

PyImageSearch Gurus Course

[🏠 \(HTTPS://GURUS.PYIMAGESEARCH.COM\)](https://gurus.pyimagesearch.com/) >

1.11.3: Advanced contour properties

Topic Progress: (<https://gurus.pyimagesearch.com/topic/finding-and-drawing-contours/>)

(<https://gurus.pyimagesearch.com/topic/simple-contour-properties/>)

(<https://gurus.pyimagesearch.com/topic/advanced-contour-properties/>)

(<https://gurus.pyimagesearch.com/topic/contour-approximation/>)

(<https://gurus.pyimagesearch.com/topic/sorting-contours/>)

[← Back to Lesson \(https://gurus.pyimagesearch.com/lessons/contours/\)](https://gurus.pyimagesearch.com/lessons/contours/)

Feedback

In the previous section we learned about a few simple contour properties such as area, perimeter, and bounding boxes. These properties actually build upon each other and we can use them to define *more advanced* contour properties. The properties we are about to inspect are more *powerful* than our previous properties, as they allow us to discriminate between and recognize various shapes in images.

Unlike the previous section where we explored a small piece of code for each contour property, I'm instead going to start by providing definitions for our advanced contour properties: *aspect ratio*, *extent*, *convex hull*, and *solidity*.

From there, we are going to take these more advanced contour properties and build two Python scripts. The first script is going to be used to recognize the X's and O's on a tic-tac-toe board. And the second script we're going to build will be used to recognize the different kinds of Tetris blocks.

Like I said, it's amazing what contour properties can accomplish, so be sure to pay attention to this section.

Objectives:

We are going to build on our simple contour properties and expand them to more advanced contour properties, including:

1. Aspect ratio
2. Extent
3. Convex hull
4. Solidity

We'll then take these advanced contour properties and create a two computer vision applications that will:

1. Identify the X's and O's on a tic-tac-toe board.
2. Identify Tetris blocks.

Advanced Contour Properties

Contour properties can take you a long, *long* way when building computer vision applications, especially when you are first getting started. It takes a bit of creative thinking and a lot of discipline not to jump to more advanced techniques such as machine learning and training your own object classifier — but by paying attention to contour properties we can actually perform object identification for simple objects quite easily.

Let's go ahead and get started reviewing some of the more advanced contour properties, beginning with *aspect ratio*.

Aspect Ratio

The first “advanced” contour property we'll discuss is the aspect ratio. The aspect ratio is actually not that complicated at all, hence why I'm putting the term “advanced” in quotations. But despite its simplicity, it can be very powerful.

For example, I've personally used aspect ratio to distinguish between squares and rectangles and detect handwritten digits in images and prune them from the rest of the contours.

The actual definition of the a contour's *aspect ratio* is as follows:

aspect ratio = *image width* / *image height*

Yep. It's really that simple. The aspect ratio is simply the ratio of the image width to the image height.

Shapes with an *aspect ratio* < 1 have a height that is greater than the width — these shapes will appear to be more “tall” and elongated. For example, most digits and characters on a license plate have an aspect ratio that is less than 1 (since most characters on a license plate are taller than they are wide).

And shapes with an *aspect ratio* > 1 have a width that is greater than the height. The license plate itself is an example of a object that will have an aspect ratio greater than 1 since the width of a physical license plate is always greater than the height.

Finally, shapes with an *aspect ratio* $= 1$ (plus or minus some ϵ of course), have approximately the same width and height. Squares and circles are examples of shapes that will have an aspect ratio of approximately 1.

Extent

The *extent* of a shape or contour is the ratio of the contour area to the bounding box area:

$$\text{extent} = \text{shape area} / \text{bounding box area}$$

Recall that the area of an actual *shape* is simply the number of pixels inside the contoured region. On the other hand, the rectangular area of the contour is determined by its bounding box, therefore:

$$\text{bounding box area} = \text{bounding box width} \times \text{bounding box height}$$

In all cases the *extent* will be less than 1 — this is because the number of pixels inside the contour cannot possibly be larger the number of pixels in the bounding box of the shape.

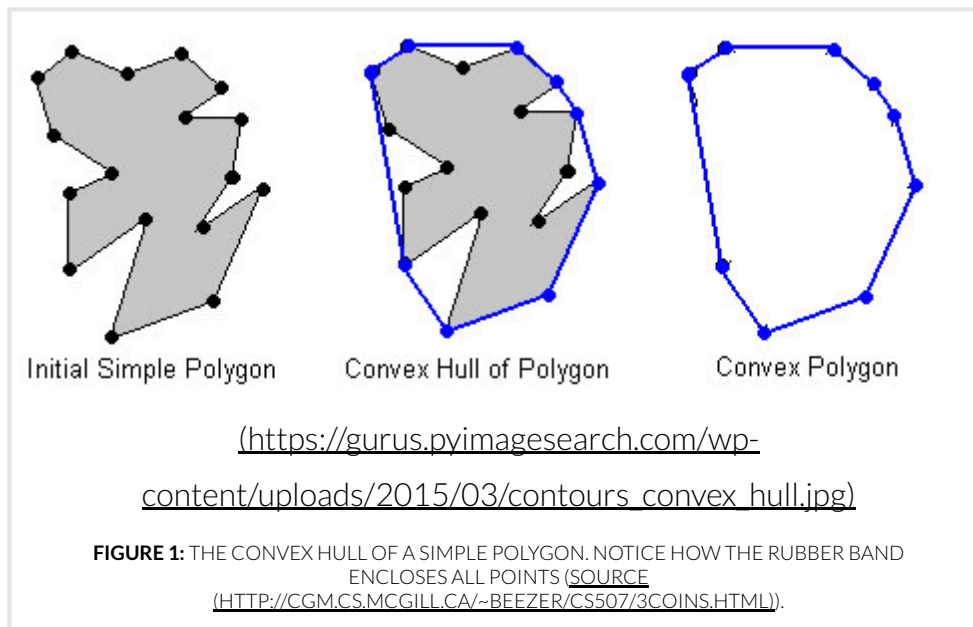
Whether or not you use the *extent* when trying to distinguish between various shapes in images is entirely dependent on your application. And furthermore, you’ll have to manually inspect the values of the extent to determine which ranges are good for distinguishing between shapes — and later in this section I’ll show you exactly how to perform this inspection.

Convex Hull

I like to think of a convex hull as a super elastic rubber band to bundle together a bunch of mail envelopes. I can place however many envelopes I want inside this rubber band, regardless of their size. And no matter what, this super elastic rubber band will surround these envelopes and hold them together. This super elastic rubber band never leaves any extra space or any extra slack — it requires the minimum amount of space to enclose all my envelopes.

A convex hull is almost like a mathematical rubber band. More formally, given a set of X points in the Euclidean space, the convex hull is the smallest possible convex set that contains these X points.

