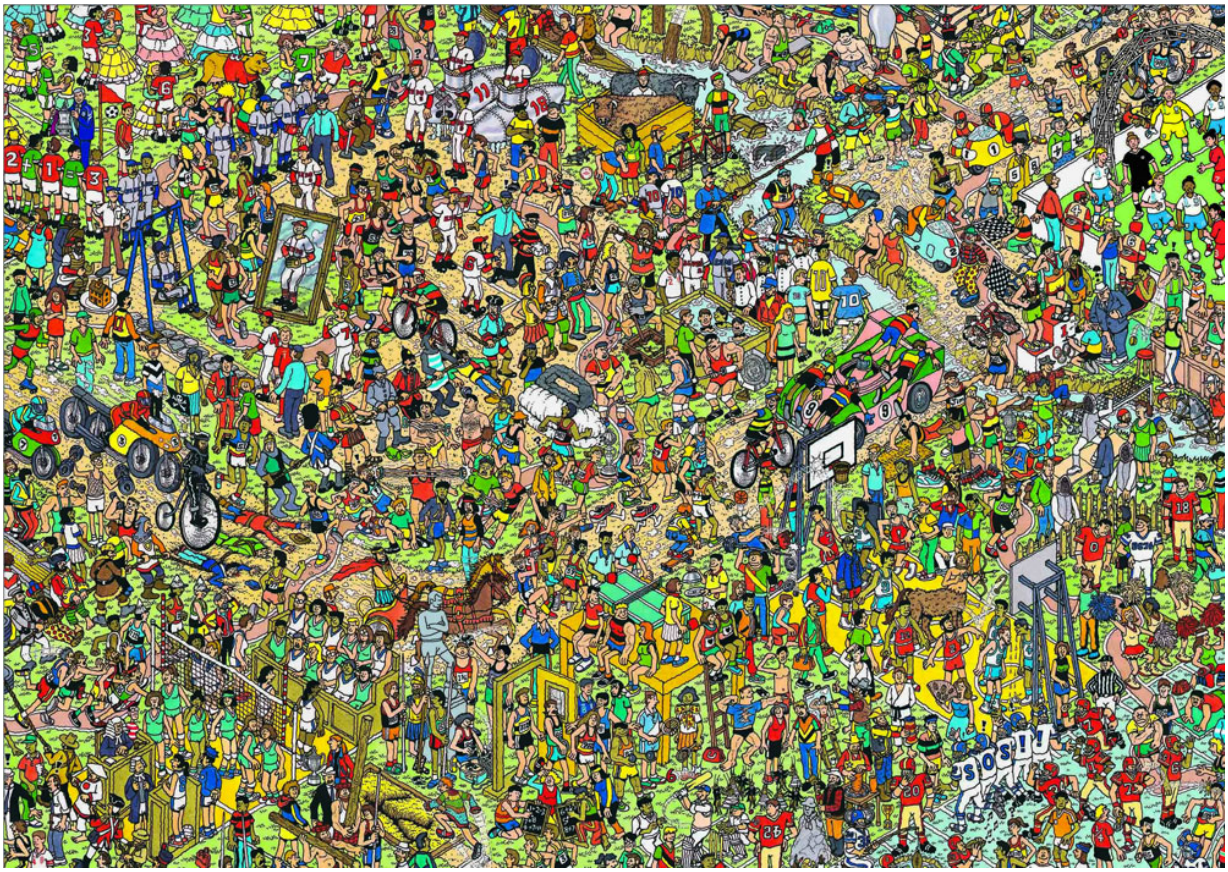
Using OpenCV, Python and Template Matching to play “Where’s Waldo?”

by Adrian Rosebrock



Take a look at the Where’s Waldo puzzle above. How long does it take you to find Waldo? 10 seconds? 30 seconds? Over a minute?

Waldo is the ultimate game of hide and seek for the human eye. He’s actually “hiding” in plain sight — but due to all the noise and distraction, we can’t pick him out immediately!

At the core, Waldo is just a visual pattern. He wears glasses. A hat. And his classic white and red horizontally striped shirt. It might take us a little bit of time to scan up and down and left to right across the page, but our brain is able to pick out this pattern, even amongst all the distraction.

The question is, can computers do better? Can we create a program that can ***automatically*** find Waldo?

