pyimagesearch
be awesome at building image search engines

How to Build a Kick-Ass Mobile Document Scanner in Just 5 Minutes

by **Adrian Rosebrock** on September 1, 2014 in **Image Processing**, **Tutorials**

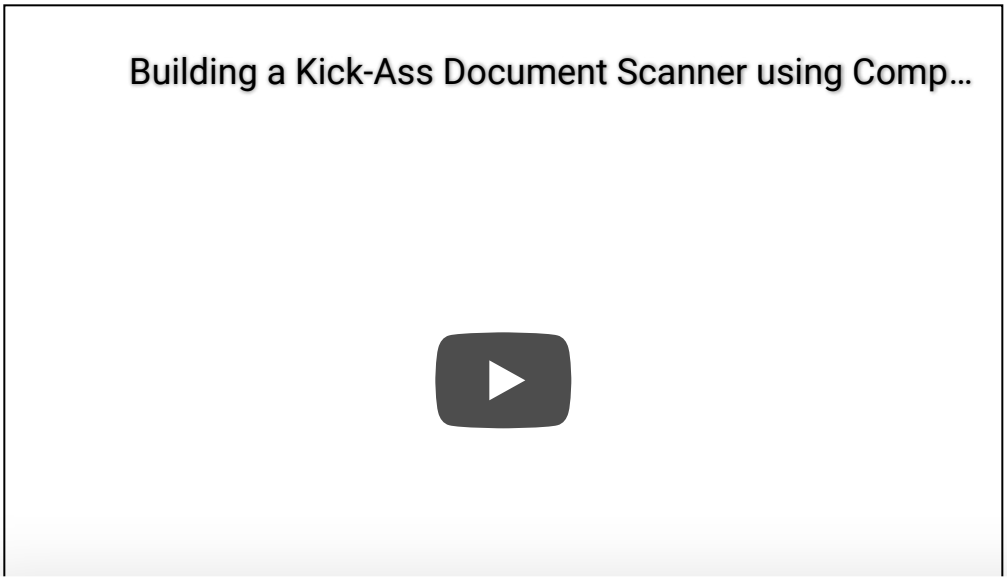
Sound interesting?

Read on. And unlock the secrets to build a mobile scanner app of your own.

Looking for the source code to this post?
Jump right to the downloads section.

OpenCV and Python versions:
This example will run on **Python 2.7/3+** and **OpenCV 2.4/3+**

How To Build a Kick-Ass Mobile Document Scanner in Just 5 Minutes



Building a Document Scanner App using Python, OpenCV, and Computer Visi Python

```python
# import the necessary packages
from pyimagesearch.transform import four_point_transform
from skimage.filters import threshold_local
import numpy as np
import argparse
import cv2
import imutils

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
    help = "Path to the image to be scanned")
args = vars(ap.parse_args())
```

Lines 2-7 handle importing the necessary Python packages that we'll need.

We'll start by importing our `four_point_transform` function which I discussed last week.

We'll also be using the `imutils` module, which contains convenience functions for resizing, rotating, and cropping images. You can read more about `imutils` in my this post. To install `imutils`, simply:

Building a Document Scanner App using Python, OpenCV, and Computer Visio Shell

```
$ pip install --upgrade imutils
```

Next up, let's import the `threshold_local` function from scikit-image. This function will help us obtain the "black and white" feel to our scanned image.

Note (15 January 2018): *The `threshold_adaptive` function has been deprecated. This post has been updated to make use of `threshold_local`.*

Lastly, we'll use NumPy for numerical processing, `argparse` for parsing command line arguments, and `cv2` for our OpenCV bindings.

Lines 10-13 handle parsing our command line arguments. We'll need only a single switch image, `--`

```
24 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
25 gray = cv2.GaussianBlur(gray, (5, 5), 0)
26 edged = cv2.Canny(gray, 75, 200)
27
28 # show the original image and the edge detected image
29 print("STEP 1: Edge Detection")
30 cv2.imshow("Image", image)
31 cv2.imshow("Edged", edged)
32 cv2.waitKey(0)
33 cv2.destroyAllWindows()
```

First, we load our image off disk on **Line 17**.

