

Track an Object in 3D Space Report

Source Code

FP.1 Match 3D Objects

Implemented as in line 305-355 in camFusion_student.cpp.

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame
&prevFrame, DataFrame &currFrame)
{
    vector<int> matchedBoxIDs;

    // Loop over all bounding boxes in the current frame.
    for (auto cur_it = currFrame.boundingBoxes.begin(); cur_it != currFrame.boundingBoxes.end(); ++cur_it)
    {
        // Stores match count, boxID -> count.
        map<int, int> bbMatchCount;

        // Loop over all matches.
        for (auto match_it = matches.begin(); match_it != matches.end(); ++match_it)
        {
            // Current frame contains match point
            if (cur_it->roi.contains(currFrame.keypoints[match_it->trainIdx].pt))
            {
                // Loop over bounding boxes in the previous frame
                for (auto prev_it = prevFrame.boundingBoxes.begin(); prev_it != prevFrame.boundingBoxes.end();
++ prev_it)
                {
                    // Update the bbMatchCount if the previous bounding box also contains the keypoint.
                    if (prev_it->roi.contains(prevFrame.keypoints[match_it->queryIdx].pt))
                    {
                        if (bbMatchCount.count(prev_it->boxID) == 0)
                        {
                            bbMatchCount.insert(pair<int, int>(prev_it->boxID, 0));
                        }
                        bbMatchCount.at(prev_it->boxID)++;
                    }
                }
            }
        }

        int bestPrevBoxID;
        int maxMatchCount = 0;
        for (auto count_it = bbMatchCount.begin(); count_it != bbMatchCount.end(); ++count_it)
        {
            if ((count_it->second > maxMatchCount) && (find(matchedBoxIDs.begin(), matchedBoxIDs.end(),
count_it->first) == matchedBoxIDs.end()))
            {
                maxMatchCount = count_it->second;
                bestPrevBoxID = count_it->first;
            }
        }

        if (maxMatchCount > 0)
        {
            matchedBoxIDs.push_back(bestPrevBoxID);
            bbBestMatches.insert(pair<int, int>(bestPrevBoxID, cur_it->boxID));
        }
    }
}
```

FP.2 Compute Lidar-based TTC

Implemented as in line 251-302 in camFusion_student.cpp. In order to avoid outliers in the Lidar data points, I choose to use the Lidar point at 5 percentile based on the x position.

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                    std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    // auxiliary variables
    double dT = 1.0 / frameRate; // time between two measurements in seconds
    double laneWidth = 4.0;      // assumed width of the ego lane

    vector<double> xPrev;

    for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end(); ++it)
    {
        // remove lidar points outside the lane
        if (abs(it->y) <= (laneWidth / 2.0))
        {
            xPrev.push_back(it->x);
        }
    }

    // Since the we want to remove the outlier, we use the 5% percentile of the
    // x position to represent the closest lidar point to the ego car.
    auto five_per_it = xPrev.begin() + xPrev.size() / 20;
    std::nth_element(xPrev.begin(), five_per_it, xPrev.end());
    double five_per_XPrev = *five_per_it;

    vector<double> xCur;

    for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end(); ++it)
    {
        // remove lidar points outside the lane
        if (abs(it->y) <= (laneWidth / 2.0))
        {
            xCur.push_back(it->x);
        }
    }

    // Since the we want to remove the outlier, we use the 5% percentile of the
    // x position to represent the closest lidar point to the ego car.
    five_per_it = xCur.begin() + xCur.size() / 20;
    std::nth_element(xCur.begin(), five_per_it, xCur.end());
    double five_per_XCur = *five_per_it;

    // compute TTC from both measurements
    if (abs(five_per_XPrev - five_per_XCur) > std::numeric_limits<double>::epsilon())
    {
        TTC = five_per_XCur * dT / (five_per_XPrev - five_per_XCur);
    }
    else
    {
        TTC = NAN;
    }
}
```

FP.3 Associate Keypoint Correspondences with Bounding Boxes

Implemented as in line 157-185 in camFusion_student.cpp.

```
// associate a given bounding box with the keypoints it contains
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev,
std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
    std::vector<double> euclideanDist;
    double distThreshold = 2.0;

    // Find the median euclidean distance between kptsPrev and kptsCurr
    for (auto it = kptMatches.begin(); it != kptMatches.end(); ++it)
    {
        if (boundingBox.roi.contains(kptsCurr[it->trainIdx].pt))
        {
            euclideanDist.push_back(cv::norm(kptsPrev[it->queryIdx].pt - kptsCurr[it->trainIdx].pt));
        }
    }
    const auto med_it = euclideanDist.begin() + euclideanDist.size() / 2;
    const auto med_it_1 = euclideanDist.begin() + euclideanDist.size() / 2 - 1;
    std::nth_element(euclideanDist.begin(), med_it, euclideanDist.end());
    std::nth_element(euclideanDist.begin(), med_it_1, euclideanDist.end());
    double medEuclideanDist = euclideanDist.size() % 2 == 0 ? (*med_it + *med_it_1) / 2.0 : *med_it;

    for (auto it = kptMatches.begin(); it != kptMatches.end(); ++it)
    {
        if (boundingBox.roi.contains(kptsCurr[it->trainIdx].pt) &&
            abs(cv::norm(kptsPrev[it->queryIdx].pt - kptsCurr[it->trainIdx].pt) - medEuclideanDist) <
            distThreshold)
        {
            boundingBox.kptMatches.push_back(*it);
        }
    }
}
```

