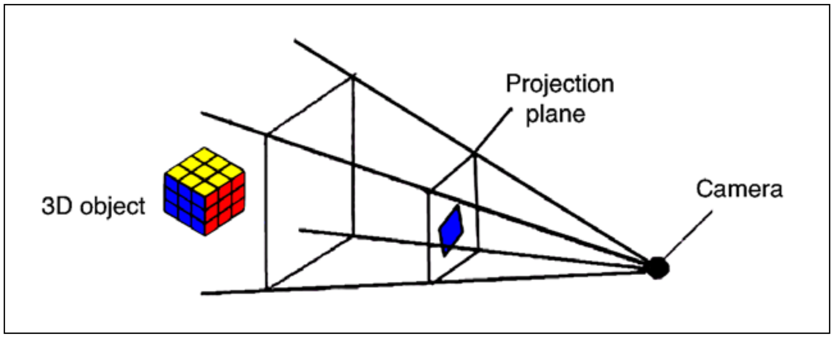
- **Object-to-scene**: This transformation refers to transforming the 3D coordinates of a virtual object and expressing them in the coordinate frame of our real-world scene. This ensures that we are positioning the virtual object in the right location.
- **Scene-to-camera**: This transformation refers to the pose of the camera in the real world. By "pose", we mean the orientation and location of the camera. We need to estimate the point of view of the camera so that we know how to overlay the virtual object.
- **Camera-to-image**: This refers to the calibration parameters of the camera. This defines how we can project a 3D object onto a 2D image plane. This is the image that we will actually see in the end.

Consider the following image:

As we can see here, the car is trying to fit into the scene but it looks very artificial. If we don't convert the coordinates in the right way, it looks unnatural. This is what we were talking about in the object-to-scene transformation! Once we transform the 3D coordinates of the virtual object into the coordinate frame of the real world, we need to estimate the pose of the camera:



We need to understand the position and rotation of the camera because that's what the user will see. Once we estimate the camera pose, we are ready to put this 3D scene on a 2D image.

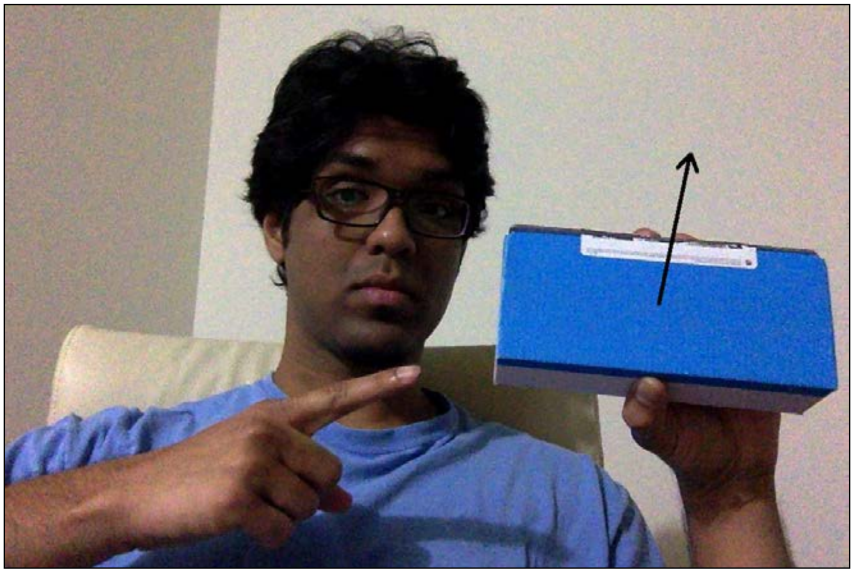


Once we have these transformations, we can build the complete system.

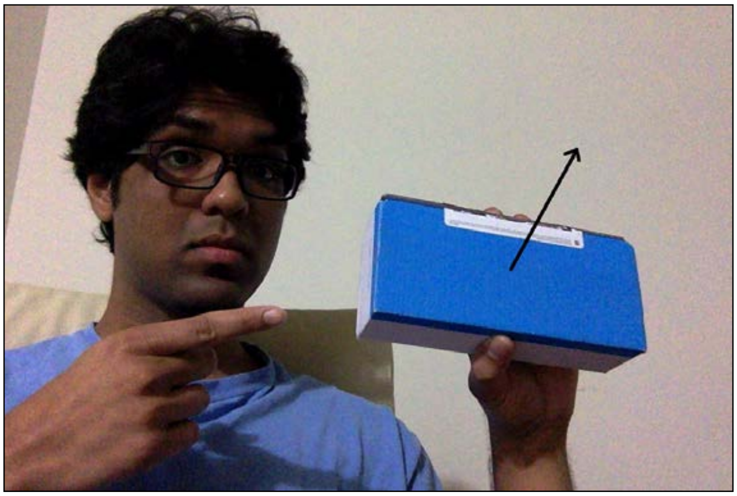
What is pose estimation?

Before we proceed, we need to understand how to estimate the camera pose. This is a very critical step in an augmented reality system and we need to get it right if we want our experience to be seamless. In the world of augmented reality, we overlay graphics on top of an object in real time. In order to do that, we need to know the location and orientation of the camera, and we need to do it quickly. This is where pose estimation becomes very important. If you don't track the pose correctly, the overlaid graphics will not look natural.

Consider the following image:



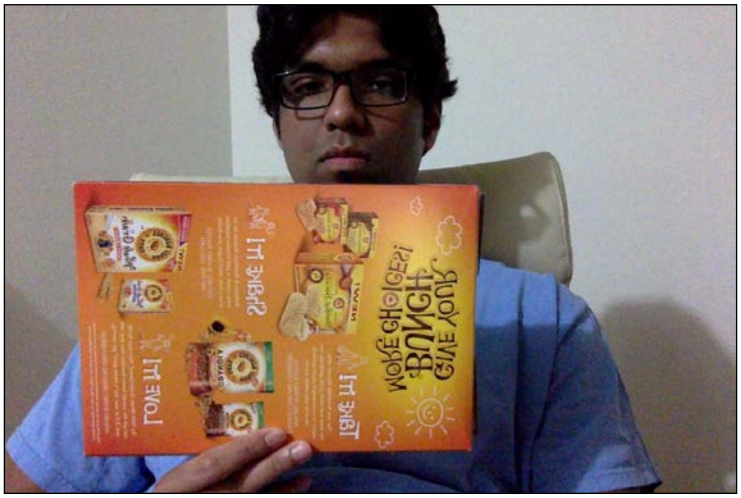
The arrow line represents that the surface is normal. Let's say the object changes its orientation:



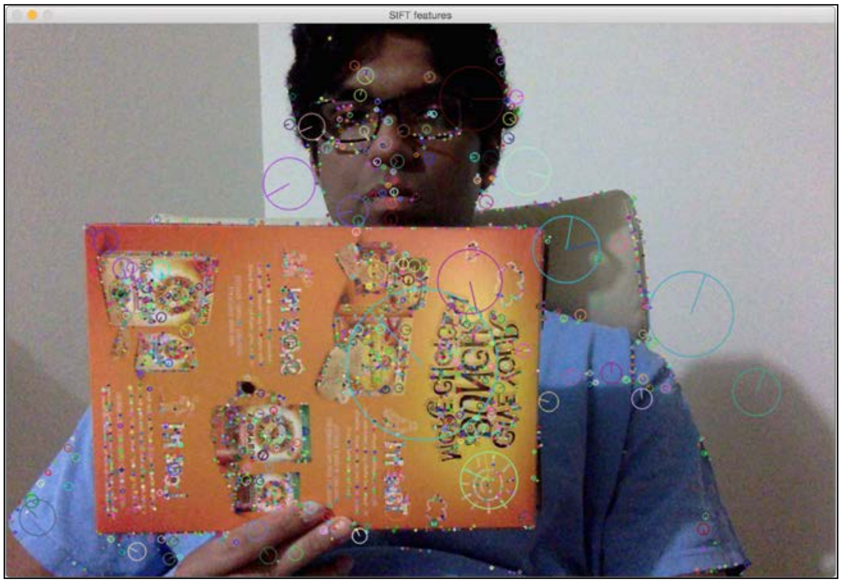
Now even though the location is the same, the orientation has changed. We need to have this information so that the overlaid graphics looks natural. We need to make sure that it's aligned to this orientation as well as position.

How to track planar objects?

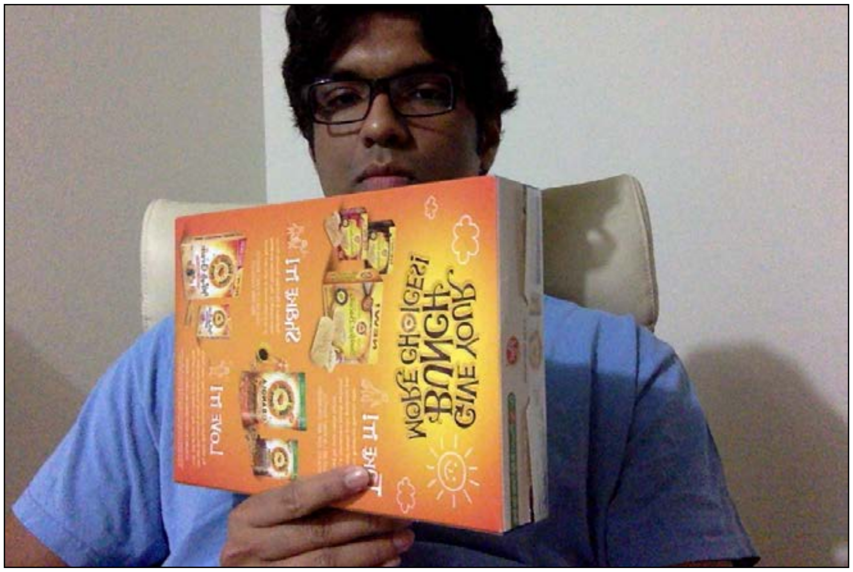
Now that you understand what pose estimation is, let's see how you can use it to track planar objects. Let's consider the following planar object:



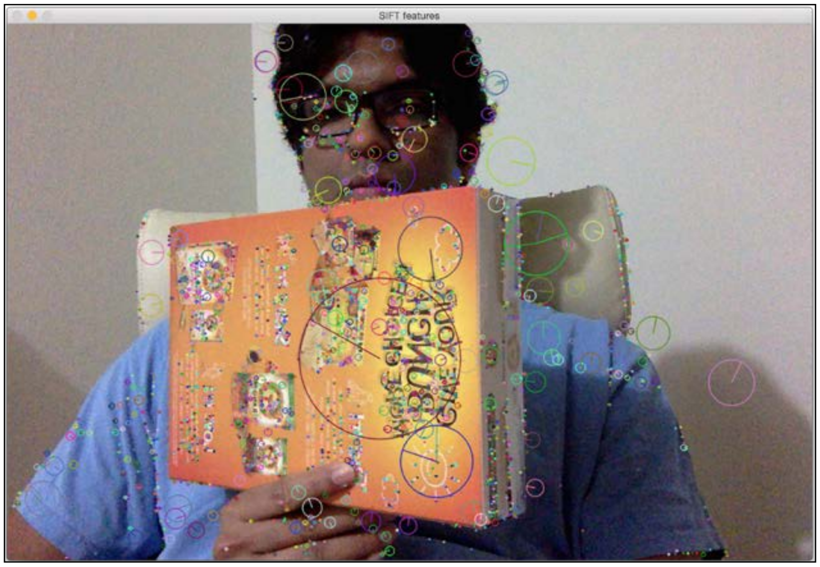
Now if we extract feature points from this image, we will see something like this:



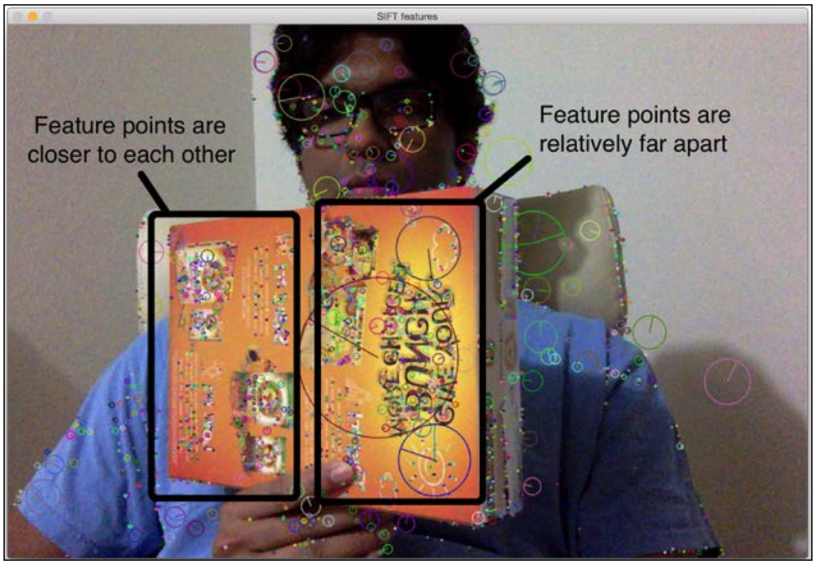
Let's tilt the cardboard:



As we can see, the cardboard is tilted in this image. Now if we want to make sure our virtual object is overlaid on top of this surface, we need to gather this planar tilt information. One way to do this is by using the relative positions of those feature points. If we extract the feature points from the preceding image, it will look like this:



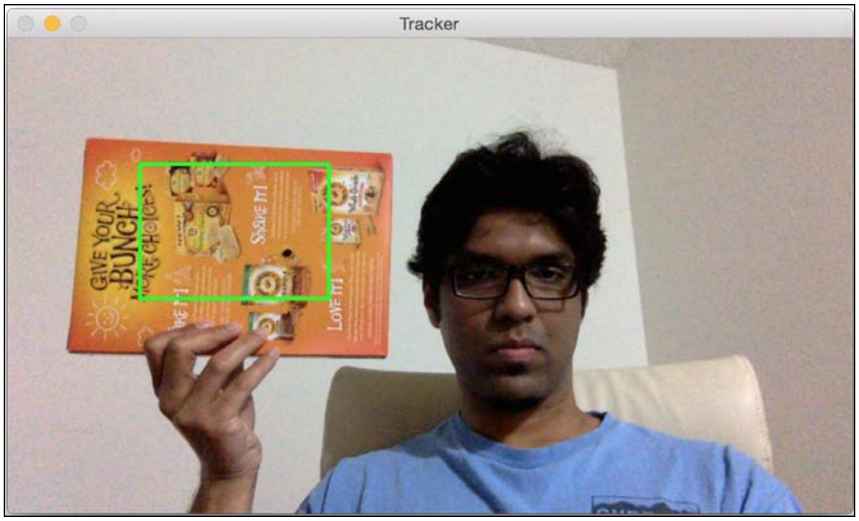
As you can see, the feature points got closer horizontally on the far end of the plane as compared to the ones on the near end.


SIFT features
Feature points are closer to each other
Feature points are relatively far apart

So we can utilize this information to extract the orientation information from the image. If you remember, we discussed perspective transformation in detail when we were discussing geometric transformations as well as panoramic imaging. All we need to do is use those two sets of points and extract the homography matrix. This homography matrix will tell us how the cardboard turned.

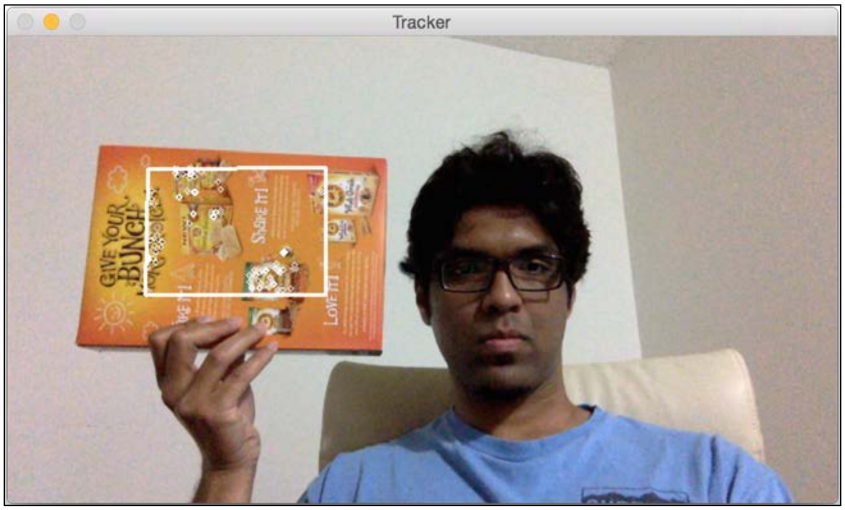
Consider the following image:

We start by selecting the region of interest.

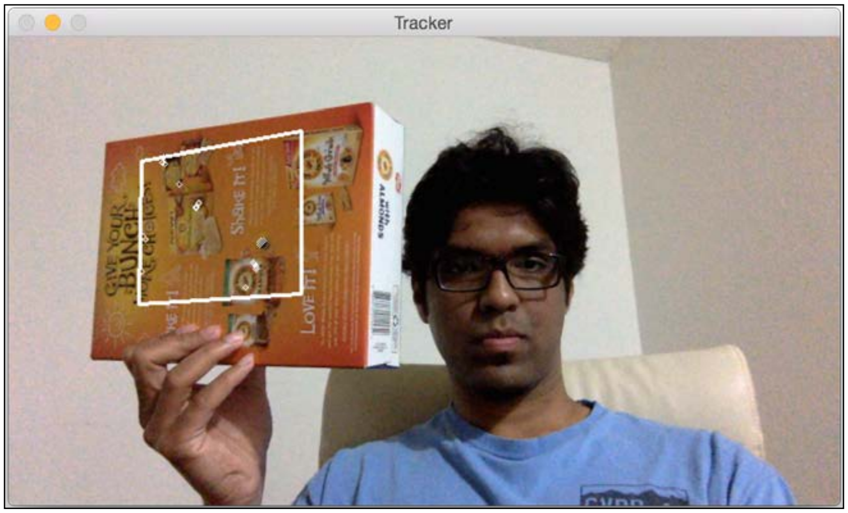


We then extract feature points from this region of interest. Since we are tracking planar objects, the algorithm assumes that this region of interest is a plane. That was obvious, but it's better to state it explicitly! So make sure you have a cardboard in your hand when you select this region of interest. Also, it'll be better if the cardboard has a bunch of patterns and distinctive points so that it's easy to detect and track the feature points on it.

Let the tracking begin! We'll move the cardboard around to see what happens:



As you can see, the feature points are being tracked inside the region of interest. Let's tilt it and see what happens:



Looks like the feature points are being tracked properly. As we can see, the overlaid rectangle is changing its orientation according to the surface of the cardboard.

Here is the code to do it:

```
import sys
from collections import namedtuple

import cv2
import numpy as np

class PoseEstimator(object):
    def __init__(self):
        # Use locality sensitive hashing algorithm
        flann_params = dict(algorithm = 6, table_number = 6,
                key_size = 12, multi_probe_level = 1)

        self.min_matches = 10
        self.cur_target = namedtuple('Current', 'image, rect,
        keypoints, descriptors, data')
        self.tracked_target = namedtuple('Tracked', 'target,
        points_prev, points_cur, H, quad')

        self.feature_detector = cv2.ORB(nfeatures=1000)
        self.feature_matcher = cv2.FlannBasedMatcher(flann_params,
{})
        self.tracking_targets = []

    # Function to add a new target for tracking
    def add_target(self, image, rect, data=None):
        x_start, y_start, x_end, y_end = rect
        keypoints, descriptors = [], []
        for keypoint, descriptor in zip(*self.detect_features(image)):
            x, y = keypoint.pt
            if x_start <= x <= x_end and y_start <= y <= y_end:
                keypoints.append(keypoint)
                descriptors.append(descriptor)

        descriptors = np.array(descriptors, dtype='uint8')
        self.feature_matcher.add([descriptors])
        target = self.cur_target(image=image, rect=rect,
        keypoints=keypoints,
                    descriptors=descriptors, data=None)
        self.tracking_targets.append(target)
    # To get a list of detected objects
    def track_target(self, frame):
        self.cur_keypoints, self.cur_descriptors =
        self.detect_features(frame)
        if len(self.cur_keypoints) < self.min_matches:
            return []

        matches =
        self.feature_matcher.knnMatch(self.cur_descriptors, k=2)
        matches = [match[0] for match in matches if len(match) ==
        2 and
                    match[0].distance < match[1].distance * 0.75]
        if len(matches) < self.min_matches:
            return []

        matches_using_index = [[] for _ in
        xrange(len(self.tracking_targets))]
        for match in matches:
            matches_using_index[match.imgIdx].append(match)

        tracked = []
        for image_index, matches in
        enumerate(matches_using_index):
            if len(matches) < self.min_matches:
                continue

            target = self.tracking_targets[image_index]
            points_prev = [target.keypoints[m.trainIdx].pt for m
            in matches]
            points_cur = [self.cur_keypoints[m.queryIdx].pt for m
            in matches]
            points_prev, points_cur = np.float32((points_prev,
            points_cur))
            H, status = cv2.findHomography(points_prev,
            points_cur, cv2.RANSAC, 3.0)
            status = status.ravel() != 0
            if status.sum() < self.min_matches:
                continue

            points_prev, points_cur = points_prev[status],
            points_cur[status]

            x_start, y_start, x_end, y_end = target.rect
            quad = np.float32([[x_start, y_start], [x_end,
            y_start], [x_end, y_end], [x_start, y_end]])
            quad = cv2.perspectiveTransform(quad.reshape(1, -1,
            2), H).reshape(-1, 2)

            track = self.tracked_target(target=target,
            points_prev=points_prev,
                        points_cur=points_cur, H=H, quad=quad)
            tracked.append(track)

        tracked.sort(key = lambda x: len(x.points_prev),
        reverse=True)
        return tracked

    # Detect features in the selected ROIs and return the keypoints
and descriptors
    def detect_features(self, frame):
        keypoints, descriptors = self.feature_detector.
detectAndCompute(frame, None)
        if descriptors is None:
            descriptors = []

        return keypoints, descriptors

    # Function to clear all the existing targets
    def clear_targets(self):
        self.feature_matcher.clear()
        self.tracking_targets = []

class VideoHandler(object):
    def __init__(self):
        self.cap = cv2.VideoCapture(0)
        self.paused = False
        self.frame = None
        self.pose_tracker = PoseEstimator()

        cv2.namedWindow('Tracker')
        self.roi_selector = ROISelector('Tracker', self.on_rect)

    def on_rect(self, rect):
        self.pose_tracker.add_target(self.frame, rect)

    def start(self):
        while True:
            is_running = not self.paused and self.roi_selector.
selected_rect is None
```

```
            if is_running or self.frame is None:
                ret, frame = self.cap.read()
                scaling_factor = 0.5
                frame = cv2.resize(frame, None, fx=scaling_factor,
                fy=scaling_factor,
                        interpolation=cv2.INTER_AREA)
                if not ret:
                    break

                self.frame = frame.copy()

            img = self.frame.copy()
            if is_running:
                tracked =
                self.pose_tracker.track_target(self.frame)
                for item in tracked:
                    cv2.polylines(img, [np.int32(item.quad)],
                    True, (255, 255, 255), 2)
                    for (x, y) in np.int32(item.points_cur):
                        cv2.circle(img, (x, y), 2, (255, 255,
                        255))

            self.roi_selector.draw_rect(img)
            cv2.imshow('Tracker', img)
            ch = cv2.waitKey(1)
            if ch == ord(' '):
                self.paused = not self.paused
            if ch == ord('c'):
                self.pose_tracker.clear_targets()
            if ch == 27:
                break

class ROISelector(object):
    def __init__(self, win_name, callback_func):
        self.win_name = win_name
        self.callback_func = callback_func
        cv2.setMouseCallback(self.win_name, self.on_mouse_event)
        self.selection_start = None
        self.selected_rect = None

    def on_mouse_event(self, event, x, y, flags, param):
        if event == cv2.EVENT_LBUTTONDOWN:
            self.selection_start = (x, y)
        if self.selection_start:
            if flags & cv2.EVENT_FLAG_LBUTTON:
                x_orig, y_orig = self.selection_start
                x_start, y_start = np.minimum([x_orig, y_orig],
                [x, y])
                x_end, y_end = np.maximum([x_orig, y_orig], [x,
                y])
                self.selected_rect = None
                if x_end > x_start and y_end > y_start:
                    self.selected_rect = (x_start, y_start, x_end,
                    y_end)
            else:
                rect = self.selected_rect
                self.selection_start = None
                self.selected_rect = None
                if rect:
                    self.callback_func(rect)

    def draw_rect(self, img):
        if not self.selected_rect:
            return False

        x_start, y_start, x_end, y_end = self.selected_rect
        cv2.rectangle(img, (x_start, y_start), (x_end, y_end), (0,
        255, 0), 2)
        return True

if __name__ == '__main__':
    VideoHandler().start()
```

What happened inside the code?

To start with, we have a PoseEstimator class that does all the heavy lifting here. We need something to detect the features in the image and something to match the features between successive images. So we use the ORB feature detector and the Flann feature matcher. As we can see, we initialize the class with these parameters in the constructor.

Whenever we select a region of interest, we call the add_target method to add that to our list of tracking targets. This method just extracts the features from that region of interest and stores in one of the class variables. Now that we have a target, we are ready to track it!

The `track_target` method handles all the tracking. We take the current frame and extract all the keypoints. However, we are not really interested in all the keypoints in the current frame of the video. We just want the keypoints that belong to our target object. So now, our job is to find the closest keypoints in the current frame.

We now have a set of keypoints in the current frame and we have another set of keypoints from our target object in the previous frame. The next step is to extract the homography matrix from these matching points. This homography matrix tells us how to transform the overlaid rectangle so that it's aligned with the cardboard surface. We just need to take this homography matrix and apply it to the overlaid rectangle to obtain the new positions of all its points.

How to augment our reality?

Now that we know how to track planar objects, let's see how to overlay 3D objects on top of the real world. The objects are 3D but the video on our screen is 2D. So the first step here is to understand how to map those 3D objects to 2D surfaces so that it looks realistic. We just need to project those 3D points onto planar surfaces.

Mapping coordinates from 3D to 2D

Once we estimate the pose, we project the points from the 3D to the 2D. Consider the following image:

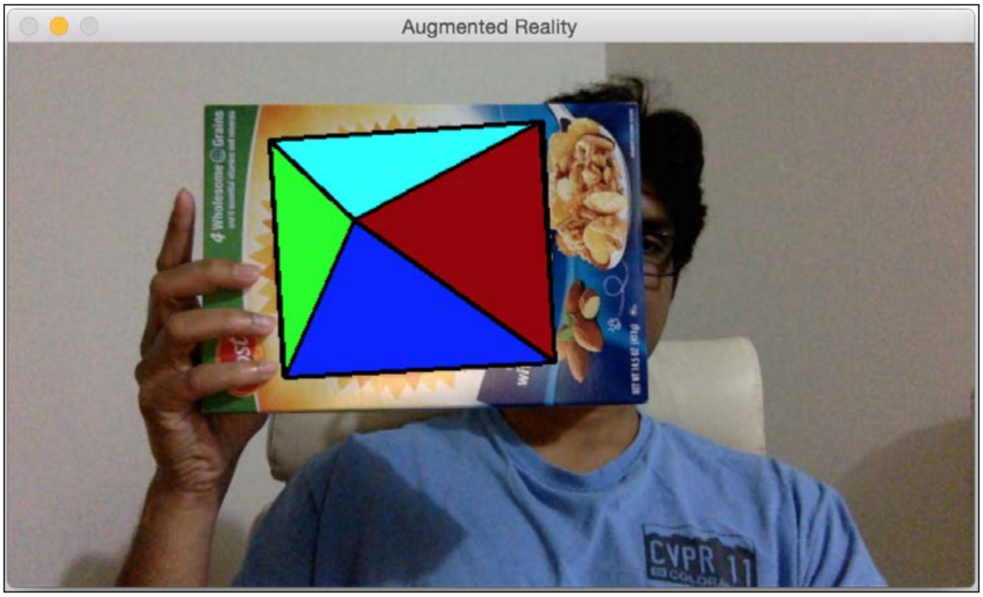
As we can see here, the TV remote control is a 3D object but we are seeing it on a 2D plane. Now if we move it around, it will look like this:

This 3D object is still on a 2D plane. The object has moved to a different location and the distance from the camera has changed as well. How do we compute these coordinates? We need a mechanism to map this 3D object onto the 2D surface. This is where the 3D to 2D projection becomes really important.

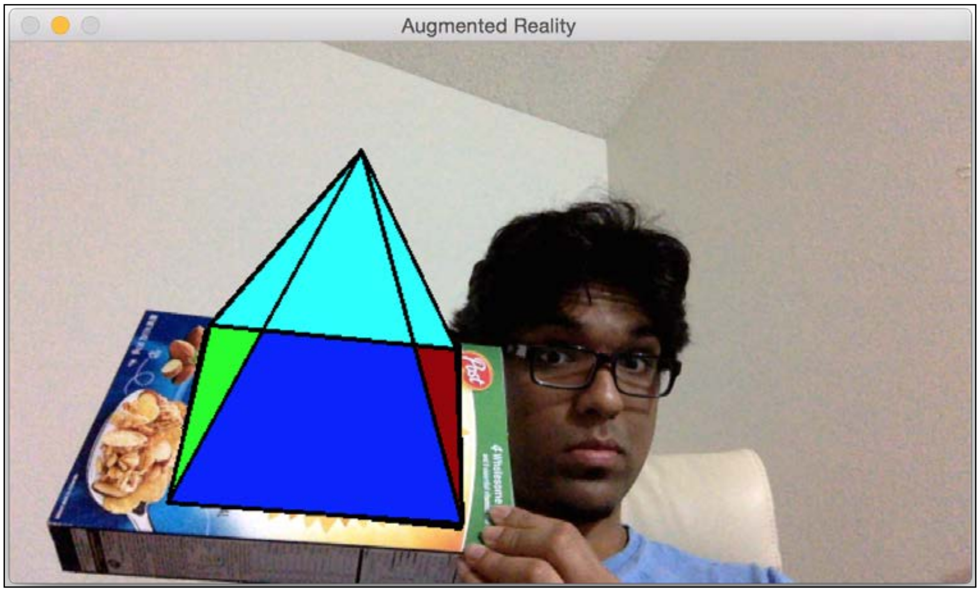
We just need to estimate the initial camera pose to start with. Now, let's assume that the intrinsic parameters of the camera are already known. So we can just use the `solvePnP` function in OpenCV to estimate the camera's pose. This function is used to estimate the object's pose using a set of points. You can read more about it at `http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#bool solvePnP(InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess, int flags)`. Once we do this, we need to project these points onto 2D. We use the OpenCV function `projectPoints` to do this. This function calculates the projections of those 3D points onto the 2D plane.

How to overlay 3D objects on a video?

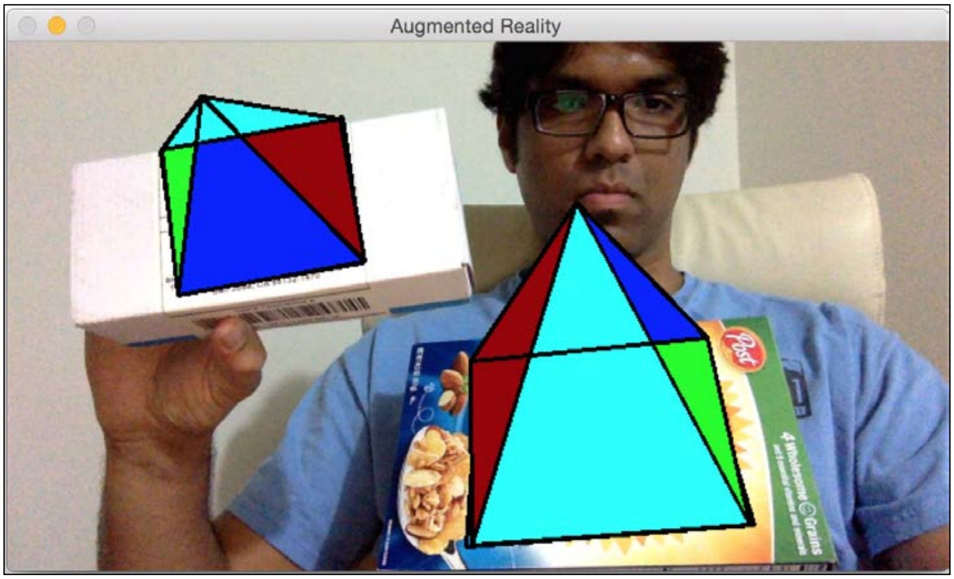
Now that we have all the different blocks, we are ready to build the final system. Let's say we want to overlay a pyramid on top of our cardboard as shown here:



Let's tilt the cardboard to see what happens:



Looks like the pyramid is following the surface. Let's add a second target:



You can keep adding more targets and all those pyramids will be tracked nicely. Let's see how to do this using OpenCV Python. Make sure to save the previous file as `pose_estimation.py` because we will be importing a couple of classes from there:

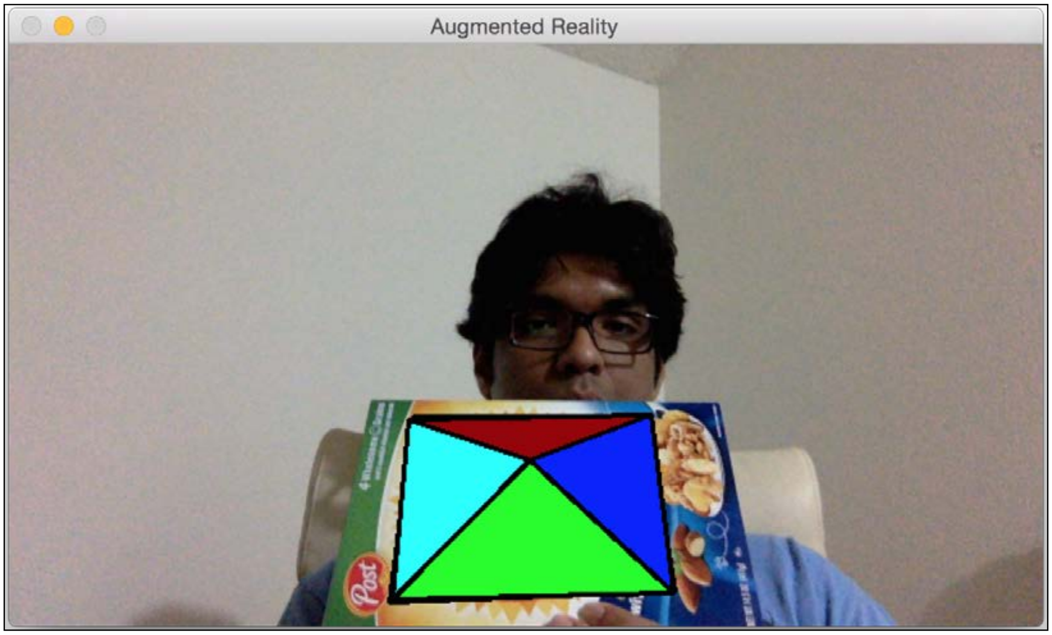
```
import cv2
import numpy as np

from pose_estimation import PoseEstimator, ROISelector

class Tracker(object):
    def __init__(self):
        self.cap = cv2.VideoCapture(0)
        self.frame = None
        self.paused = False
        self.tracker = PoseEstimator()

        cv2.namedWindow('Augmented Reality')
        self.roi_selector = ROISelector('Augmented Reality',
        self.on_rect)

        self.overlay_vertices = np.float32([[0, 0, 0], [0, 1, 0],
        [1, 1, 0], [1, 0, 0],
                               [0.5, 0.5, 4]])
        self.overlay_edges = [(0, 1), (1, 2), (2, 3), (3, 0),
                    (0,4), (1,4), (2,4), (3,4)]
        self.color_base = (0, 255, 0)
        self.color_lines = (0, 0, 0)

    def on_rect(self, rect):
        self.tracker.add_target(self.frame, rect)

    def start(self):
        while True:
            is_running = not self.paused and self.roi_selector.
selected_rect is None
            if is_running or self.frame is None:
                ret, frame = self.cap.read()
                scaling_factor = 0.5
                frame = cv2.resize(frame, None, fx=scaling_factor,
                fy=scaling_factor,
                        interpolation=cv2.INTER_AREA)
                if not ret:
                    break

                self.frame = frame.copy()

            img = self.frame.copy()
            if is_running:
                tracked = self.tracker.track_target(self.frame)
                for item in tracked:
                    cv2.polylines(img, [np.int32(item.quad)],
                    True, self.color_lines, 2)
                    for (x, y) in np.int32(item.points_cur):
                        cv2.circle(img, (x, y), 2,
                        self.color_lines)

                    self.overlay_graphics(img, item)

            self.roi_selector.draw_rect(img)
            cv2.imshow('Augmented Reality', img)
            ch = cv2.waitKey(1)
            if ch == ord(' '):
                self.paused = not self.paused
            if ch == ord('c'):
                self.tracker.clear_targets()
            if ch == 27:
                break
    def overlay_graphics(self, img, tracked):
        x_start, y_start, x_end, y_end = tracked.target.rect
        quad_3d = np.float32([[x_start, y_start, 0], [x_end,
        y_start, 0],
                    [x_end, y_end, 0], [x_start, y_end, 0]])
        h, w = img.shape[:2]
        K = np.float64([[w, 0, 0.5*(w-1)],
                        [0, w, 0.5*(h-1)],
                        [0, 0, 1.0]])
        dist_coef = np.zeros(4)
        ret, rvec, tvec = cv2.solvePnP(quad_3d, tracked.quad, K,
        dist_coef)
        verts = self.overlay_vertices * [(x_end-x_start),
        (y_end-y_start),
                    -(x_end-x_start)*0.3] + (x_start, y_start, 0)
        verts = cv2.projectPoints(verts, rvec, tvec, K,
        dist_coef)[0].reshape(-1, 2)

        verts_floor = np.int32(verts).reshape(-1,2)
        cv2.drawContours(img, [verts_floor[:4]], -1,
        self.color_base, -3)
        cv2.drawContours(img, [np.vstack((verts_floor[:2],
        verts_floor[4:5]))],
                    -1, (0,255,0), -3)
        cv2.drawContours(img, [np.vstack((verts_floor[1:3],
        verts_floor[4:5]))],
                    -1, (255,0,0), -3)
        cv2.drawContours(img, [np.vstack((verts_floor[2:4],
        verts_floor[4:5]))],
                    -1, (0,0,150), -3)
        cv2.drawContours(img, [np.vstack((verts_floor[3:4],
        verts_floor[0:1],
                    verts_floor[4:5]))], -1, (255,255,0), -3)

        for i, j in self.overlay_edges:
            (x_start, y_start), (x_end, y_end) = verts[i],
            verts[j]
            cv2.line(img, (int(x_start), int(y_start)),
            (int(x_end), int(y_end)), self.color_lines, 2)

if __name__ == '__main__':
    Tracker().start()
```

Let's look at the code

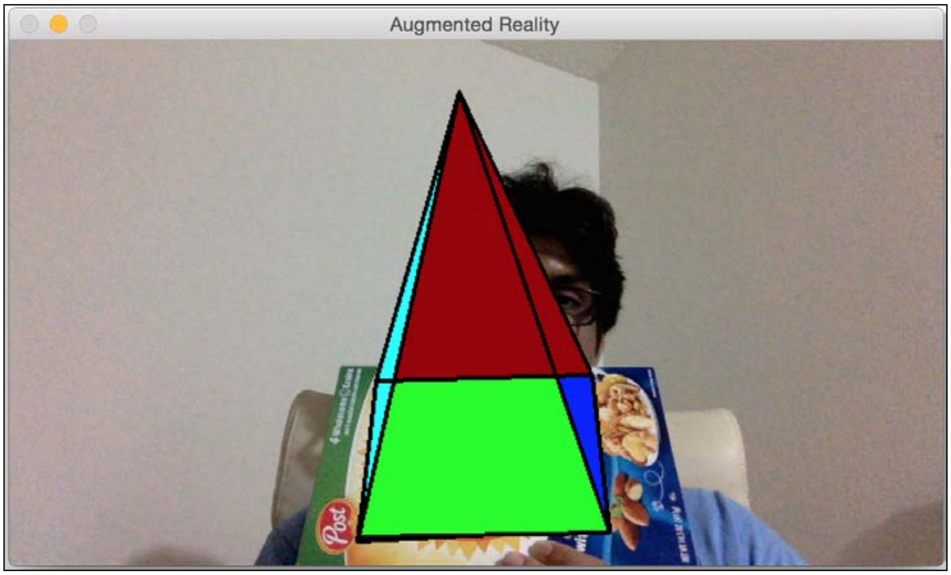
The class `Tracker` is used to perform all the computations here. We initialize the class with the pyramid structure that is defined using edges and vertices. The logic that we use to track the surface is the same as we discussed earlier because we are using the same class. We just need to use `solvePnP` and `projectPoints` to map the 3D pyramid to the 2D surface.

Let's add some movements

Now that we know how to add a virtual pyramid, let's see if we can add some movements. Let's see how we can dynamically change the height of the pyramid. When you start, the pyramid will look like this:



If you wait for some time, the pyramid gets taller and it will look like this:



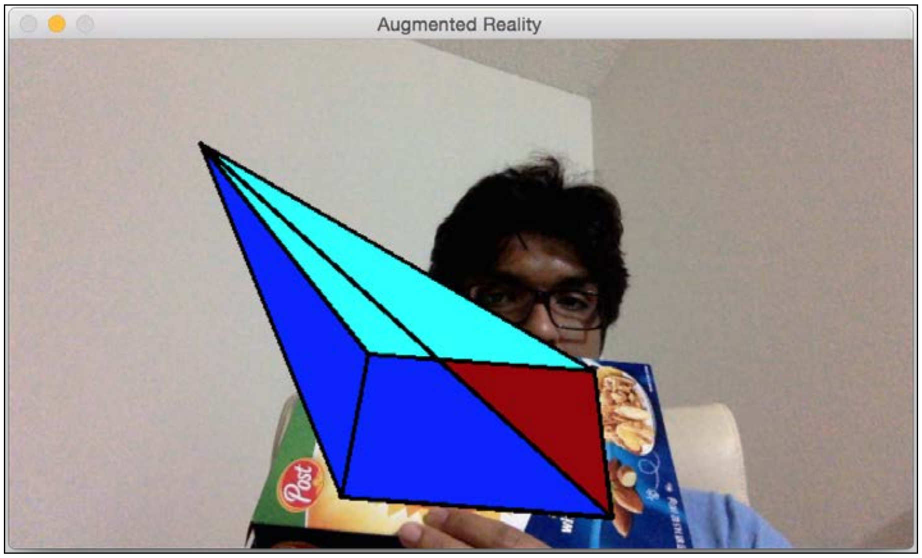
Let's see how to do it in OpenCV Python. Inside the augmented reality code that we just discussed, add the following snippet at the end of the `__init__` method in the `Tracker` class:

```
self.overlay_vertices = np.float32([[0, 0, 0], [0, 1, 0], [1, 1, 0],
[1, 0, 0], [0.5, 0.5, 4]])
self.overlay_edges = [(0, 1), (1, 2), (2, 3), (3, 0),
            (0,4), (1,4), (2,4), (3,4)]
self.color_base = (0, 255, 0)
self.color_lines = (0, 0, 0)

self.graphics_counter = 0
self.time_counter = 0
```

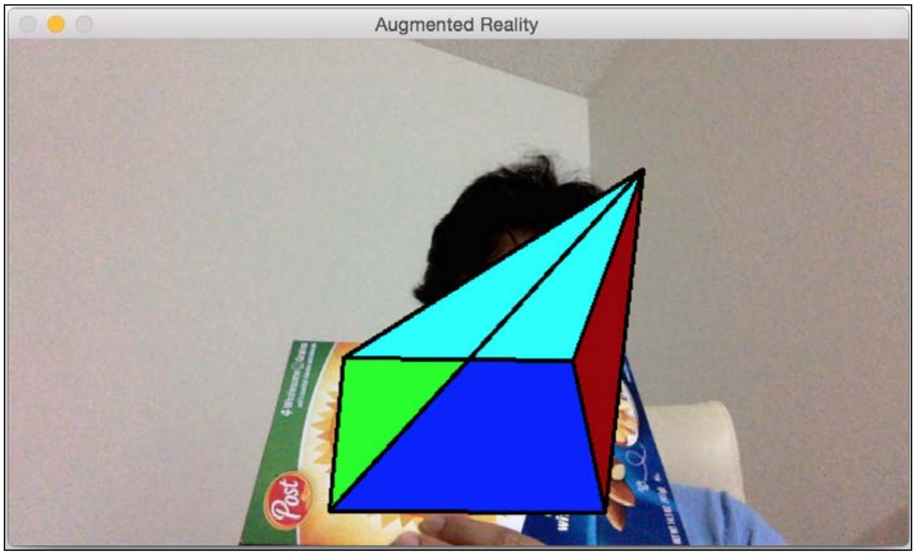
Now that we have the structure, we need to add the code to dynamically change the height. Replace the `overlay_graphics()` method with the following method:

```
def overlay_graphics(self, img, tracked):
    x_start, y_start, x_end, y_end = tracked.target.rect
    quad_3d = np.float32([[x_start, y_start, 0],
    [x_end, y_start, 0],
                [x_end, y_end, 0], [x_start, y_end, 0]])
    h, w = img.shape[:2]
    K = np.float64([[w, 0, 0.5*(w-1)],
                    [0, w, 0.5*(h-1)],
                    [0, 0, 1.0]])
    dist_coef = np.zeros(4)
    ret, rvec, tvec = cv2.solvePnP(quad_3d, tracked.quad, K,
    dist_coef)

    self.time_counter += 1
    if not self.time_counter % 20:
        self.graphics_counter = (self.graphics_counter + 1) % 8

    self.overlay_vertices = np.float32([[0, 0, 0], [0, 1, 0],
    [1, 1, 0], [1, 0, 0],
                           [0.5, 0.5, self.graphics_counter]])

    verts = self.overlay_vertices * [(x_end-x_start),
    (y_end-y_start),
                -(x_end-x_start)*0.3] + (x_start, y_start, 0)
    verts = cv2.projectPoints(verts, rvec, tvec, K,
    dist_coef)[0].reshape(-1, 2)

    verts_floor = np.int32(verts).reshape(-1,2)
    cv2.drawContours(img, [verts_floor[:4]], -1,
    self.color_base, -3)
    cv2.drawContours(img, [np.vstack((verts_floor[:2],
    verts_floor[4:5]))],
                -1, (0,255,0), -3)
    cv2.drawContours(img, [np.vstack((verts_floor[1:3],
    verts_floor[4:5]))],
                -1, (255,0,0), -3)
    cv2.drawContours(img, [np.vstack((verts_floor[2:4],
    verts_floor[4:5]))],
                -1, (0,0,150), -3)
    cv2.drawContours(img, [np.vstack((verts_floor[3:4],
    verts_floor[0:1],
                verts_floor[4:5]))], -1, (255,255,0), -3)

    for i, j in self.overlay_edges:
        (x_start, y_start), (x_end, y_end) = verts[i], verts[j]
        cv2.line(img, (int(x_start), int(y_start)), (int(x_end),
        int(y_end)), self.color_lines, 2)
```

Now that we know how to change the height, let's go ahead and make the pyramid dance for us. We can make the tip of the pyramid oscillate in a nice periodic fashion. So when you start, it will look like this:

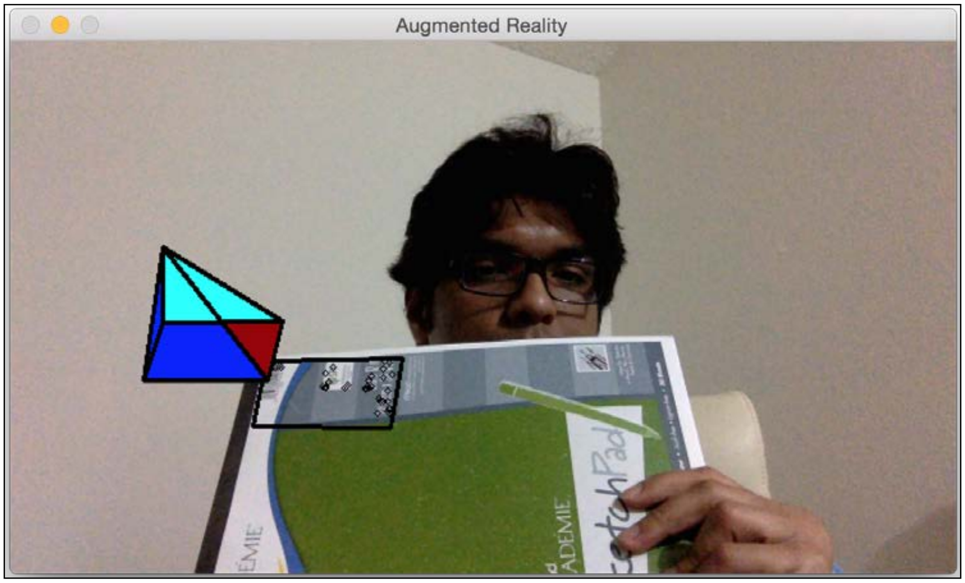


If you wait for some time, it will look like this:

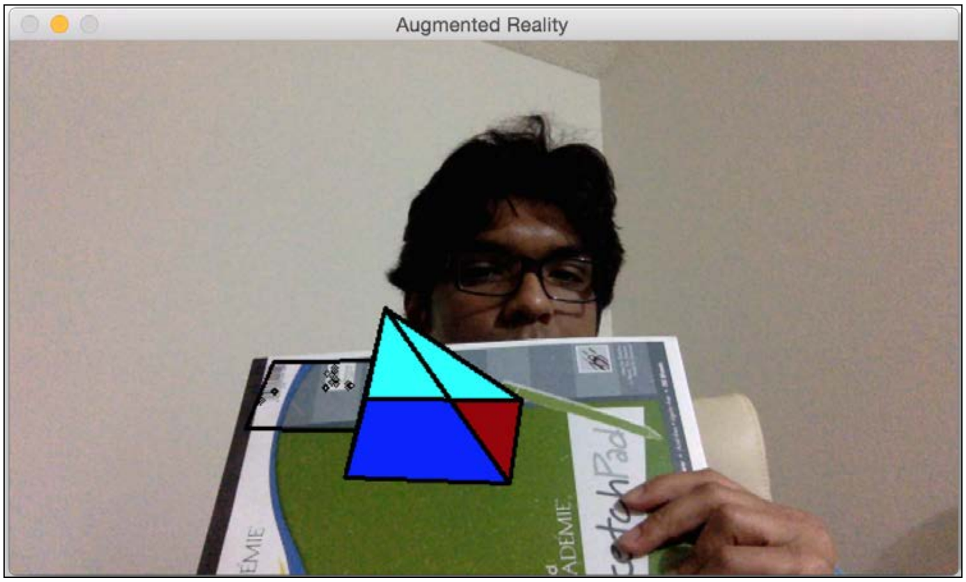


You can look at `augmented_reality_motion.py` for the implementation details.

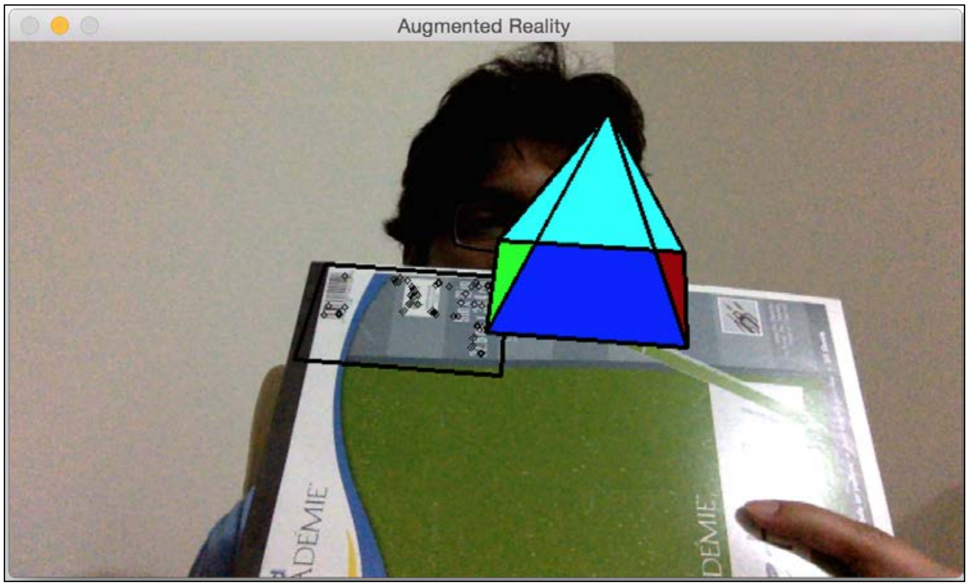
In our next experiment, we will make the whole pyramid move around the region of interest. We can make it move in any way we want. Let's start by adding linear diagonal movement around our selected region of interest. When you start, it will look like this:



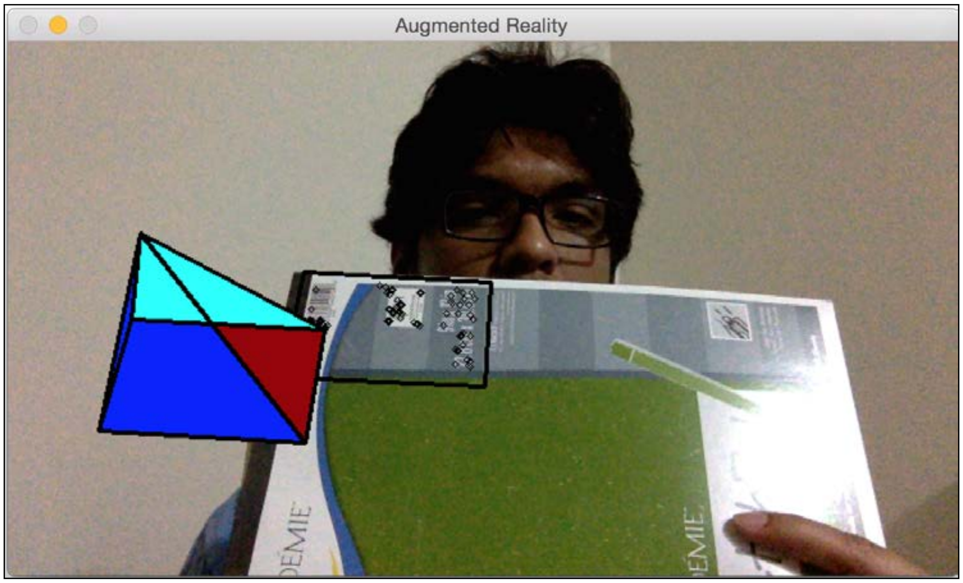
After some time, it will look like this:



Refer to augmented_reality_dancing.py to see how to change the overlay_graphics() method to make it dance. Let's see if we can make the pyramid go around in circles around our region of interest. When you start, it will look like this:



After some time, it will move to a new position:



You can refer to `augmented_reality_circular_motion.py` to see how to make this happen. You can make it do anything you want. You just need to come up with the right mathematical formula and the pyramid will literally dance to your tune! You can also try out other virtual objects to see what you can with it. There are a lot of things you can do with a lot of different objects. These examples provide a good reference point, on top of which you can build many interesting augmented reality applications.