In order to speedup image processing, as well as make our edge detection step more accurate, we resize our scanned image to have a height of 500 pixels on **Lines 17-20**.

We also take special care to keep track of the `ratio` of the original height of the image to the new height (**Line 18**) — this will allow us to perform the scan on the *original* image rather than the *resized* image.

From there, we convert the image from RGB to grayscale on **Line 24**, perform Gaussian blurring to remove high frequency noise (aiding in contour detection in Step 2), and perform Canny edge detection on **Line 26**.

The output of Step 1 is then shown on **Lines 30 and 31**.

Take a look below at the example document:

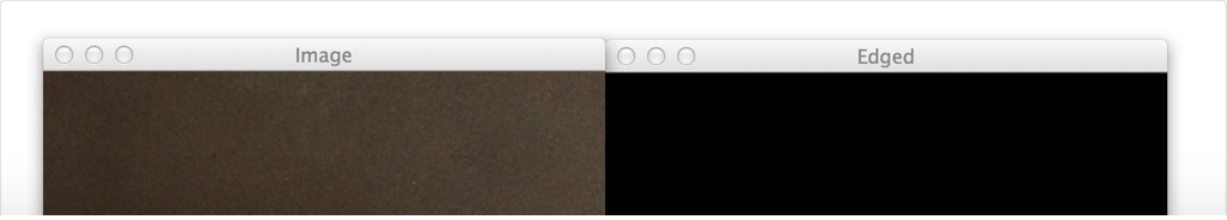


Figure 1: The first step of building a document scanning app. On the *left* we have the original image and on the *right* we have the edges detected in the image.

On the left you can see my receipt from Whole Foods. Notice how the picture is captured at an angle. It is definitely not a 90-degree, top-down view of the page. Furthermore, there is also my desk in the image. Certainly this is not a "scan" of any means. We have our work cut out for us.

However, on the right you can see the image after performing edge detection. We can clearly see the outline of the receipt.

Not a bad start.

Let's move on to Step 2.

Step 2: Finding Contours

Contour detection doesn't have to be hard.

In fact, when building a document scanner, you actually have a *serious advantage…*

Take a second to consider what we're actually building.

A document scanner simply scans in a piece of paper.

A piece of paper is assumed to be a rectangle.

And a rectangle has four edges.

Therefore, we can create a simple heuristic to help us build our document scanner.

The heuristic goes something like this: we'll assume that the *largest contour* in the image *with exactly*

```
45      approx = cv2.approxPolyDP(c, 0.02 * peri, True)
46
47      # if our approximated contour has four points, then we
48      # can assume that we have found our screen
49      if len(approx) == 4:
50          screenCnt = approx
51          break
52
53  # show the contour (outline) of the piece of paper
54  print("STEP 2: Find contours of paper")
55  cv2.drawContours(image, [screenCnt], -1, (0, 255, 0), 2)
56  cv2.imshow("Outline", image)
57  cv2.waitKey(0)
58  cv2.destroyAllWindows()
```

We start off by finding the contours in our edged image on **Line 37**. We also handle the fact that OpenCV 2.4, OpenCV 3, and OpenCV 4 return contours differently on **Line 38**.

A neat performance hack that I like to do is actually sort the contours by area and keep only the largest ones (**Line 39**). This allows us to only examine the largest of the contours, discarding the rest.

We then start looping over the contours on **Line 42** and approximate the number of points on **Line 44 and 45**.

If the approximated contour has four points (**Line 49**), we assume that we have found the document in the image.

And again, this is a fairly safe assumption. The scanner app will assume that (1) the document to be scanned is the main focus of the image and (2) the document is rectangular, and thus will have four distinct edges.

From there, **Lines 55 and 56** display the contours of the document we went to scan.

And now let's take a look at our example image:

Step 3: Apply a Perspective Transform & Threshold

The last step in building a mobile document scanner is to take the four points representing the outline of the document and apply a perspective transform to obtain a top-down, “birds eye view” of the image.

