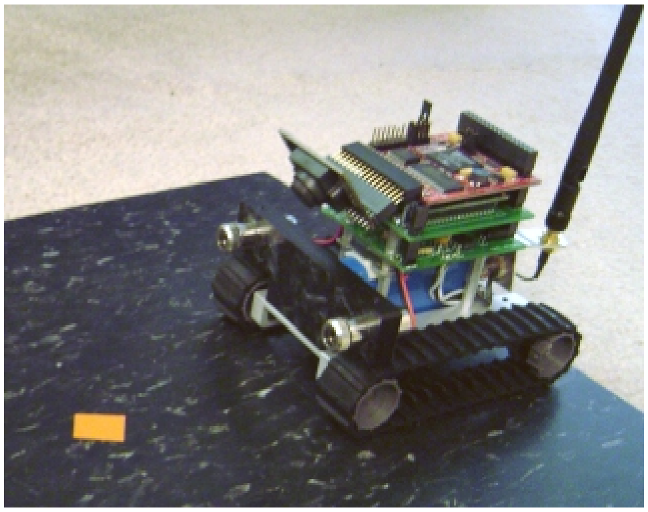
Robot Following a trail

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.

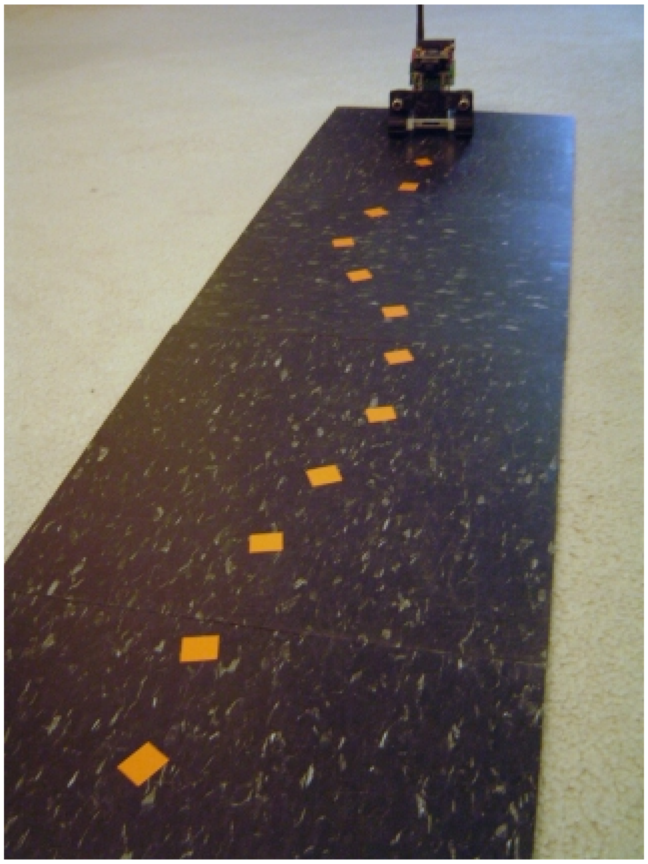
They used a robot with a wireless camera and the software is on a PC that is communicating with the camera. So they have a wireless delay, and a all the computations to do before they find the next dot. The software is for sale for about $90…..it's not open source, you can't use it for this class. But after the semester is over you could design an interface yourself and sell the techniques you learn for $89 and under cut them.

In this tutorial we once again experiment with object tracking. In this case we are looking for orange squares that define a trail for the robot the follow. The trial of squares is created using orange electrical tape cut in small pieces and placed approximately 1.5 inches apart. We will just use orange paper cut up in the classroom. Care needs to be taken to place each square near enough to the previous square otherwise the robot will not be able to see the next square as it loses track of the previous one. Once the robot determines that no additional squares are present it will turn around and proceed back over the course.