Summary

In this chapter, you learned about the premise of augmented reality and understood what an augmented reality system looks like. We discussed the geometric transformations required for augmented reality. You learned how to use those transformations to estimate the camera pose. You learned how to track planar objects. We discussed how we can add virtual objects on top of the real world. You learned how to modify the virtual objects in different ways to add cool effects. Remember that the world of computer vision is filled with endless possibilities! This book is designed to teach you the necessary skills to get started on a wide variety of projects. Now it's up to you and your imagination to use the skills you have acquired here to build something unique and interesting.

Module 3

OpenCV with Python Blueprints

Design and develop advanced computer vision projects using OpenCV with Python

1
Fun with Filters

The goal of this chapter is to develop a number of image processing filters and apply them to the video stream of a webcam in real time. These filters will rely on various OpenCV functions to manipulate matrices through splitting, merging, arithmetic operations, and applying lookup tables for complex functions.

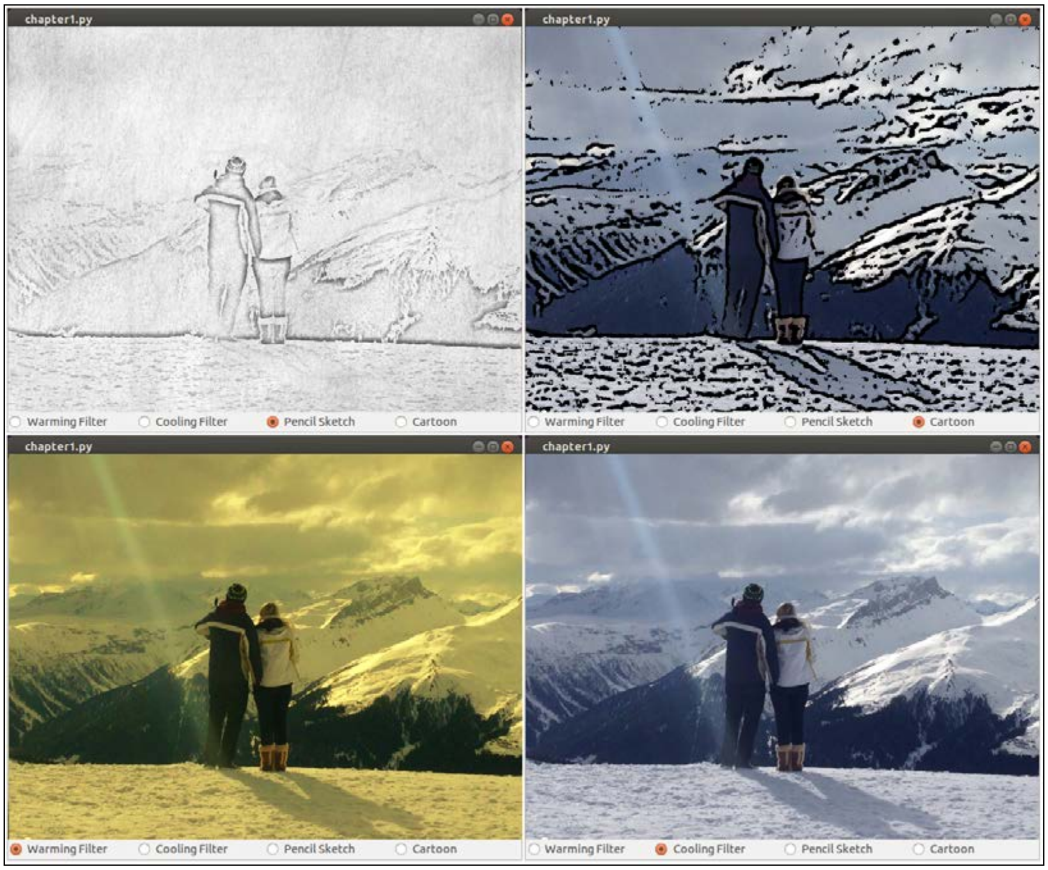
The three effects are as follows:

- **Black-and-white pencil sketch**: To create this effect, we will make use of two image blending techniques, known as **dodging** and **burning**
- **Warming/cooling filters**: To create these effects, we will implement our own **curve filters** using a lookup table
- **Cartoonizer**: To create this effect, we will combine a **bilateral filter**, a **median filter**, and **adaptive thresholding**

OpenCV is such an advanced toolchain that often the question is not how to implement something from scratch, but rather which pre-canned implementation to choose for your needs. Generating complex effects is not hard if you have a lot of computing resources to spare. The challenge usually lies in finding an approach that not only gets the job done, but also gets it done in time.

Instead of teaching the basic concepts of image manipulation through theoretical lessons, we will take a practical approach and develop a single end-to-end app that integrates a number of image filtering techniques. We will apply our theoretical knowledge to arrive at a solution that not only works but also speeds up seemingly complex effects so that a laptop can produce them in real time.

The following screenshot shows the final outcome of the three effects running on a laptop:



○ Warming Filter ○ Cooling Filter ● Pencil Sketch ○ Cartoon

○ Warming Filter ○ Cooling Filter ○ Pencil Sketch ● Cartoon

● Warming Filter ○ Cooling Filter ○ Pencil Sketch ○ Cartoon

○ Warming Filter ● Cooling Filter ○ Pencil Sketch ○ Cartoon

> All of the code in this book is targeted for OpenCV 2.4.9 and has been tested on Ubuntu 14.04. Throughout this book, we will make extensive use of the NumPy package (`http://www.numpy.org`). In addition, this chapter requires the `UnivariateSpline` module of the SciPy package (`http://www.scipy.org`) as well as the wxPython 2.8 graphical user interface (`http://www.wxpython.org/download.php`) for cross-platform GUI applications. We will try to avoid further dependencies wherever possible.

Planning the app

The final app will consist of the following modules and scripts:

- `filters`: A module comprising different classes for the three different image effects. The modular approach will allow us to use the filters independently of any **graphical user interface** (**GUI**).
- `filters.PencilSketch`: A class for applying the pencil sketch effect to an RGB color image.
- `filters.WarmingFilter`: A class for applying the warming filter to an RGB color image.
- `filters.CoolingFilter`: A class for applying the cooling filter to an RGB color image.
- `filters.Cartoonizer`: A method for applying the cartoonizer effect to an RGB color image.
- `gui`: A module that provides a wxPython GUI application to access the webcam and display the camera feed, which we will make extensive use of throughout the book.
- `gui.BaseLayout`: A generic layout from which more complicated layouts can be built.
- `chapter1`: The main script for this chapter.
- `chapter1.FilterLayout`: A custom layout based on `gui.BaseLayout` that displays the camera feed and a row of radio buttons that allows the user to select from the available image filters to be applied to each frame of the camera feed.
- `chapter1.main`: The main function routine for starting the GUI application and accessing the webcam.

Creating a black-and-white pencil sketch

In order to obtain a pencil sketch (that is, a black-and-white drawing) of the camera frame, we will make use of two image blending techniques, known as **dodging** and **burning**. These terms refer to techniques employed during the printing process in traditional photography; photographers would manipulate the exposure time of a certain area of a darkroom print in order to lighten or darken it. Dodging lightens an image, whereas burning darkens it.

Areas that were not supposed to undergo changes were protected with a **mask**. Today, modern image editing programs, such as Photoshop and Gimp, offer ways to mimic these effects in digital images. For example, masks are still used to mimic the effect of changing exposure time of an image, wherein areas of a mask with relatively intense values will *expose* the image more, thus lightening the image. OpenCV does not offer a native function to implement these techniques, but with a little insight and a few tricks, we will arrive at our own efficient implementation that can be used to produce a beautiful pencil sketch effect.

If you search on the Internet, you might stumble upon the following common procedure to achieve a pencil sketch from an RGB color image:

1. Convert the color image to grayscale.
2. Invert the grayscale image to get a negative.
3. Apply a Gaussian blur to the negative from step 2.
4. Blend the grayscale image from step 1 with the blurred negative from step 3 using a color dodge.

Whereas steps 1 to 3 are straightforward, step 4 can be a little tricky. Let's get that one out of the way first.

> OpenCV 3 comes with a pencil sketch effect right out of the box. The `cv2.pencilSketch` function uses a domain filter introduced in the 2011 paper *Domain transform for edge-aware image and video processing*, by Eduardo Gastal and Manuel Oliveira. However, for the purpose of this book, we will develop our own filter.

Implementing dodging and burning in OpenCV

In modern image editing tools, such as Photoshop, color dodging of an image `A` with a mask `B` is implemented as the following ternary statement acting on every pixel index, called `idx`:

```
((B[idx] == 255) ? B[idx] :
    min(255, ((A[idx] << 8) / (255-B[idx]))))
```

This essentially divides the value of an `A[idx]` image pixel by the inverse of the `B[idx]` mask pixel value, while making sure that the resulting pixel value will be in the range of [0, 255] and that we do not divide by zero.

We could translate this into the following naïve Python function, which accepts two OpenCV matrices (`image` and `mask`) and returns the blended image:

```
def dodgeNaive(image, mask):
    # determine the shape of the input image
    width,height = image.shape[:2]

    # prepare output argument with same size as image
    blend = np.zeros((width,height), np.uint8)

    for col in xrange(width):
        for row in xrange(height):

            # shift image pixel value by 8 bits
            # divide by the inverse of the mask
            tmp = (image[c,r] << 8) / (255.-mask)

            # make sure resulting value stays within bounds
            if tmp > 255:
                tmp = 255
            blend[c,r] = tmp
    return blend
```

As you might have guessed, although this code might be functionally correct, it will undoubtedly be horrendously slow. Firstly, the function uses `for` loops, which are almost always a bad idea in Python. Secondly, NumPy arrays (the underlying format of OpenCV images in Python) are optimized for array calculations, so accessing and modifying each `image[c,r]` pixel separately will be really slow.

Instead, we should realize that the `<<8` operation is the same as multiplying the pixel value with the number *2^8=256*, and that pixel-wise division can be achieved with the `cv2.divide` function. Thus, an improved version of our dodge function could look like this:

```
import cv2

def dodgeV2(image, mask):
    return cv2.divide(image, 255-mask, scale=256)
```

We have reduced the dodge function to a single line! The `dodgeV2` function produces the same result as `dodgeNaive` but is orders of magnitude faster. In addition, `cv2.divide` automatically takes care of division by zero, making the result `0` where `255-mask` is zero.

Now, it is straightforward to implement an analogous burning function, which divides the inverted image by the inverted mask and inverts the result:

```
import cv2

def burnV2(image, mask):
    return 255 – cv2.divide(255-image, 255-mask, scale=256)
```

Pencil sketch transformation

With these tricks in our bag, we are now ready to take a look at the entire procedure. The final code will be in its own class in the `filters` module. After we have converted a color image to grayscale, we aim to blend this image with its blurred negative:

1. We import the OpenCV and `numpy` modules:

   ```
   import cv2
   import numpy as np
   ```

2. Instantiate the `PencilSketch` class:

   ```
   class PencilSketch:
       def __init__(self, (width, height),
           bg_gray='pencilsketch_bg.jpg'):
   ```

   The constructor of this class will accept the image dimensions as well as an optional background image, which we will make use of in just a bit. If the file exists, we will open it and scale it to the right size:

   ```
   self.width = width
   self.height = height

   # try to open background canvas (if it exists)
   self.canvas = cv2.imread(bg_gray, cv2.CV_8UC1)
   if self.canvas is not None:
       self.canvas = cv2.resize(self.canvas,
           (self.width, self.height))
   ```

3. Add a render method that will perform the pencil sketch:

   ```
   def renderV2(self, img_rgb):
   ```

4. Converting an RGB image (`imgRGB`) to grayscale is straightforward:

   ```
   img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
   ```

   Note that it does not matter whether the input image is RGB or BGR.

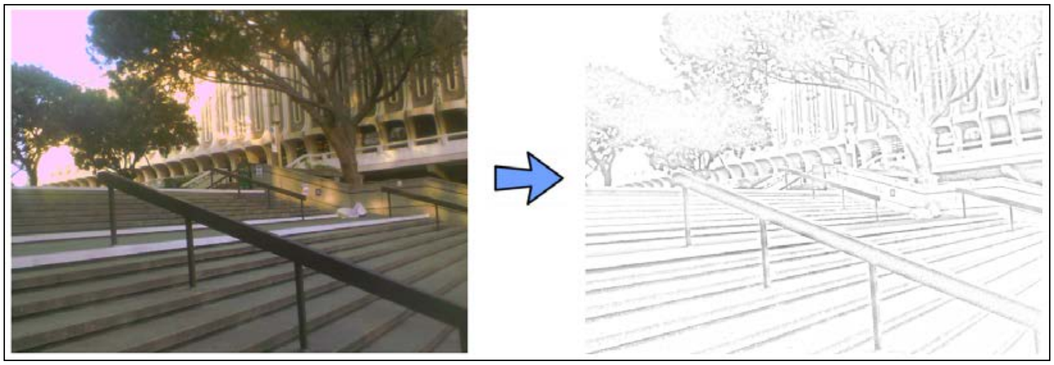
5. We then invert the image and blur it with a large Gaussian kernel of size `(21,21)`:

    ```
    img_gray_inv = 255 - img_gray
    img_blur = cv2.GaussianBlur(img_gray_inv, (21,21), 0, 0)
    ```

6. We use our `dodgeV2` dodging function from the aforementioned code to blend the original grayscale image with the blurred inverse:

    ```
    img_blend = dodgeV2(mg_gray, img_blur)
    return cv2.cvtColor(img_blend, cv2.COLOR_GRAY2RGB)
    ```

The resulting image looks like this:



Did you notice that our code can be optimized further?

A Gaussian blur is basically a convolution with a Gaussian function. One of the beauties of convolutions is their associative property. This means that it does not matter whether we first invert the image and then blur it, or first blur the image and then invert it.

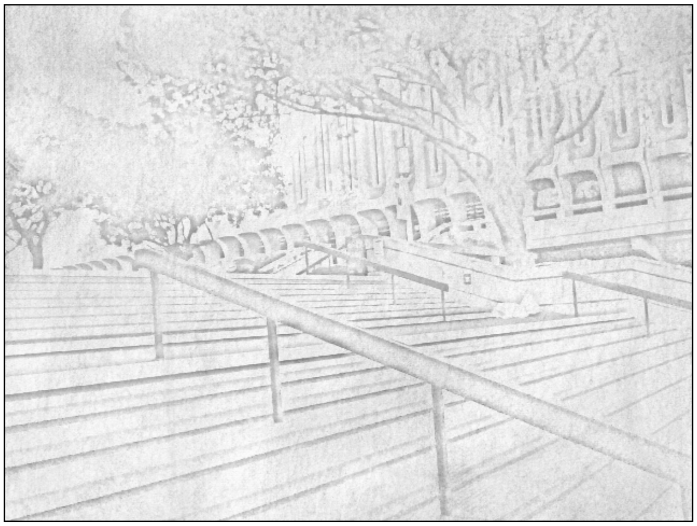
"Then what matters?" you might ask. Well, if we start with a blurred image and pass its inverse to the `dodgeV2` function, then within that function, the image will get inverted again (the `255-mask` part), essentially yielding the original image. If we get rid of these redundant operations, an optimized `render` method would look like this:

```
def render(img_rgb):
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
    img_blur = cv2.GaussianBlur(img_gray, (21,21), 0, 0)
    img_blend = cv2.divide(img_gray, img_blur, scale=256)
    return img_blend
```

For kicks and giggles, we want to lightly blend our transformed image (`img_blend`) with a background image (`self.canvas`) that makes it look as if we drew the image on a canvas:

```
if self.canvas is not None:
    img_blend = cv2.multiply(img_blend, self.canvas, scale=1./256)
return cv2.cvtColor(img_blend, cv2.COLOR_GRAY2BGR)
```

And we're done! The final output looks like what is shown here:



Generating a warming/cooling filter

When we perceive images, our brain picks up on a number of subtle clues to infer important details about the scene. For example, in broad daylight, highlights may have a slightly yellowish tint because they are in direct sunlight, whereas shadows may appear slightly bluish due to the ambient light of the blue sky. When we view an image with such color properties, we might immediately think of a *sunny day*.

This effect is no mystery to photographers, who sometimes purposely manipulate the white balance of an image to convey a certain mood. Warm colors are generally perceived as more pleasant, whereas cool colors are associated with night and drabness.

To manipulate the perceived **color temperature** of an image, we will implement a **curve filter**. These filters control how color transitions appear between different regions of an image, allowing us to subtly shift the color spectrum without adding an unnatural-looking overall tint to the image.

Color manipulation via curve shifting

A curve filter is essentially a function, *y = f(x)*, that maps an input pixel value *x* to an output pixel value *y*. The curve is parameterized by a set of *n+1* anchor points, as follows: *{(x_0,y_0), (x_1,y_1), ..., (x_n, y_n)}*.

Each anchor point is a pair of numbers that represent the input and output pixel values. For example, the pair *(30, 90)* means that an input pixel value of 30 is increased to an output value of 90. Values between anchor points are interpolated along a smooth curve (hence the name curve filter).

Such a filter can be applied to any image channel, be it a single grayscale channel or the R, G, and B channels of an RGB color image. Thus, for our purposes, all values of *x* and *y* must stay between 0 and 255.

For example, if we wanted to make a grayscale image slightly brighter, we could use a curve filter with the following set of control points: *{(0,0), (128, 192), (255,255)}*. This would mean that all input pixel values except 0 and 255 would be increased slightly, resulting in an overall brightening effect of the image.

If we want such filters to produce natural-looking images, it is important to respect the following two rules:

- Every set of anchor points should include *(0,0)* and *(255,255)*. This is important in order to prevent the image from appearing as if it has an overall tint, as black remains black and white remains white.
- The function *f(x)* should be monotonously increasing. In other words, with increasing *x*, *f(x)* either stays the same or increases (that is, it never decreases). This is important for making sure that shadows remain shadows and highlights remain highlights.

Implementing a curve filter by using lookup tables

Curve filters are computationally expensive, because the values of *f(x)* must be interpolated whenever *x* does not coincide with one of the prespecified anchor points. Performing this computation for every pixel of every image frame that we encounter would have dramatic effects on performance.

Instead, we make use of a lookup table. Since there are only 256 possible pixel values for our purposes, we need to calculate *f(x)* only for all the 256 possible values of *x*. Interpolation is handled by the `UnivariateSpline` function of the `scipy.interpolate` module, as shown in the following code snippet:

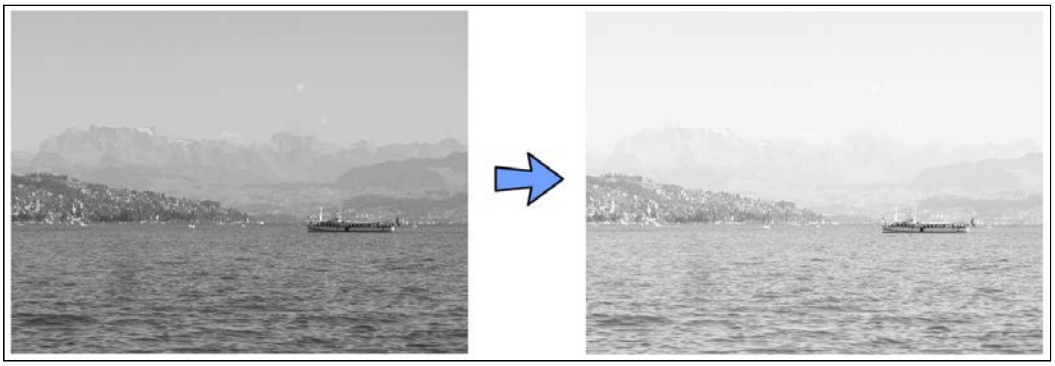
```
from scipy.interpolate import UnivariateSpline

def _create_LUT_8UC1(self, x, y):
  spl = UnivariateSpline(x, y)
  return spl(xrange(256))
```

The `return` argument of the function is a 256-element list that contains the interpolated *f(x)* values for every possible value of *x*.

All we need to do now is come up with a set of anchor points, *(x_i, y_i)*, and we are ready to apply the filter to a grayscale input image (`img_gray`):

```
import cv2
import numpy as np

x = [0, 128, 255]
y = [0, 192, 255]
myLUT = _create_LUT_8UC1(x, y)
img_curved = cv2.LUT(img_gray, myLUT).astype(np.uint8)
```

The result looks like this (the original image is on the left, and the transformed image is on the right):



Designing the warming/cooling effect

With the mechanism to quickly apply a generic curve filter to any image channel in place, we now turn to the question of how to manipulate the perceived color temperature of an image. Again, the final code will have its own class in the `filters` module.

If you have a minute to spare, I advise you to play around with the different curve settings for a while. You can choose any number of anchor points and apply the curve filter to any image channel you can think of (red, green, blue, hue, saturation, brightness, lightness, and so on). You could even combine multiple channels, or decrease one and shift another to a desired region. What will the result look like?

However, if the number of possibilities dazzles you, take a more conservative approach. First, by making use of our `_create_LUT_8UC1` function developed in the preceding steps, let's define two generic curve filters, one that (by trend) increases all pixel values of a channel, and one that generally decreases them:

```
class WarmingFilter:

  def __init__(self):
    self.incr_ch_lut = _create_LUT_8UC1([0, 64, 128, 192, 256],
      [0, 70, 140, 210, 256])
    self.decr_ch_lut = _create_LUT_8UC1([0, 64, 128, 192, 256],
      [0, 30,  80, 120, 192])
```

The easiest way to make an image appear as if it was taken on a hot, sunny day (maybe close to sunset), is to increase the reds in the image and make the colors appear vivid by increasing the color saturation. We will achieve this in two steps:

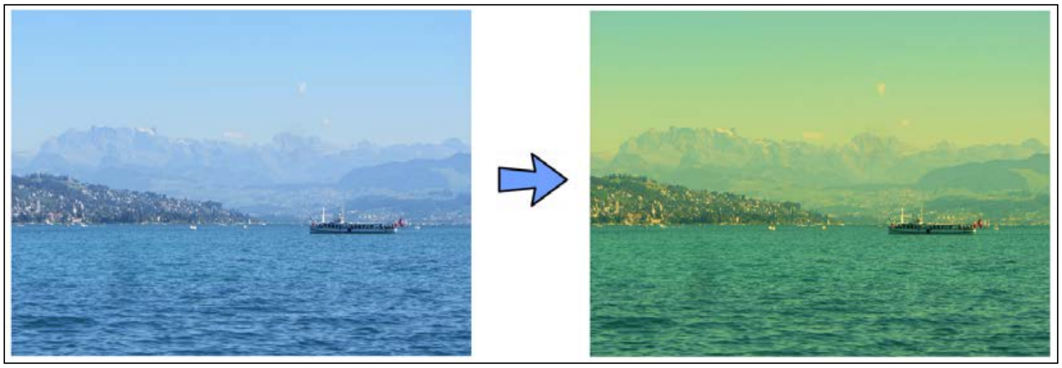
1. Increase the pixel values in the R channel and decrease the pixel values in the B channel of an RGB color image using `incr_ch_lut` and `decr_ch_lut`, respectively:

```
def render(self, img_rgb):
    c_r, c_g, c_b = cv2.split(img_rgb)
    c_r = cv2.LUT(c_r, self.incr_ch_lut).astype(np.uint8)
    c_b = cv2.LUT(c_b, self.decr_ch_lut).astype(np.uint8)
    img_rgb = cv2.merge((c_r, c_g, c_b))
```

2. Transform the image into the **HSV** color space (**H** means hue, **S** means saturation, and **V** means value), and increase the S channel using `incr_ch_lut`. This can be achieved with the following function, which expects an RGB color image as input:

```
c_b = cv2.LUT(c_b, decrChLUT).astype(np.uint8)

# increase color saturation
c_h, c_s, c_v = cv2.split(cv2.cvtColor(img_rgb,
    cv2.COLOR_RGB2HSV))
c_s = cv2.LUT(c_s, self.incr_ch_lut).astype(np.uint8)
return cv2.cvtColor(cv2.merge((c_h, c_s, c_v)),
    cv2.COLOR_HSV2RGB)
```

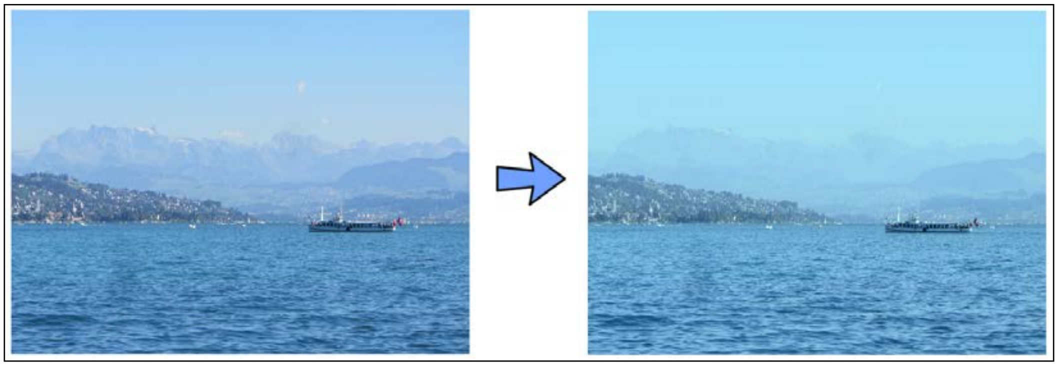
The result looks like what is shown here:



Analogously, we can define a cooling filter that increases the pixel values in the B channel, decreases the pixel values in the R channel of an RGB image, converts the image into the HSV color space, and decreases color saturation via the S channel:

```
class CoolingFilter:
    def render(self, img_rgb):
        c_r, c_g, c_b = cv2.split(img_rgb)
        c_r = cv2.LUT(c_r, self.decr_ch_lut).astype(np.uint8)
        c_b = cv2.LUT(c_b, self.incr_ch_lut).astype(np.uint8)
        img_rgb = cv2.merge((c_r, c_g, c_b))
        # decrease color saturation
        c_h, c_s, c_v = cv2.split(cv2.cvtColor(img_rgb,
            cv2.COLOR_RGB2HSV))
        c_s = cv2.LUT(c_s, self.decr_ch_lut).astype(np.uint8)
        return cv2.cvtColor(cv2.merge((c_h, c_s, c_v)),
            cv2.COLOR_HSV2RGB)
```

Now, the result looks like this:



Cartoonizing an image

Over the past few years, professional cartoonizer software has popped up all over the place. In order to achieve the basic cartoon effect, all that we need is a **bilateral filter** and some **edge detection**. The bilateral filter will reduce the color palette, or the numbers of colors that are used in the image. This mimics a cartoon drawing, wherein a cartoonist typically has few colors to work with. Then we can apply edge detection to the resulting image to generate bold silhouettes. The real challenge, however, lies in the computational cost of bilateral filters. We will thus use some tricks to produce an acceptable cartoon effect in real time.

We will adhere to the following procedure to transform an RGB color image into a cartoon:

1. Apply a bilateral filter to reduce the color palette of the image.
2. Convert the original color image into grayscale.
3. Apply a **median blur** to reduce image noise.
4. Use **adaptive thresholding** to detect and emphasize the edges in an edge mask.
5. Combine the color image from step 1 with the edge mask from step 4.

Using a bilateral filter for edge-aware smoothing

A strong bilateral filter is ideally suitable for converting an RGB image into a color painting or a cartoon, because it smoothens flat regions while keeping edges sharp. It seems that the only drawback of this filter is its computational cost, as it is orders of magnitude slower than other smoothing operations, such as a Gaussian blur.

The first measure to take when we need to reduce the computational cost is to perform an operation on an image of low resolution. In order to downscale an RGB image (`imgRGB`) to a quarter of its size (reduce the width and height to half), we could use `cv2.resize`:

```
import cv2

img_small = cv2.resize(img_rgb, (0,0), fx=0.5, fy=0.5)
```

A pixel value in the resized image will correspond to the pixel average of a small neighborhood in the original image. However, this process may produce image artifacts, which is also known as aliasing. While this is bad enough on its own, the effect might be enhanced by subsequent processing, for example, edge detection.

A better alternative might be to use the **Gaussian pyramid** for downscaling (again to a quarter of the original size). The Gaussian pyramid consists of a blur operation that is performed before the image is resampled, which reduces aliasing effects:

```
img_small = cv2.pyrDown(img_rgb)
```

However, even at this scale, the bilateral filter might still be too slow to run in real time. Another trick is to repeatedly (say, five times) apply a small bilateral filter to the image instead of applying a large bilateral filter once:

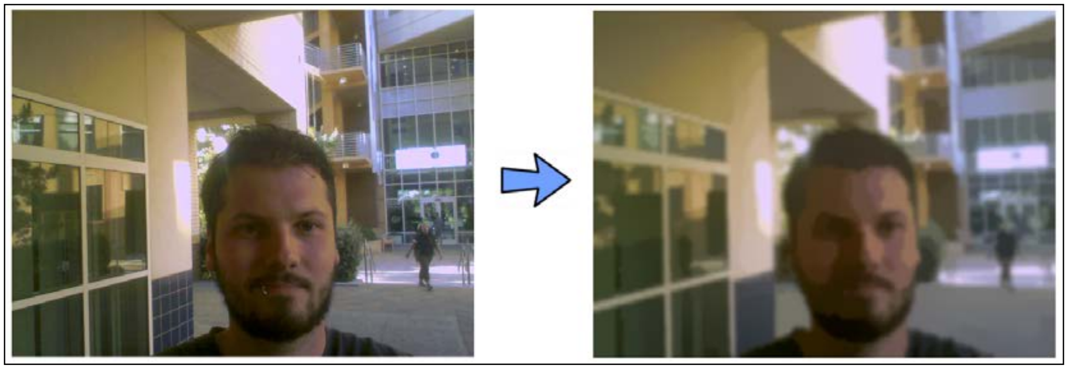
```
num_iter = 5
for _ in xrange(num_iter):
    img_small = cv2.bilateralFilter(img_small, d=9, sigmaColor=9,
        sigmaSpace=7)
```

The three parameters in `cv2.bilateralFilter` control the diameter of the pixel neighborhood (`d`) and the standard deviation of the filter in the color space (`sigmaColor`) and coordinate space (`sigmaSpace`).

Don't forget to restore the image to its original size:

```
img_rgb = cv2.pyrUp(img_small)
```

The result looks like a blurred color painting of a creepy programmer, as follows:



Detecting and emphasizing prominent edges

Again, when it comes to edge detection, the challenge often does not lie in how the underlying algorithm works, but instead which particular algorithm to choose for the task at hand. You might already be familiar with a variety of edge detectors. For example, **Canny edge detection** (`cv2.Canny`) provides a relatively simple and effective method to detect edges in an image, but it is susceptible to noise.

The **Sobel** operator (`cv2.Sobel`) can reduce such artifacts, but it is not rotationally symmetric. The **Scharr** operator (`cv2.Scharr`) was targeted at correcting this, but only looks at the first image derivative. If you are interested, there are even more operators for you, such as the **Laplacian** or **ridge** operator (which includes the second derivative), but they are far more complex. And in the end, for our specific purposes, they might not look better, maybe because they are as susceptible to lighting conditions as any other algorithm.

For the purpose of this project, we will choose a function that might not even be associated with conventional edge detection—`cv2.adaptiveThreshold`. Like `cv2.threshold`, this function uses a threshold pixel value to convert a grayscale image into a binary image. That is, if a pixel value in the original image is above the threshold, then the pixel value in the final image will be 255. Otherwise, it will be 0. However, the beauty of adaptive thresholding is that it does not look at the overall properties of the image. Instead, it detects the most salient features in each small neighborhood independently, without regard to the global image optima. This makes the algorithm extremely robust to lighting conditions, which is exactly what we want when we seek to draw bold, black outlines around objects and people in a cartoon.

However, it also makes the algorithm susceptible to noise. To counteract this, we will preprocess the image with a median filter. A median filter does what its name suggests; it replaces each pixel value with the median value of all the pixels in a small pixel neighborhood. We first convert the RGB image (`img_rgb`) to grayscale (`img_gray`) and then apply a median blur with a seven-pixel local neighborhood:

```
# convert to grayscale and apply median blur
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
img_blur = cv2.medianBlur(img_gray, 7)
```

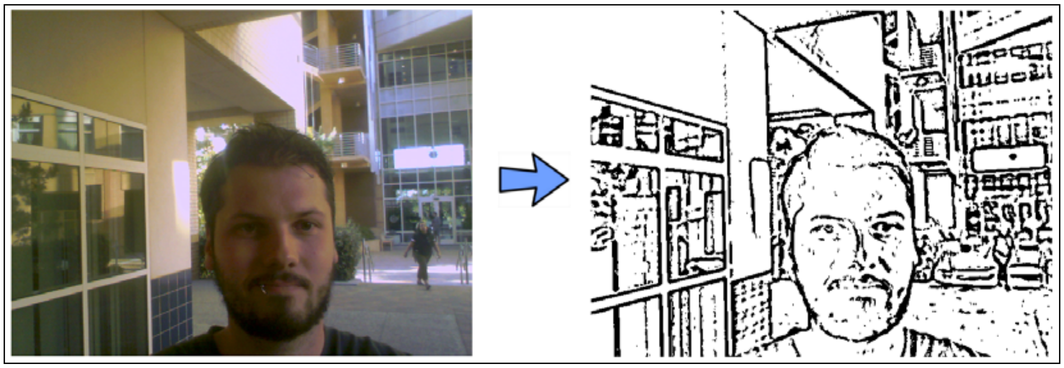
After reducing the noise, it is now safe to detect and enhance the edges using adaptive thresholding. Even if there is some image noise left, the `cv2.ADAPTIVE_THRESH_MEAN_C` algorithm with `blockSize=9` will ensure that the threshold is applied to the mean of a 9 x 9 neighborhood minus `C=2`:

```
img_edge = cv2.adaptiveThreshold(img_blur, 255,
                                 cv2.ADAPTIVE_THRESH_MEAN_C,
                                 cv2.THRESH_BINARY, 9, 2)
```

Downloading the example code

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

The result of the adaptive thresholding looks like this:



Combining colors and outlines to produce a cartoon

The last step is to combine the two. Simply fuse the two effects together into a single image using `cv2.bitwise_and`. The complete function is as follows:

```
def render(self, img_rgb):
    numDownSamples = 2 # number of downscaling steps
    numBilateralFilters = 7  # number of bilateral filtering steps

    # -- STEP 1 --
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in xrange(numDownSamples):
        img_color = cv2.pyrDown(img_color)

    # repeatedly apply small bilateral filter instead of applying
    # one large filter
    for _ in xrange(numBilateralFilters):
        img_color = cv2.bilateralFilter(img_color, 9, 9, 7)

    # upsample image to original size
    for _ in xrange(numDownSamples):
        img_color = cv2.pyrUp(img_color)

    # -- STEPS 2 and 3 --
    # convert to grayscale and apply median blur
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
```

```
        # -- STEP 4 --
        # detect and enhance edges
        img_edge = cv2.adaptiveThreshold(img_blur, 255,
            cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 2)

        # -- STEP 5 --
        # convert back to color so that it can be bit-ANDed
        # with color image
        img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
        return cv2.bitwise_and(img_color, img_edge)
```

The result looks like what is shown here:

Putting it all together

Before we can make use of the designed image filter effects in an interactive way, we need to set up the main script and design a GUI application.

Running the app

To run the application, we will turn to the `chapter1.py.` script, which we will start by importing all the necessary modules:

```
import numpy as np

import wx
import cv2
```

We will also have to import a generic GUI layout (from `gui`) and all the designed image effects (from `filters`):

```
from gui import BaseLayout
from filters import PencilSketch, WarmingFilter, CoolingFilter,
    Cartoonizer
```

OpenCV provides a straightforward way to access a computer's webcam or camera device. The following code snippet opens the default camera ID (`0`) of a computer using `cv2.VideoCapture`:

```
def main():
    capture = cv2.VideoCapture(0)
```

On some platforms, the first call to `cv2.VideoCapture` fails to open a channel. In that case, we provide a workaround by opening the channel ourselves:

```
if not(capture.isOpened()):
    capture.open()
```

In order to give our application a fair chance to run in real time, we will limit the size of the video stream to 640 x 480 pixels:

```
capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)
```

> If you are using OpenCV 3, the constants that you are looking for might be called `cv3.CAP_PROP_FRAME_WIDTH` and `cv3.CAP_PROP_FRAME_HEIGHT`.

Then the `capture` stream can be passed to our GUI application, which is an instance of the `FilterLayout` class:

```
    # start graphical user interface
    app = wx.App()
    layout = FilterLayout(None, -1, 'Fun with Filters', capture)
    layout.Show(True)
    app.MainLoop()
```

The only thing left to do now is design the said GUI.

The GUI base class

The `FilterLayout` GUI will be based on a generic, plain layout class called `BaseLayout`, which we will be able to use in subsequent chapters as well.

The BaseLayout class is designed as an **abstract base class**. You can think of this class as a blueprint or recipe that will apply to all the layouts that we are yet to design—a skeleton class, if you will, that will serve as the backbone for all of our future GUI code. In order to use abstract classes, we need the following import statement:

```
from abc import ABCMeta, abstractmethod
```

We also include some other modules that will be helpful, especially the wx Python module and OpenCV (of course):

```
import time

import wx
import cv2
```

The class is designed to be derived from the blueprint or skeleton, that is, the wx.Frame class. We also mark the class as abstract by adding the __metaclass__ attribute:

```
class BaseLayout(wx.Frame):
    __metaclass__ = ABCMeta
```

Later on, when we write our own custom layout (FilterLayout), we will use the same notation to specify that the class is based on the BaseLayout blueprint (or skeleton) class, for example, in class FilterLayout(BaseLayout):. But for now, let's focus on the BaseLayout class.

An abstract class has at least one abstract method. An abstract method is akin to specifying that a certain method must exist, but we are not sure at that time what it should look like. For example, suppose BaseLayout contains a method specified as follows:

```
@abstractmethod
def _init_custom_layout(self):
    pass
```

Then any class deriving from it, such as FilterLayout, must specify a fully fleshed-out implementation of a method with that exact signature. This will allow us to create custom layouts, as you will see in a moment.

But first, let's proceed to the GUI constructor.

The GUI constructor

The `BaseLayout` constructor accepts an ID (`-1`), a title string ('`Fun with Filters`'), a video capture object, and an optional argument that specifies the number of frames per second. Then, the first thing to do in the constructor is try and read a frame from the captured object in order to determine the image size:

```
def __init__(self, parent, id, title, capture, fps=10):
    self.capture = capture
    # determine window size and init wx.Frame
    _, frame = self.capture.read()
    self.imgHeight,self.imgWidth = frame.shape[:2]
```

We will use the image size to prepare a buffer that will store each video frame as a bitmap, and to set the size of the GUI. Because we want to display a bunch of control buttons below the current video frame, we set the height of the GUI to `self.imgHeight+20`:

```
self.bmp = wx.BitmapFromBuffer(self.imgWidth,
    self.imgHeight, frame)
wx.Frame.__init__(self, parent, id, title,
        size=(self.imgWidth, self.imgHeight+20))
```

We then provide two methods to initialize some more parameters and create the actual layout of the GUI:

```
self._init_base_layout()
self._create_base_layout()
```

Handling video streams

The video stream of the webcam is handled by a series of steps that begin with the `_init_base_layout` method. These steps might appear overly complicated at first, but they are necessary in order to allow the video to run smoothly, even at higher frame rates (that is, to counteract flicker).

The `wxPython` module works with events and callback methods. When a certain event is triggered, it can cause a certain class method to be executed (in other words, a method can *bind* to an event). We will use this mechanism to our advantage and display a new frame every so often using the following steps:

1. We create a timer that will generate a `wx.EVT_TIMER` event whenever 1000./fps milliseconds have passed:

   ```
   def _init_base_layout(self):
       self.timer = wx.Timer(self)
       self.timer.Start(1000./self.fps)
   ```

2. Whenever the timer is up, we want the `_on_next_frame` method to be called. It will try to acquire a new video frame:

   ```
   self.Bind(wx.EVT_TIMER, self._on_next_frame)
   ```

3. The `_on_next_frame` method will process the new video frame and store the processed frame in a bitmap. This will trigger another event, `wx.EVT_PAINT`. We want to bind this event to the `_on_paint` method, which will paint the display the new frame:

   ```
   self.Bind(wx.EVT_PAINT, self._on_paint)
   ```

The `_on_next_frame` method grabs a new frame and, once done, sends the frame to another method, `__process_frame`, for further processing:

```
def _on_next_frame(self, event):
    ret, frame = self.capture.read()
    if ret:
        frame = self._process_frame(cv2.cvtColor(frame,
            cv2.COLOR_BGR2RGB))
```

The processed frame (`frame`) is then stored in a bitmap buffer (`self.bmp`):

```
self.bmp.CopyFromBuffer(frame)
```

Calling `Refresh` triggers the aforementioned `wx.EVT_PAINT` event, which binds to `_on_paint`:

```
self.Refresh(eraseBackground=False)
```

The paint method then grabs the frame from the buffer and displays it:

```
def _on_paint(self, event):
    deviceContext = wx.BufferedPaintDC(self.pnl)
    deviceContext.DrawBitmap(self.bmp, 0, 0)
```

A basic GUI layout

The creation of the generic layout is done by a method called `_create_base_layout`. The most basic layout consists of only a large black panel that provides enough room to display the video feed:

```
def _create_base_layout(self):
    self.pnl = wx.Panel(self, -1,
                        size=(self.imgWidth, self.imgHeight))
    self.pnl.SetBackgroundColour(wx.BLACK)
```

In order for the layout to be extendable, we add it to a vertically arranged `wx.BoxSizer` object:

```
self.panels_vertical = wx.BoxSizer(wx.VERTICAL)
self.panels_vertical.Add(self.pnl, 1, flag=wx.EXPAND)
```

Next, we specify an abstract method, `_create_custom_layout`, for which we will not fill in any code. Instead, any user of our base class can make their own custom modifications to the basic layout:

```
self._create_custom_layout()
```

Then, we just need to set the minimum size of the resulting layout and center it:

```
self.SetMinSize((self.imgWidth, self.imgHeight))
self.SetSizer(self.panels_vertical)
self.Centre()
```

A custom filter layout

Now we are almost done! If we want to use the `BaseLayout` class, we need to provide code for the three methods that were left blank previously:

- `_init_custom_layout`: This is where we can initialize task-specific parameters
- `_create_custom_layout`: This is where we can make task-specific modifications to the GUI layout
- `_process_frame`: This is where we perform task-specific processing on each captured frame of the camera feed

At this point, initializing the image filters is self-explanatory, as it only requires us to instantiate the corresponding classes:

```
def _init_custom_layout(self):
    self.pencil_sketch = PencilSketch((self.imgWidth,
        self.imgHeight))
    self.warm_filter = WarmingFilter()
    self.cool_filter = CoolingFilter()
    self.cartoonizer = Cartoonizer()
```

To customize the layout, we arrange a number of radio buttons horizontally, one button per image effect mode:

```
def _create_custom_layout(self):
    # create a horizontal layout with all filter modes
    pnl = wx.Panel(self, -1 )
```

```
        self.mode_warm = wx.RadioButton(pnl, -1, 'Warming Filter',
            (10, 10), style=wx.RB_GROUP)
        self.mode_cool = wx.RadioButton(pnl, -1, 'Cooling Filter',
            (10, 10))
        self.mode_sketch = wx.RadioButton(pnl, -1, 'Pencil Sketch',
            (10, 10))
        self.mode_cartoon = wx.RadioButton(pnl, -1, 'Cartoon',
            (10, 10))
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add(self.mode_warm, 1)
        hbox.Add(self.mode_cool, 1)
        hbox.Add(self.mode_sketch, 1)
        hbox.Add(self.mode_cartoon, 1)
        pnl.SetSizer(hbox)
```

Here, the `style=wx.RB_GROUP` option makes sure that only one of these radio buttons can be selected at a time.

To make these changes take effect, `pnl` needs to be added to list of existing panels:

```
    self.panels_vertical.Add(pnl, flag=wx.EXPAND | wx.BOTTOM | wx.TOP,
                border=1)
```

The last method to be specified is `_process_frame`. Recall that this method is triggered whenever a new camera frame is received. All that we need to do is pick the right image effect to be applied, which depends on the radio button configuration. We simply check which of the buttons is currently selected and call the corresponding render method:

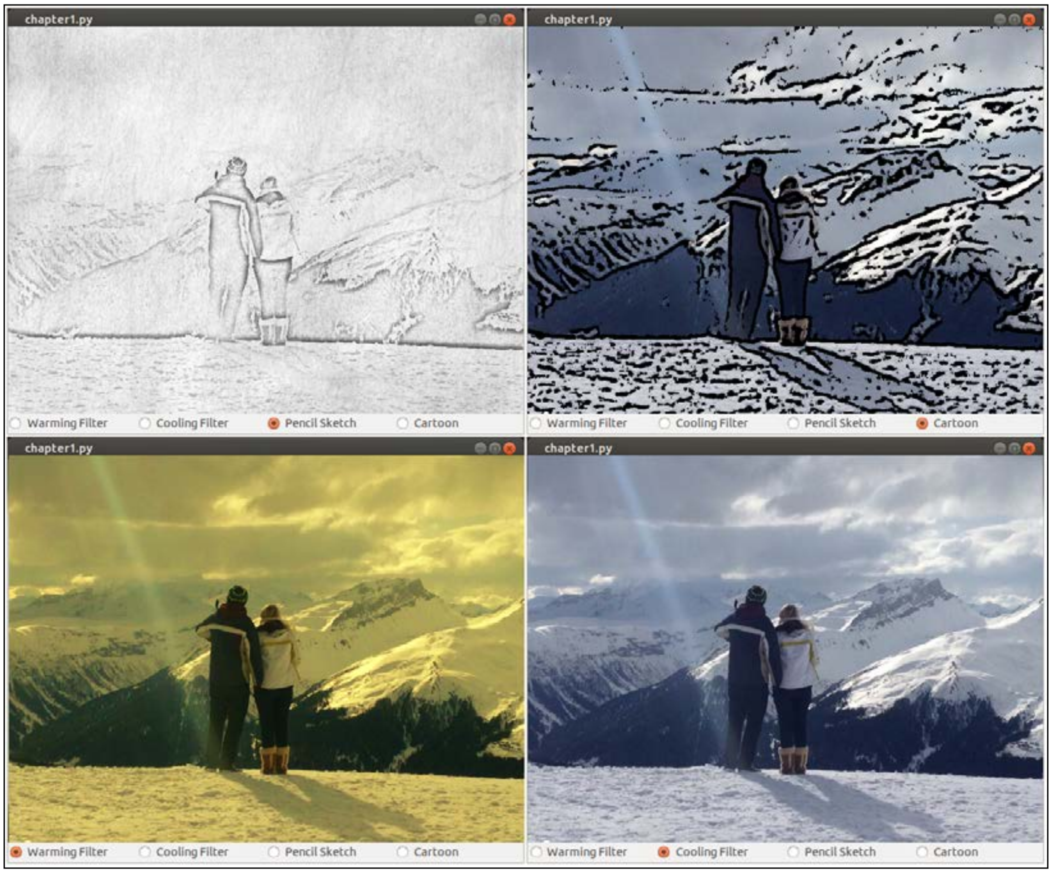
```
    def _process_frame(self, frame_rgb):
        if self.mode_warm.GetValue():
            frame = self.warm_filter.render(frame_rgb)
        elif self.mode_cool.GetValue():
            frame = self.cool_filter.render(frame_rgb)
        elif self.mode_sketch.GetValue():
            frame = self.pencil_sketch.render(frame_rgb)
        elif self.mode_cartoon.GetValue():
            frame = self.cartoonizer.render(frame_rgb)
```

Don't forget to return the processed frame:

```
    return frame
```

And we're done!

Here is the result:


chapter1.py
○ Warming Filter ○ Cooling Filter ● Pencil Sketch ○ Cartoon
chapter1.py
○ Warming Filter ○ Cooling Filter ○ Pencil Sketch ● Cartoon
chapter1.py
● Warming Filter ○ Cooling Filter ○ Pencil Sketch ○ Cartoon
chapter1.py
○ Warming Filter ● Cooling Filter ○ Pencil Sketch ○ Cartoon

Summary

In this chapter, we explored a number of interesting image processing effects. We used dodging and burning to create a black-and-white pencil sketch effect, explored lookup tables to arrive at an efficient implementation of curve filters, and got creative to produce a cartoon effect.

In the next chapter, we will shift gears a bit and explore the use of depth sensors, such as Microsoft Kinect 3D, to recognize hand gestures in real time.

2

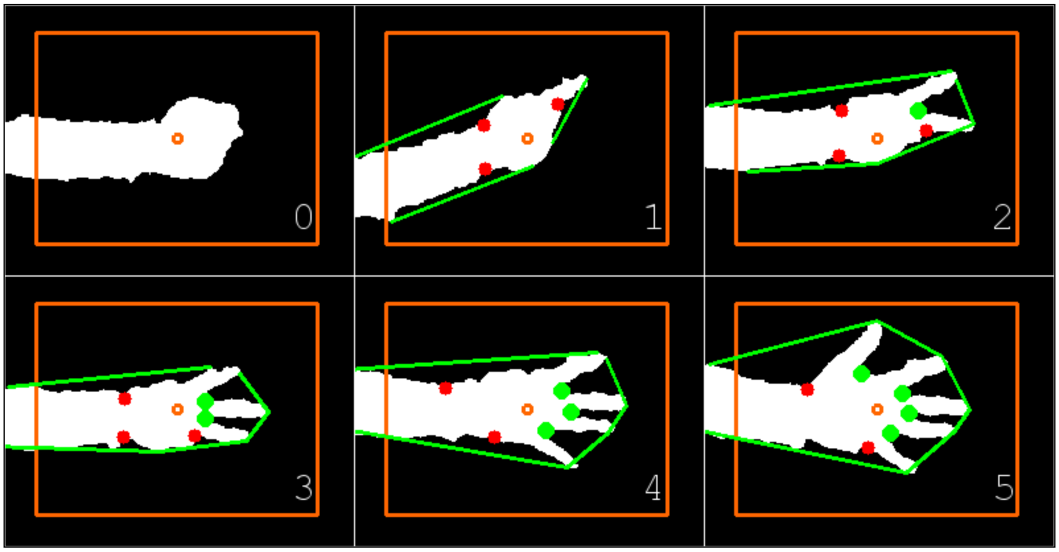
Hand Gesture Recognition Using a Kinect Depth Sensor

The goal of this chapter is to develop an app that detects and tracks simple hand gestures in real time using the output of a depth sensor, such as that of a Microsoft Kinect 3D sensor or an Asus Xtion. The app will analyze each captured frame to perform the following tasks:

- **Hand region segmentation**: The user's hand region will be extracted in each frame by analyzing the **depth map** output of the Kinect sensor, which is done by **thresholding**, applying some **morphological operations**, and finding **connected components**
- **Hand shape analysis**: The shape of the segmented hand region will be analyzed by determining **contours**, **convex hull**, and **convexity defects**
- **Hand gesture recognition**: The number of extended fingers will be determined based on the hand contour's **convexity defects**, and the gesture will be classified accordingly (with no extended fingers corresponding to a fist, and five extended fingers corresponding to an open hand)

Gesture recognition is an ever-popular topic in computer science. This is because it not only enables humans to communicate with machines (human-machine interaction or HMI), but also constitutes the first step for machines to begin understanding human body language. With affordable sensors such as Microsoft Kinect or Asus Xtion, and open source software such as OpenKinect and OpenNI, it has never been easy to get started in the field yourself. So, what shall we do with all this technology?

The beauty of the algorithm that we are going to implement in this chapter is that it works well for a number of hand gestures, yet is simple enough to run in real time on a generic laptop. Also, if we want, we can easily extend it to incorporate more complicated hand pose estimations. The end product looks like this:



No matter how many fingers of my left hand I extend, the algorithm correctly segments the hand region (white), draws the corresponding convex hull (the green line surrounding the hand), finds all convexity defects that belong to the spaces between fingers (large green points) while ignoring others (small red points), and infers the correct number of extended fingers (the number in the bottom-right corner), even for a fist.

> This chapter assumes that you have a Microsoft Kinect 3D sensor installed. Alternatively, you may install an Asus Xtion or any other depth sensor for which OpenCV has built-in support. First, install OpenKinect and libfreenect from `http://www.openkinect.org/wiki/Getting_Started`. Then, you need to build (or rebuild) OpenCV with OpenNI support. The GUI used in this chapter will again be designed with wxPython, which can be obtained from `http://www.wxpython.org/download.php`.

Planning the app

The final app will consist of the following modules and scripts:

- `gestures`: A module that consists of an algorithm for recognizing hand gestures. We separate this algorithm from the rest of the application so that it can be used as a standalone module without the need for a GUI.
- `gestures.HandGestureRecognition`: A class that implements the entire process flow of hand-gesture recognition. It accepts a single-channel depth image (acquired from the Kinect depth sensor) and returns an annotated RGB color image with an estimated number of extended fingers.
- `gui`: A module that provides a wxPython GUI application to access the capture device and display the video feed. This is the same module that we used in the last chapter. In order to have it access the Kinect depth sensor instead of a generic camera, we will have to extend some of the base class functionality.
- `gui.BaseLayout`: A generic layout from which more complicated layouts can be built.
- `chapter2`: The main script for the chapter.
- `chapter2.KinectLayout`: A custom layout based on `gui.BaseLayout` that displays the Kinect depth sensor feed. Each captured frame is processed with the `HandGestureRecognition` class described earlier.
- `chapter2.main`: The main function routine for starting the GUI application and accessing the depth sensor.