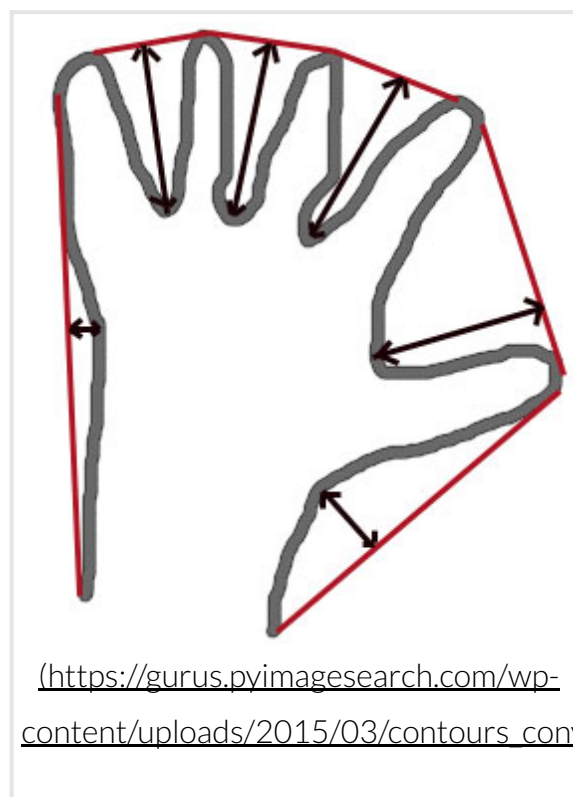
In the example image below, we can see the rubber band effect of the convex hull in action:



On the *left* we have our original shape. And in the *center* we have the convex hull of original shape. Notice how the rubber band has been stretched to around all extreme points of the shape, but leaving *no extra space* along the contour — thus the convex hull is the minimum enclosing polygon of all points of the input shape, which can be seen on the *right*.

Another important aspect of the convex hull that we should discuss is the *convexity*. Convex curves are curves that appear to “bulged out”. If a curve is not bulged out, then we call it a *convexity defect*.

For example, take a look at the following hand-drawn image of a hand taken from the OpenCV documentation:



The gray outline of the hand in the image above is our original shape. The red line is the convex hull of the hand. And the black arrows, such as in between the fingers, are where the convex hull is “bulged in” rather than “bulged out”. Whenever a region is “bulged in”, such as in the hand image above, we call them *convexity defects*.

Perhaps not surprisingly, the convex hull and convexity defects play a major role in hand gesture recognition, as it allows us to utilize the convexity defects of the hand to count the number of fingers. We’ll be exploring hand gesture recognition and convexity defects in **Module 13** of this course.

But hand gesture recognition is not the only thing the convex hull is good for. We also use it when computing another important contour property: *solidity*.

Solidity

The last advanced contour I want to discuss is the *solidity* of a shape. The solidity of a shape is the area of the contour area divided by the area of the convex hull:

$$\text{solidity} = \text{contour area} / \text{convex hull area}$$

Again, it’s not possible to have a *solidity* value greater than 1. The number of pixels inside a shape cannot possibly outnumber the number of pixels in the convex hull, because by definition, the convex hull is the smallest possible set of pixels enclosing the shape.

Just as in the extent of a shape, when using the solidity to distinguish between various objects you’ll need to manually inspect the values of the solidity to determine the appropriate ranges. For example (and as we’ll see in the next sub-section), the solidity of a shape is actually perfect for distinguishing between the X’s and O’s on a tic-tac-toe board.

Getting Our Hands Dirty

So at this point we’ve reviewed some advanced contour properties such as *aspect ratio*, *extent*, *convex hull*, and *solidity*.

That’s great — but the real question is: *How do we put these contour properties to work for us?*

In the upcoming two sections we’ll be utilizing our contour properties to distinguish between X’s and O’s on a tic-tac-toe board and how to recognize different Tetris blocks. **Both** of these example problems are situations in which I’ve seen other computer vision programmers and developers

immediately jump into using image descriptors and machine learning to solve these problems — that is simply overkill and is absolutely not necessary.

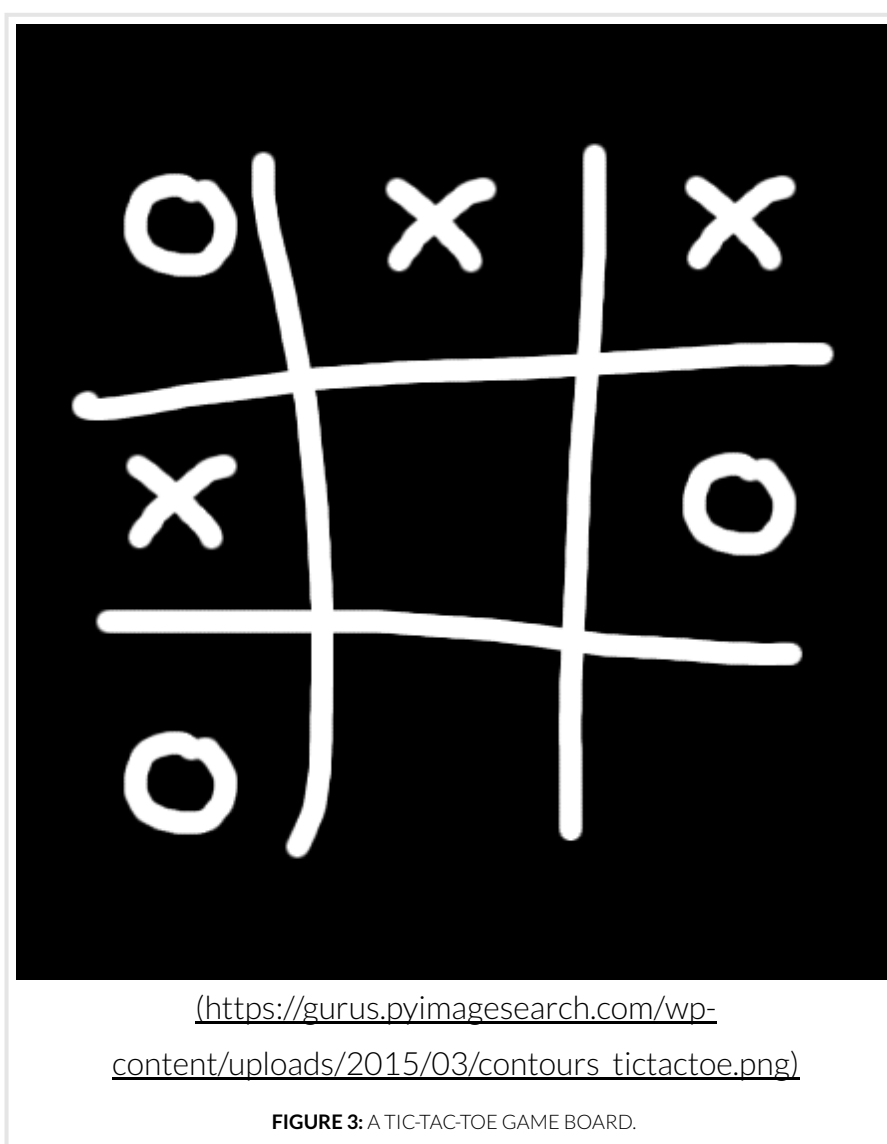
If anything, I want the key takeaway from this section to be: ***always consider contours before more advanced computer vision and machine learning methods.***

Often (and as you'll see below), a clever use of contour properties can enable us solve problems that may appear challenging on the surface, but are actually quite easy.

Distinguishing Between X's and O's

Let's get started by recognizing the X's and O's on a tic-tac-toe board.

If you are unfamiliar with the game, the playing board looks like this:



Tic-tac-toe is a two player game. One player is the “X” and the other player is the “O.” Players alternate turns placing their respective X’s and O’s on the board, with the goal of getting three of their symbols in a row, either horizontally, vertically, or diagonally. It’s very simple game to play, common among young children who are first learning about competitive games.

Interestingly, tic-tac-toe is a *solvable game*. When played optimally, you are guaranteed at best to win, at and at worst to draw (i.e. tie).

While we aren't going to dive into the game mechanics of optimal tic-tac-toe play, we are going to write a Python script that leverages computer vision and contour properties to recognize the X's and O's on the board. Using this script, you could then take the output and feed it into a tic-tac-toe solver to give you the optimal set of steps to play the game.

