Making Computers See in Python

Early Release

Practical Computer Vision

with SimpleCV

Kurt Demaagd,
Anthony Oliver,
Nathan Oostendorp
& Katherine Scott

O'REILLY®

Practical Computer Vision with SimpleCV

Nathan Oostendorp, Anthony Oliver, and Katherine Scott

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Practical Computer Vision with SimpleCV
by Nathan Oostendorp, Anthony Oliver, and Katherine Scott

Revision History for the :
2012-05-01 Early release revision 1
See *http://oreilly.com/catalog/errata.csp?isbn=9781449320362* for release details.

ISBN: 978-1-449-32036-2
1335970018

Table of Contents

Preface

SimpleCV is a framework for use with Python. Python is a relatively easy language to learn. For individuals who have no programming experience, Python is a popular language for introductory computer and web programming classes. There are a wealth of books on programming in Python and even more free resources available online. For individuals with prior programming experience but with no background in Python, it is an easy language to pick up.

As the name SimpleCV implies, the framework was designed to be simple. Nonetheless, a few new vocabulary items come up frequently when designing vision systems using SimpleCV. Some of the key background concepts are described below:

Computer Vision
: The analyzing and processing of images. These concepts can be applied to a wide array of applications, such as medical imaging, security, autonomous vehicles, etc. It often tries to duplicate human vision by using computers and cameras.

Machine Vision
: The application of computer vision concepts, typically in an industrial setting. These applications are used for quality control, process control, or robotics. These are also generally considered the "solved" problems. However, there is no simple dividing line between machine vision and computer vision. For example, some advanced machine vision applications, such as 3D scanning on a production line, may still be referred to as computer vision.

Tuple
: A list with a pair of numbers. In Python, it is written enclosed in parentheses. It is often used when describing `(x, y)` coordinates, the width and height of an object, or other cases where there is a logical pairing of numbers. It has a slightly more technical definition in mathematics, but this definition covers its use in this book.

NumPy Array or Matrix
: NumPy is a popular Python library used in many scientific computing applications, known for its fast and efficient algorithms. Since an image can also be thought of as an array of pixels, many bits of processing use NumPy's array data type. When an array has two or more dimensions, it is sometimes called a Matrix. Although

intimate knowledge of NumPy is not needed to understand this book, it is useful from time to time.

Blob
: Blobs are contiguous regions of similar pixels. For example, in a picture detecting a black cat, the cat will be a **blob** of contiguous black pixels. They are so important in computer vision that they warrant their own chapter. They also pop up from time to time throughout the entire book. Although covered in detail later, it is good to at least know the basic concept now.

JPEG, PNG, GIF or other image formats
: Images are stored in different ways, and SimpleCV can work with most major image formats. This book primarily uses PNG's, which are technically similar to GIF's. Both formats use non-lossy compression, which essentially means the image quality is not changed in the process of compressing it. This creates a smaller image file without reducing the quality of the image. Some examples also use JPEG's. This is a form of lossy compress, which results in even smaller files, but at the cost of some loss of image quality.

PyGame
: PyGame appears from time to time throughout the book. Like NumPy, PyGame is a handy library for Python. It handles a lot of window and screen management work. This will be covered in greater detail in the Drawing chapter. However, it will also pop up throughout the book when discussing drawing on the screen.

Conventions Used in This Book

The following typographical conventions are used in this book:

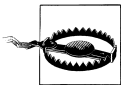
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2011 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North

Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/<catalog page>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Introduction

This chapter provides an introduction to computer vision in general and the SimpleCV framework in particular. The primary goal is to understand the possibilities and considerations to keep in mind when creating a vision system. In the process, this chapter will cover:

- The importance of computer vision
- An introduction to the SimpleCV framework
- Hard problems for computer vision
- Problems that are relatively easy for computer vision
- An introduction to vision systems
- The typical components of a vision system

Why Learn Computer Vision

As cameras are becoming standard PC hardware and a required feature of mobile devices, computer vision is moving from a niche tool to an increasingly common tool for a diverse range of applications. Some of these applications probably spring readily to mind, such as facial recognition programs or gaming interfaces like the Kinect. Computer vision is also being used in things like automotive safety systems, where your car detects when you start to drift from your lane, or when you're getting drowsy. It is used in point-and-shoot cameras to help detect faces or other central objects to focus on. The tools are used for high tech special effects or basic effects, such as the virtual yellow first-and-ten line in football games or motion blurs on a hockey puck. It has applications in industrial automation, biometrics, medicine, and even planetary exploration. It's also used in some more surprising fields, such as with food and agriculture, where it is used to inspect and grade fruits and vegetables. It's a diverse field, with more and more interesting applications popping up every day.

At its core, computer vision is built upon the fields of mathematics, physics, biology, engineering, and of course, computer science. There are many fields related to com-

puter vision, such as machine learning, signal processing, robotics, and artificial intelligence. Yet even though it is a field built on advanced concepts, more and more tools are making it accessible to everyone from hobbyists to vision engineers to academic researchers.

It is an exciting time in this field, and there are an endless number of possibilities for what you might be able to do with it. One of the things that makes it exciting is that these days, the hardware requirements are inexpensive enough to allow more casual developers entry into the field, opening the door to many new applications and innovations.

What is the SimpleCV framework?

SimpleCV, which stands for Simple Computer Vision, is an easy-to-use Python framework that bundles together open source computer vision libraries and algorithms for solving problems. Its goal is to make it easier for programmers to develop computer vision systems, streamlining and simplifying many of the most common tasks. You do not have to have a background in computer vision to use the SimpleCV framework or a computer science degree from a top-name engineering school. Even if you don't know Python, it is a pretty easy language to learn. Most of the code in this book will be relatively easy to pick up, regardless of your programming background. What you do need is an interest in computer vision or helping to make computers "see". In case you don't know much about computer vision, we'll give you some background on the subject in this chapter. Then in the next chapter, we'll jump into creating vision systems with the SimpleCV framework.

What is Computer Vision?

Vision is a classic example of a problem that humans handle well, but with which machines struggle. As you go through your day, you use your eyes to take in a huge amount of visual information and your brain then processes it all without any conscious thought. Computer vision is the science of creating a similar capability in computers and, if possible, to improve upon it. The more technical definition, though, would be that computer vision is the science of having computers acquire, process and analyze digital images. You will also see the term machine vision used in conjunction with computer vision. Machine vision is frequently defined as the application of computer vision to industrial tasks.

One of the challenges for computers is that humans have a surprising amount of "hardware" for collecting and deciphering visual data. You probably haven't spent a lot of time thinking about the challenges involved in processing what you see. For instance, consider what is involved in reading this book. As you look at it, you first need to understand what data represents the book and what is just background data that you

can ignore. One of the ways, you do this through depth perception, and your body has several reinforcing systems to help with this:

- Eye muscles that can determine distance based on how much effort is exerted to bend the eye's lens.
- Stereo vision that detects slightly different pictures of the same scene, as seen by each eye. Similar pictures mean the object is far away, while different pictures mean the object is close.
- The slight motion of the body and head, which creates the parallax effect. This is the effect where the position of an object appears to move when viewed from different positions. Since this difference is greater when the object is closer to you and smaller when the object is further away, the parallax effect helps you judge the distance to an object.

Once you have focused on the book, you then have to process the marks on the page into something useful. Your brain's advanced pattern recognition system has been taught which of the black marks on this page represent letters, and how they group together to form words. While certain elements of reading are the product of education and training, such as learning the alphabet, you also manage to map words written in `several` *different* **fonts** back to that original alphabet (Wingding fonts not withstanding).

Take the above challenges of reading, and then multiply them with the constant stream of information through time, with each moment possibly including various changes in the data. Hold the book at a slightly different angle (or tip the e-reader a little bit). Hold it closer to you or further away. Turn a page. Is it still the same book? These are all challenges that are unconsciously solved by the brain. In fact, one of the first tests given to babies is whether their eyes can track objects. A newborn baby already has a basic ability to track but computers struggle with the same task.

That said, there are quite a few things that computers can do better than humans:

- Computers can look at the same thing for hours and hours. They don't get tired and they can't get bored.
- Computers can quantify image data in a way that humans cannot. For example, computers can measure dimensions of objects very precisely, and look for angles and distances between features in an image.
- Computers can see places in a picture where the pixels next to each other have very different colors. These places are called "edges", and computers can tell you exactly where edges are, and quantitatively measure how strong they are.
- Computers can see places where adjacent pixels share a similar color, and give you measurements on shapes and sizes. These are often called "connected components", or more colloquially, "blobs".

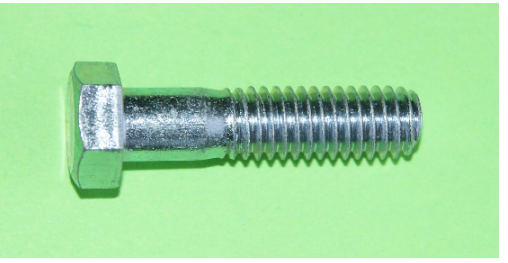


Figure 1-1. Hard: What is this? Easy: How many threads per inch?

- Computers can compare two images and see very precisely the difference between those two images. Even if something is moving imperceptibly over hours—a computer can use image differences to measure how much it changes.

Part of the practice of computer vision is finding places where the computer's eye can be used in a way that would be difficult or impractical for humans. One of the goals of this book is show how computers can be used to see in these cases.

Easy vs. Hard Problems

Computer vision problems, in many ways, mirror the challenges of using computers in general: computers are good at computation, but weak at reasoning. Computer vision will be effective with tasks such as measuring objects, identifying differences between objects, finding high contrast regions, etc. These tasks all work best under conditions of stable lighting. Computers struggle when working with irregular objects, classifying and reasoning about an object, tracking objects in motion, etc. All of these problems are compounded by poor lighting conditions or moving elements.

For example, consider the image shown in Figure 1-1. What is it a picture of? A human can easily identify it as a bolt. For a computer to make that determination, it will require a large database with pictures of bolts, pictures of objects that are not bolts, and computation time to train the algorithm. Even with that information, the computer may regularly fail, especially when dealing with similar objects, such as distinguishing between bolts and screws.

However, a computer does very well at tasks such as counting the number of threads per inch. Humans can count the threads as well, of course, but it will be a slow and error prone, not to mention headache inducing, process. In contrast, it is relatively easy to write an algorithm that detects each thread. Then it is a simple matter of computing the number of those threads in an inch. This is an excellent example of a problem prone to error when performed by a human, but easily handled by a computer.

Some other classic examples of easy vs. hard problems include:

Table 1-1. Easy and hard problems for computer vision

Easy	Hard
How wide is this plate? Is it dirty?	Look at a picture of a random kitchen and find all the dirty plates.
Did something change between these two images?	Track an object or person moving through a crowded room of other people
Measure the diameter of a wheel. Check if it is bent.	Identify arbitrary parts on pictures of bicycles.
What color is this leaf?	What kind of leaf is this?

Furthermore, all of the challenges of computer vision are amplified in certain environments. One of the largest challenges is the lighting. Low light often results in a lot of noise in the image, requiring various tricks to try to clean up the image. In addition, some types of objects are difficult to analyze, such as shiny objects that may be reflecting other objects in their surroundings.

Note that hard problems do not mean impossible problems. The later chapters of this book look at some of the more advanced features of computer vision systems. These chapters will discuss techniques such as finding, identifying, and tracking objects.

What is a Vision System?

A vision system is something that evaluates data from an image source (typically a camera), extracts data about those images, and does something with the results. For example, consider a parking space monitor. This system watches a parking space, and detects parking violations in which unauthorized cars attempt to park in the spot. If the owner's car is in the space or if the space is empty, then there is no violation. If someone else is parked in the space, then there is a problem. Figure 1-2 outlines the overall logic flow for such a system.

Although conceptually simple, the problem presents many complexities. Lighting conditions affect color detection and the ability to distinguish the car from the background. The car may be parked in a slightly different place each time, hindering the detection of the car versus an empty spot. The car might be dirty, making it hard to distinguish the owner's car versus a violator's. The parking spot could be covered in snow, making it difficult to tell whether the parking spot is empty.

To help address the above complexities, a typical vision system has two general steps. The first step is to filter the input to narrow the range of information to be processed. The second step is to extract and process the key features of the image(s).

Filtering Input

The first step in the machine vision system is to filter the information available. In the parking space example, the camera's viewing area most likely overlaps with other

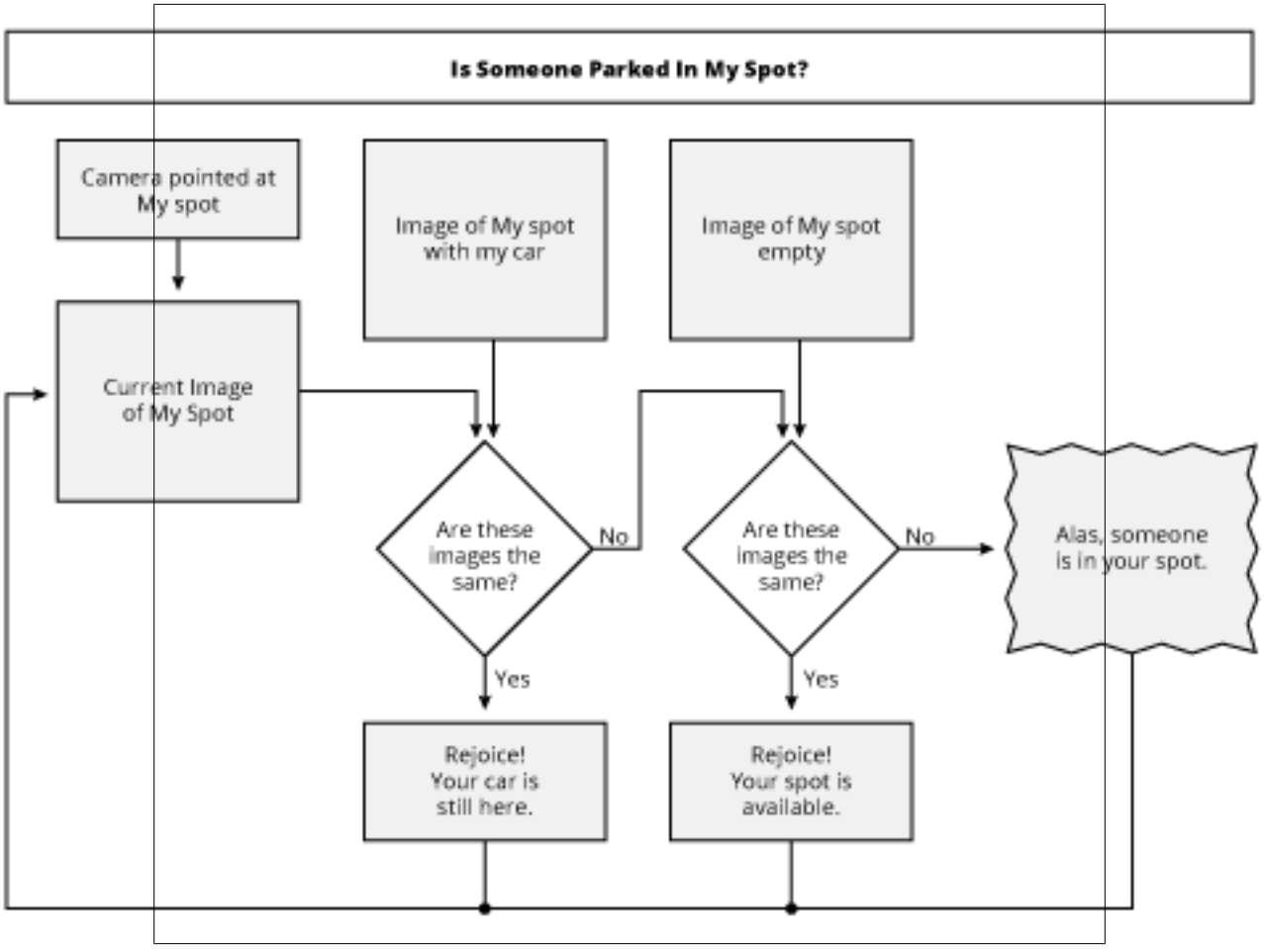


Figure 1-2. Diagram of parking spot vision system

parking spaces. A car in an adjacent parking space or a car in a space across the street is fine. Yet if they appear in the image, the car detection algorithm could inadvertently pick up these cars, creating a false positive. The obvious approach would be to crop the image to cover only the relevant parking space, though this book will also cover other approaches to filtering.

In addition to the challenge of having too much information, images must also be filtered because they have too little information. Humans work with a rich set of information, potentially detecting a car using multiple sensors of input to collect data and compare it against some sort of pre-defined car pattern. Machine vision systems have limited input, typically from a 2D camera, and therefore must use inexact and potentially error-prone proxies. This amplifies the potential for error. To minimize errors, only the necessary information should be used. For example, A brown spot in

the parking space could represent a car, but it could also represent a paper bag blowing through the parking lot. Filtering out small objects could resolve this, improving the performance of the system.

Filtering plays another important role. As camera quality improves and image sizes grow, machine vision systems become more computationally taxing. If a system needs to operate in real time or near real time, the computing requirements of examining a large image may require unacceptable processing time. However, filtering the information controls the amount of data and decreases how much processing that must be done.

Extracting Features and Information

Once the image is filtered by removing some of the noise and narrowing the field to just the region of interest, the next step is to extract the relevant features. It is up to the programmer to translate those features into more applicable information. In the car example, it is not possible to tell the system to look for a car. Instead, the algorithm looks for car-like features, such as a rectangular license plate, or rough parameters on size, shape, color, etc. Then the program assumes that something matching those features must be a car.

Some commonly used features covered in this book include:

- Color information: looking for changes in color to detect objects.
- Blob extraction: detecting adjacent, similarly colored pixels.
- Edges and corners: examining changes in brightness to identify the borders of objects.
- Pattern recognition and template matching: adding basic intelligence by matching features with the features of known objects.

In certain domains, a vision system can go a step further. For example, if it is known that the image contains a barcode or text, such as a license plate, the image could be passed to the appropriate barcode reader or Optical Character Recognition (OCR) algorithm. A robust solution might be to read the car's license plate number, and then that number could be compared against a database of authorized cars.

CHAPTER 2

Getting to Know the SimpleCV framework

The goal of the SimpleCV framework is to make common computer vision tasks easy. This chapter introduces some of the basics, including how to access a variety of different camera devices, how to use those cameras to capture and perform basic image tasks, and how to display the resulting images on the screen. Other major topics include:

- Installing the SimpleCV framework
- Working with the shell
- Accessing standard webcams
- Controlling the display window
- Creating basic applications

Installation

The SimpleCV framework has compiled installers for Windows, Mac, and Ubuntu Linux, but it can also be used on any system that Python and OpenCV can be built on. The installation procedure varies for each operating system. Since SimpleCV is an open source framework, it can also be installed from source. For the most up to date details on installation, go to *http://www.simplecv.org/doc/installation.html*. This section provides a brief overview of each installation method.

Regardless of the target operating system, the starting point for all installations is *http://www.simplecv.org*. The home page includes links for downloading the installation files for all major platforms. The installation links are displayed as icons for the Windows, Mac, and Ubuntu systems.

Clicking the download button on *http://www.simplecv.org*, goes to a page that automatically downloads the installer for your current operating system. To download the installer for a different operating system, go to *http://sourceforge.net/projects/simplecv/files/* and select the download for the desired operating system. The Windows filename extension is .exe, and the Mac's is .pkg. With Ubuntu Linux, the extension is .deb.

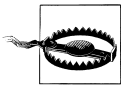
Windows

By far the easiest way to install the SimpleCV framework on Windows is by using the Windows Superpack. Clicking the download link from *http://www.simplecv.org* will download the Superpack from *http://sourceforge.net*. Simply download the Superpack and follow the instructions. In addition to the basic installation, it will also check for and download any missing dependencies. The following are the required dependencies:

- Python 2.7
- Python Setup Tools
- NumPy
- SciPy
- Easy_install
- OpenCV

Once the installation is complete, it will create a `SimpleCV` program group under the Start menu. This includes a link to start the SimpleCV interactive Python shell and a link to access the program documentation. The majority of the examples in this book can be run using the SimpleCV Shell. More details regarding the shell are covered in the SimpleCV Shell section of this chapter.

Mac



There have been substantial changes made to the available software and development tools between different versions of Mac OS X. As a result, the installation instructions for Macs are likely to change in response to the ever evolving underlying configuration of the operating system. Please see *http://www.simplecv.org* for the latest installation instructions.

Before beginning a Mac install, it is strongly recommended that XCode be installed from Apple. On some versions of Mac OS X, this will resolve dependency and installation errors. To download, see *https://developer.apple.com/xcode*. You will also need to install the Command Line Tools from within Xcode (Xcode→Preferences→Downloads→Components).

The Mac installation follows a template similar to the Windows approach. From the SimpleCV home page, click the download button. It will go to SourceForge.net and begin downloading the installation Superpack. This package will handle the installation of the SimpleCV framework and its major dependencies. The list of dependencies is the same as it is for Windows:

- Python 2.7
- Python Setup Tools
- NumPy
- SciPy
- Easy_install
- OpenCV

Because the SimpleCV framework integrates with Python, the Superpack installs files in places other than the Applications directory. It will also install binary dependencies in /usr/local, and Python libraries in /Library/Python2.7. If you already have a Mac-Linux system in place, such as Homebrew, Ports, or Fink, you may want to follow the instructions to install from source.

The easiest way to work with the examples below is from the Python shell. Once the SimpleCV framework is installed, you can either click on the `SimpleCV.command` icon in the Applications folder, or start iPython from a terminal window. To start a terminal window, go to the Applications folder, find the Utilities folder, and then click on Terminal to launch it. This will bring up a command prompt. At the command prompt, type `python -m SimpleCV.__init__` to bring up the SimpleCV interactive Python shell. Most of the examples in this book can be run from the SimpleCV Shell.

Linux

While the following instructions are for Ubuntu Linux, they should also work for other Debian-based Linux distributions. Installing SimpleCV for Ubuntu is done through a `.deb` package. From the SimpleCV home page, click the download button, which is the rightmost of the three logos. This will download the package and handle the installation of all the required dependencies.

Note, however, that even recent distributions of Ubuntu may have an out of date version of OpenCV, one of the major dependencies for SimpleCV. If the installation throws errors with OpenCV, in a Terminal window enter:

```
$ sudo add-apt-repository ppa:gijzelaar/opencv2.3
$ sudo apt-get update
```

Once SimpleCV is installed, start the SimpleCV interactive Python shell by opening a command prompt and entering `python -m SimpleCV.__init__`. A majority of the examples in this book can be completed from the SimpleCV Shell.

Installation from Source

Some users may want to have the bleeding edge version of SimpleCV installed. SimpleCV is an open source framework, so the latest versions of the source code are freely available. The source code for SimpleCV is available on GitHub. The repository can be cloned from `git@github.com:ingenuitas/SimpleCV.git`.

A full description of using GitHub is beyond the scope of this book. For more information, see `http://www.github.com`.

Once the source code is installed, go to the directory where it was downloaded. Then run the following command at the command prompt:

```
$ python setup.py install
```

Installation from source does not automatically install the required dependencies. As a shortcut for installing dependencies, it may be easiest to first install SimpleCV using the appropriate package described above. This doesn't guarantee that the newest dependencies will be resolved, but it will streamline the process.

Hello World

As is mandated by the muses of technical writing, the first example is a basic Hello World app. This application assumes that the computer has a built-in webcam, a camera attached via USB, or a similar video device attached to it. It will then use that camera to take a picture and display it on the screen.

The SimpleCV framework uses Python. For those who want a little more background on Python, check out Learning Python by Mark Lutz for a good introduction to the language.

```
from SimpleCV import Camera, Display, Image  ❶

# Initialize the camera
cam = Camera()

# Initialize the display
display = Display()  ❷

# Snap a picture using the camera
img = cam.getImage()  ❸

# Show the picture on the screen
img.save(display)  ❹
```

Either copy the code above into the SimpleCV Shell or save the above code as `helloWorld.py` using a plain text editor and run the program. For those who have not worked with Python before, these are the basic steps:

- Open a terminal window or the command prompt
- Go to the directory where `helloWorld.py` is saved
- Type `python helloWorld.py` and press the Enter key.

Once the program has started, look at the webcam, smile, wave, and say, "Hello World." The program will take a picture and display it on the screen.

This example program uses three of the most common libraries provided by the SimpleCV framework: `Camera`, `Display`, and `Image`. These will be covered in greater detail later in this book; however, a brief introduction to the program follows:

❶ The first line of code imports the libraries used in this program. Technically, `Image` does not need to be specifically listed since Python already knows that `Camera` will return `Image` objects. It is simply included here for clarity.

❷ The next two lines are constructors. The first line initializes a `Camera` object to be able to capture the images, and the second line creates a `Display` object for displaying a window on the screen.

❸ This line uses the `getImage()` function from the Camera class, which snaps a picture using the camera

❹ The final line then "saves" the image to the display, which makes the image appear on the screen.

Hello World purists may object that this program does not actually display the text, "Hello World." Instead, the example relies on the user to say the words while the picture is taken. This situation is easily resolved. `Image` objects have a `drawText()` function that can be used to display text on the image. The following example demonstrates how to create a more traditional Hello World program.

```
from SimpleCV import Camera, Display, Image
import time

# Initialize the camera
cam = Camera()
# Initialize the display
display = Display()

# Snap a picture using the camera
img = cam.getImage()
# Show some text
img.drawText("Hello World!")
# Show the picture on the screen
img.save(display)
# Wait five seconds so the window doesn't close right away
time.sleep(5)
```

The program will appear almost identical to the original Hello World example, except it now draws the text "Hello World" in the middle of the screen. You might also notice that there are a few new lines: `import time` at the top of the program, and `time.sleep(5)` at the bottom. The `import time` line imports the time module from Python's standard library. The `sleep()` function from the time module is then used at the end of the program to delay closing the window for 5 seconds.

At this point, you have created a very simple computer vision program, using the Camera, Display, and Image classes. Of course, this is only the tip of the iceberg.

The SimpleCV Shell

Before going into more depth regarding the features of the SimpleCV framework, this is a good point to pause and discuss one of the best tools to use when working with this book. Most of this book is based on using small snippets of code to do cool things. Yet the traditional way to code is to create a new file, enter the new code, save the file, and then run the file to see if there are errors. If there are errors (and there are always errors), you have to open the file, fix the errors, re-save the file, and re-run the file—and keep doing this loop until all of the errors are fixed. This is an extremely cumbersome process when working with little code snippets. A much faster way to work through the examples is with the SimpleCV Shell, or simply "the Shell" as it is called in this book. The Shell is built using IPython, an interactive shell for Python development. This section introduces the basics of working with the shell. Additional tutorials and more advanced tricks are available on IPython's web site, *http://www.ipython.org*.

Most of the example code in this book is written so it could be executed as a stand alone script. However, this code can be still be run in the SimpleCV Shell. There are a few examples that should only be run from the shell, because they won't run properly as a standard script. These examples are mainly to illustrate how the shell works. In these cases, the example code will include the characters >>> at the beginning of each line to indicate that it is expected to be run in the shell. When you type this example code into the shell, you don't want to include this initial >>> characters. For instance, in the shell, you can access an interactive tutorial, launching it with the `tutorial` command demonstrated below:

```
>>> tutorial
```

When you actually type this into the shell, only want to enter the word 'tutorial' and hit the enter key. Do not type the >>> string. The >>> merely indicates that it is a command to be entered in the shell.

Basics of the Shell

Some readers may have had prior experience working with a shell on a Linux system, MS-DOS or software like Matlab, SPSS, R, or Quake 3 Arena. Conceptually, the SimpleCV Shell has many similarities. Whereas the operating system shells take text-based

commands and pass them to the OS for execution, the SimpleCV Shell takes Python commands and passes them to the Python interpreter for execution. The SimpleCV Shell is based on iPython, and automatically loads all of the SimpleCV framework libraries. As a result, it is a great tool for tinkering with the SimpleCV framework, exploring the API, looking up documentation, and generally testing snippets of code before deployment in a full application.

All of the functions available while programming are also available when working with the shell. However, writing code in the shell still has a slightly different "feel" compared with traditional approaches. The shell will interpret commands after they are typed. This is usually on a line-by-line basis, though certain blocks, such as loops, will only be executed once the full block is entered. This places certain practical limitations on the complexity of the code, but in turn, makes an ideal environment for testing code snippets.

Starting the SimpleCV Shell varies depending on the operating system. In Windows, there is a SimpleCV link that is installed on the Desktop, or accessible from the SimpleCV folder under the Start menu. From a Mac or Linux machine, open a command prompt and type `simplecv` to start the shell.

If the shell doesn't start with the SimpleCV command, it is also possible to start it manually. Either type: `python -m SimpleCV.__init__`, or start the Python shell by typing `python` at the command prompt and enter the following:

```
>>> from SimpleCV import Shell
>>> Shell.main()
```

To quit the shell, simply type `exit()` and press Return or Enter.

Press the keyboard's up arrow to scroll back through the history of previously entered commands.

Similar to the popular shells found on Linux and Mac systems, the SimpleCV Shell supports tab completion. When typing the name of an object or function, press `tab` and the shell will attempt to complete the name of the function. For example, if attempting to reference the `getImage()` function of a Camera object, first type `getI` and press `tab`. The shell will complete the rest of the name of the function. In cases where multiple function names begin with the same letters, the shell will not be able to fully complete the function name. With a Camera object, `get` could be referring to `getImage()` or to `getDepth()`. In such cases, it is necessary to enter additional letters to eliminate the name ambiguity.

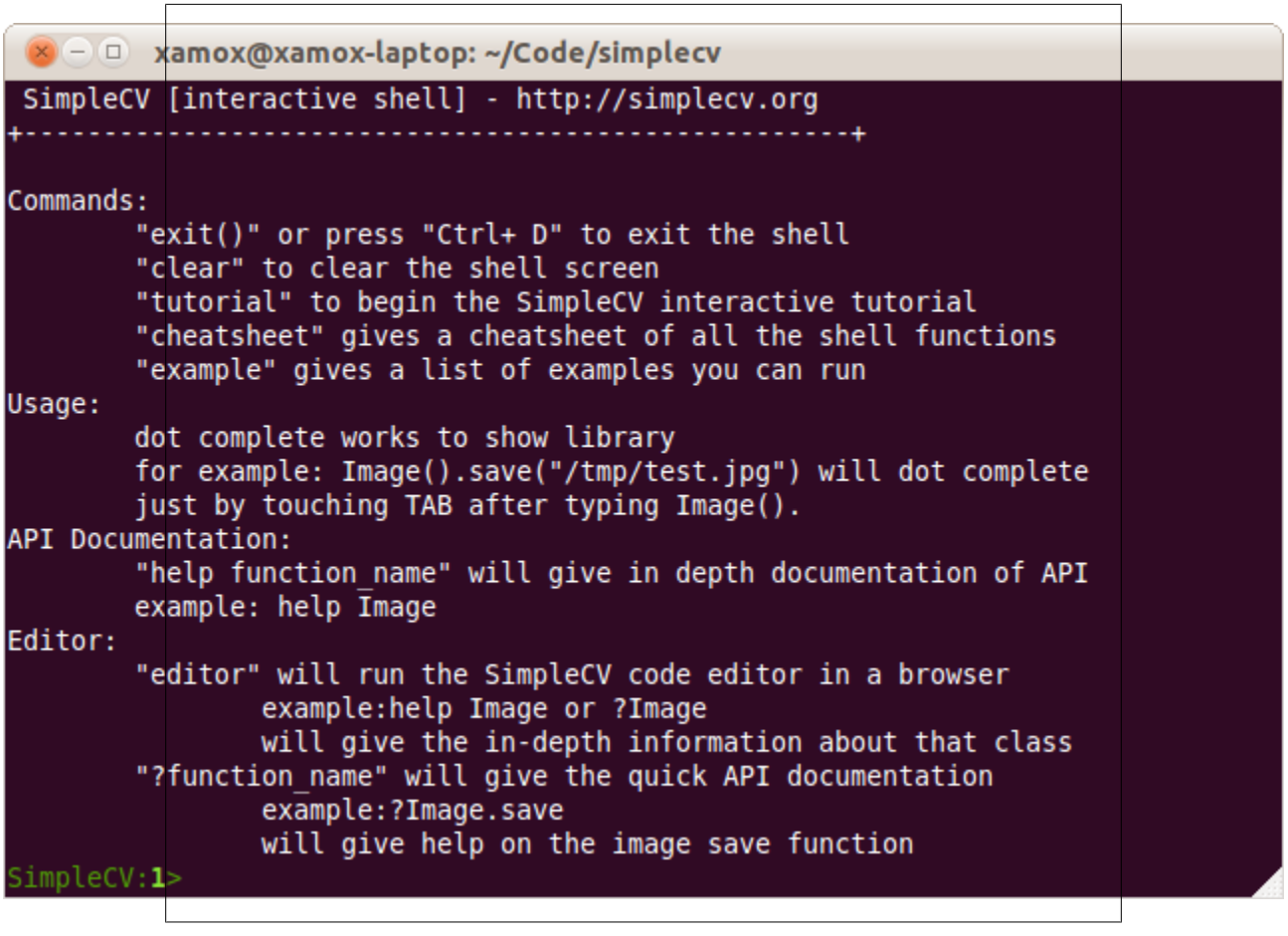
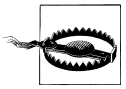


Figure 2-1. A screenshot of the SimpleCV Shell

If tab completion does not work, make sure that `pyreadline` is installed.