Setting up the app

Before we can get down to the nitty-gritty of our gesture recognition algorithm, we need to make sure that we can access the Kinect sensor and display a stream of depth frames in a simple GUI.

Accessing the Kinect 3D sensor

Accessing Microsoft Kinect from within OpenCV is not much different from accessing a computer's webcam or camera device. The easiest way to integrate a Kinect sensor with OpenCV is by using an `OpenKinect` module called `freenect`. For installation instructions, take a look at the preceding information box. The following code snippet grants access to the sensor using `cv2.VideoCapture`:

```
import cv2
import freenect


device = cv2.cv.CV_CAP_OPENNI
capture = cv2.VideoCapture(device)
```

On some platforms, the first call to `cv2.VideoCapture` fails to open a capture channel. In this case, we provide a workaround by opening the channel ourselves:

```
if not(capture.isOpened(device)):
    capture.open(device)
```

If you want to connect to your Asus Xtion, the `device` variable should be assigned the `cv2.cv.CV_CAP_OPENNI_ASUS` value instead.

In order to give our app a fair chance of running in real time, we will limit the frame size to 640 x 480 pixels:

```
capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)
```

> If you are using OpenCV 3, the constants you are looking for might be called `cv3.CAP_PROP_FRAME_WIDTH` and `cv3.CAP_PROP_FRAME_HEIGHT`.

The `read()` method of `cv2.VideoCapture` is inappropriate when we need to synchronize a set of cameras or a multihead camera, such as a Kinect. In this case, we should use the `grab()` and `retrieve()` methods instead. An even easier approach when working with `OpenKinect` is to use the `sync_get_depth()` and `sync_get_video()`methods.

For the purpose of this chapter, we will need only the Kinect's depth map, which is a single-channel (grayscale) image in which each pixel value is the estimated distance from the camera to a particular surface in the visual scene. The latest frame can be grabbed via this code:

```
depth, timestamp = freenect.sync_get_depth()
```

The preceding code returns both the depth map and a timestamp. We will ignore the latter for now. By default, the map is in 11-bit format, which is inadequate to be visualized with cv2.imshow right away. Thus, it is a good idea to convert the image to 8-bit precision first.

In order to reduce the range of depth values in the frame, we will clip the maximal distance to a value of 1,023 (or 2**10-1). This will get rid of values that correspond either to noise or distances that are far too large to be of interest to us:

```
np.clip(depth, 0, 2**10-1, depth)
depth >>= 2
```

Then, we will convert the image into 8-bit format and display it:

```
depth = depth.astype(np.uint8)
cv2.imshow("depth", depth)
```

Running the app

In order to run our app, we will need to execute a main function routine that accesses the Kinect, generates the GUI, and executes the main loop of the app. This is done by the main function of chapter2.py:

```
import numpy as np

import wx
import cv2
import freenect

from gui import BaseLayout
from gestures import HandGestureRecognition


def main():
    device = cv2.cv.CV_CAP_OPENNI
    capture = cv2.VideoCapture()
    if not(capture.isOpened()):
        capture.open(device)

    capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
    capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)
```

As in the last chapter, we will design a suitable layout (`KinectLayout`) for the current project:

```
        # start graphical user interface
        app = wx.App()
        layout = KinectLayout(None, -1, 'Kinect Hand Gesture
            Recognition', capture)
        layout.Show(True)
        app.MainLoop()
```

The Kinect GUI

The layout chosen for the current project (`KinectLayout`) is as plain as it gets. It should simply display the live stream of the Kinect depth sensor at a comfortable frame rate of 10 frames per second. Therefore, there is no need to further customize `BaseLayout`:

```
    class KinectLayout(BaseLayout):
        def _create_custom_layout(self):
            pass
```

The only parameter that needs to be initialized this time is the recognition class. This will be useful in just a moment:

```
        def _init_custom_layout(self):
            self.hand_gestures = HandGestureRecognition()
```

Instead of reading a regular camera frame, we need to acquire a depth frame via the `freenect` method `sync_get_depth()`. This can be achieved by overriding the following method:

```
    def _acquire_frame(self):
```

As mentioned earlier, by default this function returns a single-channel depth image with 11-bit precision and a timestamp. However, we are not interested in the timestamp, and we simply pass on the frame if the acquisition is successful:

```
            frame, _ = freenect.sync_get_depth()
            # return success if frame size is valid
            if frame is not None:
                return (True, frame)
            else:
                return (False, frame)
```

The rest of the visualization pipeline is handled by the BaseLayout class. We only need to make sure that we provide a _process_frame method. This method accepts a depth image with 11-bit precision, processes it, and returns an annotated 8-bit RGB color image. Conversion to a regular grayscale image is the same as mentioned in the previous subsection:

```
def _process_frame(self, frame):
    # clip max depth to 1023, convert to 8-bit grayscale
    np.clip(frame, 0, 2**10 – 1, frame)
    frame >>= 2
    frame = frame.astype(np.uint8)
```

The resulting grayscale image can then be passed to the hand gesture recognizer, which will return the estimated number of extended fingers (num_fingers) and the annotated RGB color image mentioned earlier (img_draw):

```
num_fingers, img_draw = self.hand_gestures.recognize(frame)
```

In order to simplify the segmentation task of the HandGestureRecognition class, we will instruct the user to place their hand in the center of the screen. To provide a visual aid for this, let's draw a rectangle around the image center and highlight the center pixel of the image in orange:

```
height, width = frame.shape[:2]
cv2.circle(img_draw, (width/2, height/2), 3, [255, 102, 0], 2)
cv2.rectangle(img_draw, (width/3, height/3), (width*2/3,
          height*2/3), [255, 102, 0], 2)
```

In addition, we will print num_fingers on the screen:

```
cv2.putText(img_draw, str(num_fingers), (30, 30),
          cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))

return img_draw
```

Tracking hand gestures in real time

Hand gestures are analyzed by the HandGestureRecognition class, especially by its recognize method. This class starts off with a few parameter initializations, which will be explained and used later:

```
class HandGestureRecognition:
    def __init__(self):
        # maximum depth deviation for a pixel to be considered
            # within range
        self.abs_depth_dev = 14

        # cut-off angle (deg): everything below this is a
            convexity
        # point that belongs to two extended fingers
        self.thresh_deg = 80.0
```

The `recognize` method is where the real magic takes place. This method handles the entire process flow, from the raw grayscale image all the way to a recognized hand gesture. It implements the following procedure:

1. It extracts the user's hand region by analyzing the depth map (`img_gray`) and returning a hand region mask (`segment`):

   ```
   def recognize(self, img_gray):
       segment = self._segment_arm(img_gray)
   ```

2. It performs contour analysis on the hand region mask (`segment`). Then, it returns the largest contour area found in the image (`contours`) and any convexity defects (`defects`):

   ```
   [contours, defects] = self._find_hull_defects(segment)
   ```

3. Based on the contours found and the convexity defects, it detects the number of extended fingers (`num_fingers`) in the image. Then, it annotates the output image (`img_draw`) with contours, defect points, and the number of extended fingers:

   ```
   img_draw = cv2.cvtColor(img_gray, cv2.COLOR_GRAY2RGB)
   [num_fingers, img_draw] =
       self._detect_num_fingers(contours,
           defects, img_draw)
   ```

4. It returns the estimated number of extended fingers (`num_fingers`), as well as the annotated output image (`img_draw`):

   ```
   return (num_fingers, img_draw)
   ```

Hand region segmentation

The automatic detection of an arm, and later the hand region, could be designed to be arbitrarily complicated, maybe by combining information about the shape and color of an arm or hand. However, using a skin color as a determining feature to find hands in visual scenes might fail terribly in poor lighting conditions or when the user is wearing gloves. Instead, we choose to recognize the user's hand by its shape in the depth map. Allowing hands of all sorts to be present in any region of the image unnecessarily complicates the mission of the present chapter, so we make two simplifying assumptions:

- We will instruct the user of our app to place their hand in front of the center of the screen, orienting their palm roughly parallel to the orientation of the Kinect sensor so that it is easier to identify the corresponding depth layer of the hand.
- We will also instruct the user to sit roughly one to two meters away from the Kinect, and to slightly extend their arm in front of their body so that the hand will end up in a slightly different depth layer than the arm. However, the algorithm will still work even if the full arm is visible.

In this way, it will be relatively straightforward to segment the image based on the depth layer alone. Otherwise, we would have to come up with a hand detection algorithm first, which would unnecessarily complicate our mission. If you feel adventurous, feel free to do this on your own.

Finding the most prominent depth of the image center region

Once the hand is placed roughly in the center of the screen, we can start finding all image pixels that lie on the same depth plane as the hand.

To do this, we simply need to determine the most prominent depth value of the center region of the image. The simplest approach would be as follows: look only at the depth value of the center pixel:

```
width, height = depth.shape
center_pixel_depth = depth[width/2, height/2]
```

Then, create a mask in which all pixels at a depth of `center_pixel_depth` are white and all others are black:

```
import numpy as np

depth_mask = np.where(depth == center_pixel_depth, 255,
    0).astype(np.uint8)
```

However, this approach will not be very robust, because chances are that it will be compromised by the following:

- Your hand will not be placed perfectly parallel to the Kinect sensor
- Your hand will not be perfectly flat
- The Kinect sensor values will be noisy

Therefore, different regions of your hand will have slightly different depth values.

The _segment_arm method takes a slightly better approach; that is, looking at a small neighborhood in the center of the image and determining the median (meaning the most prominent) depth value. First, we find the center region (for example, 21 x 21 pixels) of the image frame:

```
def _segment_arm(self, frame):
    """ segments the arm region based on depth """
    center_half = 10 # half-width of 21 is 21/2-1
    lowerHeight = self.height/2 - center_half
    upperHeight = self.height/2 + center_half
    lowerWidth = self.width/2 - center_half
    upperWidth = self.width/2 + center_half
    center = frame[lowerHeight:upperHeight,
        lowerWidth:upperWidth]
```

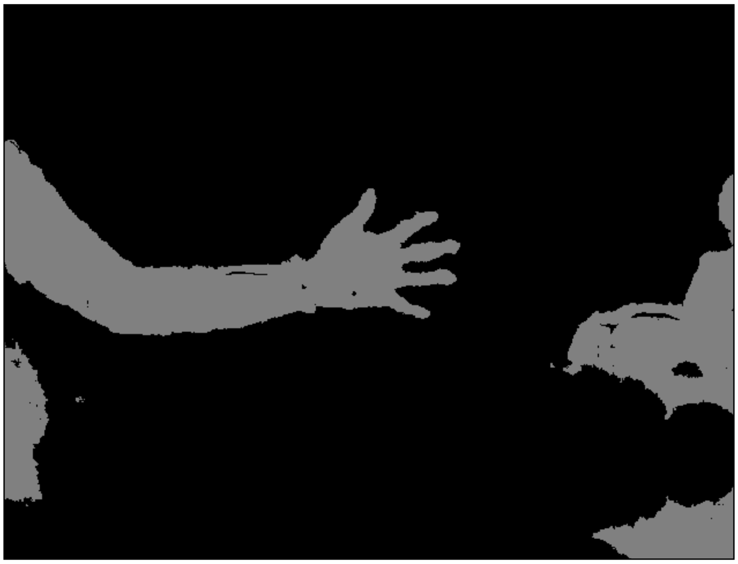
We can then reshape the depth values of this center region into a one-dimensional vector and determine the median depth value, med_val:

```
med_val = np.median(center)
```

We can now compare med_val with the depth value of all pixels in the image and create a mask in which all pixels whose depth values are within a particular range [med_val-self.abs_depth_dev, med_val+self.abs_depth_dev] are white, and all other pixels are black. However, for reasons that will be come clear in a moment, let's paint the pixels gray instead of white:

```
frame = np.where(abs(frame - med_val) <= self.abs_depth_dev,
        128, 0).astype(np.uint8)
```

The result will look like this:



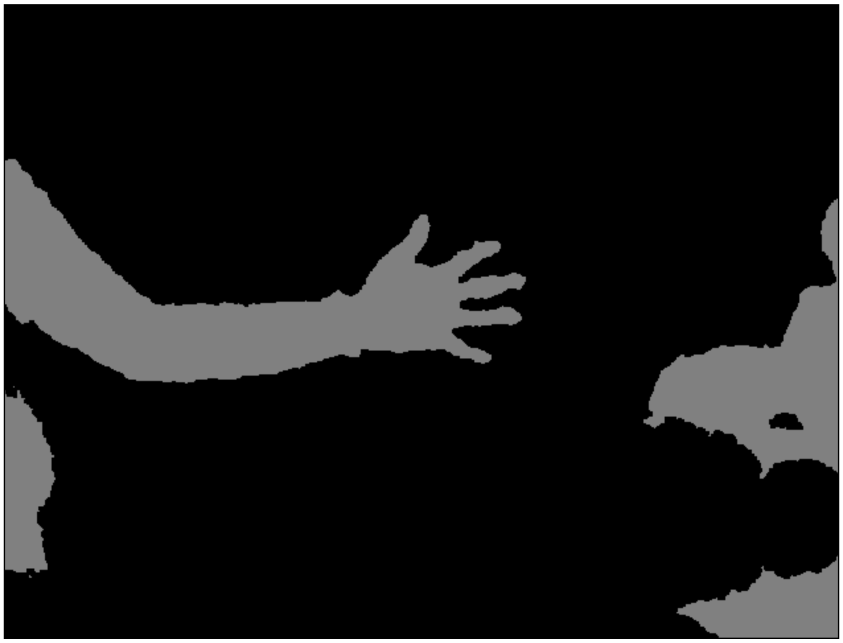
Applying morphological closing to smoothen the segmentation mask

A common problem with segmentation is that a hard threshold typically results in small imperfections (that is, holes, as in the preceding image) in the segmented region. These holes can be alleviated by using morphological opening and closing. Opening removes small objects from the foreground (assuming that the objects are bright on a dark foreground), whereas closing removes small holes (dark regions).

This means that we can get rid of the small black regions in our mask by applying morphological closing (dilation followed by erosion) with a small 3 x 3 pixel kernel:

```
kernel = np.ones((3, 3), np.uint8)
frame = cv2.morphologyEx(frame, cv2.MORPH_CLOSE, kernel)
```

The result looks a lot smoother, as follows:



Notice, however, that the mask still contains regions that do not belong to the hand or arm, such as what appears to be one of my knees on the left and some furniture on the right. These objects just happen to be on the same depth layer as my arm and hand. If possible, we could now combine the depth information with another descriptor, maybe a texture-based or skeleton-based hand classifier, that would weed out all non-skin regions.

Finding connected components in a segmentation mask

An easier approach is to realize that most of the time hands are not connected to knees or furniture. We already know that the center region belongs to the hand, so we can simply apply `cv2.floodfill` to find all the connected image regions.

Before we do this, we want to be absolutely certain that the seed point for the flood fill belongs to the right mask region. This can be achieved by assigning a grayscale value of `128` to the seed point. However, we also want to make sure that the center pixel does not, by any coincidence, lie within a cavity that the morphological operation failed to close. So, let's set a small 7 x 7 pixel region with a grayscale value of `128` instead:

```
small_kernel = 3
frame[self.height/2-small_kernel :
            self.height/2+small_kernel,
            self.width/2-small_kernel :
            self.width/2+small_kernel] = 128
```

As flood filling (as well as morphological operations) is potentially dangerous, later OpenCV versions require the specification of a mask that avoids *flooding* the entire image. This mask has to be 2 pixels wider and taller than the original image and has to be used in combination with the `cv2.FLOODFILL_MASK_ONLY` flag. It can be very helpful in constraining the flood filling to a small region of the image or a specific contour so that we need not connect two neighboring regions that should have never been connected in the first place. It's better to be safe than sorry, right?

Ah, screw it! Today, we feel courageous! Let's make the mask entirely black:

```
mask = np.zeros((self.height+2, self.width+2), np.uint8)
```

Then, we can apply the flood fill to the center pixel (the seed point) and paint all the connected regions white:

```
flood = frame.copy()
cv2.floodFill(flood, mask, (self.width/2, self.height/2),
    255, flags=4 | (255 << 8))
```

At this point, it should be clear why we decided to start with a gray mask earlier. We now have a mask that contains white regions (arm and hand), gray regions (neither arm nor hand but other things in the same depth plane), and black regions (all others). With this setup, it is easy to apply a simple binary threshold to highlight only the relevant regions of the pre-segmented depth plane:

```
ret, flooded = cv2.threshold(flood, 129, 255, cv2.THRESH_BINARY)
```

This is what the resulting mask looks like:

The resulting segmentation mask can now be returned to the `recognize` method, where it will be used as an input to `_find_hull_defects`, as well as a canvas for drawing the final output image (`img_draw`).

Hand shape analysis

Now that we know (roughly) where the hand is located, we aim to learn something about its shape.

Determining the contour of the segmented hand region

The first step involves determining the contour of the segmented hand region. Luckily, OpenCV comes with a pre-canned version of such an algorithm—`cv2.findContours`. This function acts on a binary image and returns a set of points that are believed to be part of the contour. As there might be multiple contours present in the image, it is possible to retrieve an entire hierarchy of contours:

```
def _find_hull_defects(self, segment):
    contours, hierarchy = cv2.findContours(segment,
        cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Furthermore, because we do not know which contour we are looking for, we have to make an assumption to clean up the contour result. Since it is possible that some small cavities are left over even after the morphological closing—but we are fairly certain that our mask contains only the segmented area of interest—we will assume that the largest contour found is the one that we are looking for. Thus, we simply traverse the list of contours, calculate the contour area (`cv2.contourArea`), and store only the largest one (`max_contour`):

```
max_contour = max(contours, key=cv2.contourArea)
```

Finding the convex hull of a contour area

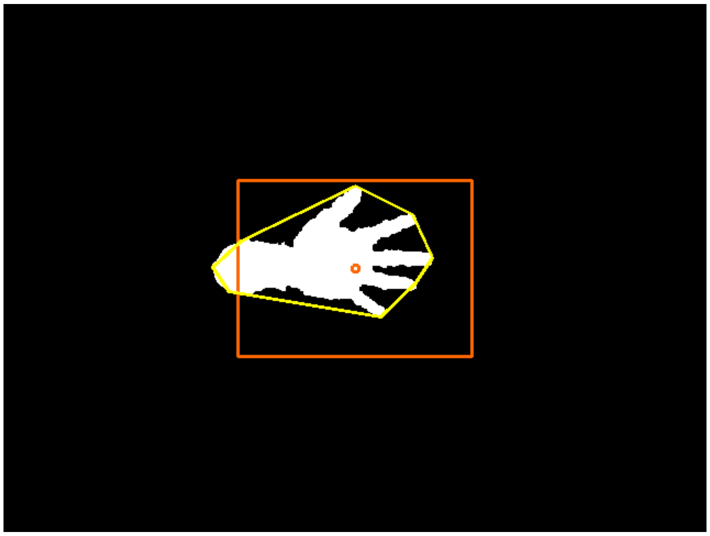
Once we have identified the largest contour in our mask, it is straightforward to compute the convex hull of the contour area. The convex hull is basically the envelope of the contour area. If you think of all the pixels that belong to the contour area as a set of nails sticking out of a board, then the convex hull is the shape formed by a tight rubber band that surrounds all the nails.

We can get the convex hull directly from our largest contour (`max_contour`):

```
hull = cv2.convexHull(max_contour, returnPoints=False)
```

As we now want to look at convexity deficits in this hull, we are instructed by the OpenCV documentation to set the `returnPoints` optional flag to `False`.

The convex hull drawn in yellow around a segmented hand region looks like this:



Finding the convexity defects of a convex hull

As is evident from the preceding screenshot, not all points on the convex hull belong to the segmented hand region. In fact, all the fingers and the wrist cause severe *convexity defects*, that is, points of the contour that are far away from the hull.

We can find these defects by looking at both the largest contour (`max_contour`) and the corresponding convex hull (`hull`):

```
defects = cv2.convexityDefects(max_contour, hull)
```

The output of this function (`defects`) is a 4-tuple that contains `start_index` (the point of the contour where the defect begins), `end_index` (the point of the contour where the defect ends), `farthest_pt_index` (the farthest from the convex hull point within the defect), and `fixpt_depth` (the distance between the farthest point and the convex hull). We will make use of this information in just a moment when we try to extract the number of extended fingers.

For now though, our job is done. The extracted contour (`max_contour`) and convexity defects (`defects`) can be passed to `recognize`, where they will be used as inputs to `_detect_num_fingers`:

```
return (cnt,defects)
```

Hand gesture recognition

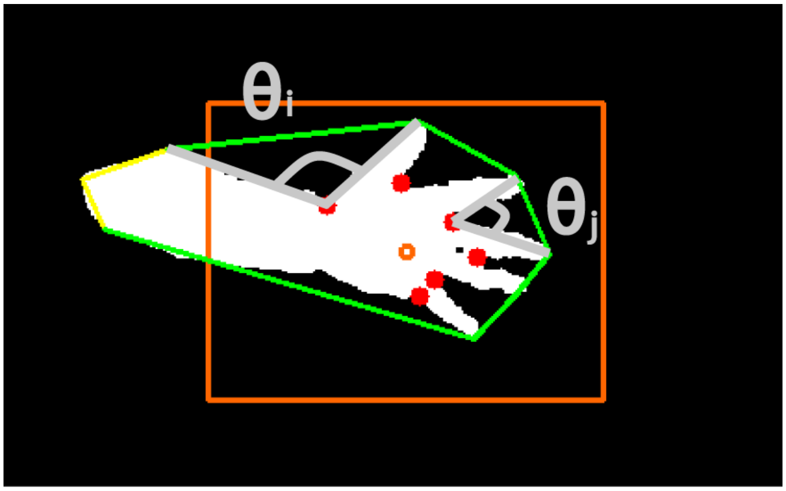
What remains to be done is to classify the hand gesture based on the number of extended fingers. For example, if we find five extended fingers, we assume the hand to be open, whereas no extended fingers implies a fist. All that we are trying to do is count from zero to five and make the app recognize the corresponding number of fingers.

This is actually trickier than it might seem at first. For example, people in Europe might count to three by extending their thumb, index finger, and middle finger. If you do that in the US, people there might get horrendously confused, because they do not tend to use their thumbs when signaling the number two. This might lead to frustration, especially in restaurants (trust me). If we could find a way to generalize these two scenarios—maybe by appropriately counting the number of extended fingers—we would have an algorithm that could teach simple hand gesture recognition to not only a machine but also (maybe) to an average waitress.

As you might have guessed, the answer is related to convexity defects. As mentioned earlier, extended fingers cause defects in the convex hull. However, the inverse is not true; that is, not all convexity defects are caused by fingers! There might be additional defects caused by the wrist, as well as the overall orientation of the hand or the arm. How can we distinguish between these different causes of defects?

Distinguishing between different causes of convexity defects

The trick is to look at the angle between the farthest point from the convex hull point within the defect (`farthest_pt_index`) and the start and end points of the defect (`start_index` and `end_index`, respectively), as illustrated in the following screenshot:



In this screenshot, the orange markers serve as a visual aid to center the hand in the middle of the screen, and the convex hull is outlined in green. Each red dot corresponds to *the point farthest from the convex hull* (`farthest_pt_index`) for every convexity defect detected. If we compare a typical angle that belongs to two extended fingers (such as **θj**) to an angle that is caused by general hand geometry (such as **θi**), we notice that the former is much smaller than the latter. This is obviously because humans can spread their fingers only a little, thus creating a narrow angle made by the farthest defect point and the neighboring fingertips.

Therefore, we can iterate over all convexity defects and compute the angle between the said points. For this, we will need a utility function that calculates the angle (in radians) between two arbitrary, list-like vectors, `v1` and `v2`:

```
def angle_rad(v1, v2):
    return np.arctan2(np.linalg.norm(np.cross(v1, v2)),
        np.dot(v1, v2))
```

This method uses the cross product to compute the angle, rather than doing it in the standard way. The standard way of calculating the angle between two vectors `v1` and `v2` is by calculating their dot product and dividing it by the `norm` of `v1` and the `norm` of `v2`. However, this method has two imperfections:

- You have to manually avoid division by zero if either the `norm` of `v1` or the `norm` of `v2` is zero
- The method returns relatively inaccurate results for small angles

Similarly, we provide a simple function to convert an angle from degrees to radians:

```
def deg2rad(angle_deg):
    return angle_deg/180.0*np.pi
```

Classifying hand gestures based on the number of extended fingers

What remains to be done is actually to classify the hand gesture based on the number of extended fingers. The `_detect_num_fingers` method will take as input the detected contour (`contours`), the convexity defects (`defects`), and a canvas to draw on (`img_draw`):

```
def _detect_num_fingers(self, contours, defects, img_draw):
```

Based on these parameters, it will then determine the number of extended fingers.

However, we first need to define a cut-off angle that can be used as a threshold to classify convexity defects as being caused by extended fingers or not. Except for the angle between the thumb and the index finger, it is rather hard to get anything close to 90 degrees, so anything close to that number should work. We do not want the cut-off angle to be too high, because that might lead to misclassifications:

```
self.thresh_deg = 80.0
```

For simplicity, let's focus on the special cases first. If we do not find any convexity defects, it means that we possibly made a mistake during the convex hull calculation, or there are simply no extended fingers in the frame, so we return `0` as the number of detected fingers:

```
if defects is None:
    return [0, img_draw]
```

However, we can take this idea even further. Due to the fact that arms are usually slimmer than hands or fists, we can assume that the hand geometry will always generate at least two convexity defects (which usually belong to the wrists). So, if there are no additional defects, it implies that there are no extended fingers:

```
if len(defects) <= 2:
    return [0, img_draw]
```

Now that we have ruled out all special cases, we can begin counting real fingers. If there is a sufficient number of defects, we will find a defect between every pair of fingers. Thus, in order to get the number right (`num_fingers`), we should start counting at `1`:

```
num_fingers = 1
```

Then, we can start iterating over all convexity defects. For each defect, we will extract the four elements and draw its hull for visualization purposes:

```
for i in range(defects.shape[0]):
    # each defect point is a 4-tuplestart_idx, end_idx,
        farthest_idx, _ == defects[i, 0]
    start = tuple(contours[start_idx][0])
    end = tuple(contours[end_idx][0])
    far = tuple(contours[farthest_idx][0])

    # draw the hull
    cv2.line(img_draw, start, end [0, 255, 0], 2)
```

Then, we will compute the angle between the two edges from `far` to `start` and from `far` to `end`. If the angle is smaller than `self.thresh_deg` degrees, it means that we are dealing with a defect that is most likely caused by two extended fingers. In such cases, we want to increment the number of detected fingers (`num_fingers`), and draw the point with green. Otherwise, we draw the point with red:

```
# if angle is below a threshold, defect point belongs
# to two extended fingers
if angle_rad(np.subtract(start, far),
        np.subtract(end, far))
        < deg2rad(self.thresh_deg):
    # increment number of fingers
```

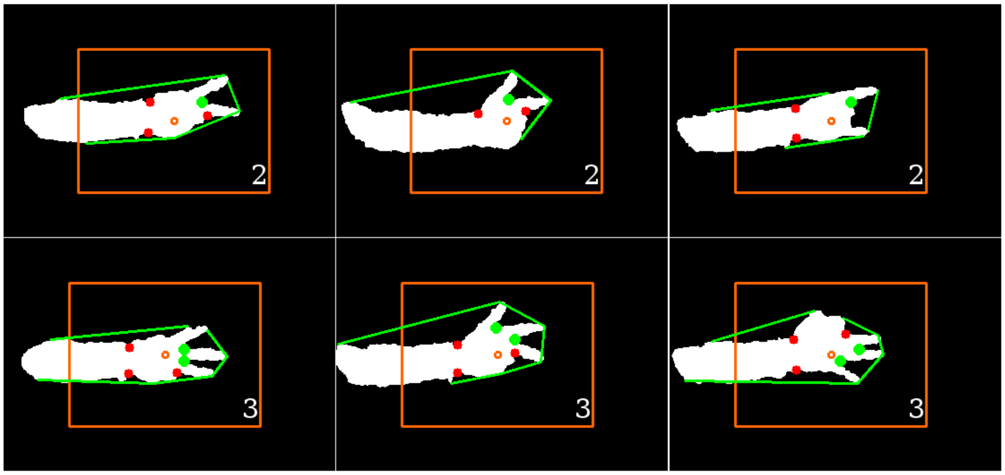
```
        num_fingers = num_fingers + 1

        # draw point as green
        cv2.circle(img_draw, far, 5, [0, 255, 0], -1)
    else:
        # draw point as red
        cv2.circle(img_draw, far, 5, [255, 0, 0], -1)
```

After iterating over all convexity defects, we pass the number of detected fingers and the assembled output image to the `recognize` method:

```
    return (min(5, num_fingers), img_draw)
```

This will make sure that we do not exceed the common number of fingers per hand.

The result can be seen in the following screenshots:



Interestingly, our app is able to detect the correct number of extended fingers in a variety of hand configurations. Defect points between extended fingers are easily classified as such by the algorithm, and others are successfully ignored.

Summary

This chapter showed a relatively simple and yet surprisingly robust way of recognizing a variety of hand gestures by counting the number of extended fingers.

The algorithm first shows how a task-relevant region of the image can be segmented using depth information acquired from a Microsoft Kinect 3D Sensor, and how morphological operations can be used to clean up the segmentation result. By analyzing the shape of the segmented hand region, the algorithm comes up with a way to classify hand gestures based on the types of convexity effects found in the image. Once again, mastering our use of OpenCV to perform a desired task did not require us to produce a large amount of code. Instead, we were challenged to gain an important insight that made us use the built-in functionality of OpenCV in the most effective way possible.

Gesture recognition is a popular but challenging field in computer science, with applications in a large number of areas, such as human-computer interaction, video surveillance, and even the video game industry. You can now use your advanced understanding of segmentation and structure analysis to build your own state-of-the-art gesture recognition system.

In the next chapter, we will continue to focus on detecting objects of interest in visual scenes, but we will assume a much more complicated case—viewing the object from an arbitrary perspective and distance. To do this, we will combine perspective transformations with scale-invariant feature descriptors to develop a robust feature-matching algorithm.

3
Finding Objects via Feature Matching and Perspective Transforms

The goal of this chapter is to develop an app that is able to detect and track an object of interest in the video stream of a webcam, even if the object is viewed from different angles or distances or under partial occlusion.

In this chapter, we will cover the following topics:

- Feature extraction
- Feature matching
- Feature tracking

In the previous chapter, you learned how to detect and track a simple object (the silhouette of a hand) in a very controlled environment. To be more specific, we instructed the user of our app to place the hand in the central region of the screen and made assumptions about the size and shape of the object (the hand). But what if we wanted to detect and track objects of arbitrary sizes, possibly viewed from a number of different angles or under partial occlusion?

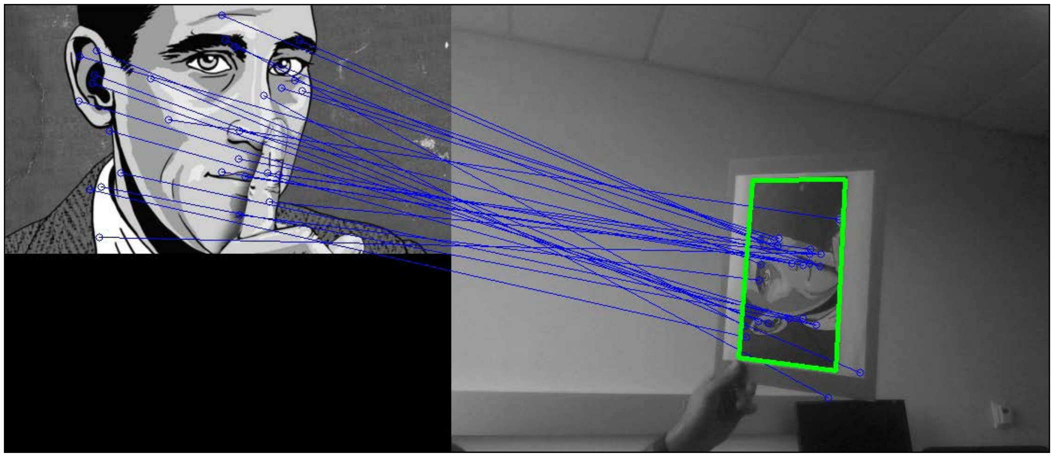
For this, we will make use of feature descriptors, which are a way of capturing the important properties of our object of interest. We do this so that the object can be located even when it is embedded in a busy visual scene. We will again apply our algorithm to the live stream of a webcam, and do our best to keep the algorithm robust yet simple enough to run in real time.

Tasks performed by the app

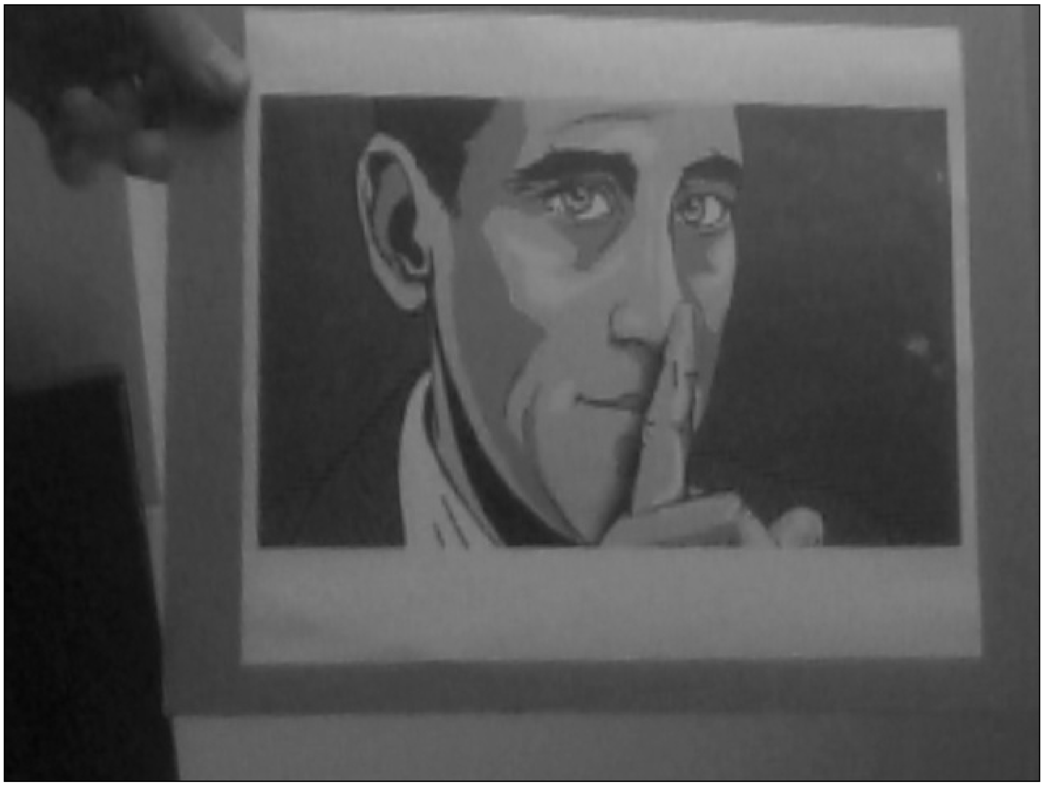
The app will analyze each captured frame to perform the following tasks:

- **Feature extraction**: We will describe an object of interest with **Speeded-Up Robust Features** (**SURF**), which is an algorithm used to find distinctive **keypoints** in an image that are both scale invariant and rotation invariant. These keypoints will help us make sure that we are tracking the right object over multiple frames. Because the appearance of the object might change from time to time, it is important to find keypoints that do not depend on the viewing distance or viewing angle of the object (hence the scale and rotation invariance).
- **Feature matching**: We will try to establish a correspondence between keypoints using the **Fast Library for Approximate Nearest Neighbors** (**FLANN**) to see whether a frame contains keypoints similar to the keypoints from our object of interest. If we find a good match, we will mark the object in each frame.
- **Feature tracking**: We will keep track of the located object of interest from frame to frame using various forms of **early outlier detection** and **outlier rejection** to speed up the algorithm.
- **Perspective transform**: We will then reverse any translations and rotations that the object has undergone by **warping the perspective** so that the object appears upright in the center of the screen. This creates a cool effect in which the object seems frozen in a position while the entire surrounding scene rotates around it.

An example of the first three steps is shown in the following image, which contains a template image of our object of interest on the left, and me holding a printout of the template image on the right. Matching features in the two frames are connected with blue lines, and the located object is outlined in green on the right:



The last step is transforming the located object so that it is projected onto the frontal plane (which should look roughly like the original template image, appearing close-up and roughly upright), while the entire scene seems to warp around it, as shown in the following figure:



> Again, the GUI will be designed with wxPython 2.8, which can be obtained from `http://www.wxpython.org/download.php`. This chapter has been tested with OpenCV 2.4.9. Note that if you are using OpenCV 3, you may have to obtain the so-called *extra* modules from `https://github.com/Itseez/opencv_contrib` and install OpenCV 3 with the `OPENCV_EXTRA_MODULES_PATH` variable set in order to get SURF and FLANN installed. Also, note that you may have to obtain a license to use SURF in commercial applications.

Planning the app

The final app will consist of a Python class for detecting, matching, and tracking image features, as well as a wxPython GUI application that accesses the webcam and displays each processed frame.

The project will contain the following modules and scripts:

- `feature_matching`: A module containing an algorithm for feature extraction, feature matching, and feature tracking. We separate this algorithm from the rest of the application so that it can be used as a standalone module without the need for a GUI.
- `feature_matching.FeatureMatching`: A class that implements the entire feature-matching process flow. It accepts an RGB camera frame and tries to locate an object of interest in it.
- `gui`: A module that provides a wxPython GUI application to access the capture device and display the video feed. This is the same module that we used in previous chapters.
- `gui.BaseLayout`: A generic layout from which more complicated layouts can be built. This chapter does not require any modifications to the basic layout.
- `chapter3`: The main script for the chapter.
- `chapter3.FeatureMatchingLayout`: A custom layout based on `gui.BaseLayout` that displays the webcam video feed. Each captured frame will be processed with the `FeatureMatching` class described earlier.
- `chapter3.main`: The main function routine for starting the GUI application and accessing the depth sensor.

Setting up the app

Before we can get down to the nitty-gritty of our feature-matching algorithm, we need to make sure that we can access the webcam and display the video stream in a simple GUI. Luckily, we have already figured out how to do this in *Chapter 1*, *Fun with Filters*.

Running the app

In order to run our app, we will need to execute a main function routine that accesses the webcam, generates the GUI, and executes the main loop of the app:

```
import cv2
import wx

from gui import BaseLayout
from feature_matching import FeatureMatching


def main():
    capture = cv2.VideoCapture(0)
    if not(capture.isOpened()):
        capture.open()

    capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
    capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)

    # start graphical user interface
    app = wx.App()

    layout = FeatureMatchingLayout(None, -1, 'Feature Matching',
        capture)
    layout.Show(True)
    app.MainLoop()
```

> If you are using OpenCV 3, the constants that you are looking for might be called `cv3.CAP_PROP_FRAME_WIDTH` and `cv3.CAP_PROP_FRAME_HEIGHT`.

The FeatureMatching GUI

Analogous to the previous chapter, the layout chosen for the current project (FeatureMatchingLayout) is as plain as it gets. It should simply display the video feed of the webcam at a comfortable frame rate of 10 frames per second. Therefore, there is no need to further customize BaseLayout:

```
class FeatureMatchingLayout(BaseLayout):
    def _create_custom_layout(self):
        pass
```

The only parameter that needs to be initialized this time is the feature-matching class. We pass to it the path to a template (or training) file that depicts the object of interest:

```
    def _init_custom_layout(self):
        self.matching = FeatureMatching
            (train_image='salinger.jpg')
```

The rest of the visualization pipeline is handled by the BaseLayout class. We only need to make sure that we provide a _process_frame method. This method accepts a RGB color image, processes it by means of the FeatureMatching method match, and passes the processed image for visualization. If the object is detected in the current frame, the match method will report success=True and we will return the processed frame. If the match method is not successful, we will simply return the input frame:

```
    def _process_frame(self, frame):
        self.matching = FeatureMatching
            (train_image='salinger.jpg')
        # if object detected, display new frame, else old one
        success, new_frame = self.matching.match(frame)
        if success:
            return new_frame
        else:
            return frame
```

The process flow

Features are extracted, matched, and tracked by the `FeatureMatching` class, especially by its public `match` method. However, before we can begin analyzing the incoming video stream, we have some homework to do. It might not be clear right away what some of these things mean (especially for SURF and FLANN), but we will discuss these steps in detail in the following sections. For now, we only have to worry about initialization:

```
class FeatureMatching:
     def __init__(self, train_image='salinger.jpg'):
```

1. This sets up a SURF detector (see the next section for details) with a Hessian threshold between 300 and 500:

   ```
   self.min_hessian = 400
   self.SURF = cv2.SURF(self.min_hessian)
   ```

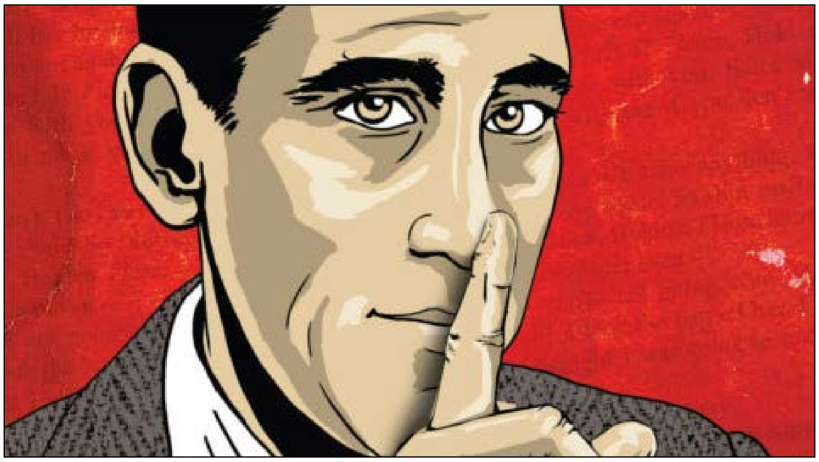
2. We load a template of our object of interest (`self.img_obj`), or print an error if it cannot be found:

   ```
   self.img_obj = cv2.imread(train_image, cv2.CV_8UC1)
   if self.img_obj is None:
       print "Could not find train image " + train_image
           raise SystemExit
   ```

3. Also, store the shape of the image (`self.sh_train`) for convenience:

   ```
   self.sh_train = self.img_train.shape[:2]  # rows, cols
   ```

   For reasons that will soon be evidently clear, we will call the template image the **train image** and every incoming frame a **query image**. The train image has a size of 480 x 270 pixels and looks like this:



4. Apply SURF to the object of interest. This can be done with a convenient function call that returns both a list of keypoints and the descriptor (see the next section for details):

   ```
   self.key_train, self.desc_train =
       self.SURF.detectAndCompute(self.img_obj, None)
   ```

   We will do the same with each incoming frame and compare lists of features across images.

5. Set up a FLANN object (see the next section for details). This requires the specification of some additional parameters via dictionaries, such as which algorithm to use and how many trees to run in parallel:

   ```
   FLANN_INDEX_KDTREE = 0
   index_params = dict(algorithm = FLANN_INDEX_KDTREE,
       trees = 5)
   search_params = dict(checks=50)
   self.flann = cv2.FlannBasedMatcher(index_params,
       search_params)
   ```

6. Finally, initialize some additional bookkeeping variables. These will come in handy when we want to make our feature tracking both quicker and more accurate. For example, we will keep track of the latest computed homography matrix and of the number of frames we have spent without locating our object of interest (see the next section for details):

   ```
   self.last_hinv = np.zeros((3,3))
   self.num_frames_no_success = 0
   self.max_frames_no_success = 5
   self.max_error_hinv = 50.
   ```

Then, the bulk of the work is done by the `FeatureMatching` method `match`. This method follows the procedure elaborated here:

1. It extracts interesting image features from each incoming video frame. This is done in `FeatureMatching._extract_features`.
2. It matches features between the template image and the video frame. This is done in `FeatureMatching._match_features`. If no such match is found, it skips to the next frame.
3. It finds the corner points of the template image in the video frame. This is done in `FeatureMatching._detect_corner_points`. If any of the corners lies (significantly) outside the frame, it skips to the next frame.
4. It calculates the area of the quadrilateral that the four corner points span. If the area is either too small or too large, it skips to the next frame.
5. It outlines the corner points of the template image in the current frame.

6. It finds the perspective transform that is necessary to bring the located object from the current frame to the `frontoparallel` plane. This is done in `FeatureMatching._warp_keypoints`. If the result is significantly different from the result we got recently for an earlier frame, it skips to the next frame.
7. It warps the perspective of the current frame to make the object of interest appear centered and upright.

In the following sections, we will discuss these steps in detail.

Feature extraction

Generally speaking, a feature is an *interesting area* of an image. It is a measurable property of an image that is very informative about what the image represents. Usually, the grayscale value of an individual pixel (the *raw data*) does not tell us a lot about the image as a whole. Instead, we need to derive a property that is more informative.

For example, knowing that there are patches in the image that look like eyes, a nose, and a mouth will allow us to reason about how likely it is that the image represents a face. In this case, the number of resources required to describe the data (are we seeing an image of a face?) is drastically reduced (does the image contain two eyes? a nose? a mouth?).

More low-level features, such as the presence of edges, corners, blobs, or ridges, may be more informative generally. Some features may be better than others, depending on the application. Once we have made up our mind on how to describe our favorite feature, we need to come up with a way to check whether or not the image contains such features and where it contains them.

Feature detection

The process of finding areas of interest in an image is called feature detection. OpenCV provides a whole range of feature detection algorithms, such as these:

- **Harris corner detection**: Knowing that edges are areas with high-intensity changes in all directions, Harris and Stephens came up with a fast way of finding such areas. This algorithm is implemented as `cv2.cornerHarris` in OpenCV.
- **Shi-Tomasi corner detection**: Shi and Tomasi have their own idea of what are good features to track, and they usually do better than Harris corner detection by finding the *N* strongest corners. This algorithm is implemented as `cv2.goodFeaturesToTrack` in OpenCV.

- **Scale-Invariant Feature Transform** (**SIFT**): Corner detection is not sufficient when the scale of the image changes. To this end, Lowe developed a method to describe keypoints in an image that are independent of orientation and size (hence the name **scale invariant**).The algorithm is implemented as `cv2.SIFT` in OpenCV2, but was moved to the *extra* modules in OpenCV3 since its code is proprietary.
- **Speeded-Up Robust Features** (**SURF**): SIFT has proven to be really good, but it is not fast enough for most applications. This is where SURF comes in, which replaces the expensive Laplacian of a Gaussian from SIFT with a box filter. The algorithm is implemented as `cv2.SURF` in OpenCV2, but, like SIFT, it was moved to the *extra* modules in OpenCV3 since its code is proprietary.

OpenCV has support for even more feature descriptors, such as **Features from Accelerated Segment Test** (**FAST**), **Binary Robust Independent Elementary Features** (**BRIEF**), and **Oriented FAST and Rotated BRIEF** (**ORB**), the latter being an open source alternative to SIFT or SURF.

Detecting features in an image with SURF

In the remainder of this chapter, we will make use of the SURF detector.

The SURF algorithm can be roughly divided into two distinctive steps: detecting points of interest, and formulating a descriptor. SURF relies on the Hessian corner detector for interest point detection, which requires the setting of a `min_hessian` threshold. This threshold determines how large the output from the Hessian filter must be in order for a point to be used as an interest point. A larger value results in fewer but (theoretically) more salient interest points, whereas a smaller value results in more numerous but less salient points. Feel free to experiment with different values. In this chapter, we will choose a value of `400`, as seen earlier in `FeatureMatching.__init__`, where we created a SURF descriptor with the following code snippet:

```
self.min_hessian = 400
self.SURF = cv2.SURF(self.min_hessian)
```

Both the features and the descriptor can then be obtained in a single step, for example, on an input image `img_query` without the use of a mask (`None`):

```
key_query, desc_query = self.SURF.detectAndCompute
    (img_query, None)
```

In OpenCV 2.4.8 or later, we can now easily draw the keypoints with the following function:

```
imgOut = cv2.drawKeypoints(img_query, key_query, None,
    (255, 0, 0), 4)
cv2.imshow(imgOut)
```

Make sure that you check `len(keyQuery)` first, as SURF might return a large number of features. If you care only about drawing the keypoints, try setting `min_hessian` to a large value until the number of returned keypoints is manageable.

If our OpenCV distribution is older than that, we might have to write our own function. Note that SURF is protected by patent laws. Therefore, if you wish to use SURF in a commercial application, you will be required to obtain a license.

Feature matching

Once we have extracted features and their descriptors from two (or more) images, we can start asking whether some of these features show up in both (or all) images. For example, if we have descriptors for both our object of interest (`self.desc_train`) and the current video frame (`desc_query`), we can try to find regions of the current frame that look like our object of interest. This is done by the following method, which makes use of the **Fast Library for Approximate Nearest Neighbors** (**FLANN**):

```
good_matches = self._match_features(desc_query)
```

The process of finding frame-to-frame correspondences can be formulated as the search for the nearest neighbor from one set of descriptors for every element of another set.

The first set of descriptors is usually called the train set, because in machine learning, these descriptors are used to train some model, such as the model of the object that we want to detect. In our case, the train set corresponds to the descriptor of the template image (our object of interest). Hence, we call our template image the *train image* (`self.img_train`).

The second set is usually called the query set, because we continually ask whether it contains our train image. In our case, the query set corresponds to the descriptor of each incoming frame. Hence, we call a frame the *query image* (`img_query`).

Features can be matched in any number of ways, for example, with the help of a brute-force matcher (`cv2.BFMatcher`) that looks for each descriptor in the first set and the closest descriptor in the second set by trying each one (exhaustive search).

Matching features across images with FLANN

The alternative is to use an approximate **k-nearest neighbor** (**kNN**) algorithm to find correspondences, which is based on the fast third-party library FLANN. A FLANN match is performed with the following code snippet, where we use kNN with `k=2`:

```
def _match_features(self, desc_frame):
    matches = self.flann.knnMatch(self.desc_train, desc_frame,
        k=2)
```

The result of `flann.knnMatch` is a list of correspondences between two sets of descriptors, both contained in the `matches` variable. These are the train set, because it corresponds to the pattern image of our object of interest, and the query set, because it corresponds to the image in which we are searching for our object of interest.

The ratio test for outlier removal

The more the correct matches found (which means that more pattern-to-image correspondences exist), the more the chances that the pattern is present in the image. However, some matches might be false positives.

A well-known technique for removing outliers is called the ratio test. Since we performed kNN-matching with k=2, the two nearest descriptors are returned for each match. The first match is the closest neighbor and the second match is the second closest neighbor. Intuitively, a correct match will have a much closer first neighbor than its second closest neighbor. On the other hand, the two closest neighbors will be at a similar distance from an incorrect match. Therefore, we can find out how good a match is by looking at the difference between the distances. The **ratio test** says that the match is good only if the distance ratio between the first match and the second match is smaller than a given number (usually around 0.5); in our case, this number chosen to be 0.7. To remove all matches that do not satisfy this requirement, we filter the list of matches and store the good matches in the `good_matches` variable:

```
# discard bad matches, ratio test as per Lowe's paper
good_matches = filter(lambda x: x[0].distance<0.7*x[1].distance,
    matches)
```

Then we pass the matches we found to `FeatureMatching.match` so that they can be processed further:

```
return good_matches
```

Visualizing feature matches

In newer versions of OpenCV, we can easily draw matches using `cv2.drawMatches` or `cv3.drawMatchesKnn`.

In older versions of OpenCV, we may need to write our own function. The goal is to draw both the object of interest and the current video frame (in which we expect the object to be embedded) next to each other:

```
def draw_good_matches(img1, kp1, img2, kp2, matches):
    # Create a new output image that concatenates the
    # two images together (a.k.a) a montage
    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]
    out = np.zeros((max([rows1, rows2]), cols1+cols2, 3),
        dtype='uint8')
```

In order to draw colored lines on the image, we create a three-channel RGB image:

```
    # Place the first image to the left, copy 3x for RGB
    out[:rows1, :cols1, :] = np.dstack([img1, img1, img1])

    # Place the next image to the right of it, copy 3x for RGB
    out[:rows2, cols1:cols1 + cols2, :] = np.dstack([img2, img2,
        img2])
```

Then, for each pair of points between both images, we draw small blue circles, and we connect the two circles with a line. For this, we have to iterate over the list of matching keypoints. The keypoints are stored as tuples in Python, with two entries for the *x* and *y* coordinates. Each match, `m`, stores the index in the keypoint lists, where `m.trainIdx` points to the index in the first keypoint list (`kp1`) and `m.queryIdx` points to the index in the second keypoint list (`kp2`):

