Master's Project: Deep Learning and Autonomous Racing

Sommersemester 2021

By *Raphael Schwinger, Rakibuzzaman Mahmud*

Supervisors : *Lars Schmarje, Claudius Anton Zelenka*

Project Overview

Autonomous racing is an emerging field within autonomous driving. A few self-racing vehicles have been developed in the last years, both in industrial and academic research. The first known autonomous vehicle competition [1] was the DARPA Grand Challenge.

Formula Student Germany organized the first autonomous racing competition in 2017, followed by other countries in 2018. Formula Student (FS) is an international engineering competition where multidisciplinary student teams compete with a self-developed racecar every year.

The main race consists of completing ten laps, as fast as possible, around an unknown track defined by small 228 × 335 mm cones. Blue and yellow cones distinguish the left and the right boundary, respectively. The track is a 500m long closed circuit, with a width of 3m, and the cones in the same boundary can be up to 5m apart. The track contains straights, hairpins, chicanes, multiple turns, and decreasing radius turns.[2]

The master project "Ground Truth Generation" is a part of the "Rosyard" project to implement a self-driving car for the Formula Student Competition.[3]

In order to make the vehicle race fully autonomous, the car uses two modes of operation, Simultaneous Localization and Mapping (SLAM) mode, and Localization mode. [4]

- In SLAM mode, the vehicle path has to be computed using the local perception sensors, and a map of the track has to be generated.
- In Localization mode, the goal is to race as fast as possible in an already mapped track.

As previously stated, the track is marked with cones, and only cones are considered landmarks, and other potential features are rejected. Cameras detect Cones that mark the racetrack to create a reconstruction of the racetrack.

The objective of our project is to design an algorithm that calculates the corresponding ground truth of the racing environment. The SLAM algorithm will use the results of this master project to optimize the car by getting an accurate ground truth of the track and the car's position during a test race.

This task of ground truth generation for the SLAM algorithm is divided into two subtasks.

- First, a ground truth of the race track has to be generated.

- Second, the position of the car has to be recorded during a race.

In order to define the ground truth of the race track, it is therefore sufficient to determine the positions of these cones. We will use 3D scene reconstruction using images/videos of the race track.

Furthermore, To determine the ground truth for the racecar, we need a video recording of the car racing around the racetrack, making at least one lap. Then we can take every frame from the video and reconstruct the car's position for each frame.

Once the ground truth is generated, the car can drive in Localization Mode, exploiting the advantage of planning on the mapped racetrack.

Possible methods:

During the project's planning phase, we discussed many ideas that could solve the racecar tracking issue.

- **LiDAR** : LiDAR is the most accurate compared to the other devices, but it is also the most expensive one. Since the budget was one of our limitations, we discarded the idea.
- **GPS** : Commercially available GPSs are highly accurate, but they are also expensive to get.
- **UWB based Triangulation** : After the release of apple AirTag, we were motivated to discuss the possibility of UWB technology. We discussed using UWB to triangulate the car's position, but we do not have enough technical knowledge and expertise with UWB to implement the ideas.
- **Image-based 3D Reconstruction** : We can use multiple cameras to record the racecar racing around the racetrack and get the position of cones and the car. After that, we can make a 3D scene reconstruction scene using the data we collected.

3D Scene Reconstruction:

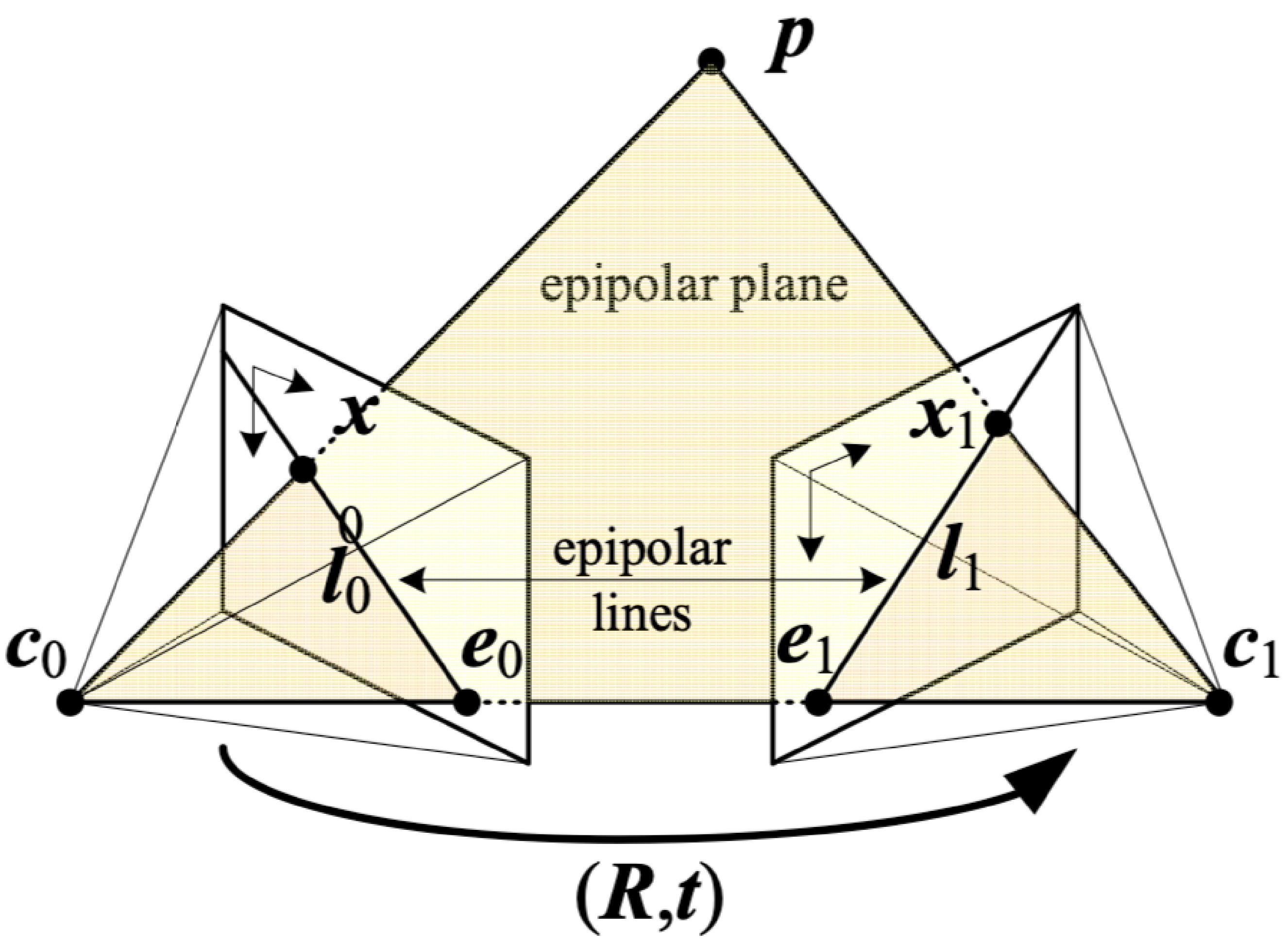


Fig: Camera matrix

To reconstruct the position of both cones and the racecar we used a approach called `Structure from Motion`, thereby we were able to simultaneously recover the 3D structure of the racetrack and the poses of the used cameras. As an input only the image coordinates of the cones and the racecar and the camera intrinsics need to be provided. The later consists in particular of the set focal length and the set resolution.