One of the most convenient features of the shell is the built-in help system. The help system can display the latest documentation for many objects. The documentation is organized by objects, but each object's functions are also documented. For example, the documentation for Camera includes general information about the Camera class and its methods and inherited methods. To get help using the shell, simply type `help` *`object`*. For example, to get help on the Image object, you would type:

```
>>> help Image
```



Python is case sensitive: typing `help image` is not the same as `help Image`. This follows a standard convention in programming called CamelCase and is used throughout the SimpleCV framework—notice it is not simplecv—and Python in general. Modules and classes will have an uppercase first letter, variables and methods will not.

A slight variant on the above example is using the `?` to get help on an object. In the code below, adding a question mark to the end of `img` will show help information about the image object. Since the SimpleCV Shell imports all of the SimpleCV libraries, you don't need to include the line "from SimpleCV import Image" to create the Image object.

```
>>> img = Image('logo')
# The next two lines do the same thing
>>> img?
>>> ?img
```

The shell is also a convenient tool for frequently changing and testing blocks of code. In the shell, commands entered are immediately executed. Compare this to the traditional development cycle of writing and saving a block of code, compiling that code—or in the case of Python, waiting for the interpreter to start up—and then actually running the code to examine the results. This traditional process substantially slows any interactive designing and testing. For example, computer vision programming often involves tweaking various filters and thresholds to correctly identify the regions or objects of interest. Rather than going through many write-compile-execute cycles, it is substantially easier to simply test different configurations using the shell.

Consider the challenge of finding blobs. Blobs will be covered in greater depth later, but for the purposes of this example, assume that a blob is simply a region of lighter colored pixels surrounded by darker colored pixels. Letters written in white on a dark background are good examples of blobs. "Lighter" versus "darker" pixels are distinguished based on a threshold value. To ensure that the blobs are correctly identified, this threshold must be tweaked, usually through a process of trial and error.

Image objects have a function named `findBlobs()` that will search for the blobs. This function has a threshold option that represents its sensitivity. As an example, to find some blobs in the SimpleCV logo, try the following:

```
>>> img = Image('logo')
>>> blobs = img.findBlobs(255)
>>> print blobs
```

The shell will print `None` because no blobs will be found with the threshold argument set that high. To easily change this, simply tap the up arrow on the keyboard, cycling back to the line of code with `blobs = img.findBlobs(255)`. Now replace the `255` with `100`, and press the enter key. Then hit the up arrow twice again, which will bring back the `print blobs` line. Hit enter again.

```
>>> blobs = img.findBlobs(100)
>>> print blobs
```

This time, the shell will output something like the following:

```
[SimpleCV.Features.Blob.Blob object at (35, 32) with area 385,
 SimpleCV.Features.Blob.Blob object at (32, 32) with area 1865]
```

This shows that with the lower threshold, the blobs were detected. Later chapters will cover fun blob tricks such as coloring them in and displaying them in a user-friendly fashion. For now, the point is to demonstrate how easy it is to rapidly tinker with the code using the SimpleCV Shell.

The Shell and the File System

One feature of the SimpleCV Shell is that it also includes capabilities of the operating system's shell. As a result, it is possible to navigate around the system without having to have everything located in the program directory. For example, if the images used in this book are stored in a folder named `SimpleCV` located in your home directory (`C:\Users\`*`your_username`*`\SimpleCV\Chapter 2` on Windows, `/home/`*`your_username`*`/SimpleCV/Chapter\ 2` on Linux, and `/Users/`*`your_username`*`/SimpleCV/Chapter\ 2` on Mac). There are also other advantages, such as being able to see what types of files are in a directory, etc.

In the following examples, the Mac/Linux directory and file commands are used. When working in Windows, the forward slash notation is also used, even though the Windows command prompt typically works with backslashes. However, some common Linux commands such as `ls` and `pwd` will still work in the Shell on Windows.

The next example will show how to find the image named `ch1-test.png` using the shell, and then load that image. First, locate the correct directory with the following commands:

```
>>> cd
>>> pwd
```

The two commands above should print out the location of your home directory, which is probably `/home/`*`your_username`* if you're working on a Linux system, `/Users/`*`your_username`* on a Mac system, and `C:\\Users\\`*`your_username`* on Windows. The example consists of two steps. First, by entering `cd` with nothing after it, the current directory is changed to the home directory. The `cd` command stands for "change directory", and when given a directory path, it moves you to that location. When no path is entered, it moves you to your home directory. The second command, `pwd`, stands for "print working directory". It prints out the current location, which should be your home directory.

Assuming that the SimpleCV folder is located in your home directory, it is possible to load the camera-example.png image with the following command:

```
>>> img = Image('SimpleCV/Chapter 2/camera-example.png')
```

However, if you're working with a lot of images, it may be more convenient to be in the image directory instead. To go to the directory with the Chapter 2 images, and then output a list of all of the images, type:

```
>>> cd SimpleCV/Chapter 2
>>> ls
```

The result will be a directory listing of all the sample image files. Notice that this works just like using the `cd` and `ls` commands in a traditional Linux shell. Not surprisingly, common command options also work, such as `ls -la` to show the file details as well as all of the hidden files. This is a good way to lookup a correct file name, and prevent annoying "file not found" error messages. Once the directory and file names are correct, loading and displaying an image is done with the following commands:

```
>>> img = Image('camera-example.png')
>>> img.show()
```

Introduction to the Camera

Having addressed the preliminaries, it is time to dive into the fundamentals of vision system development. For most applications, the first task is to get access to the camera. The SimpleCV framework can handle a variety of video sources, and can stream video from multiple devices at the same time. This section will introduce some of the most common ways to access the camera. For greater detail, including how to work with networked cameras and the Kinect, see the Cameras chapter. In addition, Appendix B will review how to select a camera that best suits the needs of your current project.

The simplest setup is a computer with a built-in webcam or an external video camera. These usually fall into a category called a USB Video Class (UVC) device. This is exactly the case described in the Hello World example in the previous section. In the Hello World example, the following line of code initializes the camera:

```
from SimpleCV import Camera

# Initialize the camera
cam = Camera()
```

This approach will work when working with just one camera, using the default camera resolution, without special calibration, etc. Although the Hello World example in the previous section outlined the standard steps for working with cameras, displays, and images, there is a convenient shortcut when the goal is simply to initialize the camera and make sure that it is working, as demonstrated in the following example:

```
from SimpleCV import Camera

# Initialize the camera
cam = Camera()
```

Figure 2-2. Example output of the basic camera example.

```
# Capture and image and display it
cam.getImage().show()
```

This code will have behavior similarly to the Hello World example, though it is a bit less verbose. An example image is shown in Figure 2-2, but obviously actual results will vary. The `show()` function simply pops up the image from the camera on the screen. It is often necessary to store the image in a variable for additional manipulation instead of simply calling `show()` after `getImage()`, but this is a good block of code for a quick test to make sure that the camera is properly initialized and capturing video.

For many projects, this is all that is needed to access and manage the camera. However, the SimpleCV framework can control the camera in many different ways, such as accessing multiple cameras. After all, if one camera is good, then it follows that two cameras are better. To access more than one camera, pass the `camera_id` as an argument to the `Camera()` constructor.

On Linux, the camera ID corresponds to the **/dev/video(*number*)** device number. On Windows, passing any number to the constructor will cause Windows to pop up a window to select which device to map to that ID number. On Macs, finding the ID number is much more complicated, where it is usually easiest to simply guess which camera is 0, which is 1, and adjust by trial and error.

```
from SimpleCV import Camera

# First attached camera
cam0 = Camera(0)

# Second attached camera
```

```
cam1 = Camera(1)

# Show a picture from the first camera
cam0.getImage().show()

# Show a picture from the second camera
cam1.getImage().show()
```

The above sample code works just like the single-camera version, except now it controls two cameras. The obvious question is: how do you determine what the camera ID is? The example conveniently uses `0` and `1`. In many cases, this will work, because 0 and 1 frequently are the camera IDs. If necessary, it is possible to lookup these numbers, which are assigned by the operating system. On Linux, all peripheral devices have a file created for them in the /dev directory. For cameras, the file names start with video, and end with the camera ID. If you have multiple cameras connected to the computer, when you look in the /dev directory, you should see file names such as /dev/video0 and /dev/video1. The number at the end of the `/dev/video(`*`number`*`)` filename is the camera ID that you want to use. With Windows the camera ID number corresponds to the order of the DirectShow devices, which is a bit more complicated to find. So with Windows, passing in any number as the camera ID will result in a pop up window that lets you map that ID number to a particular device. The ID on a Mac is most easily found with trial and error.

Of course, finding the list of camera IDs still doesn't tell you which camera corresponds to which ID number. For a programmatic solution, capture a couple images and label them. For example:

```
from SimpleCV import Camera

# First attached camera
cam0 = Camera(0)

# Second attached camera
cam1 = Camera(1)

# Show a picture from the first camera
img0 = cam0.getImage()
img0.drawText("I am Camera ID 0")
img0.show()

# Show a picture from the first camera
img1 = cam1.getImage()
img1.drawText("I am Camera ID 1")
img1.show()
```

In addition to the basic initialization, the SimpleCV framework can control many other camera properties. An easy example is forcing the resolution. Almost every webcam supports the standard resolutions of `320x240`, and `640x480` (often called "VGA" resolution). Many newer webcams can handle higher resolutions such as `800x600`, `1024x768`, `1200x1024`, or `1600x1200`. Your webcam may list higher resolutions that are "digitally

Figure 2-3. Example of Hello World application, with the "Hello World" text.

scaled" or "with software enhancement," but these don't contain any more information than your camera's native resolution.

Rather than relying on the default resolution of the camera, sometimes it will be helpful to force a resolution to a set of known dimensions. For example, the previous `drawText()` function assumes that the text should be drawn at the center of the screen. The text can be moved to different `(x, y)` coordinates, but an obvious prerequisite for that is knowing the actual image size. One solution is to force the resolution to a known size.

Here is an example of how to move text to the upper left quadrant of an image, starting at the coordinates `(160, 120)` on a `640x480` image:

```
from SimpleCV import Camera

cam = Camera(0, { "width": 640, "height": 480 })

img = cam.getImage()
img.drawText("Hello World", 160, 120)

img.show()
```

The above example is similar to the Hello World program, but now the text is moved. Notice that the camera's constructor is passed a new argument in the form of `{"key": value}`. The `Camera()` function has a properties argument which allows you to set the various properties listed below. Multiple properties are passed in as a comma delimited list, with the entire list enclosed in brackets. Note: the camera ID number is NOT passed inside the brackets, since it is a separate argument.

The available options are part of the computer's UVC system. Most of these are covered in future chapters. In addition, the configuration options are dependent on the camera and the drivers. However, the following options can be configured:

- `width` and `height`
- `brightness`
- `contrast`
- `saturation`
- `hue`
- `gain`
- `exposure`

The `show()` function is used to display the image to the screen. It keeps the image on the screen until the program terminates—or until the Shell closes. Clicking the close button on the window does not close the window. To get more control of the window, you will need to work with the `Display` object, which is covered later in this chapter.

A Note on the USB Video Device Class (UVC)

UVC has emerged as a "device class" which provides a standard way to control video streaming over USB. Most webcams today are now supported by UVC, and don't require additional drivers for basic operation. Not all UVC devices support all functions, so when in doubt, tinker with your camera in a program like **guvcview** or **Skype** to see what works and what doesn't.

A Live Camera Feed

If you would like to see a live video feed from the camera, all you need to do is use the `live()` function. This is as simple as:

```
from SimpleCV import Camera

cam = Camera()
cam.live()
```

The `live()` function also has two other very useful properties outside of displaying the video feed. It also lets you easily find both the coordinates and the color of a pixel on the screen. Why is this information useful? Well let's say you are trying to crop an image down to a particular section. Being able to click on the live feed to find the coordinates for cropping the image is a lot easier than a trial and error method of guessing what the coordinates are, and then tweaking the coordinates based on the results. Or if you want to isolate an object in the image using segmentation, it's helpful to know what the color is for either the object or the area around the object. We will look at these techniques in greater depth in later chapters, but for now, it is useful to know that you can use the feed from the `live()` function to find this type of information.

To get the coordinates or color for a pixel, use the `live()` function as outlined in the example above. When the live feed is displayed, use your mouse to left-click on the image for the area that you are interested in. The coordinates and color for the pixel at

Figure 2-4. Demonstration of the live feed. The small text on the left of the image is displaying the coordinates and the RGB color values for where the left mouse button is clicked.

that location will then be displayed on the screen. The coordinates will be in the (x, y) format, and the color will be displayed as an RGB triplet (r,g,b).

The Display

As much fun as loading and saving images may be, at some point it is also nice to see the images. The SimpleCV framework has two basic approaches to showing images: displaying them directly in a window, or opening a web browser to display the image. The first approach, showing the image in a window, has been previously demonstrated with the `show()` command. Displaying the image in a web browser is similar:

```
from SimpleCV import Image

img = Image("logo")

# This will show the logo image in a web browser
img.show(type="browser")
```

Notice that it uses the same `show()` function, but requires the argument: `type="browser"`. The major difference between the browser and the window is that the window can capture events, such as a signal to close the window. So far, the examples in this chapter have assumed that the window should remain open until the program completes. For larger and more complex programs, however, this might not be the case.

The next example shows how to take more control over when the window is displayed. First, consider the following condensed example, which is similar to the code used earlier in the chapter:

```
from SimpleCV import Display, Image

display = Display()

# Write to the display
Image("logo").save(display)
```

In this case, the user won't be able to close the window by clicking the close button in the corner of the window. The image will continue to be displayed until the program terminates, regardless of how the user interacts with the window. When the program terminates, then it will naturally clean up its windows. To control the closing of a window based on the user interaction with the window, you can use the Display's `isDone()` function.

```
from SimpleCV import Display, Image
import time

display = Display()
Image("logo").save(display)
print "I launched a window"

# This while loop will keep looping until the window is closed
while not display.isDone():
    time.sleep(0.1)

print "You closed the window"
```

Notice how the line "time.sleep(0.1)" is indented after the while statement? That indentation matters. Python groups statements together into a block of code based on the indentation. So it's that indentation that tells Python to execute the "time.sleep(0.1)" statement as the body of the while loop.

The print statement in the example above outputs to the command prompt and not the image. The `drawText` function is used for drawing on the image. The `print` command is used for outputting text to the command prompt.

Event handling does more than simply close windows. For example, it is relatively easy to write a program that draws a circle wherever the user clicks on the image. This is done by using the mouse position and button state information that is provided by the Display object.

While the window is open, the following information about the mouse is available:

- `mouseX` and `mouseY`: the coordinates of the mouse
- `mouseLeft`, `mouseRight`, and `mouseMiddle`: events triggered when the left, right, or middle buttons on the mouse are clicked
- `mouseWheelUp` and `mouseWheelDown`: events triggered then the scroll wheel on the mouse is moved.

The following example shows how to draw on a screen by using the information and events listed above.

To use indented code in the shell, it will help to use the `%cpaste` macro. From the Shell, type `%cpaste`, copy and paste the desired code into the Shell, and then on a newline enter `--` (two minus signs). It will then execute the pasted block of code. This will resolve any indentation errors thrown by the Shell.

```
from SimpleCV import Display, Image, Color

winsize = (640,480)
display = Display(winsize)  ❶

img = Image(winsize)  ❷
img.save(display)

while not display.isDone():
     if display.mouseLeft:  ❸
          img.dl().circle((display.mouseX, display.mouseY), 4,
                    Color.WHITE, filled=True)  ❹
          img.save(display)
          img.save("painting.png")
```

This example uses several new techniques:

❶ This first step is a little different than the previous examples for initializing the window. In this case, the display is given the tuple `(640, 480)` to specifically set the display size of the window. This will create an empty window with those dimensions.

❷ The same `640x480` tuple is used to create a blank image.

❸ This code checks to see if the application has received a message that the left mouse button has been clicked.

❹ If the button is clicked, draw the circle. The image has a drawing layer, which is accessed with the `dl()` function. The drawing layer then provides access to the `circle()` function. The following are the arguments passed to the `circle()` function in order of appearance: the tuple representing the coordinates for the center of the circle (in this case, the coordinates of the mouse), the desired radius for the circle in pixels (4), the color to draw the circle in (white), and a boolean value for the `filled` argument which determines whether or not to fill in the circle or leave the center empty.

The little circles from the drawing act like a paint brush, coloring in a small region of the screen wherever the mouse is clicked. In this way, example code acts like a basic painting application, that can be used to draw exciting pictures like the one in Figure 2-5.

Figure 2-5. Example using the drawing application above.

This example provides a basic overview of working with the display, and introduces the drawing layer. For more complex features regarding drawing layers, please see Chapter 7, Drawing on Images.

Examples

This section shows several straightforward examples of how to capture images, save them to disk, and render them on the display. As we progress, more advanced applications are covered in later chapters. Comments are included in the code to provide guidance about what is happening at each step along the way.

The examples below also use the time module from Python's standard library, which we first introduced in the updated Hello World program. Experienced Python programmers may already be familiar with this module. In these examples, it is used to create timers which cause a program to wait a pre-specified interval of time. For instance, one example below will capture an image every second. The timer is used to control the interval between snapping new photos.

The following examples include the following:

- How to capture images at a fixed interval for time-lapse photography.
- How to create a photo booth application which lets user interact with the screen to capture images.

Although most of the code snippets presented in this chapter are designed for convenient use in the shell, these examples are best run as independent Python scripts.

Time-Lapse Photography

This example operates like a camera with a timer: it takes a picture once per minute. It is a simple application, which does nothing but save the images to disk. A more advanced version, though, could try to detect if there is any movement, or look for changes in the objects. If left running indefinitely, it could quickly consume a great deal of disk space, so a limit is added to only snap ten photos.

Sometime the Shell has problems when code like loops are cut-and-pasted into the Shell. To solve this problem, from the Shell type `cpaste` and press Enter. Then paste the desired code. Finally, type a dash (-) on a line by itself to exit the paste mode.

```
from SimpleCV import Camera, Image
import time

cam = Camera()

# Set the number of frames to capture
numFrames = 10

# Loop until we reach the limit set in numFrames
for x in range(0, numFrames):
    img = cam.getImage()  ❶

    filepath = "image-" + str(x) + ".jpg"  ❷
    img.save(filepath)  ❸
    print "Saved image to: " + filepath

    time.sleep(60)
```

❶ Snap a picture just like in any other application.

❷ Setup a unique filename so that the image is not overwritten every time. To create unique names, the name is prefixed with `image-` followed by its sequence through the loop.

❸ Finally, save the image to the disk, based on the unique file name.

A Photo Booth Application

The next example is a simple photo booth application. After all, nothing says, "I'm an aspiring vision system developer" better than getting close with a friend or loved one and taking silly pictures. This application takes a continuous feed of images. When you

click the left mouse button on an image, it saves it to disk as `photobooth0.jpg`, `photobooth1.jpg`, etc.

```
from SimpleCV import Camera, Display, Color
import time

# Initialize the camera
cam = Camera()
# Initialize the display
display = Display()

# Take an initial picture
img = cam.getImage()  ❶

# Write a message on the image
img.drawText("Left click to save a photo.",
        color=Color().getRandom())  ❷

# Show the image on the display
img.save(display)
time.sleep(5)  ❸

counter = 0
while not display.isDone():
    # Update the display with the latest image
    img = cam.getImage() ❹
    img.save(display)

    if display.mouseLeft:
        # Save image to the current directory
        img.save("photobooth" + str(counter) + ".jpg")  ❺

        img.drawText("Photo saved.", color=Color().getRandom()) ❻
        img.save(display)
        time.sleep(5)
        counter = counter + 1
```

❶ When the application is started, an initial picture is taken.

❷ Instructions are written on the image using the `drawText()` function. Since no coordinates are passed to `drawText()`, the message will be written in the center of the screen. The `Color().getRandom()` function is used to pass a random color to the color argument. This will result in the text being in a different color each time the message is displayed. Then the image is "saved" to the screen. The `time.sleep()` is used to freeze the image and keep the message on the screen for a few seconds

❸ Wait for a few seconds to let the user view their image.

❹ Inside the `while` loop, the image is repeatedly updated. This loop continues until the window is closed.

❺ If the display receives a click from the left mouse button, as indicated by `display.mouseLeft`, then save the image as `photobooth(number).jpg`.

❻ **After** the image is saved to disk, then a message is written to the screen indicating that the image was saved. It is important to write the text after saving the file. Otherwise, the message would be included on the saved image.

CHAPTER 3

Image Sources

This chapter examines various video and image sources in more detail. This is an important prerequisite to image processing. Obviously having a source of images is helpful before processing any images. SimpleCV can capture images from a variety of image sources, ranging from a standard webcam to the Kinect. In particular, this chapter covers:

- A review of working with webcams
- How to use a Kinect to capture depth information for basic 3D processing
- Using an IP camera as your digital video source
- Working with virtual devices to process images from video feeds or pre-captured sets of images
- How to handle either a single image or a set of images

Overview

When it comes to the cameras you can use with a vision system, there are a lot of options. SimpleCV supports most cameras that connect to a computer through a variety of interfaces such as USB, FireWire, or a built in webcam. It can access networked IP cameras that are connected to the network via a wire or a wireless connection. SimpleCV can even work with many video capture boards, which work with a variety of analog video inputs(such as Composite, Component, and S-Video).

Outside of connecting to the camera, there are options such as whether or not to use a monochrome or a color camera. Or whether or not the camera has a CCD or a CMOS image sensor. You can use cameras that record different portions of the light spectrum, such as visible, infrared, or ultra-violet. Then there's always the choice of how much you want to invest, since cameras these days range from the very inexpensive to the very expensive. Still, even with all of these options, you can start out by simply using your webcam, or even just a local image file. However, if you would like more help on

how to select a camera that best suits the type of project that you are working on, please see (to come).

Outside of how you connect to the camera, there are options such as whether or not to use a monochrome or a color camera. Or whether or not you want to use a camera with a CCD or a CMOS image sensor. You can use cameras that record different portions of the light spectrum, such as visible, infrared, or ultra-violet. Then there's always the choice of how much you want to invest, since cameras these days range from the very inexpensive to the very expensive. Still, even with all of these options, you can start out by simply using your webcam, or even just a local image file. However, if you would like more help on how to select a camera that best suits the type of project that you are working on, please see (to come).

In this chapter, we'll revisit how to work with a locally connected camera again, which was first introduced in Chapter 2. Then we will delve into some more advanced topics such as using a Kinect for it's 3D depth information, working with remote cameras on the Internet, and using virtual video devices to do things like accessing streams of data previously saved to disk.

Images, Image Sets & Video

The SimpleCV framework does not need a camera to process images. Instead, it makes it easy to load and save images that were previously captured and saved to disk. This is useful both for working with pre-existing image sources and for saving images captured from a camera so they can be processed at a later time. The following demonstrates the three ways to load image files:

```
from SimpleCV import Image

builtInImg = Image("logo")  ❶

webImg = Image("http://simplecv.s3.amazonaws.com/simplecv_lg.png")  ❷

localImg = Image("image.jpg")  ❸
```

❶ Loading one of the built-in images that are distributed with the SimpleCV framework.

❷ Loading an image directly from the Internet by specifying the URL of the image.

❸ Loading an image available on the hard drive by specifying the filename of the image.

The first approach in the above example loads the SimpleCV logo, which is bundled with the SimpleCV framework. Additional options for bundled images include:

- `simplecv`: The SimpleCV logo
- `logo`: Also the SimpleCV logo
- `logo_inverted`: An inverted version of the logo
- `logo_transparent`: A version of the logo with a transparent background

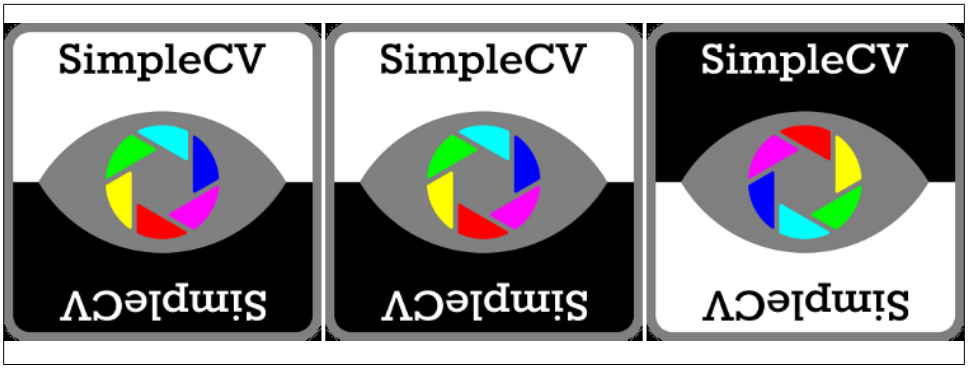


Figure 3-1. Built-in images: `simplecv`, `logo` (same as `simplecv`), and `logo_inverted`.

- `lenna`: The "Lenna" image found in many common image processing texts.

Saving an image is just as easy as loading one. In general, saving is done by passing the file name to the `save()` function. As a shortcut, if the image was already loaded from the disk, simply calling the `save()` function with no parameters will overwrite the original image. Alternatively, passing a new file name will save a new copy of the image to the disk. To demonstrate some of the options:

```
from SimpleCV import Image

img = Image("chicago.png")  ❶
img.save()  ❷

# Now save as .jpg
img.save("chicago.jpg")  ❸

# Re-saves as .jpg
img.save()  ❹
```

❶ Load the file `chicago.png`

❷ This saves the file using the original file name, `chicago.png`

❸ This now saves the image as a new file called `chicago.jpg`. Notice the change in the file extension. It is automatically converted and saved in the JPEG format.

❹ Since the last file touched was `chicago.jpg`, this will save `chicago.jpg`, and not `chicago.png`.

When saving a file loaded from either the built-in images, or fetched directly from the internet, a file name must be provided.

The SimpleCV framework actually looks at the file extension provided to determine the image format. Hence a file with a `.jpg` extension will be saved as a JPEG and

a `.png` extension will be saved as a PNG. This is done independently of the file's original format. For example, the following code is an easy way to convert an image from a JPEG to a PNG. The file extension is not case sensitive for determining the file format, but it does affect the file name. Using `img.save("myfile.PNG")` and `img.save("myfile.png")` will both create a PNG file, but the file names will preserve the caps.

```
from SimpleCV import Image

img = Image("my_file.jpg")
img.save("my_file.png")
```

Sets of Images

In addition to working with individual image files, it is common to manage a large number of files. Rather than loading them individually, the `ImageSet()` library manages sets of images. The following example demonstrates how to use an ImageSet to manage the saving of a set of files from a security camera. The `ImageSet()` command takes one optional parameter: the path to the directory containing the images to be loaded. If you don't provide a directory as a parameter, `ImageSet` will create an empty list which you can then add images to.

```
from SimpleCV import Camera, ImageSet
import time

cam = Camera()

camImages = ImageSet()  ❶

# Set to a maximum of 10 images saved
# Feel free to increase, but bewared running out of space
maxImages = 10

for counter in range(maxImages):
        # Capture a new image and add to set
        img = cam.getImage()

        camImages.append(img)  ❷

     # Show the image and wait before capturing another
        img.show()
        time.sleep(6)

camImages.save(verbose=True)  ❸
```

❶ Initialize a blank `ImageSet` object. Images will be added to this object later in the code.

❷ Append the image to the `ImageSet`

❸ Save the images to disk. Since the images did not previously have a file name, one is randomly assigned. It will be a string of random letters followed by `.png`. By passing `verbose=True` to the `save()` function, it will show the names of the files.

As a final note on `ImageSet+s, like an +Image`, an `ImageSet` also has a `show()` function. Whereas the `show()` function displays a single image when called on an `Image` object, it will show a slideshow when called on an `ImageSet` object. In the case of `ImageSet+s, the show function takes one argument of the number of seconds to pause between images. For example, +ImageSet().show(5)` will display a slide show pausing five seconds between each slide.

The Local Camera Revisited

We first looked at working with a locally connected camera in the "Hello World" program from the last chapter. For reference, here is that program again:

```
from SimpleCV import Camera, Display, Image

# Initialize the camera
cam = Camera()

# Initialize the display
display = Display()

# Snap a picture using the camera
img = cam.getImage()

# Show the picture on the screen
img.save(display)
```

The salient points to remember here are: - to work with a locally connected camera, you need to first import the Camera class - you then need to use the Camera() constructor to initialize a camera object for you to work with

Once you have a camera object—the 'cam' from the "Hello World" example—then you can use it to access the methods in the Camera class. These methods let you access the image data from the camera. If you have multiple cameras attached to the computer, then you just need to pass the device ID as a parameter in the Camera constructor. You can also pass in properties such as the brightness, hue, or exposure to the Camera constructor - as well as a calibration file, if needed. This was just a quick review, but if you would like more in-depth information, the "Introduction to the Camera" section in Chapter 2 will help.

The XBox Kinect

Historically, the computer vision market has been dominated by 2D vision systems. 3D cameras were often expensive, relegating them to niche market applications. More recently, however, basic 3D cameras have become available on the consumer market, most notably with the XBox Kinect. The Kinect is built with two different cameras. The first camera acts like a traditional 2D `640x480` webcam. The second camera generates a `640x480` depth map, which maps the distance between the camera and the object. This

obviously won't provide a Hollywood style 3D movie, but it does provide an additional degree of information that is useful for things like feature detection, 3D modeling, etc.

Want to play with Kinect code without owning a Kinect? The Freenect project has a set of drivers called *fakenect* which fake the installation of a Kinect. For more information, see *http://openkinect.org*.

Installation

The Open Kinect project provides provides free drivers that are required to use the Kinect. The standard installation on both Mac and Linux includes the Freenect drivers, so no additional installation should be required. For Windows users, however, additional drivers must be installed. Because the the installation requirements from Open Kinect may chance, please see their web site for installation requirements at *http://www.openkinect.org*.

Using the Kinect

As mentioned above, the Kinect is a combination of a standard 2D camera with a second depth sensor to capture 3D information. The overall structure of working with the 2D camera is similar to a local camera. However, initializing the camera is slightly different:

```
from SimpleCV import Kinect

# Initialize the Kinect
kin = Kinect()  ❶

# Snap a picture with the Kinect
img = kin.getImage()  ❷

img.show()
```

❶ Unlike local cameras, which are initialized by calling the `Camera()` constructor, the Kinect is initialized with the `Kinect()` constructor. If the drivers were not correctly installed, this line of code will print a warning, and future operations will fail. You should also know that unlike `Camera()`, the `Kinect()` constructor does not take any parameters.

❷ Although the initialization is different, the basic steps for capturing an image are the same. Simply call `getImage()` from the Kinect object to snap a picture with the Kinect's 2D camera.

Using the Kinect simply as a standard 2D camera is a pretty big waste of money. The Kinect is a great tool for capturing basic depth information about an object. Underneath the hood, it measures depth as a number between 0 and 1023, with 0 being the closest to the camera and 1023 being the farthest away. Although the Kinect captures values in a range from 0 to 1023, the SimpleCV framework automatically scales that range

Figure 3-2. A depth image from the Kinect

down to a 0 to 255 range. Why? Instead of treating the depth map as an array of numbers, it is often desirable to display it as a grayscale image. In this visualization, nearby objects will appear as dark grays, whereas objects in the distance will be light gray or white. To better understand this, the following example demonstrates how to extract depth information:

```
from SimpleCV import Kinect

# Initialize the Kinect
kin = Kinect()

# This works like getImage, but returns depth information
depth = kin.getDepth()

depth.show()
```

The example output, shown in the image above, shows some hints of a person in the foreground, as indicated by the darker person-shaped spot. Other more distant objects are also somewhat discernible further in the background. The image is not a traditional picture, but the relative distance of the objects still provides some indication or outline of the actual objects.

The Kinect's depth map is reduced so that it can fit into a 0 to 255 grayscale image. This reduces the granularity of the depth map. If needed, however, it is possible to get the original 0 to 1023 range depth map. The function `getDepthMatrix()` returns a NumPy matrix with the original full range of depth values. This matrix represents the `2 x 2` grid of each pixels depth. More information about the link between matrices and images is covered in the next chapter.

```
from SimpleCV import Kinect

# Initialize the Kinect
kin = Kinect()

# This returns the 0 to 1023 range depth map
depthMatrix = kin.getDepthMatrix()

print depthMatrix
```

Kinect Examples

Putting the pieces together, it is possible to create a real-time depth camera video feed using the Kinect. These examples are best run as a separate python script, rather than in the SimpleCV Shell. The three examples are:

- A video feed of the Kinect depth map.
- Using the Kinect to identify and extract just the part of an image in the foreground.
- Using the Kinect to measure an object that passes into its field of view.

The first example is a basic streaming feed from the Kinect. Like the examples in the last chapter that used the webcam to create a video feed, this provides a real time stream of images. Unlike the previous examples, however, it shows the depth map instead of the actual image.

```
from SimpleCV import Kinect

# Initialize the Kinect
kin = Kinect()

# Initialize the display
display = kin.getDepth().show()

# Run in a continuous loop forever
while (True):
     # Snaps a picture, and returns the grayscale depth map
     depth = kin.getDepth()

     # Show the actual image on the screen
       depth.save(display)
```

Networked Cameras

The previous examples in this book have assumed that the camera is directly connected to the computer. However, the SimpleCV framework can also control Internet Protocol (IP) Cameras. Popular for security applications, IP cameras contain a small web server and a camera sensor. They then stream the images from the camera over a web feed. As of the writing of this book, these cameras have recently dropped substantially in price. Low end cameras can be purchased for as little as $30 for a wired camera, and $60 for a wireless camera.

Most IP cameras support a standard HTTP transport mode, and stream video via the Motion JPEG (MJPG) format. To access a MJPG stream, use the `JpegStreamCamera` library. The basic setup is the same as before, except now the constructor must provide the address of the camera and the name of the MJPG file. This is represented by *mycamera* and *video.mjpg*, respectively, in the example below:

```
from SimpleCV import JpegStreamCamera

# Initialize the webcam by providing URL to the camera
cam = JpegStreamCamera("http://mycamera/video.mjpg")

cam.getImage().show()
```

In general, initializing an IP camera requires the following information:

- The IP address or hostname of the camera, represented by *mycamera* in the example above.
- The path to the Motion JPEG feed, represented by *video.mjpg* in the example above.
- The username and password, if required. This configuration option is demonstrated below.