```
for m in matches:
    # Get the matching keypoints for each of the images
    c1, r1 = kp1[m.trainIdx].pt
    c2, r2 = kp2[m.queryIdx].pt
```

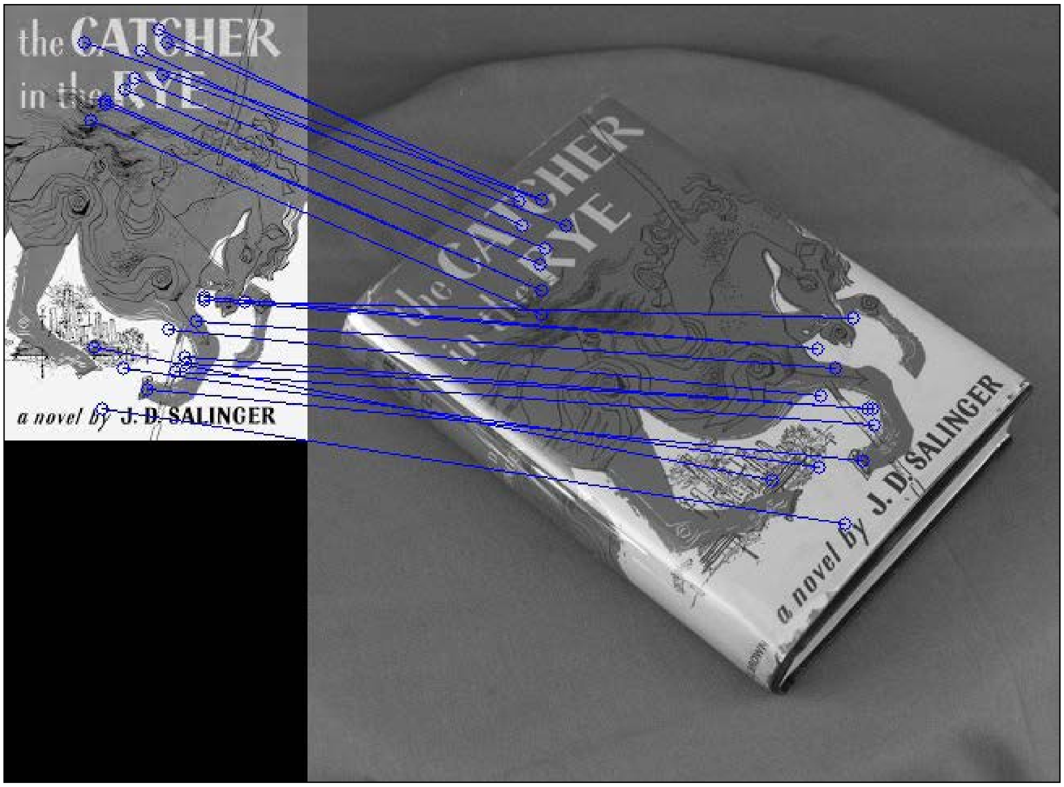
With the correct indices, we can now draw a circle at the correct location (with the radius as 4, the color as blue, and the thickness as 1) and connect the circles with a line:

```
    radius = 4
    BLUE = (255, 0, 0)
    thickness = 1
    # Draw a small circle at both co-ordinates
    cv2.circle(out, (int(c1), int(r1)), radius, BLUE, thickness)
    cv2.circle(out, (int(c2) + cols1, int(r2)), radius, BLUE,
        thickness

    # Draw a line in between the two points
    cv2.line(out, (int(c1), int(r1)), (int(c2) + cols1, int(r2)),
        BLUE, thickness)
    return out
```

Then, the returned image can be drawn with this code:

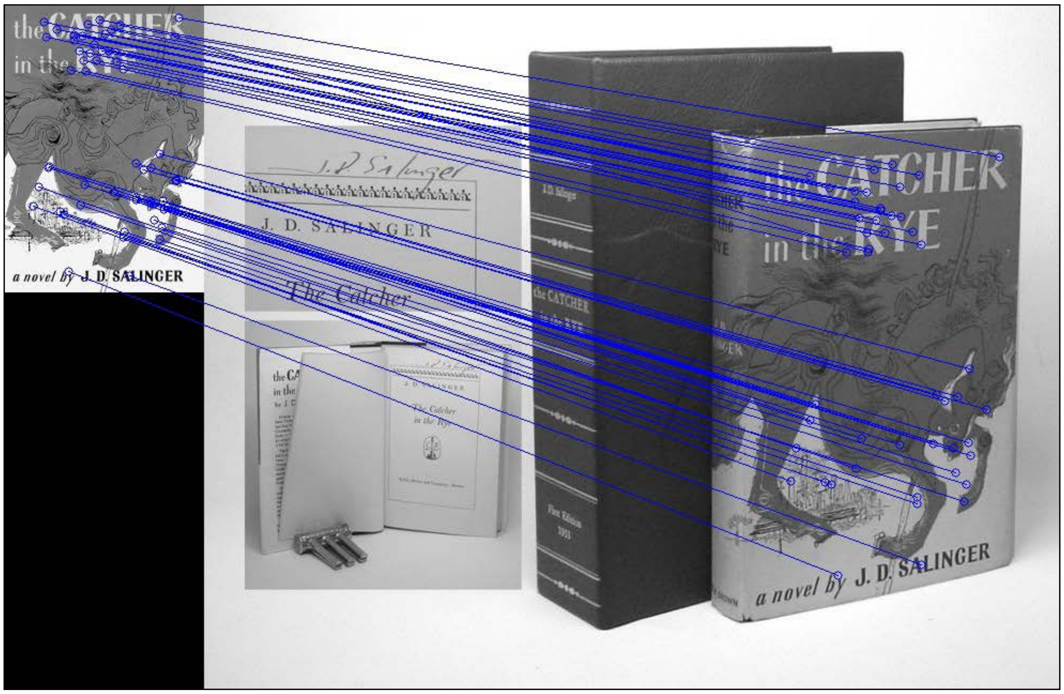
```
cv2.imshow('imgFlann', draw_good_matches(self.img_train,
    self.key_train, img_query, key_query, good_matches))
```

The blue lines connect the features in the object (left) to the features in the scenery (right), as shown here:



This works fine in a simple example such as this, but what happens when there are other objects in the scene? Since our object contains some lettering that seems highly salient, what happens when there are other words present?

As it turns out, the algorithm works even under such conditions, as you can see in this screenshot:



Interestingly, the algorithm did not confuse the name of the author as seen on the left with the black-on-white lettering next to the book in the scene, even though they spell out the same name. This is because the algorithm found a description of the object that does not rely purely on the grayscale representation. On the other hand, an algorithm doing a pixel-wise comparison could have easily gotten confused.

Homography estimation

Since we are assuming that the object of our interest is planar (an image) and rigid, we can find the homography transformation between the feature points of the two images. Homography will calculate the perspective transformation required to bring all feature points in the object image (`self.key_train`) into the same plane as all the feature points in the current image frame (`self.key_query`). But first, we need to find the image coordinates of all keypoints that are good matches:

```
def _detect_corner_points(self, key_frame, good_matches):
    src_points = [self.key_train[good_matches[i].trainIdx].pt
        for i in xrange(len(good_matches))]
    dst_points = [keyQuery[good_matches[i].queryIdx].pt
        for i in xrange(len(good_matches))]
```

To find the correct perspective transformation (a homography matrix `H`), the `cv2.findHomography` function will use the **random sample consensus** (**RANSAC**) method to probe different subsets of input points:

```
H, _ = cv2.findHomography(np.array(src_points),
    np.array(dst_points), cv2.RANSAC)
```

The homography matrix `H` can then help us transform any point in the pattern into the scenery, such as transforming a corner point in the training image to a corner point in the query image. In other words, this means that we can draw the outline of the book cover in the query image by transforming the corner points from the training image! For this, we take the list of corner points of the training image (`src_corners`) and see where they are projected in the query image by performing a perspective transform:

```
self.sh_train = self.img_train.shape[:2]  # rows, cols
src_corners = np.array([(0,0), (self.sh_train[1],0),
    (self.sh_train[1],self.sh_train[0]), (0,self.sh_train[0])],
    dtype=np.float32)
dst_corners = cv2.perspectiveTransform(src_corners[None, :, :],
    H)
```

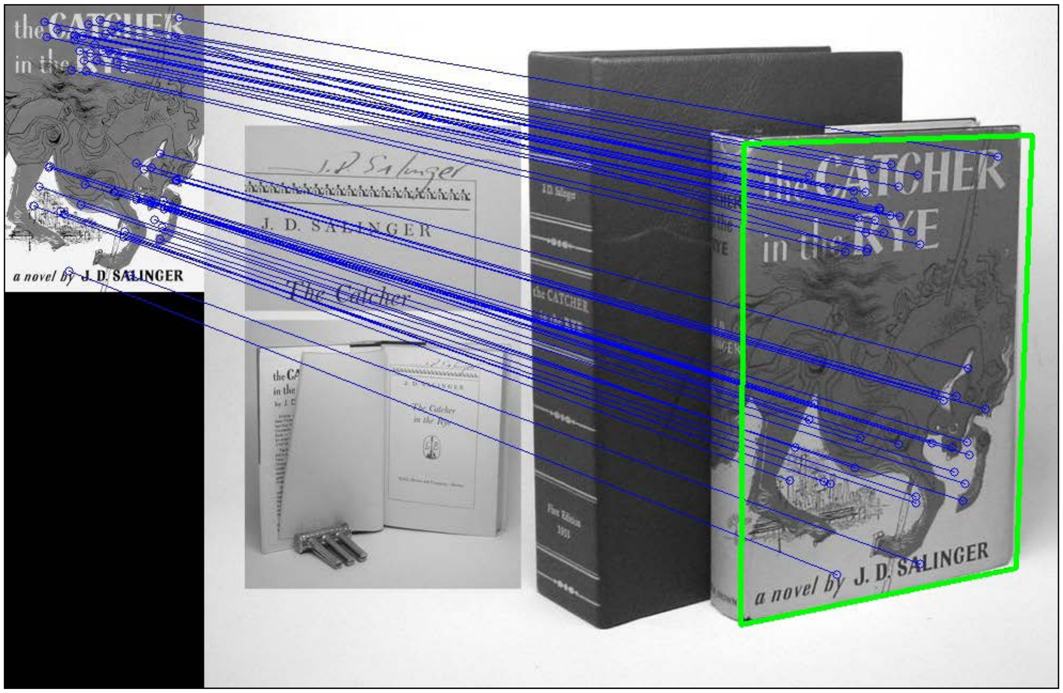
The `dst_corners` return argument is a list of image points. All that we need to do is draw a line between each point in `dst_corners` and the very next one, and we will have an outline in the scenery. But first, in order to draw the line at the right image coordinates, we need to offset the *x* coordinate by the width of the pattern image (because we are showing the two images next to each other):

```
dst_corners = map(tuple,dst_corners[0])
dst_corners = [(np.int(dst_corners[i][0]+self.sh_train[1]),
    np.int(dst_corners[i][1]))
```

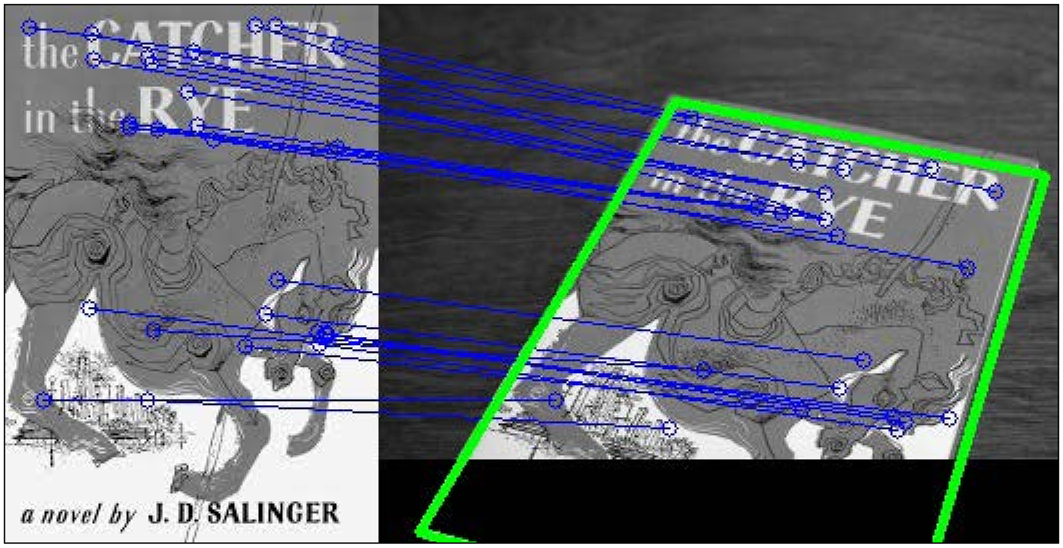
Then we can draw the lines from the *ith* point to the *(i+1)-th* point in the list (wrapping around to 0):

```
for i in xrange(0,len(dst_corners)):
    cv2.line(img_flann, dst_corners[i], dst_corners[(i+1) % 4],
        (0, 255, 0), 3)
```

Finally, we draw the outline of the book cover, like this:



This works even when the object is only partially visible, as follows:



Warping the image

We can also do the opposite—going from the probed scenery to the training pattern coordinates. This makes it possible for the book cover to be brought onto the frontal plane, as if we were looking at it directly from above. To achieve this, we can simply take the inverse of the homography matrix to get the inverse transformation:

```
Hinv = cv2.linalg.inverse(H)
```

However, this would map the top-left corner of the book cover to the origin of our new image, which would cut off everything to the left of and above the book cover. Instead, we want to roughly center the book cover in the image. Thus, we need to calculate a new homography matrix. As input, we will have our `pts_scene` scenery points. As output, we want an image that has the same shape as the pattern image:

```
dst_size = img_in.shape[:2]  # cols, rows
```

The book cover should be roughly half of that size. We can come up with a scaling factor and a bias term so that every keypoint in the scenery image is mapped to the correct coordinate in the new image:

```
scale_row = 1./src_size[0]*dst_size[0]/2.
bias_row = dst_size[0]/4.
scale_col = 1./src_size[1]*dst_size[1]/2.
bias_col = dst_size[1]/4.
```

Next, we just need to apply this linear scaling to every keypoint in the list. The easiest way to do this is with list comprehensions:

```
src_points = [key_frame[good_matches[i].trainIdx].pt
    for i in xrange(len(good_matches))]
dst_points = [self.key_train[good_matches[i].queryIdx].pt
    for i in xrange(len(good_matches))]
dst_points = [[x*scale_row+bias_row, y*scale_col+bias_col]
    for x, y in dst_points]
```

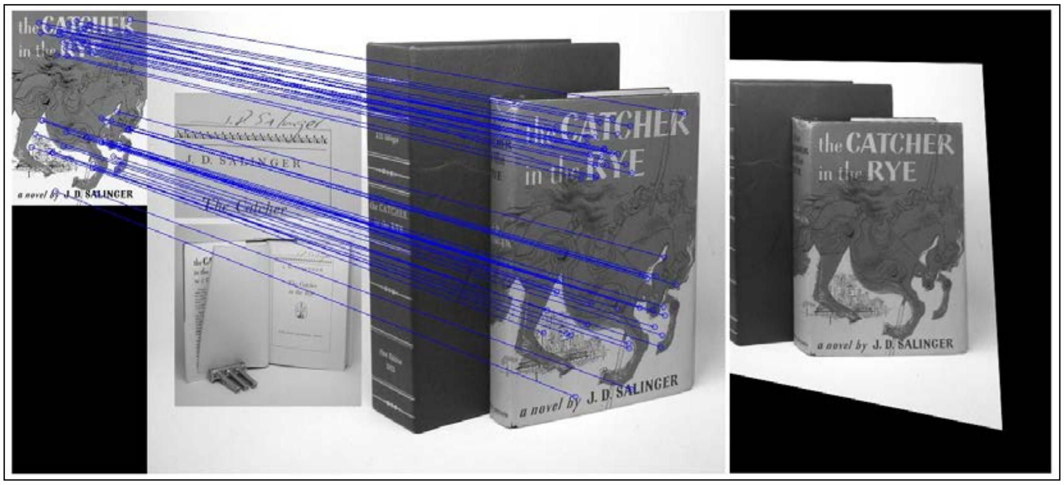
Then we can find the homography matrix between these points (make sure that the list is converted to a NumPy array):

```
Hinv, _ = cv2.findHomography(np.array(src_points),
    np.array(dst_points), cv2.RANSAC)
```

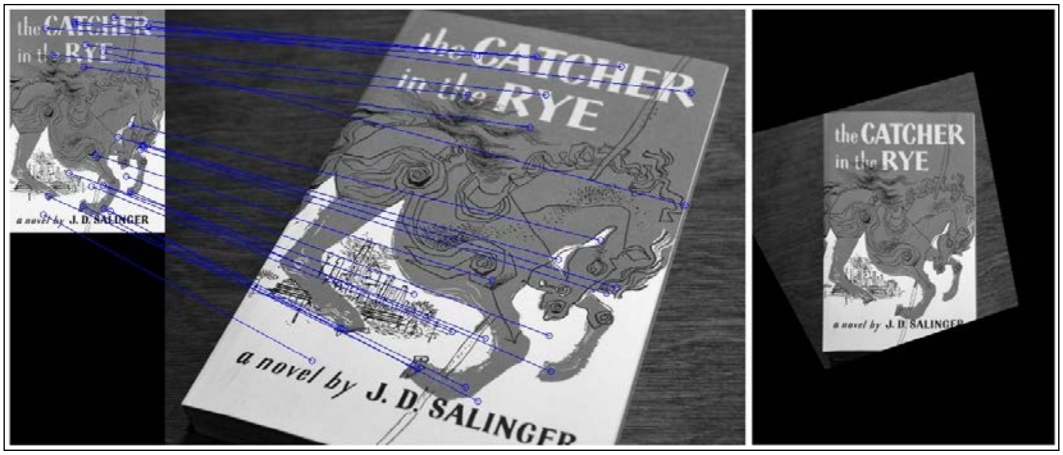
After that, we can use the homography matrix to transform every pixel in the image (this is also called warping the image):

```
img_warp = cv2.warpPerspective(img_query, Hinv, dst_size)
```

The result looks like this (matching on the left and warped image on the right):



The image resulting from the perspective transformation might not be perfectly aligned with the `frontoparallel` plane, because after all, the homography matrix is only approximate. In most cases, however, our approach works just fine, such as in the example shown in the following figure:



Feature tracking

Now that our algorithm works for single frames, how can we make sure that the image found in one frame will also be found in the very next frame?

In FeatureMatching.__init__, we created some bookkeeping variables that we said we would use for feature tracking. The main idea is to enforce some coherence while going from one frame to the next. Since we are capturing roughly 10 frames per second, it is reasonable to assume that the changes from one frame to the next will not be too radical. Therefore, we can be sure that the result we get in any given frame has to be similar to the result we got in the previous frame. Otherwise, we discard the result and move on to the next frame.

However, we have to be careful not to get stuck with a result that we think is reasonable but is actually an outlier. To solve this problem, we keep track of the number of frames we have spent without finding a suitable result. We use self.num_frames_no_success; if this number is smaller than a certain threshold, say self.max_frames_no_success, we do the comparison between the frames. If it is greater than the threshold, we assume that too much time has passed since the last result was obtained, in which case it would be unreasonable to compare the results between the frames.

Early outlier detection and rejection

We can extend the idea of outlier rejection to every step in the computation. The goal then becomes minimizing the workload while maximizing the likelihood that the result we obtain is a good one.

The resulting procedure for early outlier detection and rejection is embedded in FeatureMatching.match and looks as follows:

```
def match(self, frame):
    # create a working copy (grayscale) of the frame
    # and store its shape for convenience
    img_query = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    sh_query = img_query.shape[:2]  # rows,cols
```

1. Find good matches between the feature descriptors of the pattern and the query image:

   ```
   key_query, desc_query = self._extract_features(img_query)
   good_matches = self._match_features(descQuery)
   ```

   In order for RANSAC to work in the very next step, we need at least four matches. If fewer matches are found, we admit defeat and return False right away:

   ```
   if len(good_matches) < 4:
       self.num_frames_no_success=
           self.num_frames_no_success + 1
       return False, frame
   ```

2. Find the corner points of the pattern in the query image (`dst_corners`):

   ```
   dst_corners = self._detect_corner_points(key_query,
       good_matches)
   ```

   If any of these points lies significantly outside the image (by 20 pixels in our case), it means that either we are not looking at our object of interest, or the object of interest is not entirely in the image. In both cases, we have no interest in proceeding, and we return `False`:

   ```
   if np.any(filter(lambda x: x[0] < -20 or x[1] < -20
       or x[0] > sh_query[1] + 20 or x[1] > sh_query[0] + 20,
       dst_corners)):
           self.num_frames_no_success =
               self.num_frames_no_success + 1
           return False, frame
   ```

3. If the four recovered corner points do not span a reasonable quadrilateral (a polygon with four sides), it means that we are probably not looking at our object of interest. The area of a quadrilateral can be calculated with this code:

   ```
   area = 0
   for i in xrange(0, 4):
       next_i = (i + 1) % 4
       area = area + (dst_corners[i][0]*dst_corners[next_i][1]
           - dst_corners[i][1]*dst_corners[next_i][0])/2.
   ```

   If the area is either unreasonably small or unreasonably large, we discard the frame and return `False`:

   ```
   if area < np.prod(sh_query)/16. or area >
       np.prod(sh_query)/2.:
           self.num_frames_no_success=
               self.num_frames_no_success + 1
           return False, frame
   ```

4. If the recovered homography matrix is too different from the one that we last recovered (`self.last_hinv`), it means that we are probably looking at a different object, in which case we discard the frame and return `False`. We compare the current homography matrix to the last one by calculating the distance between the two matrices:

   ```
   np.linalg.norm(Hinv - self.last_hinv)
   ```

   However, we only want to consider `self.last_hinv` if it is fairly recent, say, from within the last `self.max_frames_no_success`. This is why we keep track of `self.num_frames_no_success`:

   ```
   recent = self.num_frames_no_success <
       self.max_frames_no_success
   ```

```
        similar = np.linalg.norm(Hinv - self.last_hinv) <
            self.max_error_hinv
        if recent and not similar:
            self.num_frames_no_success =
                self.num_frames_no_success + 1
            return False, frame
```

This will help us keep track of the one and the same object of interest over time. If, for any reason, we lose track of the pattern image for more than `self.max_frames_no_success` frames, we skip this condition and accept whatever homography matrix was recovered up to that point. This makes sure that we do not get stuck with some `self.last_hinv` matrix that is actually an outlier.

Otherwise, we can be fairly certain that we have successfully located the object of interest in the current frame. In such a case, we store the homography matrix and reset the counter:

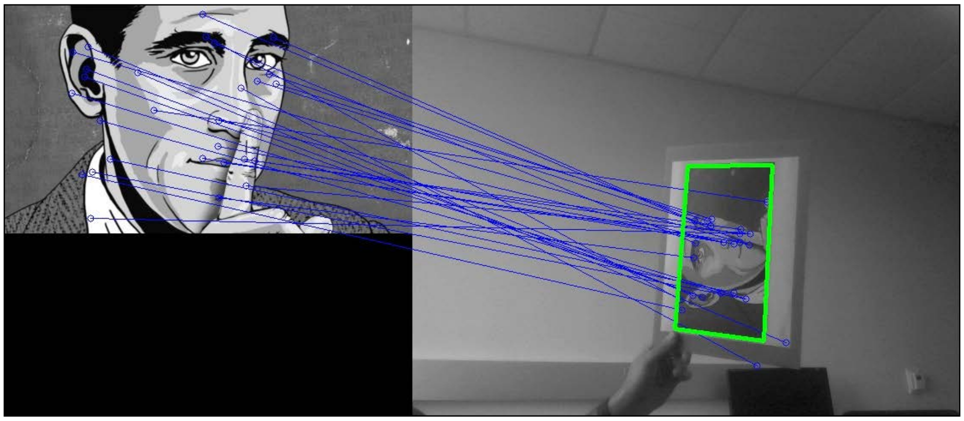
```
    self.num_frames_no_success = 0
    self.last_hinv = Hinv
```

All that is left to do is warping the image and (for the first time) returning `True` along with the warped image so that the image can be plotted:

```
    img_out = cv2.warpPerspective(img_query, Hinv, dst_size)
    img_out = cv2.cvtColor(img_out, cv2.COLOR_GRAY2RGB)
    return True, imgOut
```

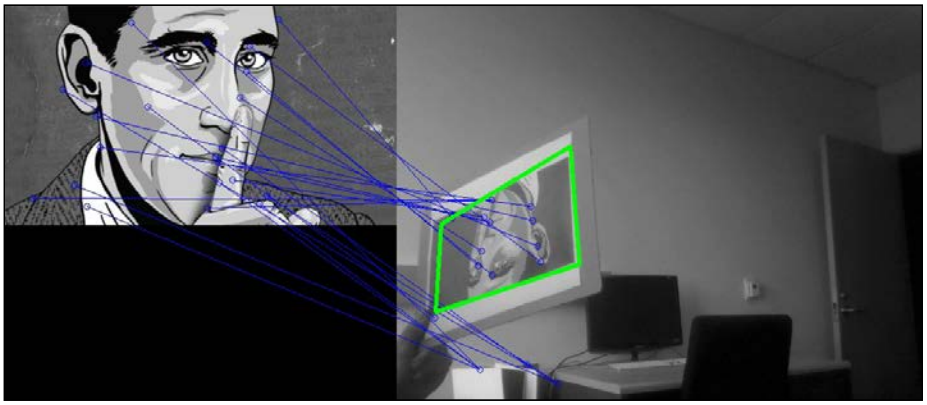
Seeing the algorithm in action

The result of the matching procedure in a live stream from my laptop's webcam looks like this:

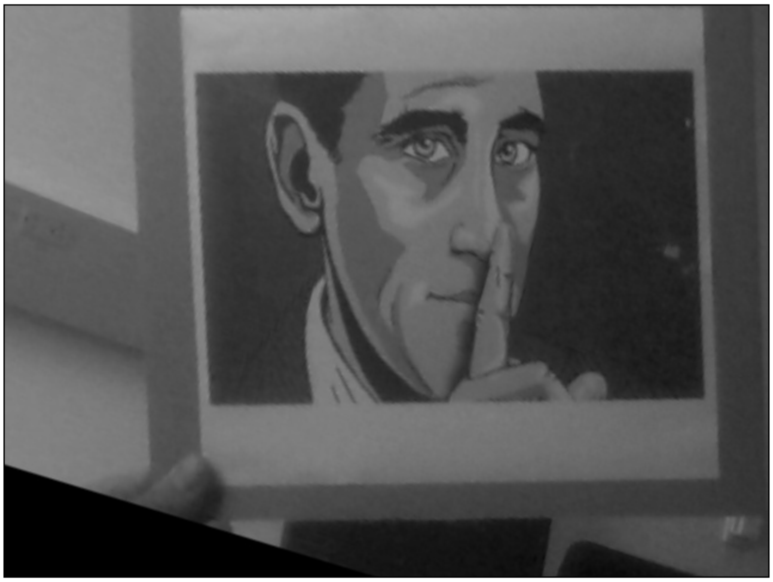


As you can see, most of the keypoints in the pattern image were matched correctly with their counterparts in the query image on the right. The printout of the pattern can now be slowly moved around, tilted, and turned. As long as all the corner points stay in the current frame, the homography matrix is updated accordingly and the outline of the pattern image is drawn correctly.

This works even if the printout is upside down, as shown here:



In all cases, the warped image brings the pattern image to an upright, centered position on the `frontoparallel` plane. This creates a cool effect of having the pattern image frozen in place in the center of the screen, while the surroundings twist and turn around it, like this:



In most cases, the warped image looks fairly accurate, as seen in the one earlier. If, for any reason, the algorithm accepts a wrong homography matrix that leads to an unreasonably warped image, then the algorithm will discard the outlier and recover within half a second (that is, within `self.max_frames_no_success` frames), leading to accurate and efficient tracking throughout.

Summary

This chapter showed a robust feature tracking method that is fast enough to run in real time when applied to the live stream of a webcam.

First, the algorithm shows you how to extract and detect important features in an image independently of perspective and size, be it in a template of our object of interest (train image) or a more complex scene in which we expect the object of interest to be embedded (query image). A match between feature points in the two images is then found by clustering the keypoints using a fast version of the nearest neighbor algorithm. From there on, it is possible to calculate a perspective transformation that maps one set of feature points to the other. With this information, we can outline the train image as found in the query image and warp the query image so that the object of interest appears upright in the center of the screen.

With this in hand, we now have a good starting point for designing a cutting-edge feature tracking, image stitching, or augmented-reality application.

In the next chapter, we will continue studying the geometrical features of a scene, but this time, we will be concentrating on motion. Specifically, we will study how to reconstruct a scene in 3D by inferring its geometrical features from camera motion. For this, we will have to combine our knowledge of feature matching with optic flow and structure-from-motion techniques.

4

3D Scene Reconstruction Using Structure from Motion

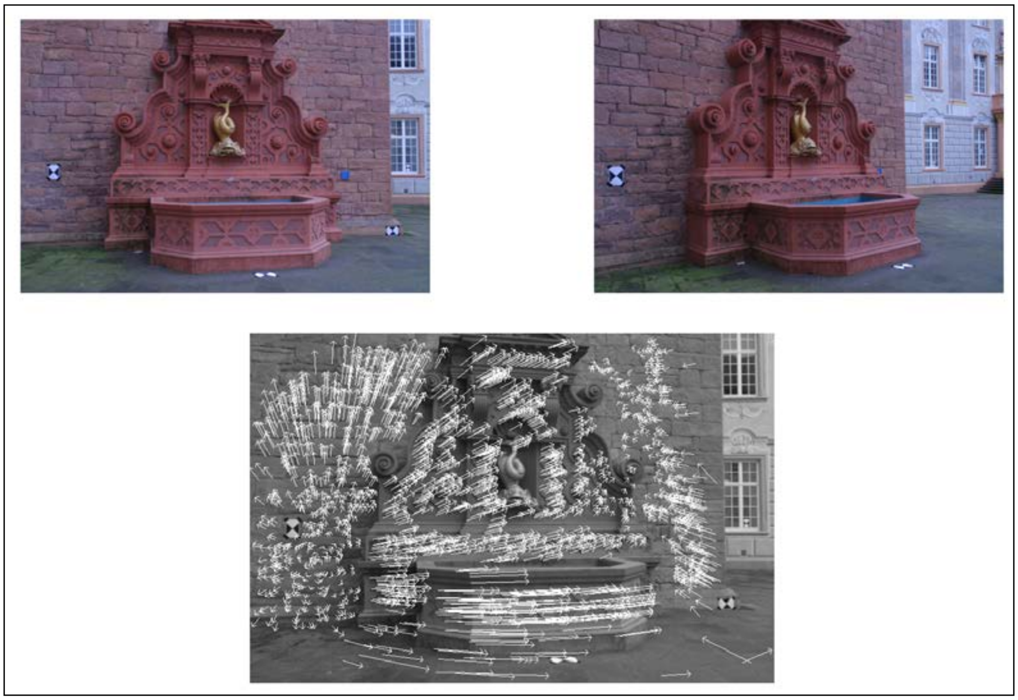
The goal of this chapter is to study how to reconstruct a scene in 3D by inferring the geometrical features of the scene from camera motion. This technique is sometimes referred to as **structure from motion**. By looking at the same scene from different angles, we will be able to infer the real-world 3D coordinates of different features in the scene. This process is known as **triangulation**, which allows us to **reconstruct** the scene as a **3D point cloud**.

In the previous chapter, you learned how to detect and track an object of interest in the video stream of a webcam, even if the object is viewed from different angles or distances, or under partial occlusion. Here, we will take the tracking of interesting features a step further and consider what we can learn about the entire visual scene by studying similarities between image frames. If we take two pictures of the same scene from different angles, we can use **feature matching** or **optic flow** to estimate any translational and rotational movement that the camera underwent between taking the two pictures. However, in order for this to work, we will first have to calibrate our camera.

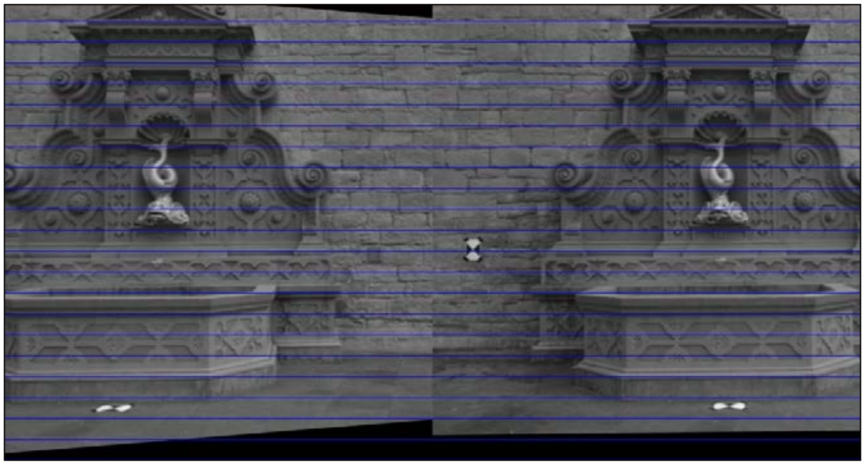
The complete procedure involves the following steps:

1. **Camera calibration**: We will use a chessboard pattern to extract the intrinsic camera matrix as well as the distortion coefficients, which are important for performing the scene reconstruction.

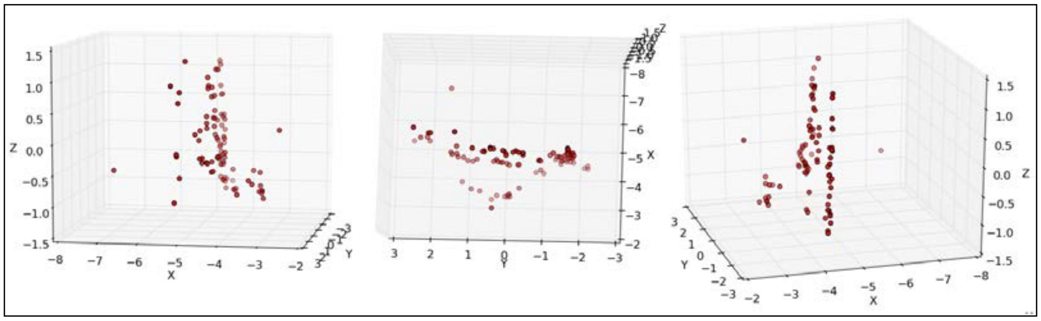
2. **Feature matching**: We will match points in two 2D images of the same visual scene, either via **Speeded-Up Robust Features** (**SURF**) or via optic flow, as seen in the following image:



3. **Image rectification**: By estimating the camera motion from a pair of images, we will extract the **essential matrix** and rectify the images.



4. **Triangulation**: We will reconstruct the 3D real-world coordinates of the image points by making use of constraints from **epipolar geometry**.
5. **3D point cloud visualization**: Finally, we will visualize the recovered 3D structure of the scene using scatterplots in matplotlib, which is most compelling when studied using pyplot's **Pan axes** button. This button lets you rotate and scale the point cloud in all three dimensions. It is a little harder to visualize in still frames, as can be seen in the following figure (left panel: standing slightly in front to the left side of the fountain, center panel: looking down on the fountain, right panel: standing slightly in front to the right of the fountain):



This chapter has been tested with OpenCV 2.4.9 and wxPython 2.8 (`http://www.wxpython.org/download.php`). It also requires NumPy (`http://www.numpy.org`) and matplotlib (`http://www.matplotlib.org/downloads.html`). Note that if you are using OpenCV3, you may have to obtain the so-called *extra* modules from `https://github.com/Itseez/opencv_contrib` and install OpenCV3 with the `OPENCV_EXTRA_MODULES_PATH` variable set in order to get SURF installed. Also note that you may have to obtain a license to use SURF in commercial applications.

Planning the app

The final app will extract and visualize structure from motion on a pair of images. We will assume that these two images have been taken with the same camera, whose internal camera parameters we know. If these parameters are not known, they need to be estimated first in a camera calibration process.

The final app will then consist of the following modules and scripts:

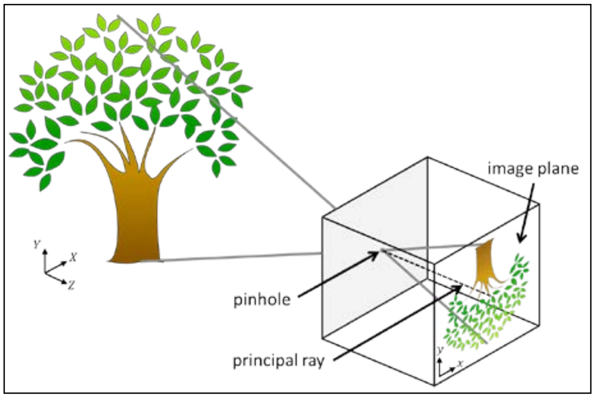
- `chapter4.main`: This is the main function routine for starting the application.
- `scene3D.SceneReconstruction3D`: This is a class that contains a range of functionalities for calculating and visualizing structure from motion. It includes the following public methods:
    - `__init__`: This constructor will accept the intrinsic camera matrix and the distortion coefficients
    - `load_image_pair`: A method used to load from the file, two images that have been taken with the camera described earlier
    - `plot_optic_flow`: A method used to visualize the optic flow between the two image frames
    - `draw_epipolar_lines`: A method used to draw the epipolar lines of the two images
    - `plot_rectified_images`: A method used to plot a rectified version of the two images
- `plot_point_cloud`: This is a method used to visualize the recovered real-world coordinates of the scene as a 3D point cloud. In order to arrive at a 3D point cloud, we will need to exploit epipolar geometry. However, epipolar geometry assumes the pinhole camera model, which no real camera follows. We need to rectify our images to make them look as if they have come from a pinhole camera. For that, we need to estimate the parameters of the camera, which leads us to the field of camera calibration.

Camera calibration

So far, we have worked with whatever image came straight out of our webcam, without questioning the way in which it was taken. However, every camera lens has unique parameters, such as focal length, principal point, and lens distortion. What happens behind the covers when a camera takes a picture, is that; light falls through a lens, followed by an aperture, before falling on the surface of a light sensor. This process can be approximated with the pinhole camera model. The process of estimating the parameters of a real-world lens such that it would fit the pinhole camera model is called camera calibration (or **camera resectioning**, and it should not be confused with photometric camera calibration).

The pinhole camera model

The **pinhole camera model** is a simplification of a real camera in which there is no lens and the camera aperture is approximated by a single point (the pinhole). When viewing a real-world 3D scene (such as a tree), light rays pass through the point-sized aperture and fall on a 2D image plane inside the camera, as seen in the following diagram:



In this model, a 3D point with coordinates *(X,Y,Z)* is mapped to a 2D point with coordinates *(x,y)* that lies on the **image plane**. Note that this leads to the tree appearing upside down on the image plane. The line that is perpendicular to the image plane, and passes through the pinhole is called the **principal ray**, and its length is called the **focal length**. The focal length is a part of the internal camera parameters, as it may vary depending on the camera being used.

Hartley and Zisserman found a mathematical formula to describe how a 2D point with coordinates *(x,y)* can be inferred from a 3D point with coordinates *(X,Y,Z)* and the camera's intrinsic parameters, as follows:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

For now, let's focus on the 3 x 3 matrix in the preceding formula, which is the **intrinsic camera matrix**—a matrix that compactly describes all internal camera parameters. The matrix comprises focal lengths (`fx` and `fy`) and optical centers (`cx` and `cy`) expressed in pixel coordinates. As mentioned earlier, the focal length is the distance between the pinhole and the image plane. A true pinhole camera has only one focal length, in which case `fx = fy = f`. However, in reality, these two values might differ, maybe due to flaws in the digital camera sensor. The point at which the principal ray intersects the image plane is called the principal point, and its relative position on the image plane is captured by the optical center (or principal point offset).

In addition, a camera might be subject to radial or tangential distortion, leading to a *fish-eye* effect. This is because of hardware imperfections and lens misalignments. These distortions can be described with a list of the **distortion coefficients**. Sometimes, radial distortions are actually a desired artistic effect. At other times, they need to be corrected.

> For more information on the pinhole camera model, there are many good tutorials out there on the Web, such as `http://ksimek.github.io/2013/08/13/intrinsic`.

Because these parameters are specific to the camera hardware (hence the name intrinsic), we need to calculate them only once in the lifetime of a camera. This is called **camera calibration**.

Estimating the intrinsic camera parameters

In OpenCV, camera calibration is fairly straightforward. The official documentation provides a good overview of the topic and some sample C++ scripts at `http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html`.

For educational purposes, we will develop our own calibration script in Python. We will need to present a special pattern image, with a known geometry (chessboard plate or black circles on a white background), to the camera we wish to calibrate. Because we know the geometry of the pattern image, we can use feature detection to study the properties of the internal camera matrix. For example, if the camera suffers from undesired radial distortion, the different corners of the chessboard pattern will appear distorted in the image and not lie on a rectangular grid. By taking about 10 to 20 snapshots of the chessboard pattern from different points of view, we can collect enough information to correctly infer the camera matrix and the distortion coefficients.

For this, we will use the `calibrate.py` script. Analogous to previous chapters, we will use a simple layout (`CameraCalibration`) based on `BaseLayout` that embeds a webcam video stream. The main function of the script will generate the GUI and execute the main loop of the app:

```
import cv2
import numpy as np
import wx

from gui import BaseLayout


    def main():
        capture = cv2.VideoCapture(0)
        if not(capture.isOpened()):
            capture.open()

        capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
        capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)

        # start graphical user interface
        app = wx.App()
        layout = CameraCalibration(None, -1, 'Camera Calibration',
            capture)
        layout.Show(True)
        app.MainLoop()
```

> If you are using OpenCV 3, the constants that you are looking for might be called `cv3.CAP_PROP_FRAME_WIDTH` and `cv3.CAP_PROP_FRAME_HEIGHT`.

The camera calibration GUI

The GUI is a customized version of the generic `BaseLayout`:

```
class CameraCalibration(BaseLayout):
```

The layout consists of only the current camera frame and a single button below it. This button allows us to start the calibration process:

```
def _create_custom_layout(self):
    """Creates a horizontal layout with a single button"""
    pnl = wx.Panel(self, -1)
    self.button_calibrate = wx.Button(pnl,
        label='Calibrate Camera')
    self.Bind(wx.EVT_BUTTON, self._on_button_calibrate)
    hbox = wx.BoxSizer(wx.HORIZONTAL)
    hbox.Add(self.button_calibrate)
    pnl.SetSizer(hbox)
```

For these changes to take effect, `pnl` needs to be added to the list of existing panels:

```
    self.panels_vertical.Add(pnl, flag=wx.EXPAND | wx.BOTTOM |
        wx.TOP, border=1)
```

The rest of the visualization pipeline is handled by the `BaseLayout` class. We only need to make sure that we provide the `_init_custom_layout` and `_process_frame` methods.

Initializing the algorithm

In order to perform the calibration process, we need to do some bookkeeping. For now, let's focus on a single 10 x 7 chessboard. The algorithm will detect all the 9 x 6 inner corners of the chessboard (referred to as *object points*) and store the detected image points of these corners in a list. So, let's first initialize the chessboard size to the number of inner corners:

```
    def _init_custom_layout(self):
        """Initializes camera calibration"""
        # setting chessboard size
        self.chessboard_size = (9, 6)
```

Next, we need to enumerate all the object points and assign them object point coordinates so that the first point has coordinates *(0,0)*, the second one (top row) has coordinates *(1,0)*, and the last one has coordinates *(8,5)*:

```
    # prepare object points
    self.objp = np.zeros((np.prod(self.chessboard_size), 3),
        dtype=np.float32)
    self.objp[:, :2] = np.mgrid[0:self.chessboard_size[0],
        0:self.chessboard_size[1]].T.reshape(-1, 2)
```

We also need to keep track of whether we are currently recording the object and image points or not. We will initiate this process once the user clicks on the `self.button_calibrate` button. After that, the algorithm will try to detect a chessboard in all subsequent frames until a number of `self.record_min_num_frames` chessboards have been detected:

```
        # prepare recording
        self.recording = False
        self.record_min_num_frames = 20
        self._reset_recording()
```

Whenever the self.button_calibrate button is clicked on, we reset all the bookkeeping variables, disable the button, and start recording:

```
def _on_button_calibrate(self, event):
    self.button_calibrate.Disable()
    self.recording = True
    self._reset_recording()
```

Resetting the bookkeeping variables involves clearing the lists of recorded object and image points (self.obj_points and self.img_points) as well as resetting the number of detected chessboards (self.recordCnt) to 0:

```
def _reset_recording(self):
    self.record_cnt = 0
    self.obj_points = []
    self.img_points = []
```

Collecting image and object points

The _process_frame method is responsible for doing the hard work of the calibration technique. After the self.button_calibrate button has been clicked on, this method starts collecting data until a total of self.record_min_num_frames chessboards are detected:

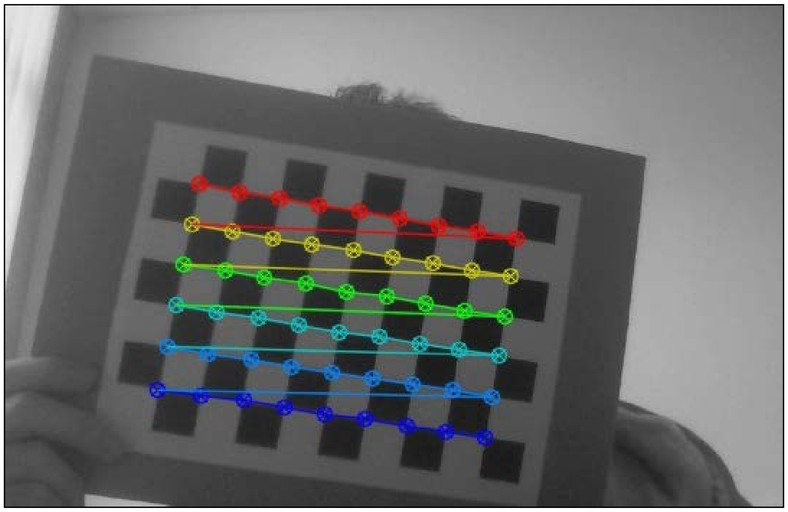
```
def _process_frame(self, frame):
    """Processes each frame"""

    # if we are not recording, just display the frame
    if not self.recording:
        return frame

    # else we're recording
    img_gray = cv2.cvtColor(frame,
        cv2.COLOR_BGR2GRAY).astype(np.uint8)

    if self.record_cnt < self.record_min_num_frames:
        ret, corners = cv2.findChessboardCorners(img_gray,
            self.chessboard_size, None)
```

The cv2.findChessboardCorners function will parse a grayscale image (img_gray) to find a chessboard of size self.chessboard_size. If the image indeed contains a chessboard, the function will return true (ret) as well as a list of chessboard corners (corners).

Then, drawing the chessboard is straightforward:

```
if ret:
    cv2.drawChessboardCorners(frame,
        self.chessboard_size, corners, ret)
```

The result looks like this (drawing the chessboard corners in color for the effect):



We could now simply store the list of detected corners and move on to the next frame. However, in order to make the calibration as accurate as possible, OpenCV provides a function to refine the corner point measurement:

```
criteria = (cv2.TERM_CRITERIA_EPS +
        cv2.TERM_CRITERIA_MAX_ITER, 30, 0.01)
cv2.cornerSubPix(img_gray, corners, (9, 9), (-1, -1),
        criteria)
```

This will refine the coordinates of the detected corners to subpixel precision. Now we are ready to append the object and image points to the list and advance the frame counter:

```
self.obj_points.append(self.objp)
self.img_points.append(corners)
self.record_cnt += 1
```

Finding the camera matrix

Once we have collected enough data (that is, once `self.record_cnt` reaches the value of `self.record_min_num_frames`), the algorithm is ready to perform the calibration. This process can be performed with a single call to `cv2.calibrateCamera`:

```
else:
    print "Calibrating..."
    ret, K, dist, rvecs, tvecs =
        cv2.calibrateCamera(self.obj_points,
            self.img_points, (self.imgHeight, self.imgWidth),
            None, None)
```

The function returns `true` on success (`ret`), the intrinsic camera matrix (`K`), the distortion coefficients (`dist`), as well as two rotation and translation matrices (`rvecs` and `tvecs`). For now, we are mainly interested in the camera matrix and the distortion coefficients, because these will allow us to compensate for any imperfections of the internal camera hardware. We will simply print them on the console for easy inspection:

```
print "K=", K
print "dist=", dist
```

For example, the calibration of my laptop's webcam recovered the following values:

```
K= [[ 3.36696445e+03 0.00000000e+00 2.99109943e+02]
    [ 0.00000000e+00 3.29683922e+03 2.69436829e+02]
    [ 0.00000000e+00 0.00000000e+00 1.00000000e+00]]
dist= [[ 9.87991355e-01 -3.18446968e+02 9.56790602e-02
        -3.42530800e-02 4.87489304e+03]]
```

This tells us that the focal lengths of my webcam are `fx=3366.9644` pixels and `fy=3296.8392` pixels, with the optical center at `cx=299.1099` pixels and `cy=269.4368` pixels.

A good idea might be to double-check the accuracy of the calibration process. This can be done by projecting the object points onto the image using the recovered camera parameters so that we can compare them with the list of image points we collected with the `cv2.findChessboardCorners` function. If the two points are roughly the same, we know that the calibration was successful. Even better, we can calculate the mean error of the reconstruction by projecting every object point in the list:

```
mean_error = 0
for i in xrange(len(self.obj_points)):
    img_points2, _ = cv2.projectPoints(self.obj_points[i],
        rvecs[i], tvecs[i], K, dist)
    error = cv2.norm(self.img_points[i], img_points2,
        cv2.NORM_L2)/len(img_points2)
    mean_error += error

print "mean error=",  {} pixels".format(mean_error)
```

Performing this check on my laptop's webcam resulted in a mean error of 0.95 pixels, which is fairly close to zero.

With the internal camera parameters recovered, we can now set out to take beautiful, undistorted pictures of the world, possibly from different viewpoints so that we can extract some structure from motion.

Setting up the app

Going forward, we will be using a famous open source dataset called `fountain-P11`. It depicts a Swiss fountain viewed from various angles. An example of this is shown in the following image:

The dataset consists of 11 high-resolution images and can be downloaded from http://cvlabwww.epfl.ch/data/multiview/denseMVS.html. Had we taken the pictures ourselves, we would have had to go through the entire camera calibration procedure to recover the intrinsic camera matrix and the distortion coefficients. Luckily, these parameters are known for the camera that took the fountain dataset, so we can go ahead and hardcode these values in our code.

The main function routine

Our main function routine will consist of creating and interacting with an instance of the SceneReconstruction3D class. This code can be found in the chapter4.py file, which imports all the necessary modules and instantiates the class:

```
import numpy as np

from scene3D import SceneReconstruction3D


def main():
    # camera matrix and distortion coefficients
    # can be recovered with calibrate.py
    # but the examples used here are already undistorted, taken
    # with a camera of known K
    K = np.array([[2759.48/4, 0, 1520.69/4, 0, 2764.16/4,
        1006.81/4, 0, 0, 1]]).reshape(3, 3)
    d = np.array([0.0, 0.0, 0.0, 0.0, 0.0]).reshape(1, 5)
```

Here, the K matrix is the intrinsic camera matrix for the camera that took the fountain dataset. According to the photographer, these images are already distortion free, so we set all the distortion coefficients (d) to zero.

> Note that if you want to run the code presented in this chapter on a dataset other than fountain-P11, you will have to adjust the intrinsic camera matrix and the distortion coefficients.

Next, we load a pair of images to which we would like to apply our structure-from-motion techniques. I downloaded the dataset into a subdirectory called fountain_dense:

```
# load a pair of images for which to perform SfM
scene = SceneReconstruction3D(K, d)
scene.load_image_pair("fountain_dense/0004.png",
    "fountain_dense/0005.png")
```

Now we are ready to perform various computations, such as the following:

```
scene.plot_optic_flow()
scene.draw_epipolar_lines()
scene.plot_rectified_images()

# draw 3D point cloud of fountain
# use "pan axes" button in pyplot to inspect the cloud (rotate
# and zoom to convince you of the result)
scene.plot_point_cloud()
```

The next sections will explain these functions in detail.

The SceneReconstruction3D class

All of the relevant 3D scene reconstruction code for this chapter can be found as part of the `SceneReconstruction3D` class in the `scene3D` module. Upon instantiation, the class stores the intrinsic camera parameters to be used in all subsequent calculations:

```
import cv2
import numpy as np
import sys

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


class SceneReconstruction3D:
    def __init__(self, K, dist):
        self.K = K
        self.K_inv = np.linalg.inv(K)
        self.d = dist
```

Then, the first step is to load a pair of images on which to operate:

```
def load_image_pair(self, img_path1, img_path2,
        downscale=True):
    self.img1 = cv2.imread(img_path1, cv2.CV_8UC3)
    self.img2 = cv2.imread(img_path2, cv2.CV_8UC3)

    # make sure images are valid
    if self.img1 is None:
        sys.exit("Image " + img_path1 + " could not be
            loaded.")
    if self.img2 is None:
        sys.exit("Image " + img_path2 + " could not be
            loaded.")
```

If the loaded images are grayscale, the method will convert to them to BGR format, because the other methods expect a three-channel image:

```
if len(self.img1.shape)==2:
    self.img1 = cv2.cvtColor(self.img1, cv2.COLOR_GRAY2BGR)
    self.img2 = cv2.cvtColor(self.img2, cv2.COLOR_GRAY2BGR)
```

In the case of the fountain sequence, all images are of a relatively high resolution. If an optional `downscale` flag is set, the method will downscale the images to a width of roughly 600 pixels:

```
# scale down image if necessary
# to something close to 600px wide
target_width = 600
if downscale and self.img1.shape[1]>target_width:
    while self.img1.shape[1] > 2*target_width:
        self.img1 = cv2.pyrDown(self.img1)
        self.img2 = cv2.pyrDown(self.img2)
```

Also, we need to compensate for the radial and tangential lens distortions using the distortion coefficients specified earlier (if there are any):

```
self.img1 = cv2.undistort(self.img1, self.K, self.d)
self.img2 = cv2.undistort(self.img2, self.K, self.d)
```

Finally, we are ready to move on to the meat of the project—estimating the camera motion and reconstructing the scene!

Estimating the camera motion from a pair of images

Now that we have loaded two images (`self.img1` and `self.img2`) of the same scene, such as two examples from the fountain dataset, we find ourselves in a similar situation as in the last chapter. We are given two images that supposedly show the same rigid object or static scene, but from different viewpoints. However, this time we want to go a step further; if the only thing that changes between taking the two pictures is the location of the camera, can we infer the relative camera motion by looking at the matching features?

Well, of course we can. Otherwise, this chapter would not make much sense, would it? We will take the location and orientation of the camera in the first image as a given and then find out how much we have to reorient and relocate the camera so that its viewpoint matches that from the second image.

In other words, we need to recover the **essential matrix** of the camera in the second image. An essential matrix is a 4 x 3 matrix that is the concatenation of a 3 x 3 rotation matrix and a 3 x 1 translation matrix. It is often denoted as [R | t]. You can think of it as capturing the position and orientation of the camera in the second image relative to the camera in the first image.

The crucial step in recovering the essential matrix (as well as all other transformations in this chapter) is feature matching. We can either reuse our code from the last chapter and apply a speeded-up robust features (SURF) detector to the two images, or calculate the optic flow between the two images. The user may choose their favorite method by specifying a feature extraction mode, which will be implemented by the following private method:

```
def ___extract_keypoints(self, feat_mode):
    if featMode == "SURF":
        # feature matching via SURF and BFMatcher
        self._extract_keypoints_surf()
    else:
        if feat_mode == "flow":
            # feature matching via optic flow
            self._extract_keypoints_flow()
        else:
            sys.exit("Unknown mode " + feat_mode
                + ". Use 'SURF' or 'FLOW'")
```

Point matching using rich feature descriptors

As we saw in the last chapter, a fast and robust way of extracting important features from an image is by using a SURF detector. In this chapter, we want to use it for two images, `self.img1` and `self.img2`:

```
def _extract_keypoints_surf(self):
    detector = cv2.SURF(250)
    first_key_points, first_des =
        detector.detectAndCompute(self.img1, None)
    second_key_points, second_desc =
        detector.detectAndCompute(self.img2, None)
```

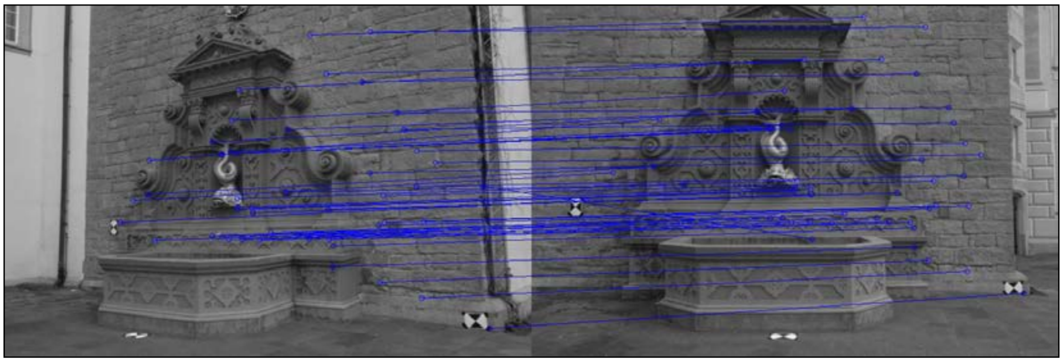
For feature matching, we will use a `BruteForce` matcher, but other matchers (such as FLANN) can work as well:

```
matcher = cv2.BFMatcher(cv2.NORM_L1, True)
matches = matcher.match(first_desc, second_desc)
```

For each of the matches, we need to recover the corresponding image coordinates. These are maintained in the `self.match_pts1` and `self.match_pts2` lists:

```
first_match_points = np.zeros((len(matches), 2),
    dtype=np.float32)
second_match_points = np.zeros_like(first_match_points)
for i in range(len(matches)):
    first_match_points[i] =
        first_key_points[matches[i].queryIdx].pt
    second_match_points[i] =
        second_key_points[matches[i].trainIdx].pt

self.match_pts1 = first_match_points
self.match_pts2 = second_match_points
```

The following screenshot shows an example of the feature matcher applied to two arbitrary frames of the fountain sequence:

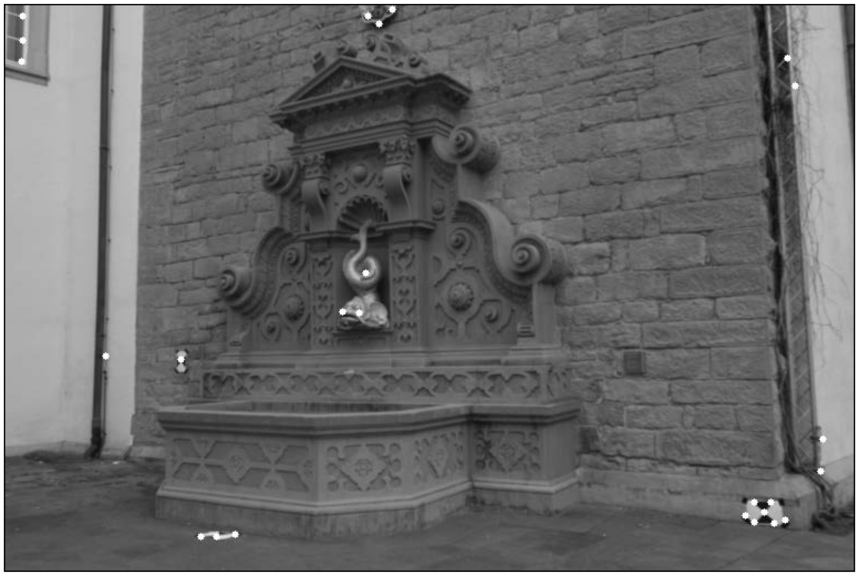


Point matching using optic flow

An alternative to using rich features, is using optic flow. Optic flow is the process of estimating motion between two consecutive image frames by calculating a displacement vector. A displacement vector can be calculated for every pixel in the image (dense) or only for selected points (sparse).

One of the most commonly used techniques for calculating dense optic flow is the **Lukas-Kanade** method. It can be implemented in OpenCV with a single line of code, by using the `cv2.calcOpticalFlowPyrLK` function.

But before that, we need to select some points in the image that are worth tracking. Again, this is a question of feature selection. If we were interested in getting an exact result for only a few highly salient image points, we can use Shi-Tomasi's `cv2.goodFeaturesToTrack` function. This function might recover features like this:



However, in order to infer structure from motion, we might need many more features and not just the most salient Harris corners. An alternative would be to detect the FAST features:

```
def _extract_keypoints_flow(self):
    fast = cv2.FastFeatureDetector()
    first_key_points = fast.detect(self.img1, None)
```

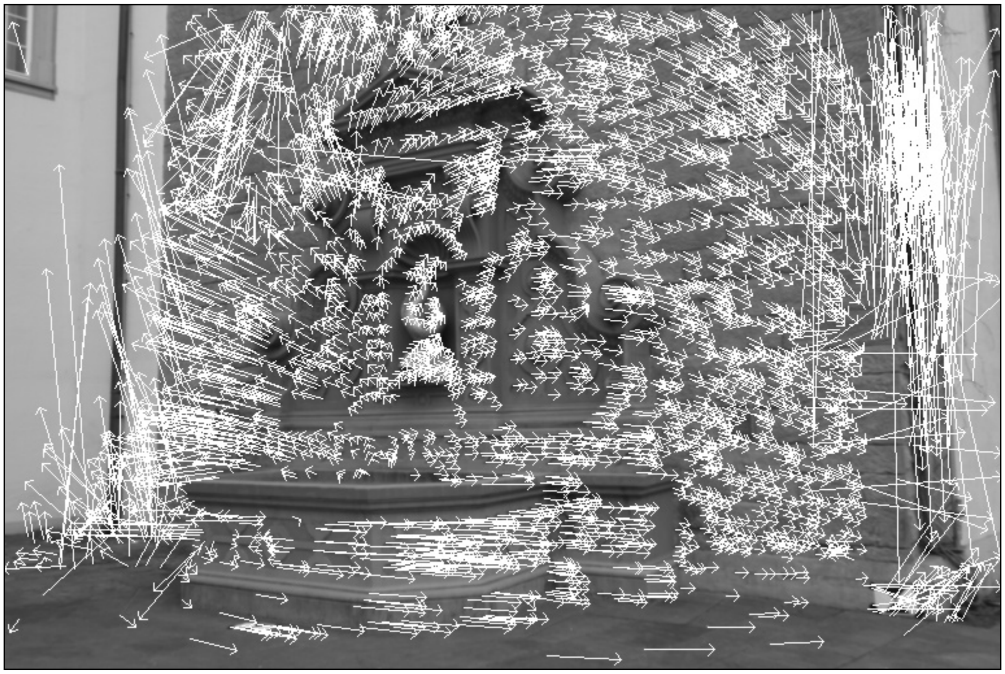
We can then calculate the optic flow for these features. In other words, we want to find the points in the second image that most likely correspond to the `first_key_points` from the first image. For this, we need to convert the keypoint list into a NumPy array of `(x,y)` coordinates:

```
first_key_list = [i.pt for i in first_key_points]
first_key_arr = np.array(first_key_list).astype(np.float32)
```

Then the optic flow will return a list of corresponding features in the second image (`second_key_arr`):

```
second_key_arr, status, err =
    cv2.calcOpticalFlowPyrLK(self.img1, self.img2,
        first_key_arr)
```

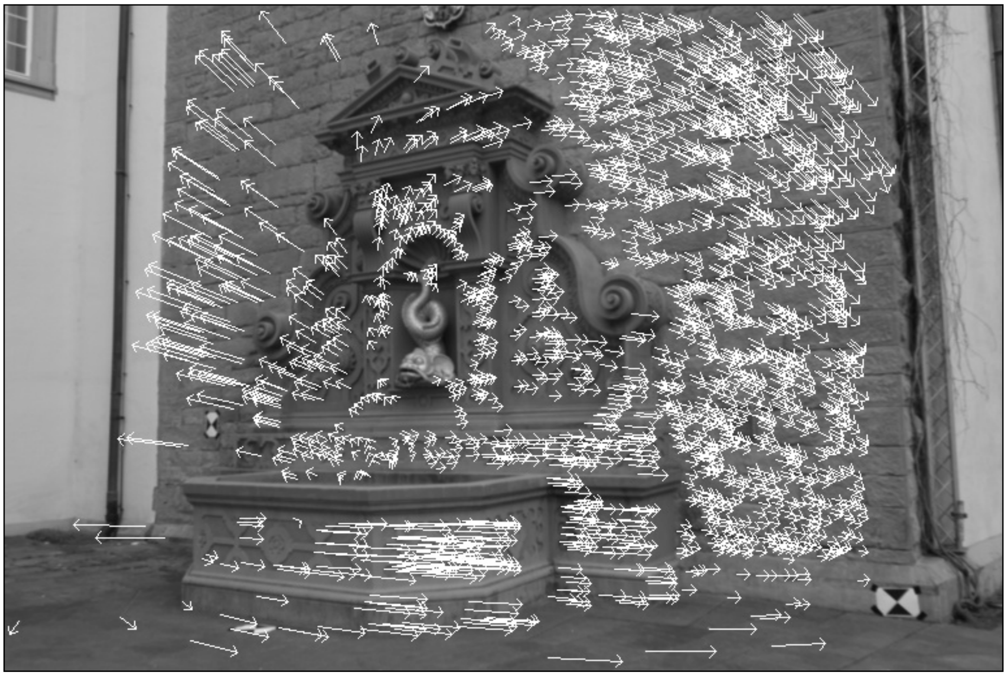
The function also returns a vector of status bits (`status`), which indicate whether the flow for a keypoint has been found or not, and a vector of estimated error values (`err`). If we were to ignore these two additional vectors, the recovered flow field could look something like this:



In this image, an arrow is drawn for each keypoint, starting at the image location of the keypoint in the first image and pointing to the image location of the same keypoint in the second image. By inspecting the flow image, we can see that the camera moved mostly to the right, but there also seems to be a rotational component. However, some of these arrows are really large, and some of them make no sense. For example, it is very unlikely that a pixel in the bottom-right image corner actually moved all the way to the top of the image. It is much more likely that the flow calculation for this particular keypoint is wrong. Thus, we want to exclude all the keypoints for which the status bit is zero or the estimated error is larger than some value:

```
condition = (status == 1) * (err < 5.)
concat = np.concatenate((condition, condition), axis=1)
first_match_points = first_key_arr[concat].reshape(-1, 2)
second_match_points = second_key_arr[concat].reshape(-1, 2)

self.match_pts1 = first_match_points
self.match_pts2 = second_match_points
```

If we draw the flow field again with a limited set of keypoints, the image will look like this:



The flow field can be drawn with the following public method, which first extracts the keypoints using the preceding code and then draws the actual lines on the image:

```
def plot_optic_flow(self):
    self._extract_key_points("flow")

    img = self.img1
    for i in xrange(len(self.match_pts1)):
        cv2.line(img, tuple(self.match_pts1[i]),
            tuple(self.match_pts2[i]), color=(255, 0, 0))
        theta = np.arctan2(self.match_pts2[i][1] -
            self.match_pts1[i][1], self.match_pts2[i][0] -
            self.match_pts1[i][0])
        cv2.line(img, tuple(self.match_pts2[i]),
             (np.int(self.match_pts2[i][0] -
             6*np.cos(theta+np.pi/4)),
             np.int(self.match_pts2[i][1] -
             6*np.sin(theta+np.pi/4))), color=(255, 0, 0))
        cv2.line(img, tuple(self.match_pts2[i]),
            (np.int(self.match_pts2[i][0] -
            6*np.cos(theta-np.pi/4)),
            np.int(self.match_pts2[i][1] -
            6*np.sin(theta-np.pi/4))), color=(255, 0, 0))
    for i in xrange(len(self.match_pts1)):
        cv2.line(img, tuple(self.match_pts1[i]),
            tuple(self.match_pts2[i]), color=(255, 0, 0))
        theta = np.arctan2(self.match_pts2[i][1] -
            self.match_pts1[i][1],
            self.match_pts2[i][0] - self.match_pts1[i][0])
    cv2.imshow("imgFlow",img)
    cv2.waitKey()
```

The advantage of using optic flow instead of rich features is that the process is usually faster and can accommodate the matching of many more points, making the reconstruction denser.

The caveat in working with optic flow is that it works best for consecutive images taken by the same hardware, whereas rich features are mostly agnostic to this.

Finding the camera matrices

Now that we have obtained the matches between keypoints, we can calculate two important camera matrices: the fundamental matrix and the essential matrix. These matrices will specify the camera motion in terms of rotational and translational components. Obtaining the fundamental matrix (`self.F`) is another OpenCV one-liner:

```
    def _find_fundamental_matrix(self):
        self.F, self.Fmask = cv2.findFundamentalMat(self.match_pts1,
            self.match_pts2, cv2.FM_RANSAC, 0.1, 0.99)
```

The only difference between the fundamental matrix and the essential matrix is that the latter operates on rectified images:

```
    def _find_essential_matrix(self):
        self.E = self.K.T.dot(self.F).dot(self.K)
```

The essential matrix (`self.E`) can then be decomposed into rotational and translational components, denoted as [R | t], using **singular value decomposition** (**SVD**):

```
    def _find_camera_matrices(self):
        U, S, Vt = np.linalg.svd(self.E)
        W = np.array([0.0, -1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            1.0]).reshape(3, 3)
```

Using the unitary matrices *U* and *V* in combination with an additional matrix, *W*, we can now reconstruct [R | t]. However, it can be shown that this decomposition has four possible solutions and only one of them is the valid second camera matrix. The only thing we can do is check all four possible solutions and find the one that predicts that all the imaged keypoints lie in front of both cameras.

But prior to that, we need to convert the keypoints from 2D image coordinates to homogeneous coordinates. We achieve this by adding a *z* coordinate, which we set to 1:

```
first_inliers = []
second_inliers = []
for i in range(len(self.Fmask)):
    if self.Fmask[i]:
        first_inliers.append(self.K_inv.dot(
            [self.match_pts1[i][0], self.match_pts1[i][1],
            1.0]))
        second_inliers.append(self.K_inv.dot(
            [self.match_pts2[i][0], self.match_pts2[i][1],
            1.0]))
```

We then iterate over the four possible solutions and choose the one that has `_in_front_of_both_cameras` returning `True`:

```
# First choice: R = U * Wt * Vt, T = +u_3 (See Hartley
# & Zisserman 9.19)
R = U.dot(W).dot(Vt)
T = U[:, 2]

if not self._in_front_of_both_cameras(first_inliers,
        second_inliers, R, T):
    # Second choice: R = U * W * Vt, T = -u_3
    T = - U[:, 2]

if not self._in_front_of_both_cameras(first_inliers,
        second_inliers, R, T):
    # Third choice: R = U * Wt * Vt, T = u_3
    R = U.dot(W.T).dot(Vt)
    T = U[:, 2]

if not self._in_front_of_both_cameras(first_inliers,
        second_inliers, R, T):
    # Fourth choice: R = U * Wt * Vt, T = -u_3
    T = - U[:, 2]
```

Now, we can finally construct the [R | t] matrices of the two cameras. The first camera is simply a canonical camera (no translation and no rotation):

```
self.Rt1 = np.hstack((np.eye(3), np.zeros((3, 1))))
```

The second camera matrix consists of [R | t] recovered earlier:

```
self.Rt2 = np.hstack((R, T.reshape(3, 1)))
```

The `__InFrontOfBothCameras` private method is a helper function that makes sure that every pair of keypoints is mapped to 3D coordinates that make them lie in front of both cameras:

```
def _in_front_of_both_cameras(self, first_points, second_points,
        rot, trans):
    rot_inv = rot
    for first, second in zip(first_points, second_points):
        first_z = np.dot(rot[0, :] - second[0]*rot[2, :], trans) /
            np.dot(rot[0, :] - second[0]*rot[2, :], second)
        first_3d_point = np.array([first[0] * first_z, second[0] *
            first_z, first_z])
        second_3d_point = np.dot(rot.T, first_3d_point) -
            np.dot(rot.T, trans)
```

If the function finds any keypoint that is not in front of both cameras, it will return `False`:

```
if first_3d_point[2] < 0 or second_3d_point[2] < 0:
    return False
return True
```

Image rectification

Maybe, the easiest way to make sure that we have recovered the correct camera matrices is to rectify the images. If they are rectified correctly, then; a point in the first image, and a point in the second image that correspond to the same 3D world point, will lie on the same vertical coordinate. In a more concrete example, such as in our case, since we know that the cameras are upright, we can verify that horizontal lines in the rectified image correspond to horizontal lines in the 3D scene.

First, we perform all the steps described in the previous subsections to obtain the [R | t] matrix of the second camera:

```
def plot_rectified_images(self, feat_mode="SURF"):
    self._extract_keypoints(feat_mode)
    self._find_fundamental_matrix()
    self._find_essential_matrix()
```

```
        self._find_camera_matrices_rt()

        R = self.Rt2[:, :3]
        T = self.Rt2[:, 3]
```

Then, rectification can be performed with two OpenCV one-liners that remap the image coordinates to the rectified coordinates based on the camera matrix (`self.K`), the distortion coefficients (`self.d`), the rotational component of the essential matrix (`R`), and the translational component of the essential matrix (`T`):

```
        R1, R2, P1, P2, Q, roi1, roi2 = cv2.stereoRectify(self.K,
            self.d, self.K, self.d, self.img1.shape[:2], R, T,
            alpha=1.0)
        mapx1, mapy1 = cv2.initUndistortRectifyMap(self.K,
            self.d, R1, self.K, self.img1.shape[:2], cv2.CV_32F)
        mapx2, mapy2 = cv2.initUndistortRectifyMap(self.K, self.d, R2,
            self.K, self.img2.shape[:2], cv2.CV_32F)
        img_rect1 = cv2.remap(self.img1, mapx1, mapy1,
            cv2.INTER_LINEAR)
        img_rect2 = cv2.remap(self.img2, mapx2, mapy2,
            cv2.INTER_LINEAR)
```

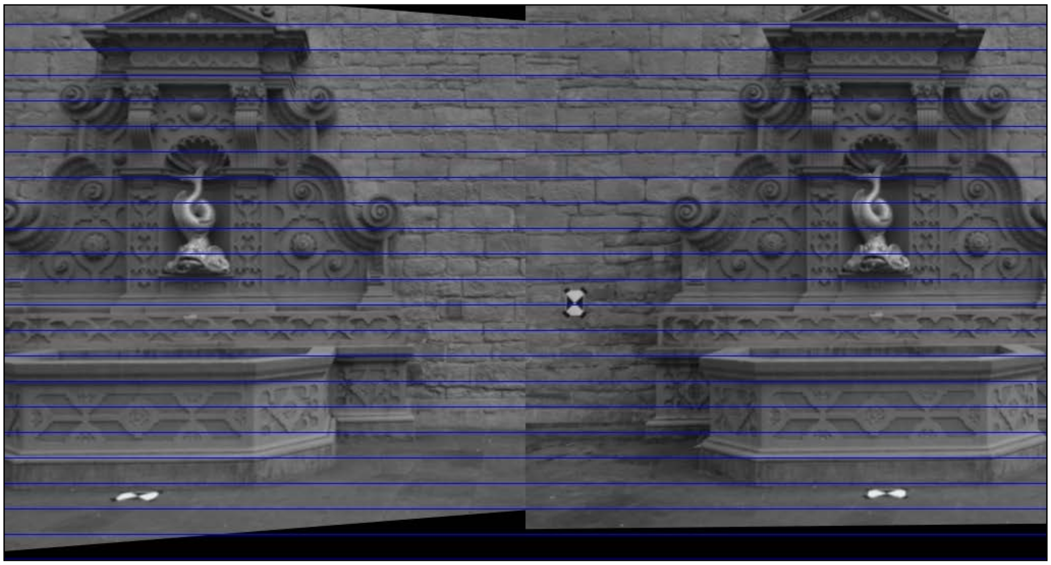
To make sure that the rectification is accurate, we plot the two rectified images (`img_rect1` and `img_rect2`) next to each other:

```
    total_size = (max(img_rect1.shape[0], img_rect2.shape[0]),
        img_rect1.shape[1] + img_rect2.shape[1], 3)
    img = np.zeros(total_size, dtype=np.uint8)
    img[:img_rect1.shape[0], :img_rect1.shape[1]] = img_rect1
    img[:img_rect2.shape[0], img_rect1.shape[1]:] = img_rect2
```

We also draw horizontal blue lines after every 25 pixels, across the side-by-side images to further help us visually investigate the rectification process:

```
    for i in range(20, img.shape[0], 25):
        cv2.line(img, (0, i), (img.shape[1], i), (255, 0, 0))
        cv2.imshow('imgRectified', img)
```

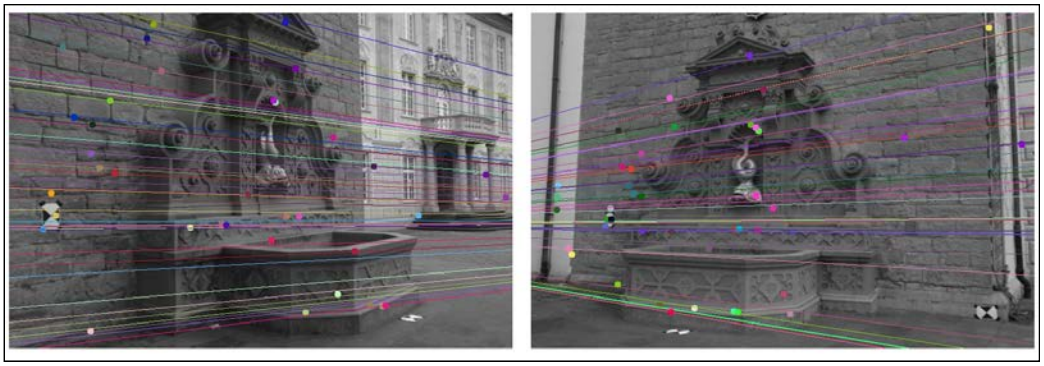
Now we can easily convince ourselves that the rectification was successful, as shown here:



Reconstructing the scene

Finally, we can reconstruct the 3D scene by making use of a process called **triangulation**. We are able to infer the 3D coordinates of a point because of the way **epipolar geometry** works. By calculating the essential matrix, we get to know more about the geometry of the visual scene than we might think. Because the two cameras depict the same real-world scene, we know that most of the 3D real-world points will be found in both images. Moreover, we know that the mapping from the 2D image points to the corresponding 3D real-world points, will follow the rules of geometry. If we study a sufficiently large number of image points, we can construct, and solve, a (large) system of linear equations to get the ground truth of the real-world coordinates.

Let's return to the Swiss fountain dataset. If we ask two photographers to take a picture of the fountain from different viewpoints at the same time, it is not hard to realize that the first photographer might show up in the picture of the second photographer, and vice-versa. The point on the image plane where the other photographer is visible is called the **epipole** or **epipolar point**. In more technical terms, the epipole is the point on one camera's image plane onto which the center of projection of the other camera projects. It is interesting to note that both the epipoles in their respective image planes, and both the centers of projection, lie on a single 3D line. By looking at the lines between the epipoles and image points, we can limit the number of possible 3D coordinates of the image points. In fact, if the projection point is known, then the epipolar line (which is the line between the image point and the epipole) is known, and in turn the same point projected onto the second image must lie on that particular epipolar line. Confusing? I thought so. Let's just look at these images:



Each line here is the epipolar line of a particular point in the image. Ideally, all the epipolar lines drawn in the left-hand-side image should intersect at a point, and that point typically lies outside the image. If the calculation is accurate, then that point should coincide with the location of the second camera as seen from the first camera. In other words, the epipolar lines in the left-hand-side image tell us that the camera that took the right-hand-side image is located to our (that is, the first camera's) right-hand side. Analogously, the epipolar lines in the right-hand-side image tell us that the camera that took the image on the left is located to our (that is, the second camera's) left-hand side.

Moreover, for each point observed in one image, the same point must be observed in the other image on a known epipolar line. This is known as **epipolar constraint**. We can use this fact to show that if two image points correspond to the same 3D point, then the projection lines of those two image points must intersect precisely at the 3D point. This means that the 3D point can be calculated from two image points, which is what we are going to do next.

Luckily, OpenCV again provides a wrapper to solve an extensive set of linear equations. First, we have to convert our list of matching feature points into a NumPy array:

```
first_inliers = np.array(self.match_inliers1).reshape
    (-1, 3)[:, :2]
second_inliers = np.array(self.match_inliers2).reshape
    (-1, 3)[:, :2]
```

Triangulation is performed next, using the preceding two [R | t] matrices (`self.Rt1` for the first camera and `self.Rt2` for the second camera):

```
pts4D = cv2.triangulatePoints(self.Rt1, self.Rt2, first_inliers.T,
    second_inliers.T).T
```

This will return the triangulated real-world points using 4D homogeneous coordinates. To convert them to 3D coordinates, we need to divide the *(X,Y,Z)* coordinates by the fourth coordinate, usually referred to as *W*:

```
pts3D = pts4D[:, :3]/np.repeat(pts4D[:, 3], 3).reshape(-1, 3)
```

3D point cloud visualization

The last step is visualizing the triangulated 3D real-world points. An easy way of creating 3D scatterplots is by using matplotlib. However, if you are looking for more professional visualization tools, you may be interested in Mayavi (`http://docs.enthought.com/mayavi/mayavi`), VisPy (`http://vispy.org`), or the Point Cloud Library (`http://pointclouds.org`). Although the latter does not have Python support for point cloud visualization yet, it is an excellent tool for point cloud segmentation, filtering, and sample consensus model fitting. For more information, head over to strawlab's GitHub repository at `https://github.com/strawlab/python-pcl`.

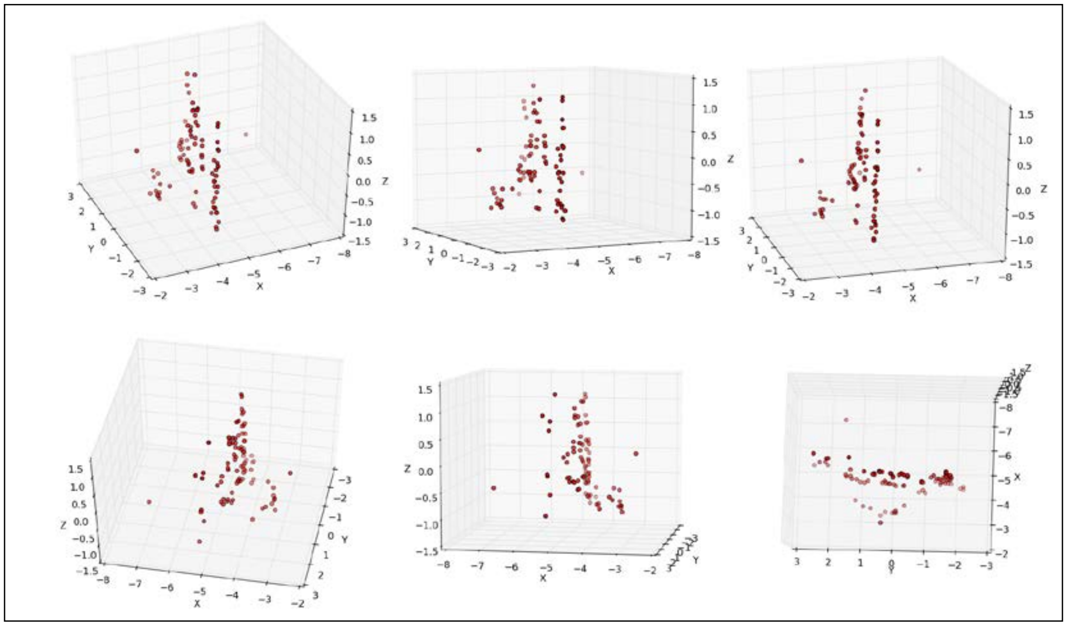
Before we can plot our 3D point cloud, we obviously have to extract the [R | t] matrix and perform the triangulation as explained earlier:

```
def plot_point_cloud(self, feat_mode="SURF"):
    self._extract_keypoints(feat_mode)
    self._find_fundamental_matrix()
    self._find_essential_matrix()
    self._find_camera_matrices_rt()

    # triangulate points
    first_inliers = np.array(
        self.match_inliers1).reshape(-1, 3)[:, :2]
    second_inliers = np.array(
        self.match_inliers2).reshape(-1, 3)[:, :2]
    pts4D = cv2.triangulatePoints(self.Rt1, self.Rt2,
        first_inliers.T, second_inliers.T).T

    # convert from homogeneous coordinates to 3D
    pts3D = pts4D[:, :3]/np.repeat(pts4D[:, 3], 3).reshape(-1, 3)
```

Then, all we need to do is open a `matplotlib` figure and draw each entry of `pts3D` in a 3D scatterplot:

```
        Ys = pts3D[:,0]
        Zs = pts3D[:,1]
        Xs = pts3D[:,2]

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(Xs, Ys, Zs, c='r', marker='o')
        ax.set_xlabel('Y')
        ax.set_ylabel('Z')
        ax.set_zlabel('X')
        plt.show()
```

The result is most compelling when studied using pyplot's **Pan axes** button, which lets you rotate and scale the point cloud in all three dimensions. This will make it immediately clear that most of the points that you see lie on the same plane, namely the wall behind the fountain, and that the fountain itself extends from that wall in negative *z* coordinates. It is a little harder to draw this convincingly, but here we go:



Each subplot shows the recovered 3D coordinates of the fountain as seen from a different angle. In the top row, we are looking at the fountain from a similar angle as the second camera in the previous images, that is, by standing to the right and slightly in front of the fountain. You can see how most of the points are mapped to a similar *x* coordinate, which corresponds to the wall behind the fountain. For a subset of points concentrated between *z* coordinates *-0.5* and *-1.0*, the *x* coordinate is significantly different, which shows different keypoints that belong to the surface of the fountain. The first two panels in the lower row look at the fountain from the other side. The last panel shows a birds-eye view of the fountain, highlighting the fountain's silhouette as a half-circle in the lower half of the image.

Summary

In this chapter, we explored a way of reconstructing a scene in 3D—by inferring the geometrical features of 2D images taken by the same camera. We wrote a script to calibrate a camera, and you learned about fundamental and essential matrices. We used this knowledge to perform triangulation. We then went on to visualize the real-world geometry of the scene in a 3D point cloud. Using simple 3D scatterplots in matplotlib, we found a way to convince ourselves that our calculations were accurate and practical.

Going forward from here, it will be possible to store the triangulated 3D points in a file that can be parsed by the Point Cloud Library, or to repeat the procedure for different image pairs so that we can generate a denser and more accurate reconstruction. Although we have covered a lot in this chapter, there is a lot more left to do. Typically, when talking about a structure-from-motion pipeline, we include two additional steps that we have not talked about so far: bundle adjustment and geometry fitting. One of the most important steps in such a pipeline is to refine the 3D estimate in order to minimize reconstruction errors. Typically, we would also want to get all points that do not belong to our object of interest out of the cloud. But with the basic code in hand, you can now go ahead and write your own advanced structure-from-motion pipeline!

In the next chapter, we will move away from rigid scenes and instead focus on tracking visually salient and moving objects in a scene. This will give you an understanding of how to deal with non-static scenes. We will also explore how we can make an algorithm focus on *what's important* in a scene, quickly, which is a technique known to speed up object detection, object recognition, object tracking, and content-aware image editing.

5
Tracking Visually Salient Objects

The goal of this chapter is to track multiple visually salient objects in a video sequence at once. Instead of labeling the objects of interest in the video ourselves, we will let the algorithm decide which regions of a video frame are worth tracking.

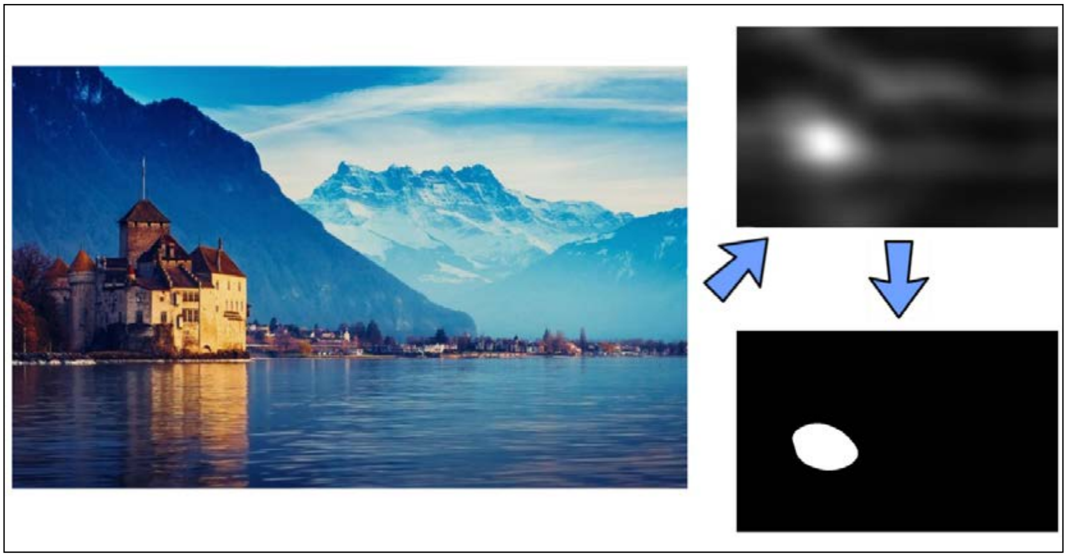
We have previously learned how to detect simple objects of interest (such as a human hand) in tightly controlled scenarios or how to infer geometrical features of a visual scene from camera motion. In this chapter, we ask what we can learn about a visual scene by looking at the **image statistics** of a large number of frames. By analyzing the **Fourier spectrum** of natural images we will build a **saliency map**, which allows us to label certain statistically interesting patches of the image as (potential or) *proto-objects*. We will then feed the location of all the proto- objects to a **mean-shift tracker** that will allow us to keep track of where the objects move from one frame to the next.

To build the app, we need to combine the following two main features:

- **Saliency map**: We will use Fourier analysis to get a general understanding of natural image statistics, which will help us build a model of what general image backgrounds look like. By comparing and contrasting the background model to a specific image frame, we can locate sub-regions of the image that *pop out* of their surroundings. Ideally, these sub-regions correspond to the image patches that tend to grab our immediate attention when looking at the image.
- **Object tracking**: Once all the potentially *interesting* patches of an image are located, we will track their movement over many frames using a simple yet effective method called mean-shift tracking. Because it is possible to have multiple proto-objects in the scene that might change appearance over time, we need to be able to distinguish between them and keep track of all of them.

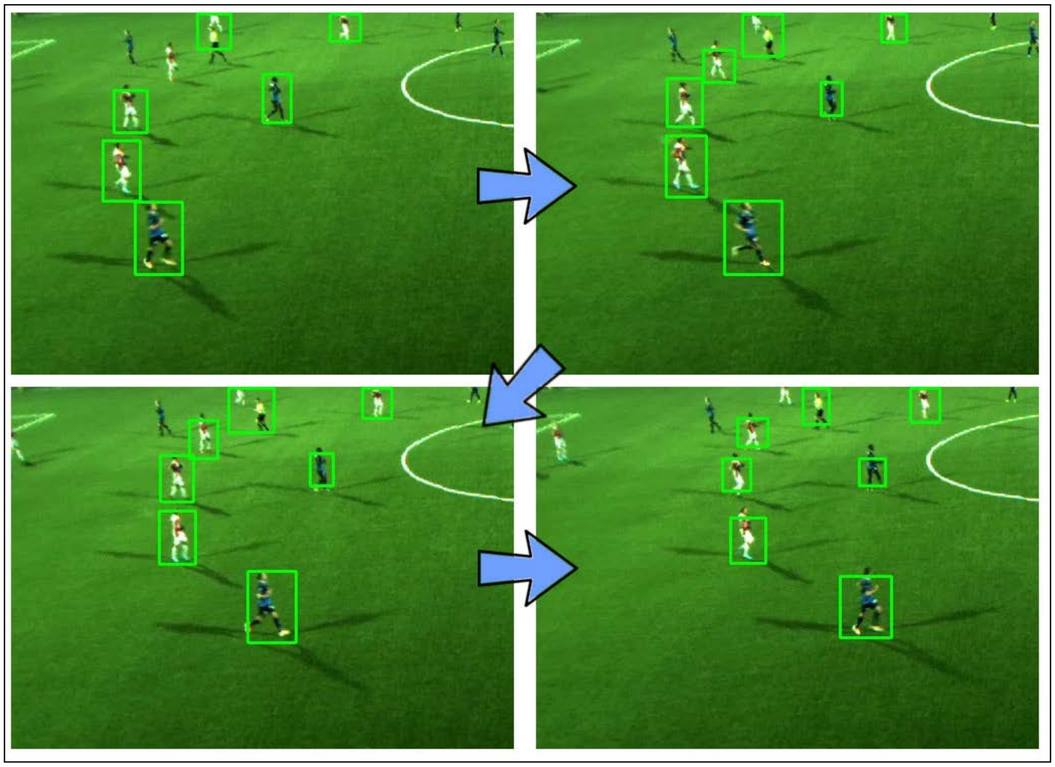
Visual saliency is a technical term from cognitive psychology that tries to describe the visual quality of certain objects or items that allows them to grab our immediate attention. Our brains constantly drive our gaze towards the *important* regions of the visual scene and keep track of them over time, allowing us to quickly scan our surroundings for interesting objects and events while neglecting the less important parts.

An example of a regular RGB image and its conversion to a saliency map, where the statistically interesting *pop-out* regions appear bright and the others dark, is shown in the following figure:



Traditional models might try to associate particular features with each target (much like our feature matching approach in *Chapter 3*, *Finding Objects via Feature Matching and Perspective Transforms*), which would convert the problem to the detection of specific categories or objects. However, these models require manual labeling and training. But what if the features or the number of the objects to track is not known?

Instead, we will try to mimic what the brain does, that is, tune our algorithm to the statistics of the natural images, so that we can immediately locate the patterns or sub-regions that "grab our attention" in the visual scene (that is, patterns that deviate from these statistical regularities) and flag them for further inspection. The result is an algorithm that works for any number of proto-objects in the scene, such as tracking all the players on a soccer field. Refer to the following image:



This chapter uses OpenCV 2.4.9, as well as the additional packages NumPy (`http://www.numpy.org`), wxPython 2.8 (`http://www.wxpython.org/download.php`), and matplotlib (`http://www.matplotlib.org/downloads.html`). Although parts of the algorithms presented in this chapter have been added to an optional Saliency module of the OpenCV 3.0.0 release, there is currently no Python API for it, so we will write our own code.

Planning the app

The final app will convert each RGB frame of a video sequence into a saliency map, extract all the interesting proto-objects, and feed them to a mean-shift tracking algorithm. To do this, we need the following components:

- `main`: The main function routine (in `chapter5.py`) to start the application.
- `Saliency`: A class that generates a saliency map from an RGB color image. It includes the following public methods:
    - `Saliency.get_saliency_map`: The main method to convert an RGB color image to a saliency map
    - `Saliency.get_proto_objects_map`: A method to convert a saliency map into a binary mask containing all the proto-objects
    - `Saliency.plot_power_density`: A method to display the 2D power density of an RGB color image, which is helpful to understand the Fourier transform
    - `Saliency.plot_power_spectrum`: A method to display the radially averaged power spectrum of an RGB color image, which is helpful to understand natural image statistics
- `MultiObjectTracker`: A class that tracks multiple objects in a video using mean-shift tracking. It includes the following public method, which itself contains a number of private helper methods:
    - `MultiObjectTracker.advance_frame`: A method to update the tracking information for a new frame, combining bounding boxes obtained from both the saliency map and mean-shift tracking

In the following sections, we will discuss these steps in detail.

Setting up the app

In order to run our app, we will need to execute a main function routine that reads a frame of a video stream, generates a saliency map, extracts the location of the proto-objects, and tracks these locations from one frame to the next.

The main function routine

The main process flow is handled by the main function in `chapter5.py`, which instantiates the two classes (`Saliency` and `MultipleObjectTracker`) and opens a video file showing the number of soccer players on the field:

```
import cv2
import numpy as np
from os import path

from saliency import Saliency
from tracking import MultipleObjectsTracker


def main(video_file='soccer.avi', roi=((140, 100), (500, 600))):
    if path.isfile(video_file):
        video = cv2.VideoCapture(video_file)
    else:
        print 'File "' + video_file + '" does not exist.'
        raise SystemExit

    # initialize tracker
    mot = MultipleObjectsTracker()
```

The function will then read the video frame by frame, extract some meaningful region of interest (for illustration purposes), and feed it to the Saliency module:

```
    while True:
        success, img = video.read()
        if success:
            if roi:
                # grab some meaningful ROI
                img = img[roi[0][0]:roi[1][0],
                    roi[0][1]:roi[1][1]]
            # generate saliency map
            sal = Saliency(img, use_numpy_fft=False,
                gauss_kernel=(3, 3))
```

The Saliency will generate a map of all the *interesting* proto-objects and feed that into the tracker module. The output of the tracker module is the input frame annotated with bounding boxes as shown in the preceding figure.

```
cv2.imshow("tracker", mot.advance_frame(img,
            sal.get_proto_objects_map(use_otsu=False)))
```

The app will run through all the frames of the video until the end of the file is reached or the user presses the *q* key:

```
if cv2.waitKey(100) & 0xFF == ord('q'):
    break
```

The Saliency class

The constructor of the Saliency class accepts a video frame, which can be either grayscale or RGB, as well as some options such as whether to use NumPy's or OpenCV's Fourier package:

```
def __init__(self, img, use_numpy_fft=True, gauss_kernel=(5, 5)):
    self.use_numpy_fft = use_numpy_fft
    self.gauss_kernel = gauss_kernel
    self.frame_orig = img
```

A saliency map will be generated from a down sampled version of the image, and because the computation is relatively time-intensive, we will maintain a flag `need_saliency_map` that makes sure we do the computations only once:

```
    self.small_shape = (64, 64)
    self.frame_small = cv2.resize(img, self.small_shape[1::-1])

    # whether we need to do the math (True) or it has already
    # been done (False)
    self.need_saliency_map = True
```

From then on, the user may call any of the class' public methods, which will all be passed on the same image.

The MultiObjectTracker class

The constructor of the tracker class is straightforward. All it does is set up the termination criteria for mean-shift tracking and store the conditions for the minimum contour area (`min_area`) and minimum frame-by-frame drift (`min_shift2`) to be considered in the subsequent computation steps:

```
    def __init__(self, min_area=400, min_shift2=5):
        self.object_roi = []
        self.object_box = []

        self.min_cnt_area = min_area
        self.min_shift2 = min_shift2

        # Setup the termination criteria, either 100 iteration or move
        # by at least 1 pt
        self.term_crit = (cv2.TERM_CRITERIA_EPS |
            cv2.TERM_CRITERIA_COUNT, 100, 1)
```

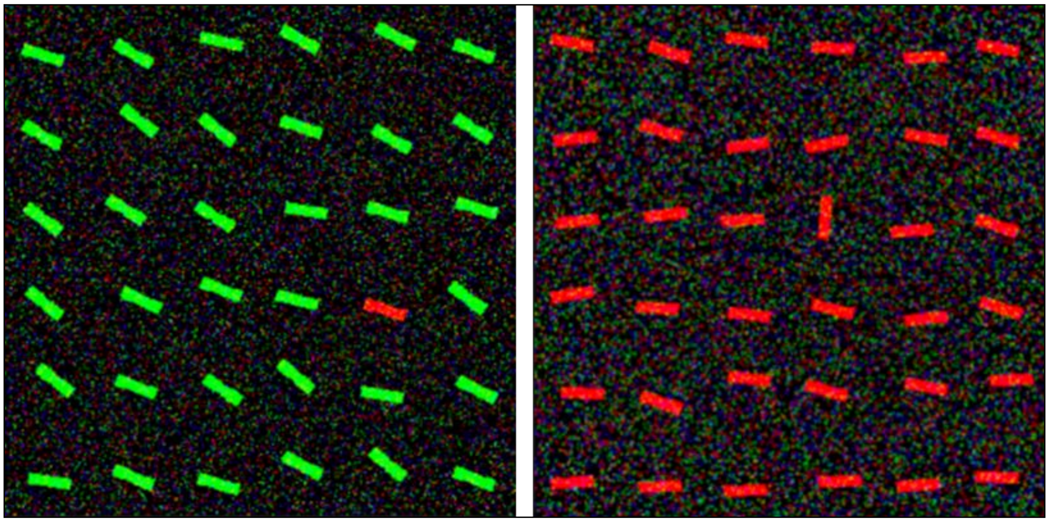
From then on, the user may call the `advance_frame` method to feed a new frame to the tracker.

However, before we make use of all this functionality, we need to learn about image statistics and how to generate a saliency map.

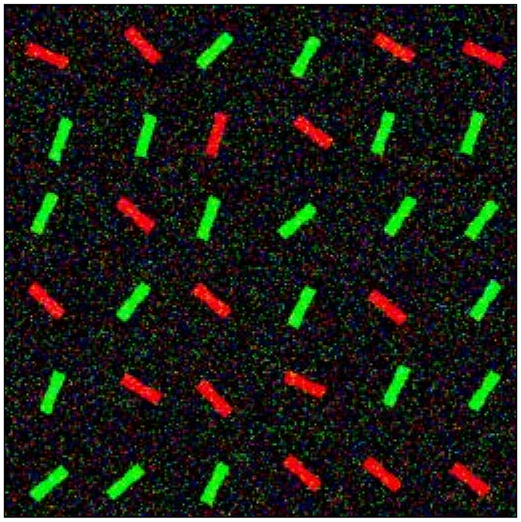
Visual saliency

As already mentioned in the introduction, visual saliency tries to describe the visual quality of certain objects or items that allows them to grab our immediate attention. Our brains constantly drive our gaze towards the *important* regions of the visual scene, as if it were to shine a flashlight on different sub-regions of the visual world, allowing us to quickly scan our surroundings for interesting objects and events while neglecting the less important parts.

It is thought that this is an evolutionary strategy to deal with the constant **information overflow** that comes with living in a visually rich environment. For example, if you take a casual walk through a jungle, you want to be able to notice the attacking tiger in the bush to your left before admiring the intricate color pattern on the butterfly's wings in front of you. As a result, the visually salient objects have the remarkable quality of *popping out* of their surroundings, much like the target bars in the following figure:



The visual quality that makes these targets pop out may not always be trivial though. If you are viewing the image on the left in color, you may immediately notice the only red bar in the image. However, if you look at the same image in grayscale, the target bar will be hard to find (it is the fourth bar from the top, fifth bar from the left). Similar to color saliency, there is a visually salient bar in the image on the right. Although the target bar is of unique color in the left image and of unique orientation in the right image, put the two characteristics together and suddenly the unique target bar does not pop out anymore:



In this preceding display, there is again one bar that is unique and different from all the other ones. However, because of the way the distracting items were designed, there is little salience to guide you towards the target bar. Instead, you find yourself scanning the image, seemingly at random, looking for something interesting. (Hint: The target is the only red and almost-vertical bar in the image, second row from the top, third column from the left.)

What does this have to do with computer vision, you ask? Quite a lot, actually. Artificial vision systems suffer from information overload much like you and me, except that they know even less about the world than we do. What if we could extract some insights from biology and use them to teach our algorithms something about the world? Imagine a dashboard camera in your car that automatically focuses on the most relevant traffic sign. Imagine a surveillance camera that is part of a wildlife observation station that will automatically detect and track the sighting of the notoriously shy platypus but will ignore everything else. How can we teach the algorithm what is important and what is not? How can we make that platypus "pop out"?

Fourier analysis

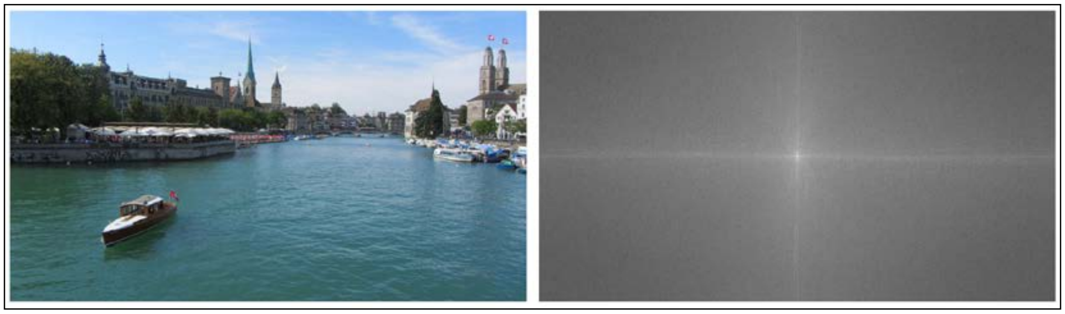
To find the visually salient sub-regions of an image, we need to look at its **frequency spectrum**. So far we have treated all our images and video frames in the **spatial domain**; that is, by analyzing the pixels or studying how the image intensity changes in different sub-regions of the image. However, the images can also be represented in the **frequency domain**; that is, by analyzing the pixel frequencies or studying how often and with what periodicity the pixels show up in the image.

An image can be transformed from the space domain into the frequency domain by applying the **Fourier transform**. In the frequency domain, we no longer think in terms of image coordinates *(x,y)*. Instead, we aim to find the spectrum of an image. Fourier's radical idea basically boils down to the following question: what if any signal or image could be transformed into a series of circular paths (also called **harmonics**)?

For example, think of a rainbow. Beautiful, isn't it? In a rainbow, white sunlight (composed of many different colors or parts of the spectrum) is spread into its spectrum. Here the color spectrum of the sunlight is exposed when the rays of light pass through raindrops (much like white light passing through a glass prism). The Fourier transform aims to do the same thing: to recover all the different parts of the spectrum that are contained in the sunlight.

A similar thing can be achieved for arbitrary images. In contrast to rainbows, where frequency corresponds to electromagnetic frequency, with images we consider spatial frequency; that is, the spatial periodicity of the pixel values. In an image of a prison cell, you can think of spatial frequency as (the inverse of) the distance between two adjacent prison bars.

The insights that can be gained from this change of perspective are very powerful. Without going into too much detail, let us just remark that a Fourier spectrum comes with both a magnitude and a phase. While the magnitude describes the amount of different frequencies in the image, the phase talks about the spatial location of these frequencies. The following image shows a natural image on the left and the corresponding Fourier magnitude spectrum (of the grayscale version) on the right:



The magnitude spectrum on the right tells us which frequency components are the most prominent (bright) in the grayscale version of the image on the left. The spectrum is adjusted so that the center of the image corresponds to zero frequency in x and y. The further you move to the border of the image, the higher the frequency gets. This particular spectrum is telling us that there are a lot of low-frequency components in the image on the left (clustered around the center of the image).

In OpenCV, this transformation can be achieved with the **Discrete Fourier Transform** (**DFT**) using the `plot_magnitude` method of the `Saliency` class. The procedure is as follows:

1. **Convert the image to grayscale if necessary**: Because the method accepts both grayscale and RGB color images, we need to make sure we operate on a single-channel image:

```
def plot_magnitude(self):
    if len(self.frame_orig.shape)>2:
        frame = cv2.cvtColor(self.frame_orig,
            cv2.COLOR_BGR2GRAY)
    else:
        frame = self.frame_orig
```

2. **Expand the image to an optimal size**: It turns out that the performance of a DFT depends on the image size. It tends to be fastest for the image sizes that are multiples of the number two. It is therefore generally a good idea to pad the image with zeros:

```
rows, cols = self.frame_orig.shape[:2]
nrows = cv2.getOptimalDFTSize(rows)
ncols = cv2.getOptimalDFTSize(cols)
frame = cv2.copyMakeBorder(frame, 0, ncols-cols, 0,
    nrows-rows, cv2.BORDER_CONSTANT, value = 0)
```

3. **Apply the DFT**: This is a single function call in NumPy. The result is a 2D matrix of complex numbers:

```
img_dft = np.fft.fft2(frame)
```

4. **Transform the real and complex values to magnitude**: A complex number has a real (Re) and a complex (imaginary - Im) part. To extract the magnitude, we take the absolute value:

```
magn = np.abs(img_dft)
```

5. **Switch to a logarithmic scale**: It turns out that the dynamic range of the Fourier coefficients is usually too large to be displayed on the screen. We have some small and some high changing values that we can't observe like this. Therefore, the high values will all turn out as white points, and the small ones as black points. To use the gray scale values for visualization, we can transform our linear scale to a logarithmic one:

```
log_magn = np.log10(magn)
```

6. **Shift quadrants**: To center the spectrum on the image. This makes it easier to visually inspect the magnitude spectrum:

```
spectrum = np.fft.fftshift(log_magn)
```

7. **Return the result for plotting**:

```
return spectrum/np.max(spectrum)*255
```

Natural scene statistics

The human brain figured out how to focus on visually salient objects a long time ago. The natural world in which we live has some statistical regularities that makes it uniquely *natural*, as opposed to a chessboard pattern or a random company logo. Probably, the most commonly known statistical regularity is the 1/f law. It states that the amplitude of the ensemble of natural images obeys a 1/f distribution, as shown in the figure later This is sometimes also referred to as **scale invariance**.

A 1D power spectrum (as a function of frequency) of a 2D image can be visualized with the `plot_power_spectrum` method of the `Saliency` class. We can use a similar recipe as for the magnitude spectrum used previously, but we will have to make sure that we correctly collapse the 2D spectrum onto a single axis.

1. Convert the image to grayscale if necessary (same as earlier):

```
def plot_power_spectrum(self):
    if len(self.frame_orig.shape)>2:
        frame = cv2.cvtColor(self.frame_orig,
            cv2.COLOR_BGR2GRAY)
    else:
        frame = self.frame_orig
```

2. Expand the image to optimal size (same as earlier):

```
rows, cols = self.frame_orig.shape[:2]
nrows = cv2.getOptimalDFTSize(rows)
ncols = cv2.getOptimalDFTSize(cols)
frame = cv2.copyMakeBorder(frame, 0, ncols-cols, 0,
    nrows-rows, cv2.BORDER_CONSTANT, value = 0)
```

3. **Apply the DFT and get the log spectrum**: Here we give the user an option (via flag `use_numpy_fft`) to use either NumPy's or OpenCV's Fourier tools:

```
if self.use_numpy_fft:
    img_dft = np.fft.fft2(frame)
    spectrum = np.log10(np.real(np.abs(img_dft))**2)
else:
    img_dft = cv2.dft(np.float32(frame),
        flags=cv2.DFT_COMPLEX_OUTPUT)
    spectrum = np.log10(img_dft[:,:,0]**2
        + img_dft[:,:,1]**2)
```

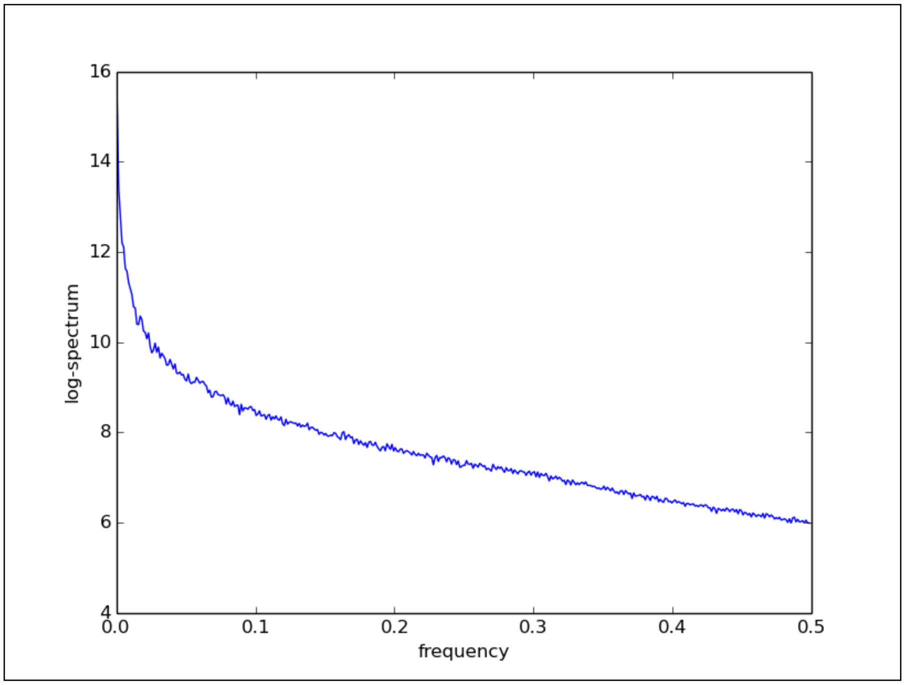
4. **Perform radial averaging**: This is the tricky part. It would be wrong to simply average the 2D spectrum in the direction of x or y. What we are interested in is a spectrum as a function of frequency, independent of the exact orientation. This is sometimes also called the **radially averaged power spectrum** (**RAPS**), and can be achieved by summing up all the frequency magnitudes, starting at the center of the image, looking into all possible (radial) directions, from some frequency `r` to `r+dr`. We use the binning function of NumPy's histogram to sum up the numbers, and accumulate them in the variable `histo`:

```
L = max(frame.shape)
freqs = np.fft.fftfreq(L)[:L/2]
dists = np.sqrt(np.fft.fftfreq(frame.shape[0])
    [:,np.newaxis]**2 + np.fft.fftfreq
        (frame.shape[1])**2)
dcount = np.histogram(dists.ravel(), bins=freqs)[0]
histo, bins = np.histogram(dists.ravel(), bins=freqs,
    weights=spectrum.ravel())
```

5. **Plot the result**: Finally, we can plot the accumulated numbers in histo, but must not forget to normalize these by the bin size (`dcount`):

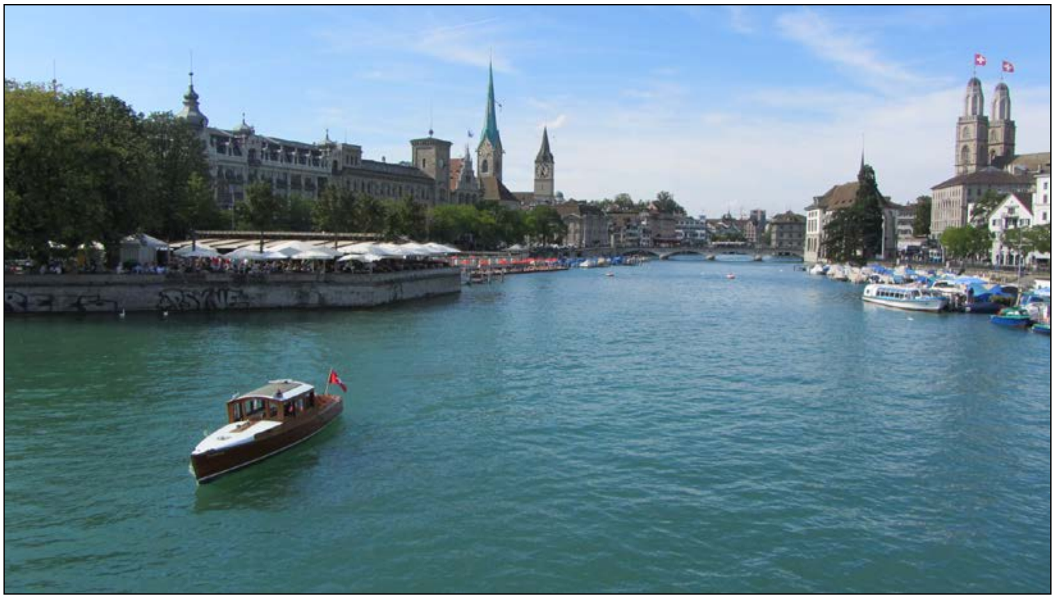
```
centers = (bins[:-1] + bins[1:]) / 2
plt.plot(centers, histo/dcount)
plt.xlabel('frequency')
plt.ylabel('log-spectrum')
plt.show()
```

The result is a function that is inversely proportional to the frequency. If you want to be absolutely certain of the 1/f property, you could take `np.log10` of all the x values and make sure the curve is decreasing roughly linearly. On a linear *x* axis and logarithmic *y* axis, the plot looks like the following:



This property is quite remarkable. It states that if we were to average all the spectra of all the images ever taken of natural scenes (neglecting all the ones taken with fancy image filters, of course), we would get a curve that would look remarkably like the one shown in the preceding image.

But going back to the image of a peaceful little boat on the Limmat river, what about single images? We have just looked at the power spectrum of this image and witnessed the 1/f property. How can we use our knowledge of natural image statistics to tell an algorithm not to stare at the tree on the left, but instead focus on the boat that is chugging in the water?



This is where we realize what saliency really means.

Generating a Saliency map with the spectral residual approach

The things that deserve our attention in an image are not the image patches that follow the 1/f law, but the patches that stick out of the smooth curves. In other words, the statistical anomalies. These anomalies are termed the **spectral residual** of an image, and correspond to the potentially *interesting* patches of an image (or proto-objects). A map that shows these statistical anomalies as bright spots is called a saliency map.

> The spectral residual approach described here is based on the original scientific publication by *Xiaodi Hou* and *Liqing Zhang* (2007). *Saliency Detection: A Spectral Residual Approach*. IEEE Transactions on Computer Vision and Pattern Recognition (CVPR), p.1-8. doi: 10.1109/CVPR.2007.383267.

In order to generate a saliency map based on the spectral residual approach, we need to process each channel of an input image separately (single channel in the case of a grayscale input image, and three separate channels in the case of an RGB input image).

The saliency map of a single channel can be generated with the private method `Saliency._get_channel_sal_magn` using the following recipe:

1. Calculate the (magnitude and phase of the) Fourier spectrum of an image, by again using either the `fft` module of NumPy or OpenCV functionality:

```
def _get_channel_sal_magn(self, channel):
    if self.use_numpy_fft:
        img_dft = np.fft.fft2(channel)
        magnitude, angle = cv2.cartToPolar
            (np.real(img_dft), np.imag(img_dft))
    else:
        img_dft = cv2.dft(np.float32(channel),
            flags=cv2.DFT_COMPLEX_OUTPUT)
        magnitude, angle = cv2.cartToPolar
            (img_dft[:, :, 0], img_dft[:, :, 1])
```

2. Calculate the log amplitude of the Fourier spectrum. We will clip the lower bound of magnitudes to 1e-9 in order to prevent a division by zero while calculating the log:

```
log_ampl = np.log10(magnitude.clip(min=1e-9))
```

3. Approximate the averaged spectrum of a typical natural image by convolving the image with a local averaging filter:

```
log_ampl_blur = cv2.blur(log_amlp, (3, 3))
```

4. Calculate the spectral residual. The spectral residual primarily contains the nontrivial (or unexpected) parts of a scene.

```
magn = np.exp(log_amlp - log_ampl_blur)
```

5. Calculate the saliency map by using the inverse Fourier transform, again either via the `fft` module in NumPy or with OpenCV:

```
    if self.use_numpy_fft:
        real_part, imag_part = cv2.polarToCart(residual,
            angle)
        img_combined = np.fft.ifft2
            (real_part + 1j*imag_part)
        magnitude, _ = cv2.cartToPolar
            ( np.real(img_combined), np.imag(img_combined))
    else:
        img_dft[:, :, 0], img_dft[:, :, 1] =
            cv2.polarToCart(
                residual, angle)
        img_combined = cv2.idft(img_dft)
        magnitude, _ = cv2.cartToPolar
            (img_combined[:, :, 0], img_combined[:, :, 1])
    return magnitude
```

The resulting single-channel saliency map (`magnitude`) is then returned to `Saliency.get_saliency_map`, where the procedure is repeated for all channels of the input image. If the input image is grayscale, we are pretty much done:

```
def get_saliency_map(self):
    if self.need_saliency_map:
        # haven't calculated saliency map for this frame yet
        num_channels = 1
        if len(self.frame_orig.shape)==2:
            # single channel
            sal = self._get_channel_sal_magn(self.frame_small)
```

However, if the input image has multiple channels, as is the case for an RGB color image, we need to consider each channel separately:

```
        else:
            # consider each channel independently
            sal = np.zeros_like
                (self.frame_small).astype(np.float32)
            for c in xrange(self.frame_small.shape[2]):
                sal[:, :, c] = self._get_channel_sal_magn
                    (self.frame_small[:, :, c])
```

The overall salience of a multi-channel image is then determined by the average over all channels:

```
sal = np.mean(sal, 2)
```

Finally, we need to apply some post-processing, such as an optional blurring stage to make the result appear smoother:

```
        if self.gauss_kernel is not None:
            sal = cv2.GaussianBlur(sal, self.gauss_kernel,
                sigmaX=8, sigmaY=0)
```

Also, we need to square the values in `sal` in order to highlight the regions of high salience, as outlined by the authors of the original paper. In order to display the image, we scale it back up to its original resolution and normalize the values, so that the largest value is one:

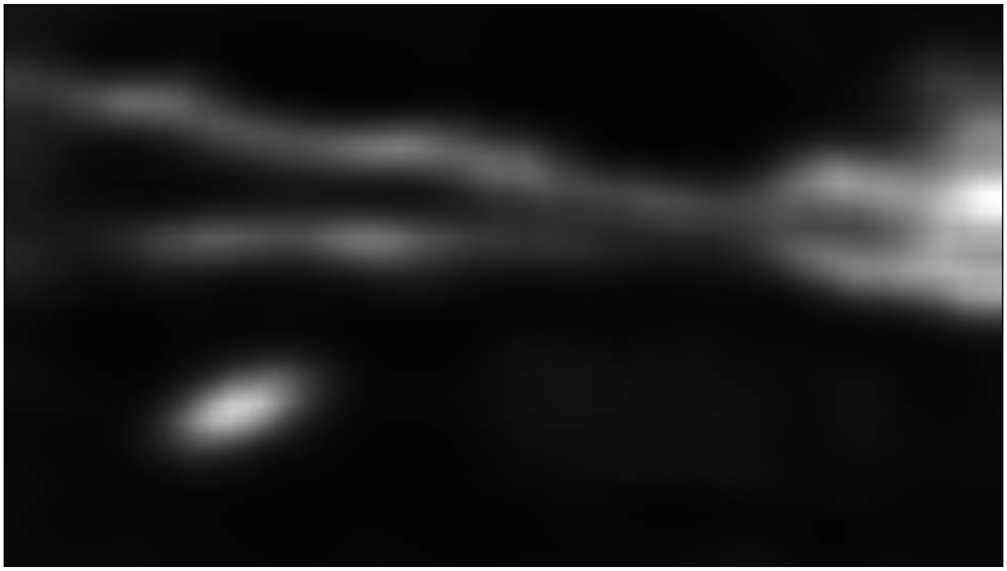
```
            sal = sal**2
            sal = np.float32(sal)/np.max(sal)
            sal = cv2.resize(sal, self.frame_orig.shape[1::-1])
```

In order to avoid having to redo all these intense calculations, we store a local copy of the saliency map for further reference and make sure to lower the flag:

```
            self.saliency_map = sal
            self.need_saliency_map = False

        return self.saliency_map
```

Then, when the user makes subsequent calls to class methods that rely on the calculation of the saliency map under the hood, we can simply refer to the local copy instead of having to do the calculations all over again.

The resulting saliency map then looks like the following image:



Now we can clearly spot the boat in the water (lower-left corner), which appears as one of the most salient sub-regions of the image. There are other salient regions, too, such as the Grossmünster on the right (have you guessed the city yet?).

> By the way, the reason these two areas are the most salient ones in the image seems to be clear and undisputable evidence that the algorithm is aware of the ridiculous number of church towers in the city center of Zurich, effectively prohibiting any chance of them being labeled as "salient".

Detecting proto-objects in a scene

In a sense, the saliency map is already an explicit representation of proto-objects, as it contains only the *interesting* parts of an image. So now that we have done all the hard work, all that is left to do in order to obtain a proto-object map is to threshold the saliency map.

The only open parameter to consider here is the threshold. Setting the threshold too low will result in labeling a lot of regions as proto-objects, including some that might not contain anything of interest (false alarm). On the other hand, setting the threshold too high will ignore most of the salient regions in the image and might leave us with no proto-objects at all. The authors of the original spectral residual paper chose to label only those regions of the image as proto-objects whose saliency was larger than three-times the mean saliency of the image. We give the user the choice to either implement this threshold, or to go with the Otsu threshold by setting the input flag `use_otsu` to `true`:

```
def get_proto_objects_map(self, use_otsu=True):
```

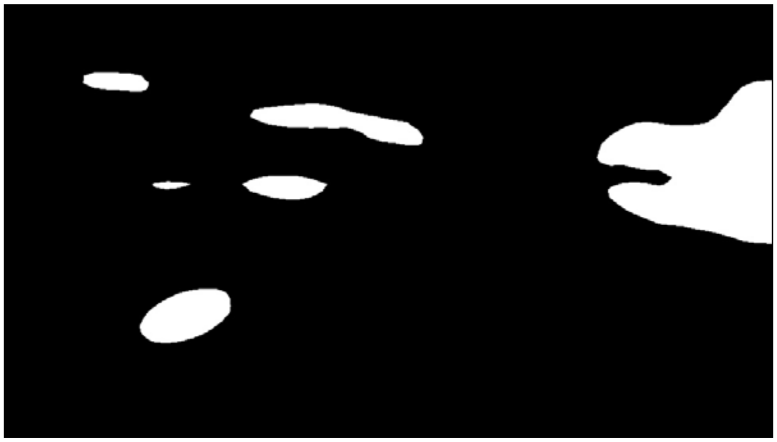
We then retrieve the saliency map of the current frame and make sure to convert it to uint8 precision, so that it can be passed to `cv2.threshold`:

```
        saliency = self.get_saliency_map()
        if use_otsu:
            _, img_objects = cv2.threshold(np.uint8(saliency*255),
                0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Otherwise, we will use the threshold `thresh`:

```
        else:
            thresh = np.mean(saliency)*255
            _, img_objects = cv2.threshold(np.uint8(saliency*255),
                thresh, 255, cv2.THRESH_BINARY)
        return img_objects
```

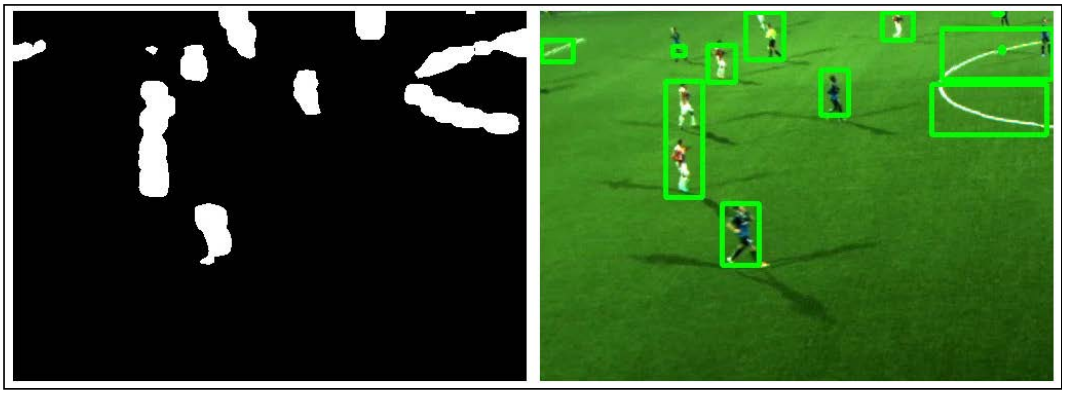
The resulting proto-objects mask looks like the following image:



The proto-objects mask then serves as an input to the tracking algorithm.

Mean-shift tracking

It turns out that the salience detector discussed previously is already a great tracker of proto-objects by itself. One could simply apply the algorithm to every frame of a video sequence and get a good idea of the location of the objects. However, what is getting lost is correspondence information. Imagine a video sequence of a busy scene, such as from a city center or a sports stadium. Although a saliency map could highlight all the proto-objects in every frame of a recorded video, the algorithm would have no way to know which proto-objects from the previous frame are still visible in the current frame. Also, the proto-objects map might contain some false-positives, such as in the following example:

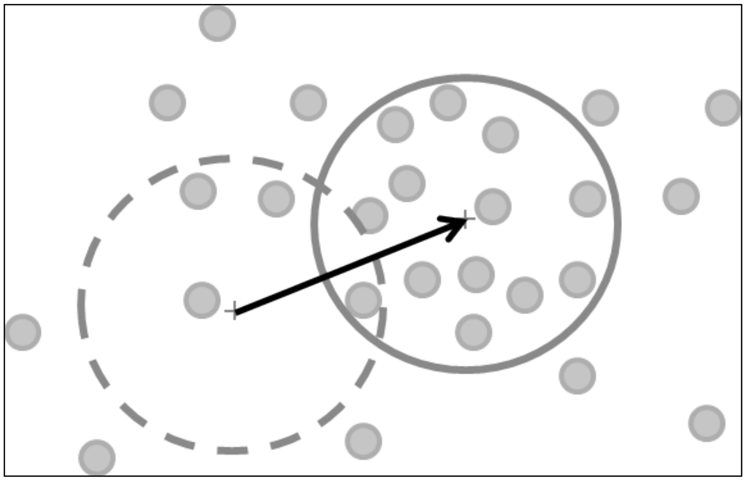


Note that the bounding boxes extracted from the proto-objects map made (at least) three mistakes in the preceding example: it missed highlighting a player (upper-left), merged two players into the same bounding box, and highlighted some additional arguably non-interesting (although visually salient) objects. In order to improve these results, we want to make use of a tracking algorithm.

To solve the correspondence problem, we could use the methods we have learned about previously, such as feature matching and optic flow. Or, we could use a different technique called mean-shift tracking.

Mean-shift is a simple yet very effective technique for tracking arbitrary objects. The intuition behind mean-shift is to consider the pixels in a small region of interest (say, a bounding box of an object we want to track) as sampled from an underlying probability density function that best describes a target.

Consider, for example, the following image:



Here, the small gray dots represent samples from a probability distribution. Assume that the closer the dots, the more similar they are to each other. Intuitively speaking, what mean-shift is trying to do is to find the densest region in this landscape and draw a circle around it. The algorithm might start out centering a circle over a region of the landscape that is not dense at all (dashed circle). Over time, it will slowly move towards the densest region (solid circle) and anchor on it. If we design the landscape to be more meaningful than dots (for example, by making the dots correspond to color histograms in the small neighborhoods of an image), we can use mean-shift tracking to find the objects of interest in the scene by finding the histogram that most closely matches the histogram of a target object.

Mean-shift has many applications (such as clustering, or finding the mode of probability density functions), but it is also particularly well-suited to target tracking. In OpenCV, the algorithm is implemented in `cv2.meanShift`, but it requires some pre-processing to function correctly. We can outline the procedure as follows:

1. **Fix a window around each data point**: For example, a bounding box around an object or region of interest.
2. **Compute the mean of data within the window**: In the context of tracking, this is usually implemented as a histogram of the pixel values in the region of interest. For best performance on color images, we will convert to HSV color space.
3. **Shift the window to the mean and repeat until convergence**: This is handled transparently by `cv2.meanShift`. We can control the length and accuracy of the iterative method by specifying termination criteria.

Automatically tracking all players on a soccer field

For the remainder of this chapter, our goal is to combine the saliency detector with mean-shift tracking to automatically track all the players on a soccer field. The proto-objects identified by the salience detector will serve as input to the mean-shift tracker. Specifically, we will focus on a video sequence from the Alfheim dataset, which can be freely obtained from `http://home.ifi.uio.no/paalh/dataset/alfheim/`.

The reason for combining the two algorithms (saliency map and mean-shift tracking), is to remove false positives and improve the accuracy of the overall tracking. This will be achieved in a two-step procedure:

1. Have both the saliency detector and mean-shift tracking assemble a list of bounding boxes for all the proto-objects in a frame. The saliency detector will operate on the current frame, whereas the mean-shift tracker will try to find the proto-objects from the previous frame in the current frame.
2. Keep only those bounding boxes for which both algorithms agree on the location and size. This will get rid of outliers that have been mislabeled as proto-objects by one of the two algorithms.

The hard work is done by the previously introduced `MultiObjectTracker` class and its `advance_frame` method. This method relies on a few private worker methods, which will be explained in detail next. The `advance_frame` method is called whenever a new frame arrives, and accepts a proto-objects map as input:

```
def advance_frame(self, frame, proto_objects_map):
    self.tracker = copy.deepcopy(frame)
```

The method then builds a list of all the candidate bounding boxes, combining the bounding boxes both from the saliency map of the current frame as well as the mean-shift tracking results from the previous to the current frame:

```
# build a list of all bounding boxes
box_all = []

# append to the list all bounding boxes found from the
# current proto-objects map
box_all = self._append_boxes_from_saliency(proto_objects_map,
    box_all)

    # find all bounding boxes extrapolated from last frame
    # via mean-shift tracking
    box_all = self._append_boxes_from_meanshift(frame, box_all)
```

The method then attempts to merge the candidate bounding boxes in order to remove the duplicates. This can be achieved with `cv2.groupRectangles`, which will return a single bounding box if `group_thresh+1` or more bounding boxes overlap in an image:

```
# only keep those that are both salient and in mean shift
if len(self.object_roi)==0:
    group_thresh = 0    # no previous frame: keep all form
    # saliency
else:
    group_thresh = 1 # previous frame + saliency
box_grouped,_ = cv2.groupRectangles(box_all, group_thresh,
    0.1)
```

In order to make mean-shift work, we will have to do some bookkeeping, which will be explained in detail in the following subsections:

```
# update mean-shift bookkeeping for remaining boxes
self._update_mean_shift_bookkeeping(frame, box_grouped)
```

Then we can draw the list of unique bounding boxes on the input image and return the image for plotting:

```
for (x, y, w, h) in box_grouped:
    cv2.rectangle(self.tracker, (x, y), (x + w, y + h),
        (0, 255, 0), 2)

return self.tracker
```

Extracting bounding boxes for proto-objects

The first private worker method is relatively straightforward. It takes a proto-objects map as input as well as a (previously aggregated) list of bounding boxes. To this list, it adds all the bounding boxes found from the contours of the proto-objects:

```
def _append_boxes_from_saliency(self, proto_objects_map, box_all):
    box_sal = []
    cnt_sal, _ = cv2.findContours(proto_objects_map, 1, 2)
```

However, it discards the bounding boxes that are smaller than some threshold, `self.min_cnt_area`, which is set in the constructor:

```
for cnt in cnt_sal:
    # discard small contours
    if cv2.contourArea(cnt) < self.min_cnt_area:
        continue
```

The result is appended to the `box_all` list and passed up for further processing:

```
    # otherwise add to list of boxes found from saliency map
    box = cv2.boundingRect(cnt)
    box_all.append(box)

return box_all
```

Setting up the necessary bookkeeping for mean-shift tracking

The second private worker method is concerned with setting up all the bookkeeping that is necessary to perform mean-shift tracking. The method accepts an input image and a list of bounding boxes for which to generate the bookkeeping information:

```
def _update_mean_shift_bookkeeping(self, frame, box_grouped):
```

Bookkeeping mainly consists of calculating a histogram of the HSV color values of each proto-object's bounding box. Thus the input RGB image is converted to HSV right away:

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Then, every bounding box in `box_grouped` is parsed. We need to store both the location and size of the bounding box (`self.object_box`), as well as a histogram of the HSV color values (`self.object_roi`):

```
self.object_roi = []
self.object_box = []
```

The location and size of the bounding box is extracted from the list, and the region of interest is cut out of the HSV image:

```
for box in box_grouped:
    (x, y, w, h) = box
    hsv_roi = hsv[y:y + h, x:x + w]
```

We then calculate a histogram of all the hue (H) values in the region of interest. We also ignore the dim or the weakly pronounced areas of the bounding box by using a mask, and normalize the histogram in the end:

```
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)),
        np.array((180., 255., 255.)))
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
```

We then store this information in the corresponding private member variables, so that it will be available in the very next frame of the process loop, where we will aim to locate the region of interest using the mean-shift algorithm:

```
self.object_roi.append(roi_hist)
self.object_box.append(box)
```

Tracking objects with the mean-shift algorithm

Finally, the third private worker method tracks the proto-objects by using the bookkeeping information stored earlier from the previous frame. Similar to `_append_boxes_from_meanshift`, we build a list of all the bounding boxes aggregated from mean-shift and pass it up for further processing. The method accepts an input image and a previously aggregated list of bounding boxes:

```
def _append_boxes_from_meanshift(self, frame, box_all):
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

The method then parses all the previously stored proto-objects (from `self.object_roi` and `self.object_box`):

```
for i in xrange(len(self.object_roi)):
    roi_hist = copy.deepcopy(self.object_roi[i])
    box_old = copy.deepcopy(self.object_box[i])
```

In order to find the new, shifted location of a region of interest recorded in the previous image frame, we feed the back-projected region of interest to the mean-shift algorithm. Termination criteria (`self.term_crit`) will make sure to try a sufficient number of iterations (`100`) and look for mean-shifts of at least some number of pixels (`1`):

```
dst = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)
ret, box_new = cv2.meanShift(dst, tuple(box_old),
    self.term_crit)
```

But, before we append the newly detected, shifted bounding box to the list, we want to make sure that we are actually tracking the objects that move. The objects that do not move are most likely false-positives, such as line markings or other visually salient patches that are irrelevant to the task at hand.

In order to discard the irrelevant tracking results, we compare the location of the bounding box from the previous frame (`box_old`) and the corresponding bounding box from the current frame (`box_new`):

```
(xo, yo, wo, ho) = box_old
(xn, yn, wn, hn) = box_new
```

If their centers of mass did not shift at least `sqrt(self.min_shift2)` pixels, we do not include the bounding box in the list:

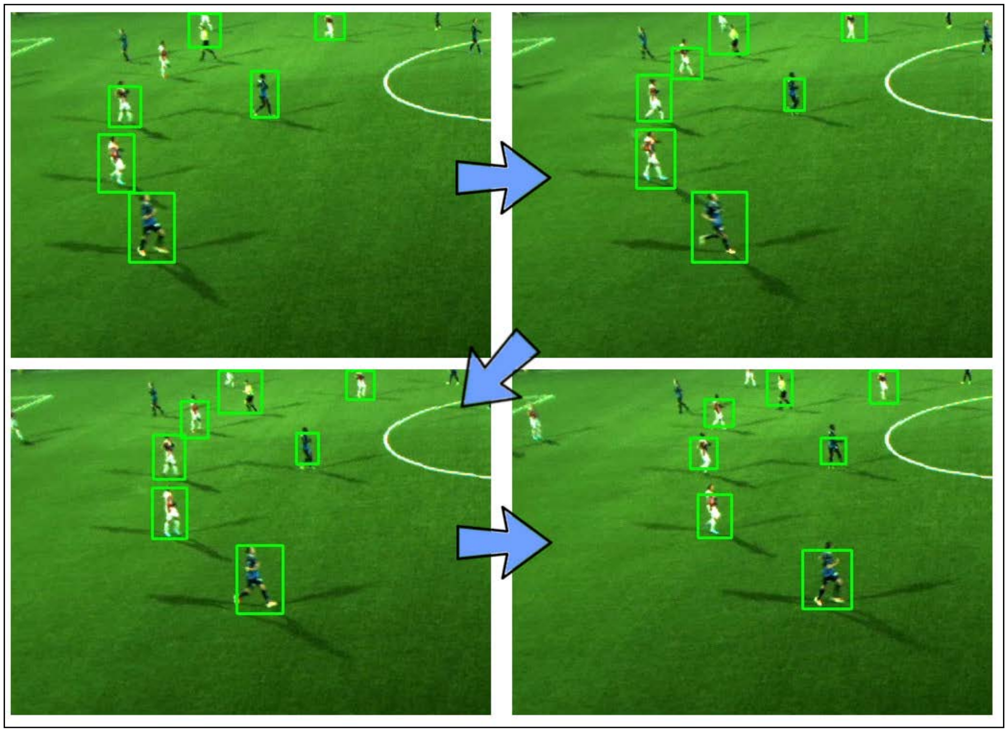
```
co = [xo + wo/2, yo + ho/2]
cn = [xn + wn/2, yn + hn/2]
if (co[0] - cn[0])**2 + (co[1] - cn[1])**2 >= self.min_shift2:
    box_all.append(box_new)
```

The resulting list of bounding boxes is again passed up for further processing:

```
        return box_all
```

Putting it all together

The result of our app can be seen in the following image:



Throughout the video sequence, the algorithm is able to pick up the location of the players, successfully tracking them frame-by-frame by using mean-shift tracking, and combining the resulting bounding boxes with the bounding boxes returned by the salience detector.

It is only through the clever combination of the saliency map and tracking that we can exclude false-positives such as line markings and artifacts of the saliency map. The magic happens in `cv2.groupRectangles`, which requires a similar bounding box to appear at least twice in the `box_all` list, otherwise it is discarded. This means that a bounding box is only then kept in the list if both mean-shift tracking and the saliency map (roughly) agree on the location and size of the bounding box.

Summary

In this chapter, we explored a way to label the potentially *interesting* objects in a visual scene, even if their shape and number is unknown. We explored natural image statistics using Fourier analysis, and implemented a state-of-the-art method for extracting the visually salient regions in the natural scenes. Furthermore, we combined the output of the salience detector with a tracking algorithm to track multiple objects of unknown shape and number in a video sequence of a soccer game.

It would now be possible to extend our algorithm to feature more complicated feature descriptions of proto-objects. In fact, mean-shift tracking might fail when the objects rapidly change size, as would be the case if an object of interest were to come straight at the camera. A more powerful tracker, which comes for free in OpenCV, is `cv2.CamShift`. **CAMShift** stands for Continuously Adaptive Mean-Shift, and bestows upon mean-shift the power to adaptively change the window size. Of course, it would also be possible to simply replace the mean-shift tracker with a previously studied technique such as feature matching or optic flow.

In the next chapter, we will move to the fascinating field of machine learning, which will allow us to build more powerful descriptors of objects. Specifically, we will focus on both detecting (where?) and identifying (what?) the street signs in images. This will allow us to train a classifier that could be used in a dashboard camera in your car, and will familiarize us with the important concepts of machine learning and object recognition.

6
Learning to Recognize Traffic Signs

The goal of this chapter is to train a multiclass classifier to recognize traffic signs. In this chapter, we will cover the following topics:

- Supervised learning concepts
- The **German Traffic Sign Recognition Benchmark** (**GTSRB**) dataset feature extraction
- **Support vector machines** (**SVMs**)

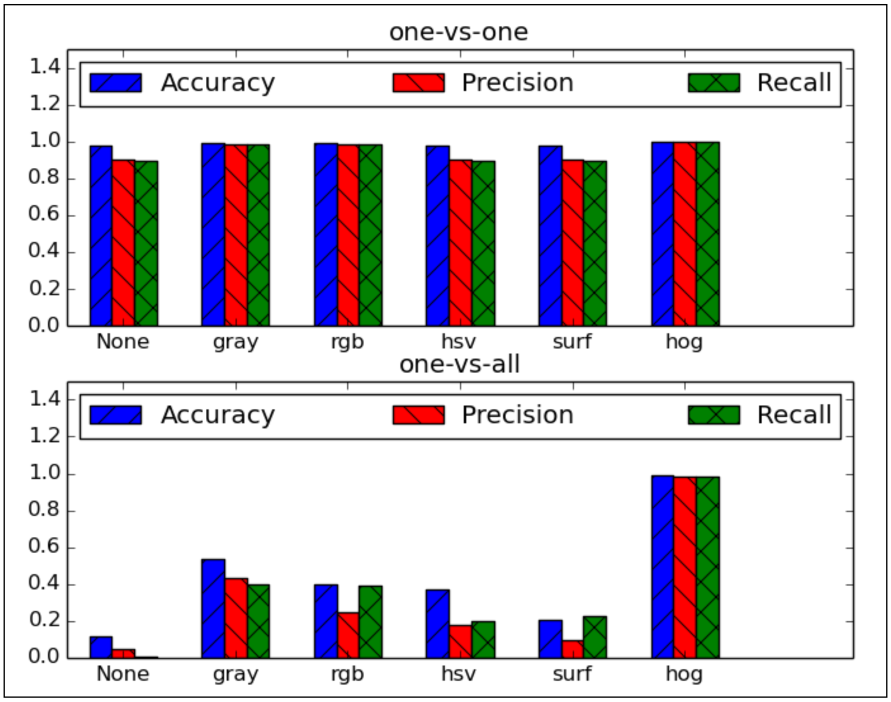
We have previously studied how to describe objects by means of keypoints and features, and how to find the correspondence points in two different images of the same physical object. However, our previous approaches were rather limited when it comes to recognizing objects in real-world settings and assigning them to conceptual categories. For example, in *Chapter 2*, *Hand Gesture Recognition Using a Kinect Depth Sensor*, the required object in the image was a hand, and it had to be nicely placed in the center of the screen. Wouldn't it be nice if we could remove these restrictions?

In this chapter, we will instead train a **Support Vector Machine** (**SVM**) to recognize all sorts of traffic signs. Although SVMs are binary classifiers (that is, they can be used to learn, at most, two categories: positives and negatives, animals and non-animals, and so on), they can be extended to be used in **multiclass** classification. In order to achieve good classification performance, we will explore a number of color spaces as well as the **Histogram of Oriented Gradients** (**HOG**) feature. Then, classification performance will be judged based on **accuracy**, **precision**, and **recall**. The following sections will explain all of these terms in detail.

To arrive at such a multiclass classifier, we need to perform the following steps:

1. **Preprocess the dataset**: We need a way to load our dataset, extract the regions of interest, and split the data into appropriate training and test sets.
2. **Extract features**: Chances are that raw pixel values are not the most informative representation of the data. We need a way to extract meaningful features from the data, such as features based on different color spaces and HOG.
3. **Train the classifier**: We will train the multiclass classifier on the training data in two different ways: the **one-vs-all** strategy (where we train a single SVM per class, with the samples of that class as positive samples and all other samples as negatives), and the **one-vs-one** strategy (where we train a single SVM for every pair of classes, with the samples of the first class as positive samples and the samples of the second class as negative samples).
4. **Score the classifier**: We will evaluate the quality of the trained ensemble classifier by calculating different performance metrics, such as accuracy, precision, and recall.

The end result will be an ensemble classifier that achieves a nearly perfect score in classifying 10 different street sign categories:



Planning the app

The final app will parse a dataset, train the ensemble classifier, assess its classification performance, and visualize the result. This will require the following components:

- `main`: The main function routine (in `chapter6.py`) for starting the application.
- `datasets.gtsrb`: A script for parsing the German Traffic Sign Recognition Benchmark (GTSRB) dataset. This script contains the following functions:
    - `load_data`: A function used to load the GTSRB dataset, extract a feature of choice, and split the data into training and test sets.
    - `_extract_features`: A function that is called by `load_data` to extract a feature of choice from the dataset.
- `classifiers.Classifier`: An abstract base class that defines the common interface for all classifiers.
- `classifiers.MultiClassSVM`: A class that implements an ensemble of SVMs for multiclass classification using the following public methods:
    - `MultiClassSVM.fit`: A method used to fit the ensemble of SVMs to training data. It takes a matrix of training data as input, where each row is a training sample and the columns contain feature values, and a vector of labels.
    - `MultiClassSVM.evaluate`: A method used to evaluate the ensemble of SVMs by applying it to some test data after training. It takes a matrix of test data as input, where each row is a test sample and the columns contain feature values, and a vector of labels. The function returns three different performance metrics: accuracy, precision, and recall.

In the following sections, we will discuss these steps in detail.

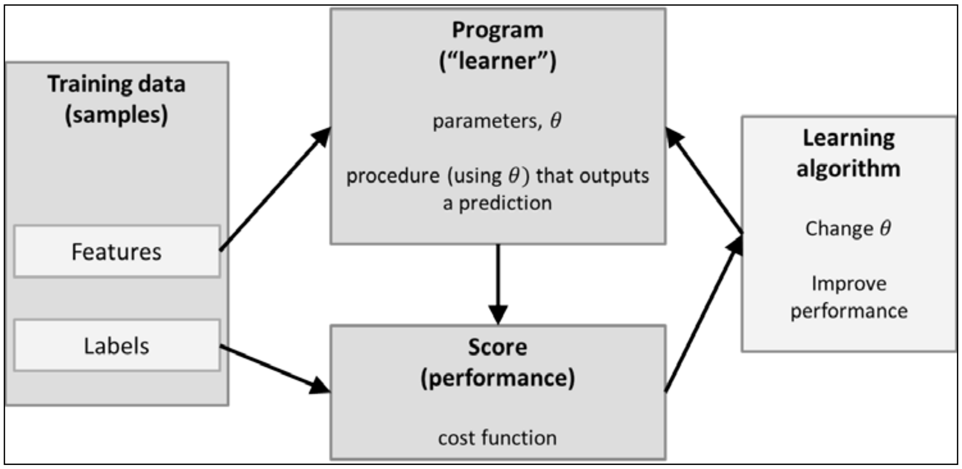
Supervised learning

An important subfield of machine learning is **supervised learning**. In supervised learning, we try to learn from a set of labeled training data; that is, every data sample has a desired target value or true output value. These target values could correspond to the continuous output of a function (such as `y` in `y = sin(x)`), or to more abstract and discrete categories (such as *cat* or *dog*). If we are dealing with continuous output, the process is called **regression**, and if we are dealing with discrete output, the process is called **classification**. Predicting housing prices from sizes of houses is an example of regression. Predicting the species from the color of a fish would be classification. In this chapter, we will focus on classification using SVMs.

The training procedure

As an example, we may want to learn what cats and dogs look like. To make this a supervised learning task, we will have to create a database of pictures of both cats and dogs (also called a **training set**), and annotate each picture in the database with its corresponding label: *cat* or *dog*. The task of the program (in literature, it is often referred to as the **learner**) is then to infer the correct label for each of these pictures (that is, for each picture, predict whether it is a picture of a cat or a dog). Based on these predictions, we derive a **score** of how well the learner performed. The score is then used to change the parameters of the learner in order to improve the score over time.

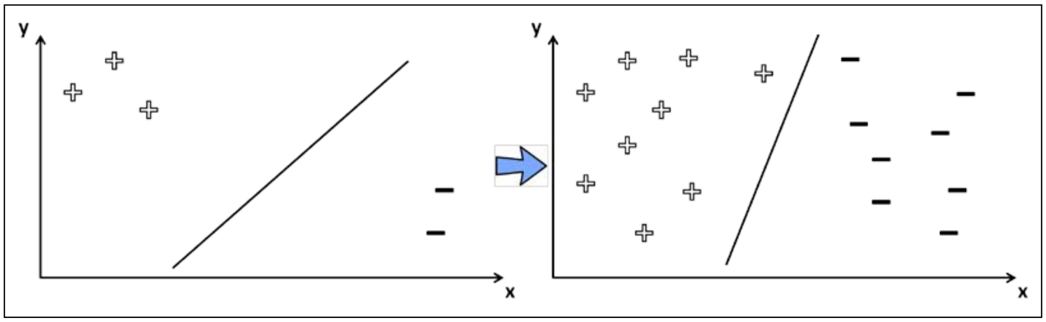
This procedure is outlined in the following figure:



Training data is represented by a set of features. For real-life classification tasks, these features are rarely the raw pixel values of an image, since these tend not to represent the data well. Often, the process of finding the features that best describe the data is an essential part of the entire learning task (also referred to as **feature selection** or **feature engineering**). That is why it is always a good idea to deeply study the statistics and appearances of the training set that you are working with before even thinking about setting up a classifier.

As you are probably aware, there is an entire zoo of learners, cost functions, and learning algorithms out there. These make up the core of the learning procedure. The learner (for example, a linear classifier, support vector machine, or decision tree) defines how input features are converted into a score or cost function (for example, mean-squared error, hinge loss, or entropy), whereas the learning algorithm (for example, gradient descent and backpropagation for neural networks) defines how the parameters of the learner are changed over time.

The training procedure in a classification task can also be thought of as finding an appropriate **decision boundary**, which is a line that best partitions the training set into two subsets, one for each class. For example, consider training samples with only two features (**x** and **y** values) and a corresponding class label (positive, **+**, or negative, **–**). At the beginning of the training procedure, the classifier tries to draw a line to separate all positives from all negatives. As the training progresses, the classifier sees more and more data samples. These are used to update the decision boundary, as illustrated in the following figure:

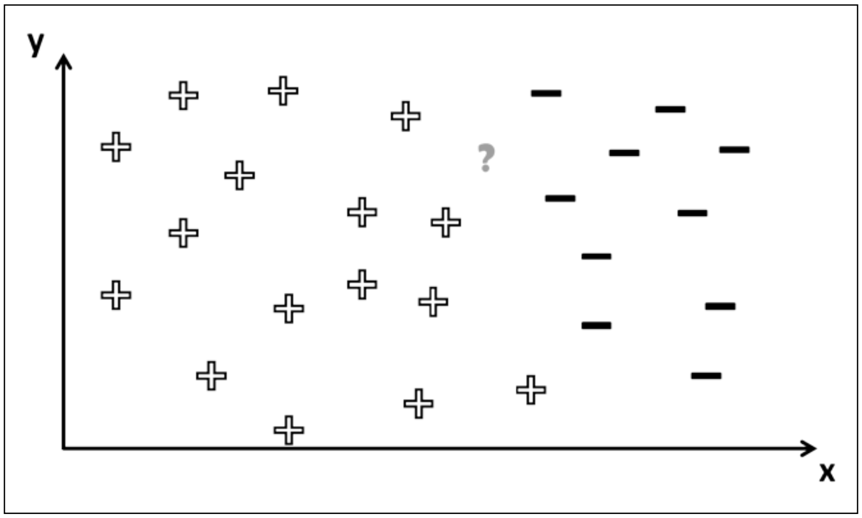


Compared to this simple illustration, an SVM tries to find the optimal decision boundary in a high-dimensional space, so the decision boundary can be more complex than a straight line.

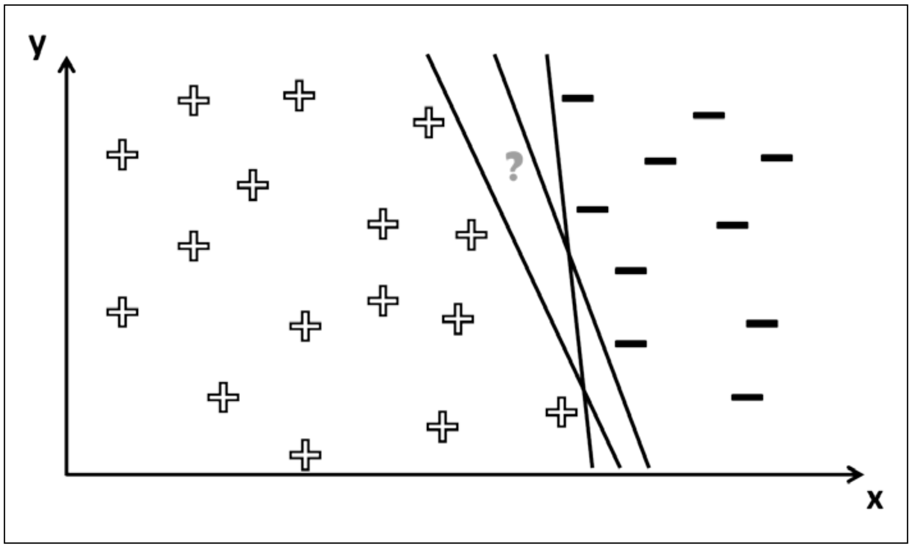
The testing procedure

In order for a trained classifier to be of any practical value, we need to know how it performs when applied to a never-seen-before data sample (also called **generalization**). To stick to our example shown earlier, we want to know which class the classifier predicts when we present it with a previously unseen picture of a cat or a dog.

More generally speaking, we want to know which class the **?** sign in the following figure corresponds to, based on the decision boundary we learned during the training phase:



You can see why this is a tricky problem. If the location of the question mark were more to the left, we would be certain that the corresponding class label is **+**. However, in this case, there are several ways to draw the decision boundary such that all the **+** signs are to the left of it and all the **–** signs are to the right of it, as illustrated in this figure:



The label of **?** thus depends on the exact decision boundary that was derived during training. If the **?** sign in the preceding figure is actually a **–**, then only one decision boundary (the leftmost) would get the correct answer. A common problem is that training results in a decision boundary that works "too well" on the training set (also known as **overfitting**), but makes a lot of mistakes when applied to unseen data. In that case, it is likely that the learner imprinted details that are specific to the training set on the decision boundary, instead of revealing general properties about the data that might also be true for unseen data.

A common technique for reducing the effect of overfitting is called **regularization**.

Long story short, the problem always comes back to finding the boundary that best splits, not only the training, but also the test set. That is why the most important metric for a classifier is its generalization performance (that is, how well it classifies data not seen in the training phase).

A classifier base class

From the insights gained in the preceding content, you are now able to write a simple **base class** suitable for all possible classifiers. You can think of this class as a blueprint or recipe that will apply to all classifiers that we are yet to design (we did this with the `BaseLayout` class in *Chapter 1*, *Fun with Filters*). In order to create an **abstract base class (ABC)** in Python, we need to include the `ABCMeta` module:

```
from abc import ABCMeta
```

This allows us to register the class as a `metaclass`:

```
class Classifier:
    """Abstract base class for all classifiers"""
    __metaclass__ = ABCMeta
```

Recall that an abstract class has at least one abstract method. An abstract method is akin to specifying that a certain method must exist, but we are not yet sure what it should look like. We now know that a classifier in its most generic form should contain a method for training, wherein a model is fitted to the training data, and for testing, wherein the trained model is evaluated by applying it to the test data:

```
    @abstractmethod
    def fit(self, X_train, y_train):
        pass
    @abstractmethod
    def evaluate(self, X_test, y_test, visualize=False):
        pass
```

Here, `X_train` and `X_test` correspond to the training and test data, respectively, where each row represents a sample, and each column is a feature value of that sample. The training and test labels are passed as `y_train` and `y_test` vectors, respectively.

The GTSRB dataset

In order to apply our classifier to traffic sign recognition, we need a suitable dataset. A good choice might be the German Traffic Sign Recognition Benchmark (GTSRB), which contains more than 50,000 images of traffic signs belonging to more than 40 classes. This is a challenging dataset that was used by professionals in a classification challenge during the **International Joint Conference on Neural Networks** (**IJCNN**) 2011. The dataset can be freely obtained from `http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset`.

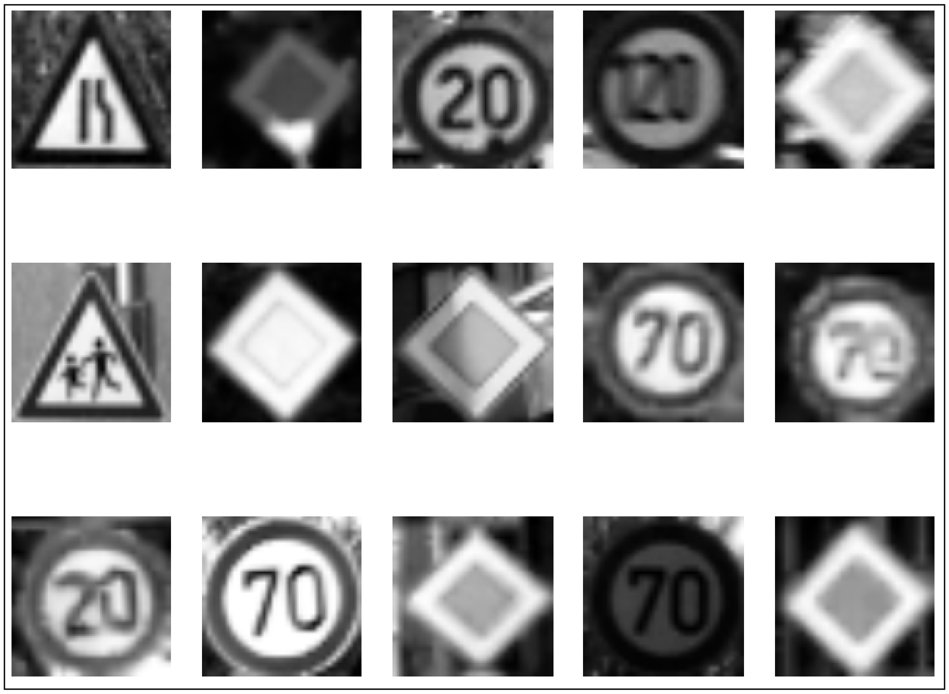
The GTSRB dataset is perfect for our purposes because it is large, organized, open source, and annotated. However, for the purpose of this book, we will limit the classification to data samples from a total of 10 classes.

Although the actual traffic sign is not necessarily a square, or centered, in each image, the dataset comes with an annotation file that specifies the bounding boxes for each sign.

A good idea before doing any sort of machine learning is usually to get a feeling of the dataset, its qualities, and its challenges. If all the exemplars of the dataset are stored in a list, `X`, then we can plot a few exemplars with the following script, where we pick a fixed number (`sample_size`) of random indices (`sample_idx`) and display each exemplar (`X[sample_idx[sp-1]]`) in a separate subplot:

```
sample_size = 15
sample_idx = np.random.randint(len(X), size=sample_size)
sp = 1
for r in xrange(3):
    for c in xrange(5):
        ax = plt.subplot(3,5,sp)
        sample = X[sample_idx[sp-1]]
        ax.imshow(sample.reshape((32,32)), cmap=cm.Greys_r)
            ax.axis('off')
        sp += 1
plt.show()
```

The following screenshot shows some examples of this dataset:



Even from this small data sample, it is immediately clear that this is a challenging dataset for any sort of classifier. The appearances of the signs change drastically based on viewing angle (orientation), viewing distance (blurriness), and lighting conditions (shadows and bright spots). For some of these signs, such as the rightmost sign in the second row, it is difficult even for humans (at least for me) to tell the correct class label right away. Good thing we are aspiring experts of machine learning!

Parsing the dataset

Luckily, the chosen dataset comes with a script for parsing the files (more information can be found at `http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset#Codesnippets`).

We spruce it up a bit and adjust it for our purposes. In particular, we want a function that not only loads the dataset, but also extracts a certain feature of interest (via the `feature` input argument), crops the sample to the hand-labeled **Region of Interest** (**ROI**) containing solely the sample (`cut_roi`), and automatically splits the data into a training and a test set (`test_split`). We also allow the specification of a random seed number (`seed`), and plot some samples for visual inspection (`plot_samples`):

```
import cv2
import numpy as np
import csv

from matplotlib import cm
from matplotlib import pyplot as plt

def load_data(rootpath="datasets", feature="hog", cut_roi=True,
    test_split=0.2, plot_samples=False, seed=113):
```

Although the full dataset contains more than 50,000 examples belonging to 43 classes, for the purpose of this chapter, we will limit ourselves to 10 classes. For easy access, we will hardcode the class labels to use here, but it is straightforward to include more classes (note that you will have to download the entire dataset for this):