FP.4 Compute Camera-based TTC

Implemented as in line 188-248 in camFusion_student.cpp. In order to avoid outlier correspondences, I choose to calculate the TTC by using the median distance ratio.

```
// Compute time-to-collision (TTC) based on keypoint correspondences in successive images
void computeTTCamera(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 keypoint 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 keypoint 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

// only continue if list of distance ratios is not empty
if (distRatios.size() == 0)
{
    TTC = NAN;
    return;
}

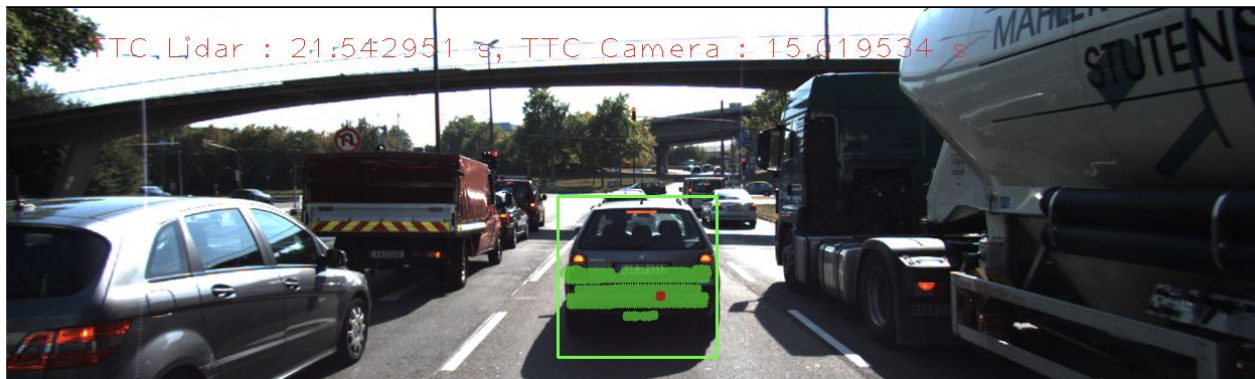
// compute camera-based TTC from distance ratios
double medianDistRatio;
if (distRatios.size() % 2 == 0){
    const auto med_it = distRatios.begin() + distRatios.size() / 2;
    const auto med_it_1 = distRatios.begin() + distRatios.size() / 2 - 1;
    std::nth_element(distRatios.begin(), med_it, distRatios.end());
    std::nth_element(distRatios.begin(), med_it_1, distRatios.end());
    medianDistRatio = (*med_it + *med_it_1) / 2.0;
} else {
    const auto med_it = distRatios.begin() + distRatios.size() / 2;
    std::nth_element(distRatios.begin(), med_it, distRatios.end());
    medianDistRatio = *med_it;
}



double dT = 1 / frameRate;
TTC = -dT / (1 - medianDistRatio);
}

```

FP.5 Performance Evaluation 1

Following figures show one example of faulty Lidar based TTC. The red dots represent the lidar point used to calculate the TTC. Here two consecutive frames are shown as below. I noticed that the Lidar point used to calculate the Lidar TTC is shifted from the car plate in the first image to the bumper in the second image. We anticipate that on the x axis direct, there is displacement between these two lidar points. As the frame rate is 10Hz and the relative speed between ego car and the car in front of the ego car is small, such displacement can potentially generate large errors in the TTC calculation.



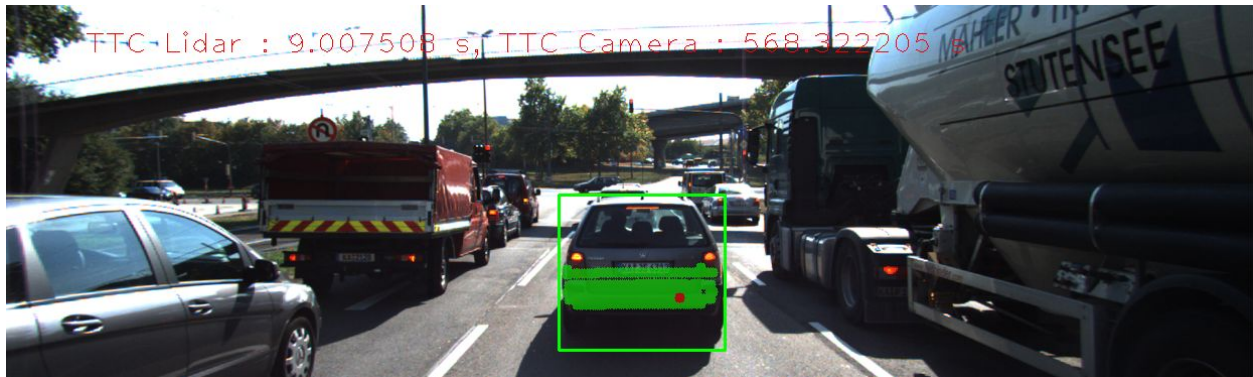
 id=6, #pts=345 xmin=7.58 m, yw=1.43 m 5 percentile x=7.59 m	 id=5, #pts=307 xmin=7.47 m, yw=1.45 m 5 percentile x=7.54 m

FP.6 Performance Evaluation 1

All detailed detector / descriptor combinations are included in [1].

Following image shows the result when using the Harris detector and BRISK descriptor. The camera TTC result for the given frame does not seem plausible. The reason for such an outcome is that the Harris detector only generates a limited number of key points. In the given

frame, only very few key points are captured by the region of interest. Hence, the median operation to remove outlier correspondences is not effective in this case.



[1]

<https://docs.google.com/spreadsheets/d/1gTqV6XOL7QyWRab3OHXpJr7Qki8uDIZQeaVOse2tXNs/edit?usp=sharing>