Master's Project: Deep Learning und Autonomous Racing

By
Raphael Schwinger, Rakibuzzaman Mahmud
Supervisors : Lars Schmarje,
Claudius Anton Zelenka


RHEINMETALL
53
BECKER MÜLLER
ALTAIR
VDE


KIEL UAS
OPUS
IBAK
hofer
REPRO
53

Project Overview:

- About "Rosyard" project
- Race-Car discription
- Race-track discription
- The SLAM algorithm

Introduction:

- To optimize the SLAM algorithm it needs an accurate ground truth of the track and the position of the car during a test race.
- This task of ground truth generation is divided into two subtasks.
    - A ground truth of the race track has to be generated.
    - The position of the car while racing has to be aquired.
- The Goal of the project is to design an algorithm that calculates the corresponding ground truth of the racecar.

Possible methods

- **UWB based Triangulation** : Using UWB to trigulate car's position. Similar technology of AirTag but we do not have enough techincal knowledge for implementation.
- **LiDAR** : More accurate but expensive.
- **GPS** : High accuracy GPS is expensive but already commercially available.
- **Image based Triangulation** : Taking the position of the cones/car and using 3D scene reconstruction using images/videos of the race-track.

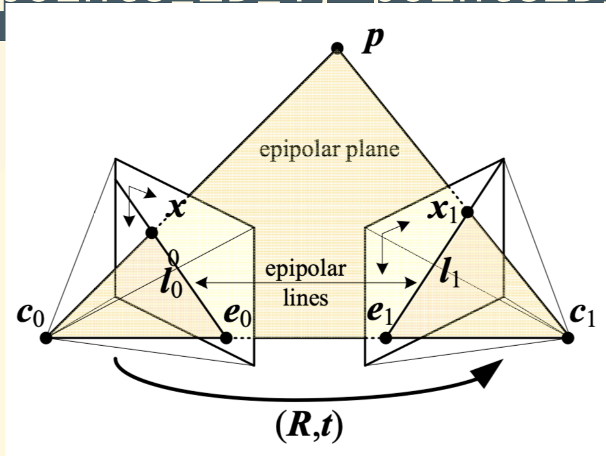
3D Reconstruction

- Structure from Motion: SLAM
  - simultaneous recover 3D structure and poses of cameras
- Input:
  - image coordinates of objects for each camera
  - camera intrinsics (focal length, resolution, ...)
  - "real" position of at least 4 objects

Overview:

- first camera set to origin
- pose of second camera can be reconstructed with essential matrix $E = [t]_x R$

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



Overview:

- 3D points can then be triangulated

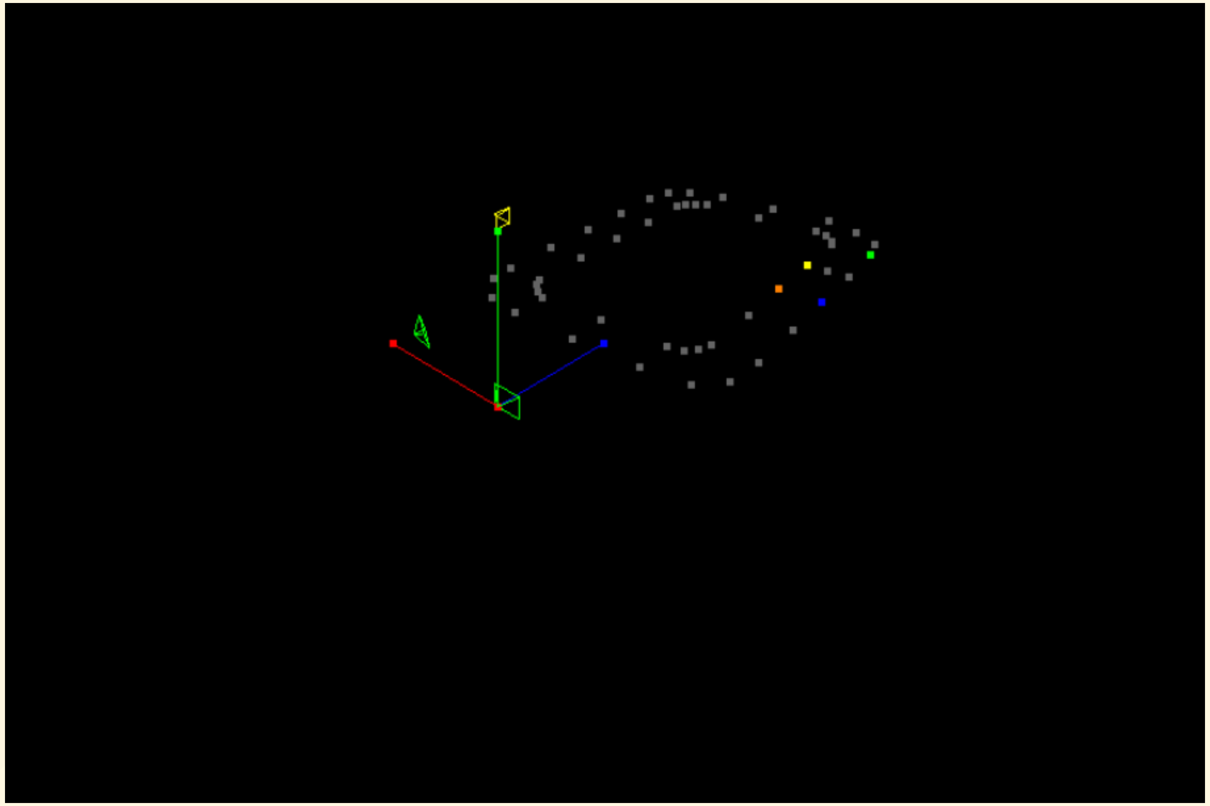
```
points3D = cv2.triangulatePoints(pose_1, pose_2, points1, points2)
```

- for consecutive camera images R and t can be recovered with Random sample consensus RANSAC algorithm
- Rodrigues algorithm can transform rotation vector $rvecs$ in Rotation matrix R

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

Overview:

- further optimization can be achieved by using bundle adjustment
- we included `g2o` library for this purpose