First the first camera is set as the origin. The task is now to acquire the pose and position of the consecutive cameras to be able to triangulate the position of the cones and racecar. For the the second camera this can be reconstructed with the `essential matrix` $E = [t]_x R$. OpenCV provides the following functions to do this.

```
E = cv2.findEssentialMat(points_2D_1, points_2D_2, cameraMatrix )
R, t = cv2.recoverPose(E, points_2D_1, points2D_2, cameraMatrix)
```

Afterwards the 3D coordinates of the cones and the racecar can be triangulated.

```
points3D = cv2.triangulatePoints(pose_1, pose_2, points1, points2)
```

For consecutive cameras R and t can be recovered with Random sample consensus RANSAC algorithm and the `Rodrigues algorithm`.

```
rvecs, t = cv2.solvePnPRansac(points_3D, points_2D, cameraMatrix)
R = cv2.Rodrigues(rvecs)
```

These informations is needed to further improve the 3D points with bundle adjustment. For this purpose we included the `g2o` library. This results in a list of 3D coordinates of the cones and the position of the racecar in the first frame of the video. To reconstruct the position of the racecar while in motion we repeated the steps for every frame of the video. Figure XX shows our initial visualisation of the result for the first frame.

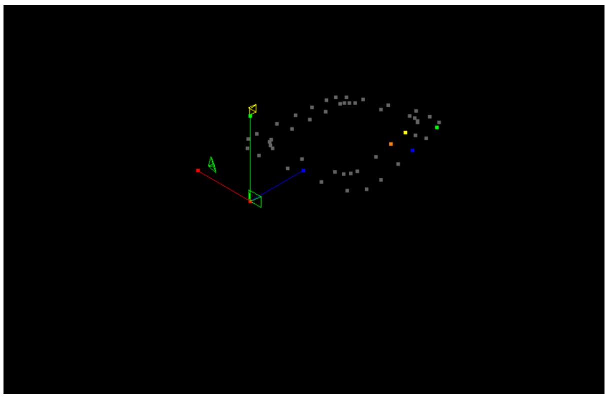


Fig: Blender Reconstruction

Affine transformation

At first glance the result looks like the real racetrack. If we take a closer look at the 3D coordinates of the reconstructed cone points we see that the points are not in the position as expected. That is because the 3D points are in relation to the first camera that is set as the origin of the coordinate system. That means the points need to be translated, rotated and in particular scaled to match the "real world". Transforming the points in all three ways is called a `Affine transformation` and can be applied by multiplying every 3D point with a `affine transformation matrix`. This matrix can be estimated with the knowledge of a correct mapping of 4 points, therefore it is necessary to measure the position from at least 4 cones. Thereby it is important that all 4 cones are not on the same plane to be able to transform 3D points that do not lie on this plane.

```
-0.778266302285012 0.2502844001475607 2.6402778721299835
-0.777195115872759 0.24985856474532567 2.626459134354914
-0.7797482837759697 0.24871681814087102 2.610038240730838
-0.7793411462980047 0.2482823892723588 2.5890051058983867
```

```
    mat = cv2.estimateAffine3D(points_3D[:4], known_points_3d)
    #  [[ 10.19   45.79   -1.93   3.93]
    #   [ 0.26   -100.2   13.86 -18.66]
    #   [ 0.00    0.00    0.00    0.15]]
```

```
-1.978 2.243 0.15
-1.976 2.131 0.15
```

```
-1.913 1.772 0.15
-1.956 1.676 0.15
```

Reconstruction of the race-track using Blender :

To make the reconstruction, we need the video file of the racecar racing around the racetrack. Then we can take the car's 2D position for each frame. However, because of Covid-19, it was not possible to make a real-world racing scenario and record a racing video of the car. So we had to improvise and work on a simulated racing environment which we created on Blender.

- Blender Environmental Elements:
  - Camera:
    - Camera Settings: Focal length 15 mm
    - Camera height: 1.5 m
    - Calibration: We used a python script for camera calibration.
    - Video: The video we exported has a resolution of 4k.
  - Car: We used a realistic Rosyard car model.
    - Height:
    - Width:
    - Length:

Fig: Blender Car Model

- Cones: We used Yellow and Blue cones with a height of 150 CM,

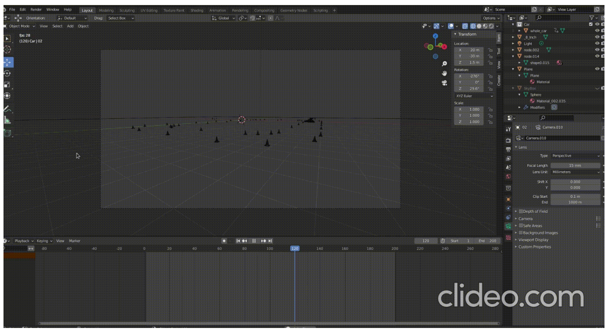


Fig: Racetrack From Camera 1

- Skydome: To make the environment look like a real-world scenario, we used skydome to make it look like a sky on the horizon.

- Asphalt floor: We added asphalt texture on the ground to make the racetrack look like a real racetrack

- Scene Construction : We made a circular racetrack with a length of XX M and XX number of cones.

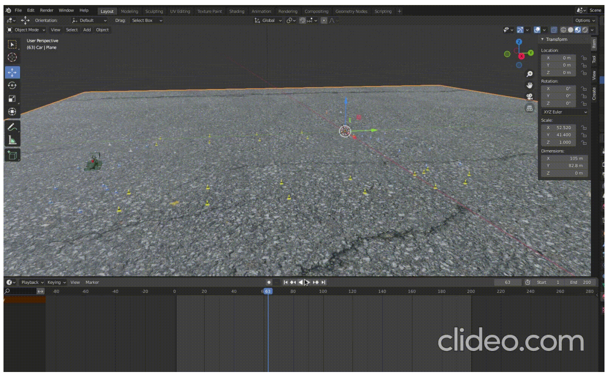


Fig: Circular Racetrack

- Blender Scripts : We can use Python scripts in Blender to get cones and race-car's position points from each camera.
    - Script for getting 2D positions of the cones:

```
for cone in coneCollection.objects:
        # get 3d coordinates of cone
        location = cone.location.copy()
        # location is the bottom of a cone and not the tip, the cone is 
25cm high
        location[2] = cone.location[2] + 0.25
        co_2d = bpy_extras.object_utils.world_to_camera_view(scene, 
camera, location)
        # If you want pixel coords
        render_scale = scene.render.resolution_percentage / 100
        render_size = (
                int(scene.render.resolution_x * render_scale),
                int(scene.render.resolution_y * render_scale),
                    )
        with open(camera.name + '.p2d', 'a') as f:
            print(f'{co_2d.x * render_size[0]} {res_y - co_2d.y * 
render_size[1]}', file=f)
```

- Script for getting 2D position of the Racecar:

```
for camera in cameraCollection.objects:
     bpy.context.scene.camera = camera
     bpy.context.scene.render.filepath = './frames/' + str(f).zfill(4) + 
'/' + camera.name
     bpy.context.scene.render.image_settings.file_format='PNG'
     bpy.ops.render.render(use_viewport=False, write_still=False)

     # erase content of .p2d file
     open(os.path.join(current_frame_path, camera.name +  '.p2d'), 
'w').close()

     # iterate through cones
     car = bpy.data.objects['whole_car']
     # get 3d coordinates of cone
     location = car.location.copy()
     # location is the bottom of a cone and not the tip, the cone is 25cm 
high
     location[2] = car.location[2]
     co_2d = bpy_extras.object_utils.world_to_camera_view(scene, camera, 
location)
     # If you want pixel coords
     render_scale = scene.render.resolution_percentage / 100
     render_size = (
             int(scene.render.resolution_x * render_scale),
             int(scene.render.resolution_y * render_scale),
                 )
```

```
        with open(os.path.join(current_frame_path, camera.name +  '.p2d'),
'a') as file:
            print(f'{co_2d.x * render_size[0]} {res_y - co_2d.y *
render_size[1]}', file=file)
```

More about scripts and how to implement them can be found in the Readme file.

