Practical Python and OpenCV

An Introductory, Example Driven Guide to Image Processing and Computer Vision

Adrian Rosebrock

pyimagesearch

Practical Python and OpenCV: An Introductory, Example Driven Guide to Image Processing and Computer Vision

Adrian Rosebrock

COPYRIGHT

This version of the book was published on 22 September 2014.

Books like this are made possible by the time investment made by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at http://www.pyimagesearch.com/practical-python-opencv/ today.

CONTENTS

PREFACE

When I first set out to write this book, I wanted it to be as hands-on as possible. I wanted lots of visual examples with lots of code. I wanted to write something that you could easily learn from, without all the rigor and detail of mathematics associated with college level computer vision and image processing courses.

I know that from all my years spent in the classroom that the way I learned best was from simply opening up an editor and writing some code. Sure, the theory and examples in my textbooks gave me a solid starting point. But I never really "learned" something until I did it myself. I was very hands on. And that's exactly how I wanted this book to be. Very hands on, with all the code easily modifiable and well documented so you could play with it on your own. That's why I'm giving you the full source code listings and images used in this book.

More importantly, I wanted this book to be accessible to a wide range of programmers. I remember when I first started learning computer vision – it was a daunting task. But I learned a lot. And I had a lot of fun.

I hope this book helps you in your journey into computer vision. I had a blast writing it. If you have any questions, suggestions or comments, or if you simply want to say hello, shoot me an email at adrian@pyimagesearch.com, or

you can visit my website at www.PyImageSearch.com and leave a comment. I look forward to hearing from you soon!

-Adrian Rosebrock

PREREQUISITES

In order to make the most of this, you will need to have a little bit of programming experience. All examples in this book are in the Python programming language. Familiarity, with Python, or other scripting languages is suggested, but not required.

You'll also need to know some basic mathematics. This book is hands-on and example driven: lots of examples and lots of code, so even if you math skills are not up to par, do not worry! The examples are very detailed and heavily documented to help you follow along.

CONVENTIONS USED IN THIS BOOK

This book includes many code listings and terms to aide you in your journey to learn computer vision and image processing. Below are the typographical conventions used in this book:

Italic

Indicates key terms and important information that you should take note of. May also denote mathematical equations or formulas based on connotation.

Bold

Important information that you should take note of.

`Constant width`

Used for source code listings, as well as paragraphs that make reference to the source code, such as function and method names.

USING THE CODE EXAMPLES

This book is meant to be a hands-on approach to computer vision and machine learning. The code included in this book, along with the source code distributed with this book, are free for you to modify, explore, and share, as you wish.

In general, you do not need to contact me for permission if you are using the source code in this book. Writing a script that uses chunks of code from this book is totally and completely okay with me.

However, selling or distributing the code listings in this book, whether as information product or in your product's documentation *does* require my permission.

If you have any questions regarding the fair use of the code examples in this book, please feel free to shoot me an email. You can reach me at adrian@pyimagesearch.com.

HOW TO CONTACT ME

Want to find me online? Look no further:

Website:	www.PyImageSearch.com
Email:	adrian@pyimagesearch.com
Twitter:	@PyImageSearch
Google+:	+AdrianRosebrock
LinkedIn:	Adrian Rosebrock

1

INTRODUCTION

The goal of computer vision is to understand the story unfolding in a picture. As humans, this is quite simple. But for computers, the task is extremely difficult.

So why bother learning computer vision?

Well, images are everywhere!

Whether it be personal photo albums on your smartphone, public photos on Facebook, or videos on YouTube, we now have more images than ever – and we need methods to analyze, categorize, and quantify the contents of these images.

For example, have you recently tagged a photo of yourself or a friend on Facebook lately? How does Facebook seem to "know" where the faces are in an image?

Facebook has implemented facial recognition algorithms into their website, meaning that they can not only *find* faces in an image, but they can also *identify* whose face it is as well! Facial recognition is an application of computer vision in the real-world.

What other types of useful applications of computer vision are there?

Well, we could build representations of our 3D world using public image repositories like Flickr. We could download thousands and thousands of pictures of Manhattan, taken by citizens with their smartphones and cameras, and then analyze them and organize them to construct a 3D representation of the city. We would then virtually navigate this city through our computers. Sound cool?

Another popular application of computer vision is surveillance.

While surveillance tends to have a negative connotation of sorts, there are many different types of surveillance. One type of surveillance is related to analyzing security videos, looking for possible suspects after a robbery.

But a different type of surveillance can be seen in the retail world. Department stores can use calibrated cameras to track how you walk through their stores and which kiosks you stop at.

On your last visit to your favorite clothing retailer, did you stop to examine the spring's latest jean trends? How long did you look at the jeans? What was your facial expression as you looked at the jeans? Did you then pickup a pair and head to the dressing room? These are all types of questions that computer vision surveillance systems can answer.

Computer vision can also be applied to the medical field. A year ago, I consulted with the National Cancer Institute to develop methods to automatically analyze breast histology images for cancer risk factors. Normally, a task like this would require a trained pathologist with years of experience – and it would be extremely time consuming!

Our research demonstrated that computer vision algorithms could be applied to these images and automatically analyze and quantify cellular structures – without human intervention! Now that we can analyze breast histology images for cancer risk factors much faster.

Of course, computer vision can also be applied to other areas of the medical field. Analyzing X-Rays, MRI scans, and cellular structures all can be performed using computer vision algorithms.

Perhaps the biggest success computer vision success story you may have heard of is the X-Box 360 Kinect. The Kinect can use a stereo camera to understand the depth of an image, allowing it to classify and recognize human poses, with the help of some machine learning, of course.

The list doesn't stop there.

Computer vision is now prevalent in many areas of your life, whether you realize it or not. We apply computer vision algorithms to analyze movies, football games, hand gesture recognition (for sign language), license plates (just in case you were driving too fast), medicine, surgery, military, and retail.

We even use computer visions in space! NASA's Mars Rover includes capabilities to model the terrain of the planet, detect obstacles in its path, and stitch together panorama images.

This list will continue to grow in the coming years.

Certainly, computer vision is an exciting field with endless possibilities.

With this in mind, ask yourself, what does your imagination want to build? Let it run wild. And let the computer vision techniques introduced in this book help you build it.

2

PYTHON AND REQUIRED PACKAGES

In order to explore the world of computer vision, we'll first need to install some packages. As a first timer in computer vision, installing some of these packages (especially OpenCV) can be quite tedious, depending on what operating system you are using. I've tried to consolidate the installation instructions into a short how-to guide, but as you know, projects change, websites change, and installation instructions change! If you run into problems, be sure to consult the package's website for the most up to date installation instructions.

I highly recommend that you use either easy_install or pip to manage the installation of your packages. It will make your life much easier!

Finally, if you don't want to undertake installing these packages, I have put together an Ubuntu virtual machine with all packages pre-installed! Using this virtual machine allows you to jump right in to the examples in this book, without having to worry about package managers, installation instructions, and compiling errors.

To find out more about this this pre-configured virtual machine, head on over to: http://www.pyimagesearch.com/practical-python-opencv/.

Now, let's install some packages!

2.1 NUMPY AND SCIPY

NumPy is a library for the Python programming language that (among other things) provides support for large, multi-dimensional arrays. Why is that important? Using NumPy, we can express images as multi-dimensional arrays. Representing images as NumPy arrays is not only computationally and resource efficient, but many other image processing and machine learning libraries use NumPy array representations as well. Furthermore, by using NumPy's built-in high-level mathematical functions, we can quickly perform numerical analysis on an image.

Going hand-in-hand with NumPy, we also have SciPy. SciPy adds further support for scientific and technical computing.

2.1.1 *Windows*

By far, the easiest way to install NumPy and SciPy on your Windows system is to download and install the binary distribution from: http://www.scipy.org/install.html.

2.1.2 *OSX*

If you are running OSX 10.7.0 (Lion) or above, NumPy and SciPy come pre-installed.

However, I like to install the `ScipySuperpack`. It includes the latest versions of NumPy, SciPy, Matplotlib, and other extremely useful packages such as `ipython`, `pandas`, and `scikit-learn`. All of these packages are worth installing, and if you read my blog over at www.PyImageSearch.com, you'll see that I make use of these libraries quite often.

2.1.3 *Linux*

On many Linux distributions, such as Ubuntu, NumPy comes pre-installed and configured.

If you want the latest versions of NumPy and Scipy, you can build the libraries from source, but the easiest method is to use a package manager, such as `apt-get` on Ubuntu.

2.2 MATPLOTLIB

Simply put, matplotlib is a plotting library. If you've ever used MATLAB before, you'll probably feel very comfortable in the matplotlib environment. When analyzing images, we'll make use of matplotlib, whether plotting image histograms or simply viewing the image itself, matplotlib is a great tool to have in your toolbox.

2.2.1 *All Platforms*

Matplotlib is available from http://matplotlib.org/. If you have already installed the ScipySuperpack, then you already have Matplotlib installed. You can also install it by using `pip` or `easy_install`.

Otherwise, a binary installer is provided for Windows.

2.3 OPENCV

If NumPy's main goal is large, efficient, multi-dimensional array representations, then, by far, the main goal of OpenCV is real-time image processing. This library has been around since 1999, but it wasn't until the 2.0 release in 2009 did we see the incredible NumPy support. The library itself is written in C/C++, but Python bindings are provided when running the installer. OpenCV is hands down my favorite computer vision library and we'll use it a lot in this book.

The installation for OpenCV is constantly changing. Since the library is written in C/C++, special care has to be taken when compiling and ensuring the prerequisites are installed. Be sure to check the OpenCV website at http://opencv.org/ for the latest installation instructions since they do (and will) change in the future.

2.3.1 *Windows and Linux*

The OpenCV Docs provide fantastic tutorials on how to install OpenCV in Windows and Linux using binary distributions. You can check out the install instructions here: http://docs.opencv.org/doc/tutorials/introduction/table_of_content_introduction/table_of_content_introduction.html#table-of-content-introduction.

2.3.2 *OSX*

Installing OpenCV in OSX has been a pain in previous years, but has luckily gotten much easier with `brew`. Go ahead and download and install `brew` from http://brew.sh/, a package manager for OSX. It's guaranteed to make your life easier in more ways than one.

After `brew` is installed, all you need to do is follow a few simple commands. In general, I find that Jeffery Thompson's instructions on how to install OpenCV on OSX to be phenomenal and an excellent starting point.

You can find the instructions here: http://www.jeffreythompson.org/blog/2013/08/22/update-installing-opencv-on-mac-mountain-lion/.

2.4 MAHOTAS

Mahotas, just as OpenCV, relies on NumPy arrays. Much of the functionality implemented in Mahotas can be found

in OpenCV but in some cases, the Mahotas interface is just easier to use. We'll use it to complement OpenCV.

2.4.1 *All Platforms*

Installing Mahotas is extremely easy on all platforms. Assuming you already have NumPy and SciPy installed, all you need is `pip install mahotas` or `easy_install mahotas`.

Now that we have all our packages installed, let's start exploring the world of computer vision!

2.5 SKIP THE INSTALLATION

As I've mentioned above, installing all these packages can be time consuming and tedious. If you want to skip the installation process and jump right in to the world of image processing and computer vision, I have setup a preconfigured Ubuntu virtual machine with all of the above libraries mentioned installed.

If you are interested and downloading this virtual machine (and saving yourself a lot of time and hassle), you can head on over to http://www.pyimagesearch.com/practical-python-opencv/.

3

LOADING, DISPLAYING, AND SAVING

This book is meant to be a hands on, how-to guide to getting started with computer vision using Python and OpenCV. With that said, let's not waste any time. Let's get our feet wet by writing some simple code to load an image off disk, display it on our screen, and write it to file in a different format. When executed, our Python script should show our image on screen, like in **Figure 3.1**.

First, let's create a file named `load_display_save.py` to contain our code. Now we can start writing some code:

Listing 3.1: load_display_save.py

```
1 import argparse
2 import cv2
3
4 ap = argparse.ArgumentParser()
5 ap.add_argument("-i", "--image", required = True,
6     help = "Path to the image")
7 args = vars(ap.parse_args())
```

The first thing we are going to do is import the packages we will need for this example. We use `argparse` to handle parsing our command line arguments. Then, `cv2` is imported – `cv2` is our OpenCV library and contains our

Figure 3.1: Example of loading and displaying a *Tyrannosaurus Rex* image on our screen.

image processing functions.

From there, **Lines 4-7** handle parsing the command line arguments. The only argument we need is `--image`: the path to our image on disk. Finally, we parse the arguments and store them in a dictionary.

Listing 3.2: load_display_save.py

```
8 image = cv2.imread(args["image"])
9 print "width: %d pixels" % (image.shape[1])
10 print "height: %d pixels" % (image.shape[0])
11 print "channels: %d" % (image.shape[2])
12
13 cv2.imshow("Image", image)
14 cv2.waitKey(0)
```

Now that we have the path to the image, we can load it off disk using the `cv2.imread` function on **Line 8**. The `cv2.imread` function returns a NumPy array representing the image.

Lines 9-11 examine the dimensions of the image. Again, since images are represented as NumPy arrays, we can simply use the `shape` attribute to examine the width, height, and the number of channels.

Finally, **Lines 13 and 14** handle displaying the actual image on our screen. The first parameter is a string, the "name" of our window. The second parameter is a reference to the image we loaded off disk on **Line 8**. Finally, a call to `cv2.waitKey` pauses the execution of the script until we press a key on our keyboard. Using a parameter of `0` indicates that any keypress will un-pause the execution.

The last thing we are going to do is write our image to file in JPG format:

Listing 3.3: load_display_save.py

```
15    cv2.imwrite("newimage.jpg", image)
```

All we are doing here is providing the path to the file (the first argument) and then the image we want to save (the second argument). It's that simple.

To run our script and display our image, we simply open up a terminal window and execute the following command:

Listing 3.4: load_display_save.py

```
$ python load_display_save.py --image ../images/trex.png
```

If everything has worked correctly you should see the *T-Rex* on your screen as in **Figure 3.1**. To stop the script from executing, simply click on the image window and press any key.

Examining the the output of the script, you should also see some basic information on our image. You'll note that the image has width of 350 pixels, a height of 228 pixels, and 3 channels (the RGB components of the image). Represented as a NumPy array, our image has a shape of `(350,228,3)`.

When we write matrices, it is common to write them in the form *(# of rows* $\times$ *# of columns)* – this is not the case for NumPy. NumPy actually gives you the number of columns, then the number of rows. This is important to keep in mind.

Finally, note the contents of your directory. You'll see a new file there: `newimage.jpg`. OpenCV has automatically converted our PNG image to JPG for us! No further effort is needed on our part to convert between image formats.

Next up, we'll explore how to access and manipulate the pixel values in an image.

4

IMAGE BASICS

In this chapter we are going to review the building blocks of an image – the pixel. We'll discuss exactly what a pixel is, how pixels are used to form an image, and then how to access and manipulate pixels in OpenCV.

4.1 SO, WHAT'S A PIXEL?

Every image consists of a set of pixels. Pixels are the raw, building blocks of an image. There is no finer granularity than the pixel.

Normally, we think of a pixel as the "color" or the "intensity" of light that appears in a given place in our image.

If we think of an image as a grid, each square in the grid contains a single pixel.

For example, let's pretend we have an image with a resolution of 500×300. This means that our image is represented as a grid of pixels, with 500 rows and 300 columns. Overall, there are $500 \times 300 = 150,000$ pixels in our image.

Most pixels are represented in two ways: grayscale and color. In a grayscale image, each pixel has a value between 0 and 255, where zero is corresponds to "black" and 255 being "white". The values in between 0 and 255 are varying shades of gray, where values closer to 0 are darker and values closer 255 are lighter.

Color pixels are normally represented in the RGB color space – one value for the Red component, one for Green, and one for Blue. Other color spaces exist, but let's start with the basics and move our way up from there.

Each of the three colors are represented by an integer in the range 0 to 255, which indicates how "much" of the color there is. Given that the pixel value only needs to be in the range $[0, 255]$ we normally use an 8-bit unsigned integer to represent each color intensity.

We then combine these values into a RGB tuple in the form `(red, green, blue)`. This tuple represents our color.

To construct a white color, we would fill each of the red, green, and blue buckets completely up, like this: `(255,255, 255)`.

Then, to create a black color, we would empty each of the buckets out: `(0,0,0)`.

To create a pure red color, we would fill up the red bucket (and only the red bucket) up completely: `(255,0,0)`.

Are you starting to see a pattern?

For your reference, here are some common colors represented as RGB tuples:

- **Black:** `(0,0,0)`
- **White:** `(255,255,255)`
- **Red:** `(255,0,0)`
- **Green:** `(0,255,0)`
- **Blue:** `(0,0,255)`
- **Aqua:** `(0,255,255)`
- **Fuchsia:** `(255,0,255)`
- **Maroon:** `(128,0,0)`
- **Navy:** `(0,0,128)`
- **Olive:** `(128,128,0)`
- **Purple:** `(128,0,128)`
- **Teal:** `(0,128,128)`
- **Yellow:** `(255,255,0)`

Now that we have a good understanding of pixels, let's have a quick review of the coordinate system.

4.2 OVERVIEW OF THE COORDINATE SYSTEM

As I mentioned above, an image is represented as a grid of pixels. Imagine our grid as a piece of graph paper. Using this graph paper, the point $(0, 0)$ corresponds to the upper left corner of the image. As we move down and to the right, both the x and y values increase.

Let's take a look at the image in Figure 4.1 to make this point more clear.

Here we have the letter "I" on a piece of graph paper. We see that we have an 8×8 grid with 64 total pixels.

The point at $(0, 0)$ corresponds to the top left pixel in our image, whereas the point $(7, 7)$ corresponds to the bottom right corner.

Finally, the point $(3, 4)$ is the pixel three columns to the right, and four rows down, once again keeping in mind that we start counting from *zero* rather than *one*.

It is important to note that we are count from *zero* rather than *one*. The Python language is *zero indexed*, meaning that we always start counting from zero. Keep this mind and you'll avoid a lot of confusion later on.