Anyway, enough talk. Let's look at some code:

tictactoe.py	Python
<pre>1 # import the necessary packages 2 import cv2 3 import imutils 4 5 # load the tic-tac-toe image and convert it to grayscale 6 image = cv2.imread("images/tictactoe.png") 7 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) 8 9 # find all contours on the tic-tac-toe board 10 cnts = cv2.findContours(gray.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) 11 cnts = cnts[0] if imutils.is_cv2() else cnts[1] 12 13 # loop over the contours 14 for (i, c) in enumerate(cnts): 15 # compute the area of the contour along with the bounding box 16 # to compute the aspect ratio 17 area = cv2.contourArea(c) 18 (x, y, w, h) = cv2.boundingRect(c) 19 20 # compute the convex hull of the contour, then use the area of the 21 # original contour and the area of the convex hull to compute the 22 # solidity 23 hull = cv2.convexHull(c) 24 hullArea = cv2.contourArea(hull) 25 solidity = area / float(hullArea)</pre>	

Feedback

We'll start off with our standard procedure of importing our necessary packages. Then we'll load our tic-tac-toe game board (**Figure 3**) from disk and convert it to grayscale.

Line 10 handles finding the actual contours in the image. Notice how we are using the `cv2.RETR_EXTERNAL` flag in the `cv2.findContours` function to indicate that we want the *external-most* contours only.

Had we not provided this flag, we would have ended up detecting the inner circle of the "O" character as well as the outer circle! And if you don't believe me, I would take a second a review the **Finding and Drawing Contours** section where we detail how to find contours in an image. Remember when we detected the contour of the inner-ovular region of the rectangle? The same premise applies here — if we were to detect *all* contours, we would end up finding the inner circle of the "O;" hence we are only interested in the external contours.

From there we start looping over each individual contour on **Line 14**.

And here is where we start computing the actual properties of our contour. We start off by computing the **area** of the contour on **Line 17**. Again, remember that the `cv2.contourArea` is *not* giving us the **area = width × height** area of the contour. Instead, it's giving us the number of pixels that reside *inside* the contour (we'll be computing the rectangular area of the contour later).

Line 18 then grabs the **bounding box** of the contour, which gives us the starting (x, y)-coordinates of the contour, followed by the width and height of the bounding box.

Given these two simple contour properties, let's compute two advanced properties: the **convex hull** and the **solidity**. The actual convex hull of the shape is computed on **Line 23** and the area of the convex hull is then computed on **Line 24**. Now that we have both the **area of the contour** along with the **area of the convex hull** we can compute the **solidity**, which we defined above.

So how are we going to put these properties to work for us? Let's take a look:

tictactoe.py	Python
<pre>27 # initialize the character text 28 char = "?" 29 30 # if the solidity is high, then we are examining an 'O' 31 if solidity > 0.9: 32 char = "O" 33 34 # otherwise, if the solidity it still reasonable high, we 35 # are examining an 'X' 36 elif solidity > 0.5: 37 char = "X" 38 39 # if the character is not unknown, draw it 40 if char != "?": 41 cv2.drawContours(image, [c], -1, (0, 255, 0), 3) 42 cv2.putText(image, char, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 1.25, 43 (0, 255, 0), 4) 44 45 # show the contour properties 46 print("{} (Contour #{}) -- solidity={:.2f}".format(char, i + 1, solidity)) 47 48 # show the output image 49 cv2.imshow("Output", image) 50 cv2.waitKey(0)</pre>	

Feedback

We'll start by initializing our `char` variable to indicate the character that we are looking at — in this case, we initialize it to be a ? indicating that the character is unknown.

Lines 31 and 32 check to see if the character is an O — we hardcode a rule that if the *solidity* > 0.9 then, then character should be marked as an O.

And similarly, if the *solidity* > 0.5 (**Lines 36 and 37**) then we are examining an X.

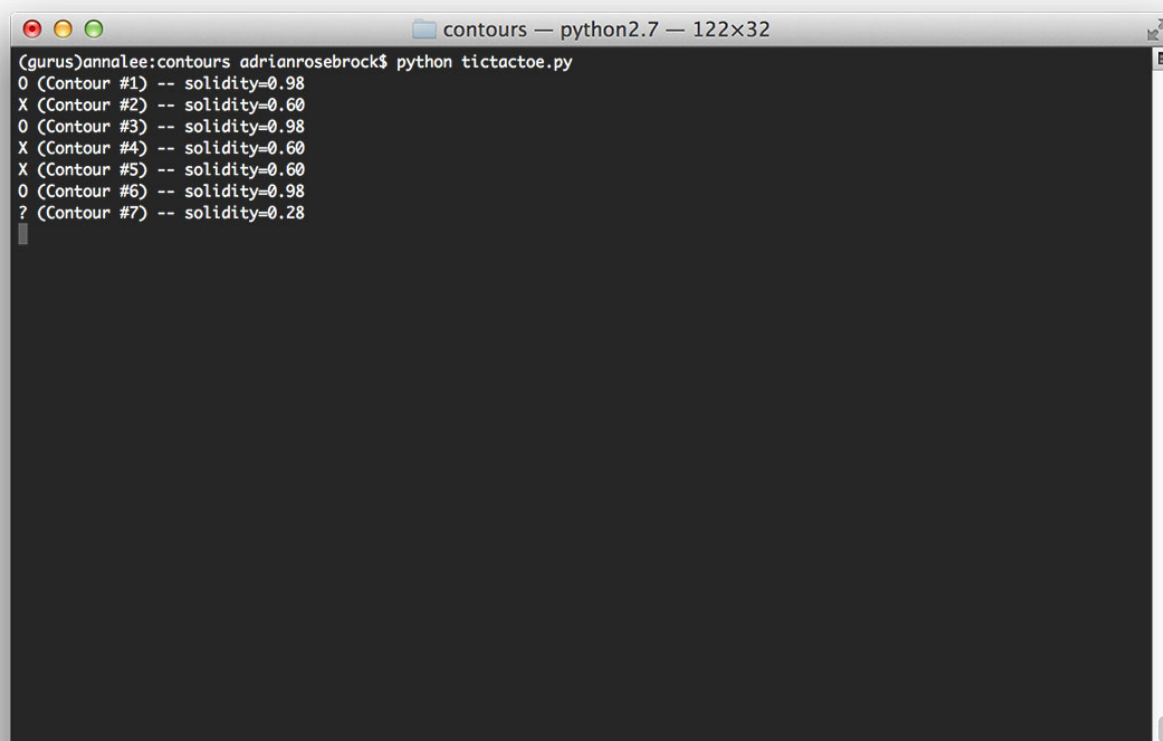
Finally, if we are able to identify the character, we then draw the contour and the character on our image on **Lines 40-43**.

Intuitively, this makes sense. The letter X has four large and obvious convexity defects — one for each of the four V's that form the X. On the other hand, the O has nearly no convexity defects, and the ones that it has are substantially less dramatic than the letter X. Therefore, the letter O is going to have a larger solidity than the letter X.

So how did I arrive that the solidity values of 0.9 and 0.5?

It wasn't magic — I simply examined the solidity of each character on **Line 46**, which prints the solidity value and current contour ID to the screen.

