Project Update #1 – Detecting and Solving Images Containing Mazes

Anthony Salani

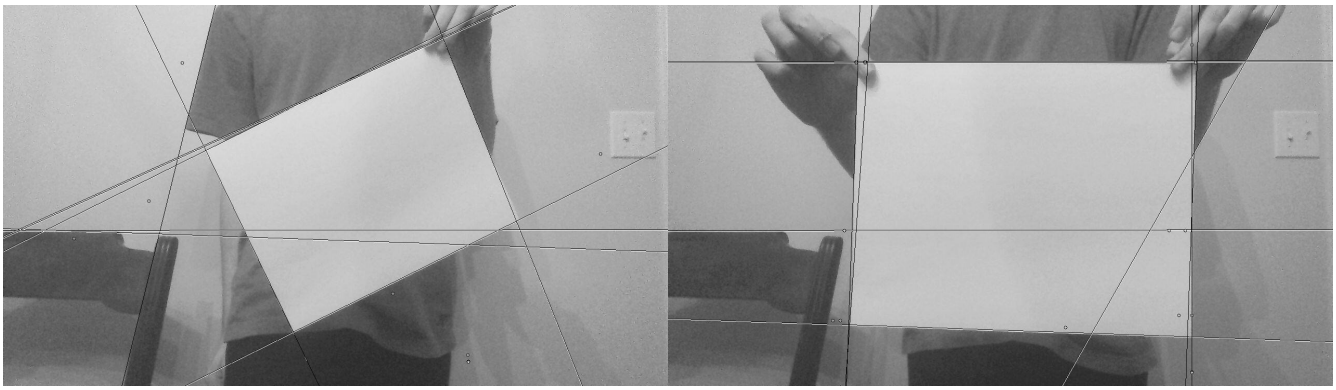
The bulk of my work in this project up to this point has been spent trying to get rectangle detection to work correctly. I am currently able to take a given image and draw lines that demarcate rectangles. Sometimes it works.

Explanation of Current Work:

Currently, my program takes an image supplied to it from the command line, and converts it to black and white. Once it is in black and white, it applies an edge detection filter to find edges, and then runs a Hough transform on the resulting matrix of edges. The Hough lines are then converted into line segments with a specific start and end point. These lines then have their intersections computed. Once the lines and intersection points are determined, they are drawn on the black and white image and presented to the user. This last step is only for debugging because, as described below, the process of detecting rectangles does not work as well as I'd like it to in order to advance to the next step.

A great deal of time has been spent trying to find ways to remove erroneous lines, and trying to find a way to accurately determine points of intersections between the lines returned from the Hough transform.

Examples of Working Rectangle Detection:



Pictured above is a half-way working rectangle detector. The lines represent detected straight lines, and the points represent (incorrectly calculated) intersections between two given lines. I have since been able to refine the process of determining intersections, but they are still not correct.

Rectangle detection works well in situations where the rectangle to be detected is a significantly different color from the background, and when only transformations representable with a perspective transformation are applied to it. In the examples above, my shirt and the wall behind me contrast well enough with the sheet of paper to detect those edges. Also, the paper is not deformed due to flexing. Flexing turns what would be a quadrilateral into a polygon with five or more sides at a minimum, and is much harder to extract a rectangle from.

Examples of Failing Rectangle Detection:

Above are two examples of improperly detected rectangles. The image on the left contains a rectangle drawn in a thick, black marker which is detected over the rectangle it is contained within. It also picks up too much noise from the cluttered background. Different points in the ceiling and the furniture behind me are also creating too many irrelevant lines.

The image on the right is picking up some noise from my sleeve, furniture, and the four rectangles drawn on the paper. Passing these tests are important, because a maze naturally contains many straight lines, which will potentially end up as noise when detecting the rectangle they are drawn on. Finding solutions that would potentially remove these lines is difficult. I can refine the parameters passed in to the Hough transform to try and detect only the longer lines somehow. I can also find a metric to cull lines that are easily detected as not part of a rectangle. Lastly, I can average together Hough lines that are very close or overlapping significantly. The last approach in particular would drastically reduce the number of lines returned in the above two images, and the implementation is significantly more trivial than detecting which lines are not part of a rectangle (although this will inevitably have to be done anyway).

Another solution I am considering is to increase the number of lines detected by the Hough transform and, using the averaging approach I mentioned above, vote on the lines that are the longest and most often represented. In the images above (even the ones deemed successful), there are often single stray lines that don't form the rectangle, and many overlapping lines that form the sides of the rectangle. The edges of the rectangle are more often represented, and this is conducive to the idea of voting on features as described in class.

From Here:

Once I have the rectangles demarcated with lines, the next step is to determine the largest rectangle amongst them and compute a homography for it. Luckily, I will actually be able to use the findHomography function. A perspective transform from the computed homography will be applied, cropping out noisy backgrounds. From here, all that needs to be done is find the lines in the maze, convert them to a navigation mesh, and solve the navigation mesh. The last two steps in particular are incredibly easy.