In fact, we can.

Using computer vision techniques we can find Waldo in under a second, much faster than any of us could!

In this blog post I'll show you how to use the OpenCV and template matching functions to find that pesky Waldo who is always hiding in plain sight.

Here's a quick overview of what we're going to do:

- **What we're going to do:** Build a Python script using OpenCV that can find Waldo in a "Where's Waldo?" puzzle.
- **What you'll learn:** How to utilize Python, OpenCV, and template matching using `cv2.matchTemplate` and `cv2.minMaxLoc`. Using these functions we will be able to find Waldo in our puzzle image.
- **What you need:** Python, NumPy, and OpenCV. A little knowledge of basic image processing concepts would help, but is *definitely not a requirement*. This how-to guide is meant to be hands on and show you how to apply template matching using OpenCV. Don't have these libraries installed? No problem. I created a pre-configured virtual machine with all the necessary computer vision, image processing, and machine learning packages pre-installed. Click here to learn more.
- **Assumptions:** I'll assume that you have NumPy and OpenCV installed in either the python2.6 or python2.7 environment. Again, you can download a pre-configured virtual machine with all the necessary packages installed here.

The Goal

So what's the overall goal of the Python script we are going to create?

The goal, given a query image of Waldo and the puzzle image, is to find Waldo in in the puzzle image and highlight his location.

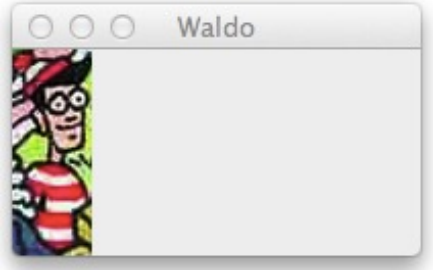
As you'll see later in this post, we'll be able to accomplish this in *only two lines of Python code.* The rest of the code simply handles logic such as argument parsing and displaying the solved puzzle to our screen.

Our Puzzle and Query Image

We require two images to build our Python script to perform template matching.

The first image is the Where's Waldo puzzle that we are going to solve. You can see our puzzle image in **Figure 1** at the top of this post.

The second image is our query image of Waldo:



Using our Waldo query image we are going to find him in the original puzzle.

Unfortunately, here is where the practicality of our approach breaks down.

In order to find Waldo in our puzzle image, we first need the image of Waldo himself. And you may be asking, if I already have the image of Waldo, why I am I playing the puzzle?

Good point.

Using computer vision and image processing techniques to find Waldo in a image *is certainly possible.*

However, it requires some slightly more advanced techniques such as:

1. Filtering out colors that are not red.
2. Calculating the correlation of a striped pattern to match the red and white transitions of Waldo's shirt.
3. Binarization of the regions of the image that have high correlation with a striped pattern.

This post is meant to be an introduction to basic computer vision techniques such as template matching. Later on we can dive into more advanced techniques. Where's Waldo was just a cool and simple way to perform template matching that I *just had* to share with you!

Getting Our Hands Dirty

Ready to see some code? Alright, let's do this:

```
# import the necessary packages
import numpy as np
import argparse
import imutils
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--puzzle", required = True,
    help = "Path to the puzzle image")
ap.add_argument("-w", "--waldo", required = True,
    help = "Path to the waldo image")
args = vars(ap.parse_args())

# load the puzzle and waldo images
puzzle = cv2.imread(args["puzzle"])
waldo = cv2.imread(args["waldo"])
(waldoHeight, waldoWidth) = waldo.shape[:2]
```

Lines 1-13 simply imports the packages we are going to use and configures our argument parser. We'll use NumPy for array manipulations, `argparse` to parse our command line arguments, and `cv2` for our OpenCV bindings. The package `imutils` is actually a set of convenience functions to handle basic image manipulations such as rotation, resizing, and translation. You can read more about these types of basic image operations here.

From there, we need to setup our two command line arguments. The first, `--puzzle` is the path to our Where's Waldo puzzle image and `--waldo` is the path to Waldo query image.

Again, our goal here is to find the query image in the puzzle image using template matching.

Now that we have the paths to our images, we load them off of disk on **Line 16 and 17** using the `cv2.imread` function — this method simply reads the image off disk and then stores it as a multi-dimensional NumPy array.