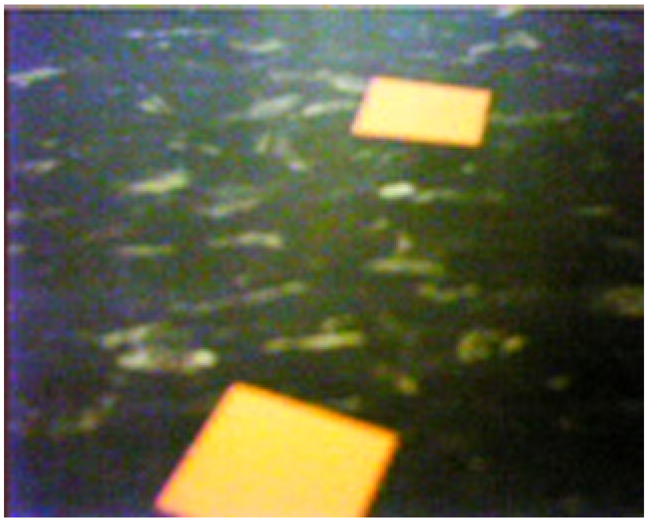
The camera angle is more or less parallel with the floor. In order for this tutorial to work correctly the camera needs to be tilted down such that more of the floor is in view. This hardware adjustment is necessary otherwise the squares appear more like orange lines due to the large perspective distortion in the default view.



First left us have a look at their 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.



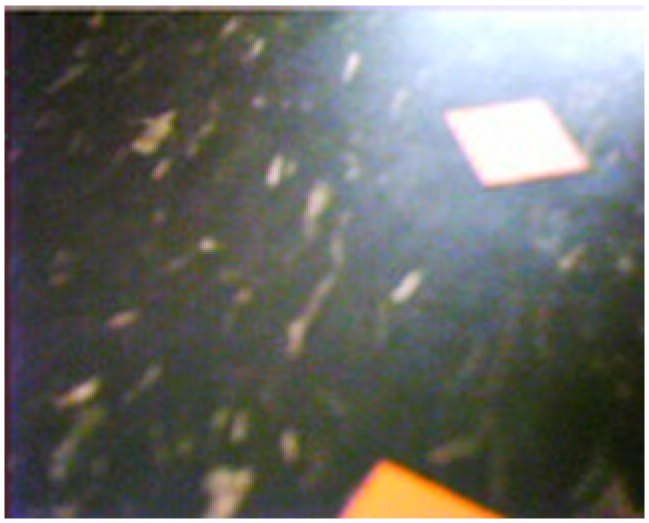
From the robot's point of view the setup looks quite different than from an overhead view. This is largely due to the proximity of the camera to the ground and the lack of adequate lighting to provide enough contrast in the image. The bad lighting of the setup was done on purpose in order to review image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

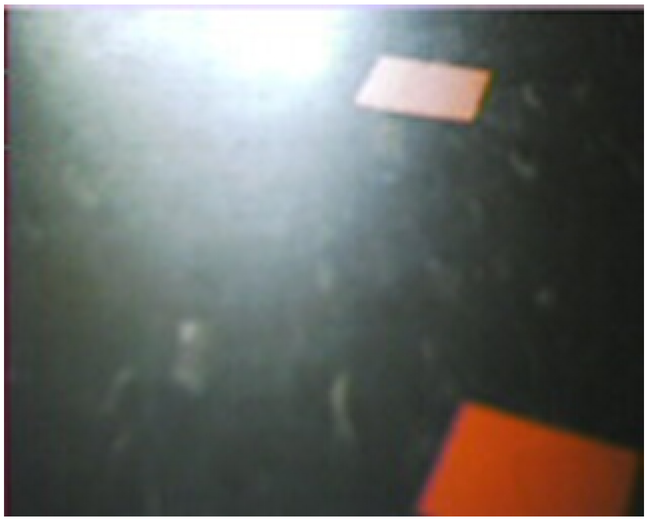
Let's have a look at a couple more images from the robot ....

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 (they're in sunny Los Angeles, CA) 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.