Tracking the racecar

After we rendered the blender scene and got the video file, we applied the OpenCV object tracking algorithm to the video file. We tried multiple tracking algorithms to track the racecar, but only one of them showed promising results. Details and background of the algorithms are included below:

- **OpenCV Tracking Algorithm** :
    - **KCF** [5]:
        - Kernelized Correlation Filter is a novel tracking framework
        - One of the recent findings which has shown good results.
        - Based on the idea of traditional correlational filter.
        - It uses kernel trick and circulant matrices to improve the computation speed significantly.
- **CSRT** [6]:
    - Channel and Spatial Reliability Tracking is a constrained filter learning with an arbitrary spatial reliability map.
    - CSRT utilizes a spatial reliability map.
    - Adjusts the filter support to the part of the object suitable for tracking.
- **GOTRUN** [7]:
    - Generic Object Tracking Using Regression Networks.
    - A Deep Learning based tracking algorithm.
    - Did not perform well.

```
 tracker_types = ['KCF', 'CSRT']
    tracker_type = tracker_types[1]

    if tracker_type == 'KCF':
        tracker = cv2.TrackerKCF_create()
    elif tracker_type == "CSRT":
        tracker = cv2.TrackerCSRT_create()
```

Output of the bounding Box Area:

```
p1 = (int(bbox[0]), int(bbox[1]))
p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
            print(p1,p2)
```

Saving the points for each frame:

We have to save the center point of the bounding box, So we divide the bounding box by 2.

```
with open(os.path.join(current_frame_path, cam_name +  '.p2d'), 'a') as f:
                print(f'{(p1[0] + p2[0]) / 2 } {(p1[1] + p2[1]) / 2 }',
file=f)
```

Color Tracking

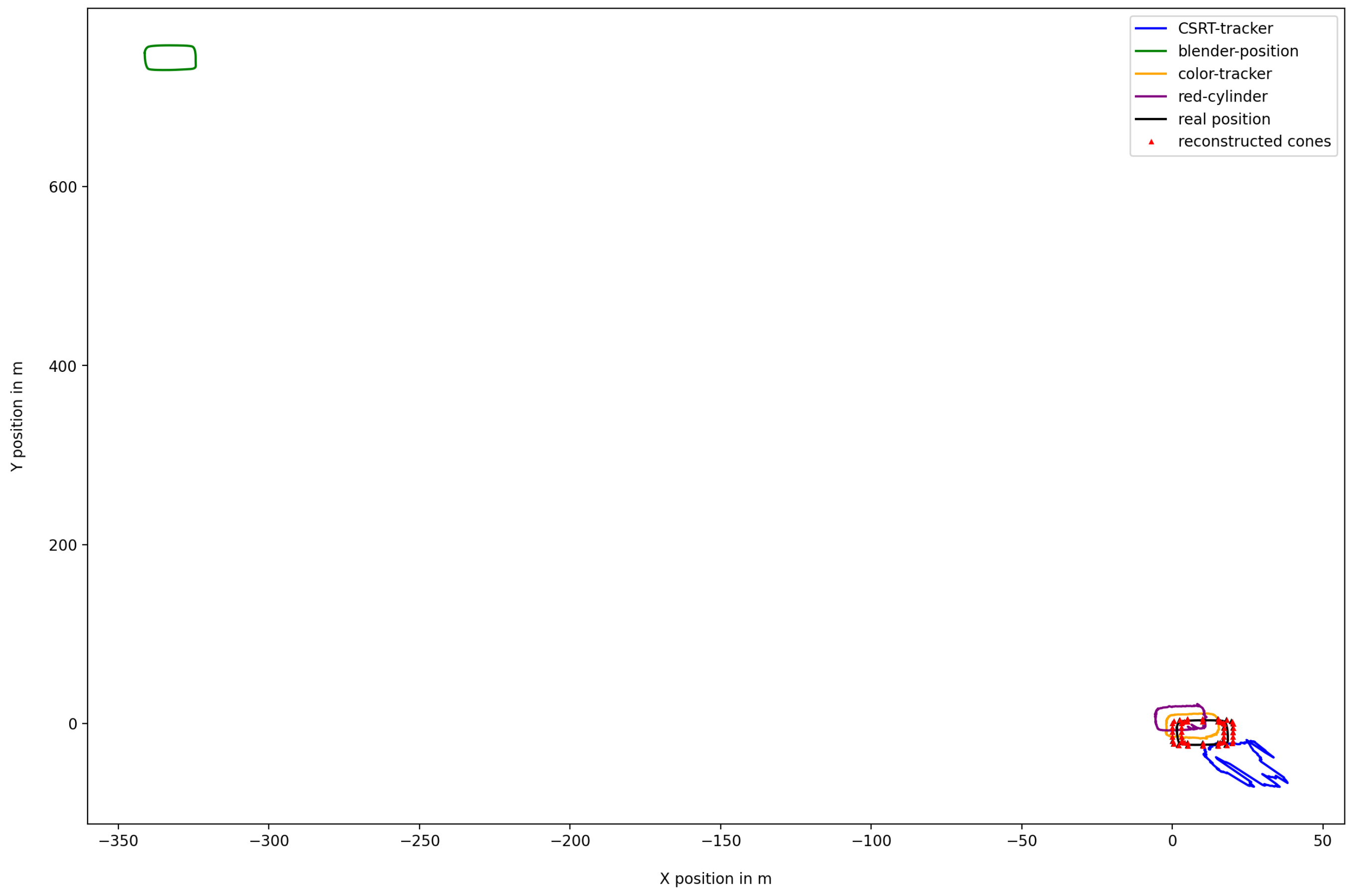
- To try another approach for tracking, we can track an object based on color—for example, Red Car, Red Colored Cylinder. Consequently, instead of tracking the whole car, we added a red-colored cylinder on top of the racecar to make the tracking even more accurate.

```


    # defining the range of red color
    # lower boundary RED color range values; Hue (0 - 10)
    lower1 = np.array([0, 50, 30])
    upper1 = np.array([5, 255, 255])

    # upper boundary RED color range values; Hue (160 - 180)
    lower2 = np.array([180,50,30])
    upper2 = np.array([180,255,255])
```

