

Experiment 3: Thresholding and contours

Adrian F. Clark

This experiment explores the thresholding capabilities of OpenCV and introduces its contour-processing routines through the example of counting the number of dots on several dice.

Contents

1	Introduction	2
2	The program	2
3	Thresholding	3
4	Processing contours	4
5	Counting the dots	5
6	Concluding remarks	6

1 Introduction

This experiment is based on [a tutorial on contours](#) using OpenCV which first touches on thresholding and then its contour functionality. Thresholding is discussed in detail Chapter 5 of the lecture notes but then we considered region labelling rather than contours. Contours are ultimately more powerful but the code to implement it is significantly more complicated and the way in which it is used is definitely more difficult.

The particular image we shall use for this experiment is shown in Figure 1. The aim of this experiment is to isolate each of the visible die faces by thresholding and then count the number of dots on each die's face.



Figure 1: Example image for this experiment

2 The program

The starting point to the experiment is the following program:

```
#!/usr/bin/env python3
"contours -- demo of OpenCV's contour-processing capabilities"
import sys, cv2

# Handle the command line.
if len (sys.argv) < 3:
    print ("Usage:", sys.argv[0], "<image> <threshold>", file=sys.stderr)
    sys.exit (1)
img = cv2.imread (sys.argv[1])
t = int (sys.argv[2])

# Produce a binary image.
```

```

gray = cv2.cvtColor (img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur (gray, (5, 5), 0)
(t, binary) = cv2.threshold (blur, t, 255, cv2.THRESH_BINARY)

# Find contours.
(_, contours, _) = cv2.findContours (binary, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)

# Print a table of the contours and their sizes.
print ("Found %d objects." % len(contours))
for (i, c) in enumerate(contours):
    print ("\tSize of contour %d: %d" % (i, len(c)))

# Draw contours over original image and display the result.
cv2.drawContours (img, contours, -1, (0, 0, 255), 5)
cv2.namedWindow (sys.argv[0], cv2.WINDOW_NORMAL)
ny, nx, nc = img.shape
cv2.resizeWindow (sys.argv[0], nx//2, ny//2)
cv2.imshow (sys.argv[0], img)
cv2.waitKey (0)

```

Rather than type it in, you should download [a zip-file](#) containing it and the example image of Figure 1. Like most programs that work purely by calling OpenCV routines, it is fairly short. It works by reading in an image and blurring it slightly with a Gaussian-shaped mask. It then thresholds the image using an OpenCV function. The resulting binary image is passed to OpenCV's contour-finding function, which essentially returns a list of the contours found. These are printed out and then drawn onto the image for display.

You should run the program on the supplied image and see how well it works. You invoke it with two arguments, the name of the file containing the image to be processed and the value of the threshold; try using 200 first and then see what effect changing it has.

3 Thresholding

You will see that the routine `cv2.threshold` accepts several arguments. Look at its documentation on the Web so that you understand what they all do. While you are doing that, also make sure you understand what the function returns: the thresholded image is only one of its outputs.

One characteristic of the image is that its background is not constant but appears lighter towards the upper right corner. That is not a problem in this case because the upper surfaces of the dice appear so white; but in many tasks, this variation in the background may become problematic. For this reason, OpenCV provides an *adaptive* thresholder, where the threshold relates to regions of the image rather than being global. This routine is used in the following way:

```

binary = cv.adaptiveThreshold (blur, 255, cv.ADAPTIVE_THRESH_MEAN_C,

```

```
cv.THRESH_BINARY, 11, 2)
```

where the last two arguments can be tuned to make the routine perform well — you should read up on them too, to help you work out good values. Comment out the call to `cv2.threshold` and see if the adaptive threshold works as well. Can you identify any difference in the run-times of the two versions?

One irritation of both versions of the thresholder that you have used is that the value of the threshold has to be provided by you on the command line. The underlying aim of computer vision is for the entire process to be automated, so it makes sense to work out the value of the threshold from the image if at all possible. Lectures described the use of Otsu's approach, which works well when the image is bimodal (*i.e.*, its histogram has two peaks).

You can use Otsu's method with `cv2.threshold` too, using a call like

```
thresh, binary = cv.threshold (blur, 0, 255,
                               cv.THRESH_BINARY + cv.THRESH_OTSU)
```

This has an additional option in the last argument and sets the threshold argument to zero; together, these tell `cv2.threshold` to compute the threshold value using Otsu's method. The first argument returned from the call, stored in `thresh` here, is the threshold value computed.

Does Otsu's method work well in this case? What rules of thumb were mentioned in the lecture regarding its use? Are they met here?