Let’s take a look:

Building a Document Scanner App using Python, OpenCV, and Computer Visi Python

```
60 # apply the four point transform to obtain a top-down
61 # view of the original image
62 warped = four_point_transform(orig, screenCnt.reshape(4, 2) * ratio)
63
64 # convert the warped image to grayscale, then threshold it
65 # to give it that 'black and white' paper effect
66 warped = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)
67 T = threshold_local(warped, 11, offset = 10, method = "gaussian")
68 warped = (warped > T).astype("uint8") * 255
69
70 # show the original and scanned images
71 print("STEP 3: Apply perspective transform")
72 cv2.imshow("Original", imutils.resize(orig, height = 650))
73 cv2.imshow("Scanned", imutils.resize(warped, height = 650))
74 cv2.waitKey(0)
```

Line 62 performs the warping transformation. In fact, all the heavy lifting is handled by the `four_point_transform` function. Again, you can read more about this function in last week’s post.

We’ll pass two arguments into `four_point_transform` : the first is our original image we loaded off disk (*not* the resized one), and the second argument is the contour representing the document, multiplied by the resized ratio.

So, you may be wondering, why are we multiplying by the resized ratio?

We multiply by the resized ratio because we performed edge detection and found contours on the

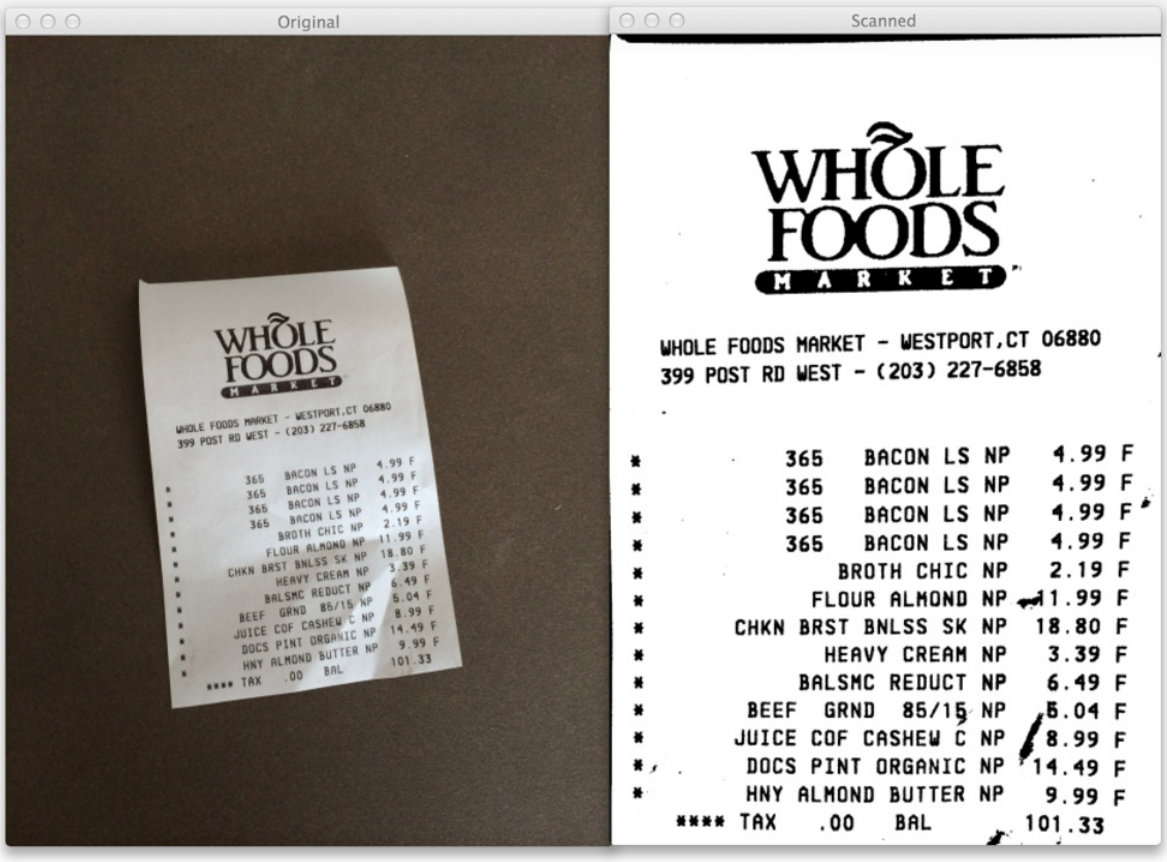


Figure 3: Applying step 3 of our document scanner, perspective transform. The original image is on the *left* and the scanned image on the *right*.

