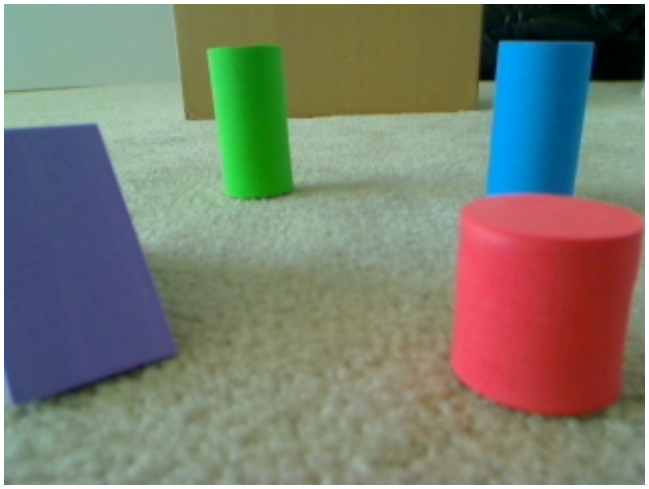
Obstacle Avoidance

Obstacle avoidance is one of the most important aspects of mobile robotics. Without it robot movement would be very restrictive and fragile. This tutorial explains several ways to accomplish the task of obstacle avoidance within the home environment. Use the Centibots so you can experiment with the provided techniques to see which one works best. Use some still images of the trash cans first to see if you pick the correct path.

There are many techniques that can be used for obstacle avoidance. The best technique for you will depend on your specific environment and what equipment you have available. We will first start with simpler techniques that are easy to get running and can be experimented on to improve their quality based on your environment.

Let's get started by first looking at an indoor scene that a mobile robot may encounter.

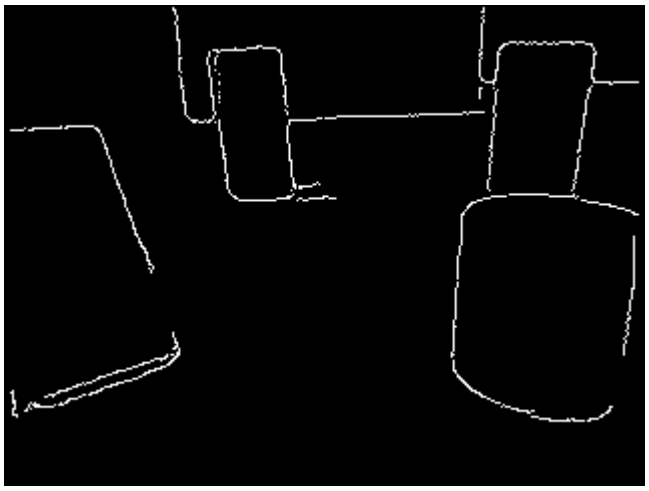


Here the robot is placed on the carpet and faced with a couple obstacles. The following algorithms will refer to aspects of this images and exploit attributes that are common in obstacle avoidance scenarios. For example, the ground plane assumption states that the robot is placed on a relatively flat ground (TangoBots will not be offroading) and that the camera is placed looking relatively straight ahead or slightly down (but not up towards the ceiling).

By looking at this image we can see that the carpet is more or less a single color with the obstacles being different in many ways than the ground plane (or carpet).

Edge Based Technique

The first technique that exploits these differences uses an edge detector like Canny to produce an edge only version of the previous image. Using this module we get an image that looks like:



You can see that the obstacles are somewhat outlined by the edge detection routine. This helps to identify the objects but still does not give us a correct bearing on what direction to go in order to avoid the obstacles.

Canny Edge Detection

Canny Edge Detector first specified by John Canny is his paper "A Computational Approach to Edge Detection". The Canny edge detector is regarded as one of the best edge detectors currently in use. The basic algorithm can be outlined as follows:

1. Blur the image slightly. This removes the effects of random black or white pixels (i.e. noise pixels) that can adversely affect the following stages. Blurring is typically accomplished using a Gaussian Blur. However, you should be able to use a box averaging filter like the Mean filter to accomplish the same effect. The mean filter can be executed faster than a Gaussian blur so for real-time applications the computational advantage can be worthwhile.

2. Compute the x and y derivatives of the image. This is basically an edge detection step. Similar to the Sobel edge detector the x and y derivatives are simply calculated by subtracting the values of the two surrounding neighboring pixels depending on which axis is being computed (left and right for x derivative and top and bottom for y derivative).

