Object Tracking in 3D Space

FP.1 Match 3D Objects:

This functionality is implemented in the **matchBoundingBoxes( )** method. The idea is to iterate over all matched keypoints and count correlations between previous and current bounding box detections.

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
    // Iterate over all keypoint matches (for current and previous frame)
    // Count bounding box correlations and store count in 2 dimensional map
    std::map<int, std::map<int, int>> boxCountMap;
    for (auto it = matches.begin(); it != matches.end(); ++it){
        // Get indecies from matches keypoints for previous and current image frames
        //int keyPtIdxPrev = it->trainIdx;
        //int keyPtIdxCurr = it->queryIdx;
        int keyPtIdxPrev = it->queryIdx;
        int keyPtIdxCurr = it->trainIdx;
        // Get actual keypoints
        cv::KeyPoint prevPt = prevFrame.keypoints[keyPtIdxPrev];
        cv::KeyPoint currPt = currFrame.keypoints[keyPtIdxCurr];

        // Iterate over all bonding boxes in the previous frame and count keypoint correlations to bounding boxes of the current frame
        for (auto boxCurr = currFrame.boundingBoxes.begin(); boxCurr != currFrame.boundingBoxes.end(); ++boxCurr){
            // Check whether current frame keypoint is contained in any bounding box
            if (boxCurr->roi.contains(currPt.pt)){
                // Check whether the corresponding (matching) current keypoint is contained in any bounding box (in its current frame)
                for (auto boxPrev = prevFrame.boundingBoxes.begin(); boxPrev != prevFrame.boundingBoxes.end(); ++boxPrev){
                    if (boxPrev->roi.contains(prevPt.pt)){
                        // The two matched keypoints are contained in bounding boxes respectively
                        // Increase the bounding box correlation counter in the map
                        boxCountMap[boxCurr->boxID][boxPrev->boxID] += 1;
                    }
                }
            }
        }
    }
    // Print the Map
    /*...*/

    // Extract max values out of map
    for (auto boxCurr : currFrame.boundingBoxes){
        std::map<int,int> boxCountMapRow = boxCountMap[boxCurr.boxID];
        auto max = std::max_element(boxCountMapRow.begin(), boxCountMapRow.end(), [] (const std::pair<int,int> & p1, const std::pair<int,int> & p2) {return p1.second < p2.second; });
        //cout << "The max value for the current bounding box " << boxCurr.boxID << " is " << max->second << "   --> Link to current bounding box " << max->first << endl;
        //cout << "Current bounding box: " << boxCurr.boxID << " <-> linked to previous bounding box: " << max->first << " (Number of matches = " << max->second << ")" << endl;
        bbBestMatches[max->first] = boxCurr.boxID;
    }
}
```

FP.2 Compute Lidar-based TTC

This functionality is implemented in the **computeTTCLidar( )** method. The x distance mean of all (relevant) lidar points is being calculated in order to filter out outliers. Afterwards the lidar based TTC estimation is being applied.

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    //double xSumPrev = std::accumulate(lidarPointsPrev.begin(), lidarPointsPrev.end(), 0.0, [&](const LidarPoint pt1, const LidarPoint pt2){ return pt1.x + pt2.x;} );
    // Extract x coordinate from the entire point cloud information
    std::vector<double> xDistPrev;
    std::vector<double> xDistCurr;
    for (auto lidarPt : lidarPointsPrev){
        xDistPrev.push_back(lidarPt.x);
    }
    for (auto lidarPt : lidarPointsCurr){
        xDistCurr.push_back(lidarPt.x);
    }
    // Calculate mean
    double xMeanPrev = std::accumulate(xDistPrev.begin(), xDistPrev.end(), 0.0) / xDistPrev.size();
    double xMeanCurr = std::accumulate(xDistCurr.begin(), xDistCurr.end(), 0.0) / xDistCurr.size();
    // Remove points which are too far away from mean in negative direction (We only care about the closest points)
    double accuracy = 0.05;
    for (auto it = xDistPrev.begin(); it != xDistPrev.end(); ++it){
        if (*it < xMeanPrev - accuracy){
            xDistPrev.erase(it);
        }
    }
    for (auto it = xDistCurr.begin(); it != xDistCurr.end(); ++it){
        if (*it < xMeanCurr - accuracy){
            xDistCurr.erase(it);
        }
    }
    // Sort the x distance value so the closest lidar point can easily be extracted
    std::sort(xDistPrev.begin(), xDistPrev.end());
    std::sort(xDistCurr.begin(), xDistCurr.end());

    // Calculate TTC
    double deltaT = 1/frameRate;
    TTC = (xDistCurr[0] * deltaT) / (xDistPrev[0] - xDistCurr[0]);
    cout << "TTC based on point clouds is " << TTC << endl;
}
```