Since images are represented as NumPy arrays in OpenCV, we can easily access the dimensions of the image. On **Line 18** we grab the height and the width of the Waldo query image, respectively.

We are now ready to perform our template matching:

```
# find the waldo in the puzzle
result = cv2.matchTemplate(puzzle, waldo, cv2.TM_CCOEFF)
(_, _, minLoc, maxLoc) = cv2.minMaxLoc(result)
```

We accomplish our template matching on **Line 21** by using the `cv2.matchTemplate` function. This method requires three parameters. The first is our `puzzle` image, the image that contains what we are searching for. The second is our query image, `waldo`. This image is contained within the puzzle

image and we are looking to pinpoint its location. Finally, the third argument is our template matching method. There are a variety of methods to perform template matching, but in this case we are using the correlation coefficient which is specified by the flag `cv2.TM_CCOEFF`.

So what exactly is the `cv2.matchTemplate` function doing?

Essentially, this function takes a "sliding window" of our `waldo` query image and slides it across our puzzle image from left to right and top to bottom, one pixel at a time. Then, for each of these locations, we compute the correlation coefficient to determine how "good" or "bad" the match is. Regions with sufficiently high correlation can be considered "matches" for our waldo template.

From there, all we need is a call to `cv2.minMaxLoc` on **Line 22** to find where our "good" matches are.

That's really all there is to template matching!

And realistically, it only took us two lines of code.

The rest of our source code involves extracting the region that contains Waldo and then highlighting him in the original puzzle image:

```
# the puzzle image
topLeft = maxLoc
botRight = (topLeft[0] + waldoWidth, topLeft[1] + waldoHeight)
roi = puzzle[topLeft[1]:botRight[1], topLeft[0]:botRight[0]]

# construct a darkened transparent 'layer' to darken everything
# in the puzzle except for waldo
mask = np.zeros(puzzle.shape, dtype = "uint8")
puzzle = cv2.addWeighted(puzzle, 0.25, mask, 0.75, 0)
```

Line 26 grabs the top-left (x, y) coordinates of the image that contains the best match based on our sliding window. Then, we compute the bottom-right (x, y) coordinates based on the width and height of our `waldo` image on **Line 27**. Finally we extract this roi (Region of Interest) on **Line 28**.

The next step is to construct a transparent layer that darkens everything in the image *but* Waldo. We do this by first initializing a `mask` on **Line 32** with the same shape as our puzzle filled with zeros. By filling the image with zeros we are creating an image filled with black.

In order to create the transparent effect, we use the `cv2.addWeighted` function on **Line 33**. The first parameter is our puzzle image, and the second parameter indicates that we want it to contribute to 25% of our output image. We then supply our mask as the third parameter, allowing it to contribute to 75% of our output image. By utilizing the `cv2.addWeighted` function we have been able to create the transparency effect.

However, we still need to highlight the Waldo region! That's simple enough:

```
# put the original waldo back in the image so that he is
# 'brighter' than the rest of the image
puzzle[topLeft[1]:botRight[1], topLeft[0]:botRight[0]] = roi

# display the images
cv2.imshow("Puzzle", imutils.resize(puzzle, height = 650))
cv2.imshow("Waldo", waldo)
cv2.waitKey(0)
```