4.3 ACCESSING AND MANIPULATING PIXELS

Admittedly, the example from Chapter 3 wasn't very exciting. All we did was load an image off disk, display it, and

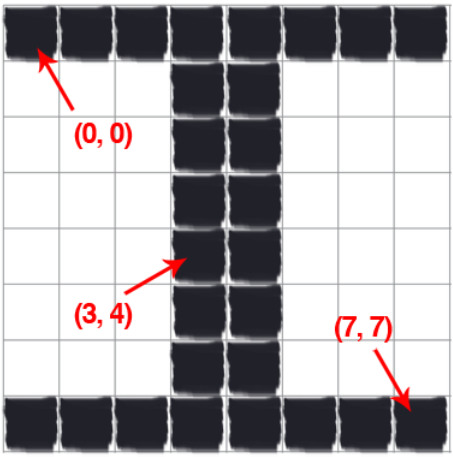

Figure 4.1: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x, y) coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed: we start counting from zero rather than one.

then write it back to disk in a different image file format.

Let's do something a little more exciting and see how we can access and manipulate the pixels in an image:

Listing 4.1: getting_and_setting.py

```
1 import argparse
2 import cv2
3
4 ap = argparse.ArgumentParser()
5 ap.add_argument("-i", "--image", required = True,
6     help = "Path to the image")
7 args = vars(ap.parse_args())
8
9 image = cv2.imread(args["image"])
10 cv2.imshow("Original", image)
```

Similar to our example in the previous chapter, **Lines 1-7** handle importing the packages we need along with setting up our argument parser. There is only one command line argument needed: the path to the image we are going to work with.

Lines 9 and 10 handle loading the actual image off of disk and displaying it to us.

So now that we have the image loaded, how can we access the actual pixel values?

Remember, OpenCV represents images as NumPy arrays. Conceptually, we can think of this representation as a matrix, as discussed in Section 4.1 above. In order to access a pixel value, we just need to supply the x and y coordinates of the pixel we are interested in. From there, we are given a tuple representing the Red, Green, and Blue components

of the image.

However, it's important to note that OpenCV stores RGB channels in *reverse order*. While we normally think in terms of Red, Green, and Blue, OpenCV actually stores them in the order of Blue, Green, and Red. This is **important to note** since it could cause some confusion later.

Alright, let's explore some code that can be used to access and manipulate pixels:

Listing 4.2: getting_and_setting.py

```
11 (b, g, r) = image[0, 0]
12 print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b
       )
13
14 image[0, 0] = (0, 0, 255)
15 (b, g, r) = image[0, 0]
16 print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b
       )
```

On **Line 11**, we grab the pixel located at $(0, 0)$ – the top-left corner of the image. This pixel is represented as a tuple. Again, OpenCV stores RGB pixels in *reverse order*, so when we unpack and access each element in the tuple, we are actually viewing them in BGR order. Then, **Line 12** then prints out the values of each channel to our console.

As you can see, accessing pixel values is quite easy! NumPy takes care of all the hard work for us. All we are doing are providing indexes into the array.

Just as NumPy makes it easy to *access* pixel values, it also makes it easy to *manipulate* pixel values.

On **Line 14** we manipulate the top-left pixel in the image, which is located at coordinate $(0, 0)$ and set it to have a value of `(0, 0, 255)`. If we were reading this pixel value in RGB format, we would have a value of `0` for red, `0` for green, and `255` for blue, thus making it a pure blue color.

However, as I mentioned above, we need to take special care when working with OpenCV. Our pixels are actually stored in BGR format, **not** RGB format.

We actually read this pixel as `255` for red, `0` for green, and `0` for blue, making it a red color, *not* a blue color.

After setting the top-left pixel to have a red color on **Line 14**, we then grab the pixel value and print it back to console on **Lines 15 and 16**, just to demonstrate that we have indeed successfully changed the color of the pixel.

Accessing and setting a single pixel value is simple enough, but what if we wanted to use NumPy's array slicing capabilities to access larger rectangular portions of the image? The code below demonstrates how we can do this:

Listing 4.3: getting_and_setting.py

```
17 corner = image[0:100, 0:100]
18 cv2.imshow("Corner", corner)
19
20 image[0:100, 0:100] = (0, 255, 0)
21
22 cv2.imshow("Updated", image)
23 cv2.waitKey(0)
```

On **line 17** we grab a 100×100 pixel region of the image. In fact, this is the top-left corner of the image! In order to grab chunks of an image, NumPy expects we provide four

indexes:

1. **Start y:** The first value is the starting y coordinate. This is where our array slice will start along the y-axis. In our example above, our slice starts at $y = 0$.
2. **End y:** Just as we supplied a starting y value, we must provide an ending y value. Our slice stops along the y-axis when $y = 100$.
3. **Start x:** The third value we must supply is the starting x coordinate for the slice. In order to grab the top-left region of the image, we start at $x = 0$.
4. **End x:** Lastly, we need to provide a x-axis value for our slice to stop. We stop when $x = 100$.

Once we have extracted the top-left corner of the image, **Line 18** shows us the result of the cropping. Notice how our image is just the 100×100 pixel region from the top-left corner of our original image.

The last thing we are going to do is use array slices to change the color of a region of pixels. On **Line 20**, you can see that we are again accessing the top-left corner of the image; however, this time we are setting this region to have a value of `(0, 255, 0)` (green).

Lines 22 and 23 then show us the results of our work.

So how do we run our Python script?

Assuming you have downloaded the source code listings for this book, simply navigate to the `chapter-04` directory

and execute the command below:

Listing 4.4: getting_and_setting.py

```
$ python getting_and_setting.py --image ../images/trex.png
```

Once our script starts running, you should see some output printed to your console (**Line 12**). The first line of output tells us that the pixel located at $(0, 0)$ has a value of 254 for all three red, green, and blue channels. This pixel appears to be almost pure white.

The second line of output shows us that we have successfully changed the pixel located at $(0, 0)$ to be red rather than white (**Lines 14-16**).

Listing 4.5: getting_and_setting.py

```
Pixel at (0, 0) - Red: 254, Green: 254, Blue: 254
Pixel at (0, 0) - Red: 255, Green: 0, Blue: 0
```

We can see the results of our work in Figure 4.2. The *Top-Left* image is our original image we loaded off disk. The image on the *Top-Right* is the result of our array slicing and cropping out a 100×100 pixel region of the image. And, if you look closely, you can see that the top-left pixel located at $(0, 0)$ is red!

Finally, the *bottom* image shows that we have successfully drawn a green square on our image.

In this chapter we have explored how to access and manipulate the pixels in an image using NumPy's built-in array slicing functionality. We were even able to draw a green

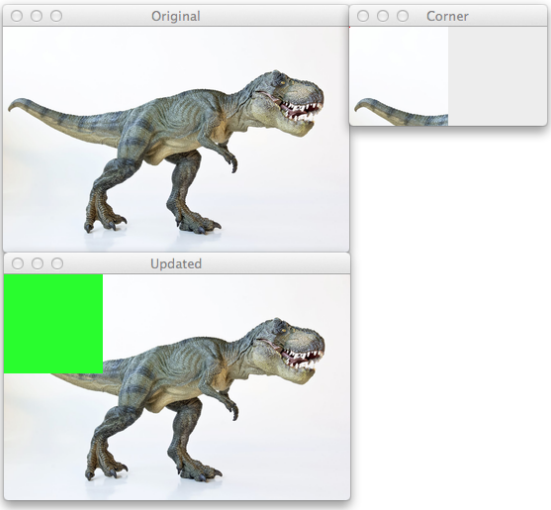

Figure 4.2: *Top-Left*: Our original image. *Top-Right*: Cropping our image using NumPy array slicing. *Bottom*: Drawing a 100×100 pixel green square on our image by using basic NumPy indexing.

square using nothing but NumPy array manipulation!

However, we won't get very far using only NumPy functions. The next chapter will show you how to draw lines, rectangles, and circles using OpenCV methods.

5

DRAWING

Using NumPy array slices in Chapter 4 we were able to draw a green square on our image. But what if we wanted to draw a single line? Or a circle? NumPy does not provide that type of functionality – it's only a numerical processing library after all!

Luckily, OpenCV provides convenient, easy to use methods to draw shapes on an image. In this chapter, we'll review the three most basic methods to draw shapes: `cv2.line`, `cv2.rectangle`, and `cv2.circle`.

While this chapter is by no means a complete, exhaustive overview of the drawing capabilities of OpenCV, it will none-the-less provide a quick, hands-on approach to get you started drawing immediately.

5.1 LINES AND RECTANGLES

Before we start exploring the the drawing capabilities of OpenCV, let's first define our canvas in which we will draw our masterpieces.

Up until this point, we have only loaded images off of disk. However, we can also define our images manually using NumPy arrays. Given that OpenCV interprets an image as a NumPy array, there is no reason why we can't manually define the image ourselves!

In order to initialize our image, let's examine the code below:

Listing 5.1: drawing.py

```
1 import numpy as np
2 import cv2
3
4 canvas = np.zeros((300, 300, 3), dtype = "uint8")
```

Lines 1 and 2 imports the packages we will be using. As a shortcut, we'll create an alias for `numpy` as `np`. We'll continue this convention throughout the rest of the book. In fact, you'll commonly see this convention in the Python community as well! We'll also import `cv2` so we can have access to the OpenCV library.

Initializing our image is handled on **Line 4**. We construct a NumPy array using the `np.zeros` method with 300 rows and 300 columns, yielding a 300×300 pixel image. We also allocate space for 3 channels – one for Red, Green, and Blue, respectively. As the name suggests, the `zeros` method fills every element in the array with an initial value of zero.

It's important to draw your attention to the second argument of the `np.zeros` method: the data type, `dtype`. Since we are representing our image as a RGB image with pixels in the range $[0, 255]$, it's important that we use an 8-bit unsigned integer, or `uint8`. There are many other data types

that we can use (common ones include 32-bit integers, and 32-bit or 64-bit floats), but we'll mainly be using `uint8` for the majority of the examples in this book.

Now that we have our canvas initialized, we can do some drawing:

Listing 5.2: drawing.py

```
5 green = (0, 255, 0)
6 cv2.line(canvas, (0, 0), (300, 300), green)
7 cv2.imshow("Canvas", canvas)
8 cv2.waitKey(0)
9
10 red = (0, 0, 255)
11 cv2.line(canvas, (300, 0), (0, 300), red, 3)
12 cv2.imshow("Canvas", canvas)
13 cv2.waitKey(0)
```

The first thing we do on **Line 5** is define a tuple used to represent the color "green". Then, we draw a green line from point $(0, 0)$ (the top-left corner of the image) to point $(300, 300)$, the bottom-right corner of the image on **Line 6**.

In order to draw the line, we make use of the `cv2.line` method. The first argument to this method is the image we are going to draw on. In this case, it's our `canvas`. The second argument is the starting point of the line. We choose to start our line from the top-left corner of the image, at point $(0, 0)$. We also need to supply an ending point for the line (the third argument). We define our ending point to be $(300, 300)$, the bottom-right corner of the image. The last argument is the color of our line, in this case green. **Lines 7 and 8** show our image and then wait for a keypress.

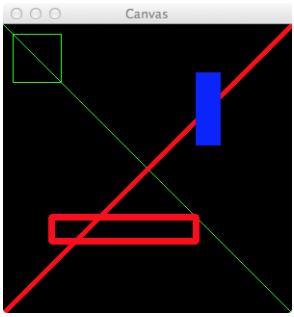

Figure 5.1: Examples of drawing lines and rectangles using OpenCV.

As you can see, drawing a line is quite simple! But there is one other important argument to consider in the `cv2.line` method: the thickness.

On **Lines 10-13** we define a red color as a tuple (again, in BGR rather than RGB format). We then draw a red line from the top-right corner of the image to the bottom left. The last parameter to the method controls the thickness of the line – we decide to make the thickness 3 pixels. Again, we show our image and wait for a keypress.

Drawing a line was simple enough. Now we can move on to drawing rectangles. Check out the code below for more details:

Listing 5.3: drawing.py

```
14 cv2.rectangle(canvas, (10, 10), (60, 60), green)
15 cv2.imshow("Canvas", canvas)
```

```
16 cv2.waitKey(0)
17
18 cv2.rectangle(canvas, (50, 200), (200, 225), red, 5)
19 cv2.imshow("Canvas", canvas)
20 cv2.waitKey(0)
21
22 blue = (255, 0, 0)
23 cv2.rectangle(canvas, (200, 50), (225, 125), blue, -1)
24 cv2.imshow("Canvas", canvas)
25 cv2.waitKey(0)
```

On **Line 14** we make use of the `cv2.rectangle` method. The signature of this method is identical to the `cv2.line` method above, but let's explore each argument anyway.

The first argument is the image in which we want to draw our rectangle on. We want to draw on our `canvas`, so we pass it into the method. The second argument is the starting (x, y) position of our rectangle – here we are starting our rectangle at point $(10, 10)$. Then, we must provide an ending (x, y) point for the rectangle. We decide to end our rectangle at $(60, 60)$, defining a region of 50×50 pixels. Finally, the last argument is the color of the rectangle we want to draw.

Just as we can control the thickness of a line, we can also control the thickness of a rectangle. **Line 18** provides one added argument: the thickness. Here, we draw a red rectangle that is 5 pixels thick, starting from point $(50, 200)$ and ending at $(200, 225)$.

At this point, we have only drawn the *outline* of a rectangle. How do we draw a rectangle that is "filled in", like when using NumPy array slices in Chapter 4?

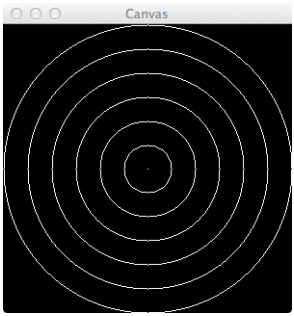

Figure 5.2: Drawing a simple bullseye with the `cv2.circle` function.

Simple. We just pass in a negative value for the thickness argument.

Line 23 demonstrates how to draw a rectangle of a solid color. We draw a blue rectangle, starting from $(200, 50)$ and ending at $(225, 125)$. By specifying `-1` as the thickness, our rectangle is drawn as a solid blue.

Congratulations! You now have a solid grasp of drawing rectangles. In the next section, we'll move on to drawing circles.

5.2 CIRCLES

Drawing circles is just as simple as drawing rectangles, but the function arguments are a little different. Let's go ahead and get started:

Listing 5.4: drawing.py

```
26 canvas = np.zeros((300, 300, 3), dtype = "uint8")
27 (centerX, centerY) = (canvas.shape[1] / 2, canvas.shape[0] / 2)
28 white = (255, 255, 255)
29
30 for r in xrange(0, 175, 25):
31     cv2.circle(canvas, (centerX, centerY), r, white)
32
33 cv2.imshow("Canvas", canvas)
34 cv2.waitKey(0)
```

On **Line 26** we re-initialize our `canvas` to be blank. The rectangles are gone! We need a fresh canvas to draw our circles.

Line 27 calculates two variables: `centerX` and `centerY`. These two variables represent the (x, y) coordinates of the center of the image. We calculate the center by examining the shape of our NumPy array, and then dividing by two. The height of the image can be found in `canvas.shape[0]` and the width in `canvas.shape[1]`. Finally, **Line 28** defines a white pixel.

Now, let's draw some circles!

On **Line 30** we loop over a number of radius values, starting from 0 and ending at 150, incrementing by 25 at each step..

Line 31 handles the actual drawing of the circle. The first parameter is our `canvas`, the image we want to draw the circle on. We then need to supply the point in which our circle will be drawn around. We pass in a tuple of `(centerX, centerY)` so that our circles will be centered at the center of the image. The third argument is the radius of the circle we wish to draw. Finally, we pass in the color of our circle,

in this case, white.

Lines 33 and 34 then show our image and wait for a keypress.

So what does our image look like?

Check out Figure 5.2 and you will see that we have drawn a simple bullseye! The "dot" in the very center of the image is drawn with a radius of `0`. The larger circles are drawn with every increasing radii sizes from our `for` loop.

Not too bad. But what else can we do?

Let's do some abstract drawing:

Listing 5.5: drawing.py

```
35 for i in xrange(0, 25):
36     radius = np.random.randint(5, high = 200)
37     color = np.random.randint(0, high = 256, size = (3,)).tolist
           ()
38     pt = np.random.randint(0, high = 300, size = (2,))
39
40     cv2.circle(canvas, tuple(pt), radius, color, -1)
41
42 cv2.imshow("Canvas", canvas)
43 cv2.waitKey(0)
```

Our code starts off on **Line 35** with more looping. This time we aren't looping over the size of our radii – we are instead going to draw `25` random circles, making use of NumPy's random number capabilities through the `np.random.randint` function.

In order to draw a random circle, we need to generate three values: the `radius` of the circle, the `color` of the circle,

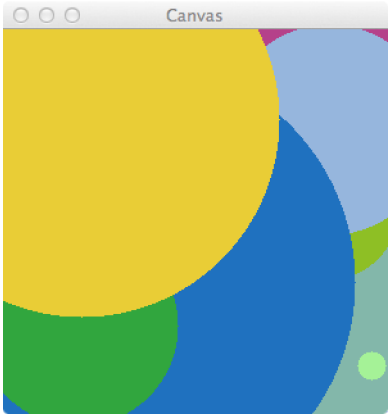

Figure 5.3: The results of our masterpiece. Notice that each circle is randomly placed on the canvas with a random color.

and the `pt` – the (x, y) coordinate of where the circle will be drawn.

We generate a `radius` value in the range $[5, 200)$ on **Line 36**. This value controls how large our circle will be.

Next, we randomly generate a color on **Line 37**. As we know, the color of a RGB pixel consists of three values in the range $[0, 255]$. In order to get three random integers rather than only one integer, we pass the keyword argument `size=(3,)`, instructing NumPy to return a list of three numbers.

Lastly, we need a (x, y) point to draw our circle. We'll generate a point in the range $[0, 300)$, again using NumPy's `np.random.randint` function.

The drawing of our circle then takes place on **Line 40**, using the `radius`, `color`, and `pt` that we randomly generated. Notice how we use a thickness of `-1` so our circles are drawn as a solid color and not just an outline.

Our masterpiece is then shown to us on **Lines 42 and 43**.

You can check out our work in Figure 5.3. Notice how each circle has a different size, color, and placement on our canvas.

In this chapter you were introduced to basic drawing functions using OpenCV. We explored how to draw shapes using the `cv2.line`, `cv2.rectangle`, and `cv2.circle` methods.

While these functions seem extremely basic and simple, make sure you understand them! They are essential building blocks that will come in handy later in this book.