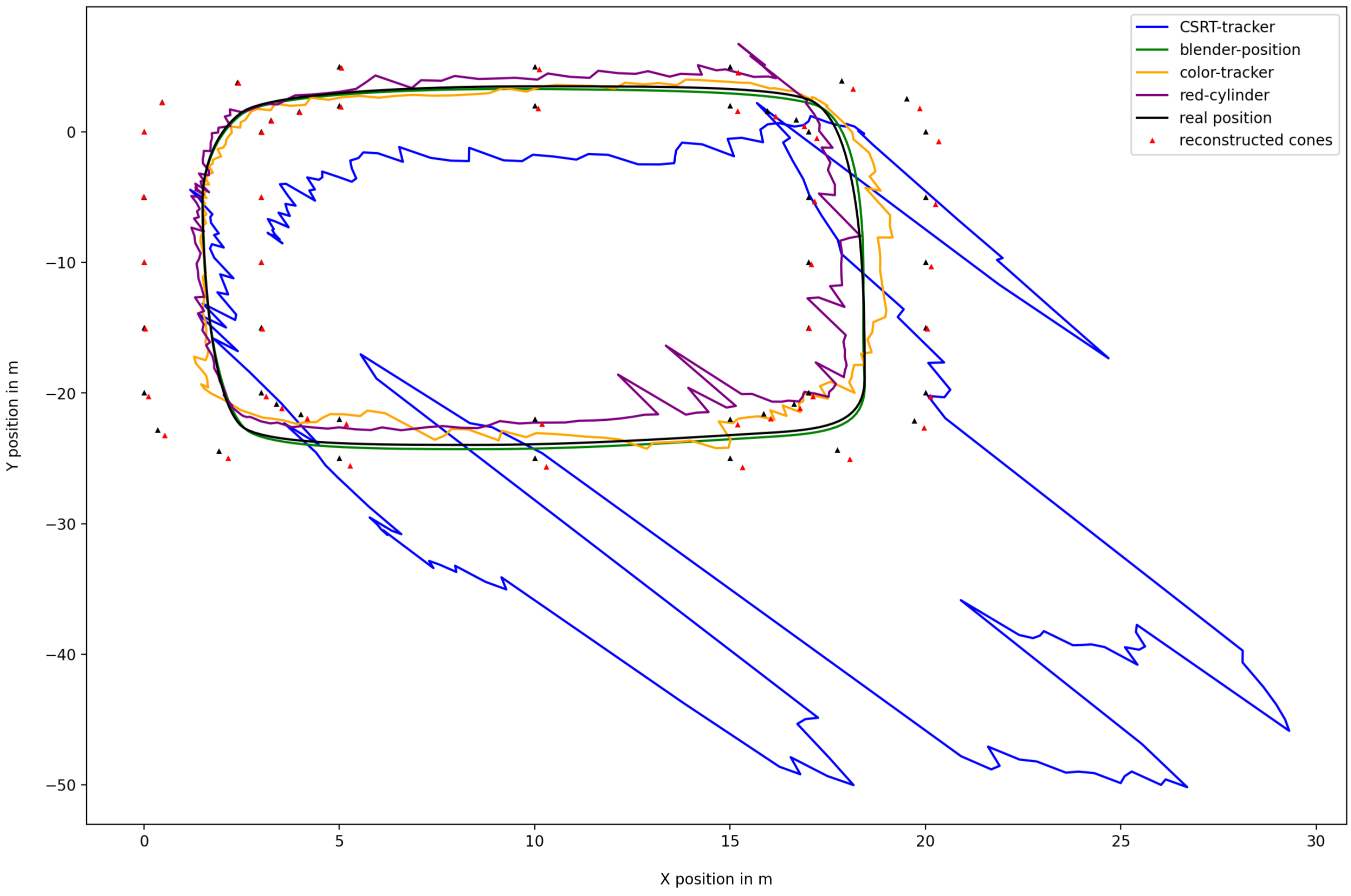
Tracking and saving the 2D points:

```

        # draw rectangle
        img = cv2.rectangle(frame, (min_x, min_y), (max_x, max_y), (0, 0,
255), 2)
        cv2.putText(frame, "RED color", (x, y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255))
        # save result
        with open(os.path.join(current_frame_path, cam_name + '.p2d'),
'a') as f:
            # use bottom of reactangle as center
            print(f'{result_x} {result_y}', file=f)
        with open(os.path.join(path, 'tracking-result-'+  cam_name +
'.p2d'), 'a') as f:
            print(f'{result_x}; {result_y}', file=f)
            cv2.rectangle(frame, [min_x, min_y], [max_x, max_y], (255, 0, 0),
2, 1)
```

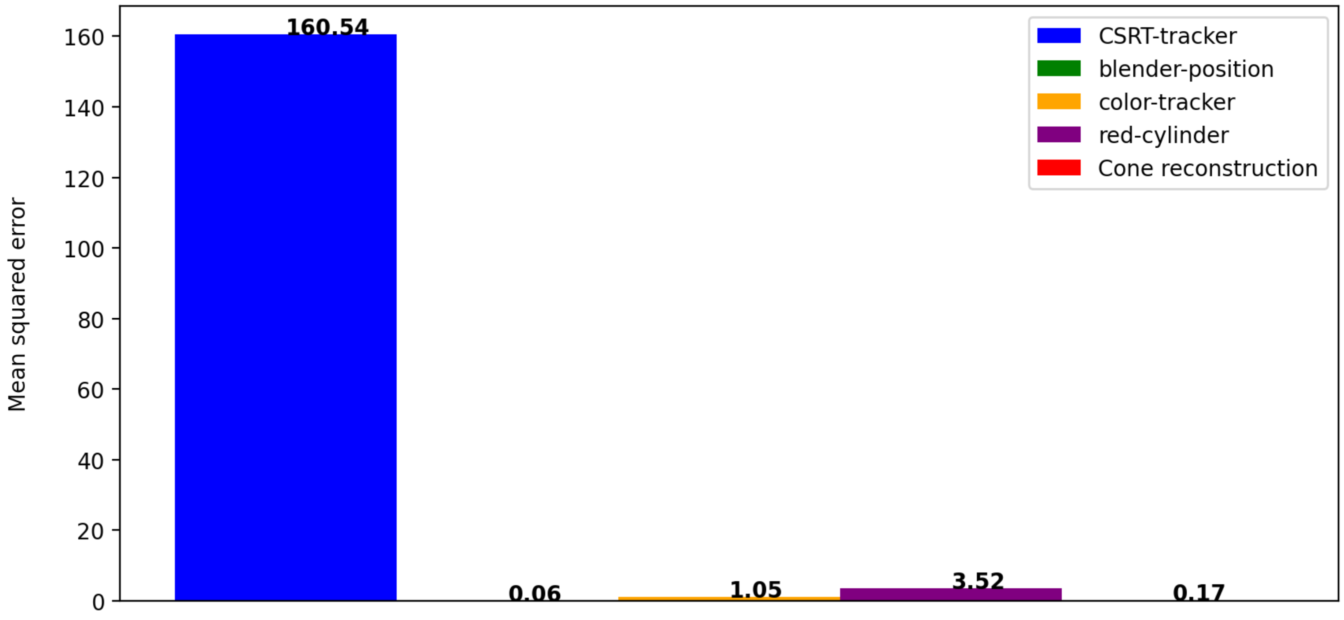
Results:



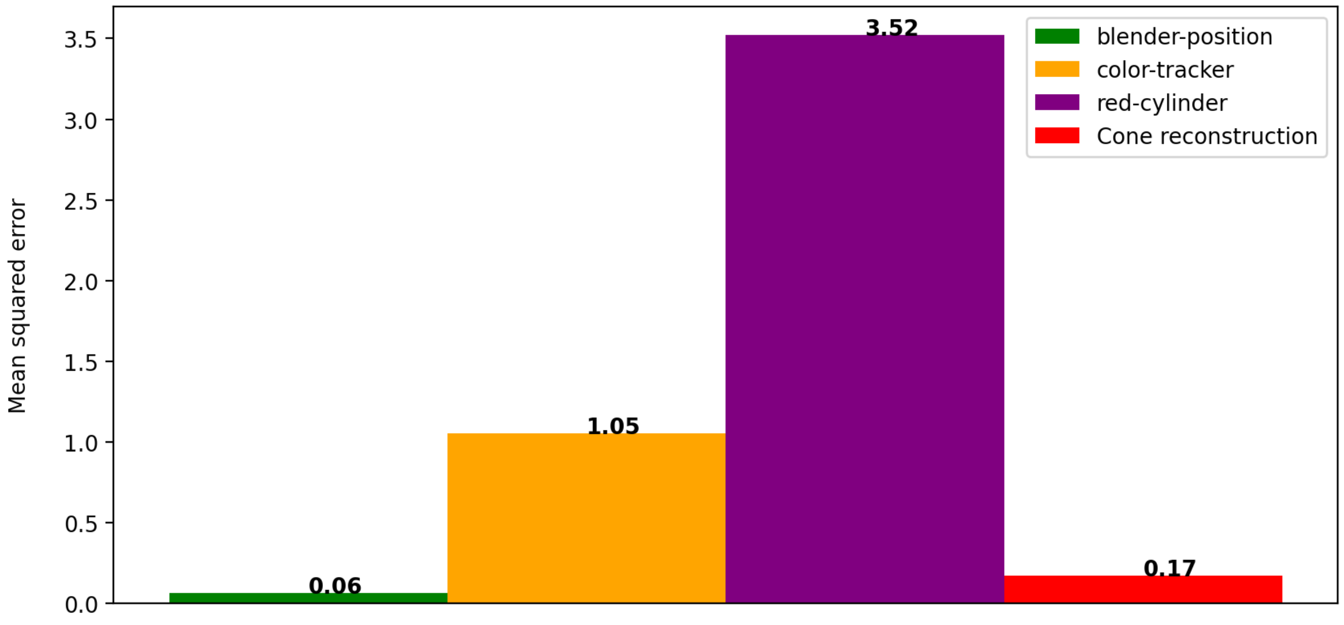
Move track to starting point



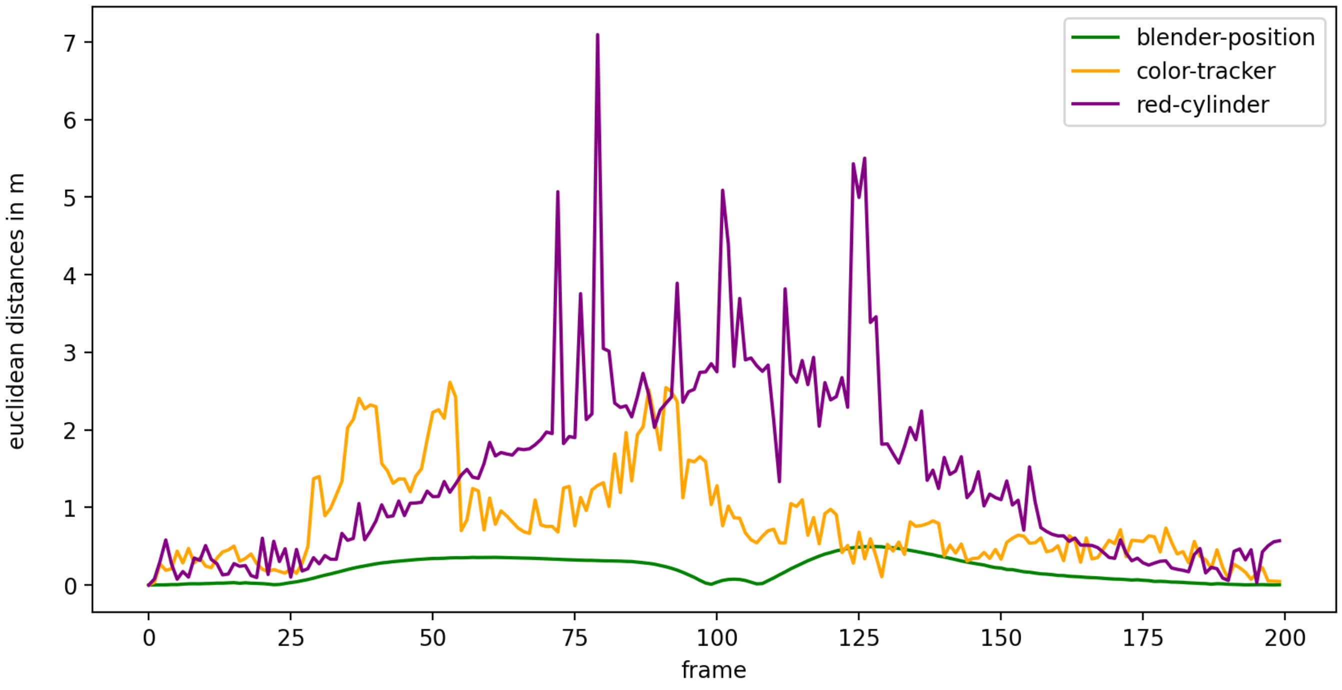
Mean squared error

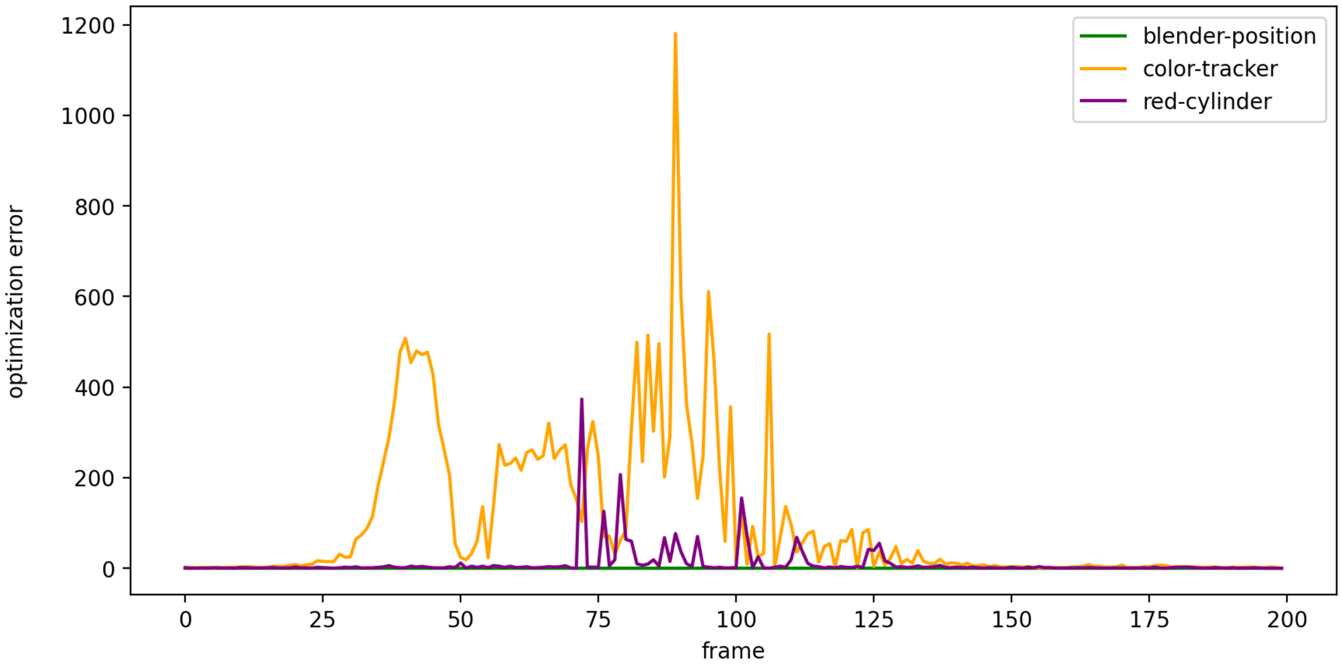


Mean squared error

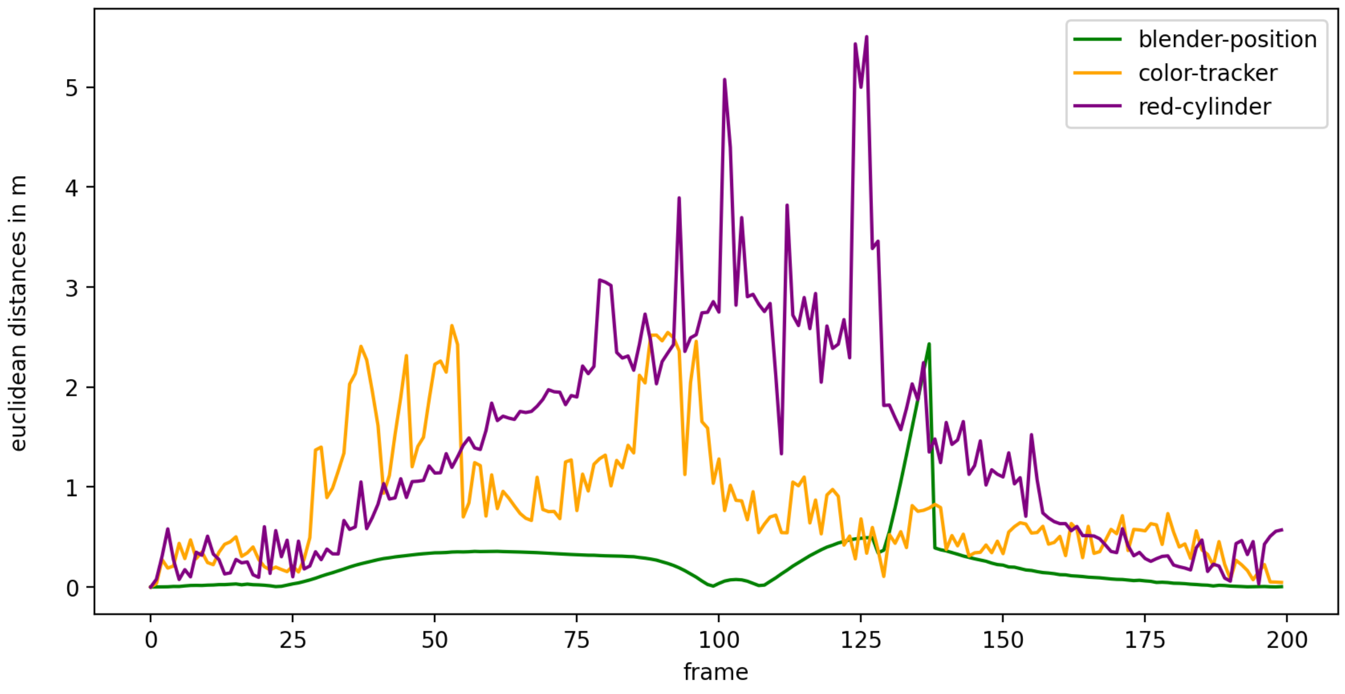


