SFND 3D Object Tracking

FP.1 Match 3D Objects

Implement the method "matchBoundingBoxes", which takes as input both the previous and the current data frames and provides as output the ids of the matched regions of interest (i.e. the boxID property). Matches must be the ones with the highest number of keypoint correspondences.

Implementation:

I have implemented "matchBoundingBoxes" function in camFusion_Student.cpp program from line 277 to 336 as given below.

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
    map<int, map<int, int>> counts;
    for (cv::DMatch match : matches)
    {
        cv::KeyPoint query = prevFrame.keypoints[match.queryIdx];
        bool query_found = false;
        int query_pos, train_pos;
        cv::KeyPoint train = currFrame.keypoints[match.trainIdx];
        bool train_found = false;
        for (int i = 0; i<prevFrame.boundingBoxes.size(); i++)
        {
            if (prevFrame.boundingBoxes[i].roi.contains(cv::Point(query.pt.x, query.pt.y)))
            {
                query_found = true;
                query_pos = i;
                break;
            }
        }
        for (int j=0; j<currFrame.boundingBoxes.size(); j++)
        {
            if (currFrame.boundingBoxes[j].roi.contains(cv::Point(train.pt.x, train.pt.y)))
            {
                train_found = true;
                train_pos = j;
                break;
            }
        }

        if (query_found && train_found)
        {
            if (counts.count(query_pos)==1)
```

```
            {
                counts[query_pos][train_pos] += 1;
            }
            else
            {
                counts[query_pos][train_pos] = 1;
            }
        }
    }
    for (auto c : counts)
    {
        int max = 0;
        int value = -1;
        for (auto m : c.second)
        {
            if (m.second > max)
            {
                max = m.second;
                value = m.first;
            }
        }
        if (value != -1)
        {
            bbBestMatches[c.first] = value;
        }
    }
}
```

FP.2 Compute Lidar-based TTC

Compute the time-to-collision in second for all matched 3D objects using only Lidar measurements from the matched bounding boxes between current and previous frame.

Implementation:

I have implemented " computeTTCLidar" function in camFusion_Student.cpp program from line 222 to 274 as given below. Have found median x value from both current and previous points and used them to calculate TTC.

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    double dT = 1/frameRate;
    double lanewidth = 4.0;
    vector<double> xPrev, xCurr;

    for (auto it = lidarPointsCurr.begin(); it!=lidarPointsCurr.end(); ++it)
```

```
    {
        if (abs(it->y) <= lanewidth/2.0)
        {
            xCurr.push_back(it->x);
        }
    }

    for (auto it = lidarPointsPrev.begin(); it!=lidarPointsPrev.end(); ++it)
    {
        if (abs(it->y) <= lanewidth/2.0)
        {
            xPrev.push_back(it->x);
        }
    }

    double minxPrev, minxCurr;

    sort(xPrev.begin(), xPrev.end());

    int size_xPrev = xPrev.size();
    if(size_xPrev%2 == 0)
    {
        minxPrev = (xPrev[size_xPrev/2]+xPrev[(size_xPrev-1)/2])/2;
    }
    else
    {
        minxPrev = xPrev[size_xPrev/2];
    }

    sort(xCurr.begin(), xCurr.end());

    int size_xCurr = xCurr.size();
    if(size_xCurr%2 == 0)
    {
        minxCurr = (xCurr[size_xCurr/2]+xCurr[(size_xCurr-1)/2])/2;
    }
    else
    {
        minxCurr = xCurr[size_xCurr/2];
    }

    TTC = minxCurr * dT / (minxPrev - minxCurr);

    cout << " TTC computed from Lidar points : " << TTC << endl;
}
```

FP.3 Associate Keypoint Correspondences with Bounding Boxes

Prepare the TTC computation based on camera measurements by associating keypoint correspondences to the bounding boxes which enclose them. All matches which satisfy this condition must be added to a vector in the respective bounding box.

Implementation:

I have implemented “clusterKptMatchesWithROI" function in camFusion_Student.cpp program from line 135 to 163 as given below. Mean distance has been calculated for the keypoints within the bounding boxe and a threshold of mean distance multiplied by 0.7 have been used to discard points having more distance than the thresholded mean distance.

```
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
   double distance_total = 0;
   std::vector<cv::DMatch> kptMatches_roi;
   for (auto it=kptMatches.begin(); it!=kptMatches.end(); ++it)
   {
      cv::KeyPoint kpt = kptsCurr.at(it->trainIdx);
      if(boundingBox.roi.contains(cv::Point(kpt.pt.x, kpt.pt.y)))
      {
         kptMatches_roi.push_back(*it);
      }
   }
   for (auto it=kptMatches_roi.begin(); it!=kptMatches_roi.end(); ++it)
   {
      distance_total += it->distance;
   }
   double distance_mean = distance_total/kptMatches_roi.size();
   // discard points beyond threshold
   double threshold = distance_mean * 0.7;
   for (auto it = kptMatches_roi.begin(); it != kptMatches_roi.end(); ++it)
   {
      if (it->distance < threshold)
      {
         boundingBox.kptMatches.push_back(*it);
      }
     // cout <<"boundingBox kpt Matches : " << boundingBox.kptMatches.size() << endl;
   }
}
```

FP.4 Compute Camera-based TTC

Compute the time-to-collision in second for all matched 3D objects using only keypoint correspondences from the matched bounding boxes between current and previous frame.

Implementation:

I have implemented "computeTTCCamera" function in camFusion_Student.cpp program from line 167 to 219 as given below.

```
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

    // 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 meanDistRatio = std::accumulate(distRatios.begin(), distRatios.end(), 0.0) / distRatios.size();
    std::sort(distRatios.begin(), distRatios.end());
    long medIndex = floor(distRatios.size()/2.0);

    double medianDistRatio = distRatios.size() % 2 == 0 ? (distRatios[medIndex - 1] +
distRatios[medIndex]) / 2.0 : distRatios[medIndex]; // compute median dist. ratio to remove outlier
influence


    double dT = 1 / frameRate;
    TTC = -dT / (1 - medianDistRatio);
    cout << " TTC from Camera  : " << TTC << endl;
}
```

FP.5 Performance Evaluation 1

I observed that the TTC calculated using Lidar for some of the frames looks not correct, in some frames (4 & 5) even though the distance reduced from earlier frame, TTC got increased slightly (which is not correct). I feel this was because of having some outlier points. I have used median approach in TTC computation process. Lidar TTC estimations can be found in attached Task 6 - Results.xlsx.

FP.6 Performance Evaluation 2

All detector / descriptor combinations were used and have been compared with regard to the Camera TTC estimate on a frame-by-frame basis and results can be found in Task 6 - Results.xlsx.
I observed that the TTC calculated using camera for some of the frames looks not correct, in some frames even though the distance reduced from earlier frame, TTC got increased slightly (which is not correct). I have used median approach in TTC computation process.
TTC estimate done using Harris & ORB detector in combinations with different descriptor types are not reliable. Based on my observation and data comparison, below are my Top 3 detector/descriptor combinations.

1.AKAZE/BRIEF
2.AKAZE/BRISK
3.SHITOMASI/BRIEF