On the left we have the original image we loaded off disk. And on the right, we have the scanned image!

Notice how the perspective of the scanned image has changed — we have a top-down, 90-degree view of the image.

And thanks to our adaptive thresholding, we also have a nice, clean black and white feel to the

```
Building a Document Scanner App using Python, OpenCV, and Computer Visio  Shell
1 $ python scan.py --image images/page.jpg
```

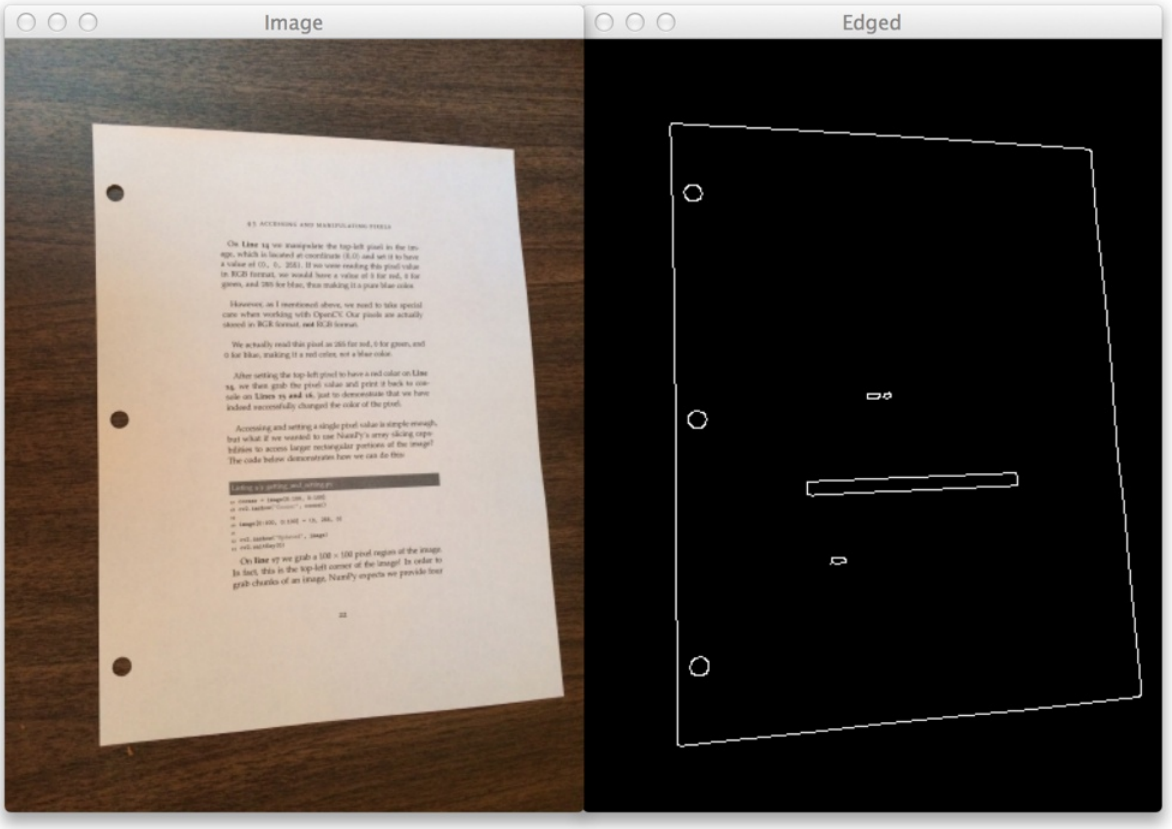
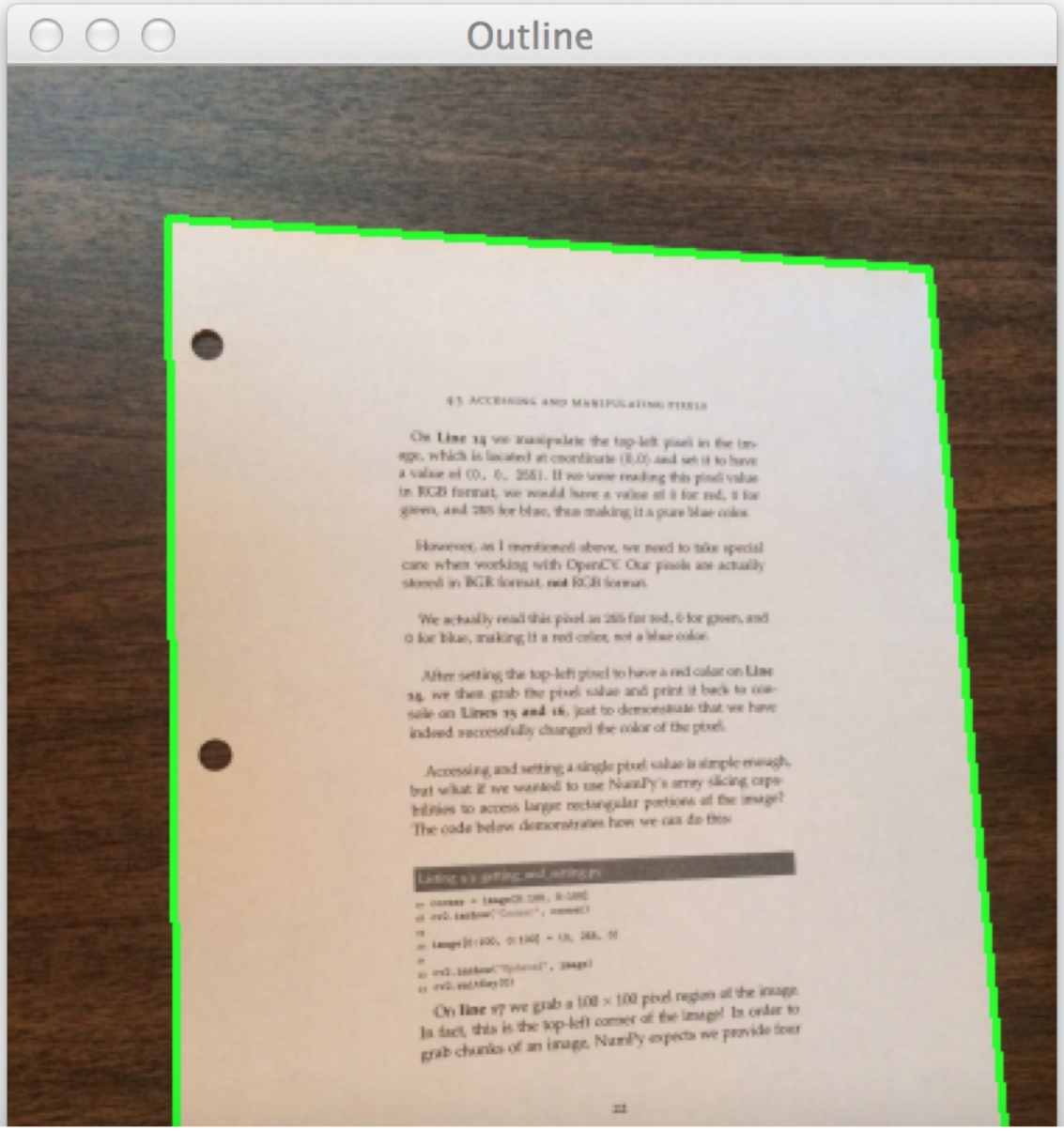


Figure 4: Applying edge detection to scan a document using computer vision.

You can see the original image on the *left* and the edge detected image on the *right*.