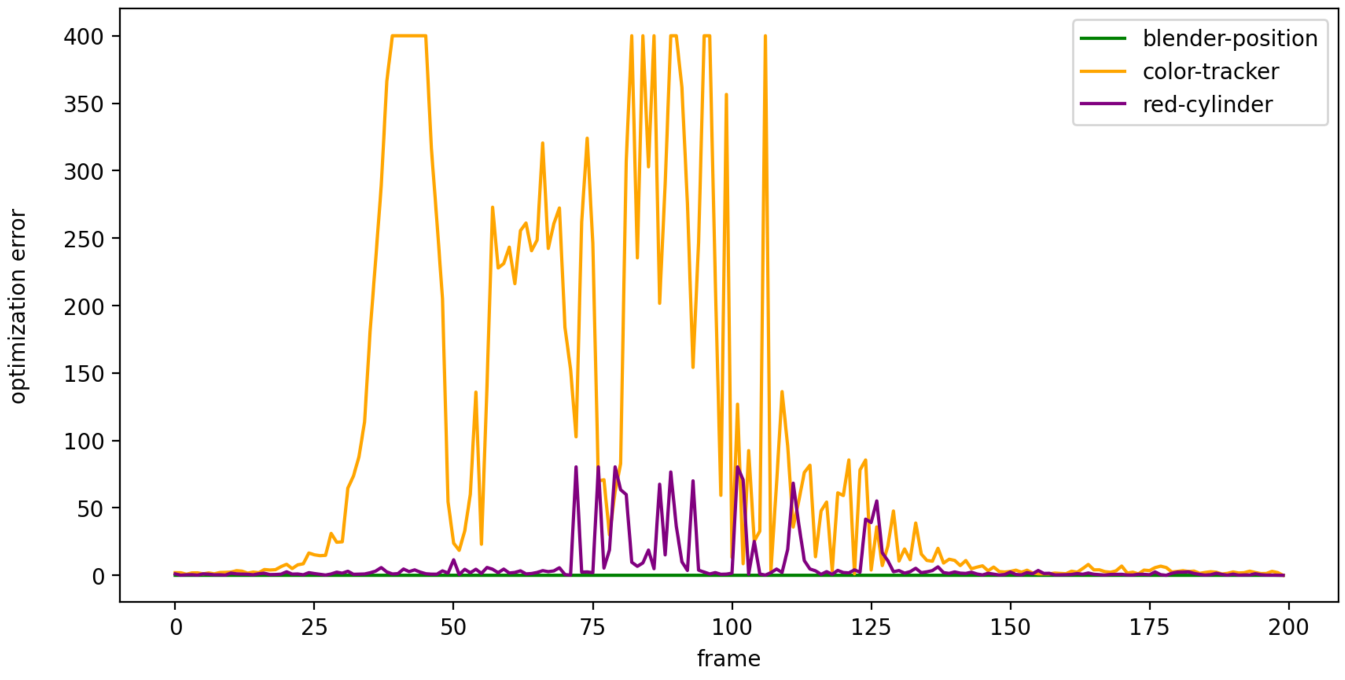
Distance / Error



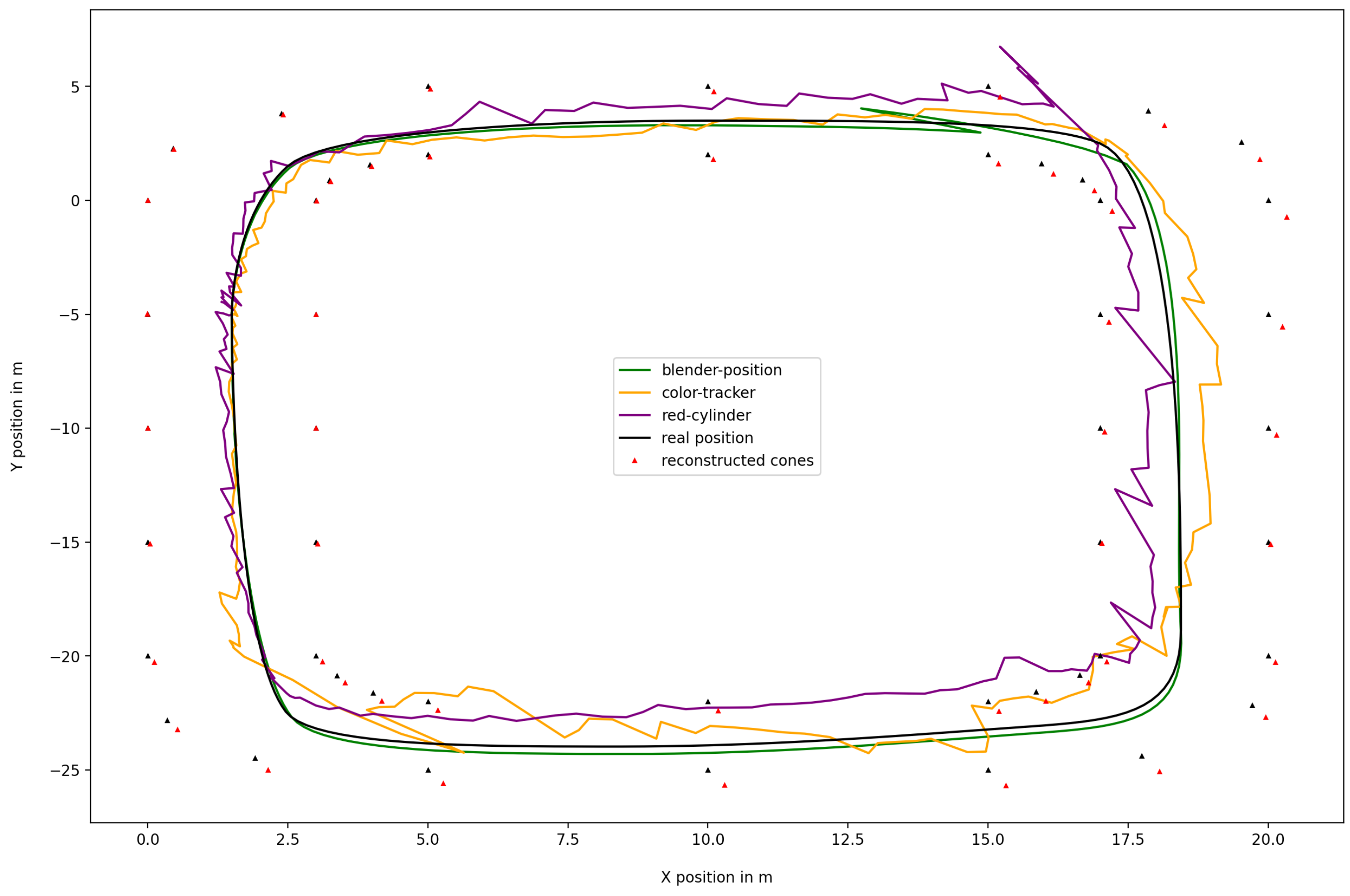


Prune points with high error

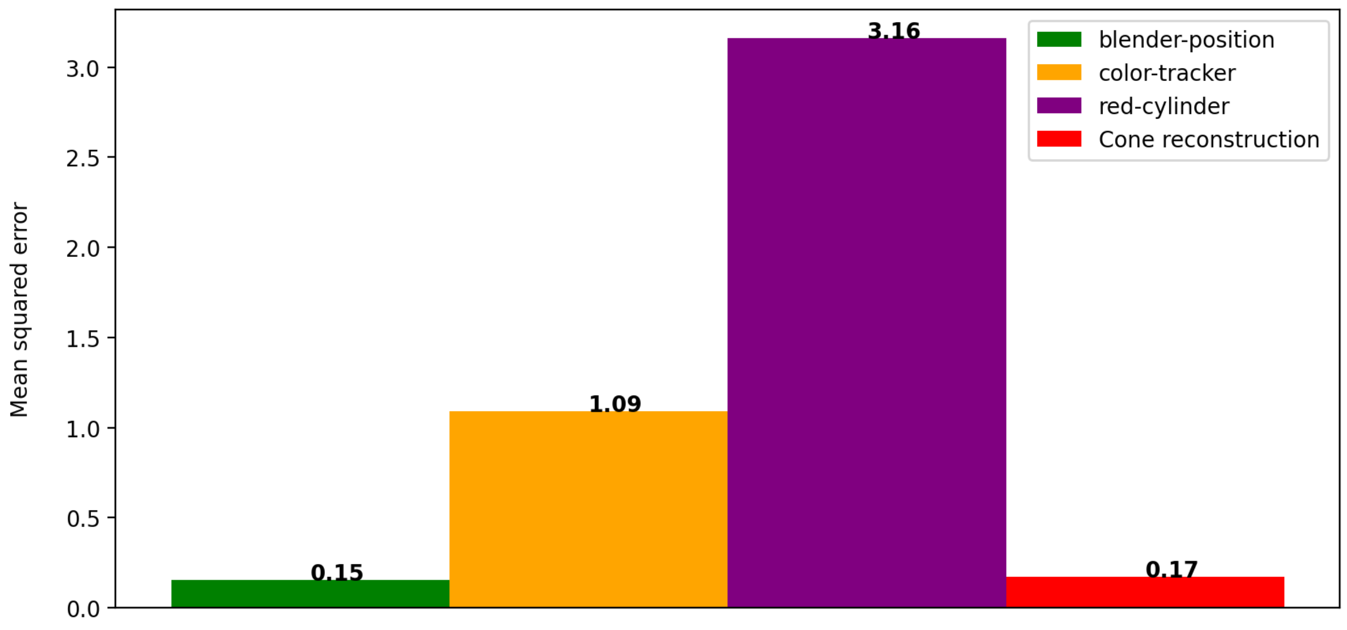

blender-position
color-tracker
red-cylinder
euclidean distances in m
frame
0
1
2
3
4
5
0
25
50
75
100
125
150
175
200


blender-position
color-tracker
red-cylinder
optimization error
frame
0
50
100
150
200
250
300
350
400
0
25
50
75
100
125
150
175
200