6

IMAGE PROCESSING

Now that you have a solid foundation to build upon, we can start to explore simple image processing techniques.

First, we'll start off with basic image transformations, such as translation, rotation, resizing, flipping, and cropping. Then, we'll explore other types of image processing techniques, including image arithmetic, bitwise operations, and masking.

Finally, we'll explore how to split an image into its respective channels and then merge them back together again. We'll conclude this chapter with a discussion of different color spaces that OpenCV supports and the benefits and limitations of each of them.

6.1 IMAGE TRANSFORMATIONS

In this section we'll cover basic image transformations. These are common techniques that you'll likely apply to images, including translation, rotation, resizing, flipping and cropping. We'll explore each of these techniques in detail.

Make sure you have a good grasp of these methods! They are important in nearly all areas of computer vision.

6.1.1 *Translation*

The first method we are going to explore is translation. Translation is the shifting of an image along the x and y axis. Using translation, we can shift an image up, down, left, or right, along with any combination of the above!

This concept is better explained through some code:

Listing 6.1: translation.py

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 M = np.float32([[1, 0, 25], [0, 1, 50]])
15 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
16 cv2.imshow("Shifted Down and Right", shifted)
17
18 M = np.float32([[1, 0, -50], [0, 1, -90]])
19 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
20 cv2.imshow("Shifted Up and Left", shifted)
```

On **Lines 1-4** we simply import the packages we will make use of. At this point, using `numpy`, `argparse`, and `cv2`

should feel like commonplace. However, I am introducing a new package here: `imutils`. This isn't a package included in NumPy or OpenCV. Rather, its a library that we are going to write ourselves and create "conveinence" methods to do common tasks like translation, rotation, and resizing.

After we have the necessary packages imported, we construct our argument parser and load our image on **Lines 6-12**.

The actual translation takes place on **Lines 14-16**. We first define our translation matrix M. This matrix tells us how many pixels to the left or right our image will shifted, and then how many pixels up or down the image will be shifted.

Our translation matrix M is defined as a floating point array – this is important because OpenCV expects this matrix to be of floating point type. The first row of the matrix is $[1, 0, t_x]$, where t_x is the number of pixels we will shift the image left or right. Negative values of t_x will shift the image to the left and positive values will shift the image to the right.

Then, we define the second row of the matrix as $[0, 1, t_y]$, where t_y is the number of pixels we will shift the image up or down. Negative valuee of t_y will shift the image up and positive values will shift the image down.

Using this notation, we can see on **Line 14** that $t_x = 25$ and $t_y = 50$, implying that we are shifting the image 25 pixels to the right and 50 pixels down.

Now that we have our translation matrix defined, the actual translation takes place on **Line 15** using the `cv2.warpAffine` function. The first argument is the image we wish to shift and the second argument is our translation matrix M. Finally, we manually supply the dimensions (width and height) of our image as the third argument. **Line 16** shows the results of the translation.

Moving on to **Lines 18-20**, we perform another translation. Here, we set $t_x = -50$ and $t_y = -90$, implying that we are shifting the image 50 pixels to the left and 90 pixels up. The image is shifted *left* and *up* rather than *right* and *down* because we are providing a negative values for both t_x and t_y.

However, manually constructing this translation matrix and calling the `cv2.warpAffine` method takes a fair amount of code – and it's not pretty code either!

Let's create a new file: `imutils.py`. This file will store basic image processing methods, allowing us to conveniently call them without writing a lot of code.

The first method we are going to define is a `translate` function:

Listing 6.2: imutils.py

```
1 import numpy as np
2 import cv2
3
4 def translate(image, x, y):
5     M = np.float32([[1, 0, x], [0, 1, y]])
6     shifted = cv2.warpAffine(image, M, (image.shape[1], image.
          shape[0]))
```

```
7
8     return shifted
```

Our `translate` method takes three parameters: the `image` we are going to translate, the number pixels that we are going to shift along the x-axis, and the number of pixels we are going to shift along the y-axis.

This method then defines our translation matrix M on **Line 5** and then applies the actual shift on **Line 6**. Finally, we return the shifted image on **Line 8**.

Let's apply our translate method and compare to the methods discussed above:

Listing 6.3: translation.py

```
21 shifted = imutils.translate(image, 0, 100)
22 cv2.imshow("Shifted Down", shifted)
23 cv2.waitKey(0)
```

Using our convenience `translate` method, we are able to shift the image 100 pixels down using a single line of code. Furthermore, this `translate` method is much easier to use – less code is required and based on the function name, we conveniently know what image processing task is being performed.

To see our translation in action, take a look at Figure 6.1. Our original image is on the *top-left*. On the *top-right*, we shift our image 25 pixels to the right and 50 pixels down. Next, we translate our image 50 pixels to the left and 90 pixels up by using negative values for t_x and t_y. Finally, on the *bottom-right* we shift our T-Rex 100 pixels down using

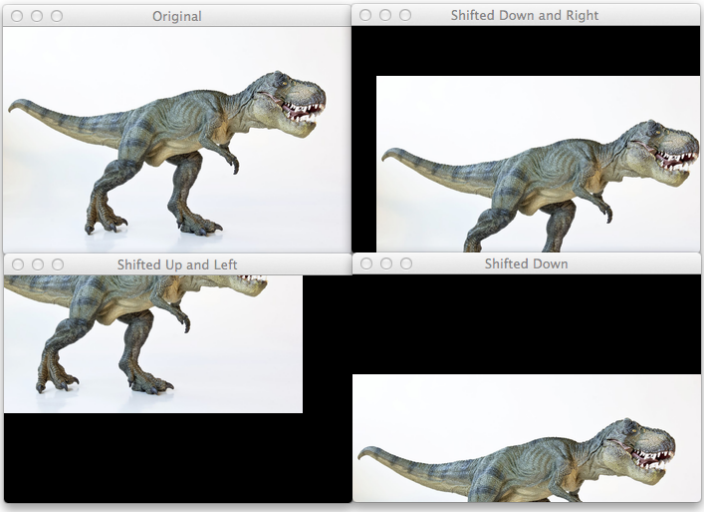

Figure 6.1: *Top-Left:* Our original T-rex image. *Top-Right:* Translating our image 25 pixels to the right and 50 pixels down. *Bottom-Left:* Shifting T-Rex 50 pixels to the left and 90 pixels up. *Bottom-Right:* Shifting the T-Rex down using our convenience method.

our convenient `translate` method defined above.

In this section we explored how to shift an image up, down, left, and right. Next up, we'll explore how to rotate an image.

6.1.2 *Rotation*

Rotation is exactly what it sounds like: rotating an image by some angle θ. In this section, we'll explore how to rotate an image. We'll use θ to represent by how many degrees we are rotating the image. Later, I'll provide another convenience method, `rotate` to make performing rotations on images easier.

Listing 6.4: rotate.py

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 (h, w) = image.shape[:2]
15 center = (w / 2, h / 2)
16
17 M = cv2.getRotationMatrix2D(center, 45, 1.0)
18 rotated = cv2.warpAffine(image, M, (w, h))
19 cv2.imshow("Rotated by 45 Degrees", rotated)
20
21 M = cv2.getRotationMatrix2D(center, -90, 1.0)
```

```
22 rotated = cv2.warpAffine(image, M, (w, h))
23 cv2.imshow("Rotated by -90 Degrees", rotated)
```

Lines 1-4 again import the packages we need. You should take note of `imutils`. Once again we will be defining a convenience method to make our lives easier.

Lines 6-12 construct our argument parser. We only need one argument: the path to the image we are going to use. We then load our image off disk and display it.

When we rotate an image, we need to specify which point we want to rotate about. In most cases, you will want to rotate around the *center* of an image; however, OpenCV allows you to specify any arbitrary point you want to rotate around. Let's just go ahead and rotate about the center of the image. **Lines 14 and 15** grabs the width and height of the image, then divides each by 2 to determine the center of the image.

Just as we defined a matrix to translate an image, we also define a matrix to rotate the image. Instead of manually constructing the matrix using NumPy, we'll just make a call to the `cv2.getRotationMatrix2D` method on **Line 17**.

The `cv2.getRotationMatrix2D` function takes three arguments: the point in which we want to rotate the image about (in this case, the `center` of the image). We then specify θ, the number of degrees we are going to rotate the image by. In this case, we are going to rotate the image 45 degrees. The last argument is the scale of the image. We haven't discussed resizing an image yet, but here you can specify a floating point value, where 1.0 means the same dimensions of the image are used. However, if you specified

a value of 2.0 the image would be doubled in size. Similarly, a value of 0.5 halve the size of the image.

Once we have our rotation matrix M from the `cv2.getRotationMatrix2D` function, we can apply the rotation to our image using the `cv2.warpAffine` method on **Line 18**. The first argument to this function is the image we want to rotate. We then specify our rotation matrix M along with the output dimensions (width and height) of our image. **Line 19** then shows our image rotated by 45 degrees. Check out Figure 6.2 *Top-Right* to see our rotated image.

Let's not waste anytime. We'll go ahead and jump into some code to perform rotations:

On **Lines 21-23** we perform another rotation. The code is identical to that **Lines 17-19**, only this time we are rotating by -90 degrees rather than 45. Figure 6.2 *Bottom-Left* shows our T-Rex rotated by -90 degrees.

Just as in translating an image, the code to rotate an image isn't the most pretty and Pythonic. Let's change that and define our own custom `rotate` method:

Listing 6.5: imutils.py

```
27 def rotate(image, angle, center = None, scale = 1.0):
28     (h, w) = image.shape[:2]
29
30     if center is None:
31         center = (w / 2, h / 2)
32
33     M = cv2.getRotationMatrix2D(center, angle, scale)
34     rotated = cv2.warpAffine(image, M, (w, h))
35
36     return rotated
```

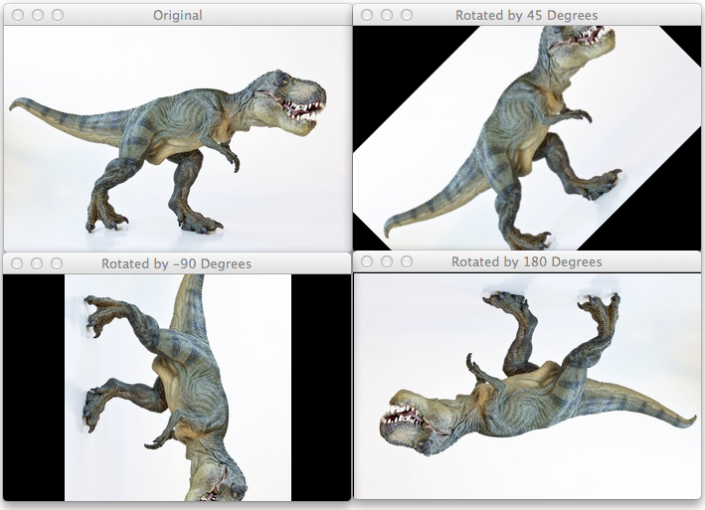

Figure 6.2: *Top-Left:* Our original T-Rex image. *Top-Right:* Rotating the image by 45 degrees. *Bottom-Left:* Rotating the image by -90 degrees. *Bottom-Right*: Flipping T-Rex upside-down by rotating the image by 180 degrees.

Our `rotate` method takes four arguments. The first is our image. The second is the angle θ in which we want to rotate the image. We provide two optional keyword arguments, `center` and `scale`. The `center` parameter is the point in which we wish to rotate our image about. If a value of `None` is provided, the method automatically determines the center of the image on **Lines 30-31**. Finally, the `scale` parameter is used to handle if the size of the image should be changed during the rotation. The `scale` parameter has a default value of 1.0, implying that no resizing should be done.

The actual rotation of the image takes place on **Lines 33 and 34**, were we construct our rotation matrix M and apply it to the image. Finally, our image is returned on **Line 36**.

Now that we have defined our `rotate` method, let's apply it:

Listing 6.6: rotate.py

```
24 rotated = imutils.rotate(image, 180)
25 cv2.imshow("Rotated by 180 Degrees", rotated)
26 cv2.waitKey(0)
```

Here we are rotating our image by 180 degrees. Figure 6.2 *Bottom-Right* shows that our T-Rex has indeed been flipped upside-down. The code for our rotate method is much easier to read and maintain that making calls to `cv2.getRotationMatrix2D` and `cv2.warpAffine` each time we want to rotate an image.

6.1.3 *Resizing*

So far we've covered two image transformations: translation and rotation. Now, we are going to explore how to resize an image. We'll also define one last method for our `imutils.py` file, a convenience method to help us resize images with ease.

Perhaps, not surprisingly, we will be using the `cv2.resize` function to resize our images. But we need to keep in mind the aspect ratio of the image when we are using this function. But before we get too deep into the details, let's jump right into an example:

Listing 6.7: resize.py

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 r = 150.0 / image.shape[1]
15 dim = (150, int(image.shape[0] * r))
16
17 resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
18 cv2.imshow("Resized (Width)", resized)
```

Lines 1-12 should start to feel quite redundant at this point. We are importing our packages, setting up our argument parser, and finally loading our image and displaying

it.

The actual interesting code doesn't start until **Lines 14 and 15**. When resizing an image, we need to keep in mind the aspect ratio of the image. The aspect ratio is the proportional relationship of the width and the height of the image. If we aren't mindful of the aspect ratio, our resizing will return results that don't look correct.

Computing the aspect ratio is handled on **Line 14**. In this line of code we define our new image width to be 150 pixels. In order to compute the ratio of the new height to the old height, we simply define our ratio `r` to be the new width (150 pixels) divided by the old width, which we access using `image.shape[1]`.

Now that we have our ratio, we can compute the new dimensions of the image on **Line 15**. Again, the width of the new image will be 150 pixels. The height is then computed by multiplying the old height by our ratio and converting it to an integer.

The actual resizing of the image takes place on **Line 17**. The first argument is the image we wish to resize and the second is our computed dimensions for the new image. Last parameter is our `interpolation` method, which is the algorithm working behind the scenes to handle how the actual image is resized. In general, I find that using `cv2.INTER_AREA` obtains the best results when resizing; however, other appropriate choices include `cv2.INTER_LINEAR`, `cv2.INTER_CUBIC`, and `cv2.INTER_NEAREST`.

Finally, we show our resized image on **Line 18**

In the example we just explored, we only resized the image by specifying the width. But what if we wanted to resize the image by specifying the height? All that requires is a change to computing the aspect ratio:

Listing 6.8: resize.py

```
19 r = 50.0 / image.shape[0]
20 dim = (int(image.shape[1] * r), 50)
21
22 resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
23 cv2.imshow("Resized (Height)", resized)
24 cv2.waitKey(0)
```

On **Line 19** we define our ratio `r`. Our new image will have a height of 50 pixels. To determine the ratio of the new height to the old height, we divide 50 by the old height.

Then, we define the dimensions of our new image. We already know that the new image will have a height of 50 pixels. The new width is obtained by multiplying the old width by the ratio.

We then perform the actual resizing of the image on **Line 22** and show it on **Line 23**.

Resizing an image is simple enough, but having to compute the aspect ratio, define the dimensions of the new image, and then perform the resizing takes three lines of code. This looks like the perfect time to define a `resize` method in our `imutils.py` file:

Listing 6.9: resize.py

```
25 resized = imutils.resize(image, width = 100)
```

```
26 cv2.imshow("Resized via Function", resized)
27 cv2.waitKey(0)
```

In this example you can see that the resizing of the image is handled by a single function: `imutils.resize`. The first argument we pass in is our image we want to resize. Then, we specify the keyword argument `width`, which is the width of our new image. The function then handles the resizing for us.

Of course, we could also resize via the height of the image by changing the function call to:

Listing 6.10: resize.py

```
1     resized = imutils.resize(image, height = 50)
```

Let's take this function apart and see what's going on under the hood:

Listing 6.11: imutils.py

```
9  def resize(image, width = None, height = None, inter = cv2.
       INTER_AREA):
10     dim = None
11     (h, w) = image.shape[:2]
12
13     if width is None and height is None:
14         return image
15
16     if width is None:
17         r = height / float(h)
18         dim = (int(w * r), height)
19
20     else:
21         r = width / float(w)
22         dim = (width, int(h * r))
23
24     resized = cv2.resize(image, dim, interpolation = inter)
25
```

```
26     return resized
```

As you can see, we have defined our `resize` function. The first argument is the image we want to resize. Then, we define two keyword arguments, `width` and `height`. Both of these arguments cannot be `None`, otherwise we won't know how to resize the image. We also provide `inter`, which is our interpolation method and defaults to `cv2.INTER_AREA`.

On **Lines 10 and 11** we define the dimensions of our new, resized image and grab the dimensions of the original image.

We perform a quick check on **Lines 13-14** to ensure that a numerical value has been provided for either the width or the height.

The computation of the ratio and new, resized image dimensions are handled on **Lines 16-22**, depending on whether we are resizing via width or via height.

Line 24 handles the actual resizing of the image, then **Line 26** returns our resized image to the user.

To see the results of our image resizings, check out Figure 6.3. On the *Top-Left* we have our original T-Rex image. Then, on the *Top-Right* we have our T-Rex resized to have a width of 150 pixels. The *Middle-Right* image then shows our image resized to have a height of 50 pixels. Finally, *Bottom-Right* shows the output of our `resize` function – the T-Rex is now resized to have a width of 100 pixels using only a single line of code.

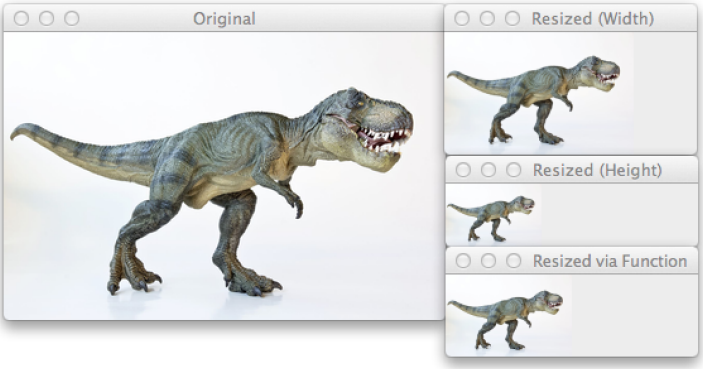

Figure 6.3: *Top-Left:* Our original T-Rex image. *Top-Right:* The T-Rex resized to have a width of 150 pixels. *Middle-Right:* Our image resized to have a height of 50 pixels. *Bottom-Right:* Resizing our image to have a width of 100 pixels using our helper function. In all cases, the aspect ratio of the image is maintained.

Translation, rotation, and resizing are certainly the more challenging and involved image transformation tasks. The next two we will explore, flipping and cropping, are substantially easier.

6.1.4 *Flipping*

Next up on our image transformations to explore is flipping an image. We can flip an image around either the x or y axis, or even both.

In fact, I think explaining how to flip an image is better explained by viewing the output of an image flip, before we get into the code. Check out Figure 6.4 to see our T-Rex image flipped horizontally, vertically, and both horizontally and vertically at the same time.

Now that you see what an image flip looks like, we can explore the code:

Listing 6.12: flipping.py

```
1 import argparse
2 import cv2
3
4 ap = argparse.ArgumentParser()
5 ap.add_argument("-i", "--image", required = True,
6     help = "Path to the image")
7 args = vars(ap.parse_args())
8
9 image = cv2.imread(args["image"])
10 cv2.imshow("Original", image)
11
12 flipped = cv2.flip(image, 1)
13 cv2.imshow("Flipped Horizontally", flipped)
14
```

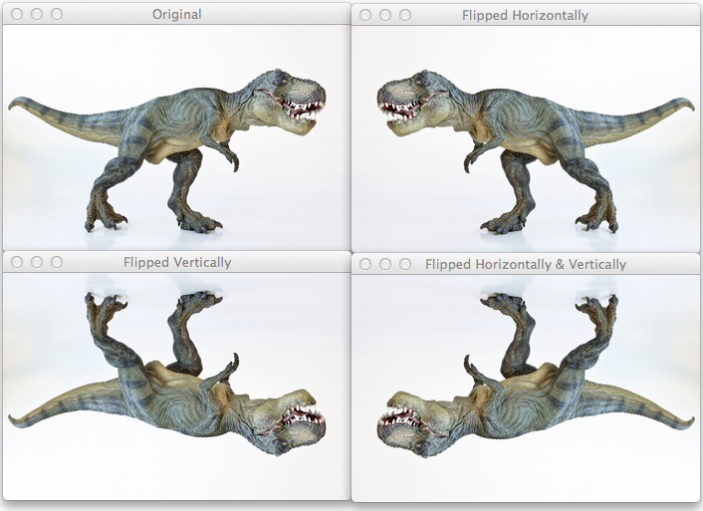

Figure 6.4: *Top-Left:* Our original T-Rex image. *Top-Right:* Flipping the T-Rex image horizontally. *Bottom-Left:* Flipping the T-Rex vertically. *Bottom-Right*: Flipping the image both horizontally and vertically.