If you have difficulty accessing an IP camera, try loading the URL in a web browser. It should show the video stream. If the video stream does not appear, it may be that the URL is incorrect or that there are other configuration issues. One possible issue is that the URL requires a login to access it, which is covered in more detailed below.

Many phones and mobile devices today include a built-in camera. Tablet computers and both the iOS and Android smart phones can be used as network cameras, with apps that stream the camera output to an MJPG server. To install one of these apps, search for "IP Camera" in the app marketplace on an iPhone/iPad or search for "IPCAM" on Android devices. Some of these apps are for viewing feeds from other IP cameras, so make sure that the app is designed as a server and not a viewer.

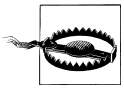
The first configuration parameter needed is the IP or hostname of the network camera, which varies from model to model. The camera manual should list this, though the exact configuration is based on both the camera's default configuration and the local network's configuration. The IP or hostname used is exactly the same as the IP or hostname used when accessing the camera via a web browser.

The next step is to find the name of the video stream file, which should end in `.mjpg`. Once the camera is online, login to the camera from a web browser. Popular username password pairs are `admin/admin` or `admin/1234`, though this information should be provided in the camera documentation. After logging in, the web page probably displays the video stream. If it doesn't, navigate to the page that does show the video stream.

Then right click on the streaming video and copy the URL. If the stream's URL is not available by right clicking, it may require a little detective work to find the MJPG stream URL for the camera. To see an initial database for some popular cameras, go to: *https://github.com/ingenuitas/SimpleCV/wiki/List-of-IP-Camera-Stream-URLs*.

If the video stream requires a username and password to access it, then you need to provide that authentication information in the URL as shown below. In the example URL, the text string `admin` should be replaced with the actual username; the string `1234` should be replaced with the actual password; the string `192.168.1.10` should be replaced with the hostname for the camera; and the string `video.mjpg` should be replaced with the name of the video stream file.

```
from SimpleCV import JpegStreamCamera

# Initialize the camera with login info in the URL
cam = JpegStreamCamera("http://admin:1234@192.168.1.10/video.mjpg")

cam.getImage().show()
```



This puts the username and password in plain text in the Python script. Make sure that the Python script is only readable by authorized users.

Notice the formatting of the URL. It takes the form: `http://username:password@hostname/MJPG_feed`. For those who have done basic HTTP authentication in the past, this is the same formatting. Once connected, the network camera will work exactly like a local camera.

IP Camera Examples

The classic real world application for an IP camera is a security camera. A wifi connected web camera can easily stream a live video feed to a central location for monitoring. These examples are focused on basic image capture. Later chapters of the book will talk about how to detect motion and other ideas for creating a more robust application. The two versions of the security camera application are demonstrated:

- Streaming a single live feed.
- Capturing multiple streams and displaying them in a single panel.

As with the previous examples, these are best run as python scripts and not in the SimpleCV Shell.

The first example is basic single IP camera. Once configured, it works like a locally connected camera. As a demonstration, the following example shows a feed captured from the IP camera:

```
from SimpleCV import JpegStreamCamera, Display
import time
```

```
#initialize the IP camera
cam = JpegStreamCamera("http://35.13.176.227/video.mjpg")  ❶

display = Display()  ❷

img = cam.getImage()
img.save(display)

while not display.isDone():
        img = cam.getImage()
        img.drawText(time.ctime())
        img.save(display)

     # This might be a good spot to also save to disk
     # But watch out for filling up the hard drive

        time.sleep(1)
```

❶ Initialize the IP camera, as described above.

❷ From this point on, the IP camera will work just like other example code described in the previous chapter.

Using Existing Images

The SimpleCV framework does not actually require a physical camera to do image processing. In addition to IP cameras and physically connected cameras, the SimpleCV framework can also process image data previously saved to disk. In other words, a pre-existing video or image can serve as a frame source. This is useful to process video captured from non-compatible devices, or for providing post-processing of previously captured video.

Virtual Cameras

One approach to using existing images is to use a Virtual Camera. Using this approach, rather than capturing data fed through the camera, the virtual camera loads a video file that is accessed as though it is a stream of video coming through a camera. By this point in the book, the overall access and use should appear familiar to most readers:

```
from SimpleCV import VirtualCamera

# Load an existing video into the virtual camera
vir = VirtualCamera("chicago.mp4", "video")

vir.getImage().show()
```

The above example looks for a video named `chicago.mp4` as the frame source. The first parameter to `VirtualCamera()` is the file name of the video to load. The second is simply the word "video", indicating that the first parameter points to a video as opposed to a

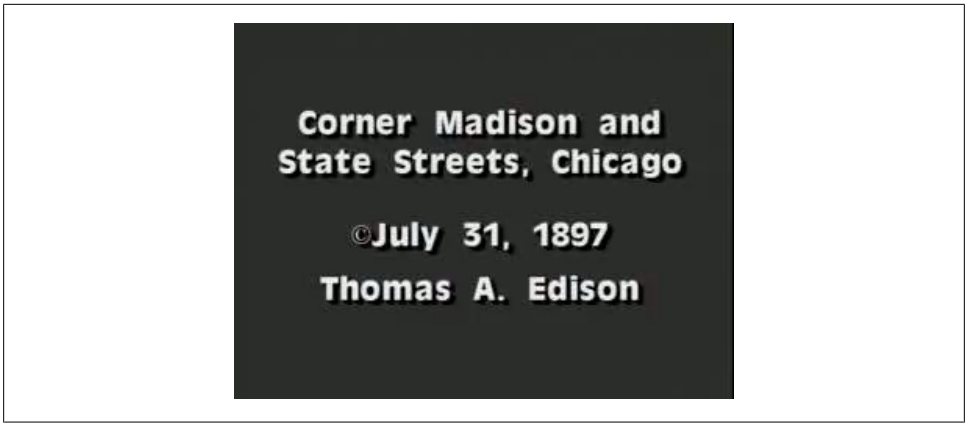


Figure 3-3. Example output from the example video file, opened with the Virtual Camera.

static image. When working with a Virtual Camera, each call to `getImage()` will advance the video by a single frame. Once the last frame of the video is reached, calling `getImage()` again will loop back to the first frame of the video.

The ability to use videos as a frame source is based on the codecs installed. The installed codecs will vary from system to system. Under Windows, the video files are decoded with Video for Windows. Linux uses ffmpeg. Mac OS uses QuickTime to decode video files. If in doubt, try to open the file first in another application to see if the video is readable.

Because Virtual Cameras are based on the Camera class, other camera functionality also works. For example, the `Camera.live()` function introduced in the previous chapter will work with virtual devices too. Just like with a regular webcam, simply click with the left mouse button on any point on the screen to get information about the pixel coordinates and color. Notice that at the end of the video, it automatically loops back to the beginning and plays it again.

```
from SimpleCV import VirtualCamera

vir = VirtualCamera("chicago.mp4", "video")

# This plays the video
vir.live()
```

Instead of a video file, a single image can also be used as a virtual camera:

```
from SimpleCV import VirtualCamera

# Notice: the second parameter is now: image
vir = VirtualCamera("chicago.png", "image")

vir.getImage().show()
```

Figure 3-4. Example of the virtual feed using the `live()` function.

This time, the `VirtualCamera` function is passed the path to an image file and then the word "image", indicating that the first parameter is a single image. Since this is only a single image and not a video, `getImage()` always returns the same image. A lot of the same functionality could be achieved by simply loading the image. In fact, the following two lines of code create identical output to the end user:

```
from SimpleCV import Image, VirtualCamera

# These two lines of code do the same thing
VirtualCamera("chicago.png", "image").getImage().show()
Image("chicago.png").show()
```

Notice that the overall functionality of the virtual camera with single images looks a lot like working with the `Image` library. Because the `Image` library also includes additional features to handle drawing and the extraction of features, it is usually the preferred method of working with images.

Examples

The examples provided in this chapter cover a range of applications of image sources. The are designed to demonstrate the range of potential sources and their practical application. The examples cover the following topics

- Converting a directory of images to the JPEG format.
- Using the Kinect to segment an image to extract the nearest object.
- Using the Kinect to measure the height of an object.
- Combining multiple IP camera feeds into a single feed for easy viewing.

Converting Set of Images

This example uses the `ImageSet` library to convert an entire directory of images to the `.jpg` format. It first uses the `ImageSet` to load the directory of images. It then iterates through the set, changing the name of the file to have a `.jpg` extension. Then it re-saves the file with the new file extension, and automatically converts it to the new file format during the save process.

```
from SimpleCV import ImageSet

set = ImageSet(".")  ❶

for img in set:  ❷
        oldname = img.filename  ❸
        newname = oldname[0:-3] + 'jpg'  ❹
        print "Converting " + oldname + " to " + newname
        img.save(newname)  ❺
```

❶ The first step is to get a collection of all the files. This example assumes that the code is run from the same directory that contains the images, the Chapter 3 folder. Note that although the chicago.mp4 file is also in the Chapter 3 folder, it is not an image file, so `ImageSet` will skip it.

❷ Next, loop over all image files. The `img` value represents each individual image while looping through the set.

❸ This line extracts the original file name of the image.

❹ This creates the new file name by first finding the name of the original file without the extension (`oldname[0:-3]`), and then appending the `png` extension.

❺ Finally, save a new copy of the file with the `.jpg` extension. The `.jpg` extension will automatically convert the file, and save it in the JPEG format.

Segmentation with the Kinect

For the next application, the depth information from the Kinect can be used to extract objects from the foreground, and then erase the background. In computer vision, this is known as segmentation, which is the process of dividing an image into groups of related content in order to make the image easier to analyze. Segmentation is covered in greater depth in later chapters, but this example shows how the Kinect can also be used to perform basic segmentation on the image.

Don't have a Kinect? Example images are provided in the electronic supplement. The 2D image is `kinect-image.png` and the depth information is stored in `kinect-depth.png`. Modify the code below to load the images instead of capturing them via the Kinect.

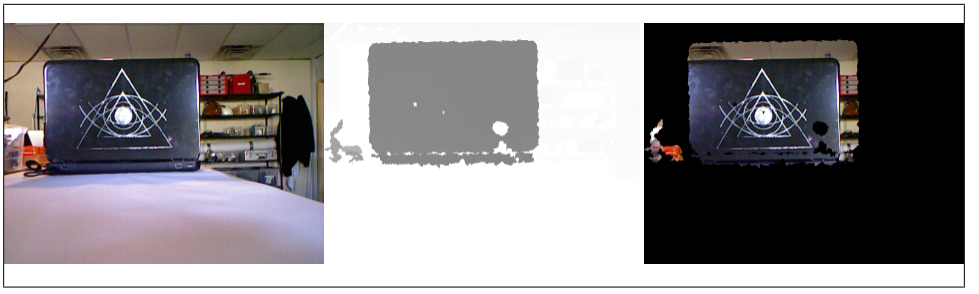


Figure 3-5. Example Result of using depth for segmentation. Left: The 2D image. Center: The depth information. Right: The final result.

```
from SimpleCV import Kinect
import time

# Initialize the Kinect
kin = Kinect()

# Get the image and depth information
dep = kin.getDepth()  ❶
img = kin.getImage()  ❷

# Turn into a pure black and white image for segmentation
fore = dep.binarize(190).invert()  ❸
fore_only = img - fore  ❹

fore_only.show()  ❺

# Keep the image open for 10 seconds
time.sleep(10)
```

❶ This gets the depth information from the Kinect. The depth information will be used to detect which parts of the image are in the foreground and which parts are in the background.

❷ Next capture an image. This should be done with a still or very slow moving object so that the depth image captured in the previous step matches the picture captured in this step.

❸ This binarizes the depth image, which converts it into a pure black and white image only (no shades of gray). In other words, rather than have many different depths, it will just have foreground and background. The binarization threshold, `190` may need to be adjusted based on the environment. It is then inverted, changing the black to white, and the white to black. By the end of this step, objects in the foreground are black and objects in the background are white.

❹ Subtract the black-and-white image from the main image. This has the effect of removing the background.

❺ Finally, show the resulting image.

The resulting segmentation is not perfect. It includes a little stuff from right and above the object and misses some material from left and below the object. This is an artifact of the distance between the normal image camera and the depth sensor. Objects further away from the camera will have less of a problem with this.

This is the first of several tricks for extracting important features from an image. It demonstrates how the depth information can be used to reduce the image to only its key components. As the book progresses, we cover these concepts in greater detail, and introduces a variety of different tips and tricks.

Kinect for Measurement

The final Kinect example goes a little further. As demonstrated in the previous example, the 3D depth information is useful for identifying an object of interest since the object of interest is likely closer to the camera than the background objects. This example detects the largest object in the field of view, and then tries to measure its height.

```
from SimpleCV import Kinect, Display
import time

# Initialize the Kinect
kin = Kinect()

# Initialize the display
disp = Display((640, 480))  ❶

# This should be adjusted to set how many pixels
# represent an inch in the system's environment
pixelsToInches = 6  ❷

while not disp.isDone():
        img = kin.getDepth()
        blobs = img.binarize().findBlobs()  ❸
        if (blobs):
                img.drawText(str(blobs[-1].height()/ pixelsToInches) + " inches",
                    10, 10)  ❹
        img.save(disp)
        time.sleep(1)
```

❶ This initializes the display with a specific resolution of `640 x 480`, which matches the output from the Kinect.

❷ This is a calibration value that will need to be adjusted based on the environment in which the code is used. The code will measure how many pixels high an object is, but it needs a way to translate pixels into inches.

❸ Now binarize the image and find blobs. The previous example showed that binarizing the image will help pick out the nearest object. Finding blobs will then look for a big contiguous object, which is assumed to be the object to be measured.

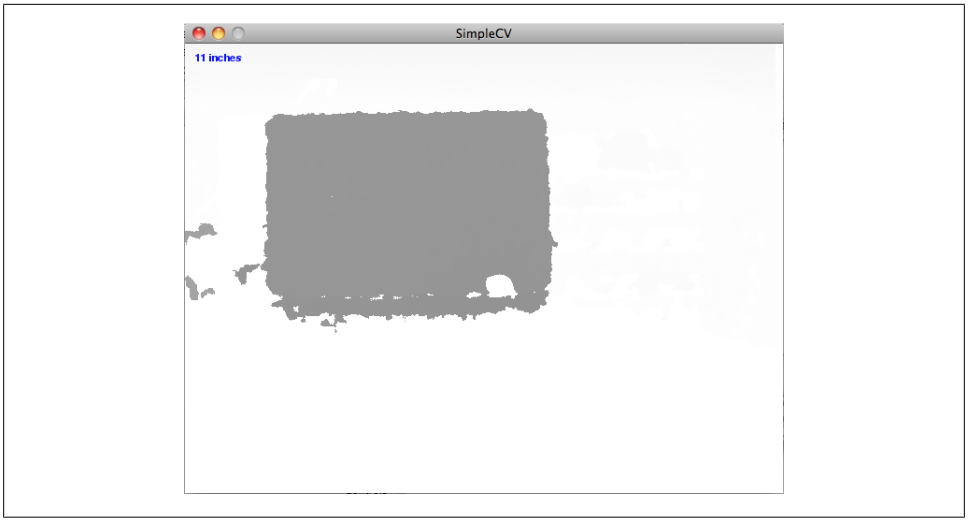


Figure 3-6. Example from height calculator program showing that nearby laptop is 11 inches tall.

❹ The drawText() function should be familiar from the previous chapter. However `blobs[-1].height()` is new. By using `blobs[-1]`, it retrieves the largest blob found. Then it calls `height()` on that blob to get the height in pixels of that object. That measurement is displayed on the screen.

This example requires some calibration. To make it work, start with an object of a known height, and place it in front of the camera. Adjust the `pixelsToInches` value until the output shows the correct height in inches. Once this calibration is done, the Kinect can be used to measure other objects. The measurements will not be perfect because of limitations in the Kinect sensor, but they should provide a decent estimate of height.

Note, however, that the objects must be put in the same spot as the original object used for calibration. (As an extra credit assignment, the depth values could also be calibrated and used to measure physical distance from the camera. This could then be used to measure an object at a more arbitrary distance from the camera. Though this is getting more complicated than is appropriate for this early in the book.)

Multiple IP Cameras

So far, the security cam examples use only one camera. However, it is common to have multiple security cameras, all of which should be monitored at the same time. The following block of example code shows how to do this by combining the output of four cameras into one display window. This code introduces the `sideBySide()` function, which combines two images together. The options for the `sideBySide()` function include the name of the image you want to add to the original image, which side to place the image on (left, right, top, or bottom), and whether or not you want the images

scaled. By default, `sideBySide()` will scale a smaller image to match the size of the larger image.

```
from SimpleCV import Camera, Display
import time

#initialize the IP cameras
cam1 = JpegStreamCamera("http://admin:1234@192.168.1.10/video.mjpg")  ❶
cam2 = JpegStreamCamera("http://admin:1234@192.168.1.11/video.mjpg")
cam3 = JpegStreamCamera("http://admin:1234@192.168.1.12/video.mjpg")
cam4 = JpegStreamCamera("http://admin:1234@192.168.1.13/video.mjpg")

display = Display((640,480))  ❷

while not display.isDone():
     img1 = cam1.getImage().resize(320, 240)  ❸
     img2 = cam2.getImage().resize(320, 240)
     img3 = cam3.getImage().resize(320, 240)
     img4 = cam4.getImage().resize(320, 240)

     top = img1.sideBySide(img2)  ❹
     bottom = img3.sideBySide(img4)

     combined = top.sideBySide(bottom, side="bottom")  ❺

     combined.save(display)  ❻
     time.sleep(5)
```

❶ Initialize the four IP cameras. Note that each camera has a unique hostname.

❷ Initialize the display at `640 x 480`. This area will support four images, each of size `320 x 240` to be stacked in a two by two grid.

❸ Capture the image, and then resize to `320 x 240`, so they will all fit into the display.

❹ The `sideBySide()` function takes two images, and pastes them together side by side into one. First assemble the top and bottom rows of the grid.

❺ Then take the top and bottom rows of the grid and paste them together into the full grid.

❻ Finally, display the results to the screen, and sleep for 5 seconds.

Figure 3-7. Example A display showing the output from multiple security cameras.

CHAPTER 4

Pixels and Images

The previous chapters have provided a broad overview of working with the SimpleCV framework, including how to capture images and display them. Now it is time to start diving into the full breadth of the framework, beginning with a deeper look at images, color, drawing, and an introduction to feature detection. This chapter will tunnel down to the level of working with individual pixels, and then move up to the higher level of basic image manipulation. Not surprisingly, images are the central object of any vision system. They contain all of the raw material that is then later segmented, extracted, processed, and analyzed. In order to understand how to extract information from images, it is first important to understand the components of a computerized image. In particular, this chapter emphasizes:

- Working with pixels, which are the basic building blocks of images
- Scaling and cropping images to get them to a manageable size
- Rotating and warping images to fit them into their final destination
- Morphing images to accentuate features, reduce noise, etc.

Pixels

Pixels are the basic building blocks for a digital image. A pixel is what we call the color or light values that occupy a specific place in an image. You can think about an image as a big grid, with each square in the grid containing one color or pixel. This grid is sometimes called a bitmap. An image with a resolution of `1024 x 768` is a grid with 1,024 columns and 768 rows, which then contains `1,024*768=786,432` pixels. Knowing how many pixels are in an image doesn't tell you the physical dimensions of the image, though - because a pixel is not a unit of size in that manner. That is to say, 1 pixel does not equate to 1 millimeter, 1 micrometer, or 1 nanometer. Instead, how "large" a pixel is will depend on the pixels per inch (ppi) setting for that image.

Each pixel is represented by a number or a set of numbers—and the range of these numbers is called the color depth or bit depth. In other words, the color depth indicates

the maximum number of potential colors that can be used in an image. An 8-bit color depth uses the numbers 0-255 (or an 8-bit byte) for each color channel in a pixel. This means a `1024 x 768` image with a single channel (black and white) 8-bit color depth would create a 768 kB image. Most images today use 24-bit color or higher, allowing three 0-255 numbers per channel. This increased amount of data about the color of each pixel means a `1024 x 768` image would take 2.25 MB. As a result of these substantial memory requirements, most image file formats don't store pixel-by-pixel color information. Image files such as GIF, PNG, and JPEG use different forms of compression to more efficiently represent images.

Most pixels come in two flavors: grayscale and color. In a grayscale image, each pixel has only a single value representing the light value, with zero being black and and 255 being white. Most color pixels have three values representing red, green, and blue (RGB). Other non-RGB representation schemes exist, but RGB is the most popular format. The three colors are each represented by one byte, or a value from 0 to 255, which indicates the amount of the given color. These are usually combined into an RGB triplet in a (red, green, blue) format. For example, (125, 0, 125) means that the pixel has some red, no green, and some blue, representing a shade of purple. Some other common examples include:

- Red: (255, 0, 0)
- Green: (0, 255, 0)
- Blue: (0, 0, 255)
- Yellow: (255, 255, 0)
- Brown: (165, 42, 42)
- Orange: (255, 165, 0)
- Black: (0, 0, 0)
- White: (255, 255, 255)

Remembering those codes can be somewhat difficult. To simplify this, the `Color` class includes a host of pre-defined colors. For example, to use the color teal, rather than needing to know that it is RGB (0, 128, 128), simply use:

```
from SimpleCV import Color

# An easy way to get the RGB triplet values for the color teal.
myPixel = Color.TEAL
```

Similarly, to lookup the RGB values for a known color:

```
from SimpleCV import Color

# Prints (0, 128, 128)
print Color.TEAL
```

Notice the convention that all the color names are written in all CAPS. To get green, use `Color.GREEN`. To get red, use `Color.RED`. Most of the standard colors are available.

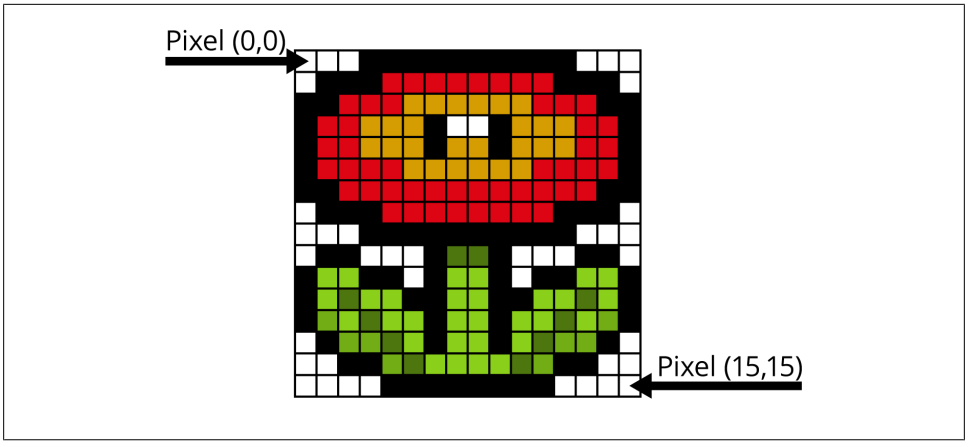


Figure 4-1. Pixels and Coordinates. Note that `(0, 0)` is in the upper left corner.

For those readers who would not otherwise guess that `Color.PUCE` is a built-in color—it is a shade red—simply type `help Color` at the SimpleCV Shell prompt and it will list all available colors. Many functions include a color parameter, and color is an important tool for segmenting images. It would be worthwhile to take a moment and review the pre-defined color codes provided by the SimpleCV framework.

Images

Bitmaps and Pixels

Underneath the hood, an image is a two dimensions array of pixels. A two dimensional array is like a piece of graph paper: there are a set number of vertical units and a set number of horizontal units. Each square is indexed by a set of two numbers: the first number representing the horizontal row for that square, and the second number is the vertical column. Perhaps not surprisingly, the row and columns are indexed by their `x` and `y` coordinates.

This approach, called the "cartesian coordinate" system, should be intuitive based on previous experience with graphs in middle school math courses. However, computer graphics vary from tradition in a very important way. In normal graphing applications, the origin point `(0, 0)` is in the lower left corner. In computer graphics applications, the `(0, 0)` point is the **upper** left corner.

Since the pixels in an image are also in a grid, it's very easy to map pixels to a two dimensional array. The low-resolution image below of a flower below demonstrates the indexing of pixels. Notice that pixels are zero indexed, meaning that the upper left corner is at `(0, 0)` not `(1, 1)`.

Figure 4-2. The picture, Jacopo Pontormo

The information for an individual pixel can be extracted from an image the same way an individual element of an array is referenced in Python. The next example blocks shows how to extract the pixel at `(120, 150)` from the picture of Jacopo Pontormo, as demonstrated in Figure 4-2.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Gets the information for the pixel located at
# x coordinate = 120, and y coordinate = 150
pixel = img[120, 150]

print pixel
```

The value of `pixel` will become the RGB triplet for the pixel at `(120, 150)`. As a result, `print pixel` will return `(242.0, 222.0, 204.0)`.

The following example code will do exactly the same thing, but uses the getPixel() function instead of the index of the array. This is the more object-oriented programming approach compared to extracting the pixel directly as an array.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Uses getPixel() to get the information for the pixel located
# at x coordinate = 120, and y coordinate = 150
pixel = img.getPixel(120, 150)

print pixel
```

Want the grayscale value of a pixel in a color image? Rather than converting the whole image to grayscale and then returning the pixel, use `getGrayPixel(x, y)`.

Accessing pixels by their index can sometimes create problems. In the example above, trying to use `img[1000, 1000]` will throw an error, and `img.getPixel(1000, 1000)` will give a warning because the image is only `300 x 389`. Because the pixel indexes start at zero, not one, the dimensions must be in the range 0-299 on the x-axis and 0-388 on the y-axis. To avoid problems like this, use the `width` and `height` properties of an image to find its dimensions. For example:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Print the pixel height of the image
# Will print 300
print img.height

# Print the pixel width of the image
# Will print 389
print img.width
```

In addition to extracting RGB triplets from an image, it is also possible to change the image using an RGB triplet. The following example will extract a pixel from the image, zero out the green and blue components, preserving only the red value, and then put that back into the image.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Retrieve the RGB triplet from (120, 150)
(red, green, blue) = img.getPixel(120, 150)  ❶

# Change the color of the pixel+
img[120, 150] = (red, 0, 0)  ❷

img.show()
```

❶ By default, each pixel is returned as a tuple of the red, green, and blue components. (The next chapter covers this in more detail.) This conveniently stores each separate value in its own variable, appropriately named `red`, `green`, and `blue`.

❷ Now instead of using the original value of green and blue, those are set to zero. Only the original red value is preserved. This effect is demonstrated in Figure 4-3

Since only one pixel was changed, it will be hard to see the difference, but now the pixel at `(120, 150)` is a dark red color. To make it easier to see, resize the image to 5 times its previous size, using the `resize()` function.

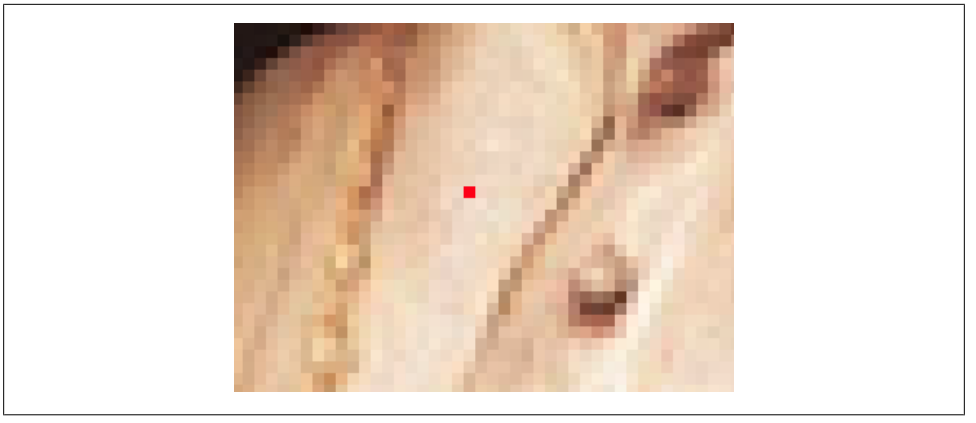


Figure 4-3. Left The image with the new red pixel. Right: A zoomed view of the changed pixel.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Get the pixel and change the color
(red, green, blue) = img.getPixel(120, 150)
img[120, 150] = (red, 0, 0)

# Resize the image so it is 5 times bigger then it's original size
bigImg = img.resize(img.width*5, img.height*5)

bigImg.show()
```

The much larger image should make it easier to see the red-only pixel that changed. Notice, however, that in the process of resizing the image, the single red pixel is interpolated, resulting in extra red in nearby pixels, as demonstrated in Figure 4-4:

Right now, this looks like random fun with pixels but with no actual purpose. However, pixel extraction is an important tool when trying to find and extract objects of a similar color. Most of these tricks are covered later in the book, but to provide a quick preview of how it is used, the following example looks at the color distance of other pixels compared with a given pixel.

```
from SimpleCV import Image
img = Image('jacopo.png')

# Get the color distance of all pixels compared to (120, 150)
distance = img.colorDistance(img.getPixel(120, 150))

# Show the resulting distances
distance.show()
```

Figure 4-4. The original red pixel after resizing

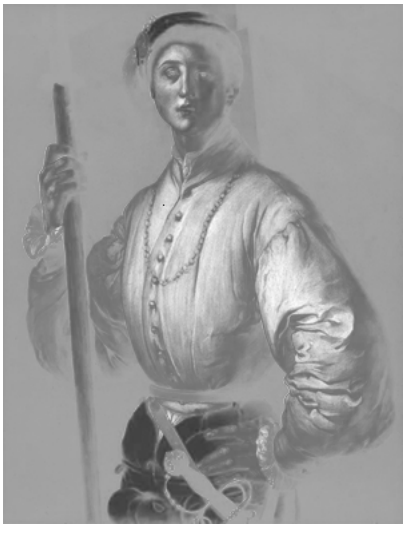


Figure 4-5. Color distance compared to the pixel at (100, 50)

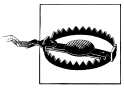
Image Scaling

The block of code above shows the next major concept with images: scaling. In the above example, both the width and the height were changed by taking the `img.height` and `img.width` parameters and multiplying them by 5. In this next case, rather than entering the new dimensions, the `scale()` function will resize the image

Figure 4-6. Poor Jacobo has gained weight

with just one parameter: the scaling factor. For example, the following code will resize the image to 5 times its original size.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Scale the image by a factor of 5
bigImg = img.scale(5)

bigImg.show()
```



Notice that two different functions were used in the two previous examples. The `resize()` function takes two arguments representing the new dimensions. The `scale()` function takes just one argument with the scaling factor (i.e., how many times bigger or smaller you want the image).

When using the `resize()` function and the aspect ratio (the ratio of the width to height) changes, it can result in funny stretches to the picture, as is demonstrated in the next example.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Resize the image, keeping the original height,
# but doubling the width
bigImg = img.resize(img.width * 2, img.height)

bigImg.show()
```

In this example, the image is stretched in the width dimension but no change is made to the height, as demonstrated in Figure 4-6. To resolve this problem, use adaptive

Figure 4-7. Resized with adaptive scaling.

scaling with the `adaptiveScale()` function. It will create a new image with the dimensions requested. However, rather than wrecking the proportions of the original image, it will add padding. For example:

```
from SimpleCV import Image

# Load the image
img = Image('jacopo.png')

# Resize the image, but use the +adaptiveScale()+ function to maintain
# the proportions of the original image
adaptImg = img.adaptiveScale((img.width * 2, img.height))

adaptImg.show()
```

In the resulting image, the original proportions are preserved, with the image content placed in the center of the image, and padding added to the top and bottom of the image.

The `adaptiveScale()` function takes a tuple of the image dimensions, not separate `x` and `y` arguments. Hence, the double parentheses.

Adaptive scaling is particularly useful when trying to enforce a standard image size on a collection of heterogeneous images. This example creates `50 x 50` thumbnail images into a directory called "thumbnails".

```
from SimpleCV import ImageSet
from os import mkdir

# Create a local directory named thumbnails for storing the images
mkdir("thumbnails")

# Load the files in the current directory
```

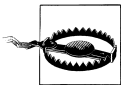
```
set = ImageSet(".")

for img in set:
     print "Thumbnailing: " + img.filename

     # Scale the image to a +50 x 50+ version of itself,
     # and then save it in the thumbnails folder
     img.adaptiveScale((50, 50)).save("thumbnails/" + img.filename)

print "Done with thumbnails.  Showing slide show."

# Create an image set of all of the thumbnail images
thumbs = ImageSet("./thumbnails/")

# Display the set of thumbnailed images to the user
thumbs.show(3)
```

The `adaptiveScale()` function has an additional parameter, `fit`, that defaults to true. When fit is true the function tries to scale the image as much as possible, while adding padding to ensure proportionality. When fit is false, instead of padding around the image to meet the new dimensions, it will instead scale it in such a way that the smallest dimension of the image fits the desired size. Then it will crop the larger dimension so that the resulting image still fits the proportioned size.

A final variant of scaling is the `embiggen()` function. This changes the size of the image by adding padding to the sides but not altering the original image. In some other image editing software, this is the equivalent of changing the canvas size without changing the image. The `embiggen()` function takes three arguments:

- A tuple with the width and height of the embiggened image.
- The color for the padding to place around the image. By default, this is black.
- A tuple of the position of the original image on the larger canvas. By default, the image is centered.

