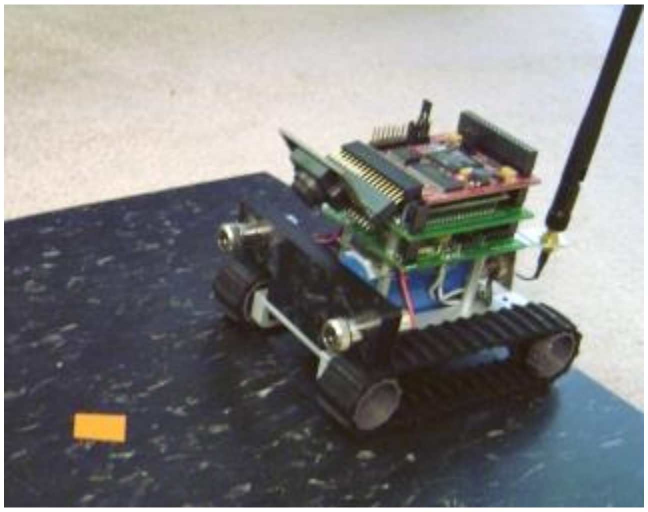
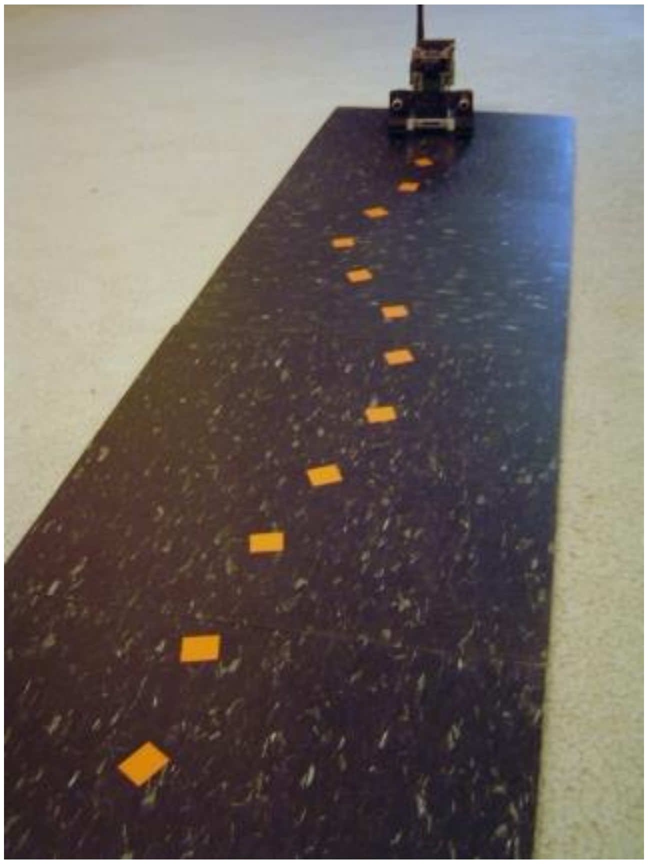
I took this tutorial from a tutorial that was using pre-packaged image processing software to complete the task. In other words the user just had to pick certain image processing techniques and the software did it for them......magic box type operation. I took the modules they selected and then added in what was actually happening in code at each operation. Hopefully it makes sense. They used a robot with a wireless camera and the software is on a PC that is communicating with the camera. So they have a wireless delay, and a all the computations to do before they find the next dot. The software is for sale for about $90.....it's not open source, you can't use it for this class. But after the semester is over you could design an interface yourself and sell the techniques you learn for $89 and under cut them.

The camera angle of the SRV-1b is more or less parallel with the floor. In order for this tutorial to work correctly the camera needs to be tilted down such that more of the floor is in view. This hardware adjustment is necessary otherwise the squares appear more like orange lines due to the large perspective distortion in the default view. You will have to mount and tilt your camera in an intelligent way.



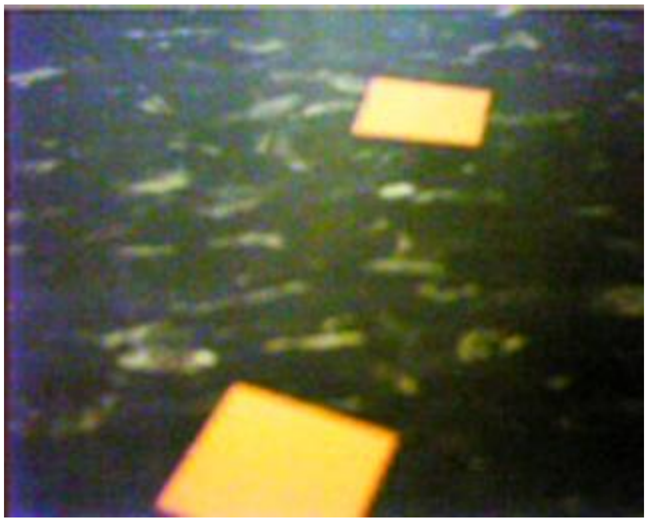
First left us have a look at our sample trail.

