Curtin University, Department of Computing

Machine Perception Assignment: Semester 2, 2018

Assignment Cover Sheet and Declaration of Originality

This form must be completed, signed, scanned and attached as the front page of your documentation.pdf file in your assignment submission.

Family Name: RHIND

Given Name(s): CONNOR

Student ID: 1739 1267

Stage Complete (student to fill in):

Circle one of the following: 1 (10%) 2 (20%) 3 (30%) 4 (50%) 5 (70%) [✓ 6 (80%)]

Final Stage complete? Circle one of the following: [✓ YES (+20%)] NO (+0%)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: [illegible]

Date of Signature: 5/10/2018.

By submitting this form, you indicate that you agree with all the above text.

Connor Rhind
Assignment
Machine Perception COMP3007

Design Decisions:

The layout of this design document will recount the initial ideas upon reading the assignment specification all the way through to how the final product works and all issues and solutions I came across.

My approach for the assignment was start on “setD” and never go back (except during testing). The initial approach was going to be to use the canny transform on the image then attempt to use the Hough line transform to predict the rectangle outlines, from here I was planning to use geometry to find the corners from intersecting lines and rotate the image using OpenCV’s affine transform to get a “true” image to work on. I quickly realised that simply using the canny transform on the image wasn’t going to work as far too much of the materials surface texture was being picked up in the transform, and the Hough lines were never reliable enough to provide a useable rectangle as lines between surface texture and hazard symbol; I wasted approximately half a day (6 hours) attempting to tweak the parameters of both transforms to find the happy medium to get reliable results.
After spending a substantial amount of time in the OpenCV documentation and various forums, I moved away from Hough line transform and moved over to OpenCV’s approxPolyDP() function which takes in a binary image (either canny or threshold) and finds contours that approximate a polygon (any number of corners) from here I used some basic logic to only consider polygons with 4 edges and a minimum area. It should be noted that initially I was getting multiple rectangles inside of one another (internal and external diamonds of the hazard label), later I found that approxPolyDP returns the hierarchy associated with the returned contours and the most outer one was acquired.
The next issue that I was having was the canny was still picking up far too much of the surface textures causing the diamond contours to be missed.
This next section is where 80% of the time spent on this assignment went; attempting to perform various transforms sequentially to remove or decrease the amount of surface noise. I am not going to list every combination I tried but the approach was to output the resulting image at each transform level and use common sense to watch what appeared to be working and continue with that path, the general format was some form of threshold, some filtering and end on a canny transform.
Once I was able to get the major of the hazard diamonds I moved over to the symbol detection. The initial idea was to use MSER to find the symbol blob in a region that I knew It to be located and then do various measurements such as relative width of blob, relative height of blob and ratio of white to black pixels in a given blob area to help classify the various symbols. This did not work. After much wasted time on reading about MSER and other feature detection methods, I found OpenCV’s ORB (oriented FAST and rotated BRIEF) which was able to generate key points in a reference image and then measure the hamming distance for those features in a provided image. This method with little pre-processing provided a high percent of correct classifications.

For the OCR I used TesseractOCR, which out of the both was able to get most words in the diamonds, the percent of read characters in the diamonds increased after I applied a threshold transform and bilateral blurring as pre-processing for the read image. There were small errors in some of the words or some words were dropped such as spontaneous was being read as “spont!IIos” and wet was being dropped in dangerous when wet. I implemented a distance comparison from the read words to a set of predefined corpus of words found on the hazmat labels. The distance calculation uses pythons difflib library, as far as I can find the distance measure is calculated using a form of edit distance. I threshold the percentage difference and anything less than 60% match is assumed to be a false read and is left as “NONE”.
The OCR of the class was one of the other major issues, for much of the assignment time I could not get any of the numbers recognised using tesseract no matter what pre-processing I applied to the image before being read. I considered switching from using tesseract as the class classifier to using ORB, the major drawback to using the ORB method over Tesseract was classification time. Even though the ORB classifier was written in C, Tesseract was the model of the trained Neural Net. After reading through the documentation on the tesseract Wiki on GitHub I found the config settings which allowed me to alter the default settings to one which will attempt to read a single character and set which characters it is looking for (first ten digits 0-9).