3. From the x and y derivatives you can compute the edge magnitude which basically signifies the strength of the edge detected at the current point.

4. If you were to look at the edge image at this point it would look very similar to the Sobel edge filter. Namely, the edges would be thick in many areas of the resulting edge image. To 'thin' these edges a nonmaximal filter needs to be applied that only preserves edges that are the top or ridge of a detected

edge. The nonmaximal filter is applied by first analyzing what the slope of the edge is (you can determine this by the sign and magnitude of the x and y derivatives calculated in the previous steps) and use that slope to determine what direction the current edge is heading. You can then determine the two edge magnitudes perpendicular to the current heading to determine if the current pixel is on an edge ridge or not. If the current pixel is on a ridge its magnitude will be greater than either of the two magnitudes of the perpendicular pixels (think of walking on a mountain ridge with the ground decreasing on both sides of your heading). Using this ridge detection and subsequant elimination will result in very thin lined edges.

5. Once the edges have been thinned the last step is to threshold the detected edges. Edge magnitudes above the upper threshold are preserved. Edges below the upper threshold but above the lower threshold are preserved ONLY if they connect (i.e. 8 neighborhood touching) to edges that are above the upper threshold. This process is know as hysteresis and allows edges to grow larger than they would by using a single threshold without introducing more noise into the resulting edge image.

Step two:

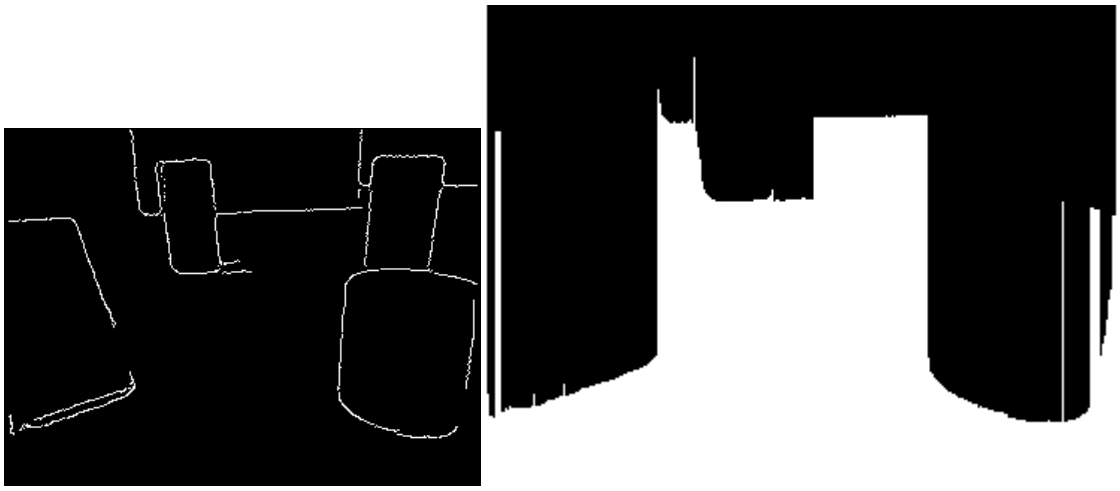
The next step is to understand which obstacles would be hit first if the robot moved forward. To start this process we use a Side_Fill technique to fill in the empty space at the bottom of the image as long as an edge is not encountered. This works by starting at the bottom of the image and proceeding vertically pixel by pixel filling each empty black pixel until a non-black pixel is seen. The filling then stops that vertical column and proceeds with the next.

SideFill Top to Bottom

The SideFill technique fills the black area of an image starting from the specified side and proceeds until a non-black pixel is found. In the case of filling from the top side to the bottom of the image you can think of water drops starting from the top of the image and stopping when they encounter a non-black pixel. As the water fills the image the dark area on top becomes white.

This function is very useful to create a silhouette outline of an edge boundary used for skyline or ground plane detection.

You will quickly notice the single width vertical lines that appear in the image.