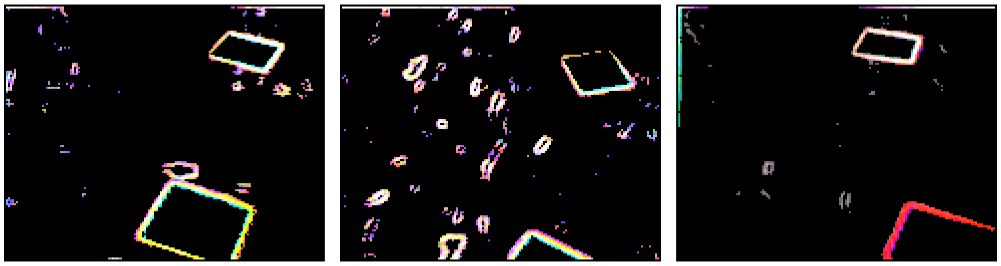
The final robot image view is also taken from the reflected sunlight angle but in this case the lower square finally reveals its true orange color. Yet in the very same image the upper square again appears white.

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 ...

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 line following lecture I just did, and you'll be doing this week for your homework. The technique requires edge detection and thresholding followed by detection of the Center of Gravity to determine the robot direction. I'm sure you're all finished by now……..



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 this trail following robot there is a problem at the end of the course when the robot turns around. During the turn procedure 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/they 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.

First, we convert the image to grayscale using the **Grayscale** formula that you feel works best. 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

The Grayscale module converts a color RGB image to grayscale values using the following formula techniques. Note that R,G,B represent the current pixels colors values in RGB color space.

Method 1

pixel = (R+G+B)/3

Method 2

pixel = 0.299R + 0.587G + 0.114B

Method 3

pixel = maximum(R,G,B)

the 'pixel' value is then assigned to the Red, Green and Blue channels to create the final image.

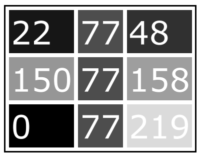
Note that each technique will change the way colors are converted to grayscale which will effect the relative intensities of each color. For example, method #1 makes Red and Blue seem like the same grayscale color whereas method #2 does not.

Next we really REALLY blur the grayscale image using the **Mean**.

Mean Filter

The Mean or Average filter is used to soften an image by averaging surrounding pixel values.

For example, given the grayscale 3x3 pixel window;



22	77	48
150	77	158
0	77	219

Center pixel = (22+77+48+150+77+158+0+77+219)/9

The center pixel would be changed from 77 to 92 as that is the mean value of all surrounding pixels.

This filter is often used to smooth images prior to processing. It can be used to reduce pixel flicker due to overhead fluorescent lights.

From this blurred image we subtract the original color image using a **Math** technique.

Image Math

The Image Math technique provides you with a way to perform arithmetic operations on a combination of two images.

Select the appropriate Image #1 and Image #2 on which to perform the function.

Source - the original image that was initially loaded or captured.

Current - the currently processed image.

Last - the last image processed.

.

Then you select the appropriate function used to combine the two images.

Add	`Image #1 + Image #2`
Subtract	`Image #1 - Image #2`
Multiply	`Image #1 * Image #2`
And	`Image #1 & Image #2 (binary AND)`
Or	`Image #1 | Image #2 (binary OR)`
XOr	`Image #1 ^ Image #2 (binary XOR)`
Maximum	`if Image #1 > Image #2 then result = Image #1 else result = Image #2`
Minimum	`if Image #1 < Image #2 then result = Image #1 else result = Image #2`
Difference	`| Image #1 - Image #2 | (Absolute value of subtraction)`
Divide	`Image #1 / Image #2`
Replace #1	`If Image #2 then Image #1 else 0`
Replace #2	`If Image #2 then Image #1 else 255`

You also must select the appropriate channel to perform the function on.

All channels - Performs the operation on each RGB channel
Red, Green, Blue - Performs the operation only on the selected channel. The result is placed in the selected Image #2 channel.

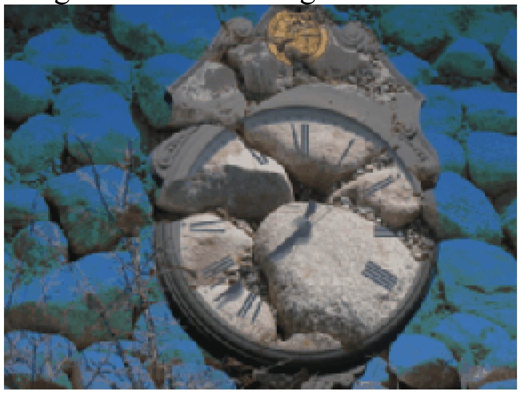
And then sometimes you need to select the appropriate divisor and bias.