Sample Trail



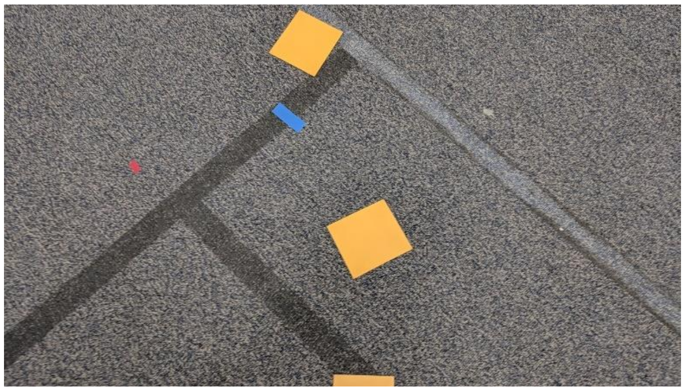
Sample Trail

The image above is an overhead shot of the sample trail creating using orange electrical tape placed on black tiles. From an overhead view the contrast is very good with the squares being nicely defined against the matt black tiles.



Robot View

Here is the same kind of picture but this is our carpet and our paper trail.



You can see the similar background noise as in the example scene.

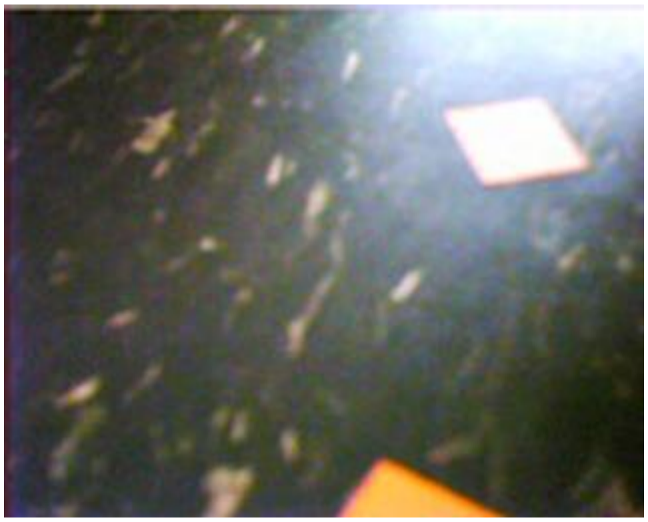
From the robot's point of view the setup looks quite different than from an overhead view. This is largely due to the proximity of the camera to the ground and the lack of adequate lighting to provide enough contrast in the image. The bad lighting of the setup was done on purpose in order to review image processing techniques that can help in these kinds of environments.

The robot view also reveals the pattern in the black tiles that includes white marks. These are less apparent in the overhead view but provide a high level of noise in the robot view.

You may also note that the squares in the image do not appear orange but instead appear more of a yellow color. This is due to color consistency problems which cause colors to appear incorrectly due to camera intrinsics, bad/low lighting and different illumination colors, i.e. colors appear different in sunlight versus florescent lights even if well lit.

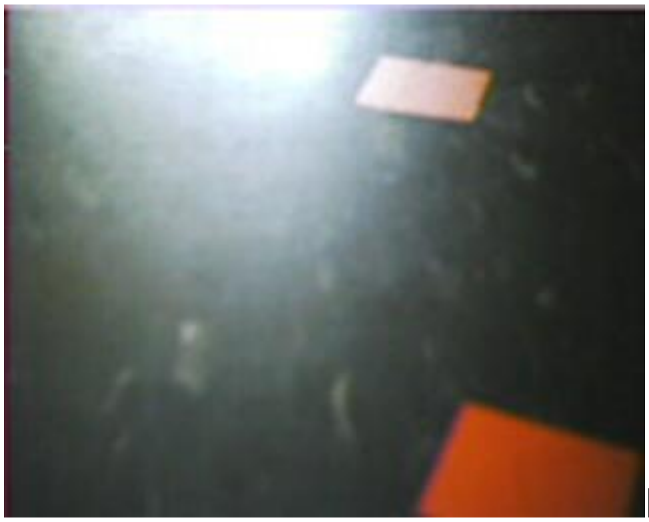
Let's have a look at a couple more images from the robot..