```
15 flipped = cv2.flip(image, 0)
16 cv2.imshow("Flipped Vertically", flipped)
17
18 flipped = cv2.flip(image, -1)
19 cv2.imshow("Flipped Horizontally & Vertically", flipped)
20 cv2.waitKey(0)
```

lines 1-10 handle our standard procedure of importing our packages, parsing arguments, and loading our image from disk.

Flipping an image is accomplished by making a call to the `cv2.flip` function on **Line 12**. The `cv2.flip` method requires two arguments: the image we want to flip and a flip code that is used to determine how we are going to flip the image.

Using a flip code value of 1 indicates that we are going to flip the image horizontally, around the y-axis (**Line 12**). Specifying a flip code of 0 indicates that we want to flip the image vertically, around the x-axis (**Line 15**). Finally, using a negative flip code (**Line 18**) flips the image around both axes.

Again, to see the output of our flipping example, take a look at Figure 6.4. Here we can see the image flipped horizontally, vertically, and around both axes.

Flipping an image is very simple, perhaps one of the most simple examples in this book! Next up, we'll go over cropping an image and how to extract regions of an image using NumPy array slices.

Figure 6.5: *Top:* Our original T-Rex image. *Bottom:* Cropping the face of the T-Rex using NumPy array slices.

6.1.5 *Cropping*

When we crop an image, we want to remove the outer parts of the image that we are not interested in. We can accomplish image cropping by using NumPy array slicing. In fact, we already performed image cropping in Chapter 4!

However, let's review it again and make sure we understand what is going on:

Listing 6.13: crop.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
```

```
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 cropped = image[30:120 , 240:335]
14 cv2.imshow("T-Rex Face", cropped)
15 cv2.waitKey(0)
```

Lines 1-11 handle importing our packages, parsing our arguments, and loading our images. For our cropping example, we will use our T-Rex image.

The actual cropping takes place on a single line of code: **Line 13**. We are supply NumPy array slices to extract a rectangular region of the image, starting at $(240, 30)$ and ending at $(335, 120)$. The order in which we supply the indexes to the crop may seem counterintuitive; however, remember that OpenCV represents images as NumPy arrays with the the height first and the width second. This means that we need to supply our y-axis values before our x-axis.

In order to perform our cropping, NumPy expects four indexes:

1. **Start y:** The starting y coordinate. In this case, we start at $y = 30$.
2. **End y:** The ending y coordinate. We will end our crop at $y = 120$.
3. **Start x:** The starting x coordinate of the slice. We start the crop at $x = 240$.

4. **End x:** The ending x-axis coordinate of the slice. Our slice ends at $x = 335$.

Executing our code detailed above, we will see from Figure 6.5 that we have cropped out the face of our T-Rex! While the T-Rex might seem a little scary, cropping sure isn't! In fact, it's quite simple when you consider all we are doing is performing array slices on NumPy arrays!

6.2 IMAGE ARITHMETIC

We all know basic arithmetic operations like addition and subtraction. But when working with images, we need to keep in mind the limits of our color space and data type.

For example, RGB images have pixels that fall within the range $[0, 255]$. What happens if we are examining a pixel with intensity 250 and we try to add 10 to it?

Under normal arithmetic rules, we would end up with a value of 260. However, since RGB images are represented as 8-bit unsigned integers, 260 is not a valid value.

So what should happen? Should we perform a check of some sorts to ensure no pixel falls outside the range of $[0, 255]$, thus clipping all pixels to have a minimum value of 0 and a maximum value of 255?

Or do we apply a modulus operation, and "wrap around"? Under modulus rules, adding 10 to 255 would simply wrap around to a value of 4.

Which way is the "correct" way to handle images additions and subtractions that fall outside the range of $[0, 255]$?

The answer is there is no correct way – it simply depends on how you manipulating your pixels and what you want the desired results to be.

However, be sure to keep in mind that there is a difference between OpenCV and NumPy addition. NumPy will perform modulus arithmetic and "wrap around". OpenCV on the other hand will perform clipping and ensure pixel values never fall outside the range $[0, 255]$.

But don't worry! These nuances will become more clear as we explore some code below.

Listing 6.14: arithmetic.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 print "max of 255: " + str(cv2.add(np.uint8([200]), np.uint8
       ([100])))
14 print "min of 0: " + str(cv2.subtract(np.uint8([50]), np.uint8
       ([100])))
15
16 print "wrap around: " + str(np.uint8([200]) + np.uint8([100]))
17 print "wrap around: " + str(np.uint8([50]) - np.uint8([100]))
```

We are going to perform our standard procedure on **Lines 1-11** by importing our packages, setting up our argument parser, and loading our image.

Remember how I mentioned the difference between OpenCV and NumPy addition above? Well now we are going to explore it further and provide a concrete example to ensure we fully understand it.

On **Line 13** we define two NumPy arrays that are 8-bit unsigned integers. The first array has one element: a value of 200. The second array also has only one element, but a value of 100. We then use OpenCV's `cv2.add` method to add the values together.

What do you think the output is going to be?

Well according, to standard arithmetic rules, we would think the result should be 300, *but*, remember that we are working with 8-bit unsigned integers that only have a range between $[0, 255]$. Since we are using the `cv2.add` method, OpenCV takes care of clipping for us, and ensures that the addition produces a maximum value of 255. When we execute this code, we can see the result on the first line of Listing 6.15. Sure enough, the addition returned a value of 255.

Line 14 then performs subtraction using `cv2.subtract`. Again, we define two NumPy arrays, each with a single element, and of the 8-bit unsigned integer data type. The first array has a value of 50 and the second a value of 100.

According to our arithmetic rules, the subtraction should return a value of -50; however, OpenCV once again performs clipping for us. We find that the value is clipped to a value of 0. The second line of Listing 6.15 verifies this: subtracting 100 from 50 using `cv2.subtract` returns a value of 0.

Listing 6.15: arithmetic.py

```
max of 255: [[255]]
min of 0: [[0]]
```

But what happens if we use NumPy to perform the arithmetic instead of OpenCV?

Line 16 and 17 explore this question.

First, we define two NumPy arrays, each with a single element, and of the 8-bit unsigned integer data type. The first array has a value of 200 and the second has a value of 100. Using the `cv2.add` function, our addition would be clipped and a value of 255 returned.

However, NumPy does not perform clipping – it instead performs modulo arithmetic and "wraps around". Once a value of 255 is reached, NumPy wraps around to zero, and then starts counting up again, until 100 steps have been reached. You can see this is true via the first line of output on Listing 6.16.

Then, we define two more NumPy arrays: one has a value of 50 and the other 100. Using the `cv2.subtract` method, this subtraction would be clipped to return a value of 0. However, we know that NumPy performs modulo arith-

metic rather than clipping. Instead, once 0 is reached during the subtraction, the modulo operations wraps around and starts counting backwards from 255 – thus the result on the second line of output on Listing 6.16.

Listing 6.16: arithmetic.py

```
wrap around: [44]
wrap around: [206]
```

When performing integer arithmetic it is important to keep in mind your desired output.

Do you want all values to be clipped if they fall outside the range $[0, 255]$? Then use OpenCV's built in methods for image arithmetic.

Do you want modulus arithmetic operations and have values wrap around if they fall outside the range of $[0, 255]$? Then simply add and subtract the NumPy arrays as you normally would.

Now that we have explored the caveats of image arithmetic in OpenCV and NumPy, let's perform the arithmetic on actual images and view the reuslts:

Listing 6.17: arithmetic.py

```
18 M = np.ones(image.shape, dtype = "uint8") * 100
19 added = cv2.add(image, M)
20 cv2.imshow("Added", added)
21
22 M = np.ones(image.shape, dtype = "uint8") * 50
23 subtracted = cv2.subtract(image, M)
24 cv2.imshow("Subtracted", subtracted)
25 cv2.waitKey(0)
```

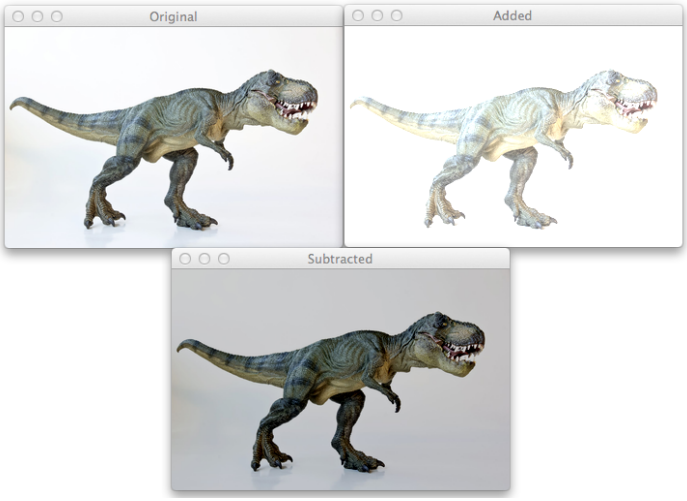

Figure 6.6: *Top-Left:* Our original T-Rex image. *Top-Right:* Adding 100 to every pixel in the image. Notice how the image looks more "washed out" and is substantially brighter than the original. *Bottom*: Subtracting 50 from every pixel in the image. Notice that the image is now darker than the original.

Line 18 defines an NumPy array of ones, with the same size as our image. Again, we are sure to use 8-bit unsigned integers as our data type. In order to fill our matrix with values of 100's rather than 1's, we simply multiply our matrix of 1's by 100. Finally, we use the `cv2.add` function to add our matrix of 100's to the original image – thus increasing every pixel intensity in the image by 100, but ensuring all values are clipped to the range $[0, 255]$ if they attempt to exceed 255.

The result of our operation can be found in Figure 6.6 *Top-Right*. Notice how the image looks more "washed out" and is substantially brighter than the original. This is because we are increasing the pixel intensities by adding 100 to them and pushing them towards brighter colors.

We then create another NumPy array filled with 50's on **Line 23** and use the `cv2.subtract` function to subtract 50 from each pixel intensity of the image. The *Bottom* image in Figure 6.6 shows the results of this subtraction. Our image now looks considerably darker than the original T-Rex. Pixels that were once white now look gray. This is because we are subtracting 50 from the pixels and pushing them towards the darker regions of the RGB color space.

In this section we explored the peculiarities of image arithmetic using OpenCV and NumPy. These caveats are important to keep in mind, otherwise you may get unwanted results when performing arithmetic operations on your images.

6.3 BITWISE OPERATIONS

In this section we will review four bitwise operations: AND, OR, XOR, and NOT. These four operations, while very basic and low level, are paramount to image processing, especially when we start working with masks in Section 6.4.

Bitwise operations operate in a binary manner and are represented as grayscale images. A given pixel is turned "off" if it has a value of zero and it is turned "on" if the pixel has a value greater than zero.

Let's go ahead and jump into some code:

Listing 6.18: bitwise.py

```
1 import numpy as np
2 import cv2
3
4 rectangle = np.zeros((300, 300), dtype = "uint8")
5 cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
6 cv2.imshow("Rectangle", rectangle)
7
8 circle = np.zeros((300, 300), dtype = "uint8")
9 cv2.circle(circle, (150, 150), 150, 255, -1)
10 cv2.imshow("Circle", circle)
```

The first two lines of code import the packages we will need: `numpy` and `cv2`. We initialize our rectangle image as a 300×300 NumPy array on **Line 4**. We then draw a 250×250 white rectangle at the center of the image.

Similarly, on **Line 8** we initialize another image to contain our circle, which we draw on **Line 9**, again centered at the center of the image, with a radius of 150 pixels.

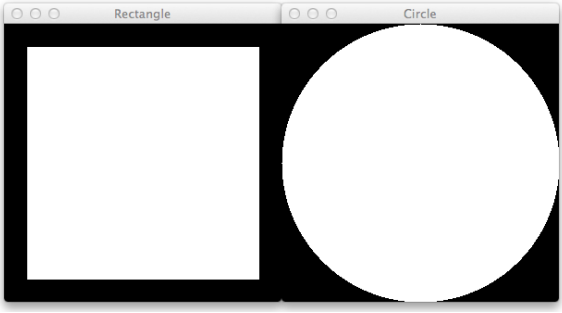

Figure 6.7: *Left*: Our rectangle image. *Right:* Our circle image. We will explore how these two images can be combined using bitwise operations.

Figure 6.7 shows our two shapes. We will make use of these shapes to demonstrate our bitwise operations:

Listing 6.19: bitwise.py

```
11 bitwiseAnd = cv2.bitwise_and(rectangle, circle)
12 cv2.imshow("AND", bitwiseAnd)
13 cv2.waitKey(0)
14
15 bitwiseOr = cv2.bitwise_or(rectangle, circle)
16 cv2.imshow("OR", bitwiseOr)
17 cv2.waitKey(0)
18
19 bitwiseXor = cv2.bitwise_xor(rectangle, circle)
20 cv2.imshow("XOR", bitwiseXor)
21 cv2.waitKey(0)
22
23 bitwiseNot = cv2.bitwise_not(circle)
24 cv2.imshow("NOT", bitwiseNot)
25 cv2.waitKey(0)
```

As I mentioned above, a given pixel is turned "on" if it has a value greater than zero and it is turned "off" if it has a value of zero. Bitwise functions operate on these binary conditions.

In order to utilize bitwise functions we assume (in most cases) that we are comparing two pixels (the only exception is the NOT function). We'll compare each of the pixels and then construct our bitwise representation.

Let's quickly review our binary operations:

1. **AND:** A bitwise AND is true if and only if both pixels are greater than zero.
2. **OR:** A bitwise OR is true if either of the two pixels are greater than zero.
3. **XOR:** A bitwise XOR is true if and only if the two pixels are greater than zero, but not both.
4. **NOT:** A bitwise NOT inverts the "on" and "off" pixels in an image.

On **Line 11** we apply a bitwise AND to our rectangle and circle images using the `cv2.bitwise_and` function. As the list above mentions, a bitwise AND is true if and only if both pixels are greater than zero. The output of our bitwise AND can be seen in Figure 6.8 *Top-Left*. We can see that edges of our square are lost – this makes sense because our rectangle does not cover as large of an area as the circle, and thus both pixels are not "on".

We then apply a bitwise OR on **Line 15** using the `cv2.bitwise_or` function. A bitwise OR is true if either of the two pixels are greater than zero. Figure 6.8 *Top-Right* shows the output of our bitwise OR. In this case, our square and rectangle have been combined together.

Next up is the bitwise XOR function, applied on **Line 19** using the `cv2.bitwise_xor` function. A XOR operation is true if both pixels are greater than zero, *but*, both pixels cannot be greater than zero. The output of the XOR operation is displayed on Figure 6.8 *Bottom-Right*. Here we see that the center of the square has been removed. Again, this makes sense because an XOR operation cannot have both pixels greater than zero.

Finally, we apply the NOT function on **Line 23** using the `cv2.bitwise_not` function. Essentially, the bitwise NOT function flips pixel values. All pixels that are greater than zero are set to zero, and all pixels that are set to zero are set to 255. Figure 6.8 *Bottom-Right* flips our white circle to a black circle.

Overall, bitwise functions are extremely simple, yet very powerful. And they are absolutely essentially when we start to discuss masking in Section 6.4.

6.4 MASKING

In the previous section we explored bitwise functions. Now we are ready to explore masking, an extremely powerful and useful technique in computer vision and image pro-

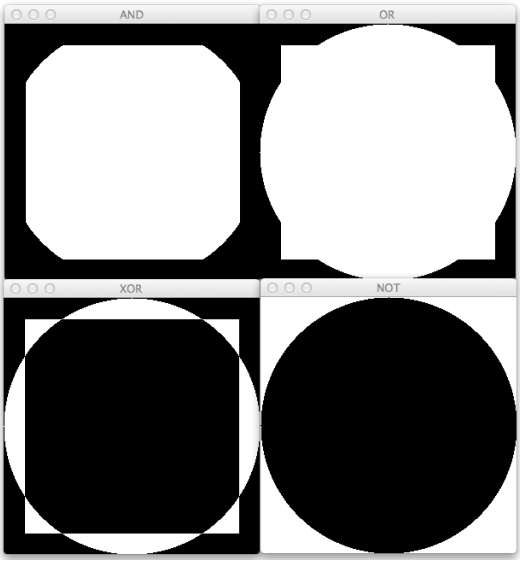

Figure 6.8: *Top-Left:* Applying a bitwise AND to our rectangle and circle image. *Top-Right:* A bitwise OR applied to our square and circle. *Bottom-Left:* An XOR applied to our shapes. *Bottom-Right:* Flipping pixel values of our circle using a bitwise NOT.

cessing.

Using a mask allows us to focus only on the portions of the image that interests us.

For example, let's say that we were building a computer vision system to recognize faces. The only part of the image we are interested in finding and describing are the parts of the image that contain faces – we simply don't care about the rest of the content of the image. Provided that we could find the faces in the image, we might construct a mask to show *only* the faces in the image.

Let's make this example a little more concrete.

In Figure 6.9 we have an image of a beach on the *Top-Left*. But I'm not interested in the beach in the image. I'm only interested in the sky and the palm tree. We could apply a cropping to extract that region of the image. Or, we could apply a mask to the image.

The image on the *Top-Right* is our mask – a white rectangle at the center of the image. By applying our mask to our beach image, we arrive at the image on the *Bottom*. By using our rectangle mask, we have focused only on the sky and palm tree in the image.

Let's examine the code to accomplish the masking in Figure 6.9:

Listing 6.20: masking.py