Looking at the output of my terminal you can clearly see the ranges of valid values to distinguish between X's and O's:



```
(gurus)annalee:contours adrianrosebrock$ python tictactoe.py
O (Contour #1) -- solidity=0.98
X (Contour #2) -- solidity=0.60
O (Contour #3) -- solidity=0.98
X (Contour #4) -- solidity=0.60
X (Contour #5) -- solidity=0.60
O (Contour #6) -- solidity=0.98
? (Contour #7) -- solidity=0.28
```

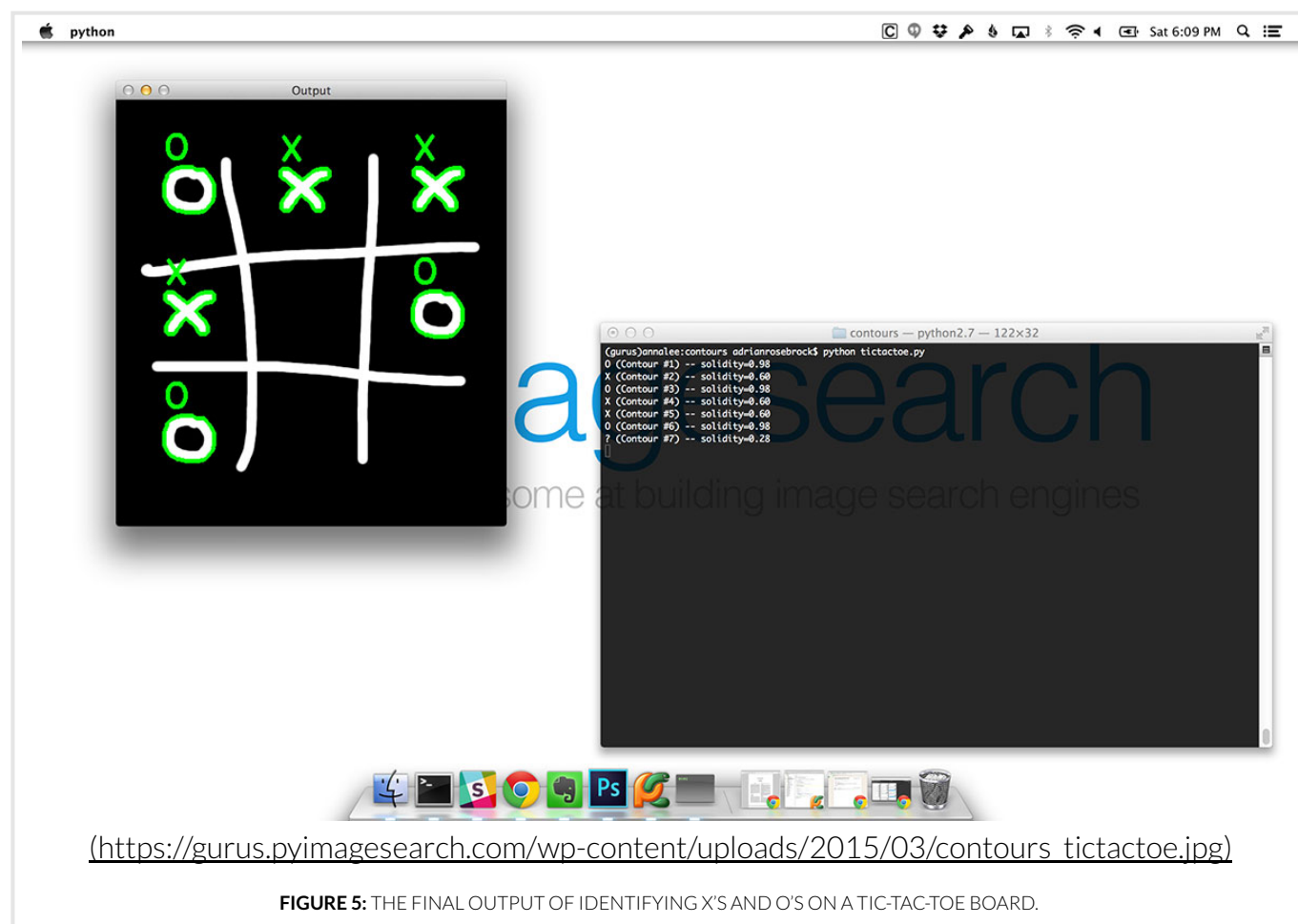
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_tictactoe_output.jpg)

FIGURE 4: EXAMINING THE SOLIDITY VALUE TO DEFINE VALID RANGES FOR X'S AND O'S.

When we look at this output, we can see that the letter O has a solidity of 0.98 for all three times it appears on the board. Similarly, the letter X has a solidity of 0.6. The last solidity value of 0.28 refers to the actual game board itself.

Like I said, by inspecting these solidity values it becomes trivial to define the solidity value ranges to distinguish between X's and O's on **Lines 31-37**.

The actual output on a game board is thus:



In this figure it's substantially more clear that our algorithm is functioning as expected. We are able to identify all of the X's and O's without a problem while totally ignoring the actual game board itself. And we accomplished all of this by examining the **solidity** of each contour. No fancy computer vision algorithms. No fancy machine learning. Just leveraging contour properties to our advantage.

Identifying Tetris Blocks

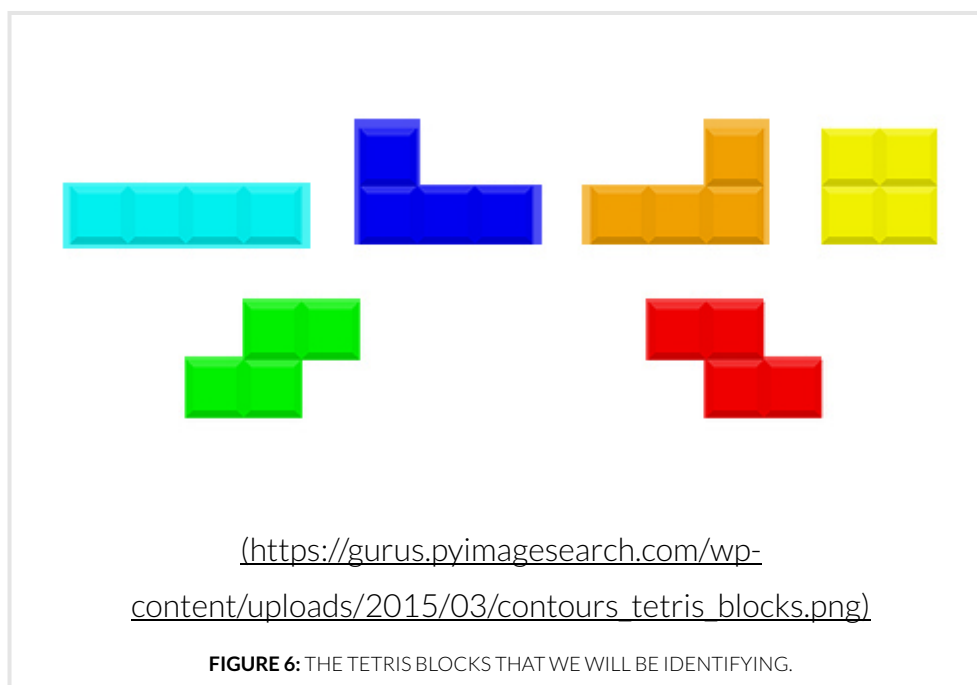
Distinguishing between X's and O's in a tic-tac-toe game is a great introduction to the power of contour properties. But in the previous example we used only the *solidity* perform our identification. In some cases, such as in identifying the various types of Tetris blocks, we need to utilize more than one contour property. Specifically, we'll be using *aspect ratio*, *extent*, *convex hull*, and *solidity* in conjunction with each other to perform our brick identification.