Robot View



It turns out that the previous robot image view was in fact one of the better images! Things just get worse from that! The above image shows a capture from the robot's point of view on its way back over the course. Our setup is placed not far from an outside window. With the strong sunlight the glare from the outside causes significant reflection on the black tiles. Despite being black the tiles appear white when in the appropriate reflective angle with respect to the outside light. Again, this bad lighting was done on purpose in order to review possible solutions to this environmental difficulty when you cannot change the environment. If you can change the environment then is always recommended that you do so, i.e. close the curtains!

You can also see from the above image that the "orange" square also suffers the same affliction as the black tiles. The upper square is actually orange but in fact appears completely white in this view due to the reflected light.



Robot View #3

The final robot image view is also taken from the reflected sunlight angle but in this case the lower square finally reveals its true orange color. Yet in the very same image the upper square again appears white.

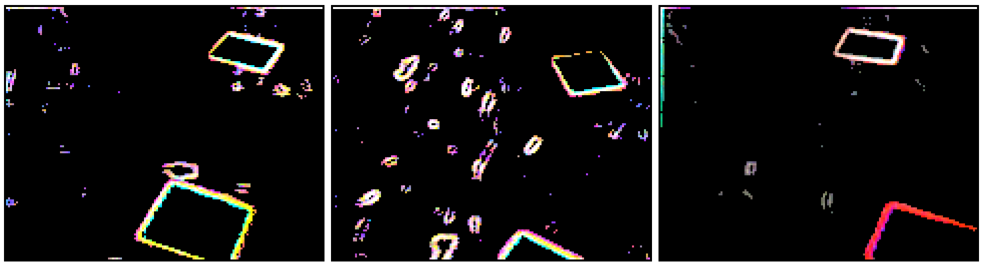
If you are new to image processing and machine vision you will by now start to gain an awareness of the importance of lighting in these scenarios. Lighting is a fundamental issue when working with robotic vision.

Given these three test images let's begin processing them in order to segment the squares from the rest of the image.

Lighting

We begin the orange square segmentation problem by working on the lighting issue. From the latter two robot images it is clear that the glare needs to be reduced.

A quick technique to resolve lighting issues is to reduce the image to edges. Most edge detection techniques use the local neighborhood of pixels in order to generate the edge strength. This local neighborhood has the advantage of reducing global lighting issues. This technique was used in the other tutorial I posted on line following. The technique requires edge detection and thresholding followed by detection of the Center of Gravity to determine the robot direction.



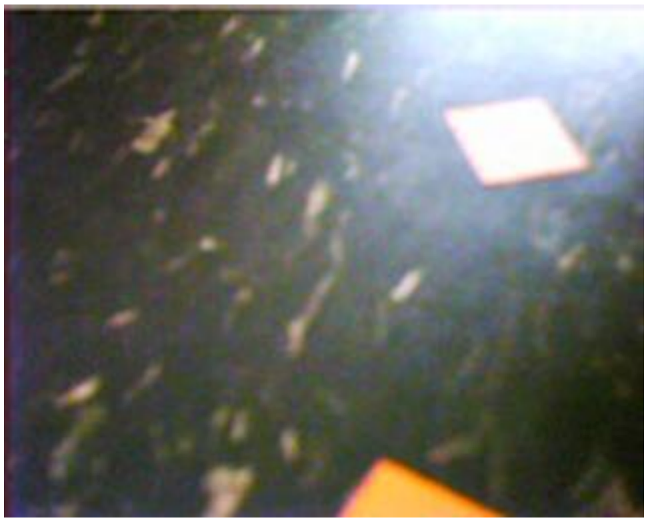
We can clearly see that while this technique does have some promise the edges detected also include edges from the white spots embedded in the black tiles. If you are using a surface that does not have these noise elements then edge detection is a possible way to go.

In the case of our trail following robot there is a problem at the end of the course when the robot turns around. During the turn proceedure the robot will momentarily see the tile edge against a lighter carpet. This boundary appears as a very strong edge which will cause the Center of Gravity measure to veer the robot off course.