```
1 import numpy as np
```

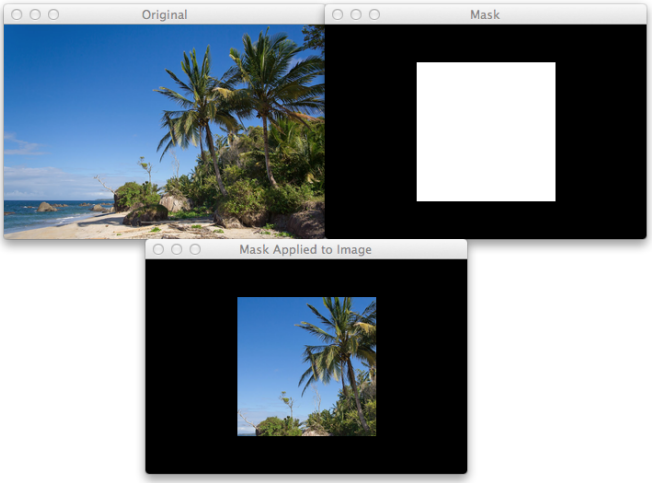

Figure 6.9: *Top-Left:* Our image of a peaceful beach scene. *Top-Right:* Our mask image – a white rectangle at the center of the image. *Bottom:* Applying the rectangular mask to the beach image. Only the parts of the image where the mask pixels are greater than zero are shown.

```
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 mask = np.zeros(image.shape[:2], dtype = "uint8")
14 (cX, cY) = (image.shape[1] / 2, image.shape[0] / 2)
15 cv2.rectangle(mask, (cX - 75, cY - 75), (cX + 75 , cY + 75), 255,
       -1)
16 cv2.imshow("Mask", mask)
17
18 masked = cv2.bitwise_and(image, image, mask = mask)
19 cv2.imshow("Mask Applied to Image", masked)
20 cv2.waitKey(0)
```

On **Lines 1-11** we import the packages we need, parse our arguments, and load our image.

We then construct a NumPy array, filled with zeros, with the same width and height as our beach image on **Line 13**. In order to draw the white rectangle, we first compute the center of the image on **Line 14** by dividing the width and height by two. Finally, we draw our white rectangle on **Line 15**.

Remember reviewing the `cv2.bitwise_and` function in the previous section? It's a function that is used extensively when applying masks to images.

We apply our mask on **Line 18** using the `cv2.bitwise_and` function. The first two parameters are the image itself. Obviously, the AND function will be `True` for all pixels in the image; however, the important part of this func-

tion is the `mask` keyword argument. By supplying a mask, the `cv2.bitwise_and` function only examines pixels that are "on" in the mask. In this case, only pixels that are part of the white rectangle.

Let's look at another example:

Listing 6.21: masking.py

```
21 mask = np.zeros(image.shape[:2], dtype = "uint8")
22 cv2.circle(mask, (cX, cY), 100, 255, -1)
23 masked = cv2.bitwise_and(image, image, mask = mask)
24 cv2.imshow("Mask", mask)
25 cv2.imshow("Mask Applied to Image", masked)
26 cv2.waitKey(0)
```

On **Line 21** we re-initialize our mask to be filled with zeros and the same dimensions as our beach image. Then, we draw a white circle on our mask image, starting at the center of the image and a radius of 100 pixels. Applying the circular mask is then performed on **Line 23**, again using the `cv2.bitwise_and` function.

The results of our circular mask can be seen in Figure 6.10. Our beach image is shown on the *Top-Left*, our circle mask on the *Top-Right*, and the application of the mask on the *Bottom*. Instead of a rectangular region of the beach being shown, we now have a circular region.

Right now masking may not seem very interesting. But we'll return to it once we start computing histograms in Chapter 7. Again, the key point of masks is that they allow us to focus our computation only on regions of the image that interests us.

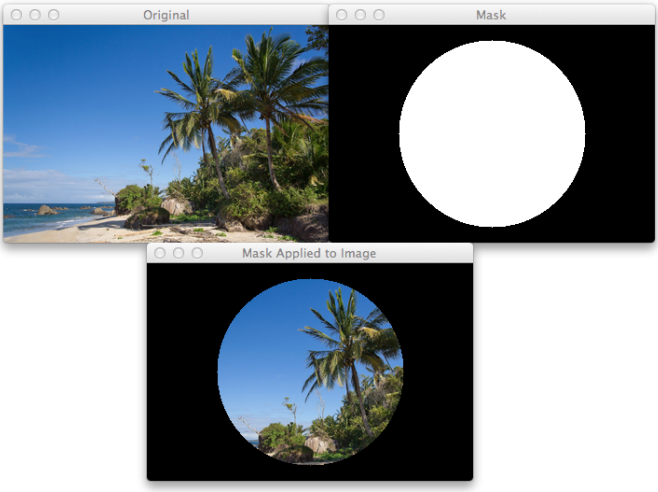

Figure 6.10: Applying the circular mask to the beach image. Only pixels within the circular white region are shown.

6.5 SPLITTING AND MERGING CHANNELS

A color image consists of multiple channels: a Red, a Green, and a Blue component. We have seen that we can access these components via indexing into NumPy arrays. But what if we wanted to split an image into its respective components?

As you'll see, we'll make use of the `cv2.split` function.

But for the time being, let's take a look at an example image in Figure 6.11.

We have an image of a wave crashing down. This image is very "blue", due to the ocean. How do we interpret the different channels of the image?

The Red channel (*Top-Left*) is very dark. This makes sense because an ocean scene has very little red colors in it. The red colors present are either very dark, and thus not represented, or very light, and likely part of the white foam of the wave as it crashes down.

The Green channel (*Top-Right*) is more represented in the image, since ocean water does contain greenish hues.

Finally, the Blue channel (*Bottom-Left*) is extremely light, and near pure white in some locations. This is because shades of blue are heavily represented in our image.

Now that we have visualized our channels, let's examine some code to accomplish this for us:

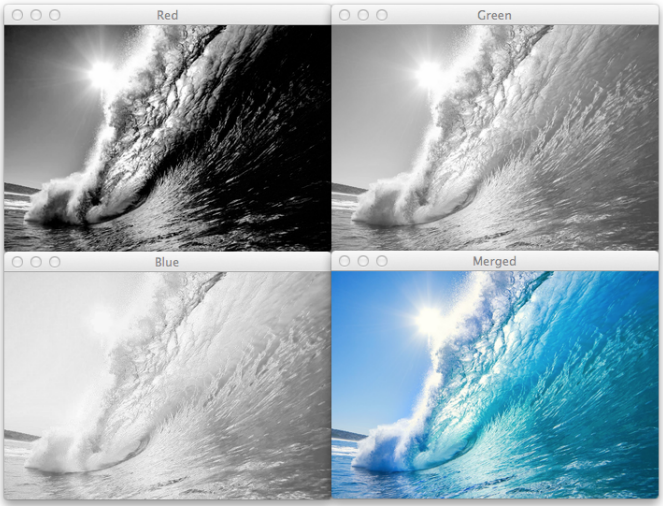

Figure 6.11: The three RGB channels of our wave image are shown on the *Bottom-Right*. The Red channels is on the *Top-Left*, the Green channel on the *Top-Right*, and the Blue channel on the *Bottom-Left*.

Listing 6.22: splitting_and_merging.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 (B, G, R) = cv2.split(image)
12
13 cv2.imshow("Red", R)
14 cv2.imshow("Green", G)
15 cv2.imshow("Blue", B)
16 cv2.waitKey(0)
17
18 merged = cv2.merge([B, G, R])
19 cv2.imshow("Merged", merged)
20 cv2.waitKey(0)
21 cv2.destroyAllWindows()
```

Lines 1-10 imports our packages, sets up our argument parser, and then loads our image. Splitting the channels is done using a call to `cv2.split` on **Line 11**.

Normally, we think of images in the RGB color space – the red pixel first, the green pixel second, and the blue pixel third. However, OpenCV stores RGB images as NumPy arrays in reverse channel order. Instead of storing an image in RGB order, it instead stores the image in BGR order, thus we unpack the tuple in reverse order.

Lines 13-16 then show each channel individually, as in Figure 6.11.

We can also merge the channels back together again using the `cv2.merge` function. We simply specify our chan-

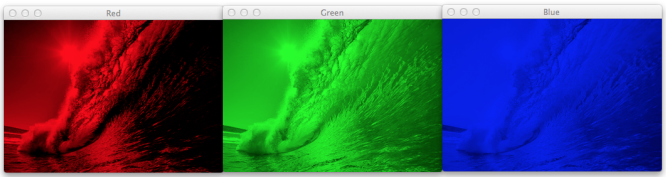

Figure 6.12: Representing the Red, Green, and Blue channels of our wave image.

nels, again in BGR order, and then `cv2.merge` takes care of the rest for us (**Line 18**)!

Listing 6.23: splitting_and_merging.py

```
22 zeros = np.zeros(image.shape[:2], dtype = "uint8")
23 cv2.imshow("Red", cv2.merge([zeros, zeros, R]))
24 cv2.imshow("Green", cv2.merge([zeros, G, zeros]))
25 cv2.imshow("Blue", cv2.merge([B, zeros, zeros]))
26 cv2.waitKey(0)
```

An alternative method to visualize the channels of an image can be seen in Figure 6.12. In order to show the actual "color" of the channel, we first need to take apart the image using `cv2.split`. Then, we need to re-construct the image, but this time, having all pixels *but the current channel* as zero.

On **Line 22** we construct a NumPy array of zeros, with the same width and height as our original image. Then, in order to construct the Red channel representation of the image, we make a call to `cv2.merge`, but specifying our `zeros` array for the Green and Blue channels. We take similar approaches to the other channels in **Line 23 and 24**.

6.6 COLOR SPACES

In this book, we have only explored the RGB color space; however, there are many other color spaces that we can utilize.

The Hue-Saturation-Value (HSV) color space is more similar to how humans think and conceive of color. Then there is the L*a*b* color space which is more tuned to how humans *perceive* color.

OpenCV provides support for many, many different color spaces. And understanding how color is perceived by humans and represented by computers occupies an entire library of literature itself.

In order to not get bogged down in the details, I'll just show you how to convert color spaces. If you think your application of image processing and computer vision might need a different color space than RGB, I will leave that as an exercise to the reader to explore the peculiarities of each color space.

Let's explore some code to change color spaces:

Listing 6.24: colorspaces.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
```

```
 9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Gray", gray)
15
16 hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
17 cv2.imshow("HSV", hsv)
18
19 lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
20 cv2.imshow("L*a*b*", lab)
21 cv2.waitKey(0)
```

Lines 1-11 imports the packages we need, parses our arguments, and loads our image. Then, on **Line 13** we convert our image from the RGB color space to grayscale by specifying the `cv2.COLOR_BGR2GRAY` flag.

Converting our image to the HSV color space is performed on **Line 16** by specifying the `cv2.COLOR_BGR2HSV` flag. Finally, on **Line 19** we convert to the L*a*b* color space by using the `cv2.COLOR_BGR2LAB` flag.

We can see the results of our color space conversions in Figure 6.13.

The role of color spaces in image processing and computer vision is important, yet complicated at the same time. If you are just getting started in computer vision, it's likely a good idea to stick to the RGB color space for the time being. However, I have included this section as a matter of completeness – it's good to show an example of how to convert color spaces for when you decide the time is right!

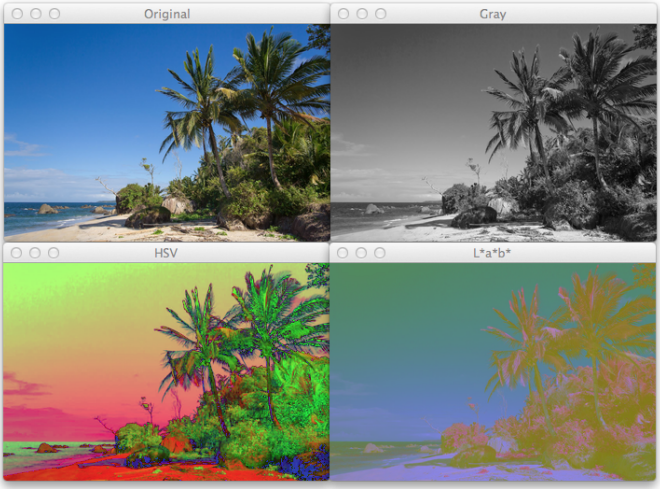

Figure 6.13: *Top-Left:* An image of beach scenery. *Top-Right:* The grayscale representation of the beach image. *Bottom-Left:* Converting the beach image to the HSV color space. *Bottom-Right:* Converting our image to the L*a*b* color space.

7

HISTOGRAMS

So, what exactly is a histogram? A histogram represents the distribution of pixel intensities (whether color or gray-scale) in an image. It can be visualized as a graph (or plot) that gives a high-level intuition of the intensity (pixel value) distribution. We are going to assume a RGB color space in this example, so these pixel values will be in the range of 0 to 255.

When plotting the histogram, the X-axis serves as our "bins". If we construct a histogram with 256 bins, then we are effectively counting the number of times each pixel value occurs. In contrast, if we use only 2 (equally spaced) bins, then we are counting the number of times a pixel is in the range $[0, 128)$ or $[128, 255]$. The number of pixels binned to the x-axis value is then plotted on the y-axis.

By simply examining the histogram of an image, you get a general understanding regarding the contrast, brightness,

and intensity distribution.

7.1 USING OPENCV TO COMPUTE HISTOGRAMS

Now, let's start building some histograms of our own.

We will be using the `cv2.calcHist` function to build our histograms. Before we get into any code examples, let's quickly review the function:

```
cv2.calcHist(images,channels,mask,histSize,ranges)
```

1. **images:** This is the image that we want to compute a histogram for. Wrap it as a list: `[myImage]`.
2. **channels:** A list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be `[0]`. To compute a histogram for all three red, green, and blue channels, the channels list would be `[0,1,2]`.
3. **mask:** Remember learning about masks in Chapter 6? Well, here we can supply a mask. If a mask is provided, a histogram will be computed for *masked pixels only*. If we do not have a mask or do not want to apply one, we can just provide a value of `None`.
4. **histSize:** This is the number of bins we want to use when computing a histogram. Again, this is a list, one for each channel we are computing a histogram for. The bin sizes do not all have to be the same. Here is an example of 32 bins for each channel: `[32,32,32]`.

5. **ranges:** The range of possible pixel values. Normally, this is $[0, 256]$ for each channel, but if you are using a color space other than RGB (such as HSV), the ranges might be different.

Next up, we'll use the `cv2.calcHist` function to compute our first histogram.

7.2 GRAYSCALE HISTOGRAMS

Now that we have an understanding of the `cv2.calcHist` function, let's write some actual code.

Listing 7.1: grayscale_histogram.py

```
1 from matplotlib import pyplot as plt
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
```

This code isn't very exciting yet. All we are doing is importing the packages we will need, setting up an argument parser, and loading our image. We'll make use of the `matplotlib` package to make plotting our histograms easier.

Listing 7.2: grayscale_histogram.py

```
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Original", image)
```

```
15
16 hist = cv2.calcHist([image], [0], None, [256], [0, 256])
17
18 plt.figure()
19 plt.title("Grayscale Histogram")
20 plt.xlabel("Bins")
21 plt.ylabel("# of Pixels")
22 plt.plot(hist)
23 plt.xlim([0, 256])
24 plt.show()
25 cv2.waitKey(0)
```

Now things are getting a little more interesting. On **Line 13**, we convert the image from the RGB colorspace to grayscale. **Line 16** computes the actual histogram. Go ahead and match the arguments of the code up with the function documentation above. We can see that our first parameter is the grayscale image. A grayscale image has only one channel, hence we a value of $[0]$ for `channels`. We don't have a mask, so we set the mask value to `None`. We will use 256 bins in our histogram, and the possible values range from 0 to 256.

Finally, a call to `plt.plot()` plots our grayscale histogram, the results of which can be seen in Figure 7.1.

Not bad. How do we interpret this histogram? Well, the bins (0-255) are plotted on the x-axis. And the y-axis counts the number of pixels in each bin. The majority of the pixels fall in the range of roughly 60 to 120. Looking at the right tail of the histogram, we see very few pixels in the range 200 to 255. This means that there are very few "white" pixels in the image.

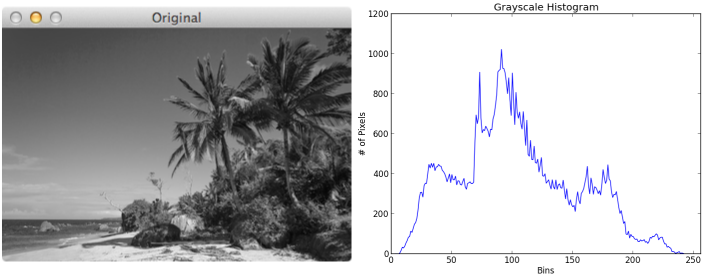

Figure 7.1: Computing a grayscale histogram of our beach image.

7.3 COLOR HISTOGRAMS

In the previous section we explored grayscale histograms. Now let's move on to computing a histogram for each channel of the image.

Listing 7.3: color_histograms.py

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3  import argparse
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
```

Again, we'll import the packages that we'll need, utilizing `matplotlib` once more to plot the histograms.