```
from SimpleCV import Image, Color

img = Image('jacopo.png')

# Embiggen the image, put it on a green background, in the upper right
emb = img.embiggen((350, 400), Color.GREEN, (0, 0))

emb.show()
```



The `embiggen()` function will throw a warning if trying to embiggen an image into a smaller set of dimensions. For example, the `300 x 389` example image can not be embiggened into a `150 x 200` image.

Figure 4-8. An embiggened image with changed color and position.

Image Cropping

In many image processing applications, only a portion of the image is actually important. For instance, in a security camera application, it may be that only the door—and whether anyone is coming or going—is of interest. Cropping lets you speed up processing by only processing a "region of interest" rather than the entire image. The SimpleCV framework has two mechanisms for cropping: the `crop()` function, and Python's slice notation.

Image.crop() takes four arguments which represent the region to be cropped. The first two are the x and y coordinates for the upper left corner of the region you want to start the crop from, and the last two are the width and height of the area you want cropped.

For example, to crop out just the bust of the picture:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Crop starting at +(50, 5)+ for an area 200 pixels wide by 200 pixels tall
cropImg = img.crop(50, 5, 200, 200)

cropImg.show()
```

When performing a crop, it is sometimes more convenient to specify the center of the region of interest rather than the upper left corner. To crop an image from the center, add one more parameter, `centered = True`, with the result shown in Figure 4-10.

Figure 4-9. Cropping to just the bust of the image.

Figure 4-10. The image cropped around the center

```
from SimpleCV import Image

img = Image('jacopo.png')

# Crop the image starting at the center of the image
cropImg = img.crop(img.width/2, img.height/2, 200, 200, centered=True)

cropImg.show()
```

Crop regions can also be defined by image features. Many of these features are covered later in the book, but blobs were briefly introduced in previous chapters. As with other features, the SimpleCV framework can crop around a blob. For example, a blob detection can also find the torso on the picture.

Figure 4-11. Jacopo cropped using blobs.

```
from SimpleCV import Image

img = Image('jacopo.png')

blobs = img.findBlobs()  ❶

img.crop(blobs[-1]).show()  ❷
```

❶ This will find the blobs in image.

❷ The `findBlobs()` function returns the blobs in ascending order by size. This will be covered in greater detail in later chapters. In this example, that means the bust of the image

Once cropped, the image should look like Figure 4-11:

The crop function is also implemented for `Blob` features, so the above code could also be written as follows. Notice that the `crop()` function is being called directly on the blob object instead of the image object.

```
from SimpleCV import Image

img = Image('jacopo.png')

blobs = img.findBlobs()

# Crop function being called directly on the blob object
blobs[-1].crop().show()
```

Image Slicing

For the Python aficionados, it is also possible to do cropping by directly manipulating the two dimensional array of the image. Individual pixels could be extracted by treating

the image like an array and specifying the (x, y) coordinates. Python can also extract ranges of pixels as well. For example, img[start_x:end_x, start_y:end_y] will provide a cropped image from (start_x, start_y) to (end_x, end_y). Not including a value for one or more of the coordinates means that the border of the image will be used as the start or end point. So something like img[: , 300:] works. That will select all of the x values, and all of the y values that are greater than 300. In essence, any of Python's functions for extracting subsets of arrays will also work to extract parts of an image, and thereby return a new image. Because of this, you can use Python's slice notation instead of the crop function:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Cropped image that is 200 pixels wide and 200 pixels tall starting at (50, 5).
cropImg = img[50:250,5:205]

cropImg.show()
```

When you use slice notation, you specify the start and end locations. When you use crop, you specify a starting coordinate and a width and height.

Transforming Perspectives: Rotate, Warp, and Shear

When writing a vision application, you might assume that the camera is positioned squarely to view an image, and that the top of the image is "up". However, sometimes the camera is held at an angle to an object, or it isn't oriented the way that you want. This can complicate the image analysis. Fortunately, sometimes you can fix these kinds of problems with rotations, shears, and skews.

Spin, Spin, Spin Around

The simplest operation is to rotate the image so that it is correctly oriented. This is accomplished with the rotate() function, which only has one required argument, angle. This value is the angle, in degrees, that you wish to rotate the image. Negative values for the angle rotate the image clockwise, and positive values rotate it counter-clockwise. To rotate the image 45 degrees counter-clockwise:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Rotate the image counter-clockwise 45 degrees
rot = img.rotate(45)

rot.show()
```

Figure 4-12. The image rotated 45 degrees to the left.

Generally, rotation means to rotate around the center point. However, a different axis of rotation can be chosen by passing an argument to the `point` parameter. This parameter is a tuple of the (x, y) coordinate for the new point of rotation.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Rotate the image around the coordinates +(16, 16)+
rot = img.rotate(45, point=(16, 16))

rot.show()
```

Note that when the image is rotated, if part of the image falls outside the original image dimensions, that section is cropped. The `rotate()` function has a parameter called `fixed` to control this. When fixed is set to false, the algorithm will return a resized image, where the size of the image is set to include the whole image after rotation.

For example, to rotate the image without clipping off the corners:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Rotate the image, and then resize it so the content isn't cropped
rot = img.rotate(45, fixed=False)

rot.show()
```

Figure 4-13. Rotation around `16, 16`.

Figure 4-14. Rotation with the canvas resized to fit the whole image.

Even when defining a rotation point, if the `fixed` parameter is false, the image will still be rotate about the center. The additional padding around the image essentially compensates for the alternative rotation point.

Figure 4-15. A mirror image of Jacopo

Finally, for convenience, the image can be scaled at the same as it is rotated. This is done with the `scale` parameter. The value of the parameter is a scaling factor, similar to the scale function.

```
from SimpleCV import Image

img = Image('jacopo.png')

# Rotate the image and make it half the size
rot = img.rotate(90, scale=.5)

rot.show()
```

Flipping Images

Similar to rotating an image, an image can also be flipped across its horizontal or vertical axis. This is done with the `flipHorizontal()` and `flipVertical()` functions. To flip the image across its horizontal axis:

```
from SimpleCV import Image

img = Image('jacopo.png')

# Flip the image along the horizontal axis, and then display the results
flip = img.flipHorizontal()

flip.show()
```

This example below applies the horizontal flip to make a web camera act like a mirror, perhaps so you can check your hair or apply your makeup with your laptop webcam.

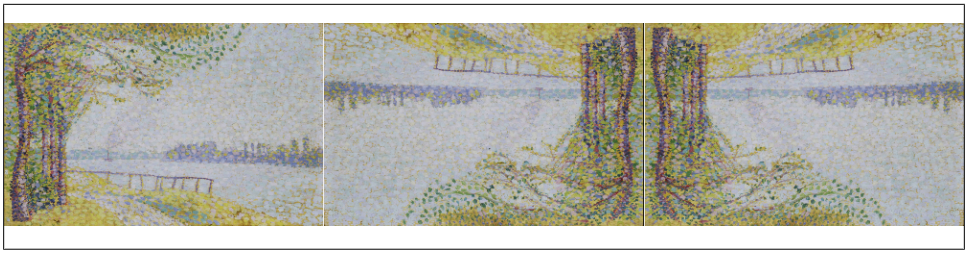


Figure 4-16. Left: The original image. Center: The image rotated 180 degrees. Right: The image flipped vertically.

```
from SimpleCV import Camera, Display

cam = Camera()

# The image captured is just used to match Display size with the Camera size
disp = Display( (cam.getProperty('width'), cam.getProperty('height')) )

while disp.isNotDone():
    cam.getImage().flipHorizontal().save(disp)
```

Note that a flip is not the same as a rotation by 180 degrees. The image below demonstrates the difference between flips and rotations.

Shears and Warps

You can also skew an image, or a portion of an image, so it fits in some other shape. A common example of this is overlaying an image on top of a square object that is viewed at an angle. WHen viewing a square object at an angle, the corners of the square no longer appear to be 90 degrees. Instead, there To align a square object to fit into this angular space, its edges must be adjusted. Underneath the hood, it is performing an affine transformation, though this is more commonly called a shear.

The tricky part when doing warps is finding all the `(x, y)` coordinates. Use `Camera.live()` and click on the image to help find the points you want.

To demonstrate shearing, the following block of code can be used to fix the Leaning Tower of Pisa. Granted, it makes the other building in the picture tip too far to the left, but some people are never happy.

```
from SimpleCV import Image

img = Image('pisa.png')

corners = [(0, 0), (450, 0), (500, 600), (50, 600)] ❶
```

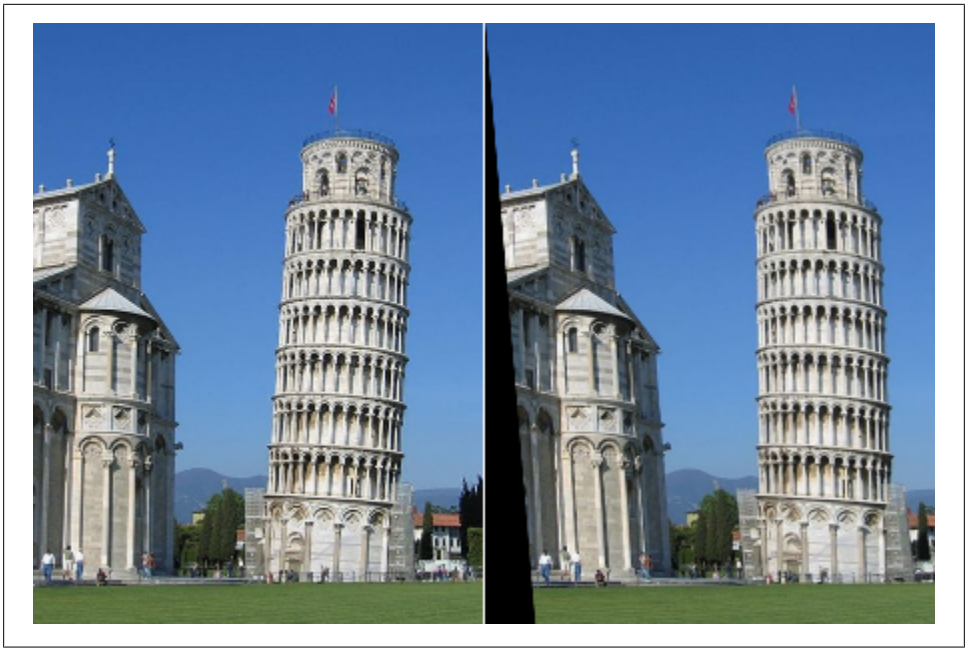


Figure 4-17. Left: The original leaning tower. Right: The repaired version of the tower.

```
straight = img.shear(corners)  ❷

straight.show()
```

❶ This is a list of the corner points for the sheared image. The original image is `450 x 600`. To fix the tower, the lower right corners are shifted by 50 pixels to the right. Note that the points for the new shape are passed in clockwise order, starting from the top left corner.

❷ Now simply call the `shear()` function, passing the list of new corner points for the image.

In addition to shearing, images can also be warped by using the `warp()` function. Warping also takes an array of corner points as its parameters. Similar to shearing, it is used to stretch an image and fit it into a non-rectangular space.

A shear will maintain the proportions of the image. As such, sometimes the actual corner points will be adjusted by the algorithm. In contrast, a warp can stretch the image so that it fits into any new shape.

Everybody wants to be on television, but with this next example, you'll now have the chance to be on TV and go back in time. The impudent might even call it a "time warp"...

```
from SimpleCV import Camera, Image, Display

tv_original = Image("family.png")  ❶

tv_coordinates = [(285,  311), (367, 311), (368, 378), (286, 376)]  ❷

tv_mask = Image(tv_original.size()).invert().warp(tv_coordinates)  ❸

tv = tv_original - tv_mask  ❹

cam = Camera()

disp = Display(tv.size())

# While the window is open, keep updating updating
# the TV with images from the camera
while disp.isNotDone():
     bwimage = cam.getImage().grayscale().resize(tv.width, tv.height)  ❺

     on_tv = tv + bwimage.warp(tv_coordinates)  ❻

     on_tv.save(disp)
```

❶ This is the image we'll be using for the background. The image captured via the web camera will be placed on top of the TV.

❷ These are the coordinates of the corners of the television.

❸ `Image(tv_original.size())` creates a new image that has the same size as the original TV image. By default, this is an all black image. The invert function makes it white. Th warp function then creates a white warped region in the middle, based on the coordinates previously defined for the TV. The result is Figure 4-18.

❹ Using image subtraction, the TV is now removed from the image. This is a trick that will be covered more extensively in the next chapter.

❺ Now capture an image from the camera. To be consistent with the black and white background image, convert it to grayscale. In addition, since this image will be added to the background image, it needs to be resized to match the background image.

❻ Finally, do another warp to make the image from the camera fit into the TV region of the image. This is then added onto the background image.

Image Morphology

It is always preferable to control the real-world lighting and environment in order to maximize the quality of an image. However, even in the best of circumstances, an image will include pixel-level noise. That noise can complicate the detection of features on the image, so it is important to clean it up. This is the job of morphology.

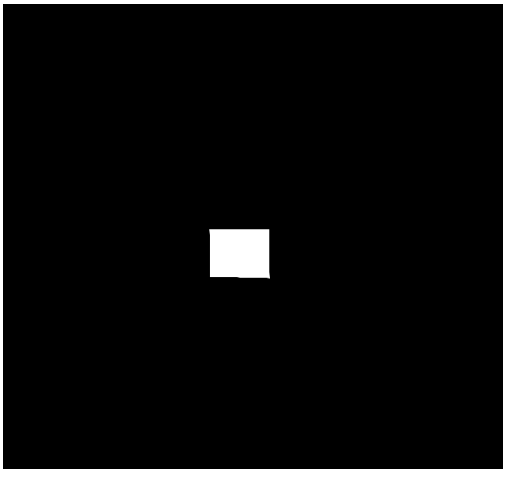


Figure 4-18. The mask of for the TV image. This is white where the TV sits, and black otherwise.

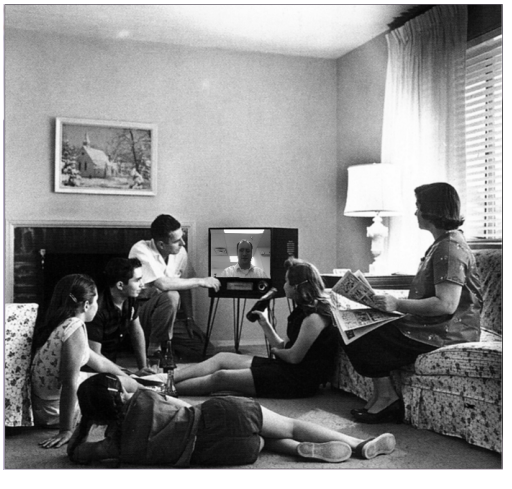


Figure 4-19. Sample output from this example. Notice the handsome book author appearing on the television.

Binarization

Many morphology functions will work with color images, but they are easiest to see in action when working with a binary (2-color) image. This means it is literally black and white, with no shades of gray. To create a binary image, use the `binarize()` function:

```
from SimpleCV import Image

img = Image('jacopo.png')
```

Figure 4-20. The Jacopo image, binarized

```
imgBin = img.binarize()
imgBin.show()
```

Whenever an image is binarized, the system needs to know which pixels get converted to black and which to white. This is called a "threshold", and any pixel where the grayscale value falls under the threshold is changed to white. Any pixel above the threshold is changed to black. By default, the SimpleCV framework uses a technique called Otsu's method to dynamically determine the binarized values. However, the binarize function also takes a parameter value between 0-255. The following example code shows the use of binarization at several levels:

```
from SimpleCV import Image

img = Image('trees.png')

# Using Otsu's method
otsu = img.binarize()

# Specify a low value
low = img.binarize(75)

# Specify a high value
high = img.binarize(125)

img = img.resize(img.width*.5, img.height*.5)
otsu = otsu.resize(otsu.width*.5, otsu.height*.5)
low = low.resize(low.width*.5, low.height*.5)
high = high.resize(high.width*.5, high.height*.5)
```

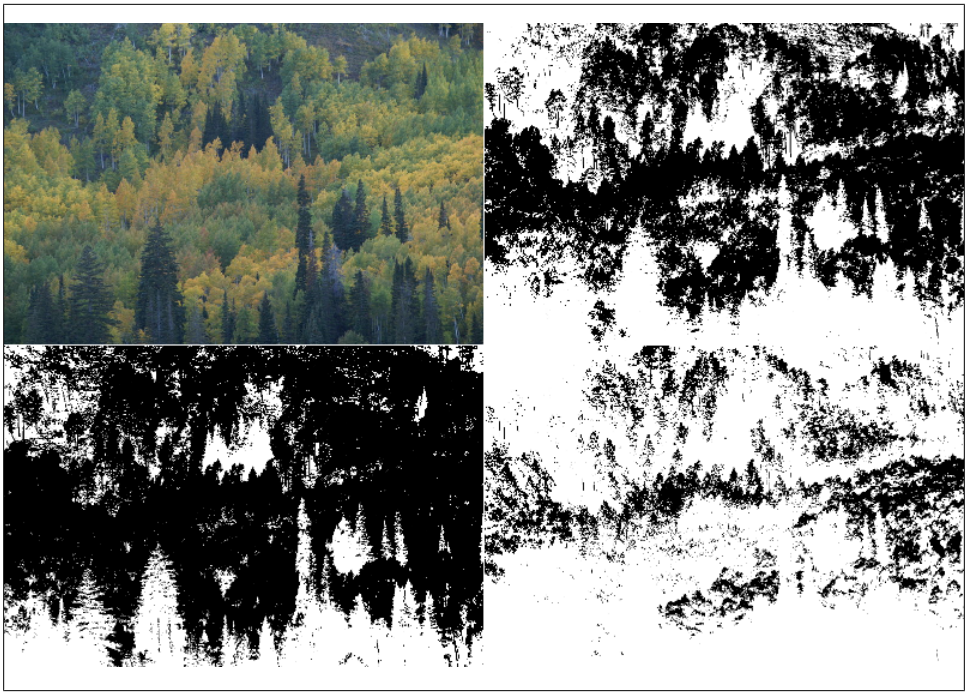


Figure 4-21. Top Left: Original image. Top Right: Binarized with Otsu's method. Bottom Left: Low threshold value. Bottom Right: High threshold value.

```
top  = img.sideBySide(otsu)
bottom = low.sideBySide(high)
combined = top.sideBySide(bottom, side="bottom")

combined.show()
```

Dilation and Erosion

Once the image is converted into a binary format, there are four common morphological operations: dilation, erosion, opening, and closing. Dilation and erosion are conceptually similar. With dilation, any background pixels (black) that are touching an object pixel (white) are turned into a white object pixel. This has the effect of making objects bigger and merging adjacent objects together. Erosion does the opposite. Any foreground pixels (white) that are touching a background pixel (black) are converted into a black background pixel. This makes the object smaller, potentially breaking large objects into smaller ones.

For the examples in this section, consider the case of a pegboard with tools. The small holes in pegboard can confuse feature detection algorithms. Tricks like morphology can help clean up the image. The first example shows dilating the image. In particular,

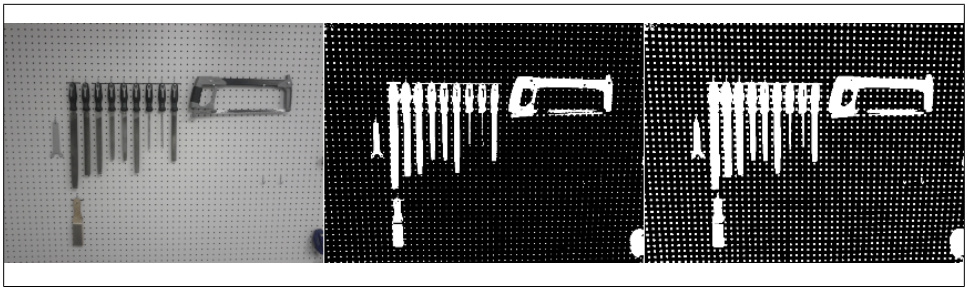


Figure 4-22. Left: The original image. Center: Binarized image. Right: Dilated image.

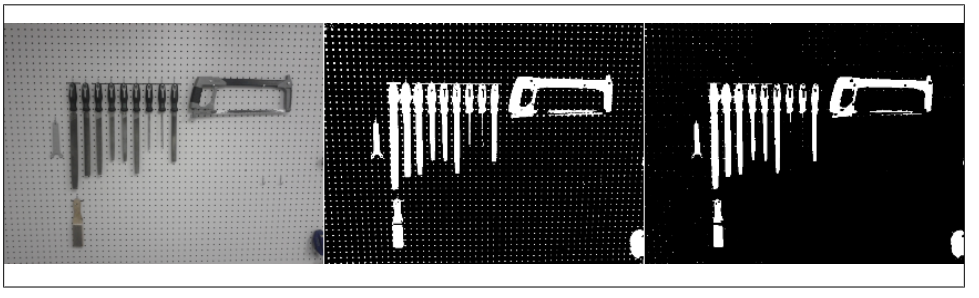


Figure 4-23. Left: the original iamge. Center: Binarized image. Right: Eroded image.

notice that after binarizing, some of the parts of the tools have disappeared where there was glare. To try to get these back, use dilation to fill in some of the missing parts.

```
from SimpleCV import Image

img = Image('pegboard.png')

# Binarize the image so that it's a black and white image
imgBin = img.binarize()

# Show the effects of dilate() on the image
imgBin.dilate().show()
```

Notice that although the this filled in some of the gaps in the tools, the pegboard holes grew. This is the opposite of the desired effect. To get rid of the holes, use the `erode()` function.

```
from SimpleCV import Image

img = Image('pegboard.png')

imgBin = img.binarize()

# Like the previous example, but erode()
imgBin.erode().show()
```

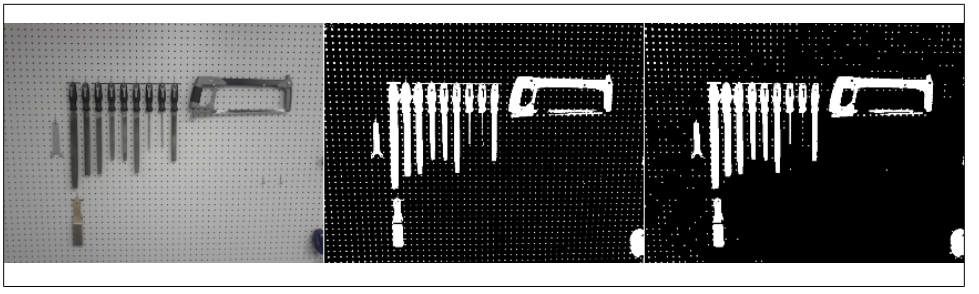


Figure 4-24. Left: The original Image. Center: Binarized Image. Right: Image after `morphOpen()`.

This has essentially the opposite effect. It made a few of the gaps in the image worse, such as with the saw blade. On the other hand, it eliminated most of the holes on the peg board.

While the `dilate()` function helps to fill in the gaps, it also amplified some of the noise. In contrast, the `erode()` function eliminated a bunch of noise, but at the cost of some good data. The solution is to combine these functions together. In fact, the combinations are so common, they have their own named functions: `morphOpen()` and `morphClose()`. The `morphOpen()` function erodes and then dilates the image. The erosion step will eliminate very small (noise) objects, following by a dilation which more or less restores the original size objects to where they were before the erosion. This has the effect of removing specks from the image. In contrast, `morphClose()` first dilates and then erodes the image. The dilation first fills in small gaps between objects. If those gaps were small enough, the dilation completely fills them in, so that the subsequent erosion does not re-open the hole. This has the effect of filling in small holes. In both cases, the goal is to reduce the amount of noise in the image.

For example, consider the use of `morphOpen()` on the pegboard. This will eliminate a lot of the pegboard holes while still trying to restore some of the damage to the tools created by the erosion.

```
from SimpleCV import Image

img = Image('pegboard.png')

imgBin = img.binarize()

# +morphOpen()+ erodes and then dilates the image
imgBin.morphOpen().show()
```

Although this helped a lot, it still leaves a lot of the pegs in the pegboard. Sometimes, the trick is simply to do multiple erosions followed by multiple dilations. To simplify this process, the `dilate()` and `erode()` functions each take a parameter representing the number of times to repeat the function. For instance, `dilate(5)` means to perform a dilation five times.

```
from SimpleCV import Image
```

Figure 4-25. Left: The original image. Center: Dilated image. Right: Eroded after dilation.

```
img = Image('pegboard.png')

# Dilate the image twice to fill in gaps
noPegs = img.dilate(2)

# Then erode the image twice to remove some noise
filled = noPegs.erode(2)

allThree = img.sideBySide(noPegs.sideBySide(filled))
allThree.scale(.5).show()
```

Examples

The examples in this section demonstrate both a fun application and a practical application. On the fun side, it shows how to do a spinning effect with the camera, using the `rotate()` function. On the practical side, it shows how to shear an object viewed at an angle, and then use the corrected image to perform a basic measurement.

The SpinCam

This is a very simple script which continually rotates the output of the camera. It continuously captures images from the camera. It also progressively increments the angle of rotation, making it appear as though the video feed is spinning.

```
from SimpleCV import Camera

cam = Camera()
display = Display()

# This variable saves the last rotation, and is used
# in the while loop to increment the rotation
rotate = 0

while display.isNotDone():
     rotate = rotate + 5  ❶
     c.getImage().rotate(rotate).save(display)  ❷
```

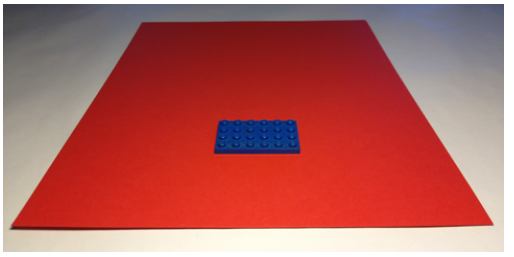


Figure 4-26. The original image of the building block on paper

❶ Increment the amount of rotation by 5 degrees. Note that when the rotation exceeds 360 degrees, it automatically loops back around.

❷ Take a new image and rotate by the amount computed in the previous step. Then display the image.

Warp and Measurement

The second example is slightly more practical. Measuring objects is covered in more detail later in this book, but this example provides a general introduction. The basic idea is to compare the object being measured to an object of a known size. For example, if an object is sitting on top of an `8.5 x 11` inch piece of paper, the relative size of the objects can be used to compute the size. However, this is complicated if the paper is not square to the camera. This example shows how to fix that with the `warp()` function. The image in Figure 4-26 is used to measure the size of the small building block on the piece of paper.

```
from SimpleCV import Image

img = Image('skew.png')

# Warp the picture to straighten the paper
corners = [(0, 0), (480, 0), (336, 237), (147, 237)]  ❶

warped = img.warp(corners)  ❷

# Find the blob that represent the paper
bgcolor = warped.getPixel(240, 115)

dist = warped.colorDistance(bgcolor)  ❸

blobs = dist.invert().findBlobs()  ❹

paper = blobs[-1].crop()  ❺

# Find the blob that represent the toy
toyBlobs = paper.invert().findBlobs()

toy = toyBlobs[-1].crop()  ❻
```

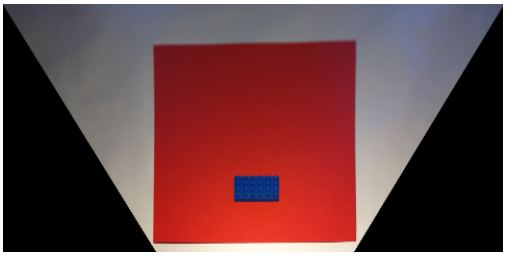


Figure 4-27. The image warped to square the sides of the paper

```
# Use the eraser/paper ration to compute the size
paperSize = paper.width
toySize = toy.width

print float(toySize) / float(paperSize) * 8.5  ❼
```

❶ These are the coordiates for the four corners of the paper. A good way to help identify the corner points is to use the SimpleCV Shell to load the image, and then use the `image.live()` function to display it. Then you can left-click on the displayed image to find the coordinates of the paper corners.

❷ This warps the image to square the edges of the piece of paper, as shown in Figure 4-27.

❸ Use the `image.live()` trick to also find the color of the paper. This will make it easier to find which part of the image is the paper versus other background objects. The image below shows the result. Notice that the paper is black whereas the rest of the image is represented in various shades of grade, as shown in Figure 4-28.

❹ By making the paper black, it is easier to pull it out of the image with the `findBlobs()` function.

❺ Now crop the original image down to just the largest blob (the paper), as represented by `blobs[-1]`. This creates a new image that is just the paper.

❻ Now looking at just the area of the paper, use the findBlobs function again to locate the eraser. Create an image of the eraser by cropping it from off the paper.

❼ Using the ratio of the width of the paper and the eraser, combined with the fact that the paper is 8.5 inches wide, compute the size of the eraser, which is 1.87435897436, which matches the objects size of 1.875 inches.

Note that this example works best when measuring relatively flat objects. When adjusting the skew, it makes the top of the object appear larger than the bottom, which can result in an over-reported size. Later chapters will work with blobs in greater depth and discuss different blob properties to get a more accurate measurement.

Figure 4-28. Color distance from the red background, making it easier to extract the paper and eraser

CHAPTER 5

The Impact of Light

All vision systems depend on quality images, and quality images in turn depend on light. Because of this, the quality of the light in your vision system environment is a key factor to it's success. This chapter will take deeper look at light, and how to use it to illuminate your vision system. This includes:

- The different types of light sources available
- Ways to evaluate light sources
- Looking at how the target object interacts with light
- Removing unwanted ambient light
- Reviewing different lighting techniques
- Calibrating the camera
- Using color to segment an image

Introduction

One of the most common mistakes of beginning computer vision developers is to overlook lighting and its effect on image quality and algorithm performance. Lighting is a critical component to any vision system, and can be the difference between success and failure. After all, without lighting, computer vision would be the study of black rooms with black objects. That would actually make vision programming incredibly easy, but not terribly useful. Instead the lighting should help you accomplish three main goals:

- Maximize the contrast of the features you're interested in analyzing.
- Be generalized enough that it works well from one object to the next.
- Be stable within the environment that you're viewing, particularly over time.

You want to keep in mind that in any environment, light radiates from one or more sources, and then bounces onto an object (or irradiates it). When filming the object, that surface then radiates the incident light into the camera. It is important to understand this process at an abstract level as it underlies all illumination situations, and

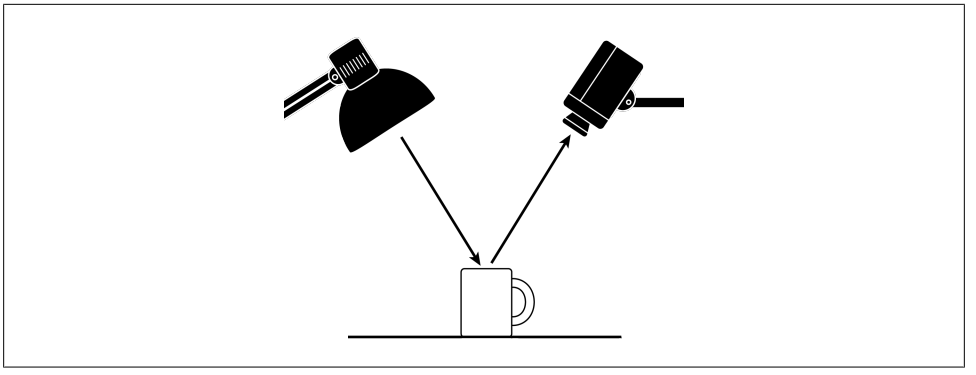


Figure 5-1. Light, Camera, Action

affects the proposed solutions. A camera doesn't film the object itself; it films the light reflected from the object.

With lighting, there are three general factors that you have to take in to consideration: the source of your light, how the objects you are filming reflect or absorb that light, and how your camera then absorbs and processes the light. With your light sources, you need to take into account things like the color of the light, the position of the light source in relationship to your target object, and how much of an impact any ambient light might be having on your setup. With the objects, things like the geometry and surface of the object have an impact, as well as it's composition and color. If the object's surface is very reflective, that's going to require a different lighting setup then an object whose surface absorbs light instead. Finally, with the camera, you need to consider what the camera is capable of, as well as the best settings to use. At the end of the day, if the sensor in your camera can't utilize the light from your light source, then it's as if the light wasn't even there in the first place.

Light and the Environment

If possible, it is easier to control the light sources in your environment than it is to write code to compensate for poor lighting. This first section provides some background on the environment, lighting, and other factors that influence the effectiveness of a vision system. It does not involve much SimpleCV code, but understanding the environment makes future coding easier. Of course, in some situations, it is not possible to carefully control the environment. For example, outdoor lighting is heavily subject to the weather and time of day. Given these challenges, the next section of this chapter covers how to use SimpleCV to compensate for different lighting situations. But first, some background on creating an environment conducive to machine vision.

Ideally, with the environment, you want to create a situation where the lighting is as consistent and controlled as possible. If there is a lot of light contamination from external light sources, you might want to create an enclosure to block that light. Or you

might want to look at the nature of that ambient light, and possibly use a light filter to minimize the impact. You also want to look at how the objects themselves will be presented to the camera. Will they be in a consistent location? If they're not going to be in the same location, then a spotlight probably isn't the best lighting choice. Will the objects be moving? If they are moving—such as on a conveyor belt—a strobe light might be what you need to capture the most relevant information. Can you add a light source or change a light's orientation to increase it's effectiveness? Any changes you can make to the environment to increase the consistency when filming one object to the next, the more successful you system will likely be.