Divisor - divides the function's result by the specified amount.
Bias - added to the function's result.

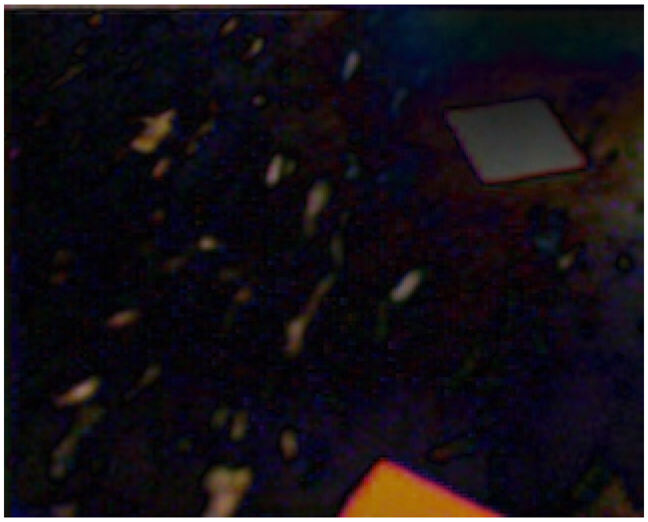
Example

Source Image #1 Source Image #2

Image #1 added to Image #2



The results of the blur with and a subtraction of the original image:



Which results in a much more even intensity across the entire image. This technique works because the grayscale conversion focuses on the intensity channel, 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.

RGB Filter

The RGB Filter uses RGB values to focus the attention towards the primary RGB colors. Depending on the color selected this filter will diminish all pixels that are not of the selected colors. This function is different than RGB Channel in that white pixels are also diminished even though they may contain the color selected.

For example, if Red is chosen:

R = ((R-B)+(R-G))

G = 0

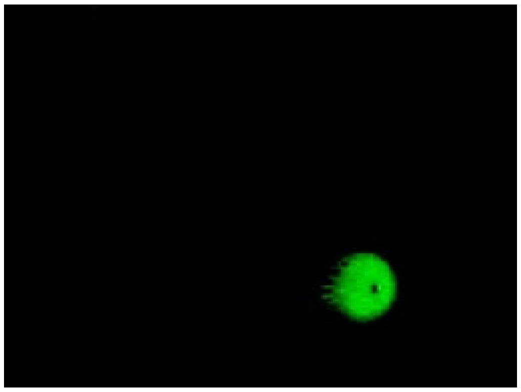
B = 0

R is then normalized with respect to the maximum red value.

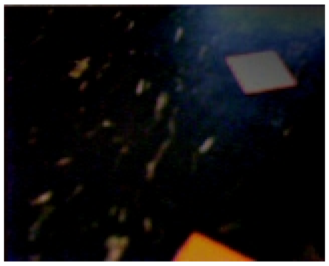
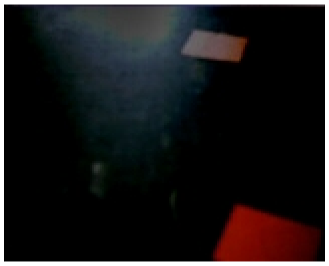
Based on the above formula it can be seen that white pixels result in a zero value whereas pure primary colors (R=255, G=0, B=0) R doubles its value. Thus function does a better job than RGB Channel in filtering for a particular color as white light is removed.

Due to normalization really dark pixels can be elevated in intensity and generate too much noise in the resulting image. The Min Pixel Value allows you to specify a minimum value below which pixels are considered to be black and will be ignored when calculating the image results. Default value is 40 (0-255).