Open up a new file, name it `contour_properties_2.py` , and let's start coding:

```
contour_properties_2.py Python
1 # import the necessary packages
2 import numpy as np
3 import cv2
4 import imutils
5
6 # load the Tetris block image, convert it to grayscale, and threshold
7 # the image
8 image = cv2.imread("images/tetris_blocks.png")
9 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
10 thresh = cv2.threshold(gray, 225, 255, cv2.THRESH_BINARY_INV)[1]
11
12 # show the original and thresholded images
13 cv2.imshow("Original", image)
14 cv2.imshow("Thresh", thresh)
15
16 # find external contours in the thresholded image and allocate memory
17 # for the convex hull image
18 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
19 cnts = cnts[0] if imutils.is_cv2() else cnts[1]
20 hullImage = np.zeros(gray.shape[:2], dtype="uint8")
```

Again, we start with our standard practice of importing the necessary packages and loading our Tetris block image off disk.

Here is what our Tetris image looks like:



The aqua piece is known as a *Rectangle*. The blue and orange blocks are called *L-pieces*. The yellow shape is obviously a *Square*. And the green and red bricks on the bottom are called *Z-pieces*.

Our goal here is to extract contours from each of these shapes and then identify which shape each of the blocks are.

To accomplish this, we'll need to create a *binary* image so we can extract the contours of the image. We apply thresholding on **Line 10**, which we covered in **Module 1.8** (<https://gurus.pyimagesearch.com/lessons/thresholding/>) to create a binary image, where the background pixels are *black* and the foreground pixels (i.e. the Tetris blocks) are *white*.

We then find the contours in our thresholded image on **Line 18** and allocate a NumPy array with the same shape as our input image on **Line 20** so we can visualize the convex hull of each shape. Visualizing the convex hull is **not** a required step in identifying the various Tetris blocks; however, I thought it would be interesting to include in this example so you could see what the contour of a convex hull looked like.

Let's continue with our example:

contour_properties_2.py	Python
<pre>22 # loop over the contours 23 for (i, c) in enumerate(cnts): 24 # compute the area of the contour along with the bounding box 25 # to compute the aspect ratio 26 area = cv2.contourArea(c) 27 (x, y, w, h) = cv2.boundingRect(c) 28 29 # compute the aspect ratio of the contour, which is simply the width 30 # divided by the height of the bounding box 31 aspectRatio = w / float(h) 32 33 # use the area of the contour and the bounding box area to compute 34 # the extent 35 extent = area / float(w * h) 36 37 # compute the convex hull of the contour, then use the area of the 38 # original contour and the area of the convex hull to compute the 39 # solidity 40 hull = cv2.convexHull(c) 41 hullArea = cv2.contourArea(hull) 42 solidity = area / float(hullArea) 43 44 # visualize the original contours and the convex hull and initialize 45 # the name of the shape 46 cv2.drawContours(hullImage, [hull], -1, 255, -1) 47 cv2.drawContours(image, [c], -1, (240, 0, 159), 3) 48 shape = ""</pre>	

Feedback

On **Line 23** we start looping over each of the contours individually so we can compute our contour properties.

We'll start off by computing the **area** and the **bounding box** of the contour on **Lines 26 and 27**.

We then compute the **aspect ratio**, which is simply the ratio of the width to the height of the **bounding box** on **Line 31**. Again, remember that the aspect ratio of a shape will be < 1 if the height is greater than the width. The aspect ratio will be > 1 if the width is larger than the height. And the aspect ratio

will be *approximately 1* if the width and height are equal.

With this in mind, do you have any guesses as to what we'll use the **aspect ratio** for?

If you guessed discriminating between the square and rectangle pieces, you would be correct. But more on that later.

Line 35 then computes the **extent** of the current contour, which is the area (i.e. number of pixels that reside within the contour) divided by the true rectangular (*area = width x height*) area of the bounding box.

Finally, **Lines 40-42** compute the **convex hull** and the **solidity** in the same manner as we saw in the tic-tac-toe example.

We then visualize the convex hull, draw the current contour, and initialize the name of our shape on **Lines 46-48**.

Now that we have computed all of our contour properties, let's define the actual rules and

`if` statements that will allow us to discriminate between the various Tetris blocks:

contour_properties_2.py	Python
50	# if the aspect ratio is approximately one, then the shape is a square
51	<code>if</code> aspectRatio >= 0.98 <code>and</code> aspectRatio <= 1.02:
52	shape = "SQUARE"
53	
54	# if the width is 3x longer than the height, then we have a rectangle
55	<code>elif</code> aspectRatio >= 3.0:
56	shape = "RECTANGLE"
57	
58	# if the extent is sufficiently small, then we have a L-piece
59	<code>elif</code> extent < 0.65:
60	shape = "L-PIECE"
61	
62	# if the solidity is sufficiently large enough, then we have a Z-piece
63	<code>elif</code> solidity > 0.80:
64	shape = "Z-PIECE"
65	
66	# draw the shape name on the image
67	cv2.putText(image, shape, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
68	(240, 0, 159), 2)
69	
70	# show the contour properties
71	print("Contour #{ } -- aspect_ratio={:.2f}, extent={:.2f}, solidity={:.2f}"
72	.format(i + 1, aspectRatio, extent, solidity))
73	
74	# show the output images
75	cv2.imshow("Convex Hull", hullImage)
76	cv2.imshow("Image", image)
77	cv2.waitKey(0)

As I hinted at above, if the **aspect ratio** of a shape is *approximately* 1, then the width and height of the shape are roughly equal. And if this is the case, we know that in context of our Tetris blocks, if the width and height are equal, then we must be examining a square (**Lines 51 and 52**).

Similarly, if the **aspect ratio** is large, then the width is much greater than the height. And in the context of identifying Tetris pieces, this must mean we are examining a horizontal rectangle (**Lines 55 and 56**).

However, if the rectangle was vertical rather than horizontal, the **aspect ratio** would then be very small since the height would be substantially larger than the width. I did not account for this in the code above and I'll leave it as an exercise for you to convince yourself of this point (it's good to have a little homework).

Finally, **Lines 59 and 60** examine the **extent** to see if we are looking at an L-piece, and **Lines 63 and 64** check to see if we are examining a Z-piece. Again, just like in the tic-tac-toe board, I **manually investigated** the values of the aspect ratio, extent, and solidity to define the valid ranges to distinguish between the Tetris block pieces — and if you were developing your own shape identification script, you would have to perform the same type of investigation. This will become very obvious once we look at the output of our Python script.

We then draw the name of the shape on our image on **Lines 67 and 68**, print our contour properties to our terminal on **Lines 71 and 72**, and display our output images to screen on **Lines 75-77**.

To execute this script, simply navigate to our source code directory and execute the following command:

contour_properties_2.py	Python
1 \$ python contour_properties_2.py	

Once the script executes, you'll see the first image popup on your screen:



On the *top* we have our original input image. Below that we have our **thresholded** image (**Line 10**) where the actual Tetris blocks we want to identify appear as white on a black background. And then on the *bottom* we have our convex hull image for the Z-piece — notice how this shape almost looks like a skewed hexagon. This is because the convex hull is accounting for the two convexity defects (i.e. where the Z-piece is “bulged in”).

Then, looking at the output of the terminal we see that the **solidity** is ≥ 0.8 — according to our rule, this must be a Z-piece.