Affine transformation

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

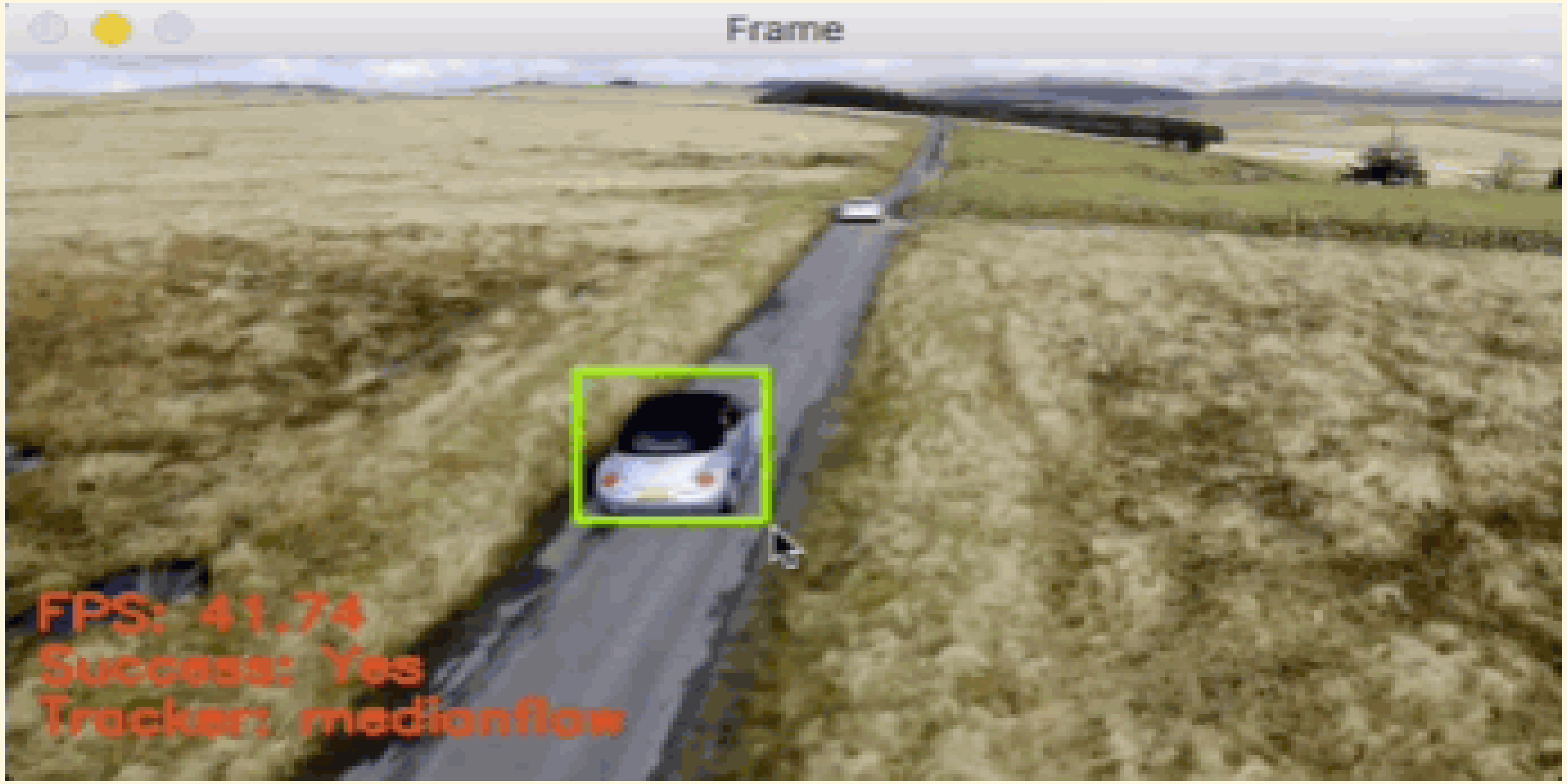
```
    mat = cv2.estimateAffine3D(points_3D[:4], known_points_3d)
    #  [[ 10.19   45.79   -1.93   3.93]
    #   [ 0.26  -100.2   13.86 -18.66]
    #   [ 0.00    0.00    0.00    0.15]]
```

```
-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 :

- **Blender** :
  - Why Blender?
  - Scene Construction
    - Camera Settings : Focal length 15 mm
    - 4k resolution
  - Getting 2D cone and race-car's position point using scripts

Tracking the racecar with OpenCV:



- **OpenCV Tracking Algorithm** :
    - **KCF** : Kernelized Correlation Filte is a novel tracking framework and one of the recent finding which has shown good results.
    - Based on the idea of traditional correlational filter, it uses kernel trick and circulant matrices to significantly improve the computation speed.

- **CSRT** : Channel and Spatial Reliability Tracking is a constrained filter learning with arbitrary spatial reliability map.
- CSRT utilizes spatial reliability map that adjusts the filter support to the part of the object suitable for tracking.
- GOTRUN: ??

- Each tracker algorithm has their own advantages and disadvantages, but for us CSRT worked the best.

```
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:

```
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**
  - Tracking the Racecar based on a color. i.e: Red Colored Cylinder.