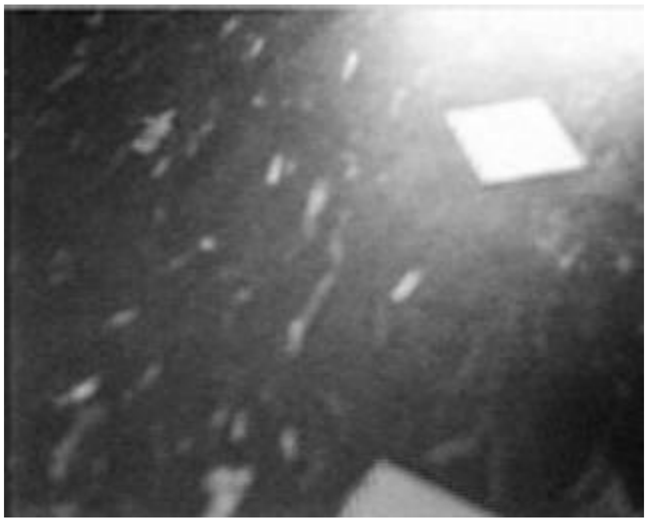
Instead, we will try another light adjusting technique.

Lighting #2

We now proceed with a common lighting technique that will level the intensity of all pixels within an image. We will use one of the images to illustrate the process.


Original

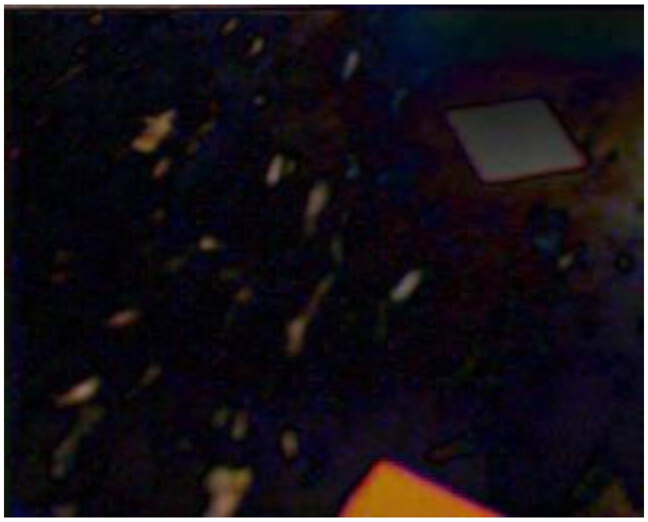
First, we convert the image to grayscale using the Grayscale module. This essentially focuses the image into its luminance or lighting channel. This is the channel that we want to even out within the image.


Grayscale

Next we really REALLY blur the grayscale image using OPENCV blur process.

Very Blurred

From this blurred image we subtract the original color image using the absolute difference.

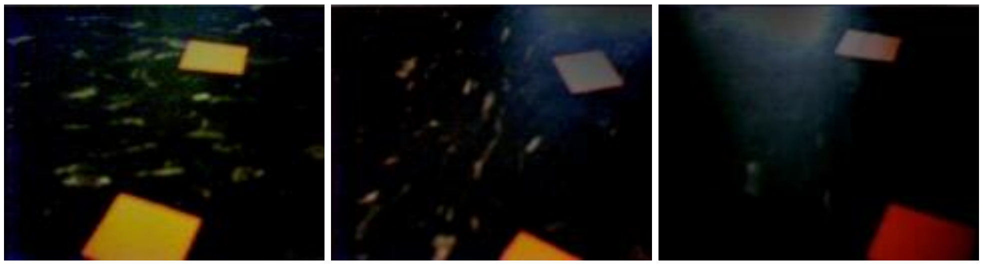

Grayscale

Which results in a much more even intensity across the entire image. This technique works because the grayscale conversion focuses on the intensity channel (use the intensity form from the first assignment) , the blurring relaxes the edges caused by lighting to effect large areas and the subtraction removes the global lighting changes to leave just the localized intensity changes which are typically associated with edges. This is in effect a form of an edge detector but one that better preserves the image colors than most edge detection techniques.

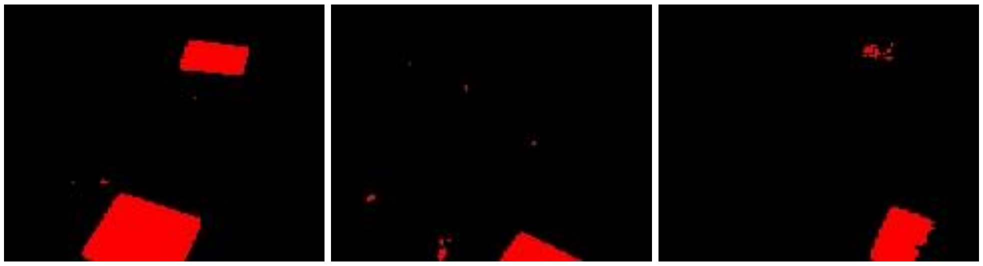
Now let's try detecting colors.

