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. 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. With the help of the results of this master project in a test race the SLAM algorithm can then be evaluated and optimized.

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

Possible methods

During the project's planning phase, we discussed many ideas that could be used to generate the ground truth of the racecars movement and the track.

- **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.

We decided to use 3D scene reconstruction using images/videos of the race track since we thought that might be the cheapest way and also scientifically the most interesting one.

3D Scene Reconstruction

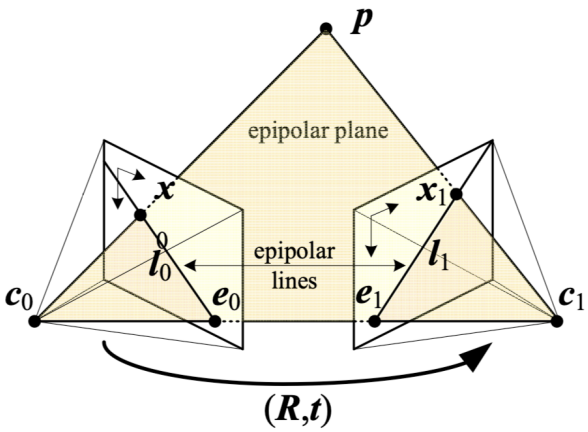

Fig: Camera matrix

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 as illustrated in the figure above. So we need to obtain the translation t and rotation R of the second camera in relation to the first camera. This can be calculated with the `essential matrix` $E = [t]_x R$. OpenCV provides the following functions that need the 2D input points of both cameras and the camera intrinsics as an input.

```
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.[8]

```
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` using the already estimated 3D points.

```
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. The figure below shows our initial visualisation of the result for the first frame.

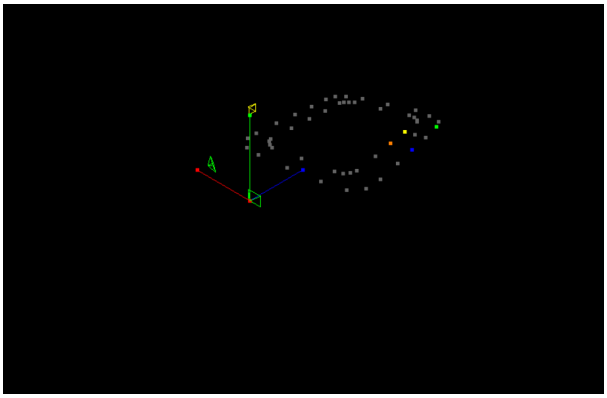

Fig: Blender Reconstruction

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 an `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 positions are not on the same plane to be able to transform 3D points that do not lie on this plane. The figures below show the initial recovered 3D coordinates of the first four cones and the 3D coordinates after applying the affine transformation and matching the expected 3D coordinates. [9]