```
# definig 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])
```

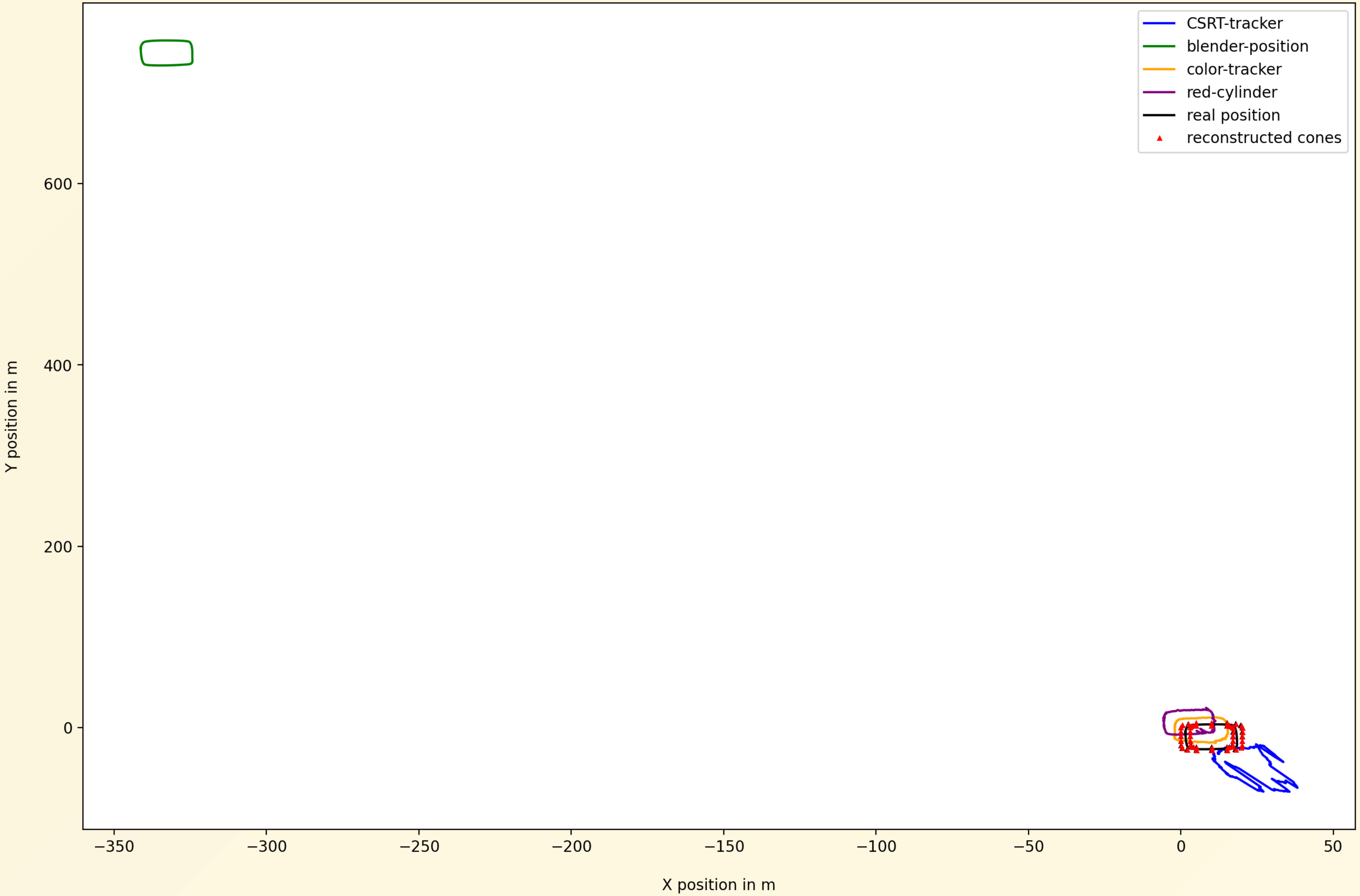
```
# draw rectangle
img = cv2.rectangle(frame, (min_x, min_y), (max_x, max_y), (0, 0, 255), 2)
cv2.putText(frame, "RED color", (x, y),
            cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255))
```

- combine all filter matches to one rectangle
- use bottom of reactangle as center

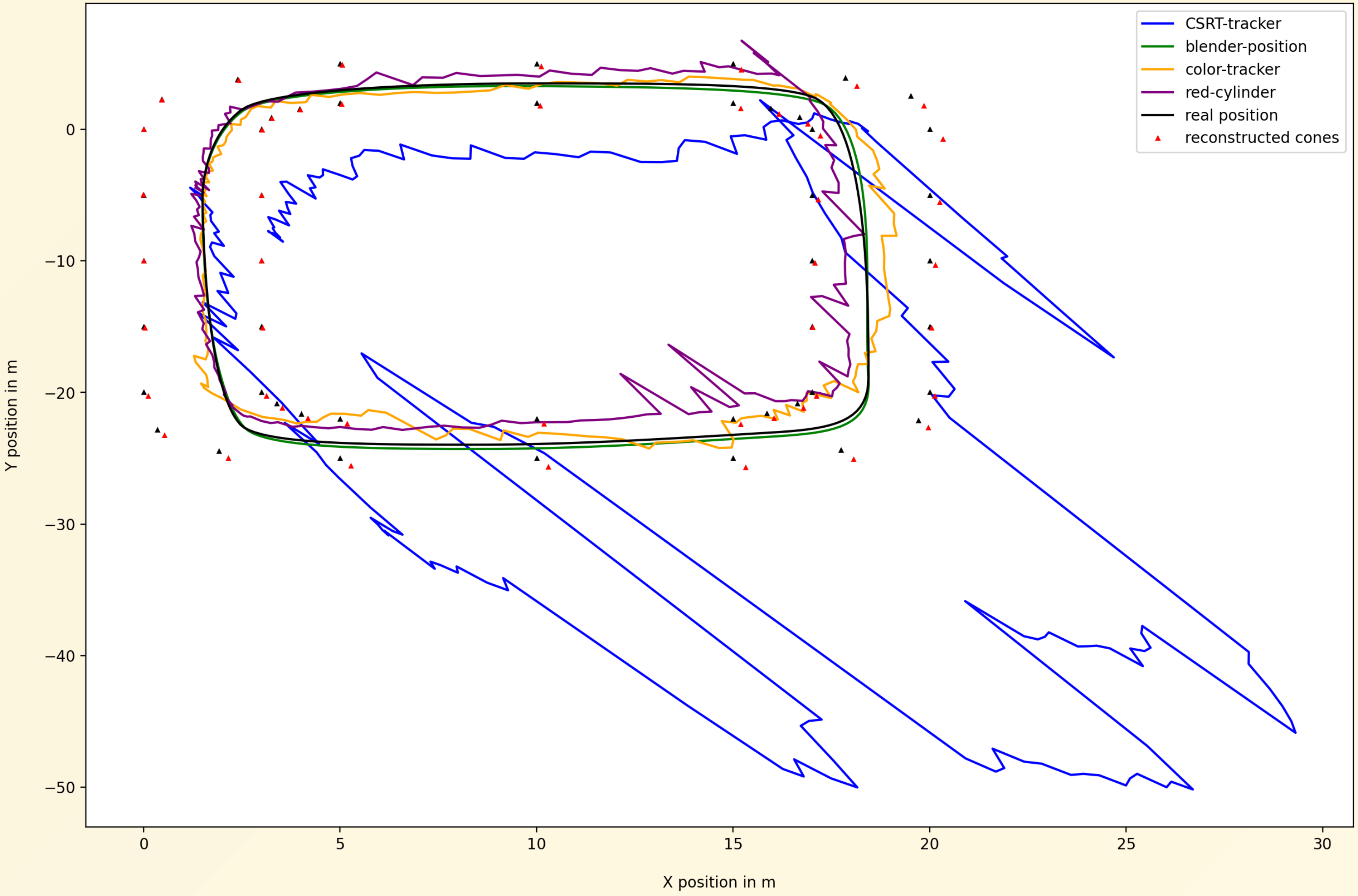
```
# save result
with open(os.path.join(current_frame_path, cam_name + '.p2d'), 'a') as f:
with open(os.path.join(path, 'tracking-result-'+  cam_name + '.p2d'), 'a') as f:

cv2.rectangle(frame, [min_x, min_y], [max_x, max_y], (255, 0, 0), 2, 1)
```

Video Demo of all the tracking methods.

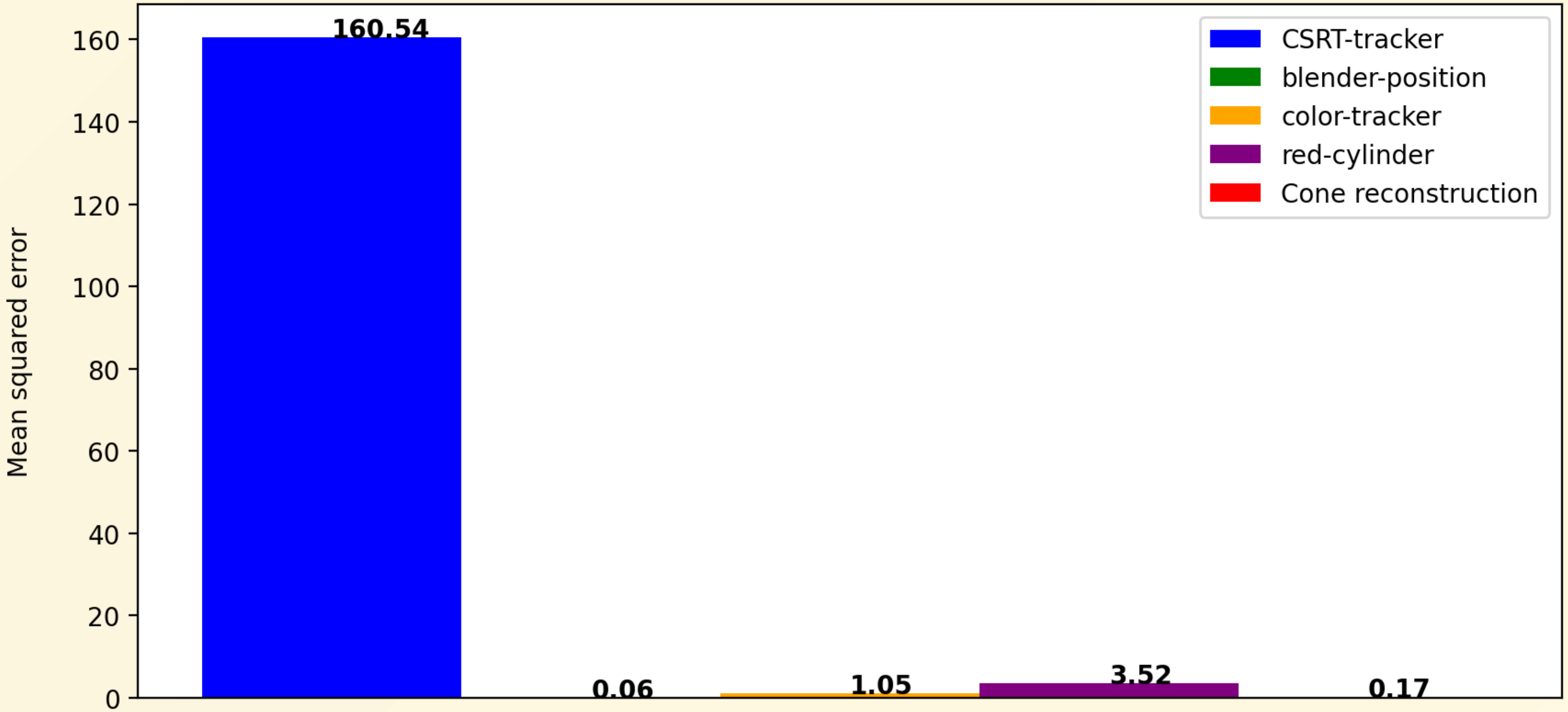
Results:


CSRT-tracker
blender-position
color-tracker
red-cylinder
real position
reconstructed cones
600
400
200
0
Y position in m
−350
−300
−250