FP.3 Associate Keypoint Correspondences with Bounding Boxes:

This functionality is implemented in the **clusterKptMatchesWithROI( )** method. The idea is to iterate over every single keypoint match and check whether which keypoints are contained in a certain bounding box. In order to filter out incorrect keypoint matches, a euclidean distance mean is being calculated.

```
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
    // Calculate keypoint distance mean which should help to identify outliers i.e. mismatches
    std::vector<double> distances;
    for (auto kptMatch : kptMatches){
        // Get actual (matched) keypoints
        cv::KeyPoint keyPtCurr = kptsCurr[kptMatch.trainIdx];
        cv::KeyPoint keyPtPrev = kptsPrev[kptMatch.queryIdx];
        // Calculate eucledian distance
        double xCurr = keyPtCurr.pt.x;
        double yCurr = keyPtCurr.pt.y;
        double xPrev = keyPtPrev.pt.x;
        double yPrev = keyPtPrev.pt.y;
        double eucledDist = std::sqrt((xCurr - xPrev)*(xCurr - xPrev) + (yCurr - yPrev)*(yCurr - yPrev));
        distances.push_back(eucledDist);
        //cout << "Keypoint Distance = " << eucledDist << endl;
    }
    double kptDistMean = std::accumulate(distances.begin(), distances.end(), 0.0) / distances.size();
    //cout << " The mean is: " << kptDistMean << endl;

    // Loop over all keypoint matches and check whether they are contained in the current bounding box
    for (auto kptMatch : kptMatches){
        // Get actual (matched) keypoints
        cv::KeyPoint keyPtCurr = kptsCurr[kptMatch.trainIdx];
        cv::KeyPoint keyPtPrev = kptsPrev[kptMatch.queryIdx];

        // Check whether current kpt is contained in current bounding box
        if(boundingBox.roi.contains(keyPtCurr.pt)){
            // Check whether keypoint distance is not too far away from distance mean (in order to identify outliers)
            // If keypoint is valid, add the actual point and the matching ids to the bounding box object
            if (distances[kptMatch.trainIdx] < kptDistMean){
                boundingBox.keypoints.push_back(keyPtCurr);
                boundingBox.kptMatches.push_back(kptMatch);
            }
        }
    }
}
```

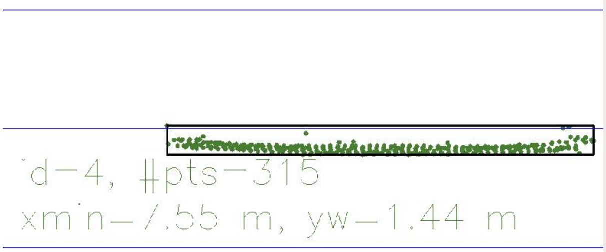
FP.4 Compute Camera-based TTC

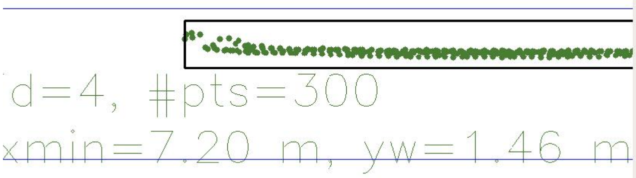
This functionality is implemented in the **computeTTCcamera( )** method. The TTC is computed based on distance ratio assumptions of keypoints. The code has been mainly taken from the corresponding exercise in the lesson.

```
void computeTTCCamera(std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
                      std::vector<cv::DMatch> kptMatches, double frameRate, double &TTC, cv::Mat *visImg)
{
    // compute distance ratios between all matched keypoints
    vector<double> distRatios; // stores the distance ratios for all keypoints between curr. and prev. frame
    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
    { // outer kpt. loop

        // get current keypoint and its matched partner in the prev. frame
        cv::KeyPoint kpOuterCurr = kptsCurr.at(it1->trainIdx);
        cv::KeyPoint kpOuterPrev = kptsPrev.at(it1->queryIdx);

        for (auto it2 = kptMatches.begin() + 1; it2 != kptMatches.end(); ++it2)
        { // inner kpt.-loop

            double minDist = 100.0; // min. required distance

            // get next keypoint and its matched partner in the prev. frame
            cv::KeyPoint kpInnerCurr = kptsCurr.at(it2->trainIdx);
            cv::KeyPoint kpInnerPrev = kptsPrev.at(it2->queryIdx);

            // compute distances and distance ratios
            double distCurr = cv::norm(kpOuterCurr.pt - kpInnerCurr.pt);
            double distPrev = cv::norm(kpOuterPrev.pt - kpInnerPrev.pt);

            if (distPrev > std::numeric_limits<double>::epsilon() && distCurr >= minDist)
            { // avoid division by zero

                double distRatio = distCurr / distPrev;
                distRatios.push_back(distRatio);
            }
        } // eof inner loop over all matched kpts
    }     // eof outer loop over all matched kpts
```