Let’s look at the next shape:

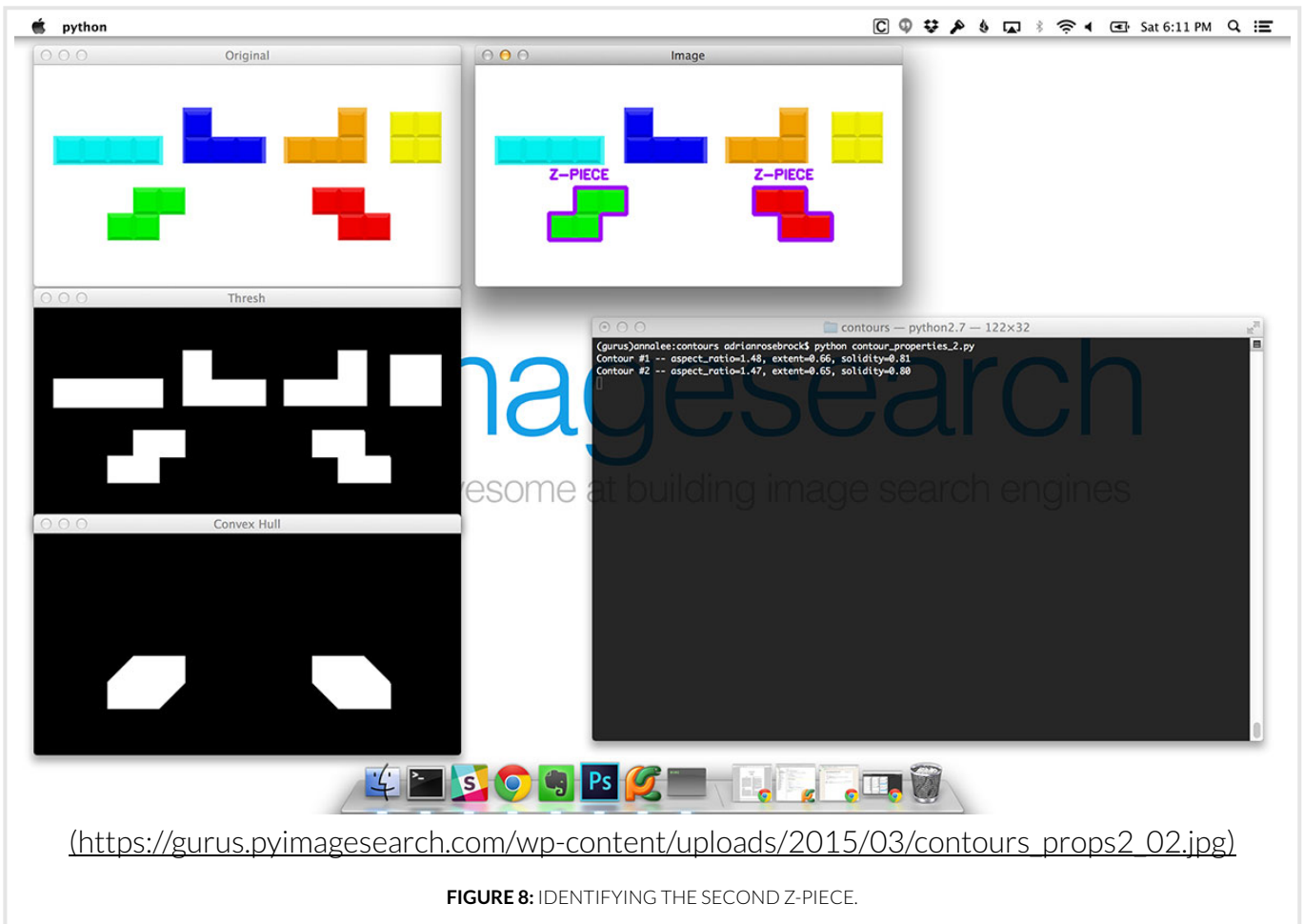


FIGURE 8: IDENTIFYING THE SECOND Z-PIECE.

Again, we are applying the same thought process as in the previous figure. We