Let's examine some code:

Listing 7.4: color_histograms.py

```
13 chans = cv2.split(image)
14 colors = ("b", "g", "r")
15 plt.figure()
16 plt.title("'Flattened' Color Histogram")
17 plt.xlabel("Bins")
18 plt.ylabel("# of Pixels")
19
20 for (chan, color) in zip(chans, colors):
21     hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
22     plt.plot(hist, color = color)
23     plt.xlim([0, 256])
```

The first thing we are going to do is split the image into its three channels: blue, green, and red. Normally, we read this is a red, green, blue (RGB). However, OpenCV stores the image as a NumPy array in reverse order: BGR. This is important to note. We then initialize a tuple of strings representing the colors. We take care of all this on **Lines 13-14**.

On **Lines 15-18** we setup our PyPlot figure. We'll plot the bins on the x-axis and the number of pixels placed into each bin on the y-axis.

We then reach a `for` loop on **Line 20**: we start looping over each of the channels in the image.

Then, for each channel we compute a histogram on **Line 21**. The code is identical to that of computing a histogram for the grayscale image; however, we are doing it for each

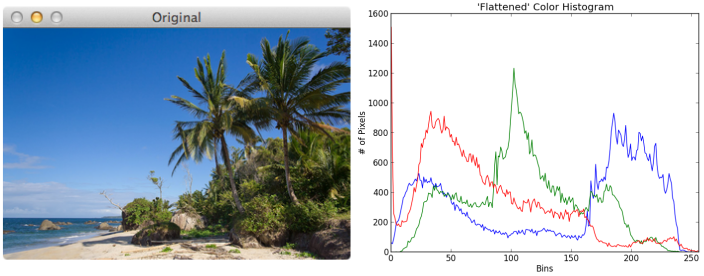

Figure 7.2: Color histograms for each Red, Green, and Blue channel of the beach image.

Red, Green, and Blue channel, allowing us to characterize the distribution of pixel intensities. We add our histogram to the plot on **Line 22**.

We can examine our color histogram in Figure 7.2. We see there is a sharp peak in the green histogram around bin 100. This indicates a darker green value, from the green vegetation and trees in the beach image.

We also see a lot of blue pixels in the range 170 to 225. Considering these pixels are much lighter, we know that they are from the blue sky in our beach image. Similarly, we see a much smaller range of blue pixels in the range 25 to 50 – these pixels are much darker, and are therefore the ocean pixels in the bottom-left corner of the image.

Up until this point, we have computed a histogram for only one channel at a time. Now we move on to multi-

dimensional histograms and take into consideration two channels at a time.

The way I like to explain multi-dimensional histograms is to use the word **AND**.

For example, we can ask a question such as "how many pixels have a Red value of 10 **AND** a Blue value of 30?". How many pixels have a Green value of 200 **AND** a Red value of 130? By using the conjunctive **AND** we are able to construct multi-dimensional histograms.

It's that simple. Let's checkout some code to automate the process of building a 2D histogram:

Listing 7.5: color_histograms.py

```
24 fig = plt.figure()
25
26 ax = fig.add_subplot(131)
27 hist = cv2.calcHist([chans[1], chans[0]], [0, 1], None,
28     [32, 32], [0, 256, 0, 256])
29 p = ax.imshow(hist, interpolation = "nearest")
30 ax.set_title("2D Color Histogram for G and B")
31 plt.colorbar(p)
32
33 ax = fig.add_subplot(132)
34 hist = cv2.calcHist([chans[1], chans[2]], [0, 1], None,
35     [32, 32], [0, 256, 0, 256])
36 p = ax.imshow(hist, interpolation = "nearest")
37 ax.set_title("2D Color Histogram for G and R")
38 plt.colorbar(p)
39
40 ax = fig.add_subplot(133)
41 hist = cv2.calcHist([chans[0], chans[2]], [0, 1], None,
42     [32, 32], [0, 256, 0, 256])
43 p = ax.imshow(hist, interpolation = "nearest")
44 ax.set_title("2D Color Histogram for B and R")
45 plt.colorbar(p)
46
```

```
47 print "2D histogram shape: %s, with %d values" % (
48     hist.shape, hist.flatten().shape[0])
```

Yes, this is a fair amount of code. But that's only because we are computing a 2D color histogram for each combination of RGB channels: Red and Green, Red and Blue, and Green and Blue.

Now that we are working with multi-dimensional histograms, we need to keep in mind the number of bins we are using. In previous examples, I've used 256 bins for demonstration purposes. However, if we used a 256 bins for each dimension in a 2D histogram, our resulting histogram would have 65,536 separate pixel counts. Not only is this wasteful of resources, it's not practical. Most applications using somewhere between 8 and 64 bins when computing multi-dimensional histograms. As **Lines 27-28** show, I am now using 32 bins instead of 256.

The most important take away from this code can be seen by inspecting the first arguments to the `cv2.calcHist` function. Here we see that we are passing in a list of two channels: the Green and Blue channels. And that's all there is to it.

So how is a 2D histogram stored in OpenCV? It's a 2D NumPy array. Since I used 32 bins for each channel, I now have a 32×32 histogram.

How do we visualize a 2D histogram? Let's take a look at Figure 7.3 where we see three graphs. The first is a 2D color histogram for the Green and Blue channels, the second for Green and Red, and the third for Blue and Red. Shades of blue represent low pixel counts, whereas shades

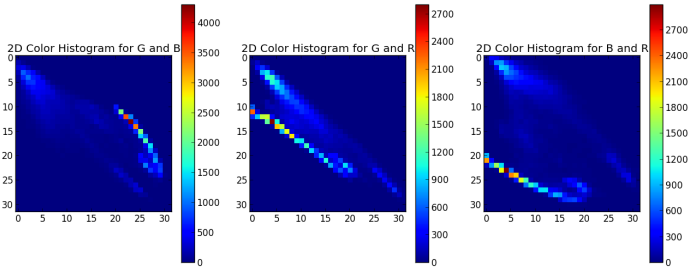

Figure 7.3: Computing 2D color histograms for each combination of Red, Green, and Blue channels.

of red represent large pixel counts (i.e. peaks in the 2D histogram). We tend to see many peaks in the Green and Blue histogram, where $x = 22$ and $y = 12$. This corresponds to the green pixels of the vegetation and trees and the blue of the sky and ocean.

Using a 2D histogram takes into account two channels at a time. But what if we wanted to account for all *three* RGB channels? You guessed it. We're now going to build a 3D histogram.

Listing 7.6: color_histograms.py

```
49 hist = cv2.calcHist([image], [0, 1, 2],
50     None, [8, 8, 8], [0, 256, 0, 256, 0, 256])
51 print "3D histogram shape: %s, with %d values" % (
52     hist.shape, hist.flatten().shape[0])
53
54 plt.show()
```

The code here is very simple – it's just an extension from the code above. We are now computing an $8 \times 8 \times 8$ histogram for each of the RGB channels. We can't visualize this histogram, but we can see that the shape is indeed `(8,8,8)` with 512 values.

7.4 HISTOGRAM EQUALIZATION

Histogram equalization improves the contrast of an image by "stretching" the distribution of pixels. Consider a histogram with a large peak at the center of it. Applying histogram equalization will stretch the peak out towards the corner of the image, thus improving the global contrast of the image. Histogram equalization is applied to grayscale images.

This method is useful when an image contains foregrounds and backgrounds that are both dark or both light. It tends to produce unrealistic effects in photographs; however, is normally useful when enhancing the contrast of medical or satellite images.

Regardless of whether you are applying histogram equalization to a photograph, a satellite image, or an X-ray, we first need to see some code so we can understand what is going on:

Listing 7.7: equalize.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
```

Figure 7.4: *Left:* The original beach image. *Right:* The beach image after applying histogram equalization.

```
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12
13 eq = cv2.equalizeHist(image)
14
15 cv2.imshow("Histogram Equalization", np.hstack([image, eq]))
16 cv2.waitKey(0)
```

Lines 1-10 handle our standard practice of importing packages, parsing arguments, and loading our image. We then convert our image to grayscale on **Line 11**.

Performing histogram equalization is done using just a single function: `cv2.equalizeHist`, which accepts a single parameter, the grayscale image we want to perform histogram equalization on. The last couple lines of code display our histogram equalized image.

The result of applying histogram equalization can be seen in Figure 7.4. On the *left*, we have our original beach image. Then, on the *right*, we have our histogram equalized beach image. Notice how the contrast of the image has been radically changed and now spans the entire range of $[0, 255]$.

7.5 HISTOGRAMS AND MASKS

In Chapter 6, Section 6.4, I mentioned that masks can be used to focus on only regions of an image that interest us. We are now going to construct a mask and compute color histograms for only the masked region.

First, we need to define a convenience function to save us from writing repetitive lines of code:

Listing 7.8: histogram_with_mask.py

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3  import argparse
4  import cv2
5
6  def plot_histogram(image, title, mask = None):
7      chans = cv2.split(image)
8      colors = ("b", "g", "r")
9      plt.figure()
10     plt.title(title)
11     plt.xlabel("Bins")
12     plt.ylabel("# of Pixels")
13
14     for (chan, color) in zip(chans, colors):
15         hist = cv2.calcHist([chan], [0], mask, [256], [0, 256])
16         plt.plot(hist, color = color)
17         plt.xlim([0, 256])
```

On **Lines 1-4** we import our packages, then on **Line 6** we define `plot_histogram`. This function excepts three parameters: an `image`, the `title` of our plot, and a `mask`. The `mask` defaults to `None` if we do not have a mask for the image.

The body of our `plot_histogram` function simply computes a histogram for each channel in the image and plots it, just as in previous examples in this chapter.

Now that we have a function to help us easily plot histograms, let's move into the bulk of our code:

Listing 7.9: histogram_with_mask.py

```
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-i", "--image", required = True,
20     help = "Path to the image")
21 args = vars(ap.parse_args())
22
23 image = cv2.imread(args["image"])
24 cv2.imshow("Original", image)
25 plot_histogram(image, "Histogram for Original Image")
```

Lines 18-21 parse our command line arguments. Then we load our beach image on **Line 23** and plot a histogram for each channel of the beach image on **Line 25**. The plot for our image can be seen in Figure 7.5. We will refer to this histogram again once we compute a histogram for the masked region.

Listing 7.10: histogram_with_mask.py

```
26 mask = np.zeros(image.shape[:2], dtype = "uint8")
27 cv2.rectangle(mask, (15, 15), (130, 100), 255, -1)
28 cv2.imshow("Mask", mask)
29
30 masked = cv2.bitwise_and(image, image, mask = mask)
31 cv2.imshow("Applying the Mask", masked)
```

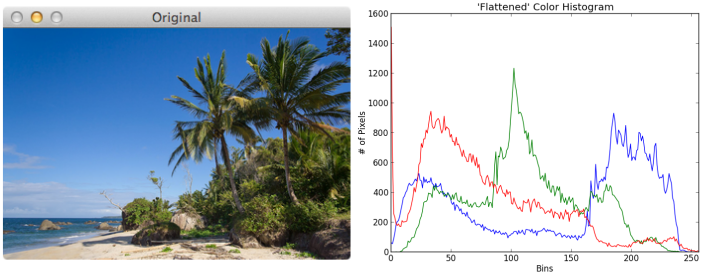

Figure 7.5: *Left:* The original beach image. *Right:* Color histograms for the red, green, and blue channels. Compare these histograms to the histograms of the masked region of blue sky in Figure 7.7.

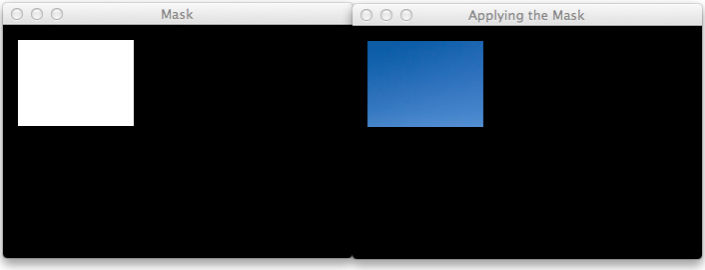

Figure 7.6: *Left:* Our rectangular mask. *Right:* Applying our mask to the beach image using a bitwise AND. Now we see only the blue sky – the rest of the image is ignored.

Now we are ready to construct a mask for the image. We define our `mask` as a NumPy array, with the same width and height as our beach image on **Line 26**. We then draw a white rectangle starting from point $(15, 15)$ to point $(130, 100)$ on **Line 27**. This rectangle will serve as our mask – only pixels in our original image belonging to the masked region will be considered in the histogram computation.

To visualize our mask, we apply a bitwise AND to the beach image (**Line 30**), the results of which can be seen in Figure 7.6. Notice how the image on the *left* is simply a white rectangle, but when we apply our mask to to beach image, we only see the blue sky (*right*).

Listing 7.11: histogram_with_mask.py

```
32 plot_histogram(image, "Histogram for Masked Image", mask = mask)
```

```
33
34 plt.show()
```

Finally, we compute a histogram for our masked image using our `plot_histogram` function and show our results (**Lines 32-34**).

We can see our masked histogram in Figure 7.7. Most red pixels fall in the range $[0, 80]$, indicating that red pixels contribute very little to our image. This makes sense, since our sky is blue. Green pixels are then present, but again, are towards the darker end of the RGB spectrum. Finally, our blue pixels fall in the range brighter range and are obviously our blue sky.

Most importantly compare our masked color histograms in Figure 7.5 to the unmasked color histograms in Figure 7.7 above. Notice how how dramatically different the color histograms are. By utilizing masks, we are able to apply our computation only to the specific regions of the image that interest us – in this example, we simply wanted to examine the distribution of the blue sky.

In this chapter you have learned all about histograms. Histograms are simple, but are used extensively in image processing and computer vision. Make sure you have a good grasp of histograms, you'll be certainly using them in the future!

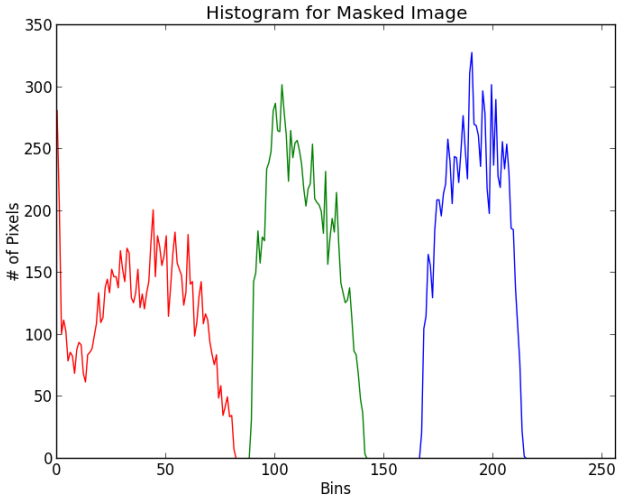

Figure 7.7: The resulting histogram of the masked image in Figure 7.6. Red contributes little to our image and is towards the darker end of the spectrum. Some lighter green values are present, and many light blue colors, corresponding to the sky in the image.

8

SMOOTHING AND BLURRING

I'm pretty sure we all know what blurring is. It's what happens when your camera takes a picture out of focus. Sharper regions in the image lose their detail, normally as a disc/circular shape.

Practically, this means that each pixel in the image is mixed in with its surrounding pixel intensities. This "mixture" of pixels in a neighborhood becomes our blurred pixel.

While this effect is usually unwanted in our photographs, it's actually quite helpful when performing image processing tasks.

In fact, many image processing and computer vision functions, such as thresholding and edge detection, perform better if the image is first smoothed or blurred.

In order to explore different types of blurring methods, let's start with a baseline of our original T-Rex image in Figure 8.1.

Listing 8.1: blurring.py

Figure 8.1: Our original T-Rex image before applying any blurring effects.