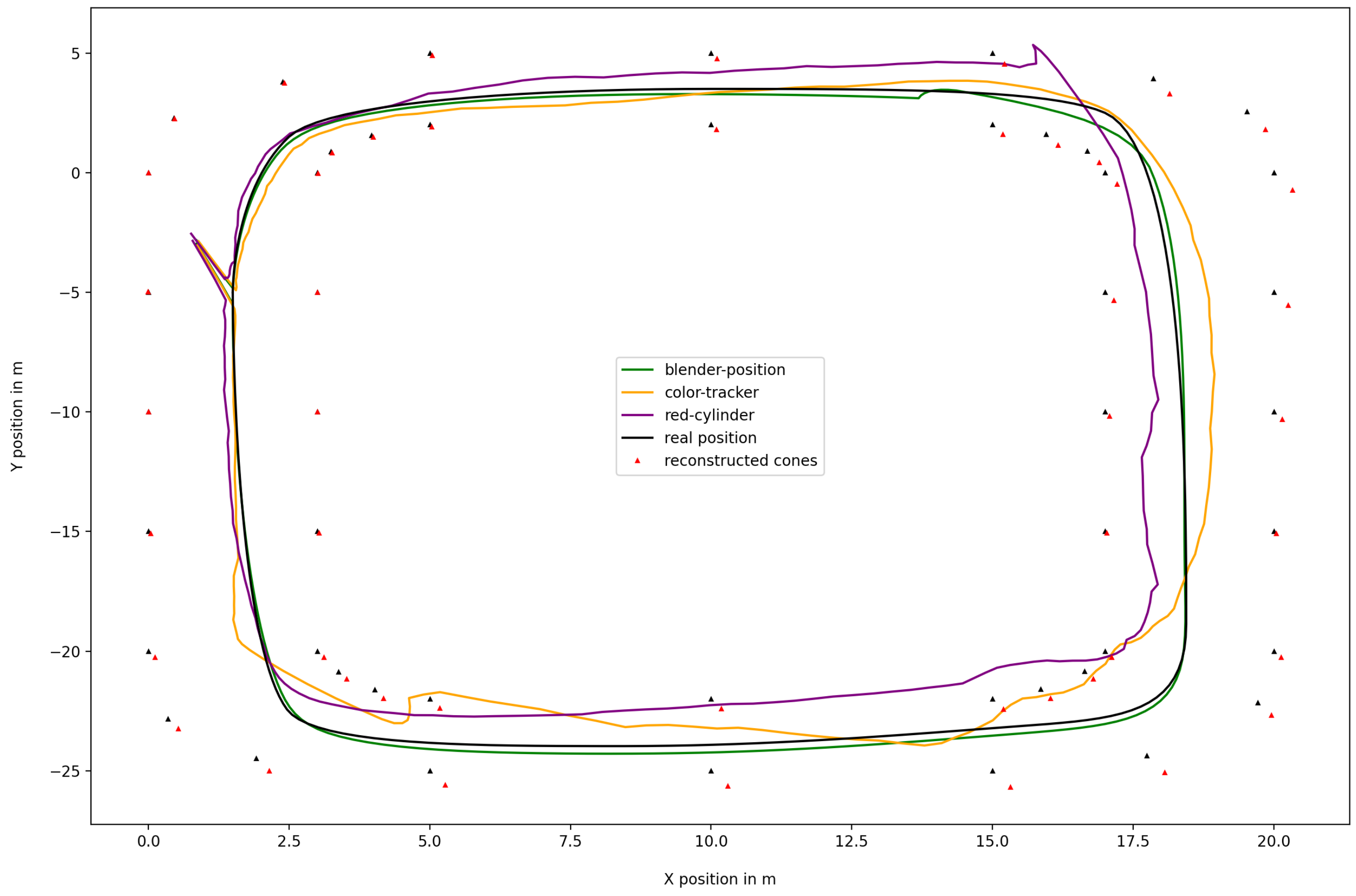
Pruned plot



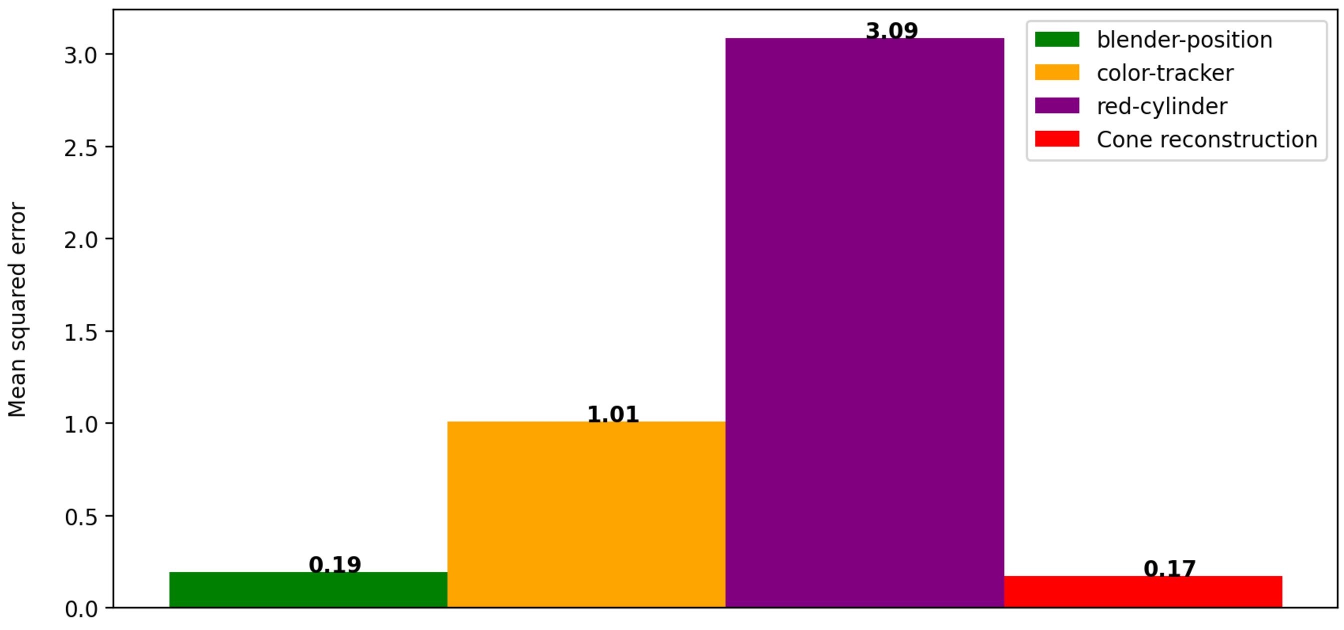
Pruned Mean squared error



Convolution filter



Mean squared error



Evaluation :

- "Perfect" 2D input points accuracy in $~10cm$ possible.
- 3D reconstruction is highly dependent on valid 2D input points.
- Slight noise in input date results in a high error.
- Point of tracking is important.

- Using only Blender generated scene.

- Accuracy and noise of the real world are not considered.

Real World Guide

TODO:

- [ ] explain steps needed to try it in the real world
- [ ] written text not bullet points

1. Set at least 3 cameras which all cover the whole track
2. Get calibration matrix of camera (depends on set focal length)
3. Measure the locations of at least 4 Objects which are not on one plane
4. Film racecar with static cameras (fast shutter speed -> no motion blur)
5. Extract first frame from all cameras and annotate cone position (like previous groupe)
6. Track car in videos to extract 2D position of car
7. Perform 3D Scene Reconstruction run_reconstruction
8. Transform result with affine transformation and the "correct" location of the 4 known objects

Project Limitations:

- We are only evaluating Blender generated scene. We don not know how our solution will perform in a real racetrack.
- Since this is a Simulated racing environment,the accuracy and noise of the real world are not considered.

Conclusion :

- **Future prospects** :
  - Implementing the algorithm in a real-world scenario and evaluating our solution.
  - To improve the tracking accuracy we can try better methods. i.e: Train a CNN model using images of the Racecar.

README

Our source code is heavily based on https://github.com/geohot/twitchslam

Dev container setup

We bundled all dependencies in a docker container as some dependencies are not available in binary form and the correct python version is necessary to be able to run the code. Take a look at the `Dockerfile` for more information. With VSCode we are able to develop directly inside the container without any further setup, just install VS Code and Docker, clone this repository and click on "Open Folder inside Dev Container". To access the graphical user interface you can use the VNC client accessible at `http://localhost:8080/vnc.html`.

Install Dependencies and get rendered video files

To install python dependencies run

```
pip install -r ./requirements
```

The rendered image files are to big to version track (and its not a best practise), you can render them localy from in the script tab of the blender file. To open the Blenderfile you can use the `openBlender.py` script inside a blender directory.

Run Reconstruction

The main python script to start the reconstruction is `run_reconstruction.py`. As a parameter it expects a foldername. In this folder the following things should be present:

- set correct camera focal length in lines 76, take a look at the related blender script to retrieve the correct value
- a `.png` file containing the first frame for every camera, named `01.png` respectivly
- a `.p2d` file containing the 2D points of the cones for every camera, named `01.p2d` respectivly
- a folder called `frames` containing folders for every frame, in each of those:
  - `.p2d` file representing the 2D coordinates of the car for every camera, named `01.p2d` respectivly

```
python run_reconstruction.py blender-racetrack
```

As a result the reconstructed 3D points of the car and the cones are saved in the files `car_reconstruction.p3d` and `cone_reconstruction.p3d`. Those coordinates are in relation to the first camera so they need to be transformed to be human readable.

Transform points

To transform the reconstructed points back to the coordinate system set in blender use the steps:

- write the coordinates of the first 4 points in `cone_reconstruction.p3d` in a file named `known_points`, take care that the first points are not in one plane

```
python transform.py blender-racetrack
```

A affine transformation matrix is calculated and applied to the points in `cone_reconstruction.p3d` and saved in `cone_transformation.p3d`.

Blender

We use Blender to generate test images for reconstruction. For this task Blender can be scripted with the help of python. Scripts are bundled inside the `.blender` file and can be executed from within the "Scripting" tab.

To start blender with the current directory set use the script `openBlender` inside the cone-track folder.

Miscs

- in `utlis.py` we implemented some functions of (pypangolin)[https://github.com/uoip/pangolin] to avoid using this outdated library

---

References

[1] M. B. K. Iagnemma, "Special issue on the darpa grand challenge, part 2," Journal of Field Robotics, vol. 23, pp. 661–692, September 2006.

[2] "Formula student rules 2017 v1.1," 2017.

[3] https://www.mip.informatik.uni-kiel.de/en/theses-and-projects/rosyard

[4] Valls, Miguel & Hendrikx, Hubertus & Reijgwart, Victor & Meier, Fabio & Sa, Inkyu & Dube, Renaud & Gawel, Abel & Bürki, Mathias & Siegwart, Roland. (2018). Design of an Autonomous Racecar: Perception, State Estimation and System Integration. 2048-2055. 10.1109/ICRA.2018.8462829.

[5] High-Speed Tracking with Kernelized Correlation Filters https://arxiv.org/abs/1404.7584

[6] Discriminative Correlation Filter with Channel and Spatial Reliability https://arxiv.org/abs/1611.08461

[7] Learning to Track at 100 FPS with Deep Regression Networks https://arxiv.org/abs/1604.01802