Colors

In order to detect the squares we need to identify them from within the image. Lets try using colors to detect them. Let us review what the three test images currently look like now with more even lighting.



Using (the OPENCV function to track colors that you used back in project 2) to search for red (yellow and orange both contain red) reveals some interesting but incorrect results.

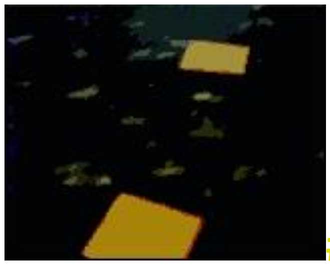
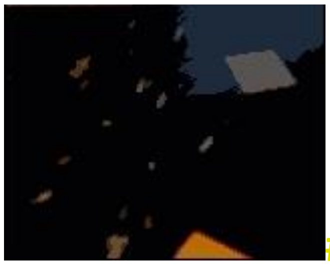
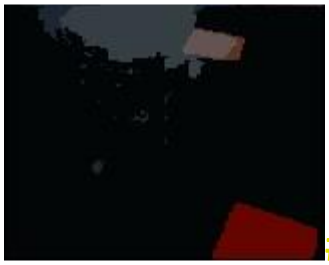


The first image seems fine but the last two mostly miss the upper square. If this happens during the course the robot will start turning around prematurely as it will think that the course has come to an end.

As in apparent from the above images using color to detect a gray object is not going to be very fruitful. In this case we now abandon color as an identification feature and use it more as a natural pixel grouping feature.

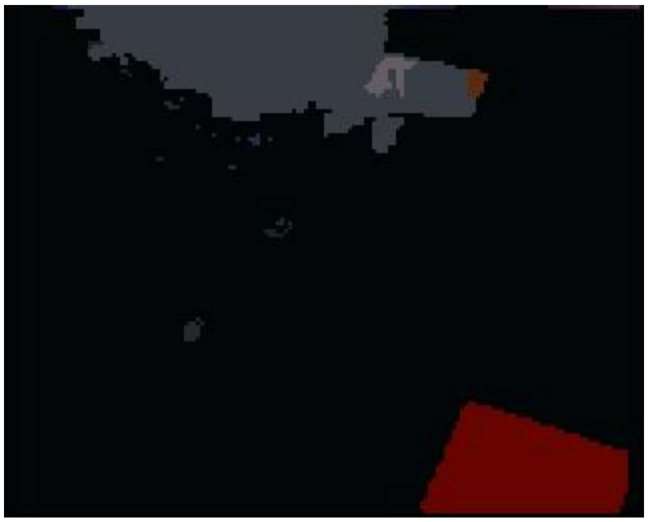
Flood Fill

As we can no longer rely on color as a tracking feature we now turn to shape. To extract objects based on shape we first need to ensure that the object can be analyzed as an object. This means we need to group similar colored pixels with each other to define an object or a blob as known in machine vision terms.

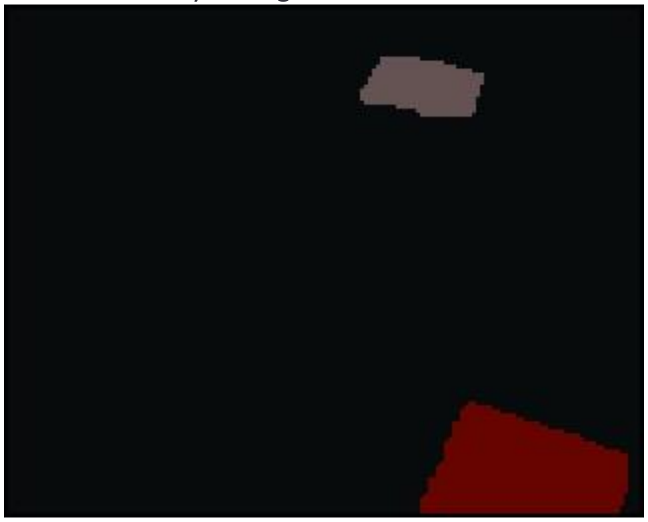

#1

#2

#3

Using the a flood fill technique we apply a color flattening technique to merge pixels into more meaningful groups. Flood filling is similar to the flood fill that is present in most paint programs. If a pixel is of similar color to its immediate neighbor the two pixels are replaced with the mean color of the two.

