pyimagesearch gurus
(https://gurus.pyimagesearch.com/)

1.3: Drawing

Using NumPy array slices in **Section 1.2** (https://gurus.pyimagesearch.com/lessons/image-basics/) 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 lesson, we'll review the three most basic methods to draw shapes: `cv2.line` , `cv2.rectangle` , and `cv2.circle` .

Feedback

While this lesson is by no means a complete, exhaustive overview of the drawing capabilities of OpenCV, it will nonetheless provide a quick, hands-on approach to get you started drawing immediately. As we work through some of the more advanced topics in the course we'll utilize more drawing functions, but in reality, these three methods are the "big" ones that you'll use most of the time.

Objectives:

The main objective of this module is to become familiar with the `cv2.line` , `cv2.rectangle` , and `cv2.circle` functions.

Lines and Rectangles

Before we start exploring the drawing capabilities of OpenCV, let's first define the 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:

drawing.py Python

```
1 # import the necessary packages
2 import numpy as np
3 import cv2
4
5 # initialize our canvas as a 300x300 with 3 channels, Red, Green,
6 # and Blue, with a black background
7 canvas = np.zeros((300, 300, 3), dtype="uint8")
```

Lines 1 and 2 import 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 course. 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 7**. We construct a NumPy array using the `np.zeros` method with *300* rows and *300* columns, yielding a *300 x 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.

Feedback

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, and 64-bit floats), but we'll mainly be using `uint8` for the majority of the examples in this lesson.

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

drawing.py Python

```
9  # draw a green line from the top-left corner of our canvas to the
10 # bottom-right
11 green = (0, 255, 0)
12 cv2.line(canvas, (0, 0), (300, 300), green)
13 cv2.imshow("Canvas", canvas)
14 cv2.waitKey(0)
15
16 # now, draw a 3 pixel thick red line from the top-right corner to the
17 # bottom-left
18 red = (0, 0, 255)
19 cv2.line(canvas, (300, 0), (0, 300), red, 3)
20 cv2.imshow("Canvas", canvas)
21 cv2.waitKey(0)
```

The first thing we do on **Line 11** 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 12**.

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)* — again, remember that the Python language is zero-index. 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 13 and 14** show our image and then wait for a keypress (see **Figure 1** below).

Feedback

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

On **Lines 18-21** we define the color red 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:

`drawing.py` Python

```
23 # draw a green 50x50 pixel square, starting at 10x10 and ending at 60x60
24 cv2.rectangle(canvas, (10, 10), (60, 60), green)
25 cv2.imshow("Canvas", canvas)
26 cv2.waitKey(0)
27
28 # draw another rectangle, this time we'll make it red and 5 pixels thick
29 cv2.rectangle(canvas, (50, 200), (200, 225), red, 5)
30 cv2.imshow("Canvas", canvas)
31 cv2.waitKey(0)
32
33 # let's draw one last rectangle: blue and filled in
34 blue = (255, 0, 0)
35 cv2.rectangle(canvas, (200, 50), (225, 125), blue, -1)
36 cv2.imshow("Canvas", canvas)
37 cv2.waitKey(0)
```

On **Line 24** 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 x 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 29** provides that thickness argument 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 **Section 1.2** (https://gurus.pyimagesearch.com/lessons/image-basics/)?

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

Line 35 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.

The output of drawing our lines and rectangles can be seen below:

Feedback

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/drawing_lines_rectangles.jpg)

FIGURE 1: DRAWING LINES AND RECTANGLES USING OPENCV.

Feedback

As you can see, the output matches our code. We were able to draw a *green line* from the top-left corner to the bottom-right corner, followed by a *thicker red line* from the top-right corner to the bottom-left corner.

We were also able to draw a green rectangle, a slightly thicker red rectangle, and a completely filled in blue rectangle.

That's really all there is to it! You now have a solid grasp of drawing rectangles. In the next section, we'll move on to drawing circles.