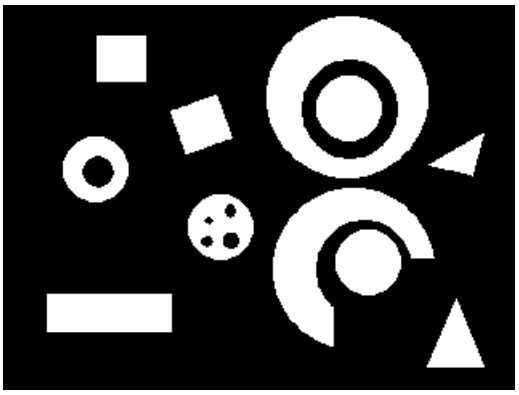


These are caused by holes where the edge detection routine failed. As they specify potential paths that are too thin for most any robot we want to remove them as possible candidates for available robot paths. We do this by using the Erode Technique and we do an eroding or shrinking the current image horizontally by an amount large enough such that the resulting white areas would be large enough for the robot to pass without hitting any obstacle. You can chose a horizontal threshold in relation to what you think is equivalent for the TangoBots.

Now that we have all potential paths we smooth the entire structure to ensure that any point picked as the goal direction is in the middle of a potential path. This is based on the assumption that it is easier to understand the highest part or peak of a mountain as compared to a flat plateau. Using Smooth Hull we can round out flat plateaus to give us better peaks.

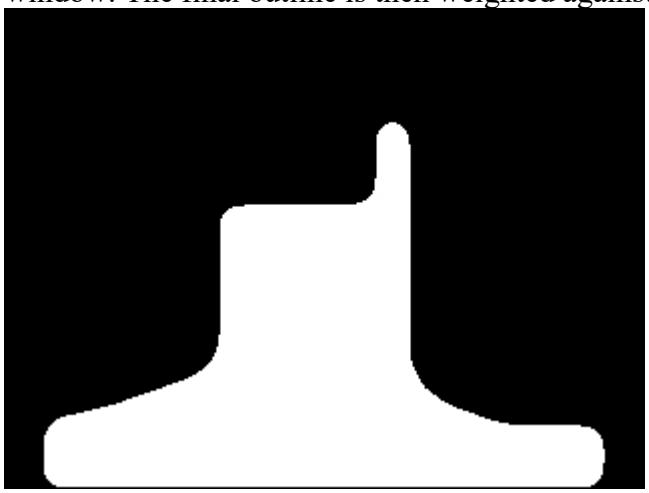
Example

Source

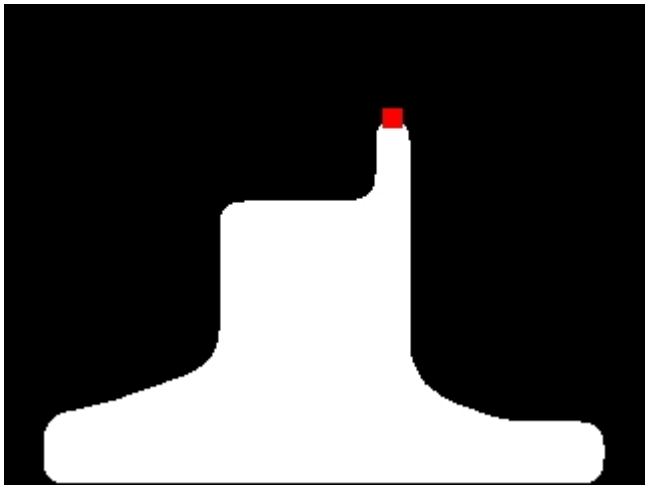


Smoothed

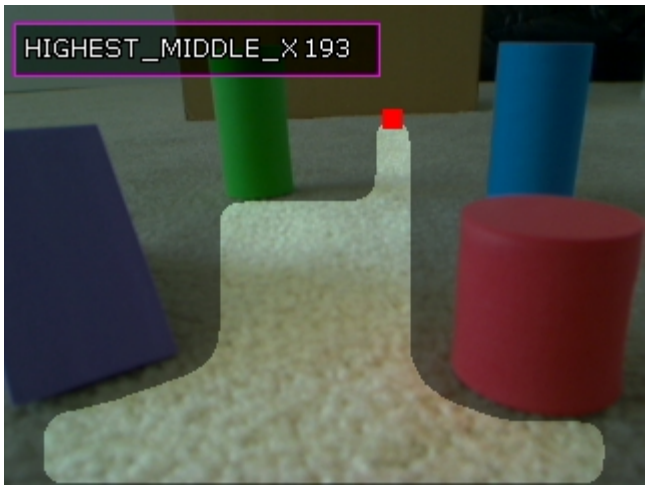
The smooth hull will smooth a blobs shape. The blobs perimeter is averaged within the specified window. The final outline is then weighted against the new averaged outline and the original.



