3D Feature Tracking

Final-Term Report

P.1 Match 3D Objects

Requirements

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.

Solution

Please refer to the file src/camFusion_Student.cpp. The matchBoudingBoxes was implemented by looping over all the matches respectively. the corresponding row-column value would be incremented if the keypoints linked to a match are into the subset of our current and previous bounding boxes.

Code Snippet

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int,
int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
   int previous = prevFrame.boundingBoxes.size();
   int current = currFrame.boundingBoxes.size();
   //  Declare array compromising of previous rows, current columns
   int point_count[previous][current];

   for (auto it = matches.begin(); it != matches.end() - 1; ++it)
   {
       cv::KeyPoint query = prevFrame.keypoints[it->queryIdx];
       auto query_pt = cv::Point(query.pt.x, query.pt.y);
       bool query_found = false;

       cv::KeyPoint train = currFrame.keypoints[it->trainIdx];
       auto train_pt = cv::Point(train.pt.x, train.pt.y);

       bool train_found = false;
```

```
        std::vector<int> query_id, train_id;
        for (int i = 0; i < previous; i++)
        {
            if (prevFrame.boundingBoxes[i].roi.contains(query_pt))
             {
                query_found = true;
                query_id.push_back(i);
             }
        }
        for (int i = 0; i < current; i++)
        {
            if (currFrame.boundingBoxes[i].roi.contains(train_pt))
            {
                train_found= true;
                train_id.push_back(i);
            }
        }

        if (query_found && train_found)
        {
            for (auto id_prev: query_id)
                for (auto id_curr: train_id)
                     point_count[id_prev][id_curr] += 1;
        }
    }

    for (int i = 0; i < previous; i++)
    {
         int max_count = 0;
         int id_max = 0;
         for (int j = 0; j < current; j++)
             if (point_count[i][j] > max_count)
             {
                  max_count = point_count[i][j];
                  id_max = j;
             }
          bbBestMatches[i] = id_max;    //Equalize
```

```
    }
};
```

P.2 Compute Lidar-based TTC Detection

Requirements

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.

Solution

For computing the time-to-collision (TTC),

1. Remove outliers first, This will help in avoiding wrong measurements.
2. In the second turn, we remove all the points that lies outside our ego lane.
3. We then get the mean distance to get stable output.

Code Snippet

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double
frameRate, double &TTC)
{
    float lane_wide = 4.2;  //lane_width
    //Here, we will only take into account the ego car lane

    std::vector<float> ppx;
    std::vector<float> pcx;

    for(auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end() -1;
++it)
        if(std::abs(it->y) < lane_wide/2) ppx.push_back(it->x);

    for(auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end() -1;
++it)
        if(std::abs(it->y) < lane_wide/2) pcx.push_back(it->x);

    float min_px, min_cx;
    int p_size = ppx.size();
```

```
    int c_size = pcx.size();
    if(p_size > 0 && c_size > 0)
    {
        for(int i=0; i<p_size; i++)
            min_px += ppx[i];

        for(int j=0; j<c_size; j++)
            min_cx += pcx[j];
    }
    else
    {
        TTC = NAN;
        return;
    }

    min_px = min_px /p_size;
    std::cout<<"Lidar_Min_Px:"<<min_px<<std::endl;

    min_cx = min_cx /c_size;
    std::cout<<"Lidar_Min_Cx:"<<min_cx<<std::endl;

    //  Calculate TTC
    float dt = 1/frameRate;
    TTC = min_cx * dt / (min_px - min_cx);
};
```

P.3 Associate Keypoint Correspondences with Bounding Boxes

Requirements

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.

Solution

To associate matches with corresponding bounding boxes, I used the function clusterKptMatchesWithROI which takes the current bounding box as its respective input. A

relative match is pushed into the kptsMatches vector if the respective keypoints are found inside the current bounding box.