Light Sources

Outside of the environment, there is the source of the light itself. The types of light sources frequently used in vision systems include:

- Fluorescent
- LED
- Quartz Halogen
- Xenon
- Metal Halide
- High Pressure Sodium

When picking a light source, the things to consider are how consistent and reliable they are; what the life expectancy is like for the bulbs and how cost effective they might be; how stable they are and what the spectrum and intensity is like for the light they emit; and how flexible they are and whether or not you can easily adapt them for different situations. Most small to medium sized machine vision applications use either fluorescent, LED, or Quartz Halogen light sources. Fluorescent lighting tends to be very stable, but it doesn't have as long a life expectancy as other sources. In comparison, LED lighting tends to be stable, adaptable, and has a long life expectancy, but it doesn't emit as intense a light as a Quartz Halogen source. Quartz Halogen lights also output a lot of heat, though, with that intense light, and don't have a particularly strong life expectancy. Of course, you're not limited to one type of light source either. You can combine sources to meet whatever your requirements are.

When evaluatiing light sources, they are generally classified in terms of:

- Brightness, e.g. a 40 watt light bulb versus a 100 watt light bulb
- Color, e.g. red and green Christmas lights
- Location, such as overhead lights versus track lighting
- Flavor, e.g., cloudy days have diffuse lighting, which contrasts to the point source lighting of a sunny day.

Most consumer light bulbs use incandescent wattage as an approximation of their brightness, but it is not a very useful measurement. The wattage of a light bulb is the amount of radiant flux, or the total amount of electromagnetic radiation, emitted by the bulb. Electromagnetic radiation includes everything in the infrared, ultraviolet, and visible light spectrum - which also includes thermal radiation, or heat. With the exception of their application for the Easy Bake Oven, the purpose of a lightbulb is to emit visible light and not heat. A better way to measure the power of a light is in **lumens** (denoted by the symbol lm). Lumens is the unit of measurement for luminous flux, which is the total amount of visible light emitted by a source. Since most vision systems are only concerned with visible light, it's therefore more useful to use lumens and not watts when determining the lighting specifications.

Another unit of light measurement that you might come across is candlepower, which is often used when rating LEDs. Candlepower is expressed in candelas (cd) or millicandelas (mcd), and is a measurement of the intensity of a light source in a given direction. The relationship between lumens and candelas is that 1 candela is equal to 1 lumen per steradian (a steridian being a unit of measurement related to the surface area of a sphere). It is because candelas take into account the directionality of the light that they are useful. If you decrease the beam angle of an LED to give it a tighter focus, you increase the brightness without actually having to increase the amount of light it emits. In other words, a 1000 mcd LED with a viewing angle of 60 degrees outputs as much total light as a 4000 mcd LED with a viewing angle of 30 degrees - but the 4000 mcd LED will be 4 times as intense. It's a difference of a factor of 4 because when you cut the angle in half, you are cutting it in two directions for both the width and the height.

The take-away being that with spotlights, you should consider the candelas or millicandelas ratings of your light sources. With more general lighting, or floodlights, the lumens rating is likely more useful.

Frequently, there will be sources of ambient light in your environment - such as a window that lets in natural light, or overhead fluorescent lights in the room. Ambient light is a problem for computer vision systems because the light tends to be inconsistent, and introduces enough variation to cause erratic results. You therefore want to limit the amount of ambient light with one or more of the following techniques. You can physically block the ambient light by using either an enclosure or a shroud around your vision system. In this case, you want to use dark materials that will absorb light, such as a heavy, black curtain. You can also minimize the impact of ambient light by using filters on your camera, so it only records light waves in a particular narrow band. You can also overwhelm the ambient light by flooding the environment with a strong, stable light source. Strobe lights are often used in this manner in vision systems. Finally, you can also buy cameras that were specifically designed to reject ambient light as well.

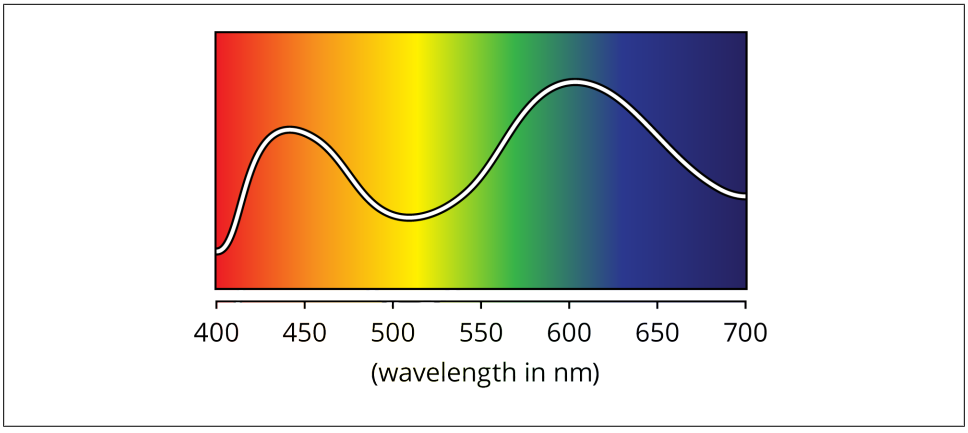


Figure 5-2. Histogram superimposed on the visible light spectrum

Light and Color

The color of a light source is another important consideration when selecting a lighting source. Scientifically speaking, visible light is the part of the electromagnetic spectrum that ranges from low frequency (long wavelength) red light, to high frequency (short wavelength) violet light. We use the term "Visible Light" to talk about the range of the spectrum that humans can see. Different illumination systems generate different colors of light. For example, sunlight at noon is different than a white LED light, which is different than the light of a laser pointer pen. When comparing sources, it is often useful to draw a graph with the rainbow of colors on the x-axis and the amount of light at each color plotted on the y-axis. See, for example, Figure 1 below.

One reason that the color of the light is important is that different surfaces respond differently to various colors of light. One example of this effect is an ordinary white t-shirt. When viewing a white t-shirt under most lighting conditions, it appears more or less white. When using a black light, the same shirt can appear to be glowing violet. For any situation where you have an object illuminated by light (regardless of it's source), this same effect is in play. This effect can have a significant impact when using color to identify objects and segment images. If the wrong color light changes the apparent color of the image, the code will fail.

Sometimes the color balance of a picture is a bit off, requiring some degree of correction. This is done with a `ColorCurve`. The color is most simply thought of as a graph where the x-axis is the intensity of the color in the original image and the y-axis is the new intensity. For example, curve for the red channel goes through the point `(100, 120)`, then any pixel that had a red value of 100 in the original image will have a value of 120 in the new image. Obviously defining these new values for all 256 possible red values, plus all 256 green values, plus all 256 possible blue values would be a time consuming mess. Instead, `ColorCurve+s are defined with a couple points and the rest are interpolated. For example, a curve for the red channel defined by the points`

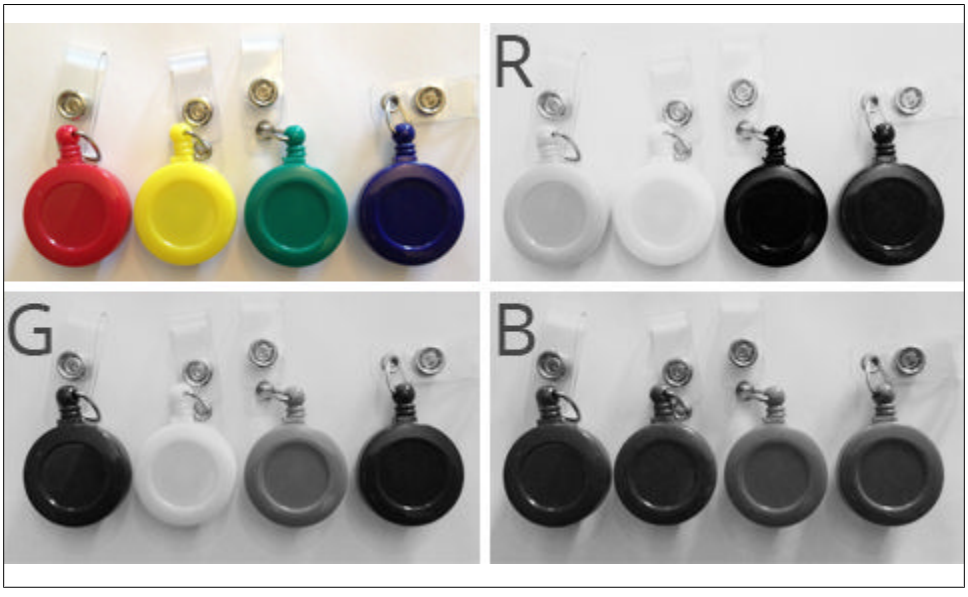


Figure 5-3. Using different colored lights creates different contrasts.

`+(0, 0)`, `(128, 128)`, and `(256, 128)` will leave all the low and middle intensity reds untouched but it will reduce the high intensity reds.

To apply a color curve, first create the `ColorCurve` for each color channel. When working with RGB color, the result is then applied with the `applyRGBCurve()` function. The function takes three arguments: a curve for the R channel, a curve for the G channel, and a curve for the B channel. Curves can also be applied to HSV images with the `applyHSVCurve()` function. It once again takes three arguments of the three curves representing the H, S, and V channels.

The following example demonstrates how to use color curves to apply an old time photo effect to images from the web cam.

```
from SimpleCV import Camera, Display, ColorCurve, Image

screenSize = (640, 480)

rCurve = ColorCurve([[0,0],[64,64],[128,128],[256,128]])  ❶

gbCurve = ColorCurve([[0,16],[64,72],[128,148],[256,256]])  ❷

cam = Camera(-1, {'width': screenSize[0], 'height': screenSize[1]} )

disp = Display(screenSize)

while not disp.isDone():
     img = cam.getImage()
```

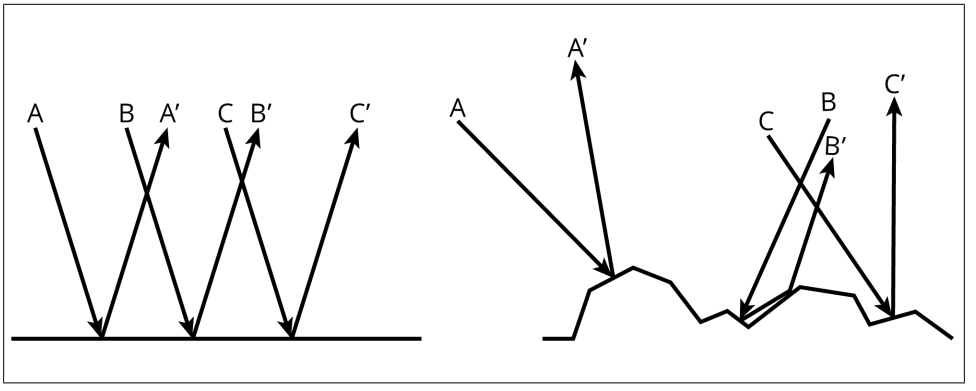


Figure 5-4. How a rough surface diffuses light.

```
coloredImg = img.applyRGBCurve(rCurve, gbCurve, gbCurve)  ❸

erodedImg = coloredImg.erode(1)  ❹

erodedImg.save(disp)
```

❶ The first curve will be used for the red channel. It reduces the high intensity red colors.

❷ The second curve will be used for the green and blue channels. It provides a slight boost to the mid level green and blue channels.

❸ Apply the curve to the image. Note that `rCurve` is used for just the red channel, whereas `gbCurve` is passed both for the green and blue channels.

❹ Adding an erode to the image provides a little additional old-time photo look to the image.

The Target Object

While the brightness and color refer to the light source, the nature of the object itself also affects how light interacts with it. When light reflects off of an object, it always follows the law of reflection, which states that the angle at which the light approached the object (the incident ray) is the same angle at which the light will leave the object (the reflected ray). When an object's surface is completely smooth, like in the case of a mirror, then all of the incoming light will be reflected uniformly away from the object. This is known as a specular reflection, and it will make the object seem shiny when you look at it because you'll be seeing all of the reflected light. However, if the surface of the object isn't smooth, such as with a piece of paper, then the incoming light hits the varied surface at different angles - and since it still obeys the law of reflection, the reflected light then leaves the object using those same angles. Since the light rays aren't leaving the object in a uniform matter, the light is scattered and the object will appear to have more of a dull finish. This is known as a diffused reflection.

Figure 5-5. The Terracotta Army, a lambertian surface.

The nature of an object, or what it consists of, impacts more then just how smooth it's surface is. Some materials absorb light and some transmit the light and let it shine right through. Some materials are fluorescent, and when they absorb light at one wavelength, may emit light at a different wavelength. Then there's also the geometry of the object, since a curved surface is going to reflect light differently than a flat one. One of the easiest ways to deal with all of these considerations is a simple trial and error process where you test how various light sources iteract with a sample object. Since the lighting can have such a dramatic impact on the image quality, it's a good idea to do this early in the process of developing a vision system.

The following terms are sometimes used when describing the surface of an object:

- Lambertian: Normal or matte surfaces. Light reflects predictably off the object, based only on the position of the light source. Examples include terra cotta, unfinished wood, paper, and fabric. These types of objects are among the easiest to capture with computer vision, and are the most robust under different lighting considerations. An example image is shown in Figure 5-5.

- Sub-surface scattering: Light penetrates the objects, interacts with the material, and exits at another point. Examples include milk, skin, bone, shells, wax, and marble. The position of the source of light can sometimes have unpredictable results, making it important to plan lightning carefully, and have consistent lighting to ensure high quality results. An example is shown in Figure 5-6.

- Specular: Shiny objects, such as polished metals, glass, and mirrors. These objects are difficult to use in computer vision systems because their surfaces may include reflections of other objects from their surroundings. When dealing with smooth specular surfaces, you may want to have lighting in a specific pattern and analyze the reflection, rather than the object itself. An example is demonstrated in Figure 5-7.

Figure 5-6. Sub-surface scattering through wax

Figure 5-7. Glass sculptures, a typo of specular surface

- Albedo: A measure of of the percentage of light reflected by an object. Albedo is measured from zero to one, with one meaning that 100% of the light directed on to the object is then reflected. Objects with a higher albedo look more white. Objects with a lower albedo appear darker, as they absorb most of the light that hits them. This determines the quantity of light that you would use in an application.

Besides strength and color, light is also classified according to "point-source", "diffuse", and "ambient" light. A point source light is basically a light bulb or the Sun. A diffuse

Figure 5-8. The albedo effect

source is light that has been diffused though another object, such as clouds or a diffuser attached to a camera flash. Ambient lighting is a catch-all term for light that has bounced off multiple objects before the object of interest.

Lighting Techniques

The final area to consider is the lighting techniques you choose to use. This is a bit beyond the scope of this book, but here is a quick outline of some of the more popular techniques:

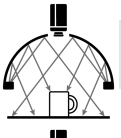
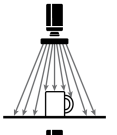
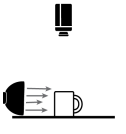
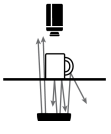
Technique	Lighting Type & Direction	Advantages	Disadvantages	
Diffuse Dome Lighting	Diffused light source, placed in front of the object	Effective at lighting curved, specular surfaces	Usually requires close proximity to the object	
Diffuse On Axis Lighting	Diffused light source, placed in front of the object	Effective at lighting flat, specular surfaces	Usually requires close proximity to the object	
Bright Field Lighting	A point light source, placed in front of the object	The most commonly used lighting technique. It's good for enhancing topographical details.	With specular or curved surfaces, it can create strong reflections	
Dark Field Lighting	A point light source, placed at	Good for finding surface imperfections	Does not illuminate flat, smooth surfaces	

Figure 5-9. The Starry Night, used in examples below

Technique	Lighting Type & Direction	Advantages	Disadvantages	Example
	the side of the object			
Diffuse Back-lighting	A diffuse light source, placed behind the object	Creates a high-contrast silhouette of an object; useful for finding the presence of holes or gaps	The edges of the silhouette may be fuzzy	
Collimated Backlighting	A point light source, placed behind the object	Creates sharp edges on a silhouette, so good for measuring the overall dimensions of an object	Not good for recording topographical details	

Color

In addition to the illumination, it is also important to understand the color of the image. Although color sounds like a relatively straightforward concept, different representations of color are useful in different contexts. The following examples work with an image of The Starry Night by van Gogh, as shown in Figure 5-9.

In the SimpleCV framework, the colors of an individual pixel are extracted with the `getPixel()` function. This was previously demonstrated in Chapter 4:

```
from SimpleCV import Image

img = Image('starry_night.png')

print img.getPixel(0, 0)  ❶
```

❶ Prints the RGB triplet for the pixel at (0,0), which will equal `(71.0, 65.0, 54.0)`.

One criticism of RGB is that it does not specifically model luminance. Yet the luminance/brightness is one of the most common properties to manipulate. In theory, the luminance is the relationship of the of R, G, and B values. Ie practice, however, it is sometimes more convenient to separate the color values from the luminance values. For example, the difference between a bright yellow and a dark yellows is non-intuitively controlled by the amount of blue. The solution is HSV, which stands for Hue, Saturation, and Value. The color is defined according to the Hue and Saturation, while Value is the measure of the luminance/brightness. The HSV color space is essentially just a transformation of the RGB color space, since all colors in the RGB space have a corresponding unique color in the HSV space, and vice versa. It is easy to convert images between the RGB and HSV color spaces, as is demonstrated below.

```
from SimpleCV import Image

img = Image('starry_night.png')

hsv = img.toHSV()  ❶

print hsv.getPixel(25,25)  ❷

rgb = hsv.toRGB()  ❸

print rgb.getPixel(25,25)  ❹
```

❶ This converts the image from the original RGB to HSV.

❷ In this first print statement, since the image was converted to HSV, it will print the HSV values for the pixel at (25,25). In this case, those are `(117.0, 178.0, 70.0)`.

❸ This line converts the image back to RGB.

❹ This will now print the RGB triplet `(21.0, 26.0, 70.0)`.

The HSV color space is particularly useful when dealing with an object that has a lot of specular highlights or reflections. In the HSV color space, specular reflections will have a high luminance value (V) and a lower saturation (S) component. The hue (H) component may get noisy depending on how bright the reflection is, but a object of solid color will have largely the same hue even under variable lighting. We'll look at hue segmentation further in Chapter 8.

Grayscale is the final color encoding scheme commonly used in programs developed with the SimpleCV framework. A grayscale image represents the luminance of the image, but lacks any color components. It is often referred to as a black-and-white image, though it is important to understand the difference between a grayscale and a binary

Figure 5-10. The Starry Night, converted to Grayscale

black-and-white image. In the later case, there are only two values: 0 and 1 for pure black and pure white, respectively. In contrast, an 8-bit grayscale image has many shades of grey, usually on a scale from 0 to 255. The challenge is to create a single value from 0 to 255 out of the three values of red, green, and blue found in an RGB image. There is no single scheme for doing this, but it is done by taking a weighted average of the three. To create a grayscale image:

```
from SimpleCV import Image

img = Image('starry_night.png')

gray = img.grayscale()  ❶

print gray.getPixel(0,0) ❷
```

❶ This will convert the image to a grayscale image. The result in shown in Figure 5-10.

❷ This will print the grayscale value for the pixel at (0,0), with the result of `(66.0, 66.0, 66.0)`.

Notice that it returns the same number three times. This keeps a consistent format with RGB and HSV, which both return three values. However, since grayscale only has one value, representing the luminance, the same value is repeated three times. You can also get the grayscale value for a particular pixel, without having to convert the image to grayscale, by using `getGrayPixel()`.

Color and Segmentation

Chapter 3 introduced the concept of segmentation, which is the process of dividing an image into areas of related content. These areas consist of pixels that all share a particular characteristic, and one of the more frequently used characteristics is color. It is easy to use color to segment an image. This technique is very effective when a color of the desired object is substantially different from the background color; such as a case where you want to track a brightly colored object such as a ball. In this case, you can use the color difference to segment the image and remove the background from the image, leaving just the object of interest.

This works by essentially subtracting one image from another. To understand this, first consider how subtraction works with pixels (a topic which is covered more extensively in the next chapter). Assume that the pixel at point `(0, 0)` is purple, with the RGB triple `(100, 0, 100)`. Take an identical pixel--`(100, 0, 100)`--and subtract it from the original pixel. To do this, simply subtract each element from it's corresponding value. `(100, 0, 100) - (100, 0, 100) = (0, 0, 0)`. Since `(0, 0, 0)` is the RGB value for black, what you find is that subtracting the same RGB value from a pixel results in a black pixel. You can also subtract different RGB values too. For example, `(100, 0, 100) - (90, 0, 10) = (10, 0, 90)`, which results in a mostly blue pixel. Subtracting images is just like subtracting pixels, with the system going through the image on a pixel-by-pixel basis and performing the subtraction for each pixel.

With color segmentation, you want to subtract away the pixels that are far away from the target color, while preserving the pixels that are similar to the color. This requires measuring all of the colors involved to gauge how far away they are from the target color. The Image class has a function called `colorDistance()` that lets you do this easily. This function takes as an argument the RGB value of the target color, and it returns another image representing the distance from the specified color. This is perhaps easier to understand by looking at an example, working with a picture of a yellow glue gun, as demonstrated in Figure 5-11.

```
from SimpleCV import Image, Color

yellowTool = Image("yellowtool.png")

yellowDist = yellowTool.colorDistance((223, 191, 29))  ❶

yellowDistBin = yellowDist.binarize(50).invert()  ❷

yellowDistBin.show()
```

❶ The first step is to find the RGB values for your target color. In this example, the RGB triplet of `100, 75, 125` is the approximate value for the purple in the star. Passing this RGB value into the `colorDistance` function will have the function return a grayscale image, where colors close to purple are black and colors far away from purple are white.

Figure 5-11. Original image and the grascale image showing the color distance.

Figure 5-12. The yellow color distance

❷ Some of the pixels in the background are purple as well. Since we're not interested in these pixels, we filter some of them out with the `binarize()` function. If you recall, the binarize function turns the grayscale image into a strictly black and white one. Since we're passing it a threshold value of 50, `binarize` will turn any pixel with a grayscale value under 50 to white, while all other pixels will be turned black.

The resulting image should look like Figure 5-12:

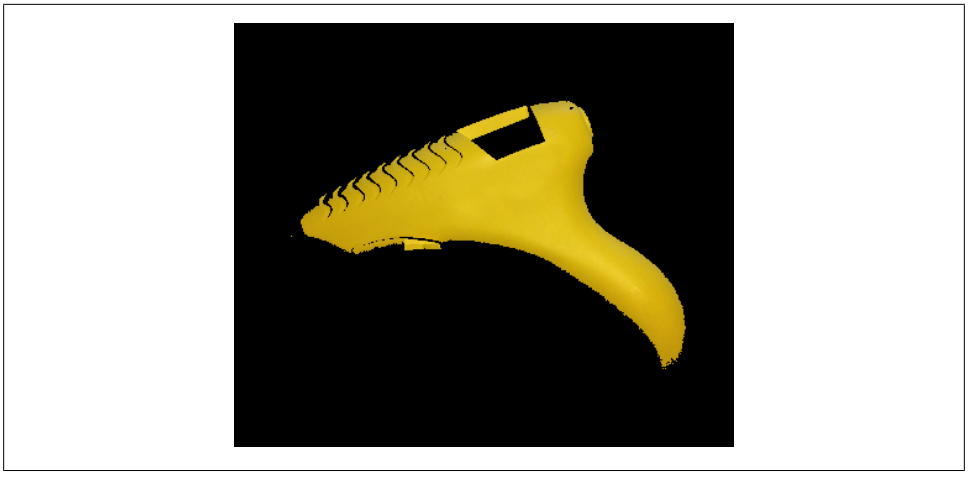


Figure 5-13. Only the yellow body of the tool remains

There is still a little noise in the image. You can use functions like `erode()` and `morphOpen()`, as covered in the previous chapter, to clean up the noise.

Now the distance image can be subtracted from the original image to remove any portions of the image which aren't purple.

```
from SimpleCV import Image, Color

yellowTool = Image("yellowtool.png")

yellowDist = yellowTool.colorDistance((223, 191, 29))

yellowDistBin = yellowDist.binarize(50).invert()

onlyYellow = yellowTool - yellowDistBin

onlyYellow.show()
```

This will result in an image with only the yellow body of the tool and everything else blacked out, such as in Figure 5-13.

Example

Let's walk through an example where we detect if a car is illegally parked in a handicap parking space. At the Ingenuitas offices, there is a repeat offender who drives a yellow car, parks illegally, and doesn't have a handicap sticker. Here is what the image looks like without the car in the spot:

Here is what the image looks like with the offending car in the spot:

A simple test would be to simply look for yellow in the image. However, if the yellow car is parked adjacent to the handicap spot, then there is no violation. Instead, this "yellow detector" vision system will have to look whether yellow appears in a particular area in the image.

First, load the images of the car:

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")  ❶

car_not_in_lot = Image("parking-no-car.png")  ❷
```

❶ Loads the image of the yellow car in the parking space.

❷ Loads the image of the empty parking space.

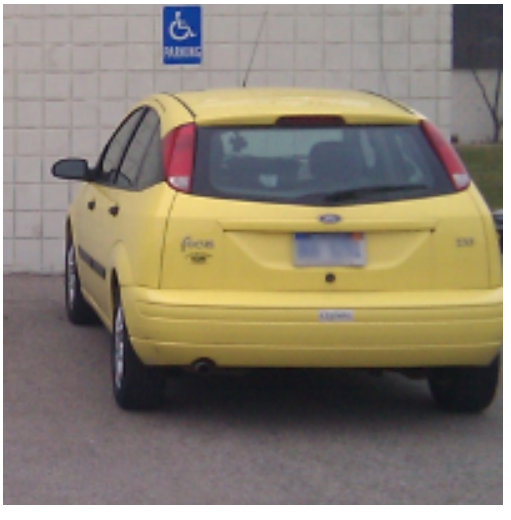
The next step is to use the picture of the car in the spot to determine the area to inspect. Since the original image contains both acceptable and illegal parking spaces, it needs to be cropped to cover only the handicap space. The whole image is `800 x 600` pixels.

The location of the handicap space is the box around the car, sometimes referred to as the Region of Interest (ROI). In this case, the ROI starts at `(470, 200)` and is about `200 x 200` pixels.

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)  ❶

# Show the results
car.show()
```

❶ Crops the image to just the area around the car in the parking space.

The resulting picture should look like:



Now that the image is narrowed down to only the handicap spot, the next step is to find the car in the image. The general approach is similar to the purple star example done earlier. First, find the pixels that are near yellow:

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)  ❶
```

```
# Show the results
yellow_car.show()
```

❶ This will return a grayscale image showing how far away from yellow all of the colors are in the image.

The resulting image should look like:

With the color distances computed, subtract out the other colors, leaving only yellow components. This should result in just the car, subtracting out the rest of the image.

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)

only_car = car - yellow_car  ❶

# Show the results
only_car.show()
```

❶ Subtracts the grayscale image from the cropped image to get an image of just the yellow car.

As expected, only the car remains:

To compare this to images that do not have the yellow car in them, there must be some sort of metric to represent the car. One simple way to do this is with the `meanColor()` function. As the name implies, this computes the average color for the image:

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)

only_car = car - yellow_car

print only_car.meanColor()  ❶
```

❶ This will print out what the mean color value is. The result should be: `(25.604575, 18.880775, 4.482825)`.

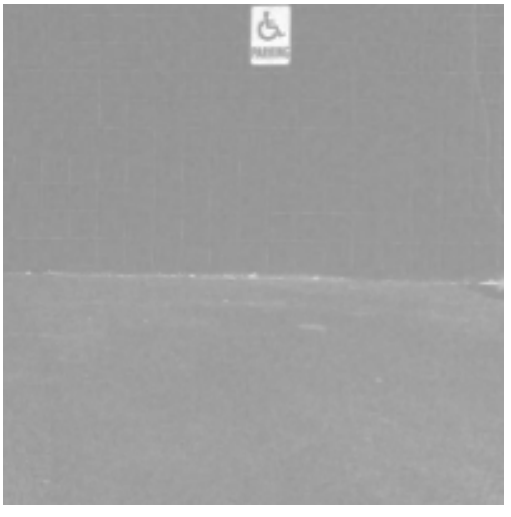
This is the metric for the space when occupied by the yellow car. Repeat the process for the empty place.

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)
```

```
only_car = car - yellow_car

no_car = car_not_in_lot.crop(470,200,200,200)

without_yellow_car = no_car.colorDistance(Color.YELLOW)  ❶

# Show the results
without_yellow_car.show()
```

❶ Returns a grayscale image showing how far away from yellow the colors are in the empty space.

Notice that this essentially creates an "empty" image.



Once again, subtract the color distance image and compute the mean color:

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)

only_car = car - yellow_car

no_car = car_not_in_lot.crop(470,200,200,200)

without_yellow_car = no_car.colorDistance(Color.YELLOW)

only_space = no_car - without_yellow_car
```

```
print only_space.meanColor()
```

The resulting mean color will be: `(5.031350000000001, 3.6336250000000003, 4.683625)`. This contrasts substantially with the mean color when the car is in the pixel, which was `(25.604575, 18.880775, 4.4940750000000005)`. The amount of blue is similar, but there is substantially more red and green when the car is in the image. This should sound right given that yellow is created by combining red and green.

Given this information, it should be relatively easy to define the thresholds for determining if the car is in the lot. For example, something like the following should do the trick:

```
from SimpleCV import Image

car_in_lot = Image("parking-car.png")

car_not_in_lot = Image("parking-no-car.png")

car = car_in_lot.crop(470,200,200,200)

yellow_car = car.colorDistance(Color.YELLOW)

only_car = car - yellow_car

(r, g, b) = only_car.meanColor()

if ((r > 15) and (g > 10):  ❶
     print "The car is in the lot.  Call the attendant."
```

❶ If the red and green values are high enough, the yellow car is probably in the parking space.

In cases where there is enough yellow—as defined by enough red and green—then it will indicate that the violating car is in the lot. If not, it will do nothing. Note that this will print the message for any yellow car in the parking space, as well as any other large, yellow object. You could refine this program by also matching the shape or car, or matching other attributes of the car.

CHAPTER 6

Image Arithmetic

As discussed in Chapters 3 and 4, we can think of images as a grid of pixels with each pixel having a color defined as an RGB triplet. Each component in that RGB triplet has a value in the range of 0 to 255. Given this structure and format, it is actually quite easy to perform arithmetic on images, such as addition, subtraction, multiplication, and division. This concept shouldn't be entirely new. Chapter 5 already introduced this idea when using color to segment images. The goal of this section is to examine this concept in further detail, and understand how mathematical manipulation of images is used in a vision system. Topics include:

- The basic mathematical operations you learned in elementary school, as applied to images
- Using histograms to calibrate the camera
- Finding the dominant colors of an image and using that information to segment the image

Basic Arithmetic

Just as with elementary school arithmetic, the easiest place to start is with addition. Two images can be added together. Underneath the hood, the SimpleCV frameowork goes through the two images pixel by pixel, and adds the RGB values of the pixels at corresponding locations together. Starting at pixel `(0, 0)` on both images (the top left corner of each image), it will add the two RGB triplets together, and that is the RGB triplet of pixel+(0, 0)+ in the resulting image. If the sum of two of the RGB components would exceed the maximum value of 255, then the value for the resulting image is capped at 255. This would be tedious work by hand, but it is a case where computer vision shines. An example is shown below:

```
from SimpleCV import Image

imgA = Image("starry_night.png")

# Add the image to itself
```

Figure 6-1. Left: The original image Right: After the image was added to itself

```
added = imgA + imgA

added.show()
```

Figure 6-1 shows the results of this addition. It looks quite similar to the original image. The most obvious difference is that the image became a lot brighter. Since the maximum value for an RGB component is 255, the "maximum" color is then the one represented by the RGB triplet `(255, 255, 255)`, which is white. This means that adding any two pixels together will always bring them closer to white, resulting in a brighter image. The pixels that were already white—or close to white—had relatively little change. On the other end of the spectrum, the dark pixels also did not change much. For example, part of the edge line around the church has a color of `(36, 31, 27)`. Doubling this only provides `(72, 62, 54)`, which is still a pretty dark color.

Adding an image to itself should conjure up thoughts of the next possible operation. Technically adding an image to itself is the same as multiplying it by two. This will have the same effect:

```
from SimpleCV import Image

imgA = Image("starry_night.png")

# Double the value of each pixel
mult = imgA * 2

mult.show()
```

Image multiplication is handy when trying to brighten an image. Although the above example is no different than simply adding the image to itself, multiplication is more flexible in an important way. An image can be multiplied by fractions, such as `mult = imgA * 1.5`. For an image that needs a lot of brightening, multiplying by larger numbers is an easy way to brighten it. Remember the images with the purple star shown in the

Figure 6-2. Left: Original image Right: Image brightened by multiplication

previous chapter? The low brightness image of the star was quite dark. This image can be easily lightened with multiplication.