Once this is done we now need to identify the highest point in this structure which represents the most distant goal that the robot could head towards without hitting an obstacle. Based on the X location of this point with respect to the center of the screen you would then decide if your robot should move left, straight, or right to reach that goal point. The highest point is identified by a red square.



Finally just for viewing purposes we merge the current point back into the original image to help us gauge if that location appears to be a reasonable result.



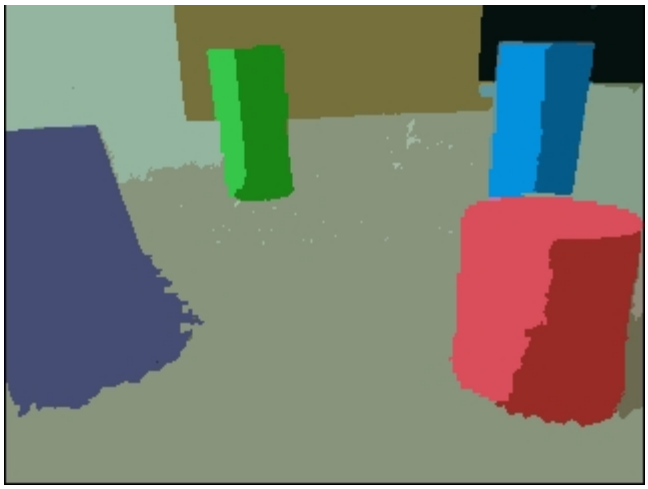
Given this point's X location at 193 and the middle of the image at 160 (the camera is set to 320x240) we will probably move the robot straight. If the X value were > 220 or < 100 we would probably steer the robot to the right or left instead.

Some other results using this technique.

This works reasonable well as long as the floor is a single color. But this is not the only way to recognize the floor plane ...

Blob Based Technique

The second technique exploits the fact that the floor is a single large object. Thus starting with the original image we can segment the image into a smaller number of colors in order to connect pixels into blobs that we can process. This grouping can use either the Flood Fill module or the Segment Colors module. Using the Using the flood fill module the image becomes



Flood Fill explanation

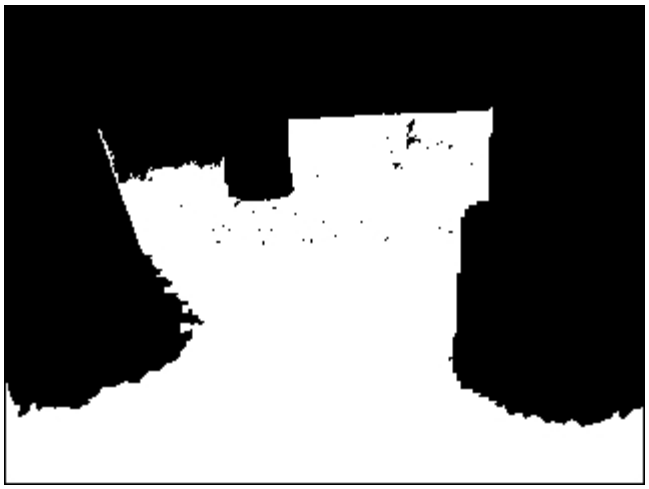
The Flood Fill module functions similar to a paint program's "flood fill" in that it fills areas of common color (within a degree of error) with a single color. This is very useful in segmenting images into discrete color blobs that can then be processed for shape or other features. Note that as the error threshold increases you will notice bleeding of one color area into another until the entire image becomes one color.

Once this process is complete there are often many pixel irregularities remaining due to interpolation that occurs between objects. These pixels can be removed by merging them with their closest larger neighbor. The size of these pixel groups that can be merged can be specified in the "Merge Minimum Size".

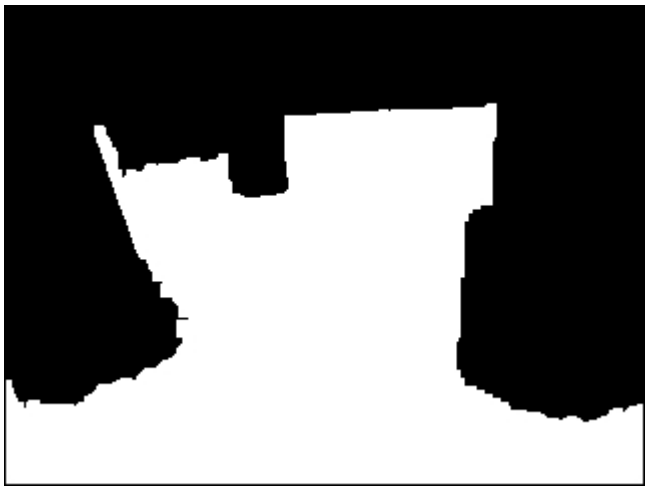
To further consolidate blobs into meaningful objects you can select neighboring blobs without a significant border (i.e. color change) between them to be merged. This is a process that happens after the initial objects have been segmented. This helps to merge similar blobs without increasing the color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