Code Snippet

```
void clusterKptMatchesWithROI(BoundingBox &boundingBox,
std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
std::vector<cv::DMatch> &kptMatches)
{
   double dist_mean = 0;
   std::vector<cv::DMatch>  kptMatches_roi;

   for (auto it = kptMatches.begin(); it != kptMatches.end(); ++it)
   {
       cv::KeyPoint kp = kptsCurr.at(it->trainIdx);
       if (boundingBox.roi.contains(cv::Point(kp.pt.x, kp.pt.y)))
           kptMatches_roi.push_back(*it);
   }

    for (auto it = kptMatches_roi.begin(); it != kptMatches_roi.end();
++it)
        dist_mean += it->distance;

   if (kptMatches_roi.size() > 0)
        dist_mean = dist_mean/kptMatches_roi.size();
   else
       return;

   double threshold = dist_mean * 1.6;
    for (auto it = kptMatches_roi.begin(); it != kptMatches_roi.end();
++it)
   {
      if (it->distance < threshold)
          boundingBox.kptMatches.push_back(*it);
   }
};
```

P.4 Compute Camera-based TTC

Requirements

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.

Solution

For computing camera based TTC, we first implement outlier filtering. This is done via looping over all the distances between keypoints that belongs to the respective bounding box. We then compute the distance ratios between them and later calculating median distance ratio which is used for TTC formula.

Code Snippet

```
void computeTTCCamera(std::vector<cv::KeyPoint> &kptsPrev,
std::vector<cv::KeyPoint> &kptsCurr,
                      std::vector<cv::DMatch> kptMatches, double frameRate,
double &TTC, cv::Mat *visImg)
{
   // Store distance ratios
   vector<double> dist_ratios;

   for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
   {
       cv::KeyPoint kpOuterCurr = kptsCurr.at(it1->trainIdx);
       cv::KeyPoint kpOuterPrev = kptsPrev.at(it1->queryIdx);

      // Second Iterator Loop
       for (auto it2 = it1 + 1; it2 != kptMatches.end() - 1; ++it2)
       {
           // Minimum required distance
           double minDist = 100.0;

           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)
            {
                //Check division by zero error.
                double distRatio = distCurr / distPrev;
                dist_ratios.push_back(distRatio);
            }
        }
    }
    // Check only if distance ratios !=0
    if (dist_ratios.size() == 0)
    {
        TTC = NAN;
        return;
    }

    // Sort distance ratios
    std::sort(dist_ratios.begin(), dist_ratios.end());

    long medIndex = floor(dist_ratios.size() / 2.0);

     // We will now compute medium distance ratio for removing outliers
influence
         double medDistRatio = dist_ratios.size() % 2 == 0 ?
(dist_ratios[medIndex - 1] + dist_ratios[medIndex]) / 2.0 :
dist_ratios[medIndex];

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

P.5 Performance Evaluation 1

Requirements

Find examples where the TTC estimate of the Lidar sensor does not seem plausible. Describe your observations and provide a sound argumentation why you think this happened.

Solution

We noticed that TTC from Lidar is somewhere giving not correct results, this is mainly due to outliers. And several points due to preceding vehicle's mirrors which actually need to filter out.

P.6 Performance Evaluation 2

Requirements

Run several detector / descriptor combinations and look at the differences in TTC estimation. Find out which methods perform best and also include several examples where camera-based TTC estimation is way off. As with Lidar, describe your observations again and also look into potential reasons.

Solution

To test all the possible combinations of detectors and descriptors and for that I iterated a loop and saved results of all possible combinations in .csv format and saved the resulting images in ./output folder.

The standing 3 detector / descriptor for our task of detecting keypoints on vehicles are found out to be,

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

Why do some detectors/descriptors provide not good TTCs?

One possible explanation relates to the fact that mono camera setups do not have a depth sensing. Though it is not much possible to separate keypoint planes, but what we can do is simply remove outliers using a robust euclidean mean computation. For example the combination ORB-FREAK provided results that were really unreliable.