```
classes = np.array([0, 4, 8, 12, 16, 20, 24, 28, 32, 36])
```

We then need to iterate over all the classes so as to read all the training samples (to be stored in `X`) and their corresponding class labels (to be stored in `labels`). Every class has a CSV file containing all of the annotation information for every sample in the class, which we will parse with `csv.reader`:

```
X = [] # images
labels =  []  # corresponding labels

# subdirectory for class
for c in xrange(len(classes)):
    prefix = rootpath + '/' + format(classes[c], '05d') + '/'

    # annotations file
    gt_file = open(prefix + 'GT-'+ format(classes[c], '05d')
        + '.csv')
    gt_reader = csv.reader(gt_file, delimiter=';')
```

Every line of the file contains the annotation for one data sample. We skip the first line (the header) and extract the sample's filename (`row[0]`) so that we can read in the image:

```
        gt_reader.next() # skip header
        # loop over all images in current annotations file
        for row in gt_reader:
            # first column is filename
            im = cv2.imread(prefix + row[0])
```

Occasionally, the object in these samples is not perfectly cut out but is embedded in its surroundings. If the `cut_roi` input argument is set, we will ignore the background and cut out the object using the bounding boxes specified in the annotation file:

```
        if cut_roi:
            im = im[np.int(row[4]):np.int(row[6]),
                np.int(row[3]):np.int(row[5]), :]
```

Then, we are ready to append the image (`im`) and its class label (`c`) to our list of samples (`X`) and class labels (`labels`):

```
        X.append(im)
        labels.append(c)
        gt_file.close()
```

Often, it is desirable to perform some form of feature extraction, because raw image data is rarely the best description of the data. We will defer this job to another function, which we will discuss in detail in the next section:

```
    if feature is not None:
        X = _extract_feature(X, feature)
```

As pointed out in the previous subsection, it is imperative to keep the samples that we use to train our classifier, separate from the samples that we use to test it. For this, we shuffle the data and split it into two separate sets such that the training set contains a fraction (`1-test_split`) of all samples and the rest of the samples belong to the test set:

```
    np.random.seed(seed)
    np.random.shuffle(X)
    np.random.seed(seed)
    np.random.shuffle(labels)

    X_train = X[:int(len(X)*(1-test_split))]
    y_train = labels[:int(len(X)*(1-test_split))]
    X_test = X[int(len(X)*(1-test_split)):]
    y_test = labels[int(len(X)*(1-test_split)):]
```

Finally, we can return the extracted data:

```
    return (X_train, y_train), (X_test, y_test)
```

Feature extraction

Chances are, that raw pixel values are not the most informative way to represent the data, as we have already realized in *Chapter 3*, *Finding Objects via Feature Matching and Perspective Transforms*. Instead, we need to derive a measurable property of the data that is more informative for classification.

However, often it is not clear which features would perform best. Instead, it is often necessary to experiment with different features that the modeler finds appropriate. After all, the choice of features might strongly depend on the specific dataset to be analyzed or the specific classification task to be performed. For example, if you have to distinguish between a stop sign and a warning sign, then the most telling feature might be the shape of the sign or the color scheme. However, if you have to distinguish between two warning signs, then color and shape will not help you at all, and you will be required to come up with more sophisticated features.

In order to demonstrate how the choice of features affects classification performance, we will focus on the following:

- A few simple color transformations, such as grayscale, RGB, and HSV. Classification based on grayscale images will give us some baseline performance for the classifier. RGB might give us slightly better performance because of the distinct color schemes of some traffic signs. Even better performance is expected from HSV. This is because it represents colors even more robustly than RGB. Traffic signs tend to have very bright, saturated colors that (ideally) are quite distinct from their surroundings.
- **Speeded-Up Robust Features** (**SURF**), which should appear very familiar to you by now. We have previously recognized SURF as an efficient and robust method of extracting meaningful features from an image, so can't we use this technique to our advantage in a classification task?
- **Histogram of Oriented Gradients** (**HOG**), which is by far the most advanced feature descriptor to be considered in this chapter. The technique counts occurrences of gradient orientations along a dense grid laid out on the image, and is well-suited for use with SVMs.

Feature extraction is performed by the `gtsrb._extract_features` function, which is implicitly called by `gtsrb.load_data`. It extracts different features as specified by the `feature` input argument.

The easiest case is not to extract any features, instead simply resizing the image to a suitable size:

```
def _extract_feature(X, feature):
    # operate on smaller image
    small_size = (32, 32)
    X = [cv2.resize(x, small_size) for x in X]
```

For most of the following features, we will be using the (already suitable) default arguments in OpenCV. However, these values are not set in stone, and even in real-world classification tasks, it is often necessary to search across the range of possible values for both feature extracting and learning parameters in a process called **hyperparameter exploration**.

Common preprocessing

There are three common forms of preprocessing that are almost always applied to any data before classification: **mean subtraction**, **normalization**, and **principal component analysis** (**PCA**). In this chapter, we will focus on the first two.

Mean subtraction is the most common form of preprocessing (sometimes also referred to as **zero-centering** or **de-meaning**), where the mean value of every feature dimension is calculated across all samples in a dataset. This feature-wise average is then subtracted from every sample in the dataset. You can think of this process as centering the *cloud* of data on the origin. Normalization refers to the scaling of data dimensions so that they are of roughly the same scale. This can be achieved by either dividing each dimension by its standard deviation (once it has been zero-centered), or scaling each dimension to lie in the range of *[-1, 1]*. It makes sense to apply this step only if you have reason to believe that different input features have different scales or units. In the case of images, the relative scales of pixels are already approximately equal (and in the range of *[0, 255]*), so it is not strictly necessary to perform this additional preprocessing step.

In this chapter, the idea is to enhance the local intensity contrast of images so that we do not focus on the overall brightness of an image:

```
# normalize all intensities to be between 0 and 1
X = np.array(X).astype(np.float32) / 255

# subtract mean
X = [x - np.mean(x) for x in X]
```

Grayscale features

The easiest feature to extract is probably the grayscale value of each pixel. Usually, grayscale values are not very indicative of the data they describe, but we will include them here for illustrative purposes (that is, to achieve baseline performance):

```
if feature == 'gray' or feature == 'surf':
    X = [cv2.cvtColor(x, cv2.COLOR_BGR2GRAY) for x in X]
```

Color spaces

Alternatively, you might find that colors contain some information that raw grayscale values cannot capture. Traffic signs often have a distinct color scheme, and it might be indicative of the information it is trying to convey (that is, red for stop signs and forbidden actions, green for informational signs, and so on). We could opt for using the RGB images as input, in which case we do not have to do anything, since the dataset is already RGB.

However, even RGB might not be informative enough. For example, a stop sign in broad daylight might appear very bright and clear, but its colors might appear much less vibrant on a rainy or foggy day. A better choice might be the HSV color space, which rearranges RGB color values in a cylindrical coordinate space along the axes of **hue**, **saturation**, and **value** (or brightness). The most telling feature of traffic signs in this color space might be the hue (a more perceptually relevant description of color or chromaticity), better distinguishing the color scheme of different sign types. Saturation and value could be equally important, however, as traffic signs tend to use relatively bright and saturated colors that do not typically appear in natural scenes (that is, their surroundings).

In OpenCV, the HSV color space is only a single call to `cv2.cvtColor` away:

```
if feature == 'hsv':
    X = [cv2.cvtColor(x, cv2.COLOR_BGR2HSV) for x in X]
```

Speeded Up Robust Features

But wait a minute! In *Chapter 3, Finding Objects via Feature Matching and Perspective Transforms* you learned that the SURF descriptor is one of the best and most robust ways to describe images independent of scale or rotations. Can we use this technique to our advantage in a classification task?

Glad you asked! To make this work, we need to adjust SURF so that it returns a fixed number of features per image. By default, the SURF descriptor is only applied to a small list of *interesting* keypoints in the image, the number of which might differ on an image-by-image basis. This is unsuitable for our current purposes, because we want to find a fixed number of feature values per data sample.

Instead, we need to apply SURF to a fixed dense grid laid out over the image, which can be achieved by creating a *dense* feature detector:

```
if feature == 'surf':
    # create dense grid of keypoints
    dense = cv2.FeatureDetector_create("Dense")
    kp = dense.detect(np.zeros(small_size).astype(np.uint8))
```

Then it is possible to obtain SURF descriptors for each point on the grid and append that data sample to our feature matrix. We initialize SURF with a `minHessian` value of 400 as we did before, and:

```
surf = cv2.SURF(400)
surf.upright = True
surf.extended = True
```

Keypoints and descriptors can then be obtained via this code:

```
kp_des = [surf.compute(x, kp) for x in X]
```

Because `surf.compute` has two output arguments, `kp_des` will actually be a concatenation of both keypoints and descriptors. The second element in the `kp_des` array is the descriptor that we care about. We select the first `num_surf_features` from each data sample and add it back to the training set:

```
num_surf_features = 36
X = [d[1][:num_surf_features, :] for d in kp_des]
```

Histogram of Oriented Gradients

The last feature descriptor to consider is the Histogram of Oriented Gradients (HOG). HOG features have previously been shown to work exceptionally well in combination with SVMs, especially when applied to tasks such as pedestrian recognition.

The essential idea behind HOG features is that the local shapes and appearance of objects within an image can be described by the distribution of edge directions. The image is divided into small connected regions, within which a histogram of gradient directions (or edge directions) is compiled. Then, the descriptor is assembled by concatenating the different histograms. For improved performance, the local histograms can be contrast-normalized, which results in better invariance to changes in illumination and shadowing. You can see why this sort of preprocessing might just be the perfect fit for recognizing traffic signs under different viewing angles and lighting conditions.

The HOG descriptor is fairly accessible in OpenCV by means of `cv2.HOGDescriptor`, which takes the detection window size (32 x 32), the block size (16 x 16), the cell size (8 x 8), and the cell stride (8 x 8), as input arguments. For each of these cells, the HOG descriptor then calculates a histogram of oriented gradients using nine bins:

```
elif feature == 'hog':
    # histogram of oriented gradients
    block_size = (small_size[0] / 2, small_size[1] / 2)
    block_stride = (small_size[0] / 4, small_size[1] / 4)
    cell_size = block_stride
    num_bins = 9
    hog = cv2.HOGDescriptor(small_size, block_size,
        block_stride, cell_size, num_bins)
```

Applying the HOG descriptor to every data sample is then as easy as calling `hog.compute`:

```
X = [hog.compute(x) for x in X]
```

After we have extracted all the features we want, we should remember to have `gtsrb._extract_features` return the assembled list of data samples so that they can be split into training and test sets:

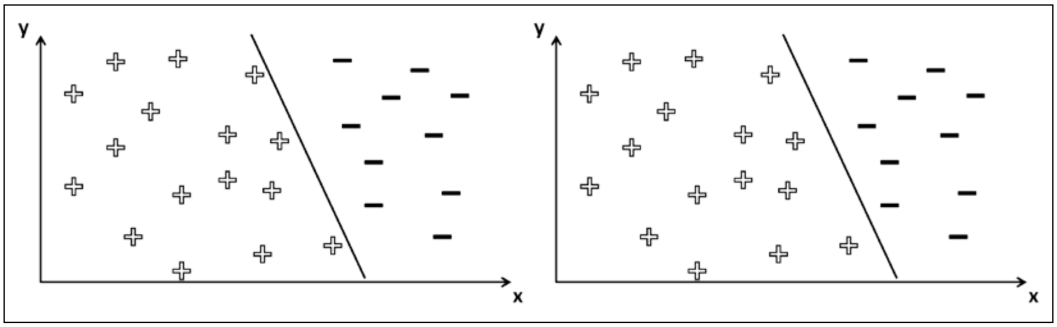
```
X = [x.flatten() for x in X]
return X
```

Now, we are finally ready to train the classifier on the preprocessed dataset.

Support Vector Machine

A Support Vector Machine (SVM) is a learner for binary classification (and regression) that tries to separate examples from the two different class labels with a decision boundary that maximizes the margin between the two classes.

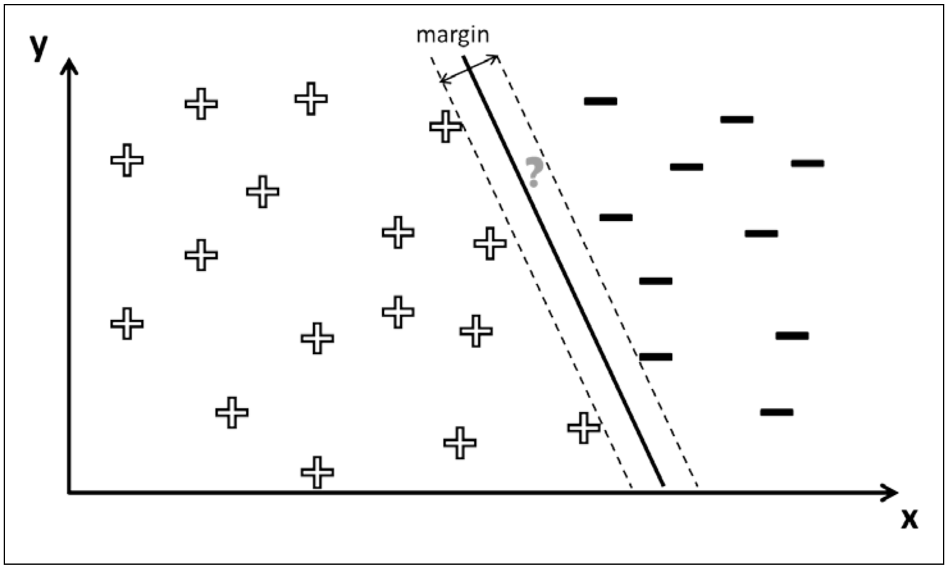
Let's return to our example of positive and negative data samples, each of which has exactly two features (**x** and **y**) and two possible decision boundaries, as follows:



Both of these decision boundaries get the job done. They partition all the samples of positives and negatives with zero misclassifications. However, one of them seems intuitively better. How can we quantify "better" and thus learn the "best" parameter settings?

This is where SVMs come in. SVMs are also called maximal margin classifiers because they can be used to do exactly that; they define the decision boundary so as to make those two clouds of **+** and **–** as far apart as possible.

For the preceding example, an SVM would find two lines that pass through the data points on the class margins (the dashed lines in the following figure), and then make the line that passes through the center of the margins, the decision boundary (the bold black line in the following figure):



It turns out that to find the maximal margin, it is only important to consider the data points that lie on the class margins. These points are sometimes also called **support vectors**.

> In addition to performing linear classification (that is, when the decision boundary is a straight line), SVMs can also perform a non-linear classification using what is called the **kernel trick**, implicitly mapping their inputs to high-dimensional feature spaces.

Using SVMs for Multi-class classification

Whereas some classification algorithms, such as neural networks, naturally lend themselves to using more than two classes, SVMs are binary classifiers by nature. They can, however, be turned into multiclass classifiers.

Here, we will consider two different strategies:

- **one-vs-all**: The one-vs-all strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. For `k` classes, this strategy thus requires the training of `k` different SVMs. During testing, all classifiers can express a "+1" vote by predicting that an unseen sample belongs to their class. In the end, an unseen sample is classified by the ensemble as the class with the most votes. Usually, this strategy is used in combination with confidence scores instead of predicted labels so that in the end, the class with the highest confidence score can be picked.
- **one-vs-one**: The one-vs-one strategy involves training a single classifier per class pair, with the samples of the first class as positive samples and the samples of the second class as negative samples. For `k` classes, this strategy requires the training of `k*(k-1)/2` classifiers. However, the classifiers have to solve a significantly easier task, so there is a trade-off when considering which strategy to use. During testing, all classifiers can express a "+1" vote for either the first or the second class. In the end, an unseen sample is classified by the ensemble as the class with the most votes.

Which strategy to use can be specified by the user via an input argument (`mode`) to the `MutliClassSVM` class:

```
class MultiClassSVM(Classifier):
    """Multi-class classification using Support Vector Machines
       (SVMs) """
    def __init__(self, num_classes, mode="one-vs-all",
        params=None):
        self.num_classes = num_classes
        self.mode = mode
        self.params = params or dict()
```

As mentioned earlier, depending on the classification strategy, we will need either `k` or `k*(k-1)/2` SVM classifiers, for which we will maintain a list in `self.classifiers`:

```
        # initialize correct number of classifiers
        self.classifiers = []
        if mode == "one-vs-one":
            # k classes: need k*(k-1)/2 classifiers
            for i in xrange(numClasses*(numClasses-1)/2):
                self.classifiers.append(cv2.SVM())
        elif mode == "one-vs-all":
            # k classes: need k classifiers
            for i in xrange(numClasses):
                self.classifiers.append(cv2.SVM())
        else:
            print "Unknown mode ",mode
```

Once the classifiers are correctly initialized, we are ready for training.

Training the SVM

Following the requirements defined by the `Classifier` base class, we need to perform training in a `fit` method:

```
def fit(self, X_train, y_train, params=None):
    """ fit model to data """
    if params is None:
        params = self.params
```

The training will differ depending on the classification strategy that is being used. The one-vs-one strategy requires us to train an SVM for each pair of classes:

```
if self.mode == "one-vs-one":
    svm_id=0
    for c1 in xrange(self.numClasses):
        for c2 in xrange(c1+1,self.numClasses):
```

Here we use `svm_id` to keep track of the number of SVMs we use. In contrast to the one-vs-all strategy, we need to train a much larger number of SVMs here. However, the training samples to consider per SVM include only samples of either class—`c1` or `c2`:

```
y_train_c1 = np.where(y_train==c1)[0]
y_train_c2 = np.where(y_train==c2)[0]

data_id = np.sort(np.concatenate((y_train_c1,
    y_train_c2), axis=0))
X_train_id = X_train[data_id,:]
y_train_id = y_train[data_id]
```

Because an SVM is a binary classifier, we need to convert our integer class labels into 0s and 1s. We assign label `1` to all samples of the `c1` class, and label `0` to all samples of the `c2` class, again using the handy `np.where` function:

```
y_train_bin = np.where(y_train_id==c1, 1,
    0).flatten()
```

Then we are ready to train the SVM:

```
self.classifiers[svm_id].train(X_train_id,
    y_train_bin, params=self.params)
```

Here, we pass a dictionary of training parameters, `self.params`, to the SVM. For now, we only consider the (already suitable) default parameter values, but feel free to experiment with different settings.

Don't forget to update `svm_id` so that you can move on to the next SVM in the list:

```
svm_id += 1
```

On the other hand, the one-vs-all strategy requires us to train a total of `self.numClasses` SVMs, which makes indexing a lot easier:

```
elif self.mode == "one-vs-all":
    for c in xrange(self.numClasses):
```

Again, we need to convert integer labels to binary labels. In contrast to the one-vs-one strategy, every SVM here considers all training samples. We assign label `1` to all samples of the `c` class and label `0` to all other samples, and pass the data to the classifier's training method:

```
y_train_bin = np.where(y_train==c,1,0).flatten()
self.classifiers[c].train(X_train, y_train_bin,
    params=self.params)
```

OpenCV will take care of the rest. What happens under the hood is that the SVM training uses Lagrange multipliers to optimize some constraints that lead to the maximum-margin decision boundary. The optimization process is usually performed until some termination criteria are met, which can be specified via the SVM's optional arguments:

```
params.term_crit = (cv2.TERM_CRITERIA_EPS +
    cv2.TERM_CRITERIA_MAX_ITER, 100, 1e-6)
```

Testing the SVM

There are many ways to evaluate a classifier, but most often, we are simply interested in the accuracy metric, that is, how many data samples from the test set were classified correctly.

In order to arrive at this metric, we need to have each individual SVM predict the labels of the test data, and assemble their consensus in a 2D voting matrix (Y_vote):

```
def evaluate(self, X_test, y_test, visualize=False):
    """Evaluates model performance"""
    Y_vote = np.zeros((len(y_test), self.numClasses))
```

For each sample in the dataset, the voting matrix will contain the number of times the sample has been voted to belong to a certain class. Populating the voting matrix will take a slightly different form depending on the classification strategy. In the case of the one-vs-one strategy, we need to loop over all k*(k-1)/2 classifiers and obtain a vector, called y_hat, that contains the predicted labels for all test samples that belong to either the c1 class or the c2 class:

```
if self.mode == "one-vs-one":
    svm_id = 0
    for c1 in xrange(self.numClasses):
        for c2 in xrange(c1+  1, self.numClasses):
            data_id = np.where((y_test==c1) + (y_test==c2))[0]
            X_test_id = X_test[data_id,:],:],:]
            y_test_id = y_test[data_id]

            # predict labels
            y_hat = self.classifiers[svm_id].predict_all
                ( X_test_id)
```

The y_hat vector will contain 1s whenever the classifier believes that the sample belongs to the c1 class, and 0s wherever the classifier believes that the sample belongs to the c2 class. The tricky part is how to +1 the correct cell in the Y_vote matrix. For example, if the *ith* entry in y_hat is 1 (meaning that we believe that the *ith* sample belongs to the c1 class), we want to increment the value of the *ith* row and *c1th* column in the Y_vote matrix. This will indicate that the present classifier expressed a vote to classify the *ith* sample as belonging to the c1 class.

Since we know the indices of all test samples that belong to either class, c1 or c2 (stored in data_id), we know which rows of Y_vote to access. Because data_id is used as an index for Y_vote, we only need to find the indices in data_id that correspond to entries where y_hat is 1:

```
            # we vote for c1 where y_hat is 1, and for c2 where
            # y_hat is 0
            # np.where serves as the inner index into the
            # data_id array, which in turn serves as index
            # into the Y_vote matrix
            Y_vote[data_id[np.where(y_hat==1)[0]],c1] += 1
```

Similarly, we can express a vote for the c2 class:

```
            Y_vote[data_id[np.where(y_hat==0)[0]],c2] += 1
```

Then we increment svm_id and move on to the next classifier:

```
            svm_id += 1
```

The one-vs-all strategy poses a different problem. Indexing in the Y_vote matrix is less tricky, because we always consider all the test data samples. We repeat the procedure of predicting labels for each data sample:

```
elif self.mode == "one-vs-all":
    for c in xrange(self.numClasses):
        # predict labels
        y_hat = self.classifiers[c].predict_all(X_test)
```

Wherever y_hat has a value of 1, the classifier expresses a vote that the data sample belongs to class c, and we update the voting matrix:

```
        # we vote for c where y_hat is 1
        if np.any(y_hat):
            Y_vote[np.where(y_hat==1)[0], c] += 1
```

The tricky part now is, "What to do with entries of y_hat that have a value of 0?" Since we classified one-vs-all, we only know that the sample is not of the c class (that is, it belongs to the "rest"), but we do not know what the exact class label is supposed to be. Thus, we cannot add these samples to the voting matrix.

Since we neglected to include samples that are consistently classified as belonging to the "rest," it is possible that some rows in the Y_vote matrix will have all 0s. In such a case, simply pick a class at random (unless you have a better idea):

```
        # find all rows without votes, pick a class at random
        no_label = np.where(np.sum(y_vote,axis=1)==0)[0]
        Y_vote[no_label,np.random.randint(self.numClasses,
            size=len(no_label))] = 1
```

Now, we are ready to calculate the desired performance metrics as described in detail in later sections. For the purpose of this chapter, we choose to calculate accuracy, precision, and recall, which are implemented in their own dedicated private methods:

```
accuracy = self.__accuracy(y_test, Y_vote)
precision = self.__precision(y_test, Y_vote)
recall = self.__recall(y_test, Y_vote)

return (accuracy,precision,recall)
```

The scikit-learn machine learning package (http://scikit-learn.org) supports the three metrics—accuracy, precision, and recall (as well as others)—straight out of the box, and also comes with a variety of other useful tools. For educational purposes (and to minimize software dependencies), we will derive the three metrics ourselves.

Confusion matrix

A confusion matrix is a 2D matrix of size equal to (self.numClasses, self.numClasses), where the rows correspond to the predicted class labels, and columns correspond to the actual class labels. Then, the [r,c] matrix element contains the number of samples that were predicted to have label r, but in reality have label c. Having access to a confusion matrix will allow us to calculate precision and recall.

The confusion matrix can be calculated from a vector of ground-truth labels (y_test) and the voting matrix (Y_vote). We first convert the voting matrix into a vector of predicted labels by picking the column index (that is, the class label) that received the most votes:

```
def __confusion(self, y_test, Y_vote):
    y_hat = np.argmax(Y_vote, axis=1)
```

Then we need to loop twice over all classes and count the number of times a data sample of the c_true ground-truth class was predicted as having the c_pred class:

```
    conf = np.zeros((self.numClasses,
        self.numClasses)).astype(np.int32)
    for c_true in xrange(self.numClasses):
        # looking at all samples of a given class, c_true
        # how many were classified as c_true? how many as others?
        for c_pred in xrange(self.numClasses):
```

All elements of interest in each iteration are thus the samples that have the c_true label in y_test and the c_pred label in y_hat:

```
            y_this = np.where((y_test==c_true) * (y_hat==c_pred))
```

The corresponding cell in the confidence matrix is then the number of non-zero elements in y_this:

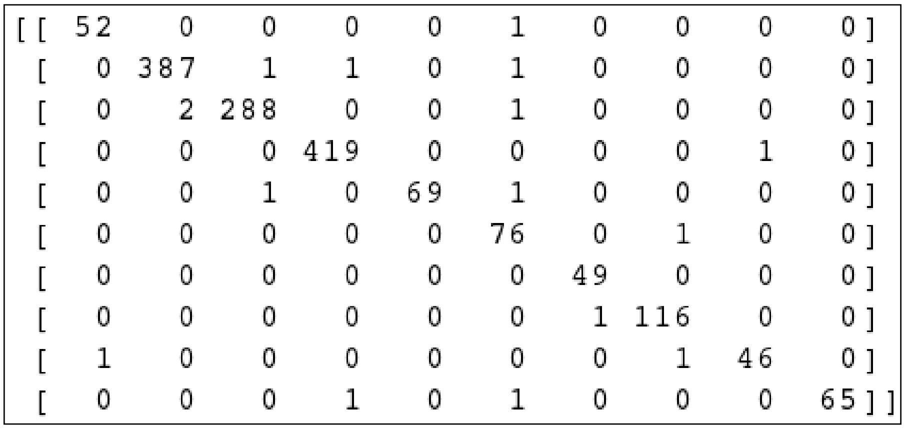
```
            conf[c_pred,c_true] = np.count_nonzero(y_this)
```

After the nested loops complete, we pass the confusion matrix for further processing:

```
    return conf
```

As you may have guessed already, the goal of a good classifier is to make the confusion matrix diagonal, which would imply that the ground-truth class (`c_true`) and the predicted class (`c_pred`) of every sample are the same.

The one-vs-one strategy, in combination with HOG features, actually performs really well, which is evident from this resulting confusion matrix, wherein most of the off-diagonal elements are zero:



```
[[ 52   0   0   0   0   1   0   0   0   0]
 [  0 387   1   1   0   1   0   0   0   0]
 [  0   2 288   0   0   1   0   0   0   0]
 [  0   0   0 419   0   0   0   0   1   0]
 [  0   0   1   0  69   1   0   0   0   0]
 [  0   0   0   0   0  76   0   1   0   0]
 [  0   0   0   0   0   0  49   0   0   0]
 [  0   0   0   0   0   0   1 116   0   0]
 [  1   0   0   0   0   0   0   1  46   0]
 [  0   0   0   1   0   1   0   0   0  65]]
```

Accuracy

The most straightforward metric to calculate is probably accuracy. This metric simply counts the number of test samples that have been predicted correctly, and returns the number as a fraction of the total number of test samples:

```
def __accuracy(self, y_test, y_vote):
    """ Calculates the accuracy based on a vector of ground-truth
        labels (y_test) and a 2D voting matrix (y_vote) of size
        (len(y_test),numClasses). """
```

The classification prediction (`y_hat`) can be extracted from the vote matrix by finding out which class has received the most votes:

```
    y_hat = np.argmax(y_vote, axis=1)
```

The correctness of the prediction can be verified by comparing the predicted label of a sample to its actual label:

```
    # all cases where predicted class was correct
    mask = (y_hat == y_test)
```

Accuracy is then calculated by counting the number of correct predictions (that is, non-zero entries in mask) and normalizing that number by the total number of test samples:

```
        return np.count_nonzero(mask)*1./len(y_test)
```

Precision

Precision in binary classification is a useful metric for measuring the fraction of retrieved instances that are relevant (also called the **positive predictive value**). In a classification task, the number of **true positives** is defined as the number of items correctly labeled as belonging to the positive class. **Precision** is defined as the number of true positives divided by the total number of positives. In other words, out of all the pictures in the test set that a classifier thinks contain a cat, precision is the fraction of pictures that actually contain a cat.

The total number of positives can also be calculated as the sum of true positives and **false positives**, the latter being the number of samples incorrectly labeled as belonging to a particular class. This is where the confusion matrix comes in handy, because it will allow us to quickly calculate the number of false positives.

Again, we start by translating the voting matrix into a vector of predicted labels:

```
    def __precision(self, y_test, Y_vote):
        """ precision extended to multi-class classification """
        # predicted classes
        y_hat = np.argmax(Y_vote, axis=1)
```

The procedure will differ slightly depending on the classification strategy. The one-vs-one strategy requires us operating with the confusion matrix:

```
        if True or self.mode == "one-vs-one":
            # need confusion matrix
            conf = self.__confusion(y_test, y_vote)

            # consider each class separately
            prec = np.zeros(self.numClasses)
            for c in xrange(self.numClasses):
```

Since true positives are the samples that are predicted to have label c and also have label c in reality, they correspond to the diagonal elements of the confusion matrix:

```
                # true positives: label is c, classifier predicted c
                tp = conf[c,c]
```

Similarly, false positives correspond to the off-diagonal elements of the confusion matrix that are in the same column as the true positive. The quickest way to calculate that number is to sum up all the elements in column `c` but subtract the true positives:

```
                # false positives: label is c, classifier predicted
                # not c
                fp = np.sum(conf[:,c]) - conf[c,c]
```

Precision is the number of true positives divided by the sum of true positives and false positives:

```
        if tp + fp != 0:
            prec[c] = tp*1./(tp+fp)
```

The one-vs-all strategy makes the math slightly easier. Since we always operate on the full test set, we need to find only those samples that have a ground-truth label of `c` in `y_test` and a predicted label of `c` in `y_hat`:

```
        elif self.mode == "one-vs-all":
            # consider each class separately
            prec = np.zeros(self.numClasses)
            for c in xrange(self.numClasses):
                # true positives: label is c, classifier predicted c
                tp = np.count_nonzero((y_test==c) * (y_hat==c))
```

Similarly, false positives have a ground-truth label of `c` in `y_test` and their predicted label is not `c` in `y_hat`:

```
                # false positives: label is c, classifier predicted
                # not c
                fp = np.count_nonzero((y_test==c) * (y_hat!=c))
```

Again, precision is the number of true positives divided by the sum of true positives and false positives:

```
                if tp + fp != 0:
                    prec[c] = tp*1./(tp+fp)
```

After that, we pass the precision vector for visualization:

```
        return prec
```

Recall

Recall is similar to precision in the sense that it measures the fraction of relevant instances that are retrieved (as opposed to the fraction of retrieved instances that are relevant). In a classification task, the number of false negatives is the number of items not labeled as belonging to the positive class but should have been labeled. Recall is the number of true positives divided by the sum of true positives and false negatives. In other words, out of all the pictures of cats in the world, recall is the fraction of pictures that have been correctly identified as pictures of cats.

Again, we start off by translating the voting matrix into a vector of predicted labels:

```
def __recall(self, y_test, Y_vote):
    """ recall extended to multi-class classification """
    # predicted classes
    y_hat = np.argmax(Y_vote, axis=1)
```

The procedure is almost identical to calculating precision. The one-vs-one strategy once again requires some arithmetic involving the confusion matrix:

```
    if True or self.mode == "one-vs-one":
        # need confusion matrix
        conf = self.__confusion(y_test, y_vote)

        # consider each class separately
        recall = np.zeros(self.numClasses)
        for c in xrange(self.numClasses):
```

True positives once again correspond to diagonal elements in the confusion matrix:

```
            # true positives: label is c, classifier predicted c
            tp = conf[c,c]
```

To get the number of false negatives, we sum up all the columns in the row `c` and subtract the number of true positives for this row. This will give us the number of samples for which the classifier predicted the class as `c` but that actually had a ground-truth label other than `c`:

```
            # false negatives: label is not c, classifier
            # predicted c
            fn = np.sum(conf[c,:]) - conf[c,c]
```

Recall is the number of true positives divided by the sum of true positives and false negatives:

```
            if tp + fn != 0:
                recall[c] = tp*1./(tp+fn)
```

Again, the one-vs-all strategy makes the math slightly easier. Since we always operate on the full test set, we need to find only those samples whose ground-truth label is not `c` in `y_test`, and predicted label is `c` in `y_hat`:

```
        elif self.mode == "one-vs-all":
            # consider each class separately
            recall = np.zeros(self.numClasses)
            for c in xrange(self.numClasses):
                # true positives: label is c, classifier predicted c
                tp = np.count_nonzero((y_test==c) * (y_hat==c))

                # false negatives: label is not c, classifier
                # predicted c
                fn = np.count_nonzero((y_test!=c) * (y_hat==c))

                if tp + fn != 0:
                    recall[c] = tp*1./(tp+fn)
```

After that, we pass the recall vector for visualization:

```
        return recall
```

Putting it all together

To run our app, we will need to execute the main function routine (in `chapter6.py`). It loads the data, trains the classifier, evaluates its performance, and visualizes the result.

But first, we need to import all the relevant modules and set up a main function:

```
    import numpy as np

    import matplotlib.pyplot as plt
    from datasets import gtsrb
    from classifiers import MultiClassSVM


    def main():
```

Then, the goal is to compare classification performance across settings and feature extraction methods. This includes running the task with both classification strategies, one-vs-all and one-vs-one, as well as preprocessing the data with a list of different feature extraction approaches:

```
        strategies = ['one-vs-one', 'one-vs-all']features = [None,
            'gray', 'rgb', 'hsv', 'surf', 'hog']
```

For each of these settings, we need to collect three performance metrics—accuracy, precision, and recall:

```
        accuracy = np.zeros((2,len(features)))
        precision = np.zeros((2,len(features)))
        recall = np.zeros((2,len(features)))
```

A nested `for` loop will run the classifier with all of these settings and populate the performance metric matrices. The outer loop is over all elements in the `features` vector:

```
        for f in xrange(len(features)):
            (X_train,y_train), (X_test,y_test) = gtsrb.load_data(
                "datasets/gtsrb_training",
                feature=features[f], test_split=0.2)
```

Before passing the training data (`X_train`,`y_train`) and test data (`X_test`,`y_test`) to the classifiers, we want to make sure that they are in the format that the classifier expects; that is, each data sample is stored in a row of `X_train` or `X_test`, with the columns corresponding to feature values:

```
    X_train = np.squeeze(np.array(X_train)).astype(np.float32)
    y_train = np.array(y_train)
    X_test = np.squeeze(np.array(X_test)).astype(np.float32)
    y_test = np.array(y_test)
```

We also need to know the number of class labels (since we did not load the complete GTSRB dataset). This can be achieved by concatenating `y_train` and `y_test` and extracting all unique labels in the combined list:

```
    labels = np.unique(np.hstack((y_train,y_test)))
```

Next, the inner loop iterates over all the elements in the `strategies` vector, which currently includes the two strategies, one-vs-all and one-vs-one:

```
    for s in xrange(len(strategies)):
```

Then we instantiate the `MultiClassSVM` class with the correct number of unique labels and the corresponding classification strategy:

```
MCS = MultiClassSVM(len(labels),strategies[s])
```

Now we are all ready to apply the ensemble classifier to the training data and extract the three performance metrics after training:

```
MCS.fit(X_train, y_train)
(accuracy[s,f],precision[s,f],recall[s,f]) =
    MCS.evaluate(X_test, y_test)
```

This ends the nested `for` loop. All that is left to do is to visualize the result. For this, we choose matplotlib's bar plot functionality. The goal is to show the three performance metrics (accuracy, precision, and recall) for all combinations of extracted features and classification strategies. We will use one plotting window per classification strategy, and arrange the corresponding data in a stacked bar plot:

```
f,ax = plt.subplots(2)
for s in xrange(len(strategies)):
    x = np.arange(len(features))
    ax[s].bar(x-0.2, accuracy[s,:], width=0.2, color='b',
        hatch='/', align='center')
    ax[s].bar(x, precision[s,:], width=0.2, color='r',
        hatch='\\', align='center')
    ax[s].bar(x+0.2, recall[s,:], width=0.2, color='g',
        hatch='x', align='center')
```

For the sake of visibility, the *y* axis is restricted to the relevant range:

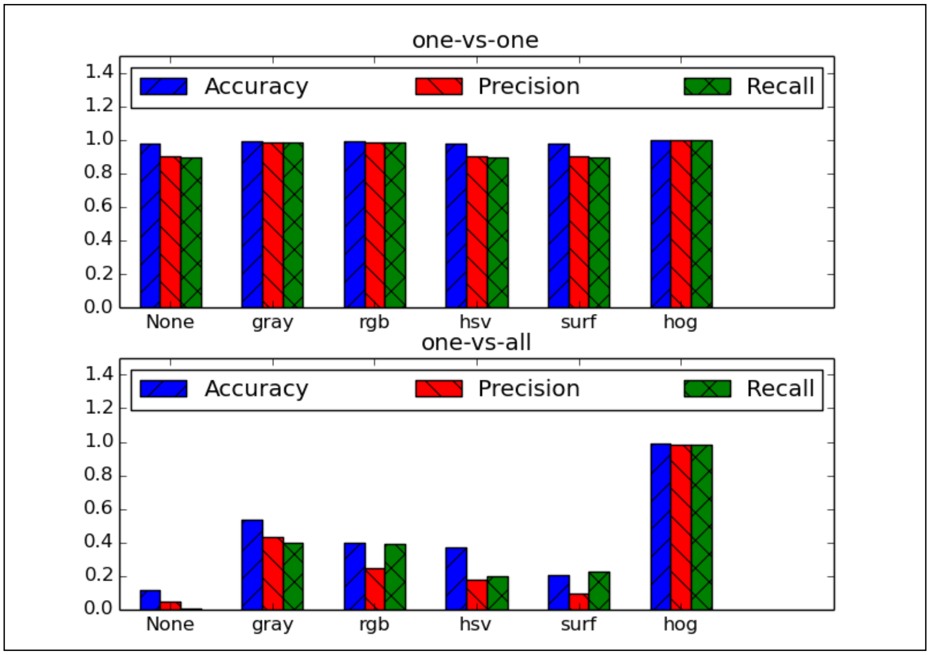
```
ax[s].axis([-0.5, len(features) + 0.5, 0, 1.5])
```

Finally, we add some plot decorations:

```
ax[s].legend(('Accuracy','Precision','Recall'), loc=2,
    ncol=3, mode="expand")
ax[s].set_xticks(np.arange(len(features)))
ax[s].set_xticklabels(features)
ax[s].set_title(strategies[s])
```

Now the data is ready to be plotted!

```
plt.show()
```



This screenshot contains a lot of information, so let's break it down step by step:

- The most straightforward observation is that the HOG feature seems mighty powerful! This feature has outperformed all other features, no matter what the classification strategy is. Again, this highlights the importance of feature extraction, which generally requires a deep understanding of the statistics of the dataset under study.
- Interestingly, with the use of the one-vs-one strategy, all approaches led to an accuracy north of 0.95, which might stem from the fact that a binary classification task (with two possible class labels) is sometimes easier to learn than a 10-class classification task. Unfortunately, the same cannot be said for the one-vs-all approach. But to be fair, the one-vs-all approach had to operate with only 10 SVMs, whereas the one-vs-one approach had 45 SVMs to work with. This gap is only likely to increase if we add more object categories.
- The effect of de-meaning can be seen by comparing the result for `None` with the result for `rgb`. These two settings were identical, except that the samples under `rgb` were normalized. The difference in performance is evident, especially for the one-vs-all strategy.

- A little disappointing is the finding that none of the color transformations (neither RGB nor HSV) were able to perform significantly better than the simple grayscale transform. SURF did not help either.

Summary

In this chapter, we trained a multiclass classifier to recognize traffic signs from the GTSRB database. We discussed the basics of supervised learning, explored the intricacies of feature extraction, and extended SVMs so that they can be used for multiclass classification.

Notably, we left out some details along the way, such as attempting to fine-tune the hyperparameters of the learning algorithm. When we restrict the traffic sign dataset to only 10 classes, the default values of the various function arguments along the way, seem to be sufficient for performing exceptionally well (just look at the perfect score achieved with the HOG feature and the one-vs-one strategy). With this functional setup and a good understanding of the underlying methodology, you can now try to classify the entire GTSRB dataset! It is definitely worth taking a look at their website, where you will find classification results for a variety of classifiers. Maybe, your own approach will soon be added to the list.

In the next (and last) chapter, we will move even deeper into the field of machine learning. Specifically, we will focus on recognizing emotional expressions in human faces using convolutional neural networks. This time, we will combine the classifier with a framework for object detection, which will allow us to localize (where?) a human face in an image, and then focus on identifying (what?) the emotional expression contained in that face. This will conclude our quest into the depths of machine learning, and provide you with all the necessary tools to develop your own advanced OpenCV projects using the principles and concepts of computer vision.

7
Learning to Recognize Emotions on Faces

We previously familiarized ourselves with the concepts of object detection and object recognition, but we never combined them to develop an app that can do both end-to-end. For the final chapter in this book, we will do exactly that.

The goal of this chapter is to develop an app that combines both **face detection** and **face recognition**, with a focus on recognizing emotional expressions in the detected face.

For this, we will touch upon two other classic algorithms that come bundled with OpenCV: **Haar Cascade Classifiers** and **multi-layer peceptrons** (**MLPs**). While the former can be used to rapidly detect (or locate, answering the question: where?) objects of various sizes and orientations in an image, the latter can be used to recognize them (or identify, answering the question: what?).

The end goal of the app will be to detect your own face in each captured frame of a webcam live stream and label your emotional expression. To make this task feasible, we will limit ourselves to the following possible emotional expressions: neutral, happy, sad, surprised, angry, and disgusted.

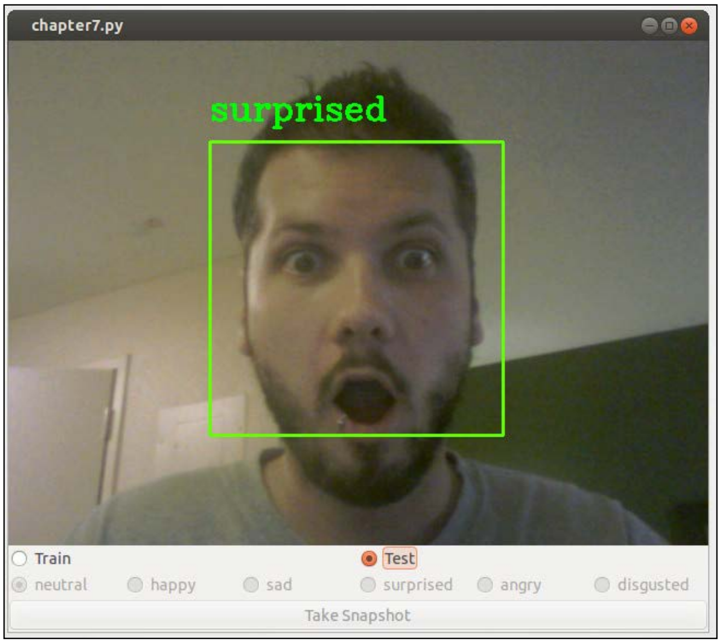
To arrive at such an app, we need to solve the following two challenges:

- **Face detection**: We will use the popular Haar cascade classifier by Viola and Jones, for which OpenCV provides a whole range of pre-trained exemplars. We will make use of face cascades and eye cascades to reliably detect and align facial regions from frame to frame.

- **Facial expression recognition**: We will train a multi-layer perceptron to recognize the six different emotional expressions listed earlier, in every detected face. The success of this approach will crucially depend on the training set that we assemble, and the preprocessing that we choose to apply to each sample in the set. In order to improve the quality of our self-recorded training set, we will make sure that all data samples are aligned using **affine transformations** and reduce the dimensionality of the feature space by applying **Principal Component Analysis** (**PCA**). The resulting representation is sometimes also referred to as **Eigenfaces**.

The reliable recognition of faces and facial expressions is a challenging task for artificial intelligence, yet humans are able to perform these kinds of tasks with apparent ease. Today's state-of-the-art models range all the way from 3D deformable face models fitting over convolutional neural networks, to deep learning algorithms. Granted, these approaches are significantly more sophisticated than our approach. Yet, MLPs are classic algorithms that helped transform the field of machine learning, so for educational purposes, we will stick to a set of algorithms that come bundled with OpenCV.

We will combine the algorithms mentioned earlier in a single end-to-end app that annotates a detected face with the corresponding facial expression label in each captured frame of a video live stream. The end result might look something like the following image, capturing my reaction when the code first compiled:



Planning the app

The final app will consist of a main script that integrates the process flow end-to-end, from face detection to facial expression recognition, as well as some utility functions to help along the way.

Thus, the end product will require several components:

- `chapter7`: The main script and entry-point for the chapter.
- `chapter7.FaceLayout`: A custom layout based on `gui.BaseLayout` that operates in two different modes:
    - Training mode: In the training mode, the app will collect image frames, detect a face therein, assign a label depending on the facial expression, and upon exiting, save all the collected data samples in a file, so that it can be parsed by `datasets.homebrew`.
    - Testing mode: In the testing mode, the app will detect a face in each video frame and predict the corresponding class label by using a pre-trained MLP.
- `chapter3.main`: The main function routine to start the GUI application.
- `detectors.FaceDetector`: A class for face detection.
    - `detect`: A method to detect faces in a grayscale image. Optionally, the image is downscaled for better reliability. Upon successful detection, the method returns the extracted head region.
    - `align_head`: A method to preprocess an extracted head region with affine transformations such that the resulting face appears centered and upright.
- `classifiers.Classifier`: An abstract base class that defines the common interface for all classifiers (same as in *Chapter 6*, *Learning to Recognize Traffic Signs*).
- `classifiers.MultiLayerPerceptron`: A class that implements an MLP by using the following public methods:
    - `fit`: A method to fit the MLP to the training data. It takes as input, a matrix of the training data, where each row is a training sample, and columns contain feature values, and a vector of labels.
    - `evaluate`: A method to evaluate the MLP by applying it to some test data after training. It takes as input, a matrix of test data, where each row is a test sample and columns contain feature values, and a vector of labels. The function returns three different performance metrics: accuracy, precision, and recall.

    - `predict`: A method to predict the class labels of some test data. We expose this method to the user so it can be applied to any number of data samples, which will be helpful in the testing mode, when we do not want to evaluate the entire dataset, but instead predict the label of only a single data sample.
    - `save`: A method to save a trained MLP to file.
    - `load`: A method to load a pre-trained MLP from file.
- `train_test_mlp`: A script to train and test an MLP by applying it to our self-recorded dataset. The script will explore different network architectures and store the one with the best generalization performance in a file, so that the pre-trained classifier can be loaded later.
- `datasets.homebrew`: A class to parse the self-recorded training set. Analogously to the previous chapter, the class contains the following methods:
    - `load_data`: A method to load the training set, perform PCA on it via the `extract_features` function, and split the data into the training and test sets. Optionally, the preprocessed data can be stored in a file so that we can load it later on without having to parse the data again.
    - `load_from_file`: A method to load a previously stored preprocessed dataset.
    - `extract_features`: A method to extract a feature of choice (in the present chapter: to perform PCA on the data). We expose this function to the user so it can be applied to any number of data samples, which will be helpful in the testing mode, when we do not want to parse the entire dataset but instead predict the label of only a single data sample.
- `gui`: A module providing a wxPython GUI application to access the capture device and display the video feed. This is the same module that we used in the previous chapters.
    - `gui.BaseLayout`: A generic layout from which more complicated layouts can be built. This chapter does not require any modifications to the basic layout.

In the following sections, we will discuss these components in detail.

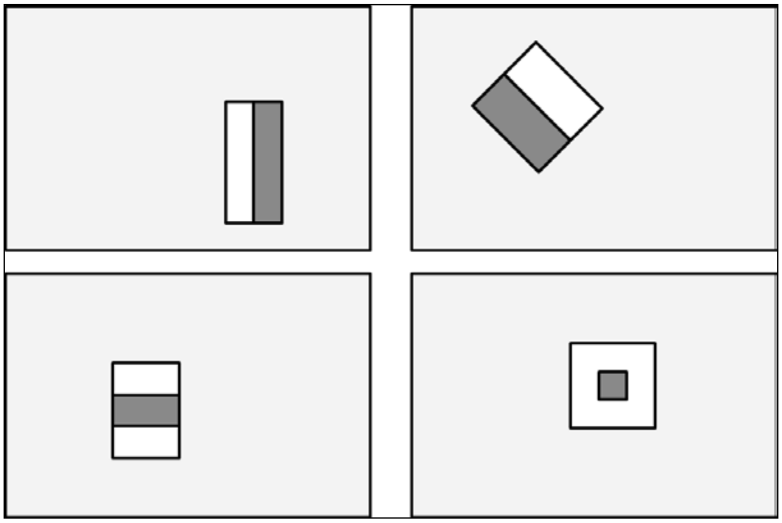
Face detection

OpenCV comes preinstalled with a range of sophisticated classifiers for general-purpose object detection. Perhaps, the most commonly known detector is the **cascade of Haar-based feature detectors** for face detection, which was invented by Paul Viola and Michael Jones.

Haar-based cascade classifiers

Every book on OpenCV should at least mention the Viola–Jones face detector. Invented in 2001, this cascade classifier disrupted the field of computer vision, as it finally allowed real-time face detection and face recognition.

The classifier is based on Haar-like features (similar to Haar basis functions), which sum up the pixel intensities in small regions of an image, as well as capture the difference between adjacent image regions. Some example rectangle features are shown in the following figure, relative to the enclosing (light gray) detection window:



Here, the top row shows two examples of an edge feature, either vertically oriented (left) or oriented at a 45 degree angle (right). The bottom row shows a line feature (left) and a center-surround feature (right). The feature value for each of these is then calculated by summing up all pixel values in the dark gray rectangle and subtracting this value from the sum of all pixel values in the white rectangle. This procedure allowed the algorithm to capture certain qualities of human faces, such as the fact that eye regions are usually darker than the region surrounding the cheeks.

Thus, a common Haar feature would have a dark rectangle (representing the eye region) atop a bright rectangle (representing the cheek region). Combining this feature with a bank of rotated and slightly more complicated *wavelets*, Viola and Jones arrived at a powerful feature descriptor for human faces. In an additional act of genius, these guys came up with an efficient way to calculate these features, making it possible for the first time to detect faces in real-time.

Pre-trained cascade classifiers

Even better, this approach does not only work for faces but also for eyes, mouths, full bodies, company logos, you name it. A number of pre-trained classifiers can be found under the OpenCV install path in the `data` folder:

Cascade classifier type	**XML file name**
Face detector (default)	`haarcascade_frontalface_default.xml`
Face detector (fast Haar)	`haarcascade_frontalface_alt2.xml`
Eye detector	`haarcascade_lefteye_2splits.xml` `haarcascade_righteye_2splits.xml`
Mouth detector	`haarcascade_mcs_mouth.xml`
Nose detector	`haarcascade_mcs_nose.xml`
Full body detector	`haarcascade_fullbody.xml`

In this chapter, we will use `haarcascade_frontalface_default.xml`, `haarcascade_lefteye_2splits.xml`, and `haarcascade_righteye_2splits.xml`.

> If you are wearing glasses, make sure to use `haarcascade_eye_tree_eyeglasses.xml` on both eyes instead.

Using a pre-trained cascade classifier

A cascade classifier can be loaded and applied to a (grayscale!) image `frame` with the following code:

```
import cv2

frame = cv2.imread('example_grayscale.jpg', cv2.CV_8UC1)
face_casc =
    cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
faces = face_casc.detectMultiScale(frame, scaleFactor=1.1,
    minNeighbors=3)
```

The detectMultiScale function comes with a number of options:

- minFeatureSize: The minimum face size to consider (for example, 20 × 20 pixels).
- searchScaleFactor: Amount by which to rescale the image (scale pyramid). For example, a value of 1.1 will gradually reduce the size of the input image by 10 percent, making it more likely for a face to be found than a larger value.
- minNeighbors: The number of neighbors each candidate rectangle should have to retain it. Typically, choose 3 or 5.
- flags: Options for old cascades (will be ignored by newer ones). For example, whether to look for all faces or just the largest (cv2.cv.CASCADE_FIND_BIGGEST_OBJECT).

If detection is successful, the function will return a list of bounding boxes (faces) that contain the coordinates of the detected face regions:

```
for (x, y, w, h) in faces:
    # draw bounding box on frame
    cv2.rectangle(frame, (x, y), (x + w, y + h), (100, 255, 0), 2)
```

> If your pre-trained face cascade does not detect anything, a common reason is usually that the path to the pre-trained cascade file could not be found. In this case, CascadeClassifier will fail silently. Thus, it is always a good idea to check whether the returned classifier casc = cv2.CascadeClassifier(filename) is empty, by checking casc.empty().

The FaceDetector class

All relevant face detection code for this chapter can be found as part of the FaceDetector class in the detectors module. Upon instantiation, this class loads three different cascade classifiers that are needed for preprocessing: a face cascade and two eye cascades.