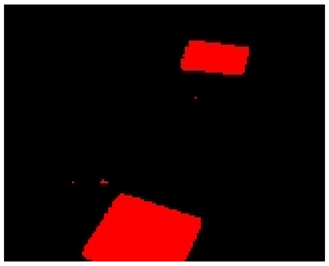
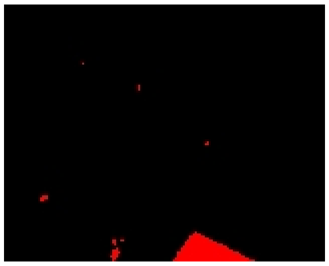
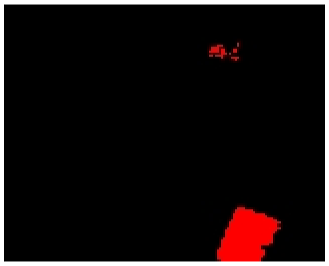
You can use this filter to focus the image towards certain colors even with diminished lighting conditions.



Let us review what the three test images currently look like now with more even lighting.

Using the **RGB_Filter** module to search for red (yellow and orange both contain red) reveals some interesting but incorrect results.

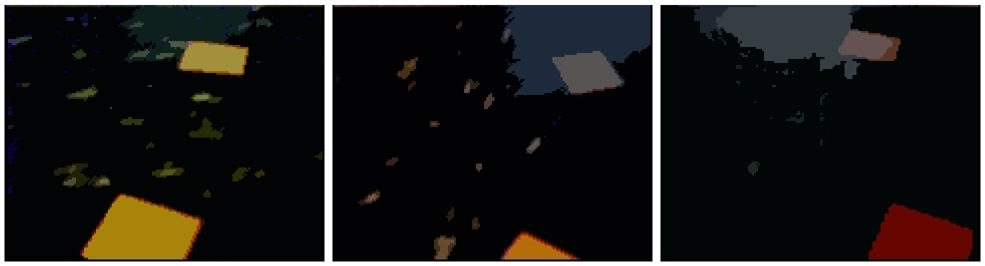




The first image seems fine but the last two mostly miss the upper square. If this happens during the course the robot will start turning around prematurely as it will think that the course has come to an end.

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 ....

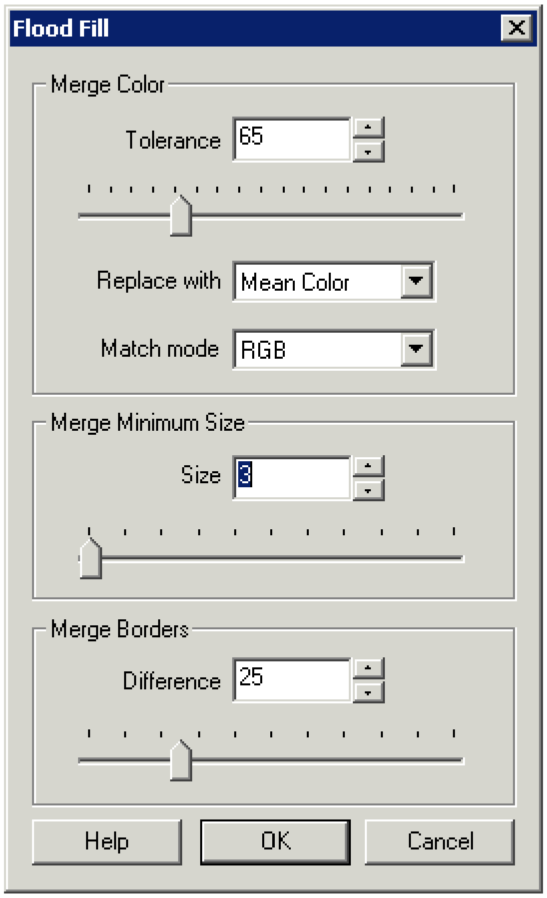
Flood Fill

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 as known in machine vision terms ... **a Blob** (not to be confused with the 70's movie of the same name, or any characters on the "The Biggest Loser".



Using the **Flood Fill** module (set to 65) 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.