can see that the **solidity** of the shape is also ≥ 0.8 , thus we are once again examining a Z-piece.

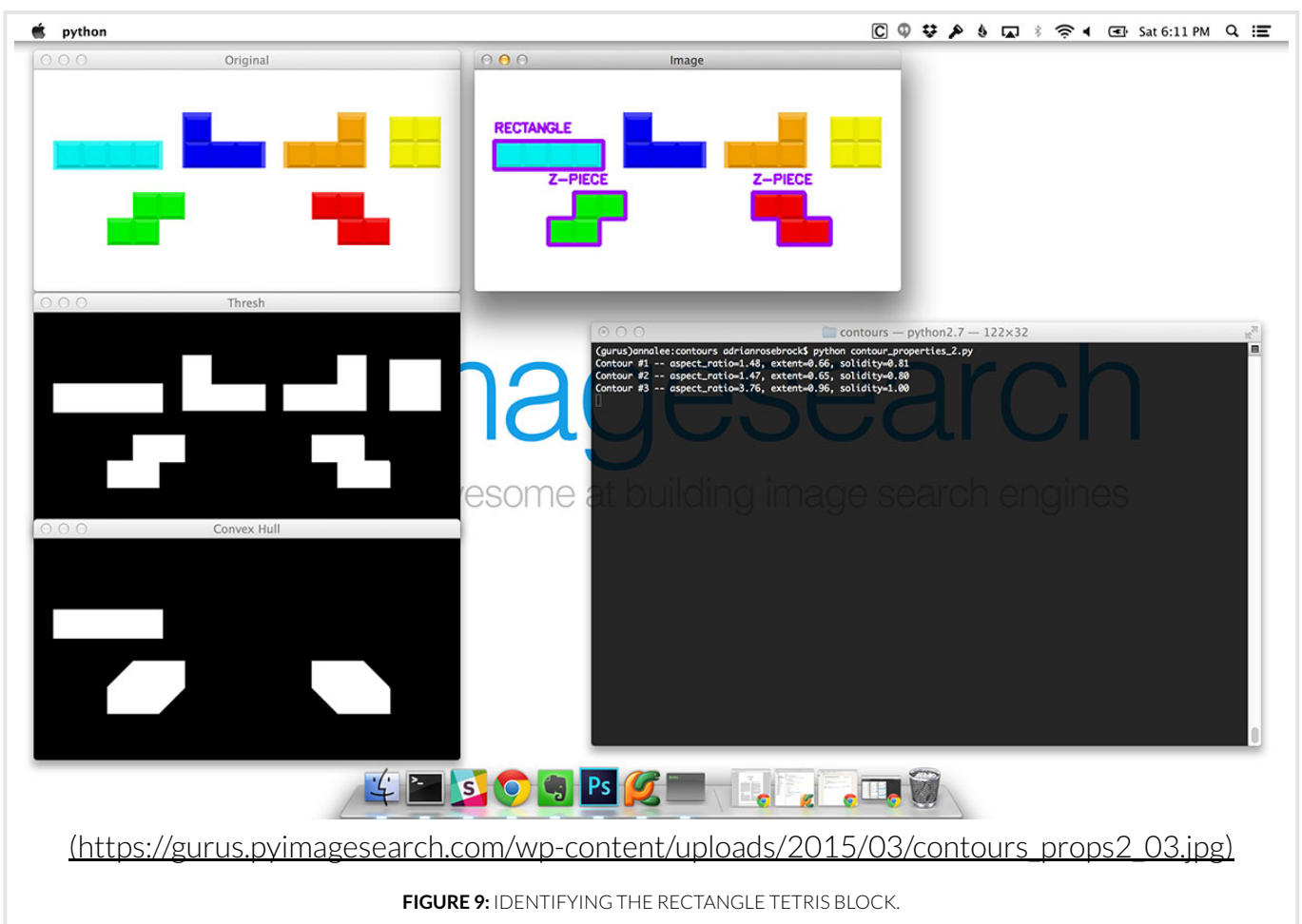
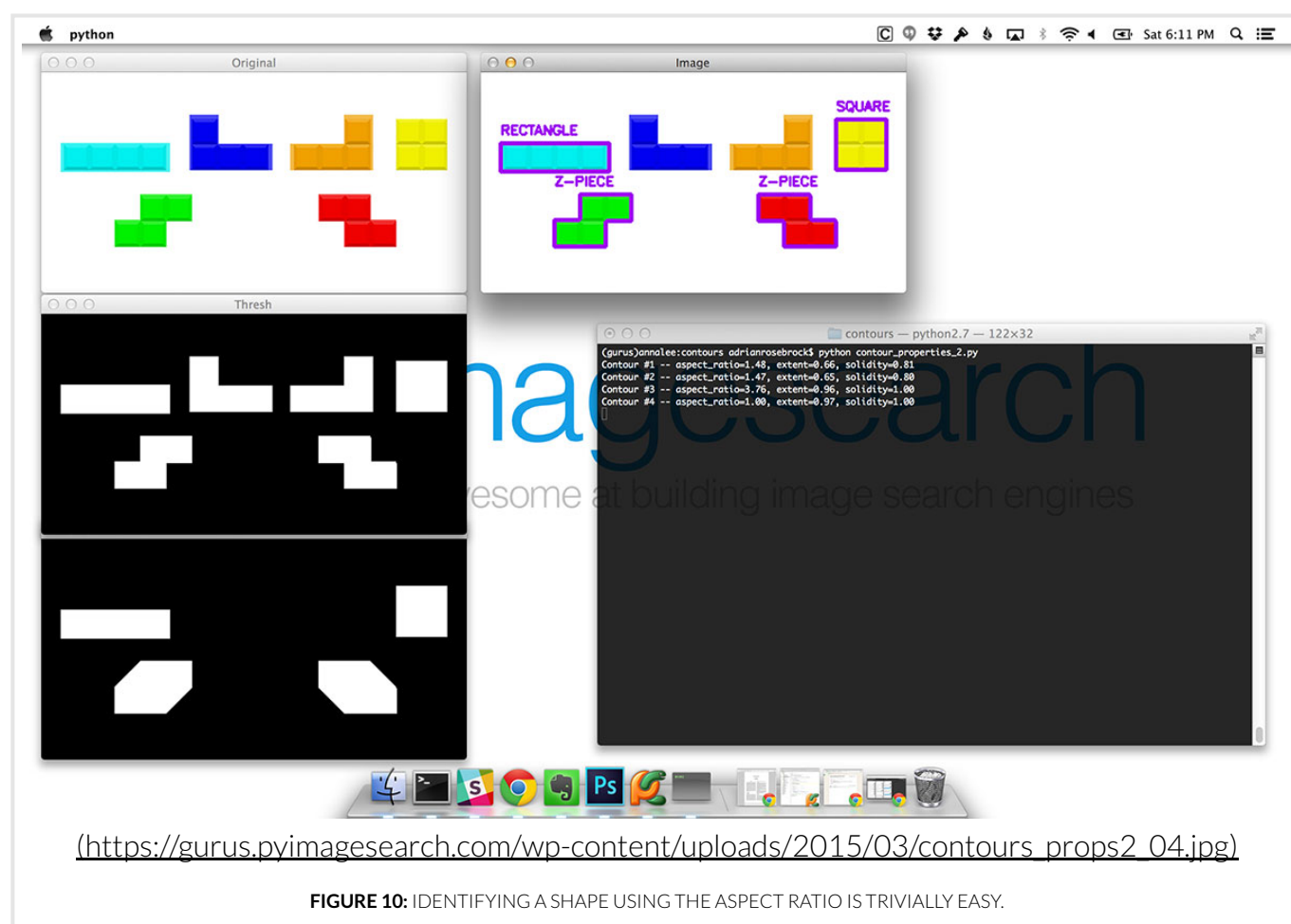