This works well to define objects but we still have a problem with image #3. If you look closely the upper square is almost a single color but has a large part of it on the right hand side in another color. This is caused by the tolerance of the flood fill not being high enough to include that part of the blob as a single object. However, increasing the tolerance causes other undesirable effects such as merging the square into the background glare.



Instead we use another feature of the Flood Fill to analyze neighboring blobs and how intense their borders are. The original edge detection tests showed that the outline of the squares were the only dominant edge in each square. The border between the parts of the upper square blob in image #3 above is then not very significant. Thus we use the Merge Borders interface in the flood fill to help merge blobs whose borders are not very strong.

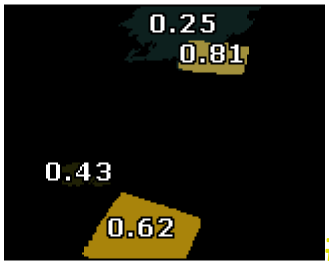


This also has the added benefit of merging the glare objects into a single background blob further decreasing the image noise. Now that we have all of our squares a single color we can start analyzing the shapes of the remaining blobs.

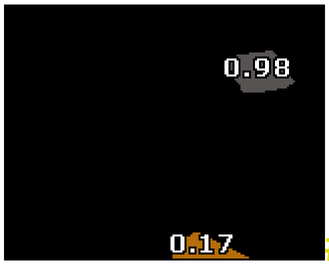
Blob Filter

To extract only the square shapes from the image we utilize the blob filter and the Quadrilateral Area to quantify the resemblence of the blob to a square. The Quadrilateral Area attribute analyses the sides and area of each of the blobs to calculate how close the blob area is to a quadrilateral defined by the 4 major sides of the blob. I.e. how well does the blob fit into the concept of squareness.

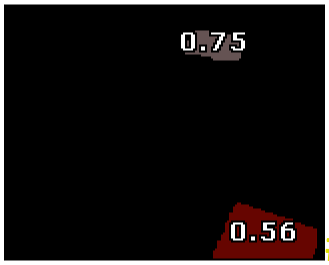
The blob filter allows you to add in descriptive attributes that rank each of the individual blobs according to the feature they represent. Thus, for a perfect square we would expect a rank of 1.0 whereas for blobs that are less square we would expect a <1.0 weight.



#1



#2

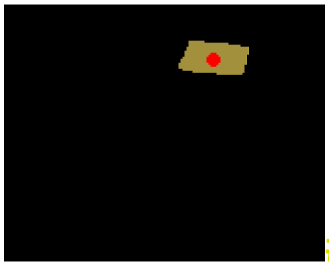


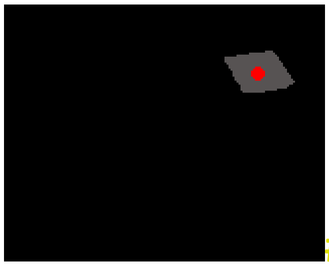
#3

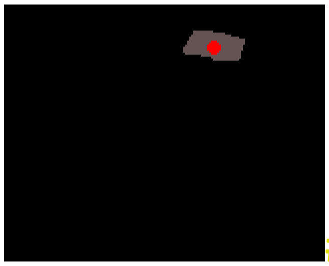
Reviewing our test images shows that a cutoff value of around 0.65 would segment all our actual squares from the rest of the detected blobs. The other detected blobs are artifacts from the tile's white spots and the sunlight glare that we reduced earlier on.

If is worth noting that the squares on the sides of the images may or may not be detected by the square shape as part of the square is cutoff from the image. This is not an issue as we also use the blob filter to remove any blobs that touch the border. The reasoning is that we know all squares will fit into the image view and anything on the sides will either come into view or if just going out of view. We also remove border objects as the robot when moving may cause previously seen blobs to once again be detected. This detection can cause the robot to oscillate back and forth as the object moves in and out of partial view of the camera.

Using a 0.65 threshold and adding the COG module ends our image processing process as we now have an approximate point (the COG) for the robot to follow. The red circle identifies the COG point.


#1


#2


#3

Note that if more than one square is visible on the screen the COG will return a average point between the two squares which is desirable as it evens out the transition from one square to the next.

Now that we have the point to move towards we need to use that information to control the TangoBot appropriately.