−200
−150
−100
−50
0
50
X position in m

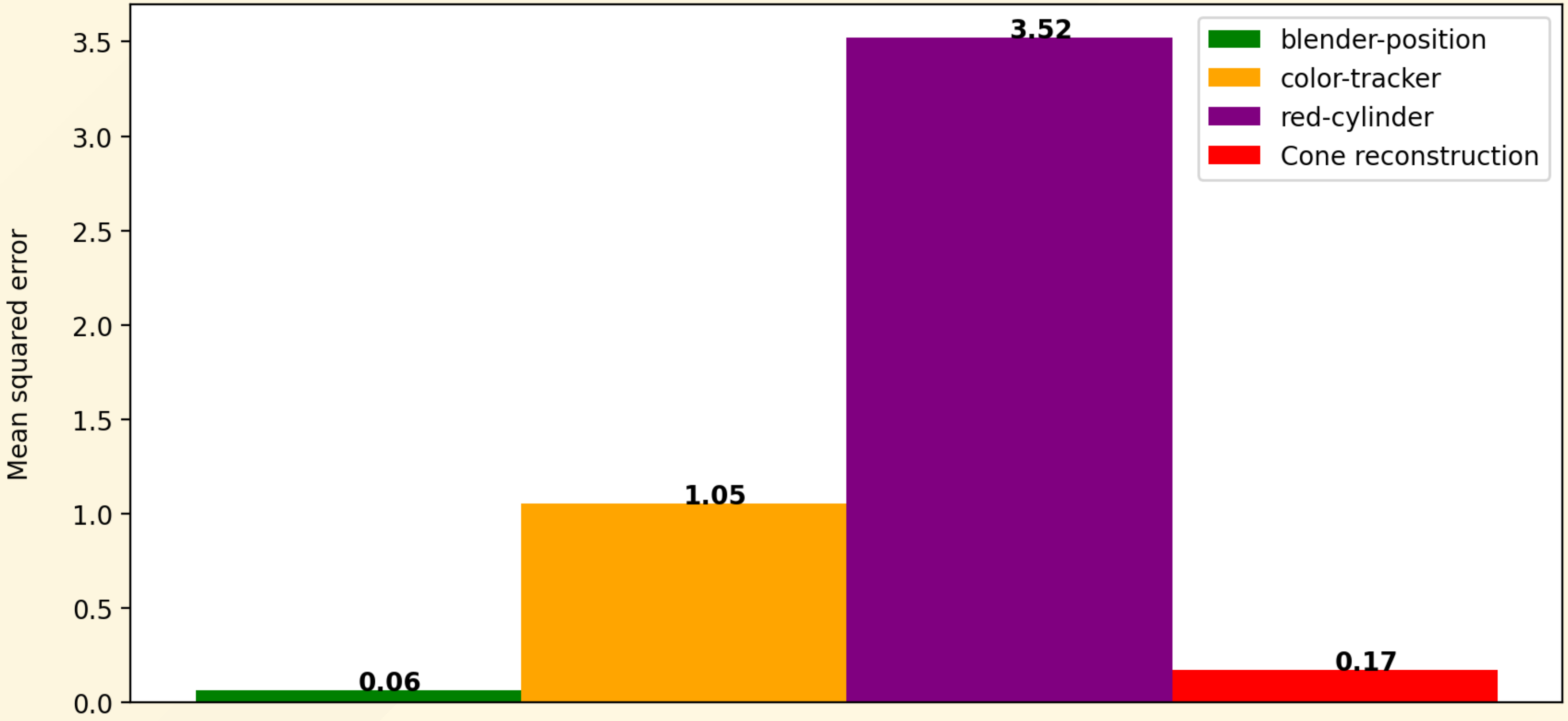
Move track to starting point



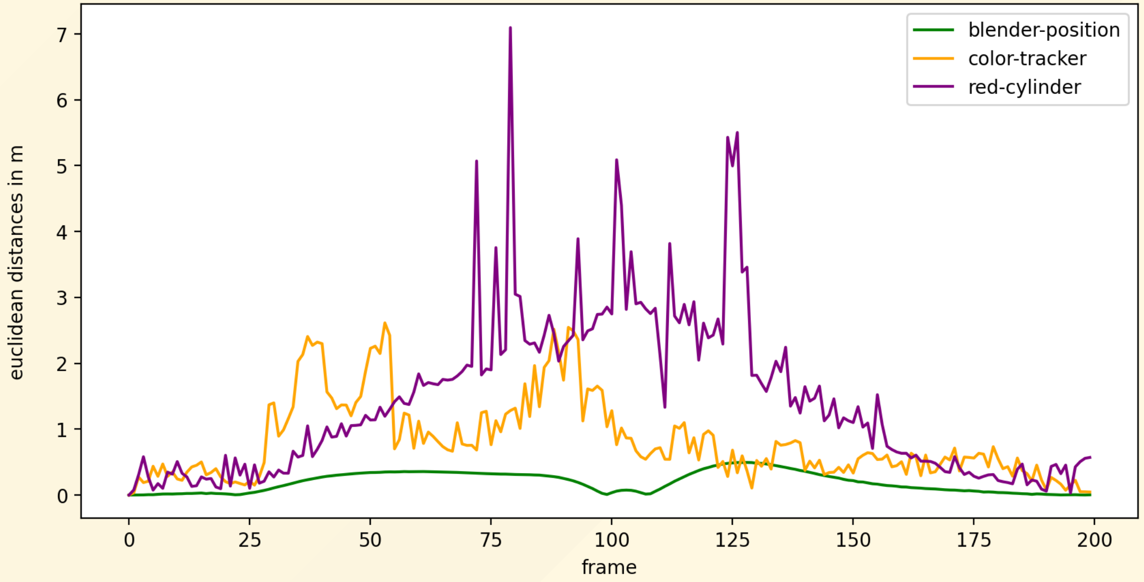
Mean squared error

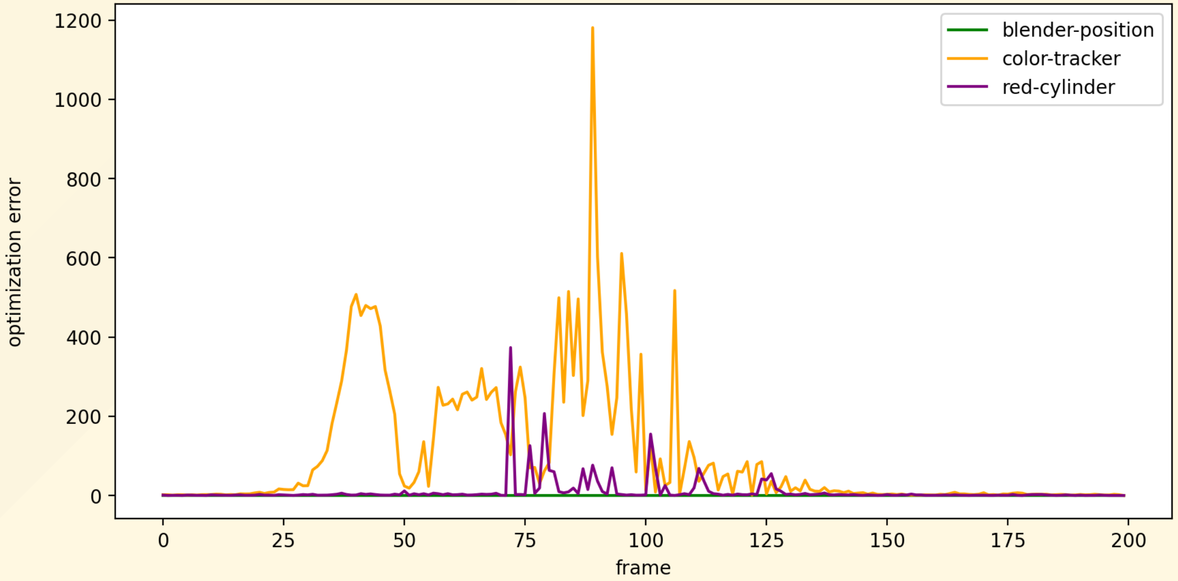


Mean squared error

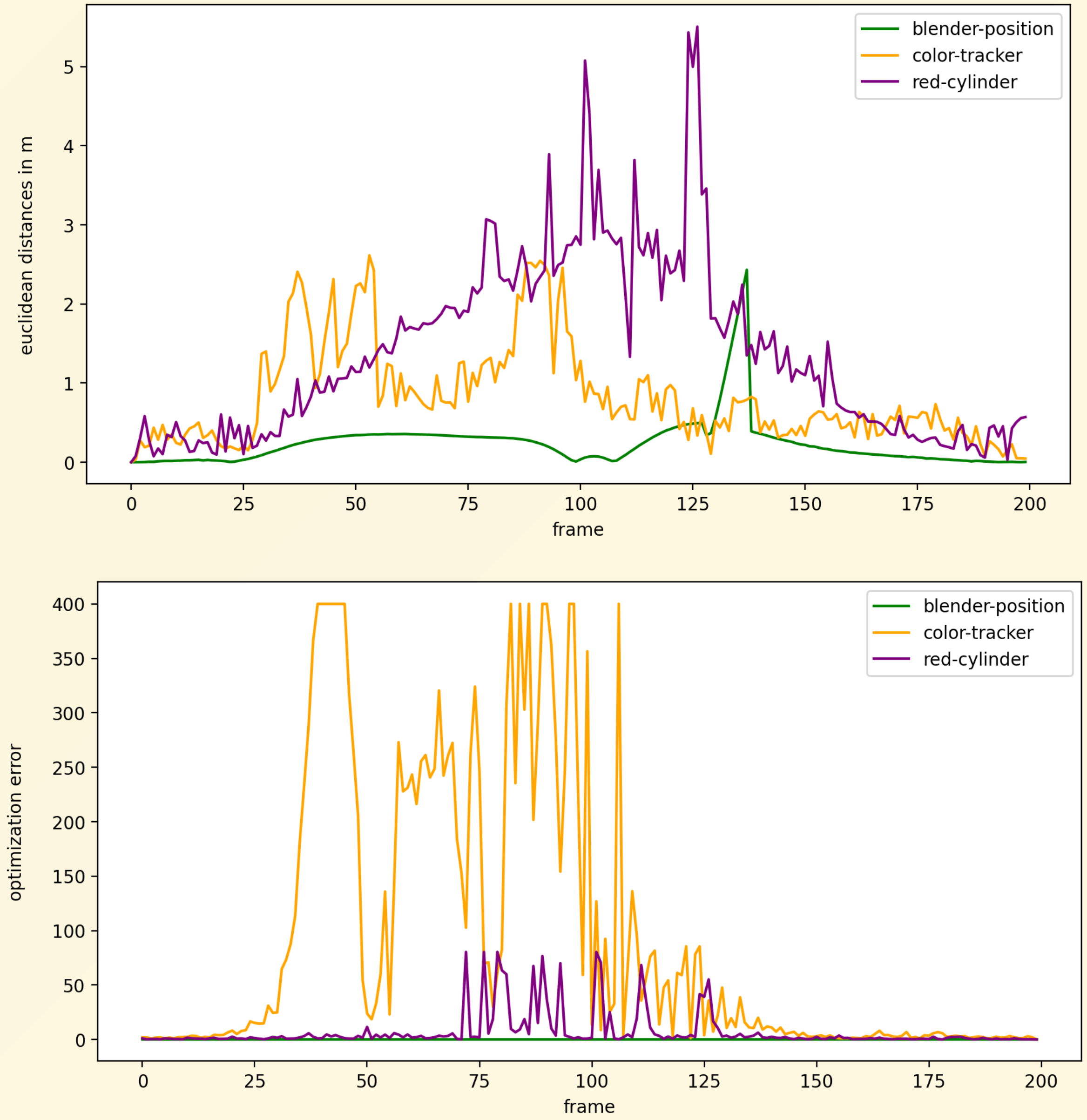


Distance / Error