Here is a example of what the $90 interface looks like for flood fill:



Flood Fill

Merge Color

Tolerance 65

Replace with Mean Color

Match mode RGB

Merge Minimum Size

Size 3

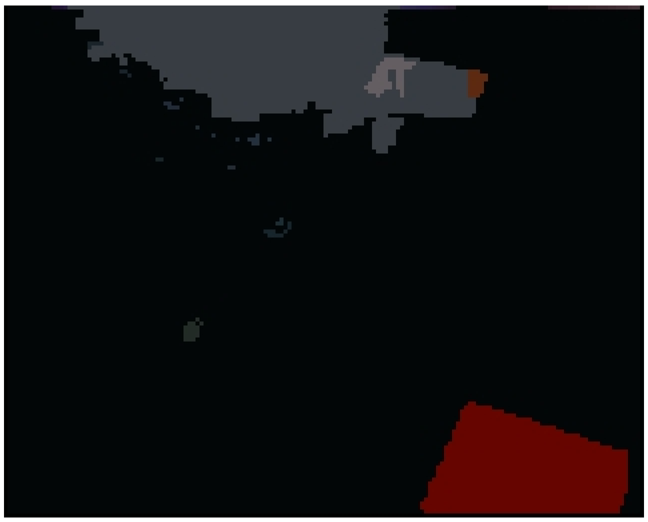
Merge Borders

Difference 25

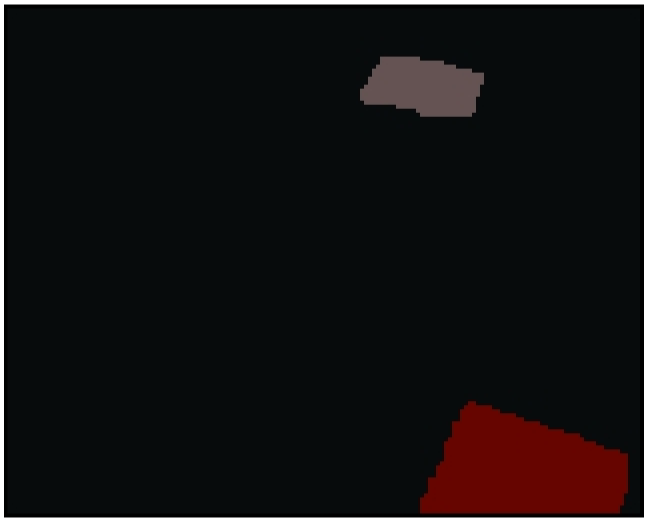
As you see they hide the details, but you can see the parameters they use to complete the task.

Flood fill can be done with a queue or using recursion. It can be a memory hog and time consuming. For those of you that took Data Structures from me you had to do a flood fill for the queue assignment.

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 technique in the flood fill to help merge blobs whose borders are not very strong.



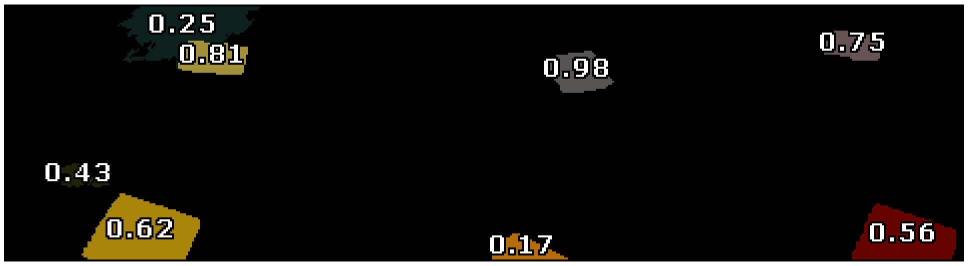
To further consolidate blobs into meaningful objects you can select neighboring blobs without a significant border (i.e. color change) between them to be merged. This is a process that happens after the initial objects have been segmented. This helps to merge similar blobs without increasing the color tolerance which causes bleeding into other unwanted objects.

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 a 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.