Next Step

The next step is to isolate the largest blob in the image which is assumed to be the floor. This is done using the Blob Size filter which is set to just return the single largest blob in the image.

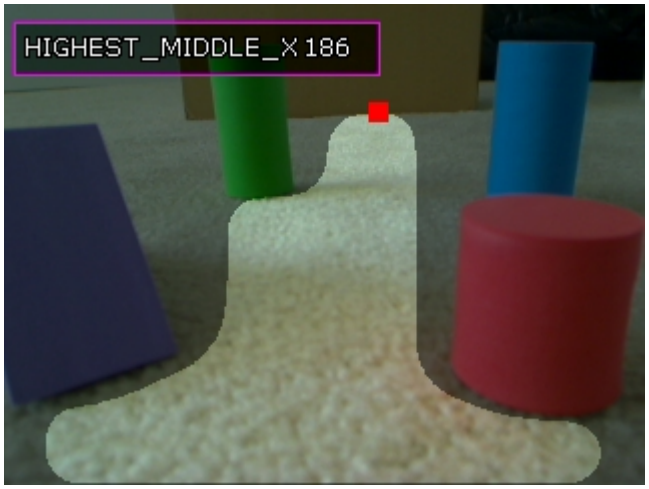


We then dilate this image by 2 pixels using the Dliate to close all the small holes in the floor blob.



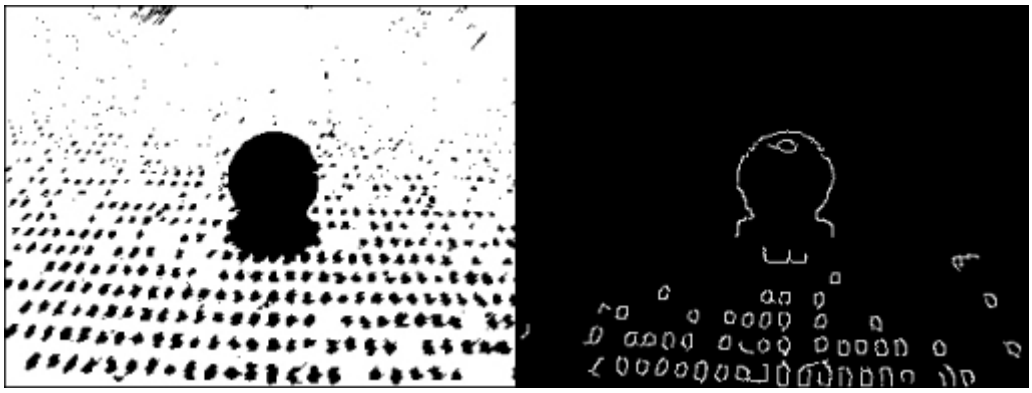
Then we negate this image and use the same Side Fill module as before to determine the possible vertical routes the robot could take. We need to negate the image prior to this module as the Side_Fill module only fills black pixels. In the above image the object to be filled is white and thus when negated will become black.

From here on the stages are the same as the previous technique. Namely Erode to remove small pathways, smooth the resulting object and identify the top most point. The final image looks similar to the previous technique.



The results are very similar but the first technique exploited edges whereas this one exploited connected pixels of similar color. But the issue of the similar colored floor plane still remains. What happens if you do not have the same colored carpet? For example, suppose that you have a high frequency pattern in a carpet.

The resulting edge and blob based techniques will not work as the blob and edge detection will pick up on the small patterns of the carpet and incorrectly see them as obstacles.



You can notice the failure of both these techniques in the above images which if fully processed would only see non-obstacle space in the lower 10 pixels of the image. This is clearly incorrect!