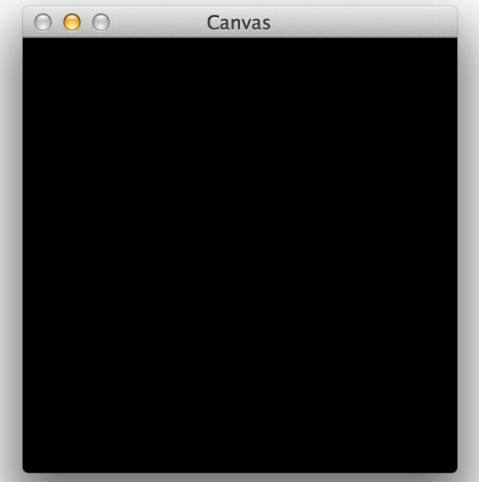
Circles

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

drawing.py Python

```
39 # reset our canvas and draw a white circle at the center of the canvas with
40 # increasing radii - from 25 pixels to 150 pixels
41 canvas = np.zeros((300, 300, 3), dtype="uint8")
42 (centerX, centerY) = (canvas.shape[1] // 2, canvas.shape[0] // 2)
43 white = (255, 255, 255)
44
45 for r in range(0, 175, 25):
46     cv2.circle(canvas, (centerX, centerY), r, white)
47
48 # show our work of art
49 cv2.imshow("Canvas", canvas)
50 cv2.waitKey(0)
```

On **Line 41** we re-initialize our `canvas` to be blank:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/drawing_blank_canvas.jpg)

FIGURE 2: EXAMPLE OF OUR BLANK CANVAS — NOTICE HOW THE LINES AND RECTANGLES ARE GONE.

The rectangles are gone! We need a fresh canvas to draw our circles.

Line 42 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 43** defines a white pixel (i.e. the buckets for each of the Red, Green, and Blue components are "full").

Now, let's draw some circles!

Feedback

On **Line 45** we loop over a number of radius values, starting from *0* and ending at *150*, incrementing by *25* at each step. The `range` function is *exclusive*; therefore, we specify a stopping value of *175* rather than *150*.

To demonstrate this for yourself, open up a Python shell and execute the following code:

Basics of the xrange function — Shell

```
$ python
>>> list(range(0, 175, 25))
[0, 25, 50, 75, 100, 125, 150]
```

Notice how the the output of `range` stops at *150* and does not include *175*.

Line 46 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 49 and 50 then show our image and wait for a keypress.

So what does our image look like? Take a look below to see:

Feedback

FIGURE 3: DRAWING A BULLSEYE USING THE CV2.CIRCLE FUNCTION.

Check out **Figure 3** 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:

drawing.py Python

```
52 # let's go crazy and draw 25 random circles
53 for i in range(0, 25):
54     # randomly generate a radius size between 5 and 200, generate a random
55     # color, and then pick a random point on our canvas where the circle
56     # will be drawn
57     radius = np.random.randint(5, high=200)
58     color = np.random.randint(0, high=256, size = (3,)).tolist()
59     pt = np.random.randint(0, high=300, size = (2,))
60
61     # draw our random circle
62     cv2.circle(canvas, tuple(pt), radius, color, -1)
63
64 # Show our masterpiece
65 cv2.imshow("Canvas", canvas)
66 cv2.waitKey(0)
```

Feedback

Our code starts off on **Line 53** 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, 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 57**. This value controls how large our circle will be.

Next, we randomly generate a color on **Line 58**. 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.