4 Processing contours

Let us now turn our attention to finding and processing contours in the image. The call to `cv2.findContours` passes in three arguments and receives three outputs, two of which are currently ignored. The first parameter in the call is easy enough, the image in which the contours are to be found. This image should be binary, with the objects for which contours are to be found in white and a black background.

The second argument is a constant indicating what kind of contours are to be found. As we are currently interested in the main objects in the image, we want only contours around the outermost edges of objects and so we pass in `cv2.RETR_EXTERNAL`. If we required more information, such as the contours of the dots marked on a die's face, then we would use another parameter such as `cv2.RETR_TREE` or `cv2.RETR_CCOMP`. (We shall come back to this shortly.) The last parameter tells `cv2.findContours` whether or not it should simplify the contours; we use `cv2.CHAIN_APPROX_SIMPLE`, which tells it to simplify by using line segments when it can — that saves memory and computation time here.

`cv2.findContours` returns a tuple of three values. The first is an intermediate image produced during the contour-finding process; we are not interested in it here so we store it in a junk variable. The second return value is a list of numpy arrays, each holding the points for a single contour in the image. The final return value is a

numpy array that contains hierarchy information about the contours, again not of interest to us here.

Knowing this, the remainder of the program should be fairly straightforward to understand.

5 Counting the dots

The program initially finds only the external contours of objects. If we change the third argument from `cv2.RETR_EXTERNAL` to `cv2.RETR_TREE`, it will also return internal contours — and we can use this to count the number of dots on each die’s face.

When using `cv2.RETR_TREE`, the contours are arranged in a hierarchy, with the outermost contours for each object at the top. Moving down the hierarchy, each new level of contours represents the next innermost contour for each object.

We obtain that information about the contour hierarchies via the third return value from the `cv2.findContours` call:

```
(junk, contours, hierarchy) = cv2.findContours(binary,
                                             cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

The third return value, saved in `hierarchy` here, is a three-dimensional numpy array, with one row, 36 columns, and a “depth” of 4 from the test image in Figure 1. The 36 columns correspond to the contours found by the method — note that there are 36 contours rather than seven because `cv2.RETR_TREE` tells the routine to return internal contours as well as external ones. Column zero corresponds to the first contour, column one the second, and so on.

Each of the columns has a four-element array of integers, representing indices of other contours, according to the scheme

```
[next, previous, firstChild, parent]
```

The next index refers to the next contour in this contour’s hierarchy level, while the previous index refers to the previous contour in this contour’s hierarchy level. The `firstChild` index refers to the first contour that is contained inside this contour. The `parent` index refers to the contour containing this contour. In all cases, a value of `-1` indicates that there is no linked contour. Knowing this, replace the second part of the contours program with the following code, being careful to make sure its indentation matches that given below:

```
# Find internal contours too.
(_, contours, hierarchy) = cv2.findContours(binary, cv2.RETR_TREE,
                                             cv2.CHAIN_APPROX_SIMPLE)

# Count the number of dots on the dice faces. We do this by iterating over
# hierarchy[0], first to find the indices of the dice contours, then again
# to find the dot contours.
dice = [] # list of dice contours
dots = [] # list of dot contours
```

```

# Find the dice contours.
for (i, c) in enumerate (hierarchy[0]):
    if c[3] == -1:
        dice.append (i)

# Now find the dot contours, and output how many we find.
for (i, c) in enumerate (hierarchy[0]):
    if c[3] in dice:
        dots.append (i)
print ("Total number of dots:", len (dots))

# Draw the contours over the original image and display the result.
cv2.drawContours (img, contours, -1, (0, 0, 255), 5)
cv2.namedWindow (sys.argv[0], cv2.WINDOW_NORMAL)
ny, nx, nc = img.shape
cv2.resizeWindow (sys.argv[0], nx//2, ny//2)
cv2.imshow (sys.argv[0], img)
cv2.waitKey (0)

```

Does the program yield the correct count of dots?

Things become clearer if you colour the different contours in the hierarchy in different colours. Rather than outlining all the contours in red as here, make your program outline only contours at the top of the hierarchy in red. Outline in green the next outermost contours, which should be the dots on the faces. The innermost contours, representing some lost paint in one of the dots in the central die, should be drawn in blue.

6 Concluding remarks

Before putting this experiment to one side and doing something more interesting, you might like to spend a few minutes thinking about how well the software explored in this experiment will work on images other than the one you have used for testing. To do that, you might reflect on the properties of this image:

- it is monochrome rather than colour;
- the lighting is fairly flat, though not uniform across the field of view;
- there is very little noise in the image;
- the dots on the dice's faces are very dark;
- imperfections in the dots affect the contours that are found.

In fact, you might look for images that break each of these properties in turn and see how well the software handles it.