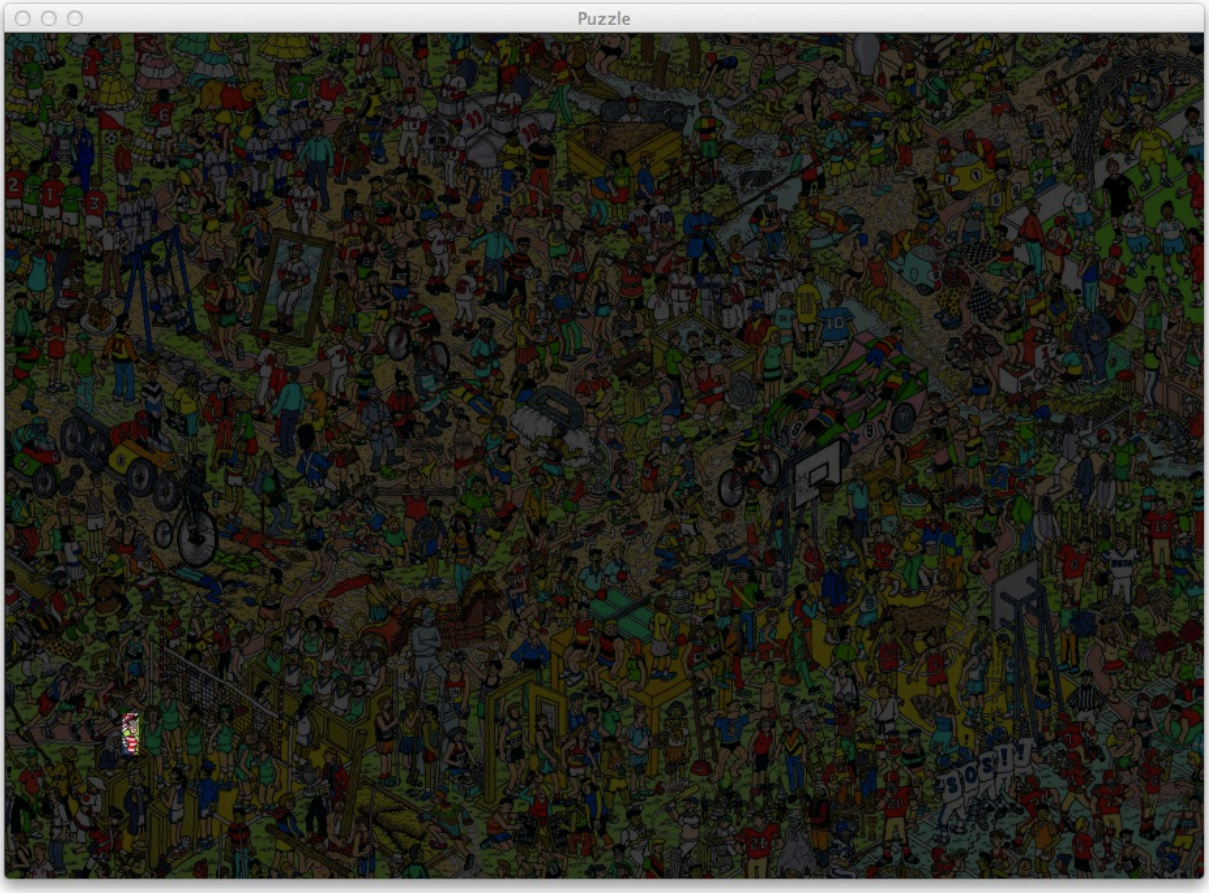
Here we are just placing the Waldo ROI back into the original image using some NumPy array slicing techniques on **Line 37**. Nothing to it.

Finally, **Lines 40-42** display the results of our work by displaying our Waldo query and puzzle image on screen and waiting for a key press.

To run our script, fire up your shell and execute the following command:

```
$ python find_waldo.py --puzzle puzzle.png --waldo waldo.png
```

When your script is finished executing you should see something like this on your screen:



We have found Waldo at the bottom-left corner of the image!

So there you have it!

Template matching using Python and OpenCV is actually quite simple. To start, you just need two images — an image of the object you want to match and an image that contains the object. From there, you just need to make calls to `cv2.matchTemplate` and `cv2.minMaxLaoc`. The rest is just wrapper code to glue the output of these functions together.

Learn Computer Vision In A Single Weekend

Of course, we are only scratching the surface of computer vision and image processing. Template matching is just the start.

Luckily, *I can teach you the basics of computer vision in a single weekend.*

I know, it sounds crazy.

But my method really works.

See, I just finished writing my new book, *Practical Python and OpenCV*. I wanted this book to be as hands-on as possible. I wanted something that you could easily learn from, without all the rigor and

details associated with a college level computer vision and image processing course.

The bottom line is that Practical Python and OpenCV is the **best, guaranteed quick start guide** to learning the fundamentals of computer vision and image processing.

Plus, I have created a downloadable Ubuntu VirtualBox virtual machine with OpenCV, PIL, mahotas, scikit-image, scikit-learn, and many other computer vision and image processing libraries pre-configured and pre-installed.

So go ahead, jump start your computer vision education. **Don't waste time installing packages...invest your time learning!**

To learn more about my new book and downloadable virtual machine, **just click here**.

This post originally appeared as a guest post on Machine Learning Mastery.