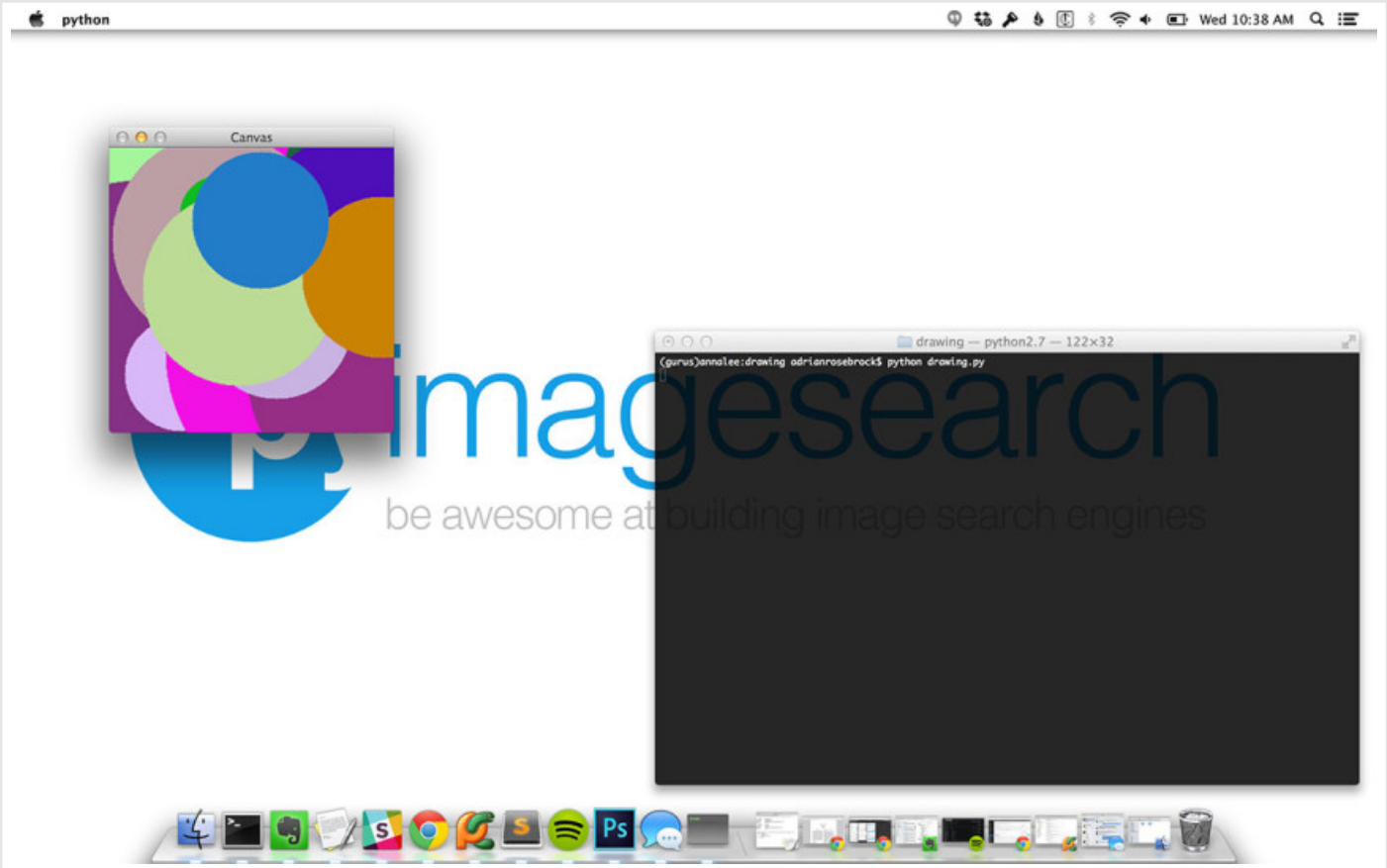
Lastly, we need an *(x, y)* center point to draw our circle. We'll generate a point in the range *[0, 300)*, again

using Numpy's `np.random.randint` function.

The drawing of our circle then takes place on **Line 62**, 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 65 and 66**.

You can check out our work in below:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/drawing_random_circles.jpg)

FIGURE 4: THE RESULTS OF OUR MASTERPIECE. NOTICE THAT EACH CIRCLE IS RANDOMLY PLACED ON THE CANVAS WITH A RANDOM COLOR.

Notice how each circle has a different size, color, and placement on our canvas.

Up until this point we have only explored to draw shapes on a *blank canvas*. But what if we wanted to draw shapes on an *existing image*?

It turns out that the code to draw shapes on an existing image is exactly the same as if you were drawing on a blank canvas generated from NumPy.

Feedback

To demonstrate this, take a look at some code:

```
drawing.py                                                    Python
68 # load the image of Adrian in Florida
69 image = cv2.imread("florida_trip.png")
70
71 # draw a circle around my face, two filled in circles covering my eyes, and
72 # a rectangle surrounding my mouth
73 cv2.circle(image, (168, 188), 90, (0, 0, 255), 2)
74 cv2.circle(image, (150, 164), 10, (0, 0, 255), -1)
75 cv2.circle(image, (192, 174), 10, (0, 0, 255), -1)
76 cv2.rectangle(image, (134, 200), (186, 218), (0, 0, 255), -1)
77
78 # show the output image
79 cv2.imshow("Output", image)
80 cv2.waitKey(0)
```

The first thing we do is load the image of myself on a trip to Florida off disk on **Line 69**.