FIGURE 9: IDENTIFYING THE RECTANGLE TETRIS BLOCK.

Identifying the rectangle Tetris block is very easy — since the block is so elongated, the width is substantially larger than the height. This means that the **aspect ratio** must be quite large. In this case the $aspect\ ratio = 3.76$, allowing our rule on **Line 55** to correctly identify the shape as a rectangle.



Just like identifying the rectangle in **Figure 9** above was easy, recognizing the square is even easier. Since a square has (approximately) the same width and height, we know that if the $aspect\ ratio = 1$, (plus or minus some ϵ , of course) the shape must be a square.

Also take a look at the visualization of the convex hull for both the rectangle and the square. Since both the rectangle and square shapes have no convexity defects, the convex hull is actually no different than the original shape.

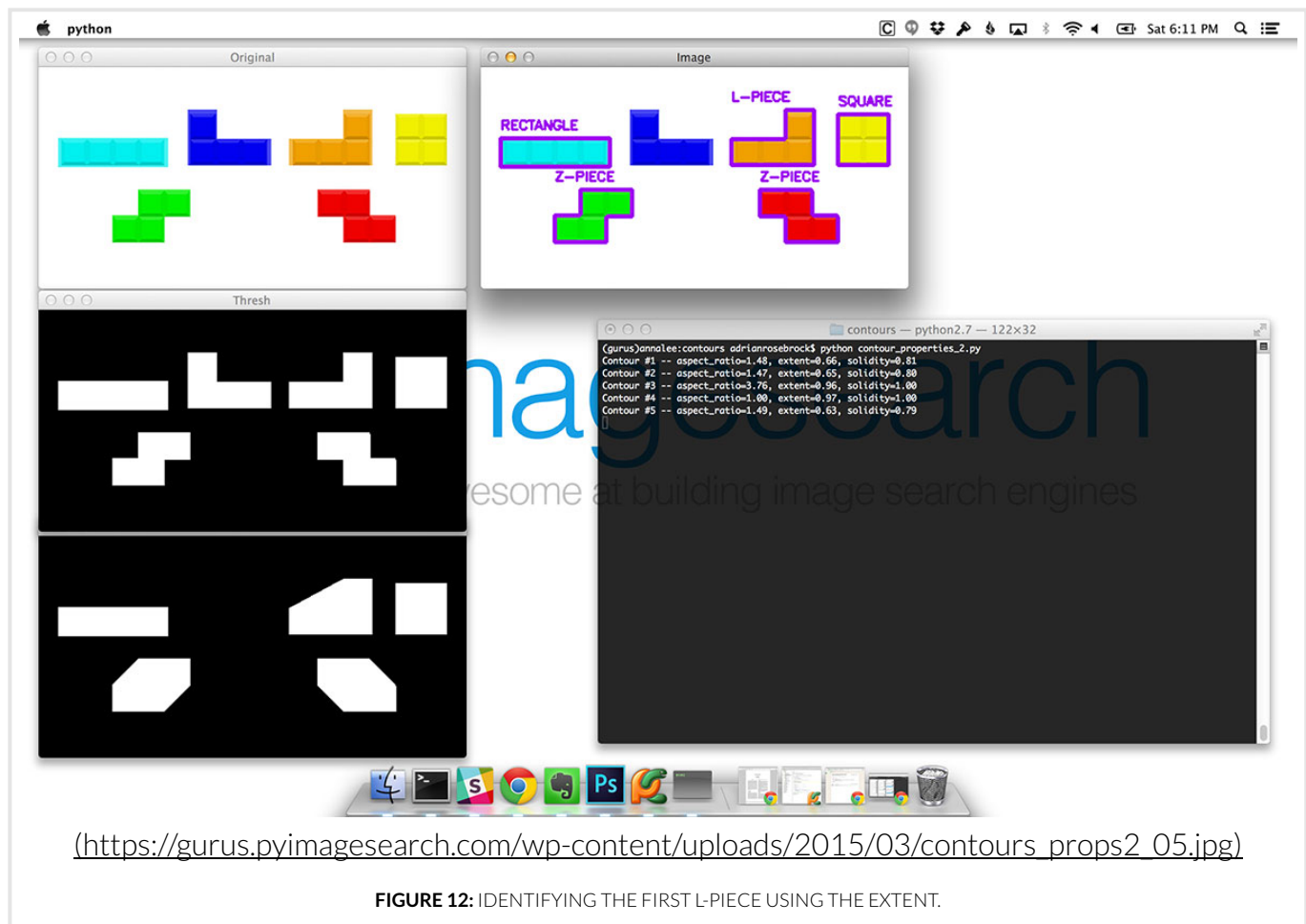
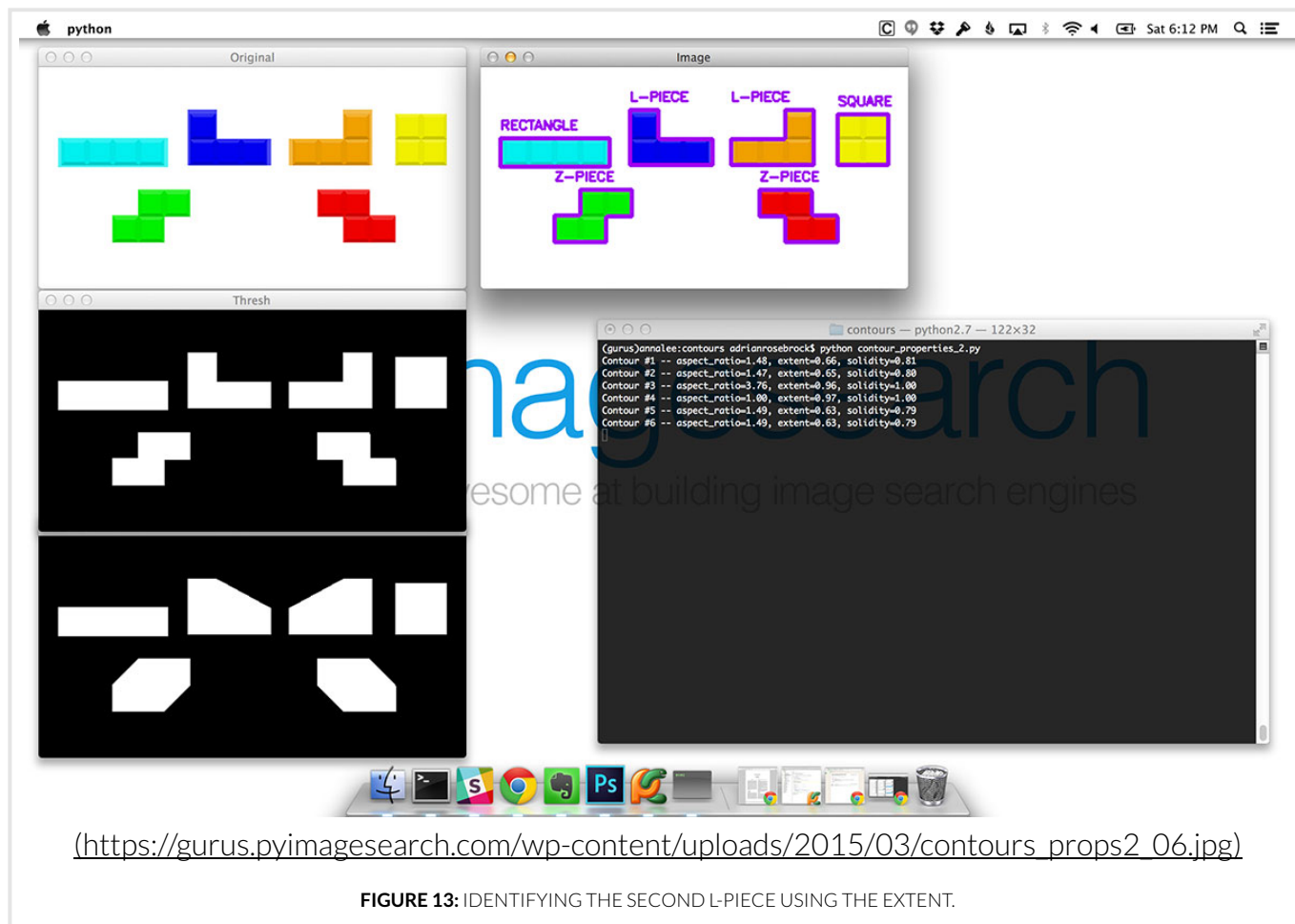


FIGURE 12: IDENTIFYING THE FIRST L-PIECE USING THE EXTENT.

Take a look at the output of our terminal. Up until Contour #5, has the **extent** of a shape dipped below 0.65? According to our output, it has not. Thus, the **extent** is a good property to distinguish the L-piece.

The same is true for the second L-piece below:



As you can see, using nothing more than the *aspect ratio*, *extent*, and *solidity* of a shape we were able to distinguish between the four different types of Tetris blocks.

Of course, there is more than one way to skin this cat. We could have also used methods to quantify and represent the shapes, like **Hu moments** (<https://gurus.pyimagesearch.com/lessons/hu-moments/>) or **Zernike moments** (<https://gurus.pyimagesearch.com/lessons/zernike-moments/>), which we'll cover later in this course.

But at the same time, why bother?

Using simple contour properties, we were able to recognize X's and O's on a tic-tac-toe board. And we were also able to recognize the various types of Tetris blocks. Again, these contour properties, which are very simple on the surface, can enable us to identify various shapes — we just need to take a step back, be a little clever, and inspect the values of each of our contour properties to construct rules to identify each shape.

Summary

In this lesson we built upon our previous simple contour properties and learned about more advanced contour properties such as *aspect ratio*, *extent*, *convex hull*, and *solidity*.

Using these contour properties then enabled us to distinguish between X's and O's on a tic-tac-toe board and recognize various Tetris block pieces. Again, these shape identifications were accomplished using *only* contour properties. We did not have to apply any type of advanced techniques such as machine learning or training our own object classifier to obtain our shape classifications.

Downloads:

[Download the Code](https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/advanced-contour-properties-code/)
(https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/advanced-contour-properties-code/)

Quizzes	Status
1	Advanced Contour Properties Quiz (https://gurus.pyimagesearch.com/quizzes/advanced-contour-properties-quiz/)

[← Previous Topic \(https://gurus.pyimagesearch.com/topic/simple-contour-properties/\)](https://gurus.pyimagesearch.com/topic/simple-contour-properties/) [Next Topic → \(https://gurus.pyimagesearch.com/topic/contour-approximation/\)](https://gurus.pyimagesearch.com/topic/contour-approximation/)

Feedback

Upgrade Your Membership

Upgrade to the *Instant Access Membership* to get **immediate access** to **every lesson** inside the PyImageSearch Gurus course for a one-time, upfront payment:

- **100%, entirely self-paced**
- Finish the course in *less than 6 months*
- Focus on the lessons that *interest you the most*
- **Access the entire course** as soon as you upgrade

This upgrade offer will expire in **30 days, 18 hours**, so don't miss out — be sure to upgrade now.

Upgrade Your Membership!
(<https://gurus.pyimagesearch.com/register/pyimagesearch-gurus-instant-access-membership-fcff7b5e/>)

Course Progress

Ready to continue the course?

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

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

Resources & Links

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

Your Account

- [Account Info \(https://gurus.pyimagesearch.com/account/\)](https://gurus.pyimagesearch.com/account/)
- [Support \(https://gurus.pyimagesearch.com/contact/\)](https://gurus.pyimagesearch.com/contact/)
- [Logout \(https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&wnonce=5736b21cae\)](https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&wnonce=5736b21cae)

🔍 Search