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
```

In order to perform image blurring, we first need to import our packages and parse our arguments (**Lines 1-8**). We then load our image and show it as a baseline to compare our blurring methods to on **Line 10 and 11**.

Now that our image is loaded, we can start blurring our images.

8.1 AVERAGING

The first blurring method we are going to explore is averaging.

As the name suggests, we are going to define a $k \times k$ sliding window on top of our image, where k is always an odd number. This window is going to slide from left-to-right and from top-to-bottom. The pixel at the center of this matrix (and hence why we have to use an odd number, otherwise there would not be a true "center") is then set to be the *average* of all other pixels surrounding it.

We call this sliding window a "convolution kernel" or just a "kernel". We'll continue to use this terminology throughout this chapter.

As we will see, as the size of the kernel increases, the more blurred our image will become.

Let's check out some code to perform average blurring:

Listing 8.2: blurring.py

```
12 blurred = np.hstack([
13     cv2.blur(image, (3, 3)),
14     cv2.blur(image, (5, 5)),
15     cv2.blur(image, (7, 7))])
16 cv2.imshow("Averaged", blurred)
17 cv2.waitKey(0)
```

Figure 8.2: Performing averaging blurring with a 3×3 kernel (*left*), 5×5 kernel (*middle*), and 7×7 kernel (*right*).

In order to average blur an image, we use the `cv2.blur` function. This function requires two arguments: the image we want to blur and the size of the kernel. As lines **13-15** show, we blur our image with increasing sizes kernels. The larger our kernel becomes, the more blurred our image will appear.

We make use of the `np.hstack` function to stack our output images together. This method "horizontally stacks" our three images into a row. This is useful since we don't want to create three separate windows using the `cv2.imshow` function.

The output of our averaged blur can be seen in Figure 8.2. The image on the left is barely blurred, but by the time we reach a kernel of size 7×7, we see that our T-Rex is very blurry indeed. Perhaps he was running at a high speed and chasing a jeep?

Figure 8.3: Performing Gaussian blurring with a 3×3 kernel (*left*), 5×5 kernel (*middle*), and 7×7 kernel (*right*). Again, our image becomes more blurred as the kernel size increases, but is less blurred than the average method in Figure 8.2.

8.2 GAUSSIAN

Next up, we are going to review Gaussian blurring. Gaussian blurring is similar to average blurring, but instead of using a simple mean, we are now using a weighted mean, where neighborhood pixels that are closer to the central pixel contribute more "weight" to the average.

The end result is that our image is less blurred, but more naturally blurred, than using the average method discussed in the previous section.

Let's look at some code to perform Gaussian blurring:

Listing 8.3: blurring.py

```
18 blurred = np.hstack([
19     cv2.GaussianBlur(image, (3, 3), 0),
```

```
20      cv2.GaussianBlur(image, (5, 5), 0),
21      cv2.GaussianBlur(image, (7, 7), 0)])
22  cv2.imshow("Gaussian", blurred)
23  cv2.waitKey(0)
```

Here you can see that we are making use of the `cv2.GaussianBlur` function on **Lines 19-21**. The first argument to the function the image we want to blur. Then, similar to `cv2.blur`, we provide a tuple representing our kernel size. Again, we start with a small kernel size of 3×3 and start to increase it.

The last parameter is our σ, the standard deviation in the x-axis direction. By setting this value to 0, we are instructing OpenCV to automatically compute them based on our kernel size.

We can see the output of our Gaussian blur in Figure 8.3. Our images have less of a blur effect than when using the averaging method in Figure 8.2; however, the blur itself is more natural, due to the computation of the weighted mean, rather than allowing all pixels in the kernel neighborhood to have equal weight.

8.3 MEDIAN

Traditionally, the median blur method has been most effective when removing salt-and-pepper noise. This type of noise is exactly what it sounds like: imagine taking a photograph, putting it on your dining room table, and sprinkling salt and pepper on top of it. Using the median blur method, you could remove the salt and pepper from your image.

When applying a median blur, we first define our kernel size k. Then, as in the averaging blurring method, we consider all pixels in the neighborhood of size $k \times k$. But, unlike the averaging method, instead of replacing the central pixel with the average of the neighborhood, we instead replace the central pixel with the median of the neighborhood.

The reason median blurring is more effective at removing salt-and-pepper style noise from an image is that each central pixel is always replaced with a pixel intensity that exists in the image.

Methods such as averaging and Gaussian compute means or weighted means for the neighborhood – this average pixel intensity may or may not be present in the neighborhood. But by definition, the median pixel *must* exist in our neighborhood. By replacing our central pixel with a median rather than an average, we can substantially reduce noise.

Now, it's time to apply our median blur:

Listing 8.4: blurring.py

```
24 blurred = np.hstack([
25     cv2.medianBlur(image, 3),
26     cv2.medianBlur(image, 5),
27     cv2.medianBlur(image, 7)])
28 cv2.imshow("Median", blurred)
29 cv2.waitKey(0)
```

Applying a median blur is accomplished by making a call to the `cv2.medianBlur` function. This method takes two parameters: the image we want to blur and the size of our kernel. On **Lines 25-27**, we start off with a kernel size of

Figure 8.4: Applying the median blur method to our T-Rex image with increasing kernel sizes of 3 (*left*), 5 (*middle*), and 7 (*right*), respectively. Notice that we are no longer creating a "motion blur".

3, then increase it to 5 and 7. The resulting blurred images are then stacked and displayed to us.

Our median blurred images can be seen in Figure 8.4. Notice that we are no longer creating a "motion blur" effect like in averaging and Gaussian blurring – instead, we are removing detail and noise.

For example, take a look at the color of the scales of the T-Rex. As our kernel size increases, the scales become less pronounced. The black and brown stripes on the legs and tail of the T-Rex especially lose their detail, all without creating a motion blur.

8.4 BILATERAL

The last method we are going to explore is bilateral blurring.

Thus far, the intention of our blurring methods have been to reduce noise and detail in an image; however, we tend to lose edges in the image.

In order to reduce noise while still maintaining edges, we can use bilateral blurring. Bilateral blurring accomplishes this by introducing two Gaussian distributions.

The first Gaussian function only considers spatial neighbors, that is, pixels that appear close together in the (x, y) coordinate space of the image. The second Gaussian then models the pixel intensity of the neighborhood, ensuring that only pixels with similar intensity are included in the actual computation of the blur.

Overall, this method is able to preserve edges of an image, while still reducing noise. The largest downside to this method is that it is considerably slower than its averaging, Gaussian, and median blurring counterparts.

Let's look at some code:

Listing 8.5: blurring.py

```
30 blurred = np.hstack([
31     cv2.bilateralFilter(image, 5, 21, 21),
32     cv2.bilateralFilter(image, 7, 31, 31),
33     cv2.bilateralFilter(image, 9, 41, 41)])
34 cv2.imshow("Bilateral", blurred)
35 cv2.waitKey(0)
```

Figure 8.5: Applying Bilateral blurring to our beach image. As the diameter of the neighborhood, color σ, and space σ increases (from left to right), our image has noised removed, yet still retains edges and does not appear to be "motion blurred".

We apply bilateral blurring by call the `cv2.bilateralFilter` function on **Lines 31-33**. The first parameter we supply is the image we want to blur. Then, we need to define the diameter of our pixel neighborhood. The third argument is our color σ. A larger value for color σ means that more colors in the neighborhood will be considered when computing the blur. Finally, we need to supply the space σ. A larger value of space σ means that pixels farther out from the central pixel will influence the blurring calculation, provided that their colors are similar enough.

We obtain results using for increasing neighborhood sizes, color σ, and space σ. These results can be seen in Figure 8.5. As the size of our parameters increases, our image has noise removed, yet the edges still remain.

Now that we know how to blur our images, we can move on to thresholding in the next chapter. You can be sure that we'll make use of blurring throughout the rest of this book!

9

THRESHOLDING

Thresholding is the binarization of an image. In general, we seek to convert a grayscale image to a binary image, where the pixels are either 0 or 255.

A simple thresholding example would be selecting a pixel value p, and then setting all pixel intensities less than p to zero, and all pixel values greater than p to 255. In this way, we are able to create a binary representation of the image.

Normally, we use thresholding to focus on objects or areas of particular interest in an image. In the examples in the sections below, we will empty out our pockets and look at our spare change. Using thresholding methods, we'll be able to find the coins in an image.

9.1 SIMPLE THRESHOLDING

Applying simple thresholding methods requires human intervention. We must specify a threshold value T. All pixel intensities below T are set to 0. And all pixel intensities greater than T are set to 255.

We could also apply the inverse of this binarization by setting all pixels below T to 255 and all pixel intensities greater than T to 0.

Let's explore some code to apply simple thresholding methods:

Listing 9.1: simple_thresholding.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
```

On **Lines 1-10** we import our packages, parse our arguments, and load our image. From there, we convert the image from the RGB color space to grayscale on **Line 11**.

At this point, we apply Gaussian blurring on **Line 12** with a $\sigma = 5$ radius. Applying Gaussian blurring helps remove some of the high frequency edges in the image that we are not concerned with.

Listing 9.2: simple_thresholding.py

```
14 (T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
15 cv2.imshow("Threshold Binary", thresh)
16
17 (T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.
       THRESH_BINARY_INV)
```

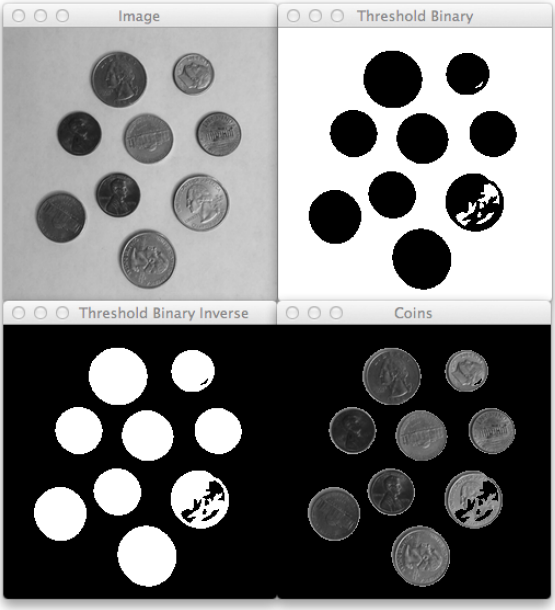

Figure 9.1: *Top-Left:* The original coins image in grayscale. *Top-Right:* Applying simple binary thresholding. The coins are shown in black and the background in white. *Bottom-Left:* Applying inverse binary thresholding. The coins are now white and the background is black. *Bottom-Right:* Applying the inverse binary threshold as a mask to the grayscale image. We are now focused on only the coins in the image.

```
18 cv2.imshow("Threshold Binary Inverse", threshInv)
19
20 cv2.imshow("Coins", cv2.bitwise_and(image, image, mask =
       threshInv))
21 cv2.waitKey(0)
```

After the image is blurred, we compute the thresholded image on **Line 14** using the `cv2.threshold` function. This method requires four arguments. The first is the grayscale image that we wish to threshold. We supply our blurred image here.

Then, we manually supply our T threshold value. We use a value of $T = 155$.

Our third argument is our maximum value applied during thresholding. Any pixel intensity p that is greater than T, is set to this value. In our example, any pixel value that is greater than 155 is set to 255. Any value that is less than 155 is set to zero.

Finally, we must provide a thresholding method. We use the `cv2.THRESH_BINARY` method, which indicates that pixel values p greater than T are set to the maximum value (the third argument).

The `cv2.threshold` function returns two values. The first is T, the value we manually specified for thresholding. The second is our actual thresholded image.

We then show our thresholded image in Figure 9.1, *Top-Right*. We can see that our coins are now black pixels and the white pixels are the background.

On **Line 17** we apply inverse thresholding rather than normal thresholding by using `cv2.THRESH_BINARY_INV` as our thresholding method. As we can see in Figure 9.1, *Bottom-Left*, our coins are now white and the background is black. This is convenient as we will see in a second.

The last task we are going to perform is to reveal the coins in the image and hide everything else.

Remember when we discussed masking? That will come in handy here.

On **Line 20** we perform masking by using the `cv2.bitwise_and` function. We supply our original coin image as the first two arguments, and then our inverted thresholded image as our mask. Remember, a mask only consider pixels in the original image where the mask is greater than zero. Since our inverted thresholded image on **Line 17** does a good job at approximating the areas coins are contained in, we can use this inverted thresholded image as our mask.

Figure 9.1 *Bottom-Right* shows the result of applying our mask – the coins are clearly revealed while the rest of the image is hidden.

9.2 ADAPTIVE THRESHOLDING

One of the downsides of using simple thresholding methods is that we need to manually supply our threshold value T. Not only does finding a good value of T require a lot of manual experiments and parameter tunings, it's not very

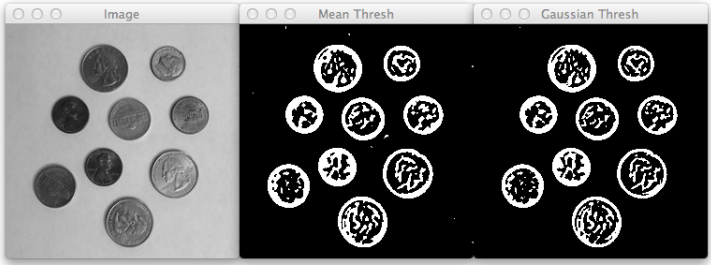

Figure 9.2: *Left:* The grayscale coins image. *Middle:* Applying adaptive thresholding using mean neighborhood values. *Right:* Applying adaptive thresholding using Gaussian neighborhood values.

helpful if the image exhibits a lot of range in pixel intensities.

Simply put, having just one value of T might not suffice.

In order to overcome this problem, we can use adaptive thresholding, which considers small neighbors of pixels and then finds an optimal threshold value T for each neighbor. This method allows us to handle cases where there may be dramatic ranges of pixel intensities and the optimal value of T may change for different parts of the image.

Let's go ahead and jump into some code that applies adaptive thresholding:

Listing 9.3: adaptive_thresholding.py

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
14
15 thresh = cv2.adaptiveThreshold(blurred, 255,
16     cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 4)
17 cv2.imshow("Mean Thresh", thresh)
18
19 thresh = cv2.adaptiveThreshold(blurred, 255,
20     cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 15, 3)
21 cv2.imshow("Gaussian Thresh", thresh)
22 cv2.waitKey(0)
```

Lines 1-10 once again handle setting up our example. We import our packages, construct our argument parser, and load the image. Just as in our simple thresholding example above, we then convert the image to grayscale and blur it slightly on **Lines 11 and 12**.

We then apply adaptive thresholding to our blurred image using the `cv2.adaptiveThreshold` function on **Line 15**. The first parameter we supply is the image we want to threshold. Then, we supply our maximum value of 255, similar to simple thresholding mentioned above.

The third argument is our method to compute the threshold for the current neighborhood of pixels. By supplying `cv2.ADAPTIVE_THRESH_MEAN_C` we indicate that we want to compute the mean of the neighborhood of pixels and treat

it as our T value.

Fourthly, we need our thresholding method. Again, the description of this parameter is identical to the simple thresholding method mentioned above. We use `cv2.THRESH_BINARY_INV` to indicate that any pixel intensity greater than T in the neighborhood should be set to 255, otherwise it should be set to 0.

The next parameter is our neighborhood size. This integer value must be odd and indicates how large our neighborhood of pixels is going to be. We supply a value of 11, indicating that we are going to examine 11×11 pixel regions of the image, instead of trying to threshold the image globally, as in simple thresholding methods.

Finally, we supply a parameter simply called C. This value is an integer that is subtracted from the mean, allowing us to fine tune our thresholding. We use $C = 4$ in this example.

The results of applying mean weighted adaptive thresholding can be seen in the *middle* image of Figure 9.2.

Besides applying standard mean thresholding, we can also apply Gaussian (weighted mean) thresholding, as we do on **Line 19**. The order of the parameters are identical to **Line 15**, but now we are tuning a few of the values.

Instead of supplying a value of `cv2.ADAPTIVE_THRESH_MEAN_C`, we instead use `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` to indicate we want to use the weighted mean. We are also using a 15×15 pixel neighborhood size rather than

an 11×11 neighborhood size as in the previous example. We also alter our C value (the value we subtract from the mean) slightly and use 3 rather than 4.

The results of applying Gaussian adaptive thresholding can be seen in the *right* image of Figure 9.2. There is little difference between the two images.

In general, choosing between mean adaptive thresholding and Gaussian adaptive thresholding requires a few experiments on your end. The most important parameters to vary are the neighborhood size and C, the value you subtract from the mean. By experimenting with this values, you will be able to dramatically change the results of your thresholding.

9.3 OTSU AND RIDDLER-CALVARD

Another way we can automatically compute the threshold value of T is to use Otsu's method.

Otsu's method assumes there are two peaks in the grayscale histogram of the image. In then tries to find an optimal value to separate these two peaks – thus our value of T.

While OpenCV provides support for Otsu's method, I prefer the implementation by Luis Pedro Coelho in the `mahotas` package since it is more Pythonic.

Let's jump into some example code:

Listing 9.4: otsu_and_riddler.py

```
1 import numpy as np
2 import argparse
3 import mahotas
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-i", "--image", required = True,
8     help = "Path to the image")
9 args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 blurred = cv2.GaussianBlur(image, (5, 5), 0)
14 cv2.imshow("Image", image)
15
16 T = mahotas.thresholding.otsu(blurred)
17 print "Otsu's threshold: %d" % (T)
```

On **Lines 1-4** we import the packages we will utilize. We have seen `numpy`, `argparse`, and `cv2` before. We are now introducing `mahotas`, another image processing package.

Lines 6-11 then handle our standard practice of parsing arguments and loading our image.

As in previous thresholding examples, we convert the image to grayscale and then blur it slightly.

To compute our optimal value of T, we use the `otsu` function in the `mahotas.thresholding` package. As our output will later show us, Otsu's method finds a value of $T = 137$ that we will use for thresholding.

Listing 9.5: otsu_and_riddler.py

```
18 thresh = image.copy()
19 thresh[thresh > T] = 255
20 thresh[thresh < 255] = 0
```

```
21 thresh = cv2.bitwise_not(thresh)
22 cv2.imshow("Otsu", thresh)
23
24 T = mahotas.thresholding.rc(blurred)
25 print "Riddler-Calvard: %d" % (T)
26 thresh = image.copy()
27 thresh[thresh > T] = 255
28 thresh[thresh < 255] = 0
29 thresh = cv2.bitwise_not(thresh)
30 cv2.imshow("Riddler-Calvard", thresh)
31 cv2.waitKey(0)
```

Applying the thresholding is accomplished on **Lines 18-21**. First, we make a copy of our grayscale image so that we have an image to threshold. **Line 19** then makes any values greater than T white, whereas **Line 20** makes all remaining pixels that are not white into black pixels. We then invert our threshold by using `cv2.bitwise_not`. This is equivalent to applying a `cv2.THRESH_BINARY_INV` thresholding type as in previous examples in this chapter.

The results of Otsu's method can be seen in the *middle* image of Figure 9.3. We can clearly see that the coins in the image have been highlighted.

Another method to keep in mind when finding optimal values for T is the Riddler-Calvard method. Just as in Otsu's method, the Riddler-Calvard method also computes an optimal value of 137 for T. We apply this method on **Line 24** using the `rc` function in `mahotas.thresholding`. Finally, the actual thresholding of the image takes place on **Lines 26-29**, as in the previous example. Given that the values of T are identical for Otsu and Riddler-Calvard, the thresholded image in Figure 9.3 (*right*) is identical to the thresholded image in the center.

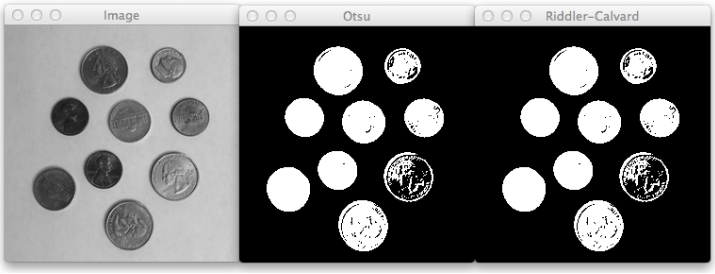

Figure 9.3: *Left:* The original grayscale coins image. *Middle:* Applying Otsu's method to find an optimal value of T. *Right:* Applying the Riddler-Calvard method to find an optimal value of T.

Listing 9.6: otsu_and_riddler.py

```
Otsu's threshold: 137
Riddler-Calvard: 137
```

Now that we have explored thresholding, we will move on to another powerful image processing technique – edge detection.

10

GRADIENTS AND EDGE DETECTION

This chapter is primarily concerned with gradients and edge detection. Formally, edge detection embodies mathematical methods to find points in an image where the brightness of pixel intensities changes distinctly.

The first thing we are going to do is find the "gradient" of the grayscale image, allowing us to find edge like regions in the x and y direction.

We'll then apply Canny edge detection, a multi-stage process of noise reduction (blurring), finding the gradient of the image (utilizing the Sobel kernel in both the horizontal and vertical direction), non-maximum suppression, and hysteresis thresholding.

If that sounds like a mouthful, it's because it is. Again, we won't jump too far into the details since this book is concerned with practical examples of computer vision; however, if you are interested in the mathematics behind gradients and edge detection, I encourage you to read up on the algorithms. Overall, they are not complicated and can be

Figure 10.1: *Left:* The original coins image. *Right:* Applying the Laplacian method to obtain the gradient of the image.

insightful to the behind the scenes action of OpenCV.

10.1 LAPLACIAN AND SOBEL

Let's go ahead and explore some code:

Listing 10.1: sobel_and_laplacian.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 cv2.imshow("Original", image)
```

```
13
14 lap = cv2.Laplacian(image, cv2.CV_64F)
15 lap = np.uint8(np.absolute(lap))
16 cv2.imshow("Laplacian", lap)
17 cv2.waitKey(0)
```

Lines 1-8 import our packages and setup our argument parser. From, there we load our image and convert it to grayscale on **Lines 10 and 11**. When computing gradients and edges we (normally) compute them on a single channel – in this case, we are using the grayscale image; however, we could also compute gradients for each channel of the RGB image. For the sake of simplicity, let's stick with the grayscale image since that is what you will use in most cases.