FP.5 Performance Evaluation 1:

A reasonable number for the TTC would be around 12 seconds. The lidar based TTC estimation does not work too bad. However, there are a few frames where the TTC estimation is significantly off. There are two example frames shown below. I have two reasons, why the TTC estimation might fail for some frames. The obvious one is lidar point outliers, which are not caught by the x distance mean filtering. (Adding distance filtering based on the y distance might lead to some further improvements in terms of robustness) Also, I realized that there are few frames where the bounding box matching was not successful. This means that keypoint matching failures (which highly affects the bounding box matching) can lead to some incorrect lidar point matches. Therefore, making YOLO and the keypoint creation more robust could potentially also lead to an improved and more stable lidar based TTC estimation.



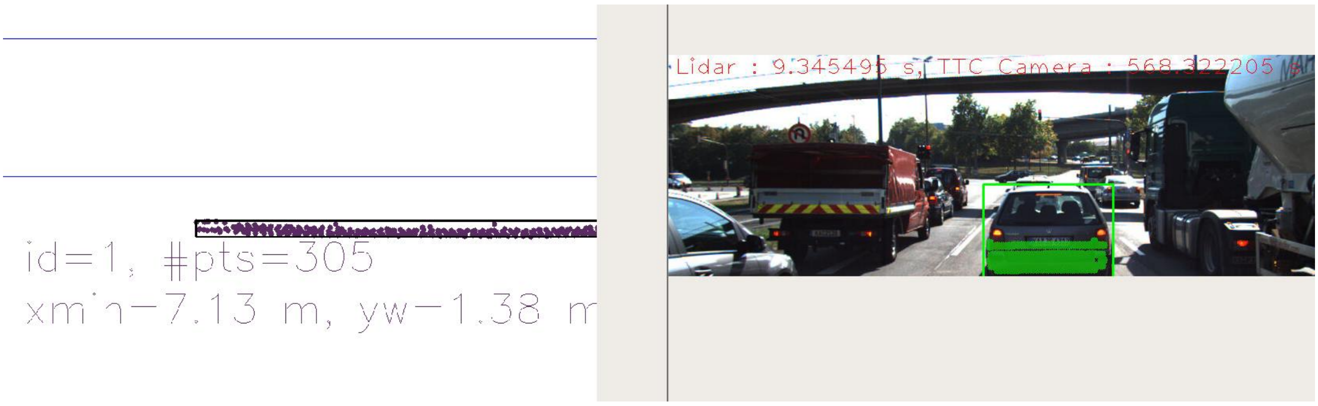


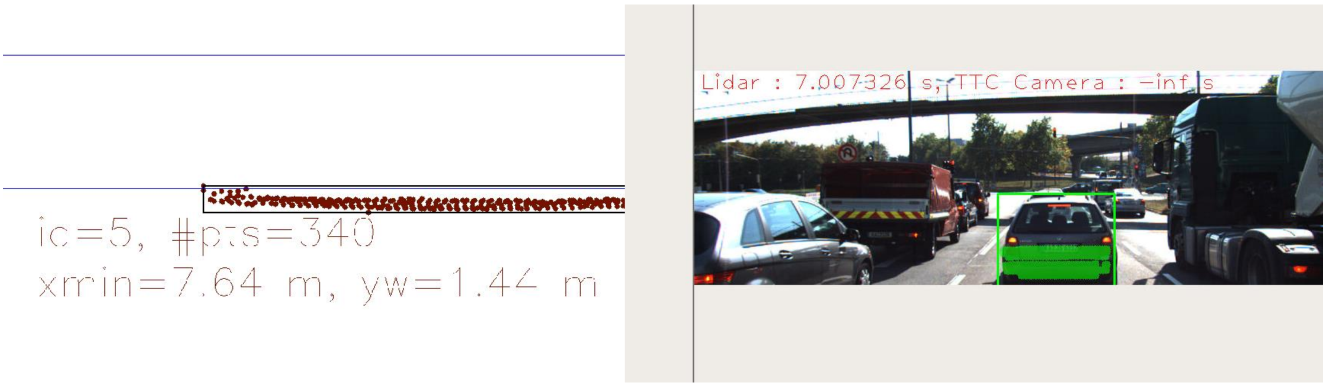
FP.6 Performance Evaluation 2:

The following table shows the TTC estimation performance for all different detector/descriptor combinations. It shows the number of frames where no bounding box match has been identified and the number of frames in which either the camera based or the lidar based TTC estimation was off. There are several examples/frames where the keypoint (i.e. camera) based TTC estimation is way off. Like described above, incorrect keypoint matches might be the main root cause for these inaccuracies. For the 2D camera images which are being used in this project, it's sort of expectable that similar keypoints occur. There are multiple vehicles with similar shape and lights so the description for corresponding keypoints can also be expected to be similar. The pipeline does use euclidean distance based filtering, but not all mismatches might be caught by this. Examples of frames where the camera based TTC estimation is off are shown below.

Descriptor over Detector	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
SHITOMASI	TTC estimation off Lidar: 5 Camera: 0 No match: 3	TTC estimation off Lidar: 4 Camera: 1 No match: 4	TTC estimation off Lidar: 3 Camera: 0 No match: 5	TTC estimation off Lidar: 3 Camera: 0 No match: 3		TTC estimation off Lidar: 3 Camera: 0 No match: 4
HARRIS	TTC estimation off Lidar: 2 Camera: 4 No match: 6	TTC estimation off Lidar: 2 Camera: 4 No match: 4	TTC estimation off Lidar: 4 Camera: 5 No match: 7	TTC estimation off Lidar: 3 Camera: 4 No match: 4		Segmentation Fault
FAST	TTC estimation off Lidar: 2 Camera: 0 No match: 6	TTC estimation off Lidar: 3 Camera: 0 No match: 3	TTC estimation off Lidar: 3 Camera: 0 No match: 3	TTC estimation off Lidar: 3 Camera: 2 No match: 3		TTC estimation off Lidar: 3 Camera: 2 No match: 1
BRISK	TTC estimation off Lidar: 3 Camera: 4 No match: 3	TTC estimation off Lidar: 4 Camera: 5 No match: 4	TTC estimation off Lidar: 3 Camera: 5 No match: 4	TTC estimation off Lidar: 3 Camera: 5 No match: 4		TTC estimation off Lidar: 3 Camera: 1 No match: 2
ORB	TTC estimation off Lidar: 0 Camera: 4 No match: 5	TTC estimation off Lidar: 1 Camera: 8 No match: 4	TTC estimation off Lidar: 2 Camera: 4 No match: 4	TTC estimation off Lidar: 2 Camera: 3 No match: 4		TTC estimation off Lidar: 0 Camera: 4 No match: 4
AKAZE	TTC estimation off Lidar: 4 Camera: 1 No match: 4	TTC estimation off Lidar: 3 Camera: No match: 3	TTC estimation off Lidar: 4 Camera: 0 No match:3	TTC estimation off Lidar: 3 Camera: 0 No match: 4	TTC estimation off Lidar: 4 Camera: 0 No match: 4	TTC estimation off Lidar: 4 Camera: 0 No match: 3
SIFT	TTC estimation off Lidar: 1 Camera: 2 No match: 3	TTC estimation off Lidar: 4 Camera: 2 No match: 4		TTC estimation off Lidar: 1 Camera: 2 No match: 5		TTC estimation off Lidar: 3 Camera: 4 No match: 1


id=5, #pts=340
xmin=7.64 m, yw=1.44 m
TTC Lidar : 7.007326 s, TTC Camera : -11.869329 s


id=1, #pts=305
xmin=7.13 m, yw=1.38 m
Lidar : 9.345495 s, TTC Camera : 568.322205


id=5, #pts=340
xmin=7.64 m, yw=1.44 m
Lidar : 7.007326 s, TTC Camera : -inf s


id=5, #pts=321
xmin=7.79 m, yw=1.47 m
Lidar : 18.155922 s, TTC Camera : -80.852533 s


id=4, #pts=315
xmin=7.55 m, yw=1.44 m
TTC Lidar : 37.800036 s, TTC Camera : -inf s