FP.1 Match 3D Objects

- I implemented it inside the function “matchBoundingBoxes”.
- I needed to write a helper function “findBB” to find the bounding box for a point by searching in a vector of bounding boxes.
- In “matchBoundingBoxes”, I used a 2-D array to count the number of matches between bounding boxes.
- At the end of the function, I choose the max count to find the best match.
- When storing inside the map, we have to use the box ID value: not the index of the box inside the vector.

FP.2 Compute Lidar-based TTC

- I used the function “computeTTCLidar” to calculate the TTC based on LiDAR.
- I have also developed a helper function “findMinX” to find the minimum X value. I used it for both previous and current.
- For the purpose of FP.5, I made the function “findMinX” return the mean value as well.
- The function “findMinX”, starts first by calculating the mean X value.
- Then, looping on all the LiDAR points, the difference between each X value and the calculated mean value is evaluated. If the difference is greater than a defined threshold value, the point is filtered out and considered as an outlier.
- Using a threshold of 10cm, results were reasonable.

FP.3 Associate Keypoint Correspondences with Bounding Boxes

- I implemented this inside the function “clusterKptMatchesWithROI”.
- The implementation consists of 2 steps:
    - Find the matches
        - Loop on all the match points.
        - Check if the match pairs lie within the bounding box or not.
        - Push the found matches in the vector.
    - Filter the matches whose distance is relatively far from the mean distance.
        - I chose the difference threshold to be equal to the mean value.
        - Remove the entries from the vector.

FP.4 Compute Camera-based TTC

- This is implemented in the function “computeTTCCamera”.
- To compute the TTC, I used 2 nested loops to measure the distance between each 2 points of the matched points.
- To filter outliers, a minimum distance is defined in a new file “config.h” with a value of 100.
- In the for loops, the distance ratio is calculated.
- At the end, the median of the distance ratio is calculated in order to calculate the TTC.
- I implemented a helper function to calculate the median called “getMedian”.

FP.5 Performance Evaluation 1

- To analyze the performance of Lidar TTC, I increased the threshold value of filtering Lidar points to 1m instead of 10cm. i.e. no filtering is done at all.

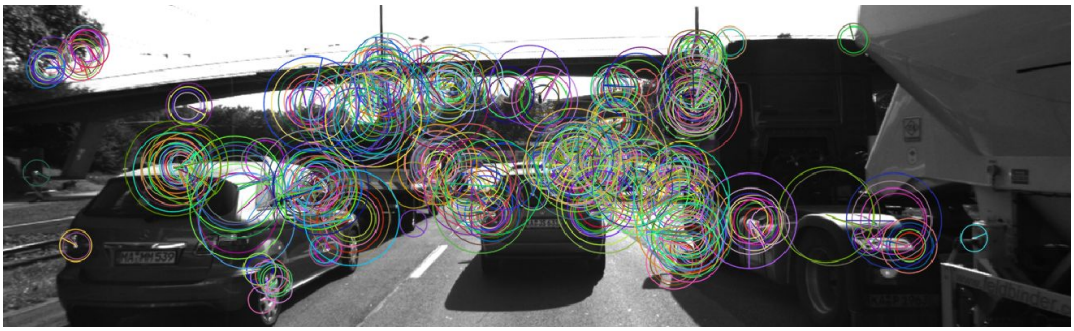
- To collect the data, I added “cout” statements whose result is seen in the attached file FP5.txt
- From the file, here are the values of TTC Lidar for 4 images that do not sound reasonable (images start with Id =0).
    - Image 4: 7.11572s
        - Max difference between xMean and xMin = 16cm
    - Image 7: 34.3404s
        - Max difference between xMean and xMin = 15cm
    - Image 12: -10.8537s
        - Max difference between xMean and xMin = 23cm
    - Image 17: -9.99424s
        - Max difference between xMean and xMin = 21cm
- Conclusion:
    - Having LiDAR outliers in those readings that are not filtered causes those strange values of TTC Lidar.
-
- To fix this:
    - I re-configured the threshold for filtering LiDAR outliers to be 10cm. Then, I obtained the following new results for TTC.
        - Image 4: 13.8342s
        - Image 7: 13.2877s
        - Image 12: 10.1s
        - Image 17: 10.2926s
    - The stdout is found is in the file FP5_fix.txt

FP.6 Performance Evaluation 2

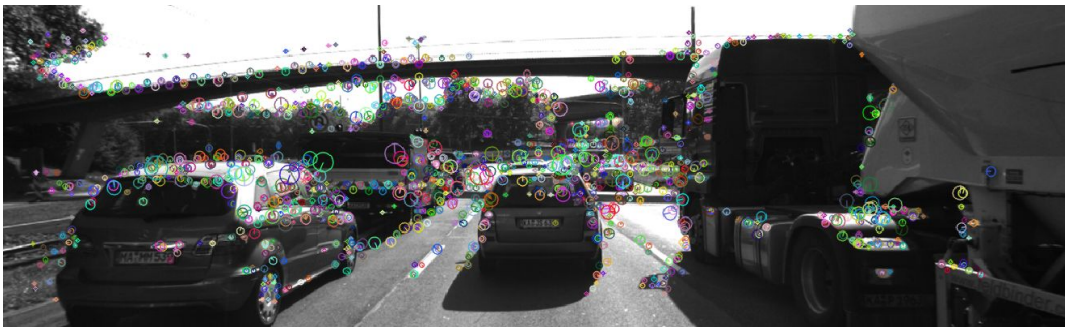
- I renamed main into “doMain” passing all the types of detector, descriptor, matcher, selector as input arguments. I fixed the matcher type of “Brute Force” and selector to “KNN”.
- The results have been printed from the code into the stdout in csv format that I redirected to a csv file (FP6.csv). I converted the csv to excel (Udacity -Sensor Fusion - 3D Objects - FP6.xlsx).
- Results:
    - Detectors AKAZE, SHITOMASI have good results with all descriptors.
    - Detectors FAST, BRISK: All the results with all possible descriptors are good except only one case for each of them in image 5 (FAST-BRISK, BRISK-BRIEF). In that case, the TTC value is larger than the mean of other detectors as well as LiDAR.
    - Detector SIFT: Only one result is slightly big in image 4. This happened only for the following descriptors: BRIEF, BRISK, FREAK. In that case, the TTC value is larger than the mean of other detectors as well as LiDAR. The other descriptors showed good results.
    - Detector ORB: 4-6 results of each of the 18 are wrong. This is independent of the descriptor type. In those cases, the TTC value is either large or -infinity.
    - Detector HARRIS: Almost all the results are bad where TTC is either too large, or negative or NaN or -Infinity.

- Summary of the results:
    - The occurrence of bad results is dependent mostly on the detector type.
        - AKAZE and SHITOMASI detectors are the best.
        - HARRIS detector is the worst.
- Analysis:
    - To find the root cause of bad results, I enabled visualization after the keypoint detection.
    - HARRIS: The number of keypoints within the ROI is too small. Hence, relying on the distance ratio between the matched points of those will not yield an accurate result.

    - ORB: The number of keypoints is not bad, but there is a big overlap between the points.



        - On the other hand, AKAZE detector that showed good results, had the detected keypoints as follows. In the image, we can see that there are many keypoints within ROI whose distance ratios can be used reliably to calculate TTC.



Additional Notes

- I refactored the function “clusterLidarWithROI” to minimize the amount of matrix multiplication for each LiDAR point. To do this, I calculated the product of “P_rect_00, R_rect_00, RT” only once.
- I created a new header file “config.h” to include all configurations for tuning the parameters as well as turning on and off logging of messages to the standard output.