On **Line 14** we use the Laplacian method to compute the gradient magnitude image by calling the `cv2.Laplacian` function. The first argument is our grayscale image – the image we want to compute the gradient magnitude representation for. The second argument is our data type for the output image.

Throughout this book, we have mainly used 8-bit unsigned integers. Why are we using a 64-bit float now?

The reason involves the transition of black-to-white and white-to-black in the image.

Transitioning from black-to-white is considered a positive slope, whereas a transition from white-to-black is a negative slope. If you remember back to our discussion of image arithmetic in Chapter 6, you'll know that an 8-bit unsigned integer does not represent negative values. Either it will be clipped to zero if you are using OpenCV or a mod-

ulus operation will be performed using NumPy.

The short answer here is that if you don't use a floating point data type when computing the gradient magnitude image, you will miss edges, specifically the white-to-black transitions.

In order to ensure you catch all edges, use a floating point data type, then take the absolute value of the gradient image and convert it back to an 8-bit unsigned integer, as in **Line 15**. This is definitely an important technique to take note of – otherwise you'll be missing edges in your image!

To see the results of our gradient processing, take a look at Figure 10.1.

Let's move on to computing the Sobel gradient representation:

Listing 10.2: sobel_and_laplacian.py

```
18 sobelX = cv2.Sobel(image, cv2.CV_64F, 1, 0)
19 sobelY = cv2.Sobel(image, cv2.CV_64F, 0, 1)
20
21 sobelX = np.uint8(np.absolute(sobelX))
22 sobelY = np.uint8(np.absolute(sobelY))
23
24 sobelCombined = cv2.bitwise_or(sobelX, sobelY)
25
26 cv2.imshow("Sobel X", sobelX)
27 cv2.imshow("Sobel Y", sobelY)
28 cv2.imshow("Sobel Combined", sobelCombined)
```

Using the Sobel operator, we can compute gradient magnitude representations along the x and y axis, allowing us to find both horizontal and vertical edge-like regions.

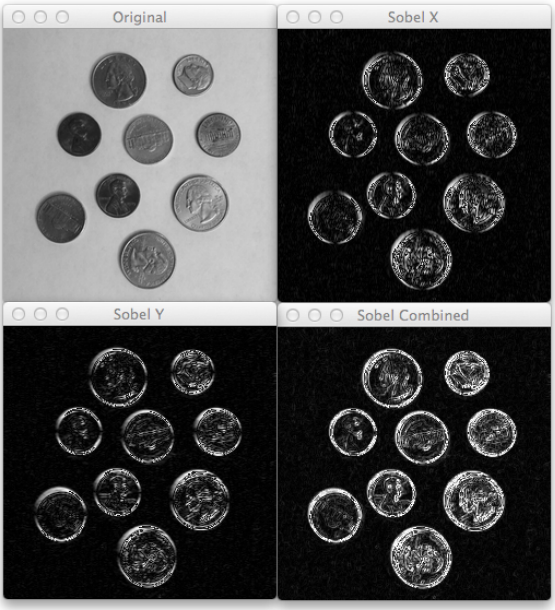

Figure 10.2: *Top-Left:* The original coins image. *Top-Right:* Computing the Sobel gradient magnitude along the x-axis (finding vertical edges). *Bottom-Left:* Computing the Sobel gradient along the y-axis (finding horizontal edges). *Bottom-Right:* Applying a bitwise OR to combine the two Sobel representations.

In fact, that's exactly what **Lines 18 and 19** do by using the `cv2.Sobel` method. The first argument to the Sobel operator is the image we want to compute the gradient representation for. Then, just like in the Laplacian example above, we use a floating point data type. The last two arguments are the order of the derivatives in the x and y direction, respectively. Specify a value of 1 and 0 to find vertical edge-like regions and 0 and 1 to find horizontal edge-like regions

On **Lines 21 and 22** we then ensure we find all edges by taking the absolute value of the floating point image and then converting it to an 8-bit unsigned integer.

In order to combine the gradient images in both the x and y direction, we can apply a bitwise OR. Remember, an OR operation is true when *either* pixel is greater than zero. Therefore, a given pixel will be `True` if either a horizontal or vertical edge is present.

Finally, we show our gradient images on **Lines 26-28**.

You can see the result of our work in Figure 10.2. We start with our original image *Top-Left* and then find vertical edges *Top-Right* and horizontal edges *Bottom-Left*. Finally, we compute a bitwise OR to combine the two directions into a single image *Bottom-Right*.

One thing you'll notice is that the edges are very "noisy". They are not clean and crisp. We'll remedy that by using the Canny edge detector in the next section.

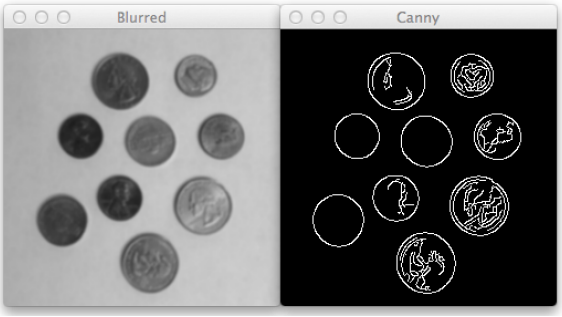

Figure 10.3: *Left:* Our coins image in grayscale and blurred slightly. *Right:* Applying the Canny edge detector to the blurred image to find edges. Notice how our edges more "crisp" and the outlines of the coins are found.

10.2 CANNY EDGE DETECTOR

The Canny edge detector is a multi-step process. It involves blurring the image to remove noise, computing Sobel gradient images in the x and y direction, suppression of edges, and finally a hysteresis thresholding stage that determines if a pixel is "edge-like" or not.

We won't get into all these steps in detail. Instead, we'll just look at some code and show how it's done:

Listing 10.3: canny.py

```
1 import numpy as np
2 import argparse
```

```
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 image = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Blurred", image)
14
15 canny = cv2.Canny(image, 30, 150)
16 cv2.imshow("Canny", canny)
17 cv2.waitKey(0)
```

The first thing we do is import our packages and parse our arguments. We then load our image, convert it to grayscale, and blur it using the Gaussian blurring method. By applying a blur prior to edge detection, we will help remove "noisey" edges in the image that are not of interest to us. Our goal here is to find *only* the outlines of the coins.

Applying the Canny edge detector is performed on **Line 15** using the `cv2.Canny` function. The first argument we supply is our blurred, grayscale image. Then, we need to provide two values: `threshold1` and `threshold2`.

Any gradient value larger than `threshold2` are considered to be an edge. Any value below `threshold1` are considered not to be an edge. Values in between `threshold1` and `threshold2` are either classified as edges or non-edges based on how their intensities are "connected". In this case, any gradient values below 30 are considered non-edges whereas any value above 150 are considered edges.

We then show the results of our edge detection on **Line 16**.

Figure 10.3 shows the results of the Canny edge detector. The image on the *left* is our grayscale, blurred image that we pass into the Canny operator. The image on the *right* is the result of applying the Canny operator.

Notice how the edges are more "crisp". We have substantially less noise than we used the Laplacian or Sobel gradient images. Furthermore, the outline of our coins are clearly revealed.

In the next chapter we'll continue to make use of the Canny edge detector and use it to count the number of coins in our image.

11

CONTOURS

In the previous chapter we explored how to find edges in an image of coins.

Now we are going to use these edges to help us find the actual coins in the image count them.

OpenCV provides methods to find "curves" in an image, called contours. A contour is a curve of points, with no gaps in the curve. Contours are extremely useful for such things as shape approximation and analysis.

In order to find contours in an image, you need to first obtain a binarization of the image, using either edge detection methods or thresholding. In the examples below, we'll use the Canny edge detector to find the outlines of the coins, and then find the actual contours of the coins.

Ready?

Here we go:

11.1 COUNTING COINS

Listing 11.1: counting_coins.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(gray, (11, 11), 0)
13 cv2.imshow("Image", image)
14
15 edged = cv2.Canny(blurred, 30, 150)
16 cv2.imshow("Edges", edged)
```

The first 10 lines of code simply setup our environment by importing packages, parsing arguments, and loading the image.

Just as in the edge detection methods discussed in the previous chapter, we are going to convert our image to grayscale and then apply a Gaussian blur, making it easier for the edge detector to find the outline of the coins. We use a much larger blurring size this time, with $\sigma = 11$. All this is handled on **Lines 10-12**.

We then obtain the edged image by applying the Canny edge detector on **Line 15**. Again, just as in previous edge detection examples, any gradient values below 30 are considered non-edges whereas any value above 150 are considered edges.

Listing 11.2: counting_coins.py

```
17 (cnts, _) = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL, cv2
       .CHAIN_APPROX_SIMPLE)
18
19 print "I count %d coins in this image" % (len(cnts))
20
21 coins = image.copy()
22 cv2.drawContours(coins, cnts, -1, (0, 255, 0), 2)
23 cv2.imshow("Coins", coins)
24 cv2.waitKey(0)
```

Now that we have the outlines of the coins, we can find the contours of the outlines. We do this using the `cv2.findContours` function on **Line 17**. This method returns a tuple of the contours themselves, `cnts`, and the hierarchy of the contours (see below).

The first argument is our edged image. It's important to note that this function is destructive to the image you pass in. If you intend on using that image later on in your code, it's best to make a copy of it, using the NumPy `copy` method.

The second argument is the type of contours we want. We use `cv2.RETR_EXTERNAL` to retrieve only the outermost contours (i.e. the contours that follow the outline of the coin). We could also pass in `cv2.RETR_LIST` to grab *all* contours. Other methods include hierarchical contours using `cv2.RETR_COMP` and `cv2.RETR_TREE`, but hierarchical contours are outside the scope of this book.

Our last argument is how we want to approximate the contour. We use `cv2.CHAIN_APPROX_SIMPLE` to compress horizontal, vertical, and diagonal segments into only their endpoints. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we could pass in `cv2.CHAIN_APPROX_NONE`; however,

be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

Our contours `cnts` is simply a Python list. We can use the `len` function on it to count the number of contours that were returned. We do this on **Line 19** to show how many contours we have found.

When we execute our script, we will have the output "I count 9 coins in this image" printed out to our console.

Now we are able to draw our contours. In order not to draw on our original image, we make a copy of the original image, called `coins` on **Line 21**.

A call to `cv2.drawContours` draws the actual contours on our image. The first argument to the function is the image we want to draw on. The second is our list of contours. Next, we have the contour index. By specifying a negative value of -1, we are indicating that we want to draw *all* of the contours. However, we would also supply an index i, which would be the i'th contour in `cnts`. This would allow us to draw only a *single* contour rather than *all* of them.

For example, here is some code to draw the first, second, and third contours respectively:

Listing 11.3: Drawing Contours via an Index

```
1 cv2.drawContours(coins, cnts, 0, (0, 255, 0), 2)
2 cv2.drawContours(coins, cnts, 1, (0, 255, 0), 2)
3 cv2.drawContours(coins, cnts, 2, (0, 255, 0), 2)
```

The fourth argument to the `cv2.drawContours` function is the color of the line we are going to draw. Here, we use a green color.

Finally, our last argument is the thickness of the line we are drawing. We'll draw the contour with a thickness of two pixels.

Now that our contours are drawn on the image, we can visualize them on **Line 23**.

Take a look at Figure 11.1 to see the results of our work. On the *left* is our original image. Then, we apply Canny edge detection to find the outlines of the coins *middle*. Finally, we find the contours of the coin outlines and draw them. You can see that each contour has been drawn with a two pixel thick green line.

But we're not done yet!

Let's crop each individual coin from the image:

Listing 11.4: counting_coins.py

```
25 for (i, c) in enumerate(cnts):
26     (x, y, w, h) = cv2.boundingRect(c)
27
28     print "Coin #%d" % (i + 1)
29     coin = image[y:y + h, x:x + w]
30     cv2.imshow("Coin", coin)
31
32     mask = np.zeros(image.shape[:2], dtype = "uint8")
33     ((centerX, centerY), radius) = cv2.minEnclosingCircle(c)
34     cv2.circle(mask, (int(centerX), int(centerY)), int(radius),
           255, -1)
35     mask = mask[y:y + h, x:x + w]
```

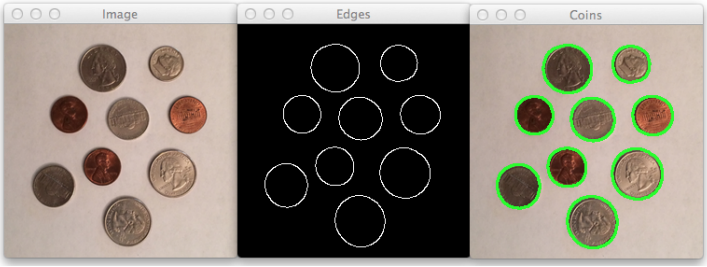

Figure 11.1: *Left:* The original coin image. *Middle:* Applying the Canny edge detector to find the outlines of the coins. *Right:* Finding the contours of the coin outlines and then drawing the contours. We have now successfully found the coins and are able to count them.

```
36      cv2.imshow("Masked Coin", cv2.bitwise_and(coin, coin, mask =
            mask))
37      cv2.waitKey(0)
```

We start off on **Line 25** by looping over our contours.

We then use the `cv2.boundingRect` function on the current contour. This method finds the "enclosing box" that our contour will fit into, allowing us to crop it from the image. The function takes a single parameter, a contour, and then returns a tuple of the x and y position that the rectangle starts at, followed by the width and height of the rectangle.

We then crop the coin from the image using our bounding box coordinates and NumPy array slicing on **Line 29**. The coin itself is shown to us on **line 30**.

If we can find the bounding box of a contour, why not fit a circle to the contour as well? Coins are circles, after all.

We first initialize our mask on **Line 32** as a NumPy array of zeros, with the same width and height of our original image.

A call to `cv2.minEnclosingCircle` on **Line 33** fits a circle to our contour. We pass in a `circle` variable, the current contour, and are given the x and y coordinates of the circle, along with its radius.

Using the (x, y) coordinates and the radius we can draw a circle on our mask, representing the coin. Drawing circles was covered in Chapter 5, Section 5.2.

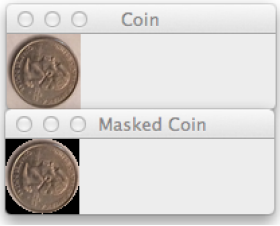

Figure 11.2: *Top:* Cropping the coin by finding the bounding box and applying NumPy array slicing. *Bottom:* Fitting a circle to the contour and masking the coin.

We then crop the mask in the exact same manner as we cropped the coin on **Line 35**.

In order to show only the foreground of the coin and ignore the background, we make a call to our trusty bitwise AND function using the coin image and the mask for the coin. The coin, with the background removed, is shown to us on **Line 36**.

Figure 11.2 shows the output of our hard work. The *top* figure shows that we cropped the coin by finding the bounding box and applying NumPy array slicing. The *bottom* image then shows our masking of the coin by fitting a circle to the contour. The background is removed and only

the coin is shown.

As you can see, contours are extremely powerful tools to have in our toolbox. They allow us to count objects in images and allow us to extract these objects from images. We are just scratching the surface of what contours can do, so be sure to play around with them and explore for yourself! It's the best way to learn!

12

WHERE TO NOW?

In this book we've explored many image processing and computer vision techniques, including basic image processing, such as translation, rotating, and resizing. We learned all about image arithmetic and how to apply bitwise operations. Then, we explored how a simple technique like masking can be used to focus our attention and computation to only a single part of an image.

To better understand the pixel intensity distribution of an image, we then explored histograms. We started by computing grayscale histograms, then worked our way up to color, including 2D and 3D color histograms. We adjusted the contrast of images using histogram equalization, then moved on to blurring our images, using different methods, such as averaging, Gaussian, and median filtering.

We thresholded our images to find objects of interest, then applied edge detection.

Finally, learned how to use contours to count the number of coins in the image.

So where do you go from here?

You continue learning, exploring and experimenting!

Use the source code and images provided in this book to create projects of your own. That's the best way to learn!

If you need project ideas, be sure to contact me. I love talking with readers and helping out when I can. You can reach me at adrian@pyimagesearch.com.

Finally, I constantly post on my blog, www.PyImageSearch.com, new and interesting techniques related to computer vision and image search engines. Be sure to follow the blog for new posts, along with new books as I write them.