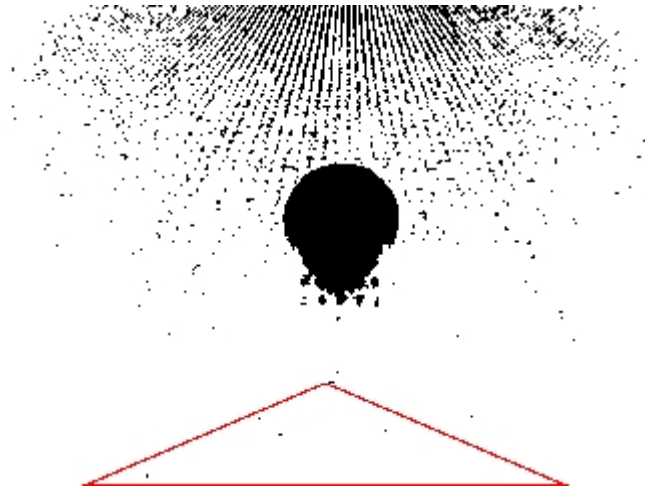
The Blob Size filter can be used to remove objects below a certain size. The Threshold histogram is used to threshold the image into object/no object values. Once objects have been identified you can use the 'size histogram' values to remove objects below a certain pixel size. Note that the pixel size is the objects pixel area.

Floor Finder Technique

To improve on our last image we will need to understand that the carpet or floor plane contains more than one pixel color. While you could detect both colors and perhaps merge them in some way an easier approach is to make an assumption that the immediate foreground of the robot is obstacle free. If we were to sample the colors in the lowest part of the image which is the immediate space in front of the robot we could use these color samples and find them in the rest of the image. By searching for all pixels who share the same or similar color to those pixels in this sample space we can theorize that those pixels are also part of the floor plane.

This process can be accomplished using the Floor Finder module. Given our test image

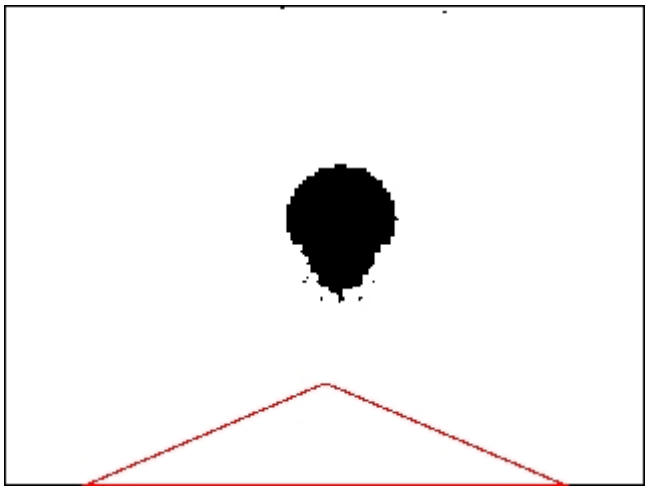
we can run the floor finder module to procedure a theoretical mask of what the floor might be.



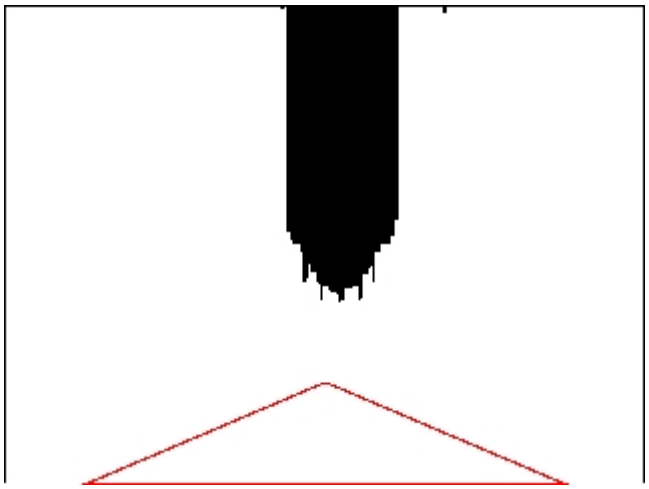
Floor Finder

The Floor Finder module is used to identify the floor area within an image. The assumptions that make this possible are that the robot or camera is on a planar floor that extends from the bottom of the camera image outwards away from the camera. This assumption is required since the Floor Finder module will sample the pixels in the bottom of the image area (the sample space) and use those pixels to identify similar pixels in the rest of the image. Thus it is assumed that the floor space right in front of the robot is relatively free from obstacles and represents a portion of the floor.