```
from SimpleCV import Image

imgA = Image("starry_night.png")

# Double the value of each pixel
mult = imgA * 1.5

mult.show()
```

With the exception of regions that are pure black (`0, 0, 0`), multiplying an image by a large number will eventually result in a white image.

The resulting picture will look like Figure 6-2. Notice that unlike simply adding the image to itself, this brightened the image without blowing out the white/light portions of the image. It provides a finer degree of control over the brightening process. However, this approach is not perfect. In many photographs, brightening an image also amplifies the background noise, which results in a grainy image.

Subtraction was first introduced in Chapter 4, and then revisited again in Chapter 5. However, it can be used for more than just segmenting an image with color. In fact, the examples at the end of this book will show how important subtraction is for tricks like green screening. The first and most basic example is subtracting an image from itself:

```
from SimpleCV import Image

logo = Image("starry_night.png")

# Subtract an image from itself
```

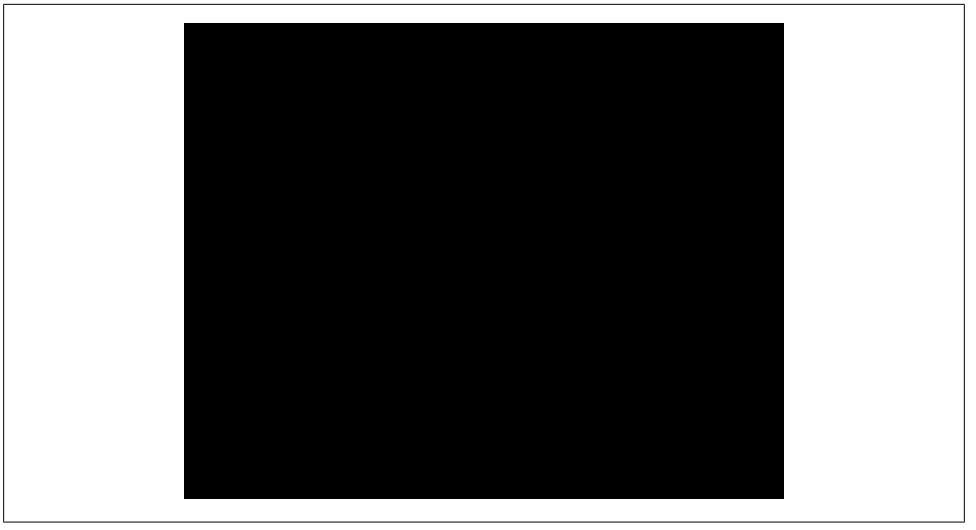


Figure 6-3. The most exciting image in the book: subtracting an image from itself.

```
# Resulting in a blank image
sub = logo - logo

sub.show()
```

When subtracting, pixels can not have an RGB component with a value less than zero. Negative values are converted to zeros.

OK, that is a really silly example. It just creates a black image, as demonstrated in Figure 6-3. After all, any number minus itself is zero. Just as in general arithmetic, subtraction is useful for finding the difference between two things. The difference between two of the exact same things is zero, which is pure black in the world of images. But it also is a very practical way to check if anything changed between two points in time. For example, assume that a hypothetical book author has been raiding the candy jar instead of writing. A simple application could detect if the candy jar has been moved between two points in time. Notice what happens in the case below:

```
from SimpleCV import Image

# Load the pictures from time A and time B
timeA = Image("candyA.png")
timeB = Image("candyB.png")

# Compare the images by subtracting
diff = timeA - timeB
```

Figure 6-4. Left: Candy jar at time 1. Middle: Candy jar at time 2. Right: Difference between images.

```
# The little marks on the image indicate the images are slightly different
diff.show()
```

If the jar was not moved, then the two images should be identical (assuming no change in ambient lighting conditions, etc.) Based on a casual look at the two images, the candy jar looks pretty similar to the two. Obviously, if someone was sneaking candy, they were pretty careful about the placement. However, when looking at the results of the subtraction in the example, there is an obvious artifact both from the jar being moved and from the candy in the jar being rustled around. This is another case where computer vision excels. People are not good at noticing very small changes over time. Computers can detect those changes easily.

The above example can be further expanded. For example, this still requires a human to watch the output and determine if the image changes. An even better system would automate the process, looking for changes, and creating a notice when the jar was moved. One possible automation approach is to check if more than 50% of the pixels changed.

```
from SimpleCV import Image

# Take the candy difference computed in the previous example
diff = Image("candydiff.png")

# Extract the individual pixels into a flat matrix
matrix = diff.getNumpy()  ❶
flat = matrix.flatten()  ❷

# Find how much changed
num_change = np.count_nonzero(flat)  ❸
percent_change = float(num_change) / float(len(flat))  ❹

if percent_change > 0.1:  ❺
     print "Stop eating candy!"
```

❶ An image is essentially just a matrix of pixel values. Therefore, to apply mathematical techniques to an image, it is sometimes easier to have it in a matrix format. The `getNumpy()` function returns a NumPy matrix of the image.

❷ The NumPy matrix is actually a three dimensional matrix with the dimensions `640 x 480 x 3`. This corresponds to the `640 x 480` image dimensions, with an additional dimension for the three values for red, green, and blue. However, rather than looping over a three dimensional array, it will be easier to flatten this into a `921,600 x 1` array.

❸ We can use Numpy (imported by default in the SimpleCV framework as "np") to quickly count the number of non-zero pixels.

❹ Compute the percentage of values that changed. Recall that `num_change` is the counter of the number of pixels that are not black, indicating a difference in the image. The `len(flat)` computes the size of the total number of pixels in the image (or more accurately, in the flattened matrix that represents the image.

❺ Finally, check if the number of pixels changed exceeds the threshold (10%). If it does, print out a warning. In this case, around 22% of the pixels changed, which triggers the message.

The `meanColor()` function introduced in the previous chapter would be another way to detect a change. If the images are the same, the mean color will be black. Otherwise, the differences will have different colors, which will impact the value of the mean color. Of course, for minor changes, the differences in the mean color will be minimal. Once again, setting an appropriate threshold for the difference in mean color values will help manage the false positive or negative alerts.

The final piece of image arithmetic to cover is division. This is useful when adjusting the contrast. This would be a good time to remember that multiplying and then dividing an image by a number does not necessarily return the image to its original state. For example:

```
from SimpleCV import Image

img = Image('starry_night.png')

mult = img * 2
div = mult / 2

div.show()
```

Elementary school mathematics might suggest that multiplying an image by two and then dividing it by two would restore the image to its original state. Yet the results demonstrated in Figure 6-5 contradict this. Notice that some of the yellows in the original image are turned into white after multiplying. Then they become a shade of gray after dividing. The reason is because of that maximum value of 255 for any given RGB component. During the multiplication process, some of the RGB values are capped at 255. When they are subsequently divided by two, they get a value of 127. For example, if a pixel starts with a red value of 166, multiplying it by two would be 332, which is capped to the maximum value of 255. Dividing that by two, then results in

Figure 6-5. Left: the original image. Center: After multiplying by 2. Right: Dividing the multiplied image.

127 instead of 166. As a result, chains of mathematical operations on images may create unexpected results. The order of operations can, therefore, be very important.

In general, any operation that can be performed on a Numpy matrix can also be done on an image. Computations like averages (means) and standard deviations are particularly useful. The example below is a basic security camera application that looks for movement or change in the image. Since it is trying to capture a nearly continuous feed of images, speed is much more important. Hence, relying on the `mean()` function in Numpy is a much more efficient approach than the previous candy jar example.

```
from SimpleCV import Camera, Display
import time

# if mean color exceeds this amount, do something
threshold = 5.0

cam = Camera()
previous = cam.getImage()

disp = Display(previous.size())

while not disp.isDone():
     # Grab another frame and compare with previous
     current = cam.getImage()
     diff = current - previous

     # Convert to NumPy matrix and compute mean color
     matrix = diff.getNumpy()
     mean = matrix.mean()

     # Show on screen
     diff.save(disp)

     # Check if changed
     if mean >= threshold:
          print "Motion Detected"

     #wait for a second
```

```
    time.sleep(1)
    previous = current
```

In this example, the threshold represents the mean color, not the percent of pixels that changed.

Histograms

Another useful tool when performing mathematical calculations on images is to use a histogram. In a non-vision sense, histograms are commonly used in statistics to plot of the values in a list. They are used to graphically identify frequently or infrequently occurring items. In the world of computer vision, histograms are typically used for listing the colors from each color channel in an image. This is sometimes known as a color channel histogram. This is helpful when manipulating color, brightness, etc. of an image.

The simplest color channel histogram is with a grayscale image. Each pixel in a grayscale image only has one value representing the brightness of that pixel (or what shade of gray it is)--as opposed to the three values representing the different color values in an RGB image. Each of these grayscale values will be in the range from 0 to 255, with 0 being black and 255 being white. To view the brightness histogram of the SimpleCV logo:

```
from SimpleCV import Image

img = Image('starry_night.png')

# Generate the histogram
histogram = img.histogram()

# Output the raw histogram data
print histogram
```

This code example outputs a list of 50 values. Each value is the number of pixels that occur at each level of brightness. But wait, why were 50 values outputted when there are 256 shades of gray? The `histogram()` function divides the dataset into bins which combine values from the x-axis of the chart. This has a tendency to smooth out the chart by grouping levels together. However, the number of bins is customizable.

```
from SimpleCV import Image

img = Image('starry_night.png')

# Generate the histogram with 256 bins, one for each color
histogram = img.histogram(256)

# Show how many elements are in the list
len(histogram)
```

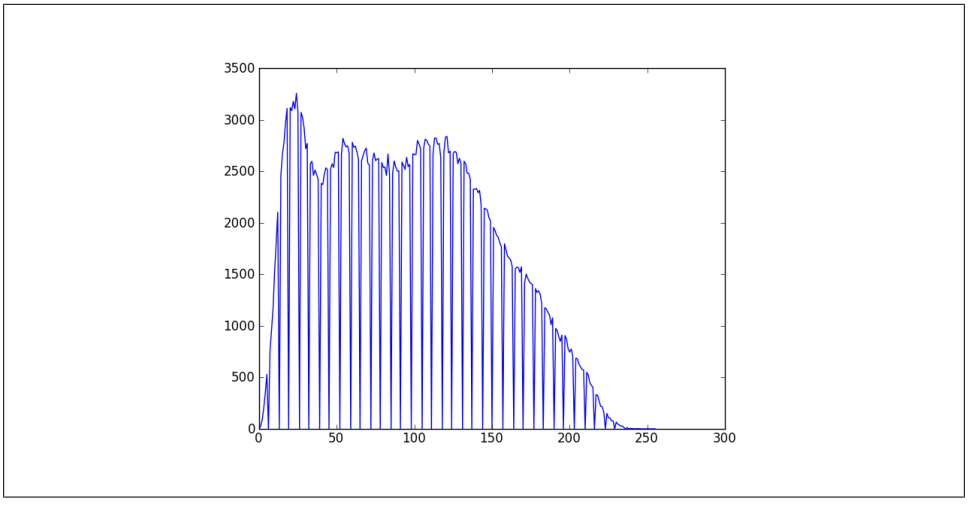


Figure 6-6. Histogram of The Starry Night

This will report 256 bins. As much fun as it is to look at lists of numbers, it is usually more intuitive to plot these in a chart. This is done with the `plot()` function.

```
from SimpleCV import Image

img = Image('starry_night.png')

# Generate the histogram with 256 bins, one for each color
histogram = img.histogram(256)

# Graphically display the histogram
plot(histogram)
```

The resulting figure should look like:

This histogram in Figure 6-6 indicates that a lot of pixels have very low values, representing dark shades. Given the original picture, this should not be surprising. A brighter picture would have a graph that was shifted further to the right. Rather than looking at the overall histogram, it is sometimes more useful to split the image into its individual color channels to look at the balance of color. This is demonstrated in the next example.

```
from SimpleCV import Image

img = Image('starry_night.png')

# Extract the three color channels
(red, green, blue) = img.splitChannels(False)  ❶

# The individual histograms
red_histogram = red.histogram(255)  ❷
green_histogram = green.histogram(255)
blue_histogram = blue.histogram(255)

# Plot the histograms
```

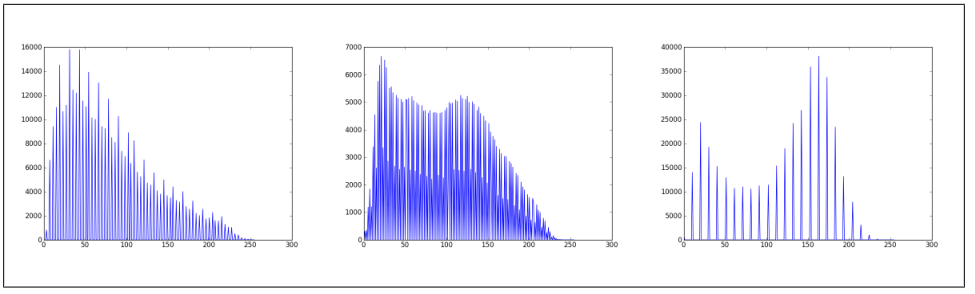


Figure 6-7. Left: Red channel histogram. Middle: Green channel histogram. Right: Blue channel histogram.

```
plot(red_histogram)  ❸
plot(green_histogram)
plot(blue_histogram)
```

❶ The `splitChannels()` function creates three images. The three images contain the red components, the green components, and the blue components, respectively. By default, this actually creates three grayscale images, showing the intensity of each individual color. By passing the parameter `False`, it keeps the original color, which is more intuitively appealing.

❷ Once the individual color channels are split apart, generate the histogram, as was done in the grayscale example. Three channels means three histograms. The three histograms are displayed below.

❸ Plot the histograms. Note that each plot command blocks until the histogram window is closed. In other words, the code will first show the red histogram. The green histogram will not be displayed until the red histogram window is closed. Then the blue histogram will not be displayed until the green window is closed.

Histograms are very important when calibrating a camera. They are used to make sure that the exposure level is correct. Consider the example image in Figure 6-8. The picture on the left is a good photo of the painting. The center image is an overexposed version of the same photo. This is evident from the washed out white areas around the stars that used to be yellow. The histogram demonstrated this problem. Notice that it has a large spike at the far right of the graph. This indicates a lot of white. A well balanced histogram tends to look like a hump, without large spikes at either the dark or light end of the spectrum, similar to what was shown in Figure 6-6.

The above example works well for the white balance of an image. True imaging aficionados should perform a similar exercise on the three color channels as well to ensure that each individual color is also in balance.

Figure 6-8. Upper Left: Regular picture. Center: Over-exposed picture. Right: Histogram for over-exposed picture.

Using Hue Peaks

After looking at the distribution of different colors in an image, the next logical question is to identify the dominant color in an image. This is known as finding the hue peaks and it is donw using the `huePeaks()` function to find the dominant color(s).

Although an image could be first converted into the HSV format and then a histogram computed to get the hue information, the SimpleCV framework has a shortcut to do this on any image format. It is the hueHistogram() function. It works just like the standard `histogram()` function.

```
from SimpleCV import Image

img = Image('monalisa.png')

# Get the histogram
histogram = img.hueHistogram()

plot(histogram)

# Get the hue peaks
peaks = img.huePeaks()

# Prints: [(12.927374301675977, 0.11747342828238252)]
print peaks
```

The output of the `huePeaks` function is an array of tuples. If there is a single dominant peak, then there is only one element in the array. The first value of the tuple is the bin that occurs most frequently. The second value is the percent of pixels that match that color. For example, in the example above, bin 13 is the most common, and appears 12% of the time. This should appear approximately correct given the histogram of the Monsa Lisa image:

This data can be used to subtract out the less frequent hues. This trick is similar to the previous examples which subtract values based on the color distance.

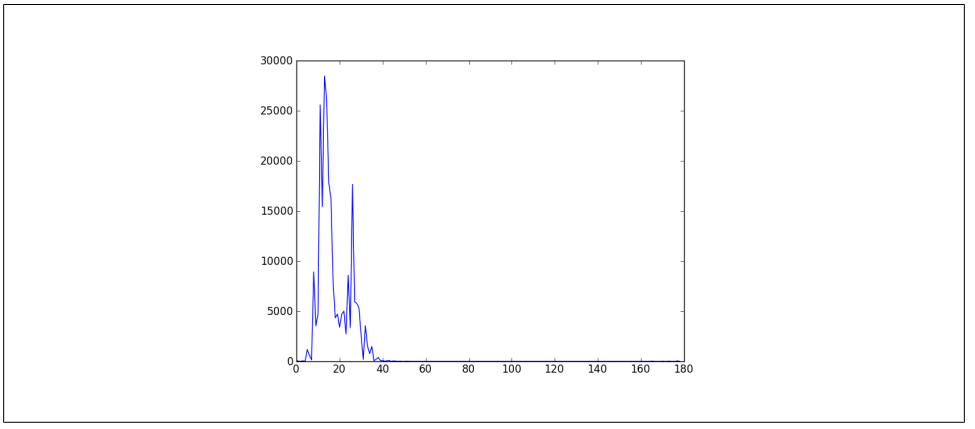


Figure 6-9. Histogram of Mona Lisa image

```
from SimpleCV import Image

img = Image('monalisa.png')

# Find the dominant hue
peaks = img.huePeaks()
peak_one = peaks[0][0]  ❶

# Get the hue distance to create a mask
hue = img.hueDistance(peak_one)  ❷
mask = hue.binarize().invert()  ❸

# Subtract out the dominant hue
lessBackground = img - mask  ❹

lessBackground.show()
```

❶ The function `huePeaks()` returns an array of tuples. The first `[0]` returns the first (and only) element of the array. The second `[0]` returns the first value of the tuple, which corresponds to the most frequently occurring hue.

❷ The `hueDistance()` function is similar to the `colorDistance()` function introduced in Chapter 4. (See earlier comments about the benefits of hue in HSV versus color in RGB.) It will return an image representing the distance of each pixel from the peak. Smaller values, which will appear black, are close to the dominant hue. Larger values, which appear white, are farther from the dominant hue.

❸ Binarize the distance image, eliminating the shades of grey. This means that the mask can either subtract out the entire pixel or does not touch it.

❹ Subtract the mast from the original image. Those parts of the mask that are white, erase corresponding pixels. Those parts that are black do not have an effect.

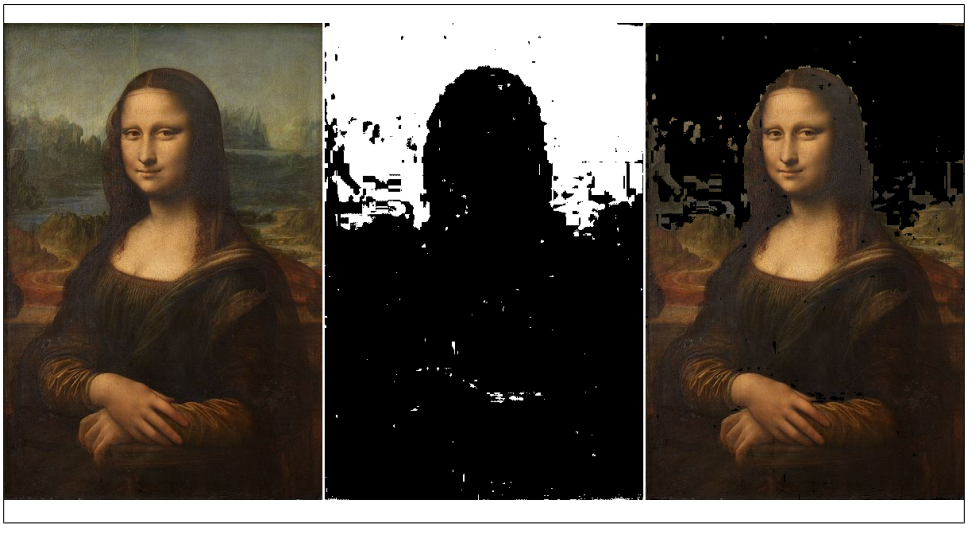


Figure 6-10. Left: Mona Lisa image. Center: Mask removing the most common hue. Right: Part of the background removed.

This still a moderate amount of noise left in the background. Tricks like the morphology functions covered in Chapter 4 can help to clean up the mask.

Binary Masking

In the previous examples we show how to make a mask by using image math. The SimpleCV framework includes a shortcut method for standard masking operations. It is a binary mask function that allows you to specify the colors to be masked out. The following example shows how to extract the face from the Mona Lisa.

```
from SimpleCV import Image

img = Image("monalisa.png")

mask = img.createBinaryMask(color1=(130,125,100),color2=Color.BLACK)  ❶

mask = mask.morphClose()  ❷

result = img - mask.invert()  ❸

result.show()
```

❶ Create a binary mask. It will mask out any colors in the range between `color1` and `color2`.

❷ Use a `morphClose()` to clean up some noise in the image.

Figure 6-11. Result of mask

❸ Apply the mask to the original image.

Examples

The examples for this chapter include two tricks for applying basic arithmetic to images. In the first example, multiple images are added together to create a motion blur effect. The second example shows how to use a green screen to put a foreground object into a different background scene.

Creating a Motion Blur Effect

One of the classic tricks in artistic photography is to use a long exposure. By leaving the camera shutter open, it creates a blur effect as objects move through the field of view. Although web cameras do not have shutters to directly replicate this effect, it can be simulated by adding together a series of images, effectively creating a motion blur.

```
from SimpleCV import Camera, Display
```

Figure 6-12. Blue example

```
frameWeight = .2  ❶

cam = Camera()

lastImage = cam.getImage()

display = Display( (cam.getProperty('width'), cam.getProperty('height')) )

while not display.isDone():
    img = cam.getImage()
    img = (img * frameWeight) + (lastImage * (1 - frameWeight))  ❷

    img.save(display)
    lastImage = img  ❸
```

❶ This controls the amount of blur. It means that 20% of the image will be the most recently captured image, with 80% coming from the previous frames.

❷ To implement the weighting, the first image is multiplied by 0.2 and the previous frame is multiplied by `1 - 0.2 = 0.8`. It is important that the sum of these weights equals 1. If less than one, the screen will grow very dark. If greater than 1, the screen will eventually become white.

❸ Finally, keep track of the image so that it can be used on the next iteration of the loop.

An example of the output is demonstrated in Figure 6-12:

The result isn't quite as smooth as a real long exposure, but it has a similar effect. Generally, the quality of this trick increases as a larger number of images are captured in a fixed period of time.

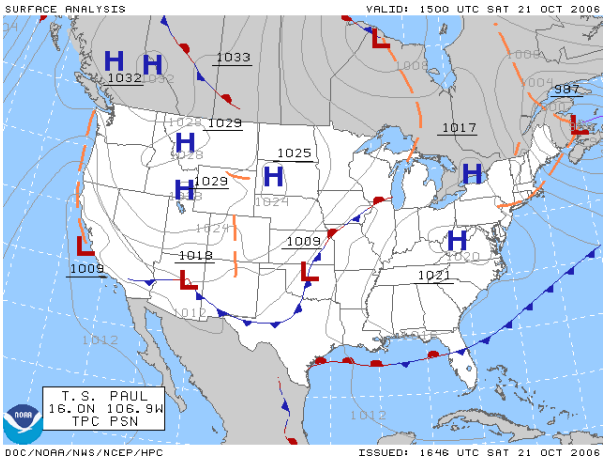


Figure 6-13. Example weather map

Chroma Key (Green Screen)

The final example for this chapter is simulating a green screen. This is a classic trick in television or movie productions. The weather reporter on the evening news typically stands in front of a green screen, and then the weather maps are painted onto the green background by a computer. It's also a classic tool used in movies to add an actor to a background or an environment without the actor actually being there. Technically, there is nothing magical about the color green in a green screen. Other colors can be used instead. However, since the computer is performing a substitution based on the particular color, it is important to make sure that the reporter or actor is not also wearing that color. Otherwise, the computer will paint the background onto the actor as well.

The following example lets you play weather man with your webcam, standing in front of a weather image such as demonstrated in Figure 6-13. Of course this example will work best with an evenly lit green screen. However, any solid colored background should generally work, though it may create some problems. The greater the color difference between the background color and the person/object in the foreground, the better this example works.

```
from SimpleCV import Camera, Color, Display, Image

cam = Camera()
background = Image('weather.png')  ❶

disp = Display()

while not disp.isDone():
     img = cam.getImage()
     img = img.flipHorizontal()  ❷

     bgcolor = img.getPixel(10, 10)  ❸
```

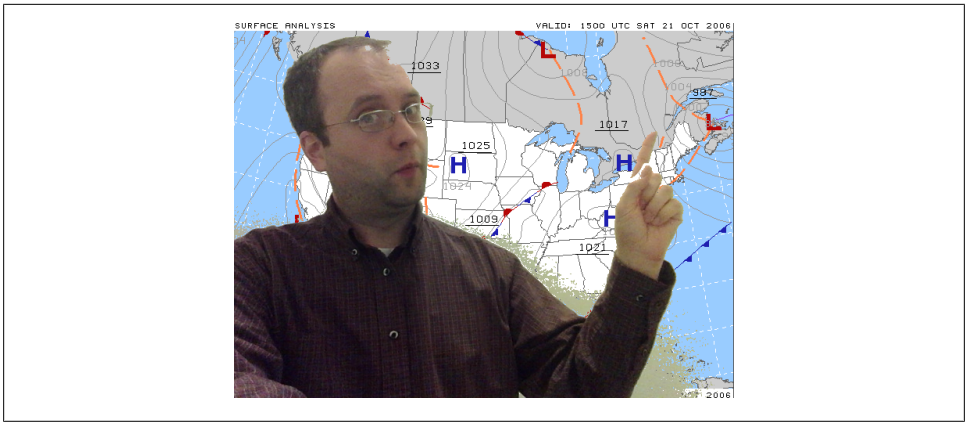


Figure 6-14. Result of green screen

```
dist = img.colorDistance(bgcolor)  ❹
mask = dist.binarize(50)  ❺

foreground = img - mask  ❻

background = background - mask.invert()  ❼

combined = background + foreground  ❽

combined.save(disp)
```

❶ This loads the background image of a weather map.

❷ Grab an image with the webcam. Mirror the image so that motions made in front of the webcam more intuitively match what is displayed on the screen.

❸ Rather than hard-code the color for the background, assume that the pixel at the coordinates `(10, 10)` is the background color.

❹ Find the color distance between each pixel and the background color. This is used to distinguish between foreground and background pixels.

❺ Since most pixels on the screen will have some similarity to the background color, binarize the image. This will ensure that the foreground components are pure black and the background components are pure white.

❻ By subtracting the mask from the foreground image, all of the background pixels will be zeroed out and the foreground pixels will be left untouched.

❼ Subtract the inverse of the mask from the background, it will create a hole in the image where the foreground image can be inserted.

❽ Finally, put the two images together.

CHAPTER 7

Drawing on Images

Vision systems usually need to provide some form of feedback to its users. Although this could be in the form of messages printed to a log, a spreadsheet, or other data output, users are most comfortable with graphical output, where key information is drawn directly on the image. This feedback is often more user friendly, because it is noticeable and can give the user more context about the message. When the program claims to have found features on an image, which ones did it find? Was it picking up noise? Is it finding the right objects? Even if the program will ultimately run without any user interaction, this type of feedback during development is vital. If, on the other hand, the system is designed to have users interacting with it, then drawing on images can be an important means for improving the user interface. For example, if the system measures several different objects, rather than printing a list of measurements to a console, it would be easier for the operator if the measurements were printed directly on the screen next to each object. This will spare the operator from having to guess which measurements correspond to the different objects on the screen, or trying to estimate which object is the correct one based on their coordinates.

The SimpleCV framework has a variety of methods for drawing and marking up images. Some of these are standard tools, found in most basic image manipulation programs. These are functions like drawing boxes, circles, or text on the screen. The SimpleCV framework also includes features more commonly found in intermediate or higher end applications, such as manipulating layers. In general, this chapter covers:

- A review of the Display and how images interact with the display
- An introduction to layers and the default Drawing Layer
- Drawing lines and circles
- Manipulating text, and its various characteristics such as color, font, text size, etc.

The Display

However, rather than recklessly diving right into drawing, it would be helpful to first understand the canvas. In Chapter 2, the `Display` object was introduced. This represents the window in which the image is displayed. The SimpleCV framework currently only supports a single display window at a time, and only one image can be displayed in that window. In most cases, the display is automatically created based on the size of the image to be displayed. However, if the display is manually initialized, the image will be automatically scaled or padded to fit into the display.

To work around the single display limitation, two images can be pasted side-by-side into a single larger image. The `sideBySide()` function does this automatically. For example, `img1.sideBySide(img2)` will place the two images side by side. Then these two images can be drawn to the single window, helping to work around the single window limitation.

```
from SimpleCV import Display, Camera, Image

display = Display(resolution = (800, 400)) ❶

cam = Camera(0, {'width': 320, 'height': 240}) ❷

img = cam.getImage()

img.save(display) ❸

# Should print: (320, 240)
print img.size()
```

❶ Initialize the display. Notice that the display's resolution is intentionally set to a very strange aspect ratio. This is to demonstrate how the image then fits into the resulting window.

❷ Initialize the camera. The camera is intentionally initialized with smaller dimensions and a different aspect ratio than the display.

❸ As mentioned in Chapter 1, "saving" an image to the display will show it in the window.

Notice the result in this case. The window remains `800 x 400` pixels. The image is scaled up so that the height now matches the 400 pixels of the window. Then padding is added to the left and right sides. The effect is similar to the `adaptiveScale()` function. The original image size is not altered. It remains `320 x 240` as is shown in the print statement, but it is scaled for display.

Images can also be divided into layers. Layers act like multiple images that sit on top of each other. The order of the layers is important. Items on the top layer cover items in the same area on the layer beneath it. Otherwise, if there is nothing covering them,

Figure 7-1. Left: The image as captured by the camera. Right: The image was scaled up so that it vertically filled the display. Then padding is added to the right and left to horizontally fill the display.

the items from the lower layers will show through. Although all processing could be done on a single layer of an image, multiple layers can simplify processing. For example, the original image can sit on the bottom layer, and then lines drawn around the edges of objects can appear on the next layer up. Then text describing the object can then sit on the top layer. This preserves the original image, while still ensuring that the drawings and markup appear to the end user.

Working with Layers

With the preliminaries aside, it is time to start drawing. Layers are implemented in the `DrawingLayer` class. For those who have previous experience working with PyGame, a `DrawingLayer` is a wrapper around PyGame's `Surface` class. This class has features for drawing text, managing fonts, drawing shapes, and generally marking up an image. Remember that one key advantage is that this all occurs on a layer above the main image, leaving the original image intact. This enables convenient markup while preserving the image for analysis.

Note: When saving an image with layers, the saved image is flattened — all of the data is merged together so that it appears as a single layer. The Image object itself remains separated into layers.

One of the easiest examples of using a layer is to draw one image on top of another. The following example demonstrates how to insert yourself into the classic American Gothic picture, as demonstrated in Figure 7-2.

```
from SimpleCV import Image

head = Image('head.png')

amgothic = Image('amgothic.png')

amgothic.dl().blit(head, (175, 110)) ❶
```

Figure 7-2. American Gothic

```
amgothic.show()
```

❶ The `dl()` function is just a shortcut for the `getDrawingLayer()` function. It saves you from having to write the whole getDrawingLayer name out. Since the layer did not already exist, `dl()` created the layer so that it could hold the face. Then `blit()` function—which is short for BLock Image Transfer—copied the provided image onto the background. The first parameter is the image to add, the second parameter is the coordinate at which to add the image.

The resulting program will add a third person into the image, as demonstrated in Figure 7-3.

The `blit()` function can be used without layers too. For example, `new Image = amgoth.blit(head)` will create a new flat image (without layers) that appears exactly the same. However, in this case the face will be copied right onto the main image, and any information about the original pixels that are covered in that spot are lost.

Figure 7-3. Three's a crowd

A lot of stuff is happening under the hood in the previous example. To look at this in more detail, consider the following example which works with the underlying layers:

```
from SimpleCV import Image, DrawingLayer

amgothic = Image('amgothic.png')
size = amgothic.size()

# Will print the image size: (468, 562)
print size

# Will print something like: [<SimpleCV.DrawingLayer object size (468, 562)>]
print amgothic.layers() ❶

layer1 = DrawingLayer(size) ❷

amgothic.addDrawingLayer(layer1) ❸

# Now prints information for two DrawingLayer objects
print amgothic.layers()
```

The above code essentially manually does everything that was done earlier with `background.dl()`. Obviously this is more complex, but it shows in more detail how to work with layers:

❶ The `layers()` property represents the list of all of the layers on the image.

❷ This creates a new drawing layer. By passing `DrawingLayer()` a tuple with the width and height of the original image, it creates a new layer that is the same size. At this point, the layer is not attached to the image.

❸ The `addDrawingLayer()` function adds the layer to the image, and appends it as the top layer. To add a layer in a different order, you would use the `insertDrawingLayer()` function instead.

The above process—creating a drawing layer and then adding it to an image—can be repeated over and over again. The layer added in the example above doesn't do anything yet. The next example shows how to get access to the layers so that they can be manipulated.

```
# This code is a continuation of the previous block of example code

# Will output: 2
print len(amgothic.layers())

drawing = amgothic.getDrawingLayer() ❶

drawing = amgothic.getDrawingLayer(1) ❷
```

❶ Using `getDrawingLayer()` with out an argument will have the function return the topmost drawing layer.

❷ In this case, since 1 was passed in as an argument, the function will return the drawing layer that has 1 as its index number.

In the previous examples, a drawing layer is first created, and then that drawing layer is added to an image. The layer is not actually fixed to the image. Instead, a layer from one image can actually be applied to another image. The following example builds off the first example in this chapter, where a face was placed on an image. This will re-extract the face from a layer, and then copy that whole layer to a new image.