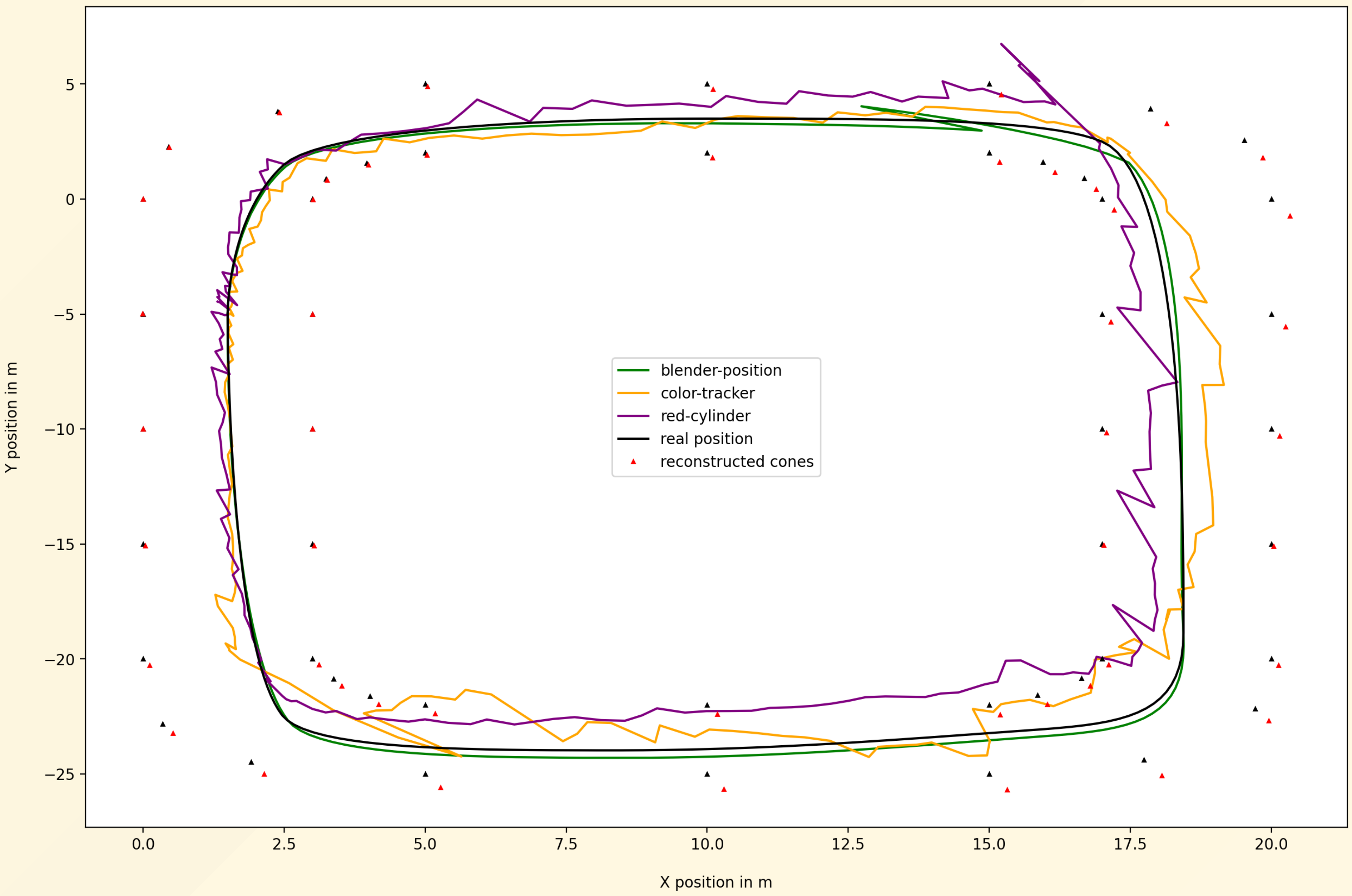




Prune points with high error

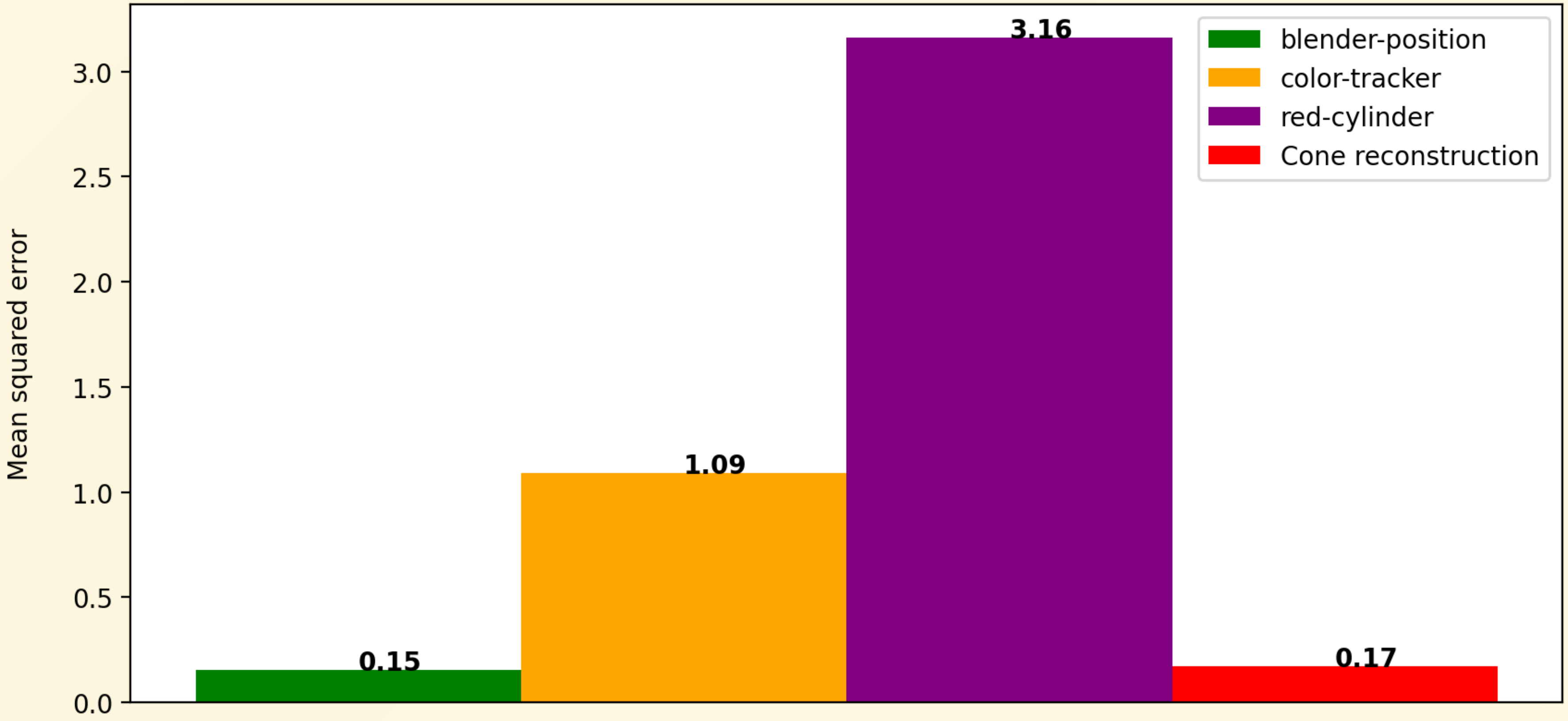


Pruned plot

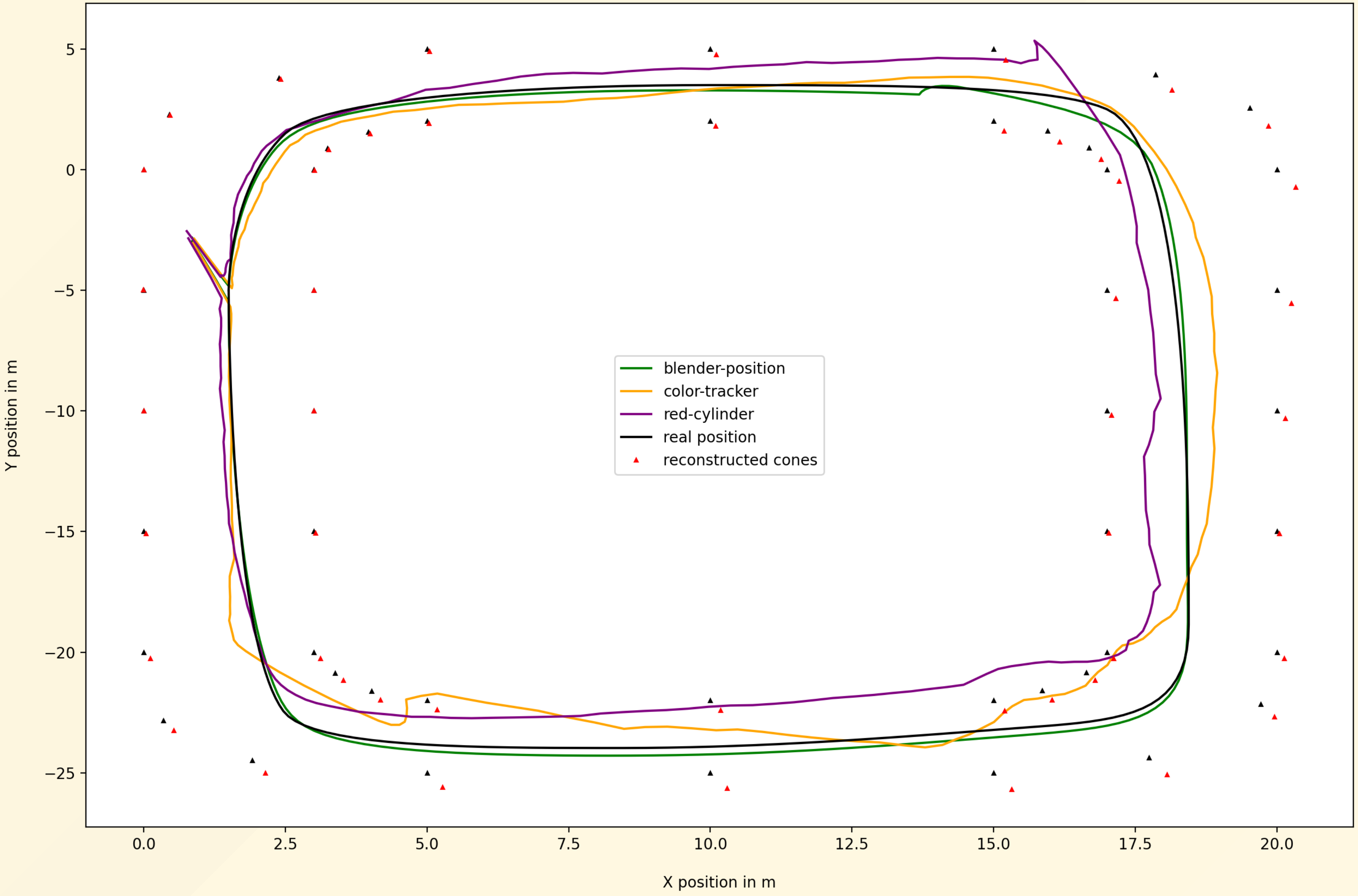


blender-position
color-tracker
red-cylinder
real position
reconstructed cones
X position in m
Y position in m