```
import cv2
import numpy as np


class FaceDetector:
    def __init__(
        self,
        face_casc='params/haarcascade_frontalface_default.xml',
        left_eye_casc='params/haarcascade_lefteye_2splits.xml',
        right_eye_casc='params/haarcascade_righteye_2splits.xml',
        scale_factor=4):
```

Because our preprocessing requires a valid face cascade, we make sure that the file can be loaded. If not, we print an error message and exit the program:

```
            self.face_casc = cv2.CascadeClassifier(face_casc)
            if self.face_casc.empty():
                print 'Warning: Could not load face cascade:',
                    face_casc
                raise SystemExit
```

For reasons that will become clear in just a moment, we also need two eye cascades, for which we proceed analogously:

```
            self.left_eye_casc = cv2.CascadeClassifier(left_eye_casc)
            if self.left_eye_casc.empty():
                print 'Warning: Could not load left eye cascade:',
                    left_eye_casc
                raise SystemExit
            self.right_eye_casc =
                cv2.CascadeClassifier(right_eye_casc)
            if self.right_eye_casc.empty():
                print 'Warning: Could not load right eye cascade:',
                    right_eye_casc
                raise SystemExit
```

Face detection works best on low-resolution grayscale images. This is why we also store a scaling factor (`scale_factor`) so that we can operate on downscaled versions of the input image if necessary:

```
    self.scale_factor = scale_factor
```

Detecting faces in grayscale images

Faces can then be detected using the `detect` method. Here, we ensure that we operate on a downscaled grayscale image:

```
    def detect(self, frame):
        frameCasc = cv2.cvtColor(cv2.resize(frame, (0, 0),
            fx=1.0 / self.scale_factor, fy=1.0 / self.scale_factor),
            cv2.COLOR_RGB2GRAY)
        faces = self.face_casc.detectMultiScale(frameCasc,
            scaleFactor=1.1, minNeighbors=3,
            flags=cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT) *
            self.scale_factor
```

If a face is found, we continue to extract the head region from the bounding box information and store the result in `head`:

```
for (x, y, w, h) in faces:
    head = cv2.cvtColor(frame[y:y + h, x:x + w],
        cv2.COLOR_RGB2GRAY)
```

We also draw the bounding box onto the input image:

```
cv2.rectangle(frame, (x, y), (x + w, y + h), (100, 255, 0), 2)
```

In case of success, the method should return a Boolean indicating success (`True`), the annotated input image (`frame`), and the extracted head region (`head`):

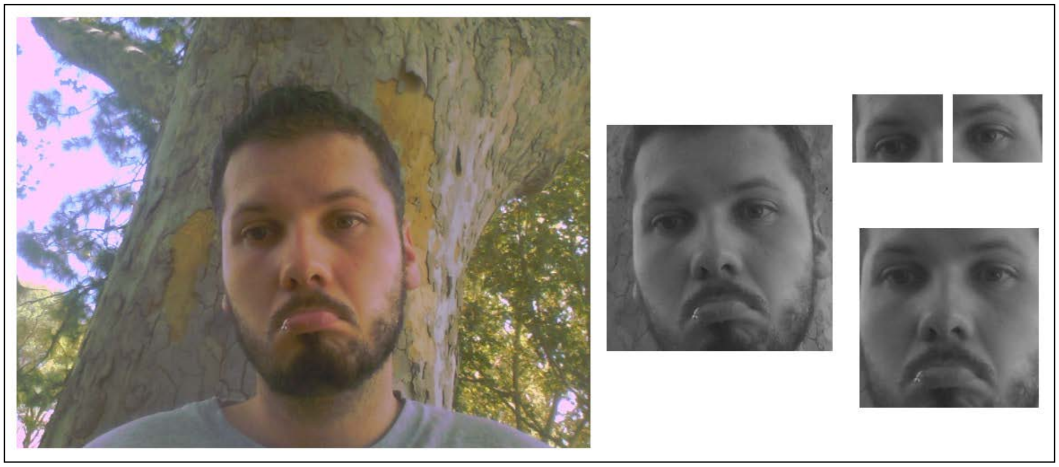
```
return True, frame, head
```

Otherwise, if no faces were detected, the method indicates failure with a Boolean (`False`) and returns the unchanged input image (`frame`) and `None` for the head region:

```
return False, frame, None
```

Preprocessing detected faces

After a face has been detected, we might want to preprocess the extracted head region before applying classification on it. Although the face cascade is fairly accurate, for recognition, it is important that all the faces are upright and centered on the image. This idea is best illustrated with an image. Consider a sad programmer under a tree:



Because of his emotional state, the programmer tends to keep his head slightly tilted to the side while looking down. The facial region as extracted by the face cascade is shown as the leftmost grayscale thumbnail on the right. In order to compensate for the head orientation, we aim to rotate and scale the face so that all data samples will be perfectly aligned. This is the job of the `align_head` method in the `FaceDetector` class:

```
def align_head(self, head):
    height, width = head.shape[:2]
```

Fortunately, OpenCV comes with a few eye cascades that can detect both open and closed eyes, such as `haarcascade_lefteye_2splits.xml` and `haarcascade_righteye_2splits.xml`. This allows us to calculate the angle between the line that connects the center of the two eyes and the horizon so that we can rotate the face accordingly. In addition, adding eye detectors will reduce the risk of having false positives in our dataset, allowing us to add a data sample only if both the head and the eyes have been successfully detected.

After loading these eye cascades from file in the FaceDetector constructor, they are applied to the input image (`head`):

```
    left_eye_region = head[0.2*height:0.5*height,
        0.1*width:0.5*width]
    left_eye = self.left_eye_casc.detectMultiScale(
        left_eye_region, scaleFactor=1.1, minNeighbors=3,
        flags=cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT)
```

Here, it is important that we pass only a small, relevant region (`left_eye_region`; compare small thumbnails in the top-right corner of the preceding figure) to the eye cascades. For simplicity, we use hardcoded values that focus on the top half of the facial region and assume the left eye to be in the left half.

If an eye is detected, we extract the coordinates of its center point:

```
left_eye_center = None
for (xl, yl, wl, hl) in left_eye:
    # find the center of the detected eye region
    left_eye_center = np.array([0.1 * width + xl + wl / 2,
        0.2 * height + yl + hl / 2])
    break # need only look at first, largest eye
```

Then, we proceed to do the same for the right eye:

```
right_eye_region = head[0.2*height:0.5*height,
    0.5*width:0.9*width]
right_eye = self.right_eye_casc.detectMultiScale(
    right_eye_region, scaleFactor=1.1, minNeighbors=3,
    flags=cv2.cv.CV_HAAR_FIND_BIGGEST_OBJECT)
right_eye_center = None
for (xr, yr, wr, hr) in right_eye:
    # find the center of the detected eye region
    right_eye_center = np.array([0.5 * width + xr + wr / 2,
        0.2 * height + yr + hr / 2])
    break  # need only look at first, largest eye
```

As mentioned earlier, if we do not detect both the eyes, we discard the sample as a false positive:

```
if left_eye_center is None or right_eye_center is None:
    return False, head
```

Now, this is where the magic happens. No matter how crooked the face that we detected is, before we add it to the training set, we want the eyes to be exactly at 25 percent and 75 percent of the image width (so that the face is in the center) and at 20 percent of the image height:

```
desired_eye_x = 0.25
desired_eye_y = 0.2
desired_img_width = 200
desired_img_height = desired_img_width
```

This can be achieved by warping the image using `cv2.warpAffine` (remember *Chapter 3*, *Finding Objects via Feature Matching and Perspective Transforms*?). First, we calculate the angle (in degrees) between the line that connects the two eyes and a horizontal line:

```
eye_center = (left_eye_center + right_eye_center) / 2
eye_angle_deg = np.arctan2(
    right_eye_center[1] - left_eye_center[1],
    right_eye_center[0] - left_eye_center[0]) *
    180.0 / cv2.cv.CV_PI
```

Then, we derive a scaling factor that will scale the distance between the two eyes to be exactly 50 percent of the image width:

```
eye_size_scale = (1.0 - desired_eye_x * 2) *
    desired_img_width / np.linalg.norm(
    right_eye_center - left_eye_center)
```

With these two parameters (`eye_angle_deg` and `eye_size_scale`) in hand, we can now come up with a suitable rotation matrix that will transform our image:

```
rot_mat = cv2.getRotationMatrix2D(tuple(eye_center),
eye_angle_deg, eye_size_scale)
```

We make sure that the center of the eyes will be centered in the image:

```
rot_mat[0,2] += desired_img_width*0.5 - eye_center[0]
rot_mat[1,2] += desired_eye_y*desired_img_height -
    eye_center[1]
```

Finally, we arrive at an upright scaled version of the facial region that looks like the lower-right thumbnail of the preceding image:

```
res = cv2.warpAffine(head, rot_mat,
    (desired_img_width, desired_img_height))
return True, res
```

Facial expression recognition

The facial expression recognition pipeline is encapsulated by `chapter7.py`. This file consists of an interactive GUI that operates in two modes (training and testing), as described earlier.

In order to arrive at our end-to-end app, we need to cover the following three steps:

1. Load the `chapter7.py` GUI in the training mode to assemble a training set.
2. Train an MLP classifier on the training set via `train_test_mlp.py`. Because this step can take a long time, the process takes place in its own script. After successful training, store the trained weights in a file, so that we can load the pre-trained MLP in the next step.
3. Load the `chapter7.py` GUI in the testing mode to classify facial expressions on a live video stream in real-time. This step involves loading several pre-trained cascade classifiers as well as our pre-trained MLP classifier. These classifiers will then be applied to every captured video frame.

Assembling a training set

Before we can train an MLP, we need to assemble a suitable training set. Because chances are, that your face is not yet part of any dataset out there (the NSA's private collection doesn't count), we will have to assemble our own. This is done most easily by returning to our GUI application from the previous chapters, which can access a webcam, and operate on each frame of a video stream.

The GUI will present the user with the option of recording one of the following six emotional expressions: neutral, happy, sad, surprised, angry, and disgusted. Upon clicking a button, the app will take a snapshot of the detected facial region, and upon exiting, it will store all collected data samples in a file. These samples can then be loaded from file and used to train an MLP classifier in `train_test_mlp.py`, as described in step two given earlier.

Running the screen capture

In order to run this app (`chapter7.py`), we need to set up a screen capture by using `cv2.VideoCapture` and pass the handle to the `FaceLayout` class:

```
import time
import wx
from os import path
import cPickle as pickle

import cv2
import numpy as np

from datasets import homebrew
from detectors import FaceDetector
from classifiers import MultiLayerPerceptron
from gui import BaseLayout


def main():
    capture = cv2.VideoCapture(0)
    if not(capture.isOpened()):
        capture.open()

    capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
    capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)

    # start graphical user interface
    app = wx.App()
    layout = FaceLayout(None, -1, 'Facial Expression Recognition',
        capture)
    layout.init_algorithm()
    layout.Show(True)
    app.MainLoop()

if __name__ == '__main__':
    main()
```

> If you happen to have installed some non-canonical releases of OpenCV, the frame width and frame weight parameters might have a slightly different name (for example, `cv3.CAP_PROP_FRAME_WIDTH`). However, in newer releases, it is the easiest to access the old OpenCV1 sub-module `cv` and its variables `cv2.cv.CV_CAP_PROP_FRAME_WIDTH` and `cv2.cv.CV_CAP_PROP_FRAME_HEIGHT`.

The GUI constructor

Analogous to the previous chapters, the GUI of the app is a customized version of the generic `BaseLayout`:

```
class FaceLayout(BaseLayout):
```

We initialize the training samples and labels as empty lists, and make sure to call the `_on_exit` method upon closing the window so that the training data is dumped to file:

```
    def _init_custom_layout(self):
        # initialize data structure
        self.samples = []
        self.labels = []

        # call method to save data upon exiting
        self.Bind(wx.EVT_CLOSE, self._on_exit)
```

We also have to load several classifiers to make the preprocessing and (later on) the real-time classification work. For convenience, default file names are provided:

```
    def init_algorithm(
        self,
        save_training_file='datasets/faces_training.pkl',
        load_preprocessed_data='datasets/faces_preprocessed.pkl',
        load_mlp='params/mlp.xml',
        face_casc='params/haarcascade_frontalface_default.xml',
        left_eye_casc='params/haarcascade_lefteye_2splits.xml',
        right_eye_casc='params/haarcascade_righteye_2splits.xml'):
```

Here, save_training_file indicates the name of a pickle file in which to store all training samples after data acquisition is complete:

```
self.dataFile = save_training_file
```

The three cascades are passed to the FaceDetector class as explained in the previous section:

```
self.faces = FaceDetector(face_casc, left_eye_casc,
    right_eye_casc)
```

As their names suggest, the remaining two arguments (load_preprocessed_data and load_mlp) are concerned with a real-time classification of the detected faces by using the pre-trained MLP classifier:

```
# load preprocessed dataset to access labels and PCA
# params
if path.isfile(load_preprocessed_data):
    (_, y_train), (_, y_test), self.pca_V, self.pca_m =
        homebrew.load_from_file(load_preprocessed_data)
    self.all_labels = np.unique(np.hstack((y_train,
        y_test)))

    # load pre-trained multi-layer perceptron
    if path.isfile(load_mlp):
        self.MLP = MultiLayerPerceptron(
            np.array([self.pca_V.shape[1],
            len(self.all_labels)]),
            self.all_labels)
        self.MLP.load(load_mlp)
```

If any of the parts required for the testing mode are missing, we print a warning and disable the testing mode altogether:

```
    else:
        print "Warning: Testing is disabled"
        print "Could not find pre-trained MLP file ",
            load_mlp
        self.testing.Disable()
else:
    print "Warning: Testing is disabled"
    print "Could not find preprocessed data file ",
        loadPreprocessedData
self.testing.Disable()
```

The GUI layout

Creation of the layout is again deferred to a method called `_create_custom_layout`. We keep the layout as simple as possible: We create a panel for the acquired video frame, and draw a row of buttons below it.

The idea is to then click one of the six radio buttons to indicate which facial expression you are trying to record, then place your head within the bounding box, and click the `Take Snapshot` button.

Below the current camera frame, we place two radio buttons to select either the training or the testing mode, and tell the GUI that the two are mutually exclusive by specifying `style=wx.RB_GROUP`:

```
def _create_custom_layout(self):
    # create horizontal layout with train/test buttons
    pnl1 = wx.Panel(self, -1)
    self.training = wx.RadioButton(pnl1, -1, 'Train', (10, 10),
        style=wx.RB_GROUP)
    self.testing = wx.RadioButton(pnl1, -1, 'Test')
    hbox1 = wx.BoxSizer(wx.HORIZONTAL)
    hbox1.Add(self.training, 1)
    hbox1.Add(self.testing, 1)
    pnl1.SetSizer(hbox1)
```

Also, we want the event of a button click to bind to the `self._on_training` and `self._on_testing` methods, respectively:

```
self.Bind(wx.EVT_RADIOBUTTON, self._on_training,
    self.training)
self.Bind(wx.EVT_RADIOBUTTON, self._on_testing, self.testing)
```

The second row should contain similar arrangements for the six facial expression buttons:

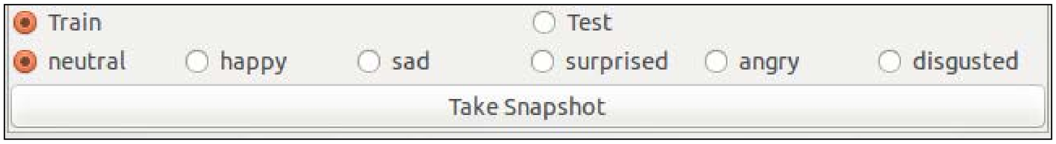
```
# create a horizontal layout with all buttons
pnl2 = wx.Panel(self, -1 )
self.neutral = wx.RadioButton(pnl2, -1, 'neutral',
    (10, 10), style=wx.RB_GROUP)
self.happy = wx.RadioButton(pnl2, -1, 'happy')
self.sad = wx.RadioButton(pnl2, -1, 'sad')
self.surprised = wx.RadioButton(pnl2, -1, 'surprised')
self.angry = wx.RadioButton(pnl2, -1, 'angry')
self.disgusted = wx.RadioButton(pnl2, -1, 'disgusted')
hbox2 = wx.BoxSizer(wx.HORIZONTAL)
hbox2.Add(self.neutral, 1)
hbox2.Add(self.happy, 1)
```

```
hbox2.Add(self.sad, 1)
hbox2.Add(self.surprised, 1)
hbox2.Add(self.angry, 1)
hbox2.Add(self.disgusted, 1)
pnl2.SetSizer(hbox2)
```

The **Take Snapshot** button is placed below the radio buttons and will bind to the `_on_snapshot` method:

```
pnl3 = wx.Panel(self, -1)
self.snapshot = wx.Button(pnl3, -1, 'Take Snapshot')
self.Bind(wx.EVT_BUTTON, self.OnSnapshot, self.snapshot)
hbox3 = wx.BoxSizer(wx.HORIZONTAL)
hbox3.Add(self.snapshot, 1)
pnl3.SetSizer(hbox3)
```

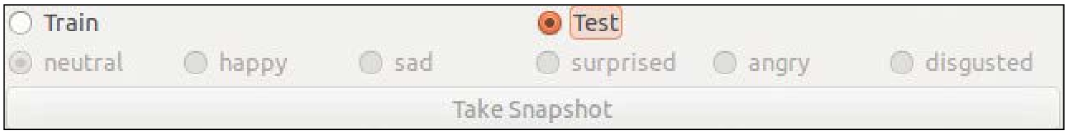
This will look like the following:



To make these changes take effect, the created panels need to be added to the list of existing panels:

```
# display the button layout beneath the video stream
self.panels_vertical.Add (pnl1, flag=wx.EXPAND | wx.TOP, border=1)
self.panels_vertical.Add(pnl2, flag=wx.EXPAND | wx.BOTTOM,
    border=1)
self.panels_vertical.Add(pnl3, flag=wx.EXPAND | wx.BOTTOM,
    border=1)
```

The rest of the visualization pipeline is handled by the `BaseLayout` class. Now, whenever the user clicks the `self.testing` button, we no longer want to record training samples, but instead switch to the testing mode. In the testing mode, none of the training-related buttons should be enabled, as shown in the following image:



This can be achieved with the following method that disables all the relevant buttons:

```
    def _on_testing(self, evt):
        """Whenever testing mode is selected, disable all
            training-related buttons"""
        self.neutral.Disable()
        self.happy.Disable()
        self.sad.Disable()
        self.surprised.Disable()
        self.angry.Disable()
        self.disgusted.Disable()
        self.snapshot.Disable()
```

Analogously, when we switch back to the training mode, the buttons should be enabled again:

```
    def _on_training(self, evt):
        """Whenever training mode is selected, enable all
            training-related buttons"""
        self.neutral.Enable()
        self.happy.Enable()
        self.sad.Enable()
        self.surprised.Enable()
        self.angry.Enable()
        self.disgusted.Enable()
        self.snapshot.Enable()
```

Processing the current frame

The rest of the visualization pipeline is handled by the `BaseLayout` class. We only need to make sure to provide the `_process_frame` method. This method begins by detecting faces in a downscaled grayscale version of the current frame, as explained in the previous section:

```
    def _process_frame(self, frame):
        success, frame, self.head = self.faces.detect(frame)
```

If a face is found, `success` is `True`, and the method has access to an annotated version of the current frame (`frame`) and the extracted head region (`self.head`). Note that we store the extracted head region for further reference, so that we can access it in `_on_snapshot`.

We will return to this method when we talk about the testing mode, but for now, this is all we need to know. Don't forget to pass the processed frame:

```
        return frame
```

Adding a training sample to the training set

When the Take Snapshot button is clicked upon, the _on_snapshot method is called. This method detects the emotional expression that we are trying to record by checking the value of all radio buttons, and assigns a class label accordingly:

```
def _on_snapshot(self, evt):
    if self.neutral.GetValue():
        label = 'neutral'
    elif self.happy.GetValue():
        label = 'happy'
    elif self.sad.GetValue():
        label = 'sad'
    elif self.surprised.GetValue():
        label = 'surprised'
    elif self.angry.GetValue():
        label = 'angry'
    elif self.disgusted.GetValue():
        label = 'disgusted'
```

We next need to look at the detected facial region of the current frame (stored in self.head by _process_frame), and align it with all the other collected frames. That is, we want all the faces to be upright and the eyes to be aligned. Otherwise, if we do not align the data samples, we run the risk of having the classifier compare eyes to noses. Because this computation can be costly, we do not apply it on every frame, but instead only upon taking a snapshot. The alignment takes place in the following method:

```
    if self.head is None:
        print "No face detected"
    else:
        success, head = self.faces.align_head(self.head)
```

If this method returns True for success, indicating that the sample was successfully aligned with all other samples, we add the sample to our dataset:

```
if success:
    print "Added sample to training set"
    self.samples.append(head.flatten())
    self.labels.append(label)
else:
    print "Could not align head (eye detection failed?)"
```

All that is left to do now is to make sure that we save the training set upon exiting.

Dumping the complete training set to a file

Upon exiting the app (for example, by clicking the **Close** button of the window), an event EVT_CLOSE is triggered, which binds to the _on_exit method. This method simply dumps the collected samples and the corresponding class labels to file:

```
def _on_exit(self, evt):
    """Called whenever window is closed"""
    # if we have collected some samples, dump them to file
    if len(self.samples) > 0:
```

However, we want to make sure that we do not accidentally overwrite previously stored training sets. If the provided filename already exists, we append a suffix and save the data to the new filename instead:

```
        # make sure we don't overwrite an existing file
        if path.isfile(self.data_file):
            filename, fileext = path.splitext(self.data_file)
            offset = 0
            while True: # a do while loop
                file = filename + "-" + str(offset) + fileext
                if path.isfile(file):
                    offset += 1
                else:
                    break
            self.data_file = file
```

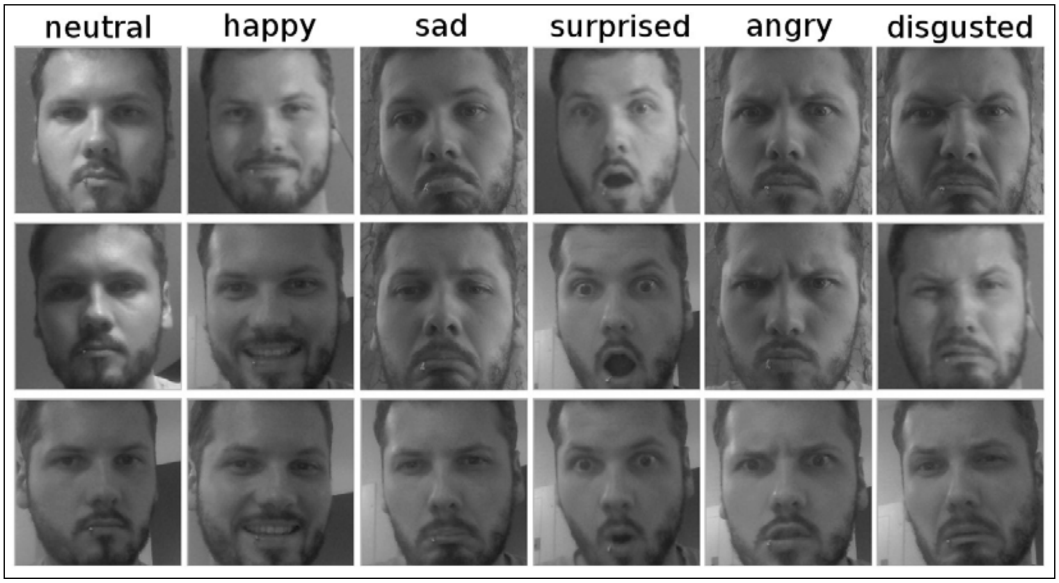
Once we have created an unused filename, we dump the data to file by making use of the pickle module:

```
        f = open(self.dataFile, 'wb')
        pickle.dump(self.samples, f)
        pickle.dump(self.labels, f)
        f.close()
```

Upon exiting, we inform the user that a file was created and make sure that all data structures are correctly deallocated:

```
        print "Saved", len(self.samples), "samples to", self.data_file
        self.Destroy()
```

Here are some examples from the assembled training set I:



Feature extraction

We have previously made the point that, finding the features that best describe the data is often an essential part of the entire learning task. We have also looked at common preprocessing methods such as **mean subtraction** and **normalization**. Here, we will look at an additional method that has a long tradition in face recognition: **principal component analysis** (**PCA**).

Preprocessing the dataset

Analogous to *Chapter 6*, *Learning to Recognize Traffic Signs*, we write a new dataset parser in `dataset/homebrew.py` that will parse our self-assembled training set. We define a `load_data` function that will parse the dataset, perform feature extraction, split the data into training and testing sets, and return the results:

```
import cv2
import numpy as np

import csv
from matplotlib import cm
from matplotlib import pyplot as plt

from os import path
```

```
import cPickle as pickle


def load_data(load_from_file, test_split=0.2, num_components=50,
    save_to_file=None, plot_samples=False, seed=113):
    """load dataset from pickle """
```

Here, `load_from_file` specifies the path to the data file that we created in the previous section. We can also specify another file called `save_to_file`, which will contain the dataset after feature extraction. This will be helpful later on when we perform real-time classification.

The first step is thus to try and open `load_from_file`. If the file exists, we use the pickle module to load the `samples` and `labels` data structures; else, we throw an error:

```
# prepare lists for samples and labels
X = []
labels = []
if not path.isfile(load_from_file):
    print "Could not find file", load_from_file
    return (X, labels), (X, labels), None, None
else:
    print "Loading data from", load_from_file
    f = open(load_from_file, 'rb')
    samples = pickle.load(f)
    labels = pickle.load(f)
    print "Loaded", len(samples), "training samples"
```

If the file was successfully loaded, we perform PCA on all samples. The `num_components` variable specifies the number of principal components that we want to consider. The function also returns a list of basis vectors (`V`) and a mean value (`m`) for every sample in the set:

```
# perform feature extraction
# returns preprocessed samples, PCA basis vectors & mean
X, V, m = extract_features(samples,
    num_components=num_components)
```

As pointed out earlier, it is imperative to keep the samples that we use to train our classifier separate from the samples that we use to test it. For this, we shuffle the data and split it into two separate sets, such that the training set contains a fraction (`1 - test_split`) of all samples, and the rest of the samples belong to the test set:

```
# shuffle dataset
np.random.seed(seed)
np.random.shuffle(X)
np.random.seed(seed)
np.random.shuffle(labels)

# split data according to test_split
X_train = X[:int(len(X)*(1-test_split))]
y_train = labels[:int(len(X)*(1-test_split))]

X_test = X[int(len(X)*(1-test_split)):]
y_test = labels[int(len(X)*(1-test_split)):]
```

If specified, we want to save the preprocessed data to file:

```
if save_to_file is not None:
    # dump all relevant data structures to file
    f = open(save_to_file, 'wb')
    pickle.dump(X_train, f)
    pickle.dump(y_train, f)
    pickle.dump(X_test, f)
    pickle.dump(y_test, f)
    pickle.dump(V, f)
    pickle.dump(m, f)
    f.close()
    print "Save preprocessed data to", save_to_file
```

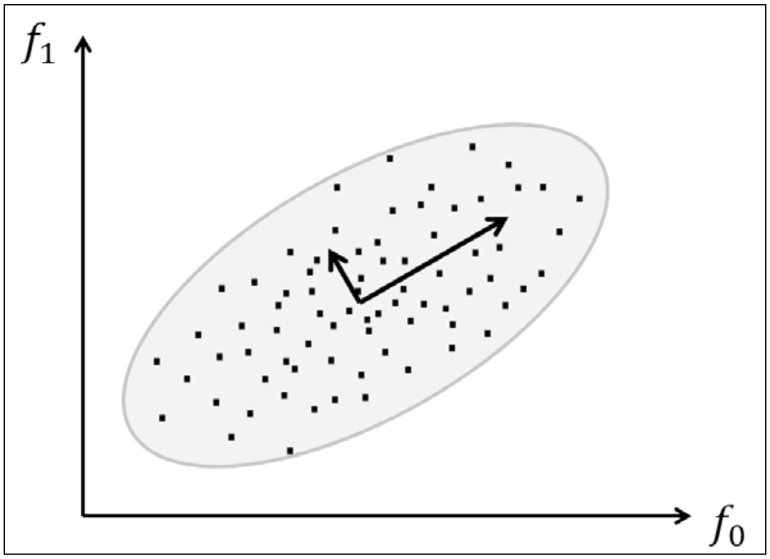
Finally, we can return the extracted data:

```
return (X_train, y_train), (X_test, y_test), V, m
```

Principal component analysis

PCA is a dimensionality reduction technique that is helpful whenever we are dealing with high-dimensional data. In a sense, you can think of an image as a point in a high-dimensional space. If we flatten a 2D image of height m and width n (by concatenating either all rows or all columns), we get a (feature) vector of length m × n. The value of the i-th element in this vector is the grayscale value of the i-th pixel in the image. To describe every possible 2D grayscale image with these exact dimensions, we will need an m × n dimensional vector space that contains 256 raised to the power of m × n vectors. Wow! An interesting question that comes to mind when considering these numbers is as follows: could there be a smaller, more compact vector space (using less than m × n features) that describes all these images equally well? After all, we have previously realized that grayscale values are not the most informative measures of content.

This is where PCA comes in. Consider a dataset from which we extracted exactly two features. These features could be the grayscale values of pixels at some *x* and *y* positions, but they could also be more complex than that. If we plot the dataset along these two feature axes, the data might lie within some multivariate Gaussian, as shown in the following image:



What PCA does is *rotate* all data points until the data lie aligned with the two *axes* (the two inset vectors) that explain most of the *spread* of the data. PCA considers these two axes to be the most informative, because if you walk along them, you can see most of the data points separated. In more technical terms, PCA aims to transform the data to a new coordinate system by means of an orthogonal linear transformation. The new coordinate system is chosen such that if you project the data onto these new axes, the first coordinate (called the **first principal component**) observes the greatest variance. In the preceding image, the small vectors drawn correspond to the eigenvectors of the covariance matrix, shifted so that their tails come to lie at the mean of the distribution.

Fortunately, someone else has already figured out how to do all this in Python. In OpenCV, performing PCA is as simple as calling `cv2.PCACompute`. Embedded in our feature extraction method, the option reads as follows:

```
def extract_feature(X, V=None, m=None, num_components=50):
```

Here, the function can be used to either perform PCA from scratch or use a previously calculated set of basis vectors (V) and mean (m), which is helpful during testing when we want to perform real-time classification. The number of principal components to consider is specified via num_components. If we do not specify any of the optional arguments, PCA is performed from scratch on all the data samples in X:

```
        if V is None or m is None:
            # need to perform PCA from scratch
            if num_components is None:
                num_components = 50

            # cols are pixels, rows are frames
            Xarr = np.squeeze(np.array(X).astype(np.float32))

            # perform PCA, returns mean and basis vectors
            m, V = cv2.PCACompute(Xarr)
```

The beauty of PCA is that the first principal component by definition explains most of the variance of the data. In other words, the first principal component is the most informative of the data. This means that we do not need to keep all of the components to get a good representation of the data! We can limit ourselves to the num_components most informative ones:

```
        # use only the first num_components principal components
        V = V[:num_components]
```

Finally, a compressed representation of the data is achieved by projecting the zero-centered original data onto the new coordinate system:

```
        for i in xrange(len(X)):
            X[i] = np.dot(V, X[i] - m[0, i])

        return X, V, m
```

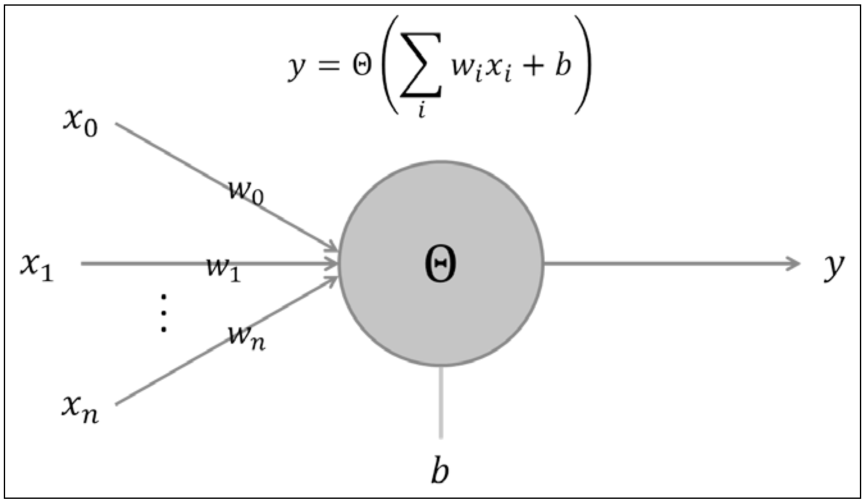
Multi-layer perceptrons

Multi-layer perceptrons (MLPs) have been around for a while. MLPs are artificial neural networks (ANNs) used to convert a set of input data into a set of output data. At the heart of an MLP is a **perceptron**, which resembles (yet overly simplifies) a biological neuron. By combining a large number of perceptrons in multiple layers, the MLP is able to make non-linear decisions about its input data. Furthermore, MLPs can be trained with **backpropagation**, which makes them very interesting for supervised learning.

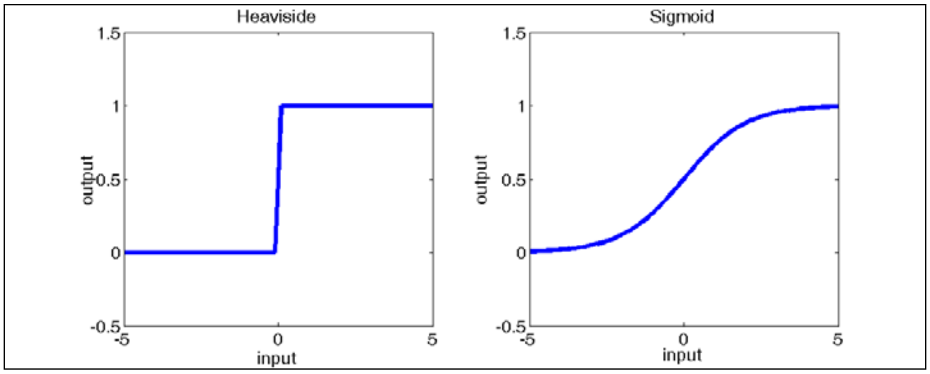
The perceptron

A perceptron is a binary classifier that was invented in the 1950s by Frank Rosenblatt. A perceptron calculates a weighted sum of its inputs, and if this sum exceeds a threshold, it outputs a `1`; else, it outputs a `0`. In some sense, a perceptron is integrating evidence that its afferents signal the presence (or absence) of some object instance, and if this evidence is strong enough, the perceptron will be active (or silent). This is loosely connected to what researchers believe biological neurons are doing (or can be used to do) in the brain, hence, the term artificial neural network.

A sketch of a perceptron is depicted in the following figure:



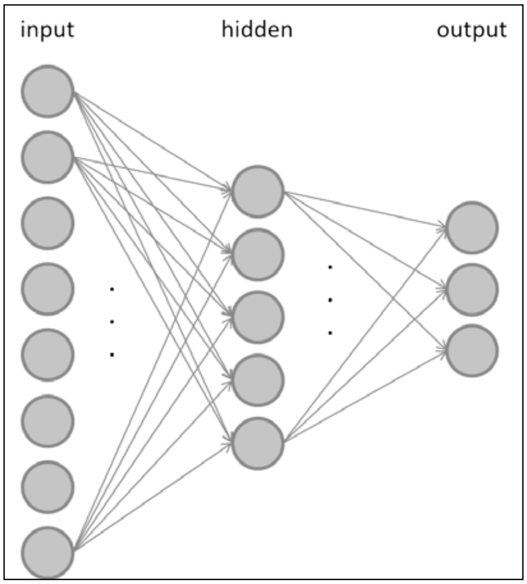
Here, a perceptron computes a weighted (`w_i`) sum of all its inputs (`x_i`), combined with a bias term (`b`). This input is fed into a nonlinear activation function (`θ`) that determines the output of the perceptron (`y`). In the original algorithm, the activation function was the Heaviside step function. In modern implementations of ANNs, the activation function can be anything ranging from sigmoid to hyperbolic tangent functions. The Heaviside function and the sigmoid function are plotted in the following image:



Depending on the activation function, these networks might be able to perform either classification or regression. Traditionally, one only talks of MLPs when nodes use the Heaviside step function.

Deep architectures

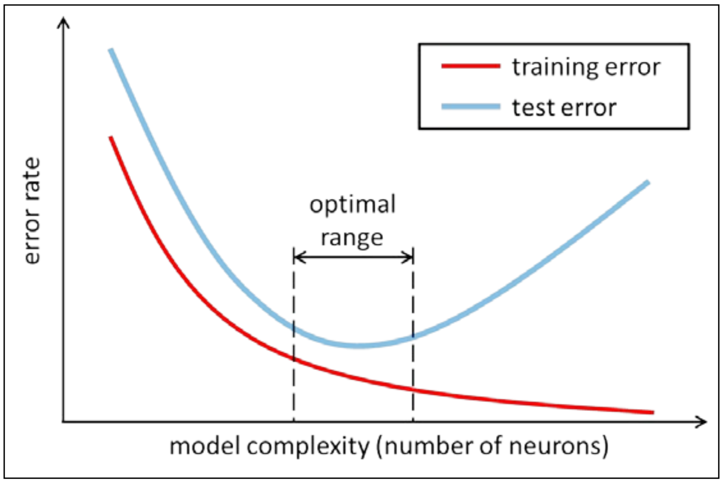
Once you have the perceptron figured out, it would make sense to combine multiple perceptrons to form a larger network. Multi-layer perceptrons usually consist of at least three layers, where the first layer has a node (or neuron) for every input feature of the dataset, and the last layer has a node for every class label. The layer in between is called the **hidden layer**. An example of this **feed-forward neural network** is shown in the following figure:



In a feed-forward neural network, some or all of the nodes in the input layer are connected to all the nodes in the hidden layer, and some or all of the nodes in the hidden layer are connected to some or all of the nodes in the output layer. You would usually choose the number of nodes in the input layer to be equal to the number of features in the dataset, so that each node represents one feature. Analogously, the number of nodes in the output layer is usually equal to the number of classes in the data, so that when an input sample of class `c` is presented, the c-th node in the output layer is active and all others are silent.

It is also possible to have multiple hidden layers of course. Often, it is not clear beforehand what the optimal size of the network should be.

Typically, you will see the error rate on the training set decrease when you add more neurons to the network, as is depicted in the following figure (thinner, red curve):



This is because the expressive power or complexity (also referred to as the **Vapnik–Chervonenkis** or **VC dimension**) of the model increases with the increasing size of the neural network. However, the same cannot be said for the error rate on the test set (thicker, blue curve)! Instead, what you will find is that with increasing model complexity, the test error goes through a minimum, and adding more neurons to the network will not improve the generalization performance any more. Therefore, you would want to steer the size of the neural network to what is labeled the optimal range in the preceding figure, which is where the network achieves the best generalization performance.

You can think of it this way. A model of weak complexity (on the far left of the plot) is probably too small to really understand the dataset that it is trying to learn, thus yielding large error rates on both the training and the test sets. This is commonly referred to as **underfitting**. On the other hand, a model on the far right of the plot is probably so complex that it begins to memorize the specifics of each sample in the training data, without paying attention to the general attributes that make a sample stand apart from the others. Therefore, the model will fail when it has to predict data that it is has never seen before, effectively yielding a large error rate on the test set. This is commonly referred to as **overfitting**.

Instead, what you want is to develop a model that neither underfits nor overfits. Often this can only be achieved by trial-and-error; that is, by considering the network size as a hyperparameter that needs to be tweaked and tuned depending on the exact task to be performed.

An MLP learns by adjusting its weights so that when an input sample of class `c` is presented, the `c`-th node in the output layer is active and all the others are silent. MLPs are trained by means of **backpropagation**, which is an algorithm to calculate the partial derivative of a **loss function** with respect to any synaptic weight or neuron bias in the network. These partial derivatives can then be used to update the weights and biases in the network in order to reduce the overall loss step-by-step.

A loss function can be obtained by presenting training samples to the network and by observing the network's output. By observing which output nodes are active and which are silent, we can calculate the relative error between what the output layer is doing and what it should be doing (that is, the loss function). We then make corrections to all the weights in the network so that the error decreases over time. It turns out that the error in the hidden layer depends on the output layer, and the error in the input layer depends on the error in both the hidden layer and the output layer. Thus, in a sense, the error (back)propagates through the network. In OpenCV, backpropagation is used by specifying `cv3.ANN_MLP_TRAIN_PARAMS_BACKPROP` in the training parameters.

> Gradient descent comes in two common flavors: In **stochastic gradient descent**, we update the weights after each presentation of a training example, whereas in **batch learning**, we present training examples in batches and update the weights only after each batch is presented. In both scenarios, we have to make sure that we adjust the weights only slightly per sample (by adjusting the **learning rate**) so that the network slowly converges to a stable solution over time.

An MLP for facial expression recognition

Analogous to *Chapter 6, Learning to Recognize Traffic Signs*, we will develop a multi-layer perceptron class that is modeled after the classifier base class. The base classifier contains a method for training, where a model is fitted to the training data, and for testing, where the trained model is evaluated by applying it to the test data:

```
from abc import ABCMeta, abstractmethod

class Classifier:
    """Abstract base class for all classifiers"""
    __metaclass__ = ABCMeta

    @abstractmethod
    def fit(self, X_train, y_train):
        pass

    @abstractmethod
    def evaluate(self, X_test, y_test, visualize=False):
        pass
```

Here, `X_train` and `X_test` correspond to the training and the test data, respectively, where each row represents a sample and each column is a feature value of this sample. The training and test labels are passed as the `y_train` and `y_test` vectors, respectively.

We thus define a new class, `MultiLayerPerceptron`, which derives from the classifier base class:

```
class MultiLayerPerceptron(Classifier):
```

The constructor of this class accepts an array called `layer_sizes` that specifies the number of neurons in each layer of the network and an array called `class_labels` that spells out all available class labels. The first layer will contain a neuron for each feature in the input, whereas the last layer will contain a neuron per output class:

```
    def __init__(self, layer_sizes, class_labels, params=None):
        self.num_features = layer_sizes[0]
        self.num_classes = layer_sizes[-1]
        self.class_labels = class_labels
        self.params = params or dict()
```

The constructor initializes the multi-layer perceptron by means of an OpenCV module called `cv2.ANN_MLP`:

```
        self.model = cv2.ANN_MLP()
        self.model.create(layer_sizes)
```

For the sake of convenience to the user, the MLP class allows operations on string labels as enumerated via `class_labels` (for example, *neutral*, *happy*, and *sad*). Under the hood, the class will convert back and forth from strings to integers and from integers to strings, so that `cv2.ANN_MLP` will only be exposed to integers. These transformations are handled by the following two private methods:

```
def _labels_str_to_num(self, labels):
    """ convert string labels to their corresponding ints """
    return np.array([int(np.where(self.class_labels == l)[0])
        for l in labels])

def _labels_num_to_str(self, labels):
    """Convert integer labels to their corresponding string
        names """
    return self.class_labels[labels]
```

Load and save methods provide simple wrappers for the underlying `cv2.ANN_MLP` class:

```
def load(self, file):
    """ load a pre-trained MLP from file """
    self.model.load(file)

def save(self, file):
    """ save a trained MLP to file """
    self.model.save(file)
```

Training the MLP

Following the requirements defined by the `Classifier` base class, we need to perform training in a `fit` method:

```
def fit(self, X_train, y_train, params=None):
    """ fit model to data """
    if params is None:
        params = self.params
```

Here, `params` is an optional dictionary that contains a number of options relevant to training, such as the termination criteria (`term_crit`) and the learning algorithm (`train_method`) to be used during training. For example, to use backpropagation and terminate training either after 300 iterations or when the loss reaches values smaller than 0.01, specify `params` as follows:

```
params = dict(
    term_crit = (cv2.TERM_CRITERIA_COUNT, 300, 0.01),
    train_method = cv2.ANN_MLP_TRAIN_PARAMS_BACKPROP)
```

Because the `train` method of the `cv2.ANN_MLP` module does not allow integer-valued class labels, we need to first convert `y_train` into "one-hot" code, consisting only of zeros and ones, which can then be fed to the `train` method:

```
y_train = self._labels_str_to_num(y_train)
y_train = self._one_hot(y_train).reshape(-1,
    self.num_classes)
    self.model.train(X_train, y_train, None, params=params)
```

The one-hot code is taken care of in `_one_hot`:

```
def _one_hot(self, y_train):
    """Convert a list of labels into one-hot code """
```

Each class label `c` in `y_train` needs to be converted into a `self.num_classes`-long vector of zeros and ones, where all entries are zeros except the c-th, which is a one. We prepare this operation by allocating a vector of zeros:

```
    num_samples = len(y_train)
    new_responses = np.zeros(num_samples * self.num_classes,
        np.float32)
```

Then, we identify the indices of the vector that correspond to all the c-th class labels:

```
    resp_idx = np.int32(y_train +
        np.arange(num_samples)  self.num_classes)
```

The vector elements at these indices then need to be set to one:

```
    new_responses[resp_idx] = 1
    return new_responses
```

Testing the MLP

Following the requirements defined by the `Classifier` base class, we need to perform training in an `evaluate` method:

```
def evaluate(self, X_test, y_test, visualize=False):
    """ evaluate model performance """
```

Analogous to the previous chapter, we will evaluate the performance of our classifier in terms of accuracy, precision, and recall. To reuse our previous code, we again need to come up with a 2D voting matrix, where each row stands for a data sample in the testing set and the c-th column contains the number of votes for the `c`-th class.

In the world of perceptrons, the voting matrix actually has a straightforward interpretation: The higher the activity of the `c`-th neuron in the output layer, the stronger the vote for the `c`-th class. So, all we need to do is to read out the activity of the output layer and plug it into our accuracy method:

```
    ret, Y_vote = self.model.predict(X_test)
    y_test = self._labels_str_to_num(y_test)
    accuracy = self._accuracy(y_test, Y_vote)
    precision = self._precision(y_test, Y_vote)
    recall = self._recall(y_test, Y_vote)

    return (accuracy, precision, recall)
```

In addition, we expose the predict method to the user, so that it is possible to predict the label of even a single data sample. This will be helpful when we perform real-time classification, where we do not want to iterate over all test samples, but instead only consider the current frame. This method simply predicts the label of an arbitrary number of samples and returns the class label as a human-readable string:

```
def predict(self, X_test):
    """ predict the labels of test data """
    ret, Y_vote = self.model.predict(X_test)

    # find the most active cell in the output layer
    y_hat = np.argmax(Y_vote, 1)

    # return string labels
    return self._labels_num_to_str(y_hat)
```

Running the script

The MLP classifier can be trained and tested by using the `train_test_mlp.py` script. The script first parses the homebrew dataset and extracts all class labels:

```
import cv2
import numpy as np

from datasets import homebrew
from classifiers import MultiLayerPerceptron

def main():
    # load training data
    # training data can be recorded using chapter7.py in training
    # mode
    (X_train, y_train),(X_test, y_test) =
        homebrew.load_data("datasets/faces_training.pkl",
        num_components=50, test_split=0.2,
        save_to_file="datasets/faces_preprocessed.pkl",
        seed=42)
    if len(X_train) == 0 or len(X_test) == 0:
        print "Empty data"
    raise SystemExit

    # convert to numpy
    X_train = np.squeeze(np.array(X_train)).astype(np.float32)
    y_train = np.array(y_train)
    X_test = np.squeeze(np.array(X_test)).astype(np.float32)
    y_test = np.array(y_test)

    # find all class labels
    labels = np.unique(np.hstack((y_train, y_test)))
```

We also make sure to provide some valid termination criteria as described above:

```
    params = dict( term_crit = (cv2.TERM_CRITERIA_COUNT, 300,
        0.01), train_method=cv2.ANN_MLP_TRAIN_PARAMS_BACKPROP,
        bp_dw_scale=0.001, bp_moment_scale=0.9 )
```

Often, the optimal size of the neural network is not known a priori, but instead, a hyperparameter needs to be tuned. In the end, we want the network that gives us the best generalization performance (that is, the network with the best accuracy measure on the test set). Since we do not know the answer, we will run a number of different-sized MLPs in a loop and store the best in a file called `saveFile`:

```
    save_file = 'params/mlp.xml'
    num_features = len(X_train[0])
    num_classes = len(labels)

    # find best MLP configuration
    print "1-hidden layer networks"
    best_acc = 0.0 # keep track of best accuracy
    for l1 in xrange(10):
        # gradually increase the hidden-layer size
        layer_sizes = np.int32([num_features,
            (l1 + 1) * num_features / 5,
            num_classes])
    MLP = MultiLayerPerceptron(layer_sizes, labels)
    print layer_sizes
```

The MLP is trained on X_train and tested on X_test:

```
        MLP.fit(X_train, y_train, params=params)
        (acc, _, _) = MLP.evaluate(X_train, y_train)
        print " - train acc = ", acc
        (acc, _, _) = MLP.evaluate(X_test, y_test)
        print " - test acc = ", acc
```

Finally, the best MLP is saved to file:

```
        if acc > best_acc:
            # save best MLP configuration to file
            MLP.save(saveFile)
        best_acc = acc
```

The saved params/mlp.xml file that contains the network configuration and learned weights can then be loaded into the main GUI application (chapter7.py) by passing loadMLP='params/mlp.xml' to the init_algorithm method of the FaceLayout class. The default arguments throughout this chapter will make sure that everything works straight out of the box.

Putting it all together

In order to run our app, we will need to execute the main function routine (in chapter7.py) that loads the pre-trained cascade classifier and the pre-trained multi-layer perceptron, and applies them to each frame of the webcam live stream.

However, this time, instead of collecting more training samples, we will select the radio button that says **Test**. This will trigger an EVT_RADIOBUTTON event, which binds to FaceLayout._on_testing, disabling all training-related buttons in the GUI and switching the app to the testing mode. In this mode, the pre-trained MLP classifier is applied to every frame of the live stream, trying to predict the current facial expression.

As promised earlier, we now return to FaceLayout._process_frame:

```
    def _process_frame(self, frame):
        """ detects face, predicts face label in testing mode """
```

Unchanged from what we discussed earlier, the method begins by detecting faces in a downscaled grayscale version of the current frame:

```
        success, frame, self.head = self.faces.detect(frame)
```

However, in the testing mode, there is more to the function:

```
# in testing mode: predict label of facial expression
if success and self.testing.GetValue():
```

In order to apply our pre-trained MLP classifier to the current frame, we need to apply the same preprocessing to the current frame as we did with the entire training set. After aligning the head region, we apply PCA by using the pre-loaded basis vectors (`self.pca_V`) and mean values (`self.pca_m`):

```
# if face found: preprocess (align)
success, head = self.faces.align_head(self.head)
if success:
    # extract features using PCA (loaded from file)
    X, _, _ = homebrew.extract_features(
        [head.flatten()], self.pca_V, self.pca_m)
```

Then, we are ready to predict the class label of the current frame:

```
# predict label with pre-trained MLP
label = self.MLP.predict(np.array(X))[0]
```

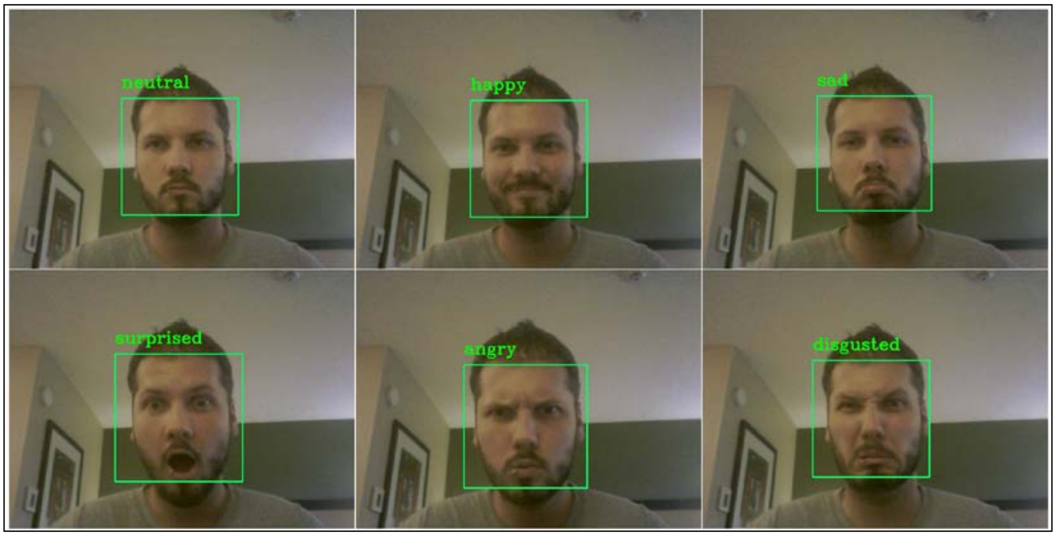
Since the `predict` method already returns a string label, all that is left to do is to display it above the bounding box in the current frame:

```
# draw label above bounding box
cv2.putText(frame, str(label), (x,y-20),
    cv2.FONT_HERSHEY_COMPLEX, 1, (0,255,0), 2)
break # need only look at first, largest face
```

Finally, we are done!

```
return frame
```

The end result looks like the following:



Although the classifier has only been trained on (roughly) 100 training samples, it reliably detects my various facial expressions in every frame of the live stream, no matter how distorted my face seemed to be at the moment. This is a good indication that the neural network that was learned neither underfits nor overfits the data, since it is capable of predicting the correct class labels even for new data samples.

Summary

The final chapter of this book has really rounded up our experience and made us combine a variety of our skills to arrive at an end-to-end app that consists of both object detection and object recognition. We became familiar with a range of pre-trained cascade classifiers that OpenCV has to offer, collected our very own training dataset, learned about multi-layer perceptrons, and trained them to recognize emotional expressions in faces. Well, at least my face.

The classifier was undoubtedly able to benefit from the fact that I was the only subject in the dataset, but with all the knowledge and experience that you have gathered with this book, it is now time to overcome these limitations! You can start small and train the classifier on images of you indoors and outdoors, at night and day, during summer and winter. Or, maybe, you are anxious to tackle a real-world dataset and be part of Kaggle's Facial Expression Recognition challenge (see `https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge`).

If you are into machine learning, you might already know that there is a variety of accessible libraries out there, such as pylearn (`https://github.com/lisa-lab/pylearn2`), scikit-learn (`http://scikit-learn.org`), and pycaffe (`http://caffe.berkeleyvision.org`). Deep learning enthusiasts might want to look into Theano (`http://deeplearning.net/software/theano`) or Torch (`http://torch.ch`). Finally, if you find yourself stuck with all these algorithms and no datasets to apply them to, make sure to stop by the UC Irvine Machine Learning Repository (`http://archive.ics.uci.edu/ml`).

Congratulations! You are now an OpenCV expert.

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *OpenCV Computer Vision with Python, Joseph Howse*
- *OpenCV with Python By Example, Prateek Joshi*
- *OpenCV with Python Blueprints, Michael Beyeler*