```
from SimpleCV import Image, DrawingLayer

head = Image('head.png')

amgothic = Image('amgothic.png')

scream = Image('scream.png')

amgothic.dl().blit(head, (175, 110)) ❶

layer = amgothic.getDrawingLayer() ❷

scream.addDrawingLayer(layer) ❸
```

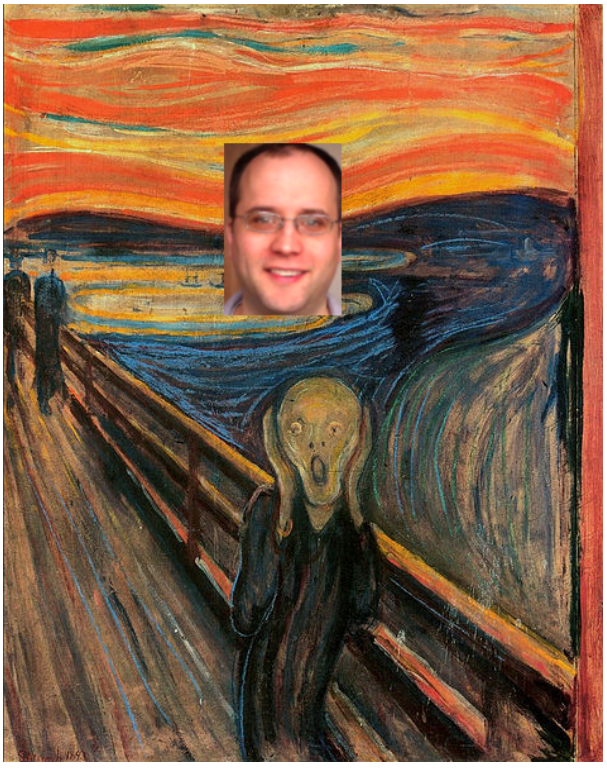


Figure 7-4. A face that makes you want to scream

```
scream.show()
```

❶ Up to this point, the example is similar to what we have seen before. This line adds the third head onto the American Gothic image.

❷ We then extract the layer back from the American Gothic image using the `getDrawingLayer()` function. In this case, there is just one layer, so no index is required when retrieving the layer.

❸ Now copy that layer to the image of The Scream. This will add a floating head above the screaming person in the original image. Note that even though the layer from the American Gothic image is a different size than The Scream image, the layer can still be copied.

The resulting image will look like Figure 7-4:

As demonstrated above, when copying a layer between images, the same layer is actually shown on both images. In other words, by modifying the layer, the resulting change will appear on both images.

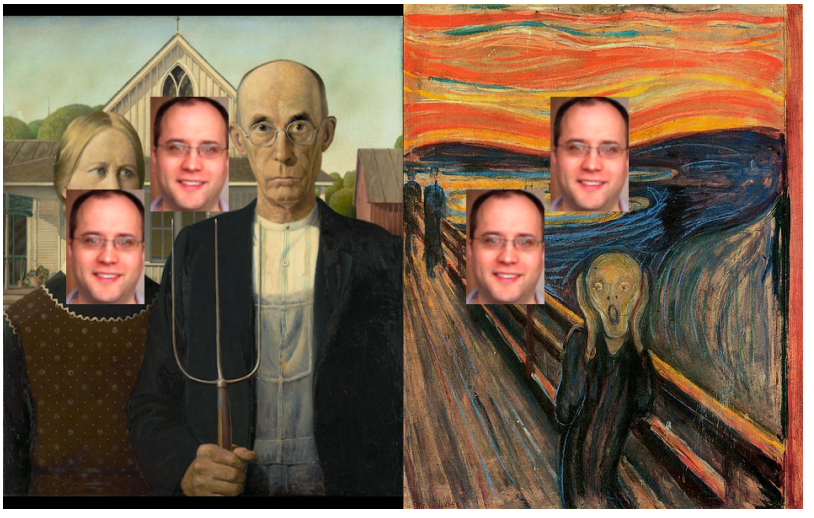


Figure 7-5. The picture was added to the layer, making it appear on both images.

```
# This is a continuation of the previous block of example code

print amgothic._mLayers ❶
print scream._mLayers

layer.blit(head, (75, 220)) ❷

amgothic.show() ❸
scream.show()
```

❶ These two print statements should print out identical information, showing it's the same object.

❷ By Passing the coordinates `(75, 220)` into the blit function, adding a second copy of the head image to the layer.

❸ Even though the layers was changed in one spot, the result propagates to both images.

Notice that now the change appears on both images, as demonstrated in Figure 7-5.

Drawing

The SimpleCV framework includes tools to draw several basic shapes such as lines, bezier curves, circles, ellipses, squares, and polygons. To show you the true power of the drawing engine, here is a rendering of a lollipop (there is a reason we didn't go to art school):

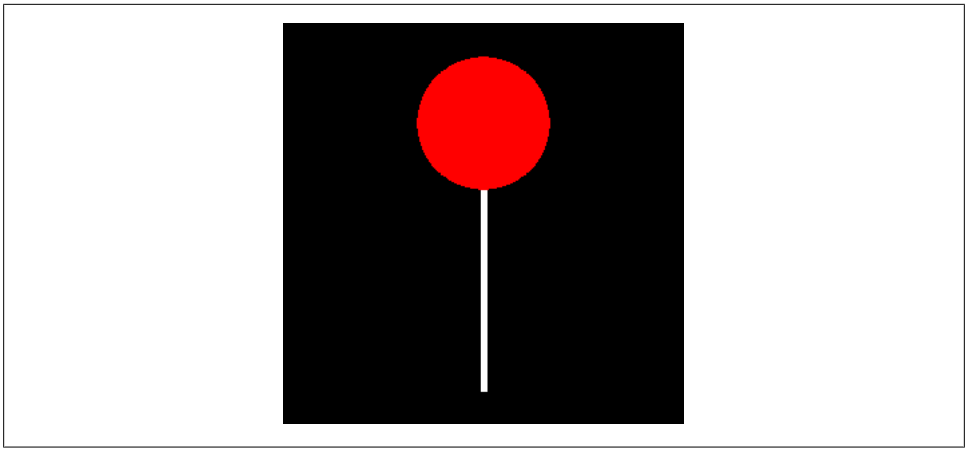


Figure 7-6. An example lollipop using the circle and line commands.

```
from SimpleCV import Image, Color

img = Image((300,300)) ❶

img.dl().circle((150, 75), 50, Color.RED, filled = True) ❷

img.dl().line((150, 125), (150, 275), Color.WHITE, width = 5) ❸

img.show()
```

❶ Create a blank 300x300 image

❷ Fetch the drawing layer, and draw a filled red circle

❸ Fetch the drawing layer, and draw a 5 pixel-wide line.

Most of the time, rather than drawing on a blank image we want to mark up existing data. Here we can show how to mark up your Camera feed so that you can easily calibrate the center, by adding markup similar to a scope:

```
from SimpleCV import Camera, Color, Display

cam = Camera()
size = cam.getImage().size()
disp = Display(size)

center = (size[0] /2, size[1] / 2)

while disp.isNotDone():
     img = cam.getImage()

     # Draws the inside circle
     img.dl().circle(center, 50, Color.BLACK, width = 3)  ❶

     # Draws the outside circle
     img.dl().circle(center, 200, Color.BLACK, width = 6)  ❷
```

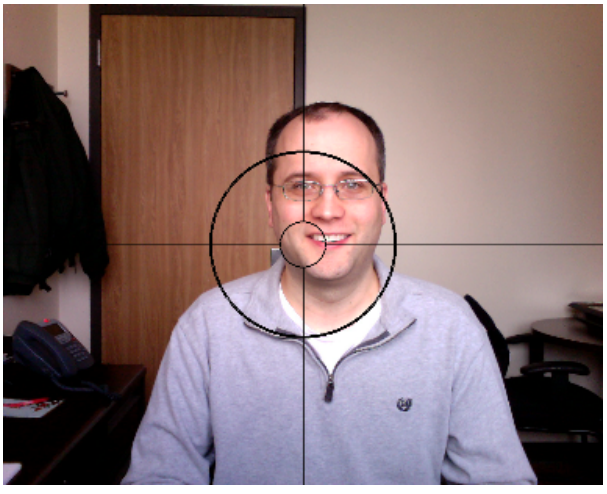


Figure 7-7. Example of painting crosshairs on an image

```
# Draw the radiating lines
img.dl().line((center[0], center[1] - 50), (center[0], 0),
     Color.BLACK, width = 2)
img.dl().line((center[0], center[1] + 50), (center[0], size[1]),
     Color.BLACK, width = 2)
img.dl().line((center[0] - 50, center[1]), (0, center[1]),
     Color.BLACK, width = 2)
img.dl().line((center[0] + 50, center[1]), (size[0], center[1]),
     Color.BLACK, width = 2)

img.save(disp)
```

We can also manage layers independently of images. Since the scope marks are the same in each frame, we can draw them once, and apply the layer to subsequent images. This reduces the amount of processing that needs to be done on each frame.

```
from SimpleCV import Camera, Color, Display, DrawingLayer

cam = Camera()

size = cam.getImage().size()

disp = Display(size)

center = (size[0] /2, size[1] / 2)

# Create a new layer and draw on it
scopelayer = DrawingLayer(size)

# This part works just like the previous example
scopelayer.circle(center, 50, Color.BLACK, width = 3)
scopelayer.circle(center, 200, Color.BLACK, width = 6)
```

```
scopelayer.line((center[0], center[1] - 50), (center[0], 0),
     Color.BLACK, width = 2)
scopelayer.line((center[0], center[1] + 50), (center[0], size[1]),
     Color.BLACK, width = 2)
scopelayer.line((center[0] - 50, center[1]), (0, center[1]),
     Color.BLACK, width = 2)
scopelayer.line((center[0] + 50, center[1]), (size[0], center[1]),
     Color.BLACK, width = 2)


while disp.isNotDone():
     img = cam.getImage()

     # Rather than a lot of drawing code, now we
     # can just add the layer to the image
     img.addDrawingLayer(scopelayer)

     img.save(disp)
```

When using markup on images, it's often a good idea to use it to highlight regions of interest, as in this example which uses the parking photo first demonstrated in Chapter 4. To demonstrate to the application user which part of the image will be cropped, we can draw a box around the relevant region:

```
from SimpleCV import Image, Color, DrawingLayer

car = Image('parking-car.png')

boxLayer = DrawingLayer((car.width, car.height)) ❶

boxLayer.rectangle((400, 100), (400, 400), color=Color.RED) ❷

car.addDrawingLayer(boxLayer) ❸

car.show()
```

❶ First, construct a new drawing layer to draw the box.
❶

❷ Next draw a rectangle on the layer using the `rectangle()` function. The first tuple is
❷ the `x, y` coordinates of the upper left corner of the rectangle. The second tuple represents the height and width of the rectangle. It is also colored red to make it stand out more.

❸ Now add the layer with the rectangle onto the original image.

This block of code will produce output as demonstrated in Figure 7-8:

This layer can then be used to compare the crop region with the parking image without a car. This will help to compare the regions of interest between the two versions of the photos.

```
# This is a continuation of the previous example

nocar = Image('parking-no-car.png')
```

Figure 7-8. Bounding box drawn around the yellow car

Figure 7-9. Box showing where the car is in the other picture

```
nocar.addDrawingLayer(boxLayer)
```

This works just like the previous example, but now it shows the region of interest on the similar image without the illegally parked car. The resulting photo should look like Figure 7-9.

For a slight variant on the above example, drawing circles on the image could help to identify the distance between objects. For example, using the same example image as above, assume that 100 pixels is about 3 feet. How far away is the volvo on the left from

the illegal parking spot? One way to visually estimate the distance is to draw a series of concentric circles radiating out from the handicap parking space.

```
from SimpleCV import Image, Color, DrawingLayer

car = Image('parking-car.png')

parkingSpot = (600, 300) ❶

circleLayer = DrawingLayer((car.width, car.height)) ❷

circleLayer.circle(parkingSpot, 100, color=Color.RED) ❸
circleLayer.circle(parkingSpot, 200, color=Color.RED)
circleLayer.circle(parkingSpot, 300, color=Color.RED)
circleLayer.circle(parkingSpot, 400, color=Color.RED)
circleLayer.circle(parkingSpot, 500, color=Color.RED)

car.addDrawingLayer(circleLayer) ❹

car.show()
```

❶ Since this example will draw a bunch of concentric circles, it will be easier to simply define the center once rather than constantly retyping it. This point corresponds approximately to the middle of the handicap spot.

❷ This line of code should be familiar from the rectangle drawing example. Create a new drawing layer with the same dimensions as the underlying image.

❸ Draw the circle using the `circle()` function. The first parameter is the center of the circle, which was predefined in step 1. The second parameter sets the radius of the circle. The third parameter sets the color. It defaults to black, so a red circle is specified to make it appear more obvious on the image.

❹ This line of code should now look familiar. It adds the new drawing layer to the image.

The resulting image should look like the image in Figure 7-10. Based on the image, the car is four to five circles away from the center of the handicap spot. Since the radius increase by 100 pixels per circle, that translates into 400 to 500 pixels. Under the rough assumption that 100 pixels equals three feet, then the Volvo is currently around 15 feet away from the center of the illegal parking space. It is probably safe to assume that it is legally parked.

The Blobs chapter demonstrates a more rigorous approach to measuring between objects on an image.

After all this drawing, it may be necessary to remove all the stuff and start with a clean slate. To clear the image, use the `clearLayers()` function.

Figure 7-10. Circles approximating distance from the car

```
# This is a continuation of the previous example

car.clearLayers()

car.show()

print car.layers()
```

The car image is back to its original state before the circles were drawn. Visually, the circles no longer appear. When printing out a list of all the layers on the image, no layers appear. Note, however, that the original `circleLayer` object still exists. It can be placed back on the image with the `addDrawingLayer()` function.

This section did not cover all of the possible shapes, but it did present the standard approach. Several other frequently used options for drawing include:

- Line: `line(start=(x1, y1), end=(x2, y2)`. Draws a straight line between the two provided points.
- Elipses: `elipse(center=(x, y), dimensions = (width, height))`. Draws a "squished" circle.
- Centered Rectangle: `centeredRectangle(center=(x, y), dimensions=(x,y))`. Similar to rectangle(), except the center point is provided instead of the upper-left corner, in some cases saving some arithmetic.
- Polygon: `polygon(points = [ (x1, y1), (x2, y2), (x3, y3) …])` by providing a list of `x, y` point, this will draw lines between all connected points.
- Bezier Curve: `bezier(points = (p1, p1, p3, p4, …), steps=s)` Draws a curve based on the specified control points and the number of steps.

Text and Fonts

Displaying text on the screen is fairly simple. In the previous example using circles to show distances, the example assumed that users knew that each circle was about three feet in radius. It would be better to mark those distances on the screen. Fortunately, drawing text on an image is just as easy as drawing shapes.

```
from SimpleCV import Image, Color, DrawingLayer

# This first part of the example is the same as before

car = Image('parking-car.png')

parkingSpot = (600, 300)

circleLayer = DrawingLayer((car.width, car.height))

circleLayer.circle(parkingSpot, 100, color=Color.RED)
circleLayer.circle(parkingSpot, 200, color=Color.RED)
circleLayer.circle(parkingSpot, 300, color=Color.RED)
circleLayer.circle(parkingSpot, 400, color=Color.RED)
circleLayer.circle(parkingSpot, 500, color=Color.RED)

car.addDrawingLayer(circleLayer)

# Begin new material for this example:

textLayer = DrawingLayer((car.width, car.height)) ❶

textLayer.text("3 ft", (500, 300), color=Color.RED) ❷
textLayer.text("6 ft", (400, 300), color=Color.RED)
textLayer.text("9 ft", (300, 300), color=Color.RED)
textLayer.text("12 ft", (200, 300), color=Color.RED)
textLayer.text("15 ft", (100, 300), color=Color.RED)

car.addDrawingLayer(textLayer)

car.show()
```

Many of the components of this example should now be familiar based on previous examples. However, a couple items are worth noting:

❶ This example will add the text to its own drawing layer. Technically, this could be done on the same layer as the circles, but this makes it easier to easily add/subtract the labels.

❷ To do the actual writing on the screen, use the `text()` function. The first parameter defines the actual text to be written to the screen. The second parameter is a tuple with the `x, y` coordinates of the upper left corner of the text. The final parameter defines the color as red to make it slightly easier to read.

The resulting image should look like Figure 7-11:

Figure 7-11. Distance circles with labels

Although the red colored font is slightly easier to read than a simple black font, the labels are still a bit difficult to read. A larger, easier to read font would help. Changing the font is relatively simple. The first step is to get a list of fonts that are available on the system. The `listFonts()` function will do the trick:

```
# This is a continuation of the previous example
textLayer().listFonts()
```

This will print a list of all the fonts installed on the system. Actual results will vary from system to system. The name of the font is just the part between the single quotation marks. For example, `u'arial'` indicates that the `arial` font is installed. The `u` prefix in front of some entries merely indicates that the string was encoded in unicode. Because `arial` is a pretty common font on many systems, it will be used in the following examples.

The following examples may need to be adjusted slightly based on available fonts.

Then to change the font, simply insert code such as indicated below before calling the `text()` function:

```
textLayer.selectFont('arial') ❶
textLayer.setFontSize(24) ❷
```

❶ Select the font based on the name. This name is taken from the `listFonts()` function previously demonstrated.

❷ The font size is changed with the `setFontSize()` function.

Figure 7-12. Circles with larger, easier to read labels

If used with the previous example, the output will look like the example in Figure 7-12.

Several other font configuration commands are also available:

- `setFontBold()`: make the font bold
- `setFontItalic()`: make the font italic
- `setFontUnderling()`: underline the text

One final way to make it easier to see the drawn material is to control the layer's transparency. By making it more opaque, it will mask the underlying material, making the drawn shapes and text stand out.

```
from SimpleCV import Image, Color, DrawingLayer

car = Image('parking-car.png')

parkingSpot = (600, 300)

circleLayer = DrawingLayer((car.width, car.height))

#This is the new part:
circleLayer.setLayerAlpha(175)

# The remaining sections is the same as before
circleLayer.circle(parkingSpot, 100, color=Color.RED)
circleLayer.circle(parkingSpot, 200, color=Color.RED)
circleLayer.circle(parkingSpot, 300, color=Color.RED)
circleLayer.circle(parkingSpot, 400, color=Color.RED)
circleLayer.circle(parkingSpot, 500, color=Color.RED)

car.addDrawingLayer(circleLayer)
```

Figure 7-13. Partially transparent layer on top of image

```
textLayer = DrawingLayer((car.width, car.height))

textLayer.selectFont('arial')
textLayer.setFontSize(24)

textLayer.text("3 ft", (500, 300), color=Color.RED)
textLayer.text("6 ft", (400, 300), color=Color.RED)
textLayer.text("9 ft", (300, 300), color=Color.RED)
textLayer.text("12 ft", (200, 300), color=Color.RED)
textLayer.text("15 ft", (100, 300), color=Color.RED)

car.addDrawingLayer(textLayer)

car.show()
```

The one new line of code was using the `setLayerAlpha` function. This takes a single parameter of an alpha value between 0 and 255. An alpha of zero means that the background of the layer is transparent. If the alpha is 255, then the background is opaque. By setting it to a value of 175 in the example, the background is darkened but still visible, as is shown in Figure 7-13.

Be careful when changing the alpha on multiple layers. If each layer progressively makes the background more opaque, they will eventually completely black out the background.

Examples

The two examples in this chapter demonstrate how to provide user feedback on the screen. The first example draws stop and go shapes on the screen in response to the

amount of light captured on the camera. The second example expands on the crosshair code snippets above, integrating mouse events. Finally, the third example shows how to simulate a zoom effect in a window, controlled with the mouse wheel.

Making a custom display object

This example is for a basic Walk or Don't Walk type scenario. It detects hypothetical vehicles approaching based on whether the light is present. If it detects light, it will show a message on the screen saying `STOP`.

This example code is intended to demonstrate layers and drawing. It is not intended to be an actual public safety application.

```
from SimpleCV import Camera, Color, Display, DrawingLayer, np

cam = Camera()

img = cam.getImage()

display = Display()

width = img.width
height = img.height

screensize = width * height

# used for automatically breaking up image.
divisor = 5

# color value to detect blob is a light
threshold = 150

# Create the layer to display a stop sign
def stoplayer():
     newlayer = DrawingLayer(img.size()) ❶

     # The corner points for the stop sign's hexagon
     points = [(2 * width / divisor, height / divisor),
               (3 * width / divisor, height / divisor),
               (4 * width / divisor, 2 * height / divisor),
               (4 * width / divisor, 3 * height / divisor),
               (3 * width / divisor, 4 * height / divisor),
               (2 * width / divisor, 4 * height / divisor),
               (1 * width / divisor, 3 * height / divisor),
               (1 * width / divisor, 2 * height / divisor)
                        ]

     newlayer.polygon(points, filled=True, color=Color.RED) ❷
```

```
        newlayer.setLayerAlpha(75) ❸

        newlayer.text("STOP", (width / 2, height / 2), color=Color.WHITE) ❹

        return newlayer

    # Create the layer to display a go sign
    def golayer():
        newlayer = DrawingLayer(img.size()) ❺

        newlayer.circle((width / 2, height / 2), width / 4, filled=True,
                        color=Color.GREEN)

        newlayer.setLayerAlpha(75)

        newlayer.text("GO", (width / 2, height / 2), color=Color.WHITE)

        return newlayer


    while display.isNotDone():
        img = cam.getImage()

        # The minimum blob is at least 10% of screen
        min_blob_size = 0.10 * screensize

        # The maximum blob is at most 80% of screen
        max_blob_size = 0.80 * screensize

        # Get the largest blob on the screen
        blobs = img.findBlobs(minsize=min_blob_size, maxsize=max_blob_size)

        # By default, show the go layer
        layer = golayer()

        # If there is a light then show the stop
        if blobs:
            # Get the average color of the blob
            avgcolor = np.mean(blobs[-1].meanColor())

            # This is triggered by a bright light
            if avgcolor >= threshold:
                layer = stoplayer()

        # Finally, add the drawing layer
        img.addDrawingLayer(layer) ❻

        img.save(display)
```

❶ The stop sign will be a hexagon. The SimpleCV framework does not have a function specifically for drawing a hexagon, so instead it needs a set of corner points which define the shape.

❷ Use the `drawPolygon()` function and the list of corner points to construct the stop sign.

Figure 7-14. Example display when it is safe to walk.

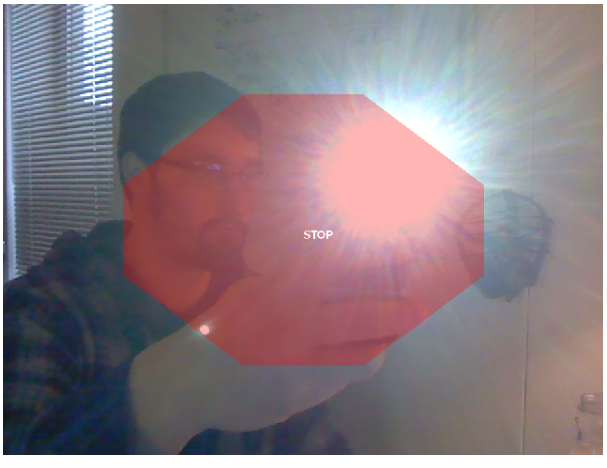


Figure 7-15. Example display when it is not safe to walk.

❸ Set the layer's alpha in order to partially gray out the background. Although the background is still visible, this helps draw attention to the stop sign.

❹ Add text in the middle of the stop sign to say "Stop."

❺ The overall flow of this code is the same as in the previous steps, but it is slightly simpler since the sign in a circle instead of a hexagon.

❻ Finally add the layer. Depending on whether the light was shone at the camera, this will be either the stop sign or the go sign.

Keep in mind that lighting conditions and other environmental factors will affect how this application works. Play around with the threshold values to adjust the assurance. Although a train/traffic detector is a somewhat silly application, it demonstrates the basic principles behind detecting an object on the screen, and then changing the screen output to alert the user — a very common activity in vision systems.

Moving Target

The next example is similar to the crosshairs example shown earlier in the chapter. However, unlike the earlier example that fixed the crosshairs in the middle of the screen, this example lets the user control where it is aiming by clicking the left mouse button.

```
from SimpleCV import Camera, Color, Display

cam = Camera()

size = cam.getImage().size()

disp = Display(size)

center = (size[0] /2, size[1] / 2)

while disp.isNotDone():
     img = cam.getImage()

     img = img.flipHorizontal() ❶

     if disp.mouseLeft:
          center = (disp.mouseX, disp.mouseY) ❷


     # The remainder of the example is similar to the
     # short version from earlier in the chapter

     # Inside circle
     img.dl().circle(center, 50, Color.BLACK, width = 3)
     # Outside circle
     img.dl().circle(center, 200, Color.BLACK, width = 6)

     # Radiating lines
     img.dl().line((center[0], center[1] - 50), (center[0], 0),
          Color.BLACK, width = 2)
     img.dl().line((center[0], center[1] + 50), (center[0], size[1]),
          Color.BLACK, width = 2)
     img.dl().line((center[0] - 50, center[1]), (0, center[1]),
          Color.BLACK, width = 2)
     img.dl().line((center[0] + 50, center[1]), (size[0], center[1]),
          Color.BLACK, width = 2)

     img.save(disp)
```

❶ The `flipHorizontal()` function make the camera act more like a mirror. This step is not strictly necessary. However, since most users are likely using this while sitting

in front of a monitor-mounted web cam, this will give the application a slightly more intuitive feel.

❷ Update the center point by looking for mouse clicks. The `mouseLeft` event is triggered when the user clicks the left mouse button on the screen. Then the center point is updated to the point where the mouse is clicked, based on the `mouseX` and `mouseY` coordinates.

The `mouseLeft` button is triggered if the user presses the button. They do not need to release the button to trigger this. So clicking and holding the button while dragging the mouse around the screen will continuously update the crosshair location.

Image Zoom

The scroll wheel on a mouse is frequently used to zoom in or out on an object. Although most web cameras do not have a zoom feature, this can be faked by scaling up (or down) an image, and then cropping to maintain the original image dimensions. This example reviews how to interact with the display window, and it shows how to manage images when they are different sizes than the display window.

```
from SimpleCV import Camera, Display

cam = Camera()

disp = Display((cam.getProperty('width'), cam.getProperty('height')))

# How much to scale the image initially
scaleFactor = 2

while not disp.isDone():
     # Check for user input
     if disp.mouseWheelUp: ❶
          scaleFactor *= 1.1
     if disp.mouseWheelDown:
          scaleFactor /= 1.1

     # Adjust the image size
     img = cam.getImage().scale(scaleFactor) ❷

     # Fix the image so it still fits in the original display window
     if scaleFactor < 1:
          img = img.embiggen(disp.resolution) ❸
     if scaleFactor > 1:
          img = img.crop(img.width / 2, img.height / 2,
                    disp.resolution[0], disp.resolution[1], True) ❹

     img.save(disp)
```

❶ Check if the mouse is scrolled up or down, and scale the image accordingly. Each event will increase or decrease the image size by 10%.

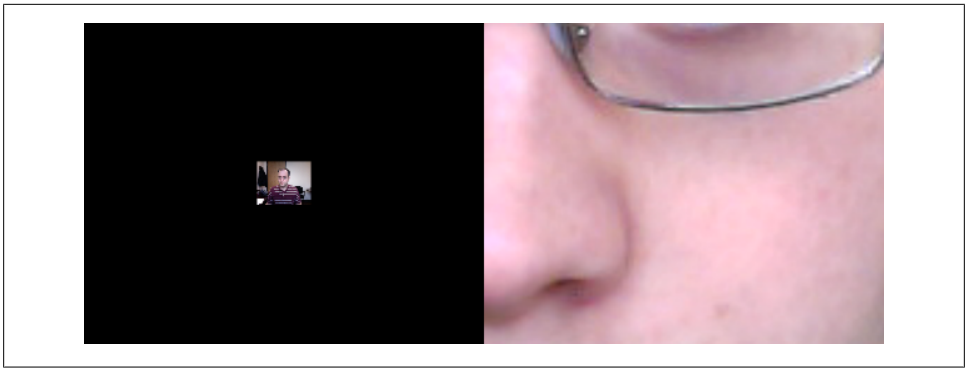


Figure 7-16. Left: camera zoomed out. Right: camera zoomed in.

❷ This step does the actual scaling. Note that at this step, the image is no longer the same size as the display window.

❸ If the image is smaller than the display—as represented by a scale factor that is less than one—then it needs to be embiggened. Recall that `embiggen()` changes the canvas size of the image but leaves the underlying image intact.

❹ If the image is larger than the display, crop it. This example assumes that it should zoom into the middle. That means the crop region is defined based on the center of the image. It is cropped to the same dimensions as the display so that it will fit.

CHAPTER 8
Basic Feature Detection

When you look at something, your brain does a lot of pattern recognition to "make sense" of what you see. You focus on an object, identify the characteristics of the object — such as its shape, color, or texture — and then compare these to the characteristics of familiar objects. When you have a match, you recognize the object you see as being the same or similar to an object you already know. This is how your brain organizes and defines what you see.

In computer vision, that process of deciding what to focus on is called feature detection. A feature in this sense can be formally defined as "one or more measurements of some quantifiable property of an object, computed so that it quantifies some significant characteristics of the object" (Kenneth R. Castleman, *Digital Image Processing*, Prentice Hall, 1996). An easier way to think of it, though, is that a feature is an “interesting” part of an image. What makes it interesting? Well, consider a photograph of a red ball on a gray sidewalk. The sidewalk itself probably isn’t that interesting. It’s the ball that you would focus on, because it’s significantly different from the rest of the photograph. Similarly, if you were having a computer analyze the photograph, the gray pixels representing the sidewalk wouldn’t necessarily tell you much, and we might consider it background. It would be the pixels that represent the ball that might be able to tell you something, like how big the ball is, or where on the sidewalk it lies.

With your vision system, you don’t want to waste time (or processing power) analyzing the unimportant or uninteresting parts of an image, so feature detection helps find which pixels you should focus on. This chapter focuses on the most basic types of features: Blobs, Lines, Circles and Corners. Features represent a new way of thinking about the image — rather than using the image as a whole, we’ll instead focus on just a few relevant pieces of an image.

With features, we’re also now looking at the image in a way that may be independent of the whole image. If the detection is robust, a feature is something that could be reliably detected across multiple images. For instance, with the red ball, the features of the ball do not really change based on the location of the ball in the image. We could use feature detection to tell if the ball is rolling or at rest. The shape and color of the

ball would be the same in the lower left corner of the image as it would be in the upper right corner. Similarly, if we changed the color of the light illuminating the ball, the color of the pixels representing the ball might change, but the hue and shape of the ball wouldn't. This means that how we describe the feature can also determine in which situations we can detect the feature.

Our detection criteria for the feature will determine whether we can:

- find the features in different locations of the picture (position invariant)
- find the feature if it's large or small, near or far (scale invariant)
- find the feature if it's rotated at different orientations (rotation invariant)

In this chapter, we will examine the different things we can look for with feature detection, starting with the meat and potatoes of feature detection, the "Blob".

Blobs

Blobs, also called objects or connected components, are regions of similar pixels in an image. This could be a group of brownish pixels together, which might represent food in a pet food detector. It could be a group of shiny metal looking pixels, which on a door detector would represent the door knob. A blob could be a group of matte white pixels, which on a medicine bottle detector could represent the cap. Blobs are valuable in machine vision because many things can be described as an area of a certain color or shade in contrast to a background.

The term background in computer vision can mean any part of the image which is not the object of interest, it may not necessarily describe what is closer or more distant to the camera.

Once a blob is identified, we can measure a lot of different things:

- A blob's area will tell us how many pixels are in it
- We can measure the dimensions such as width and height
- We can find the center, based on the "mid point" or the center of mass (centroid)
- We can count the number of blobs to find different objects
- We can look at the color of blobs
- We can look at its angle to see its rotation
- We can find how close it is to a circle, square, or rectangle — or compare its shape to another blob.

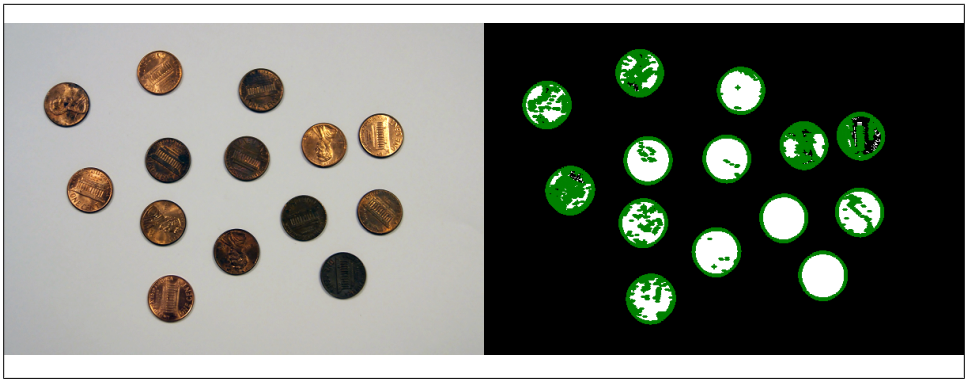


Figure 8-1. Left: Original image of pennies. Right: Blobs detected.

Finding Blobs

Basic Blob Detection

At its most basic, `findBlobs()` can be used to find objects which are lightly colored in an image. If you do not specify any parameters, the function will try to automatically detect what is bright and what is dark. We can use it to find the pennies as shown in Figure 8-1:

```
from SimpleCV import Image

pennies = Image("pennies.png")

binPen = pennies.binarize() ❶
blobs = binPen.findBlobs()  ❷

blobs.show(width=5)  ❸
```

❶ Blobs are most easily detected on a binarized image since that creates contrast between the feature in question and the background. This step is not strictly required on all images, but it makes it easier to detect the blobs.

❷ Since no arguments are being passed to the `findBlobs()` function, it will return a `FeatureSet` of the continuos light colored regions in the image. We'll talk about a `FeatureSet` in more depth later on in this chapter, but the general idea is that this will be a list of features about the blobs found. A `FeatureSet` also has a set of defined methods that are useful when handling features.

❸ Notice that the show() function is being called on blobs and not the `Image` object. The `show()` function is one of the methods defined with a `FeatureSet`. What it does is draw each feature in the `FeatureSet` on top of the original image, and then displays the results. By default, it will draw the features in green, but you can pass it a color and width argument as well. For instance, if you wanted to see the blobs in red, you could change the above code to `packs.show(Color.RED)`.

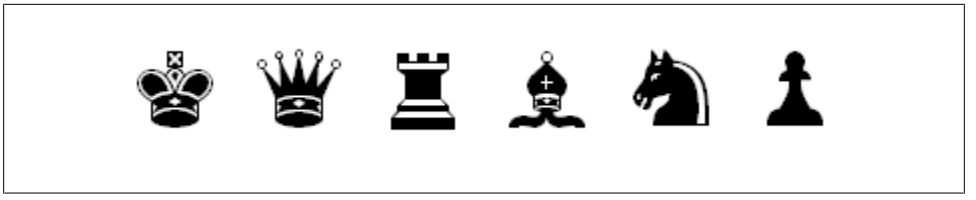


Figure 8-2. Black chess pieces

The pennies are now shown in green, as is shown in Figure 8-1. To find out more information about the features found, you can use other methods, such as:

```
from SimpleCV import Image

pennies = Image("pennies.png")

binPen = pennies.binarize()
blobs = binPen.findBlobs()

print "Areas: ", blobs.area()  ❶

print "Angles: ", blobs.angle()  ❷

print "Centers: ", blobs.coordinates()  ❸
```

❶ The area function returns an array of the area of each feature in pixels. By default, the blobs are sorted by size, so the areas should be ascending in size. The sizes will vary since sometimes it detected the full penny but other times it only detected a portion of the penny.

❷ The angle function returns an array of the angles, as measured in degrees, for each feature. The angle is the measure of rotation of the feature away from the X axis, which is 0. A positive number indicates a counter-clockwise rotation, and a negative number is a clockwise rotation.

❸ The coordinates function returns a 2-dimensional array of the `(X, Y)` coordinates for the center of each feature.

These are just a few of the available `FeatureSet` functions. As you can see from these examples, the functions are designed to return various attributes of the features. For information about these functions, see the FeatureSet section of this chapter.

Finding Dark Blobs

The `findBlobs()` function easily finds lightly colored blobs on a dark background. This is one reason why binarizing an image makes it easier to detect the object of interest. But what if the objects of interest are darkly colored on a light background? In that case, you just want to use the `invert()` function. For instance, let's say you wanted to find the chess pieces in Figure 8-2:

You could use the following code to find the blobs that represent the chess pieces:

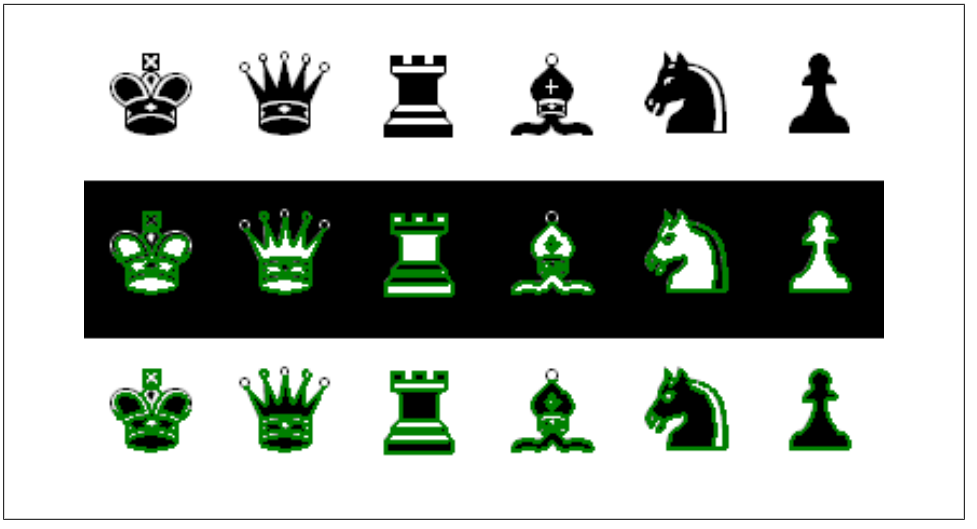


Figure 8-3. Top: the original image. Middle: blobs drawn on the inverted image. Bottom: Blobs on the original image.