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
5
0
−5
−10
−15
−20
−25

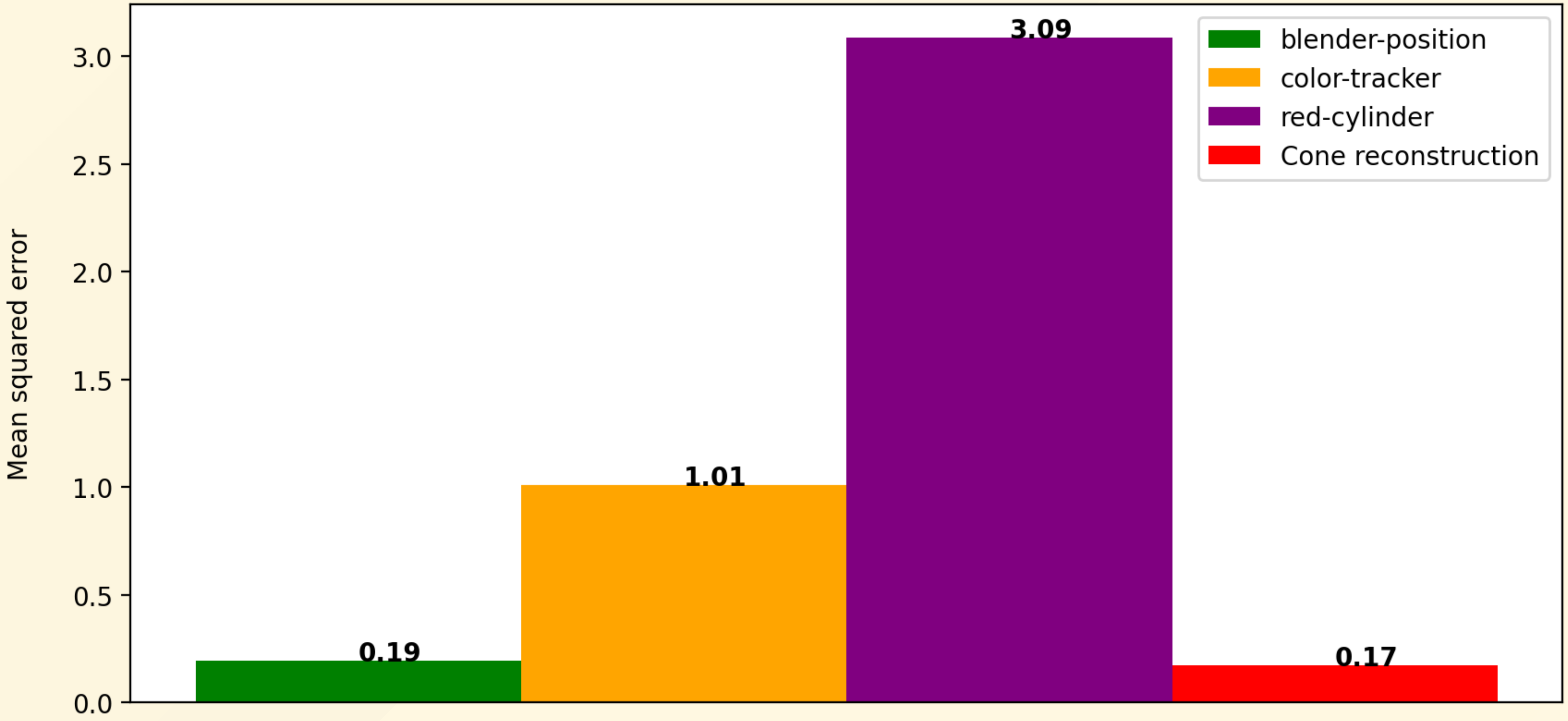
Pruned Mean squared error



Convolution filter



Mean squared error



Evaluation :

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

Project Limiations:

- Using only Blender generated scene.
- Accuracy and noise of the real world are not considered.

Conclusion :

- **Future prospects** :
  - Implementing the algorithm on a real-word scenario.
  - To improve the tracking accuracy we can try better methods. i.e: Train a CNN model using images of the Racecar

References:

- https://szeliski.org/Book/