Now, let's find the contour of the page:



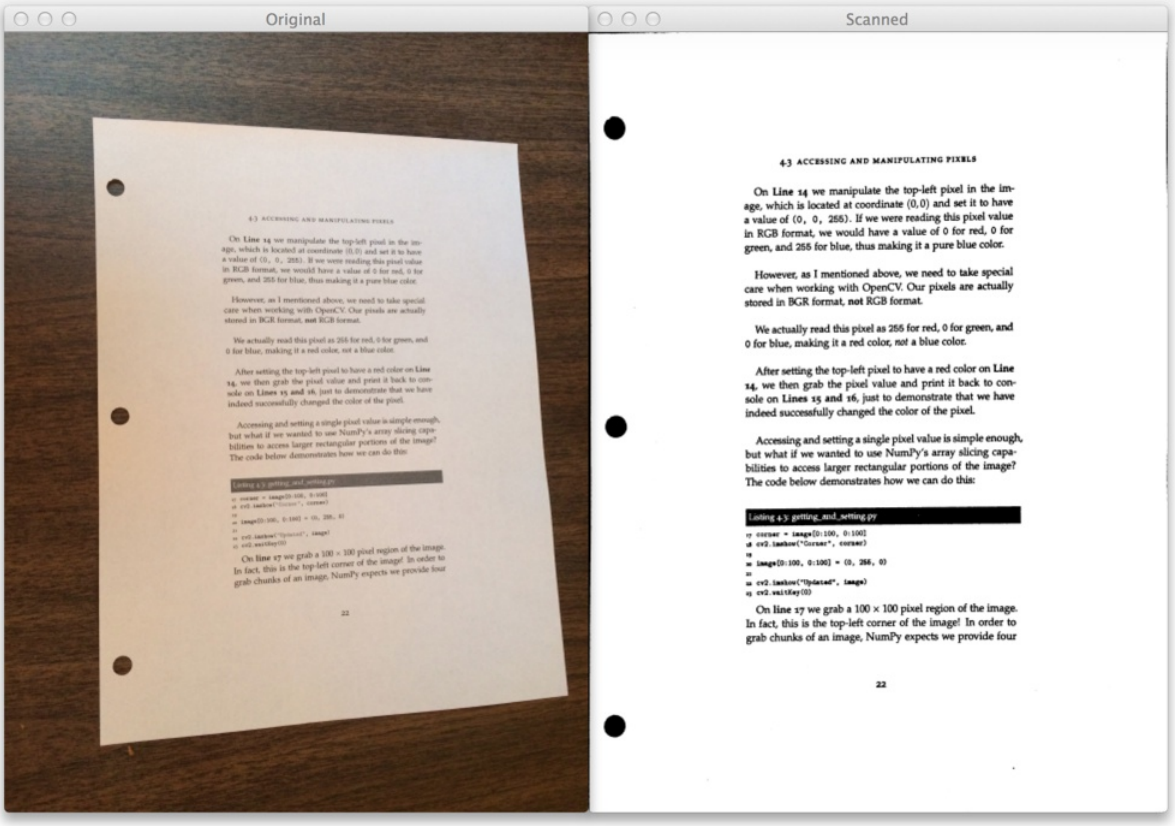


Figure 6: On the *left* we have our original image. And on the *right*, we can see the scanned version. The scan is successful!

Another successful scan!

Where to Next?

Now that you have the code to build a mobile document scanner, maybe you want to build an app and submit to the App Store yourself!

The **second step** is to find the contours in the image that represent the document we want to scan.

And the **final step** is to apply a perspective transform to obtain a top-down, 90-degree view of the image, just as if we scanned the document.

Optionally, you can also apply thresholding to obtain a nice, clean black and white feel to the piece of paper.

So there you have it.

A mobile document scanner in 5 minutes.

Excuse me while I call James and collect my money…

Did You Like this Post?

Hey, did you enjoy this post on building a mobile document scanner?

If so, I think you'll like my book, *Practical Python and OpenCV*.

Inside you'll learn how to **detect faces in images**, **recognize handwriting**, and **utilize keypoint detection and the SIFT descriptors** to build a system to recognize the book covers!

Sound interesting?

Just click here and pickup a copy.

And in a single weekend you'll unlock the secrets the computer vision pros use…*and become a pro yourself!*