```
from SimpleCV import Image

img = Image("chessmen.png")

invImg = img.invert() ❶
blobs = invImg.findBlobs() ❷

blobs.show(width=2) ❸

img.addDrawingLayer(invImg.dl()) ❹
img.show()
```

❶ The `invert()` function will turn the black chess pieces white, and turn the white background to black.

❷ The `findBlobs()` function can then find the lightly colored blobs as it normally does.

❸ Show the blobs. Note, however, that this function will show the blobs on the inverted image, not the original image.

❹ To make the blobs appear on the original image, take the drawing layer from the inverted image—which is where the blob lines were drawn—and add that layer to the original image.

The result will look like Figure 8-3:

Figure 8-4. Everyone knows the blue ones are the tastiest.

Finding Blobs of a Specific Color

Our world does not consist solely of light and dark objects. More often than not, you're going to want to identify features in an image that share a similar color. Perhaps you want to find all of the blue candies in Figure 8-4.

To find the blobs that represent the blue candies, we'd use the color information returned from the `colorDistance()` function that we first introduced in Chapter 4, and later revisited in Chapter 5. Here's the example code:

```
from SimpleCV import Color, Image

img = Image("mandms.png")

blue_distance = img.colorDistance(Color.BLUE).invert()  ❶

blobs = blue_distance.findBlobs()  ❷

blobs.draw(color=Color.PUCE, width=3)  ❸
blue_distance.show()

img.addDrawingLayer(blue_distance.dl()) ❹
img.show()
```

❶ The `colorDistance()` function returns an image that shows how far away the colors in the original image are from the passed in Color.Blue argument. To make this even more accurate, we could find the RGB triplet for the actual blue color on the candy, but this works well enough. Since any color close to blue will be black, and colors far away from blue will be white, we again use the `invert()` function to switch the target blue colors to white instead.

❷ We use the new image to find the blobs representing the blue candies. We can also fine tune what the `findBlobs()` function discovers by passing in a threshold argument. The threshold can either be an integer or an RGB triplet. When a threshold

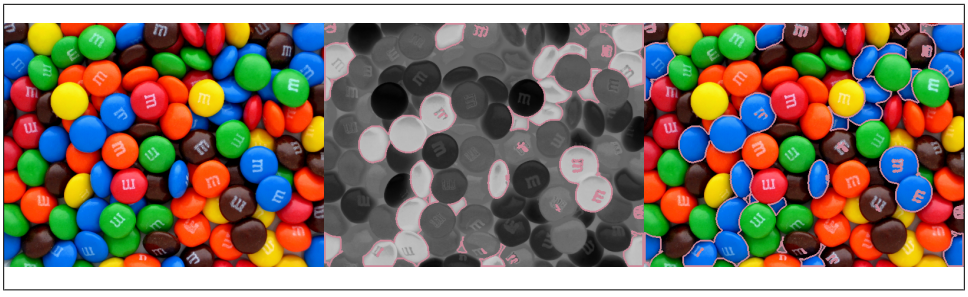


Figure 8-5. Left: the original image. Center: blobs based on the blue distance. Right: The blobs on the original image.

value is passed in, the function will change any pixels that are darker than the threshold value to white, and any pixels above the value to black.

❸ In the previous examples, we have used the `FeatureSet show()` method instead of these two lines (ie, blobs.show()). That would also work here. We've broken this out into the two lines here just to show that they are the equivalent of using the other method. To outline the blue candies in a color not otherwise found in candy, they are drawn in puce, which is a redish color.

❹ Similar to the previous example, the drawing ends up on the `blue_distance` image. Copy the drawing layer back to the original image.

The resulting matches for the blue candies will look like Figure 8-5.

Sometimes the lighting conditions can make color detection more difficult. To demonstrate the problem, the following image is based on the original picture of M&M's, but the left part of the image is darkened. the `hueDistance()` function a better choice for this type of scenario. For instance, if the photo looked like Figure 8-6, where the right half is darkened. The right half of the image is intentionally very dark, but the following example will show how the colors can still be used to find blobs, and provide a good example of why hue is more valuable than RGB color in

To see how this is a problem, first consider using the previous code on this picture:

```
from SimpleCV import Color, Image

img = Image("mandms-dark.png")

blue_distance = img.colorDistance(Color.BLUE).invert()
blobs = blue_distance.findBlobs()
blobs.draw(color=Color.PUCE, width=3)

img.addDrawingLayer(blue_distance.dl())
img.show()
```

The results are shown in Figure 8-7. Notice that only the blue candies on the left are detected.

Figure 8-6. The candies with the right darkened

Figure 8-7. Darkened right half of the image

To resolve this problem use `hueDistance()` instead of `colorDistance()`. Since the hue is more robust to changes in light, the darkened right half of the image will not create a problem.

```
from SimpleCV import Color, Image

img = Image("mandms-dark.png")

blue_distance = img.hueDistance(Color.BLUE).invert() ❶
blobs = blue_distance.findBlobs()
blobs.draw(color=Color.PUCE, width=3)

img.addDrawingLayer(blue_distance.dl())
img.show()
```

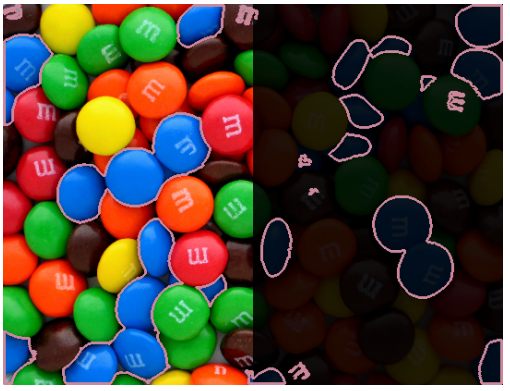


Figure 8-8. Blobs detected with `hueDistance()`

❶ The one difference in this example is the use of the `hueDistance()` function. It works like the `colorDistance()` function, but notice that is produeces better results, as indicated in Figure 8-8.

Lines and Circles

Lines

A line feature is a straight edge in an image, which usually denotes the boundary of an object. It sounds fairly straight-forward, but the calculations involved for identifying lines are likely more complex then you might think. The reason is because an edge is really a list of (X,Y) coordinates, and any two coordinates could possibly be connected by a straight line. For instance, [figure to come] shows four coordinates, and then two different examples of line segments that might connect those four points. It's hard to say which one is right — or if either of them are since there are also other possible solutions. The way this problem is handled behind-the-scenes is by using the Hough transform technique. This technique effectively looks at all of the possible lines for the points, and then figures out which lines show up the most often. The more frequent a line is, the more likely the line is an actual feature.

You don't have to worry about performing these calculations yourself in your code. To find the line features in an image, all you need to do is use the `findLines()` function. The function itself utilizes the Hough transform, and returns to you a `FeatureSet` of the lines found. The functions available with a `FeatureSet` are the same regardless of the type of feature involved. However, there are `FeatureSet` functions that you may find more useful when dealing with lines. These functions include:

- `coordinates()`: returns the `(X, Y)` coordinates of the starting point of the line(s)

Figure 8-9. Example of a basic line detection with a package

- `width()`: returns the width of the line, which in this context is the difference between the starting and ending `X` coordinates of the line.
- `height()`: returns the height of the line, or the difference between the starting and ending `Y` coordinates of the line.
- `length()`: returns the length of the line in pixels.

The following code demonstrates how to find and then display the lines in an image. The example looks for lines on a block of wood.

```
from SimpleCV import Image

img = Image("block.png")

lines = img.findLines()  ❶

lines.draw(width=3)  ❷

img.show()
```

❶ The `findLines()` function will return a `FeatureSet` of the line features.

❷ This will draw the lines in green on the image, with each line having a width of 3 pixels.

Since this is just a simple block of wood in a well-lit environment, the `findLines()` function does a reasonable job finding the line features using the default values for it's parameters. Many situations may require some tweaking to `findLines()` to get the desired results. For instance, notice that it found the long lines along the top and bottom of the block, but it did not find the lines along the side.

The `findLines()` function includes five different tuning parameters to help improve the quality of the results:

- Threshold: determines how strong the edge should before it is recognized as a line
- Minlinelength: what the minimum length of recognized lines will be
- Maxlinegap: how much of a gap will be tolerated in a line

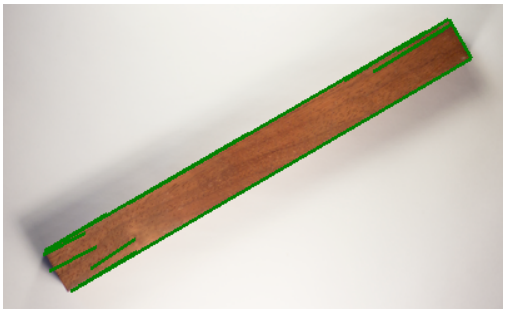


Figure 8-10. Line detection at a lower threshold.

- Cannyth1: a threshold parameter used with the edge detection step which sets the minimum "edge strength"
- Cannyth2: a second parameter for edge detection which sets "edge persistence"

The threshold parameter for `findLines()` works in much the same way as the threshold parameter does for the `findBlobs()` function. If you don't pass in a threshold argument, the default value it uses is set to 80. The lower threshold value, then more lines will be found by the function. For higher values, fewer lines will be found. Using the block of wood picture again, here's the code using a low threshold value:

```
from SimpleCV import Image

img = Image("block.png")

# Set a low threshold
lines = img.findLines(threshold=10)

lines.draw(width=3)

img.show()
```

Here's what the resulting image would look like — with many more lines detected on the block. Notice that it found one of the side lines, but at the cost of several superfluous lines along the grains of wood.

If the threshold value for `findLines()` is set too high, then no lines will be found.

One way to get rid of small lines is to use the `minlinelength` parameter to weed out short lines. The length of a line is measured in pixels, and the default value `findLines()` uses is 30 pixels long. In the example below, the minimum length is boosted, weeding out some lines.

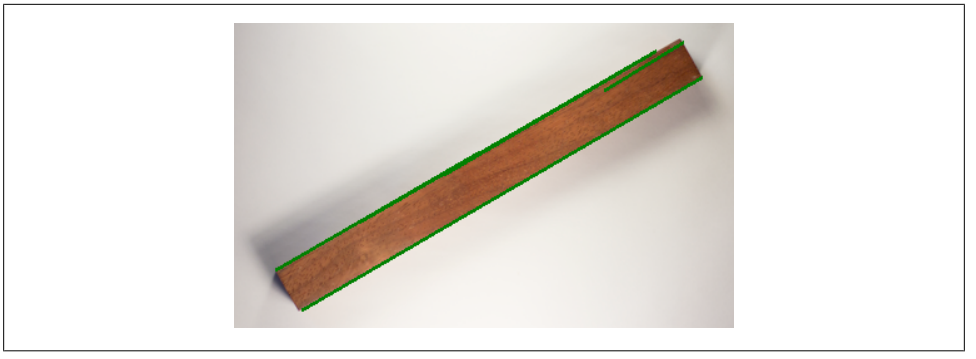


Figure 8-11. Line detection with an increased minimum length.

```
from SimpleCV import Image

img = Image("block.png")

lines = img.findLines(threshold=10, minlinelength=50)

lines.draw(width=3)

img.show()
```

The result is shown in Figure 8-11. Notice that it eliminated a couple of the extra lines, but it came at a cost of eliminating the the side lines again.

Changing the line length doesn't solve the problem. The two ends of the image still are not found. In fact, it could create the opposite problem. Sometimes the algorithm may find very small lines and it needs to know whether those small line segments actually represent one larger continuous line. The `findLines()` function can ignore small gaps in a line, and recognize it as the larger overall line. By default, the function will combine two segments of a line if the gap between them is 10 pixels or less. You can use the `maxlinegap` parameter to fine tune how this works. The following example allows for a larger gap between lines, potentially allowing it to discover a few small lines that constitute the edge.

```
from SimpleCV import Image

img = Image("block.png")

lines = img.findLines(threshold=10, maxlinegap=20)

lines.draw(width=3)

img.show()
```

The result restores the right edge line again, but once again, it finds a lot of superfluous lines, as demonstrated in Figure 8-12.

Notice that while setting the minimum line length decreased the number of lines found in the painting, adding in a longer gap then dramatically increased the number of lines.

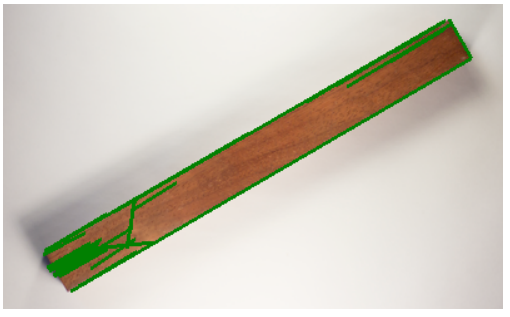


Figure 8-12. Line detection with a larger allowance for the gap.

With the bigger gap, the line segments can be combined to meet the line length requirements, and more lines are then recognized. The last two parameters, `cannyth1` and `cannyth2`, are thresholds for the Canny edge detector. Roughly speaking, edges are detected by looking for changes in brightness. The first of these threshold parameters controls how much the brightness needs to change to detect an edge. The second parameter controls the threhold for linking together multiple edges. Both of these parameters act the same way the previous three parameters acted: smaller values mean more lines will be detected, which could just be adding noise. Conversely, larger values will have less lines detected, but may mean that some valid lines aren't being returned. With all of these parameters, the trick is to work with the parameters until you're in the range that makes the most sense for your application.

Of course, sometimes it is easier to simply modify the image to reduce the noise rather than fine tuning the parameters. The following example, finds the desired lines on the block:

```
from SimpleCV import Image

img = Image('block.png')
dist = img.colorDistance((150, 90, 50))
bin = dist.binarize(70).morphClose()

lines = bin.findLines(threshold=10, minlinelength=15)
lines.draw(width=3)

# Move the lines drawn on the binary image back to the main image
img.addDrawingLayer(bin.dl())
img.show()
```

Notice that by banalizing the image, it eliminated a bunch of the noise that was causing the false-positive lines to appear where they should not. As a result, this draws lines around the block, as shown in Figure 8-13

Figure 8-13. Block with lines on all edges

Circles

In addition to line features, you can also work with circles. The method to find circular features is called `findCircle(), and it works the same way +findLines()` does. It returns a `FeatureSet` of the circular features it finds, and it also has parameters to help set it's sensitivity. These parameters include:

- Canny: This is a threshold parameter for the Canny edge detector. The default value is 100. If you set canny to a lower value, a greater number of circles will be recognized, while a higher value will instead mean fewer circles.
- Thresh: This is the equivalent of the `threshold` parameter for `findLines()`. It sets how strong an edge has to be before a circle is recognized. The default value for this parameter is 350.
- Distance: Similar to the `maxlinegap` parameter for `findLines()`. It determines how close circles can be before they are treated as the same circle. If left undefined, the system tries to find the best value, based on the image being analyzed.

As with lines, there are `FeatureSet` functions that are more appropriate when dealing with circles, too. These functions include:

- `radius()`: As the name implies, this is the radius of the circle.
- `diameter()`: The diameter of the circle.
- `perimeter()`: This returns the perimeter of the feature, which in the case of a circle, is it's circumference. It may seem strange that this isn't called circumference, but the term perimeter makes more sense when you're dealing with a non-circular features. Using perimeter here then allows for a standardized naming convention.

To showcase how to use the `findCircle()` function, we'll take a look at an image of coffee mugs — where one coffee mug also happens to have a ping-pong ball in it. Since there isn't any beer in the mugs, we'll assume that someone is very good at the game

of beer pong — or perhaps is practicing for a later game? Either way, here's the example code:

```
from SimpleCV import Image

img = Image("pong.png")

circles = img.findCircle(canny=200,thresh=250,distance=15)  ❶

circles = circles.sortArea()  ❷

circles.draw(width=4)  ❸

circles[0].draw(color=Color.RED, width=4)  ❹

img_with_circles = img.applyLayers()  ❺

edges_in_image = img.edges(t2=200)  ❻

final = img.sideBySide(edges_in_image.sideBySide(img_with_circles)).scale(0.5)  ❼

final.show()
```

❶ Finds the circles in the image. We tested the values for the arguments we're using here to focus on the circles that we're interested in.

❷ The `sortArea()` function is used to sort the circles from the smallest one found to the largest. This lets us identify which circle is the ping-pong ball, since it will be the smallest one. It is possible to combine this step with the previous line too — so it would be `circles = img.findCircle(canny=200,thresh=250,distance=15).sortArea()`. We only separated this into two steps in this example to make it easier to follow.

❸ This draws all of the circles detected in the green default color. The default line width is a little tough to see, so we're passing in an argument to increase the width of the line to 4 pixels.

❹ This draws a circle in red around the smallest circle, which should be the ping-pong ball.

❺ The `draw()` function uses a drawing layer for the various circles. This line creates a new image which shows the original image with the drawn circles on top of it.

❻ In order to show you which edges the `findCircle()` function was working with, we use the `edges()` function to return an image of the edges found in the image. In order for the edges to be the same, we're passing in the same threshold value (200) to `edges()` as we did to `findCircle()`. For `edges()` this threshold parameter is called `t2`, but it's the same thing as the `canny` parameter for `findCircles()`.

❼ This combines the various images into a single image. The final result will show the original image, the image of the found edges, and then the original image with the circles drawn on top of it.

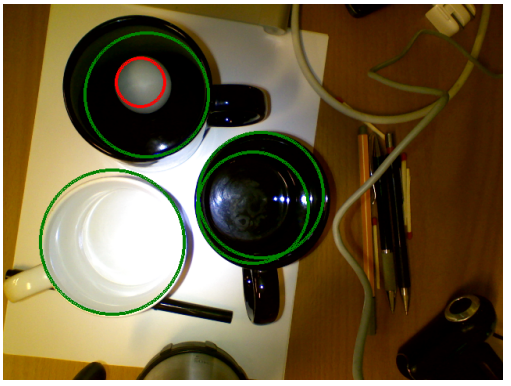


Figure 8-14. Image showing the detected circles

Figure 8-15. An analog dial like those found in older thermostats

The next example will combine finding both circle and line features by reading the line on a dial. These types of dials can be found in a variety of places, from large equipment to lighting controls around the house. For our example, we'll use the dial in Figure 8-15 with four different settings. The code will first search for the dial in the images, and then search for the line on each dial. Then it will measure the angle of that line, and print it on the image of the dial..

```
from SimpleCV import Image

# Load the images of 4 dials with different settings
img0 = Image("dial1.png")
img1 = Image("dial2.png")
img2 = Image("dial3.png")
img3 = Image("dial4.png")

# Store them in an array of images
```

```
images = (img0,img1,img2,img3)

# This will store the dial-only part of the images
dials = []

for img in images:
     circles = img.findCircle().sortArea()  ❶
     dial = circles[-1].crop()  ❷

     lines = dial.findLines(threshold=40,cannyth1=270,cannyth2=400)  ❸
     lines = lines.sortLength()  ❹
     lines[-1].draw(color=Color.RED)

     lineAngle = lines[-1].angle()  ❺

     if (lines[-1].x < (dial.width / 2)):  ❻
          if (lines[-1].y < (dial.width / 2)):
               lineAngle = lineAngle - 180
          else:
               lineAngle = 180 + lineAngle

     dial.drawText(str(lineAngle),10,10)
     dial = dial.applyLayers()
     dials.append(dial)

result = dials[0].sideBySide(dials[1].sideBySide(dials[2].sideBySide(dials[3])))  ❼
result.show()
```

❶ The first step is to find the dial on the image using the `findCircle()` function. The `sortArea()` function sorts the circles, and since the dial will be the largest circle, we know that it is `circles[-1]`, the last one in the list.

❷ Calling the `crop()` function on a feature will crop the image to just the area of the circle. This then stores the cropped image in the `dial` variable.

❸ Now call the `findLines()` function on the cropped image (stored in `dial`).

❹ The `sortLength()` function sorts the list of lines based on their length. Since dial's indicator line is most likely the longest line in the image, this will make it easy to identify it in the image.

❺ The `angle()` function computes the angle of a line. This is the angle between the line and a horizontal axis connected at the leftmost point of the line.

❻ The angles computed in the previous step are correct when the line is on the right side of the dial, but are incorrect for our purposes when it's on the left side. This is because the angle is calculated from the leftmost point of the line, and not the center of the dial. This block of code compensates for this by determining which quadrant the line is in on the left side, and then either adding or subtracting 180 degrees to show the angle as it if were being calculated from the center of the dial instead.

❼ After looping through all of the dials, create a single image with the results side by side.

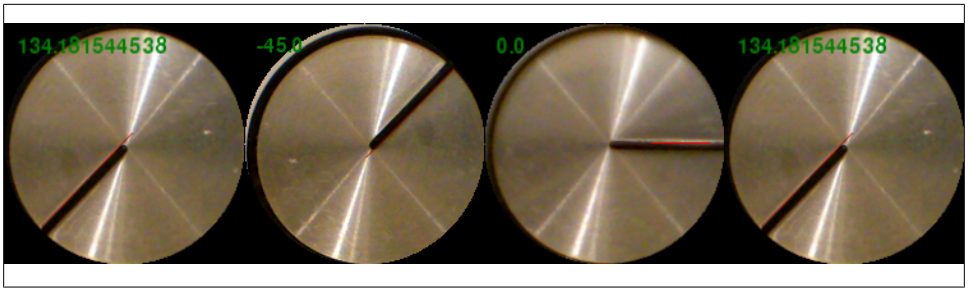


Figure 8-16. The results of the analog dial reader.

Corners

Roughly speaking, corners are places in an image where two lines meet. Corners are interesting in terms of computer vision because corners, unlike edges, are relatively unique and good for identifying parts of an image. For instance, if you were trying to analyze a square, if you tried finding the lines that represent the sides of the square, you could find two horizontal lines and two vertical lines for each of the sides. However, you wouldn't be able to tell which of the two horizontal lines was the top or the bottom, or which of the two vertical lines was the left or right side. Each corner, on the other hand, is unique, so you can easily identify where it is located on the square.

The `findCorners()` function analyzes an image and returns the locations of all of the corners it can find. Note that a corner does not need to be a right angle at 90 degrees. Two intersecting lines at any angle can constitute a corner. As with `findLines()` and `findCircle()`, the `findCorners()` method returns a `FeatureSet` of all of the corner features it finds. While a corners `FeatureSet` shares the same functions as any other `FeatureSet`, there are functions that wouldn't make much sense in the context of a corner. For example, trying to find the width, length, or area of a corner doesn't really make much sense. Technically, the functions used to find these things will still work, but what they'll return are default values and not real data about the corners.

Similar to the `findLines()` and `findCircle()` functions, `findCorners()` also has parameters to help fine tune which corners are found in an image. We'll walk through the available parameters using an image of a bracket as an example.

```
from SimpleCV import Image

img = Image('corners.png')

img.findCorners.show()
```

The little green circles represent the detected corners. Notice that the example finds a lot of corners. By default it looks for 50, which is obviously picking up a lot of false positives. Based on visual inspection, it appears that there are four main corners. To restrict the number of corners returned, we can use the `maxnum` parameter.

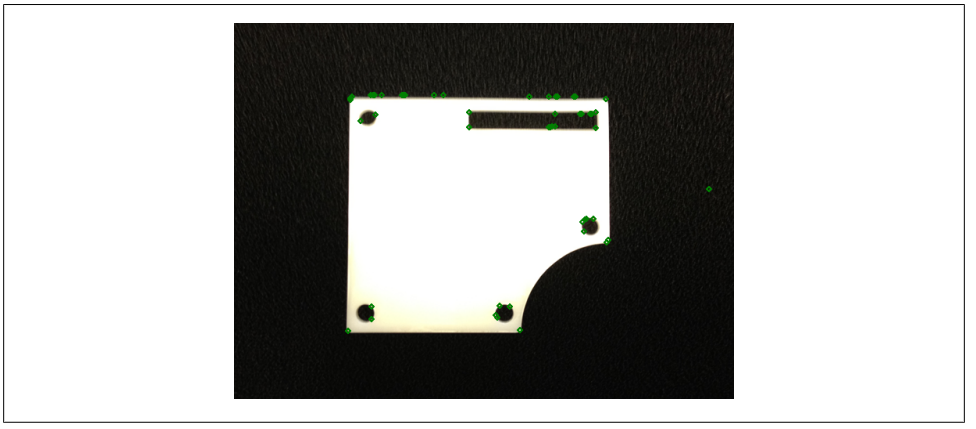


Figure 8-17. The corners found on a part

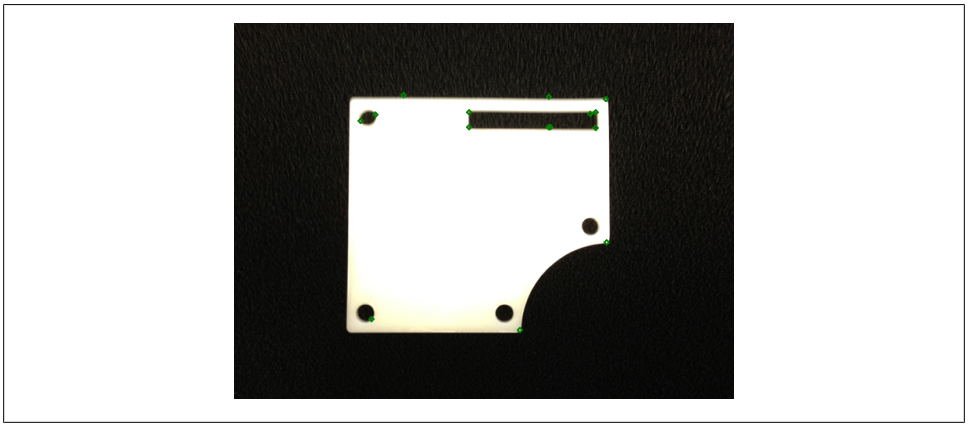


Figure 8-18. Limiting `findCorners()` to a maximum of nine corners.

```
from SimpleCV import Image

img = Image('corners.png')

img.findCorners.(maxnum=9).show()
```

The `findCorners()` method sorts all of the corners prior to returning its results, so if it finds more than the maximum bumber of corners, it will return only the best ones. Alternatively, the `minquality` parameter sets the minimum quality of a corner before it is shown. This approach filters out the noise without having to hard code a maximum number of corners for an object.

This is getting better, but the algorithm found two corners in the lower left and zero in the upper right. The two in the lower left are really part of the same corner, but the lighting and colors are confusing the algorithm. To prevent nearby corners from being

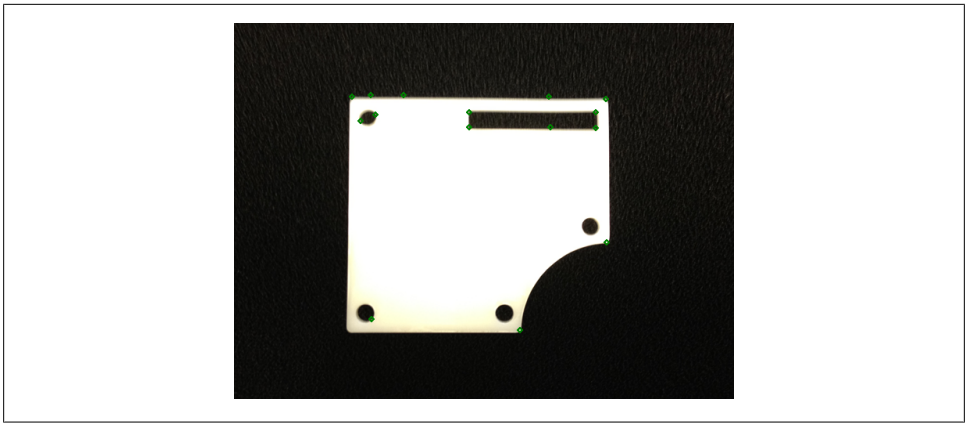


Figure 8-19. The corners when each corner is at least 10 pixels away from one another.

counted as two separate corners, set the `mindistance` parameter, which sets the minimum number of pixels between two corners.

```
from SimpleCV import Image

img = Image('corners.png')

img.findCorners.(maxnum=9, mindistance=10).show()
```

Examples

In this example, we take a photo of some US coins and calculate what their total value is. To do this, we find the blobs that represent each coin, and then compare them a table of their diameters to look up each coin's value. In the example, we presume we have a quarter among the coins to act as a reference object.The photo of the coins is showin in [to come].

```
from SimpleCV import Image, Blob
import numpy as np

img = Image("coins.png")
coins = img.invert().findBlobs(minsize = 500) ❶
value = 0.0

# The value of the coins in order of their size
# http://www.usmint.gov/about_the_mint/?action=coin_specifications
coin_diameter_values = np.array([
    [ 19.05, 0.10],
    [ 21.21, 0.01],
    [ 17.91, 0.05],
    [ 24.26, 0.25]]); ❷

#use a quarter to calibrate (in this example we must have one)
px2mm = coin_diameter_values[3,0] /  max([c.radius()*2 for c in coins]) ❸
```

```
for c in coins:
  diameter_in_mm = c.radius() * 2 * px2mm
  distance = np.abs(diameter_in_mm - coin_diameter_values[:,0]) ❹
  index = np.where(distance == np.min(distance))[0][0] ❺
  value += coin_diameter_values[index, 1] ❻

print "The total value of the coins is $", value
```

❶ We invert the image to make it easy to find the darker blobs that represent the coins, and eliminate small noisy objects by setting a minimum size.

❷ The `coin_diameter_values` array represents our knowledge about the relative sizes of the coins versus their values. The first entry in the array is for the smallest US coin, a dime. The next largest coin is a penny, then a nickel, a finally a quarter. (This could easily be extended to include half dollars or dollar coins, based on their size.)

❸ We get a sense of scale by taking the largest diameter in our FeatureSet and assuming it's a quarter, similar to the Quarter for Scale example

❹ This generates a list of distances our blob radius is from each coin. Notice that it is slicing the diameter column of the value lookup table

❺ This finds the index of which ideal coin the target is closest to (minimizing the difference)

❻ Add the value from the diameter/value table to our total value.

When running this code using our example photo, the total value of the coins pictured should be $0.91. In this example, the basic detection was fairly straightforward, but we spent most of the time analyzing the data we gathered from the image. In the following chapter, we'll delve further into these techniques.