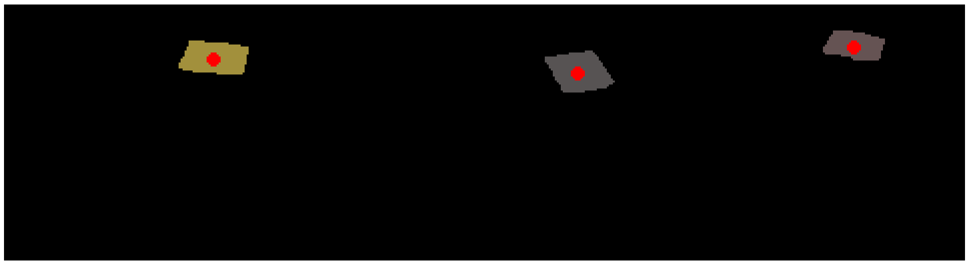
The blob filter allows you to add in descriptive attributes that rank each of the individual blobs according to the feature they represent. Thus, for a perfect square we would expect a rank of 1.0 whereas for blobs that are less square we would expect a <1.0 weight.



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.

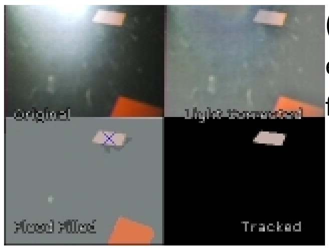
If is worth noting that the squares on the sides of the images may or may not be detected by the square shape as part of the square is cutoff from the image. This is not an issue as we also use the blob filter to remove any blobs that touch the border. The reasoning is that we know all squares will fit 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 occilate 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 technique we talked about last week 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.



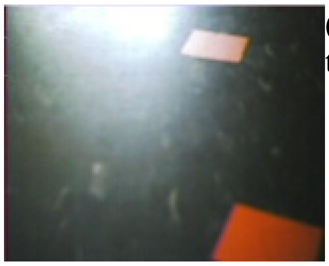
Note that if more than one square is visible on the screen the COG will return a average point between the two squares which is desirable as it evens out the transition from one square to the next.

Now that we have the point to move towards we need to use that information to control the Robot…….that will come next week.



(1.5 MB) Video of the SRV-1b as it makes its way along the trail. You can see each of the major steps along the way as the image is processed for lighting, flood fill and blob filtering.

(1.7 MB) Overhead video of the robot moving along the trail.



(1.3 MB) Standalone video of what the robot sees while moving along the trail for your own testing purposes.

Problems

It is worth pointing out some of the issues and mistakes that we experienced during the creating of this tutorial to help you solve similar issues.

1. Lighting - To illustrate different lighting techniques we purposely made the setup environment non-optimal to run the robot. In reality one would probably attempt as much as possible to ensure that the lighting requirements are better met and no sunlight glare is present. This could also have been better mitigated by using different tiles that reflect less light.

2. Blob-Filter - Removing blobs based on attribute values is somewhat of a black art. As with any image processing applications noise is ever present in the image and can cause some of the squares to become non-square's momentarily. This causes them to disappear from the robots tracking. Better flood filling and blob filtering might be able to reduce this loss of tracking seen as object flicker in the final videos.

3. Motion blur - Capturing images while the robot is moving causes motion blur in the captured image. This motion blur tends to cause the squares to become more irregular in shape and therefore not match the blob filter criteria. To reduce the effects of motion blur the robot is pulsed in-between image captures to ensure that the robot is not moving while an image is being captured. This does slow down the robot movement considerably but is a necessary requirement for stable operation.

4. Frame rate - Despite the SRV-1b being capable of more than 10fps when in 160x128 mode the low/bad lighting further reduces the processed frame rate to less than 4fps. This reduction also requires that the robot move slower in order to capture enough frames to react quickly enough to steer the robot. The pulsing added to reduce motion blur provided enough speed reduction to ensure that the robot could react to the visual scene fast enough.

5. Oscillation - There are still times where the robot has successfully passed a square but during the turn (due to the robot being a tracked vehicle) the robot may back up enough to again see the most recently passed square. This causes the turn to terminate and the robot to reengage tracking. Once passed, the robot returns to turning mode where the process can repeat itself. This issue can be reduced by further increasing the time0 after which the robot determines it is no longer seeing a square and when to start a turn execution. Addition filters can also be added to eliminate detection of any square in the lower part of the image while turning with the assumption that any square that should now be tracked would appear somewhat distant from the robot.