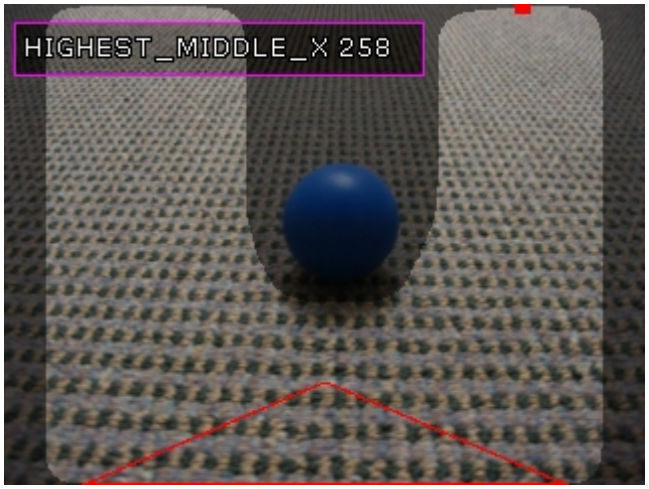
We can see the red triangle that represents the sample area that is used to understand what pixel colors are likely to be floor pixels. The white pixels in the above image now represent all pixels in the image that are similar to those found in the triangle. We can see that this works quite nicely to segment out the floor plane. We then dilate (grows the current white image. Objects that are close to each other will merge based on the dilation count) the image to remove small holes in the floor plane.



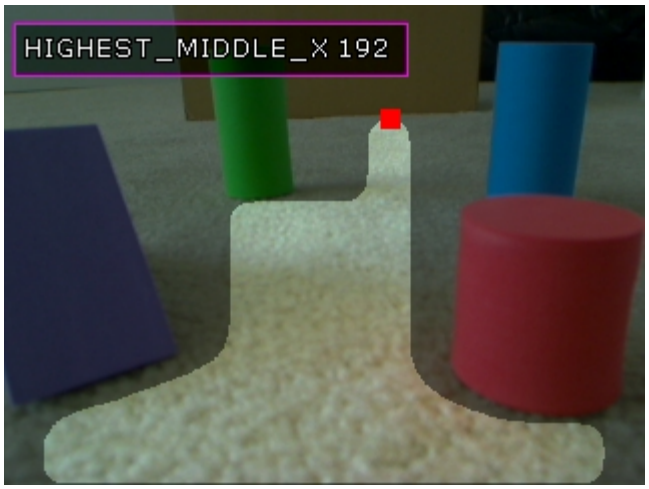
Then we negate this image and use the same Side Fill module as before to determine the possible vertical routes the robot could take. We need to negate the image prior to this module as the Side_Fill module fills black pixels. In the above image the object to be filled is white and thus negated it will become black.



Following a similar sequence as before we Erode to remove small pathways, smooth the resulting object and identify the top most point. The final image is used to verify the recommended goal point.



We then run the same technique over the previous image to check if that still works.



Which it apparently does! So the floor finder module has allowed us to use a single technique for both similar colored carpet to one that has a lot of small internal patterns in it.

Obstacle Avoidance

Notes:

1. Three techniques were discussed with the final technique being the most stable given the test images we used. Your experience will be different. It is worth testing each technique on your own images to get a sense of how stable they are and which one will work best for you.

2. Keep in mind when moving the code into a robotic control that the horizontal erode is used as a gauge to the robot's width. You will have to experiment with your setup to determine which erosion width is best for your robot.

3. The final variable displayed has the X value of the goal point. That X value should be then used by your application to create a value that will control your motors. It could be as simple as

```
//psuedocode….

x = GetVariable("HIGHEST_MIDDLE_X")
midx = GetVariable("IMAGE_WIDTH") / 2
leftThreshold = midx - 50
rightThreshold = midx + 50


//turn right
if x <= leftThreshold then
  SetVariable "left_motor", 0
  SetVariable "right_motor", 255
//turn left
```

```
else
  if x >= rightThreshold then
    SetVariable "left_motor", 255
    SetVariable "right_motor", 0

//go straight
  else
    SetVariable "left_motor", 255
    SetVariable "right_motor", 255
  end if
end if
```

which will rotate the left and right motors towards the goal or go straight if in the middle. Keep in mind that the above only creates two variables (left_motor and right_motor) that need to be selected in the appropriate hardware control module. You will have to adjust the values for your robot based on what servos, motors, etc. you are using.