Image Pre-processing:

The order of manipulation for retrieving the diamonds from the input images are an adaptiveThreshold to give a binary image, the idea was to get every image into a standard form so that the following pre-processing would work for all images. A gaussian blur was next to attempt to soften the surface textures picked up by the threshold, the gaussian was chose any other blur as there was so much noise from surface texture I only wanted a blur that was local to the pixel, the median blur was more likely to catch much of the noise in the kernel while the gaussian didn’t (as much). After the gaussian blur I use a bilateral filter to attempt to maintain edges while still removing much of the remaining surface textures, the bilateral was chosen simply because it is the only filter I am aware of that is edge preserving, the only issue I was worried about was the computation time of the bilateral filter is much higher than other filters of the same kernel size. Final a canny was used on the image to extract edges for the findContour method, the canny edge detection was chosen because it provided almost clear lines for the contours to pick up and find. Later in my research I had found a kernel that was line specific and was incredible at picking up the diamonds in the image with no pre-processing. If I had more time I would have liked to implement my own Canny edge detection and use that kernel instead of the Sobel that is built into Opencv’s Canny edge detector.

Image Processing:

Once the canny edge detection was completed Opencv’s findContours method is used to retrieve the contours of any diamonds it believes are found in the image. It should be noted that findContours inherently finds the outer most contour of those that share the same geometric centre. The findContours method was used over the initial idea of using intersecting Hough lines to find corner points as the Hough lines method was unreliable due to the large volume of noise in the image, the findContours method was also nicely wrapped up and ready to use.

Feature Recognition:

Once the image was cropped and transformed to extract all diamonds in the image each hazard label is subject to symbol recognition, text recognition, colour recognition and class recognition.

Symbol Recognition:
For the symbol detection I use the ORB feature detector and the BFMatcher to compare the symbols with label to the symbols that are located in the image. This method is definitely not the most computational efficient method considered as BF stands for brute force, but the accuracy was very high comparatively. When the program is run the symbols with labels are run through ORB and the key points and descriptors are stored for later use. Once the unclassified image is presented it is run through ORB and using the BFMatcher, which measures hamming distance between key points, the closest 5 matched points are stored for all comparison symbols then finally the one with the smallest hamming distance is assumed to be the correct classification. Using the computationally worse ORB classification method over the initial idea of using height, width and white to black pixel ratios to determine class was chosen for lack of time. In the future I would have liked to have done relative predictions combining both methods to give a percentage confidence of symbol classification and why.

Text Recognition:
For text recognition I use the tesseract OCR as it is well established in the computer vision forums. Unfortunately, I did not have a chance to use any other OCR methods for text recognition. In the way of pre-processing I cropped the initial image to an area that I knew the text to be located and then used a threshold to get black text on a white background, which according to the documentation Tesseract prefers. With the output of the tesseract read I apply a sanity check on the word to ensure that it matches one of the entries in the corpus of expected words within 50% similarity. If I had more time I would have liked to have done a form of blob detection to feed into the crop to get just the text instead of using a general location where I assume the text to be, this sort of approach would have assisted me in classifying the different sized text such as “Dangerous When Wet” where each word is a different size and would come up as its own blob.

Colour Recognition:
For the colour recognition I crop the image to two small areas on the top and bottom and find the average pixel value in RGB and use that as the input sample for the KNN classifier trained on colour points from the images. Ideally an approach that was more robust for acquiring the sample point would be preferred such as majority average for multiple samples from different points around the diamond.

Class Classification:
Similarly to the text recognition, I use the Tesseract OCR to acquire the class digit and sanity check it against a dictionary of expected digits. The class classification is the worst performing part of the entire program, and during research I attempted to use the ORB method of classification for the digits but it had very similar performance leading me to believe it is a pre-processing issue. The pre-processing is very similar to the text recognition pre-processing; I crop the input image to where I believe the class will be and then threshold it. With more time I would have liked to have done the blob detection method on the digit to find it approximate location before cropping.