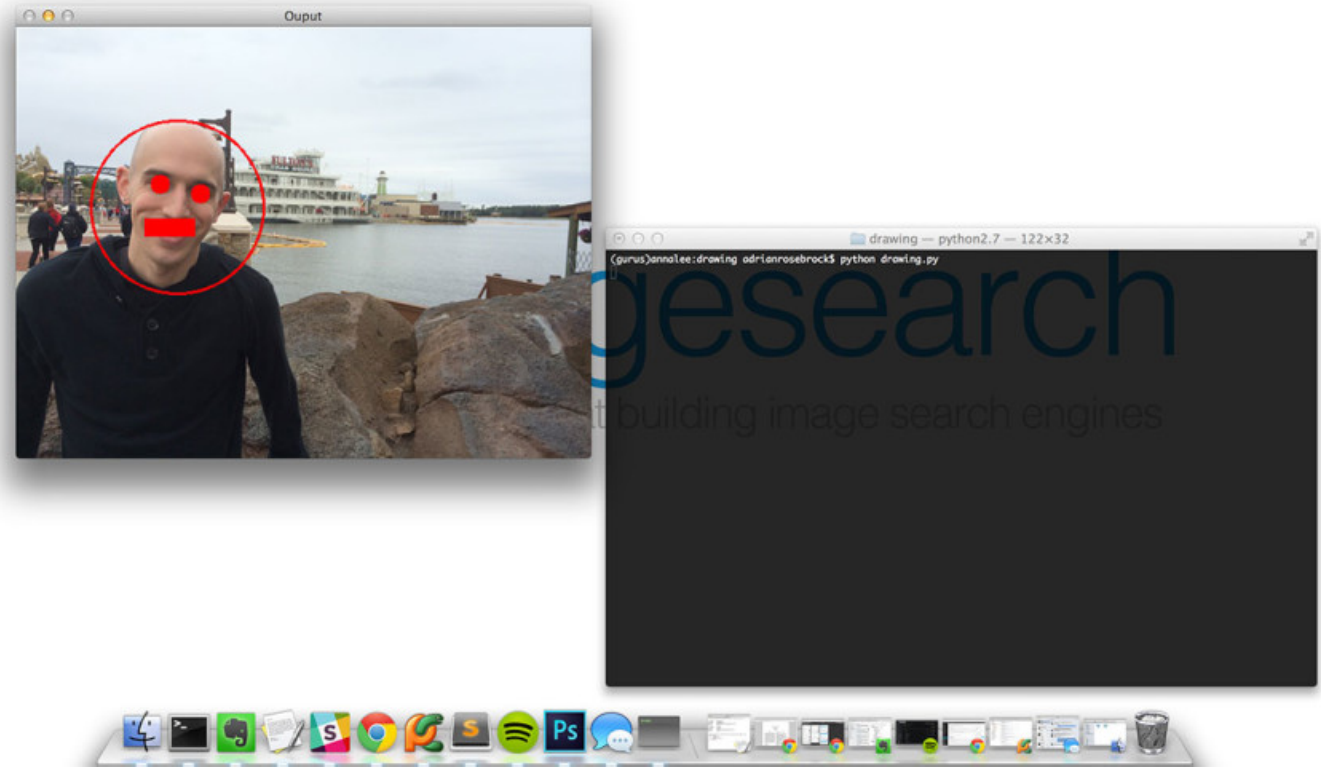
First, we draw a circle surrounding my head on **Line 73**.

And then we draw two circles on circles on **Lines 74 and 75** — these circles are actually where my eyes are in the original image.

Finally, we draw a rectangle around my mouth on **Line 76** and display the output of our work on **Lines 79 and 80**.

Feedback

We can see our results here:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/drawing_faces.jpg)

FIGURE 5: DRAWING SHAPES ON AN EXISTING IMAGE.

Feedback

As you can see, there is *no difference* between drawing shapes on existing images versus blank canvases — it's all the same!

To execute our Python script, download the code from the bottom of this article, navigate to the source code directory, and execute the following command:

drawing.py Shell

```
1 $ python drawing.py
```

Your output should match the screenshots detailed above in this article.

Summary

In this section 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 course.

Downloads

Download the Code (https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/drawing.zip)

	Quizzes	Status
1	Drawing Quiz (https://gurus.pyimagesearch.com/quizzes/drawing-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/image-basics/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/basic-image-processing/)

Course Progress

Feedback

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

© 2018 PyImageSearch. All Rights Reserved.

Feedback