```
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]]
```

Reconstruction of the race-track using Blender :

As mentioned above, to apply our 3D scene reconstruction algorithm we need video files of the car racing around the track filmed from at least 3 angles. Then we need to acquire the car's and cones 2D position for each frame. However, because of Covid-19, it was not possible to make a real-world racing scenario and record racing videos of the car. So we had to improvise and work on a simulated racing environment which we created on Blender. This gives us also the benefit of directly exporting the 2D coordinates for each frame of the video. Additionally we do not need to worry about any noise in the video files as the cameras in blender are not suffering from physical constraints.

Blender environmental elements:

- **Camera**: We set the height of the camera as 1.5 m and focal length to 15 mm. We used a python script for camera calibration which is included in README. The video we exported has a resolution of 4k.
- **Car**: We used a realistic Rosyard car model with height of 0.90 m , width 1.15 and length 2.45 m.

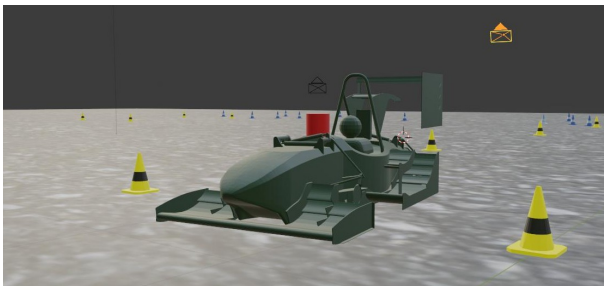

Fig: Blender Car Model

- **Track**: To mark the track, We used yellow and blue cones with a height of 0.15 m. We made a circular racetrack with 48 cones. We added asphalt texture on the ground to make the racetrack look like a real racetrack. Also to make the environment look more realistic, we used a "skydome" to make it look like a sky on the horizon.

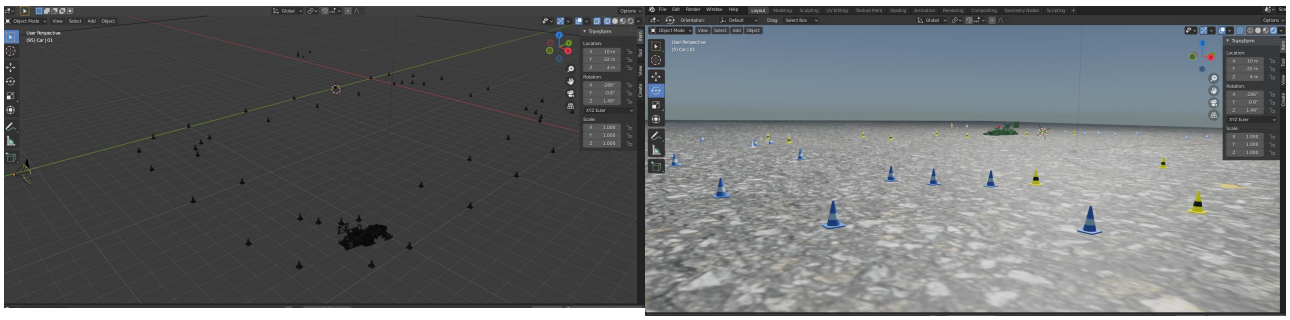

Fig: Circular Racetrack

Tracking the racecar

Since we can not export the 2D image coordinates from Blender in the real world we need to acquire them differently. For the cones this task can be done manually as since the cameras and cones do not move that needs to be done for only for the first frame. Also the previous group did some research in that area to automate this task and had no satisfying results as not only the cones need to be detected but also correctly mapped from all angles. To get the 2D position of the racecar we tried multiple tracking algorithms. Details and background of the algorithms are included below:

- **OpenCV Tracking Algorithm** :
  - **KCF** [5]:
    Kernelized Correlation Filter (KCF) is a novel tracking framework, and it is one of the recent findings which has shown promising results. KCF is based on the idea of the 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. It utilizes a spatial reliability map. CSRT adjusts the filter support to the part of the object suitable for tracking.
  - **GOTRUN** [7]:
    Generic Object Tracking Using Regression Networks (GOTRUN) is a Deep Learning based tracking algorithm. GOTRUN is significantly faster than previous methods that use neural networks for tracking. The tracker uses a simple feed-forward network without any online training, and it can track generic objects at 100 fps.

After testing the tracking algorithms extensively we found out that `CSRT` performed best on our generated video files. Since even the results of `CSRT` were not very convincing and we could not get any real world data where further optimization would be required anyway we tried tracking a color object. Therefor we colored ether the whole car red or added a red-colored cylinder on top of the racecar to make the tracking even more accurate. Then we only had to apply a color filter of the image and retrieve the 2D coordinates of the center of the filtered portion.

Results

We ran our 3D scene reconstruction algorithm with the tracking results of the `blender-position` script, the `CSRT-tracker` ,the `color-tacker` tracking the whole car and tracking only a `red-cylinder` on top of the car. To better compare the results we than aligned the starting positions of the racecar. The result is shown in the figure below.

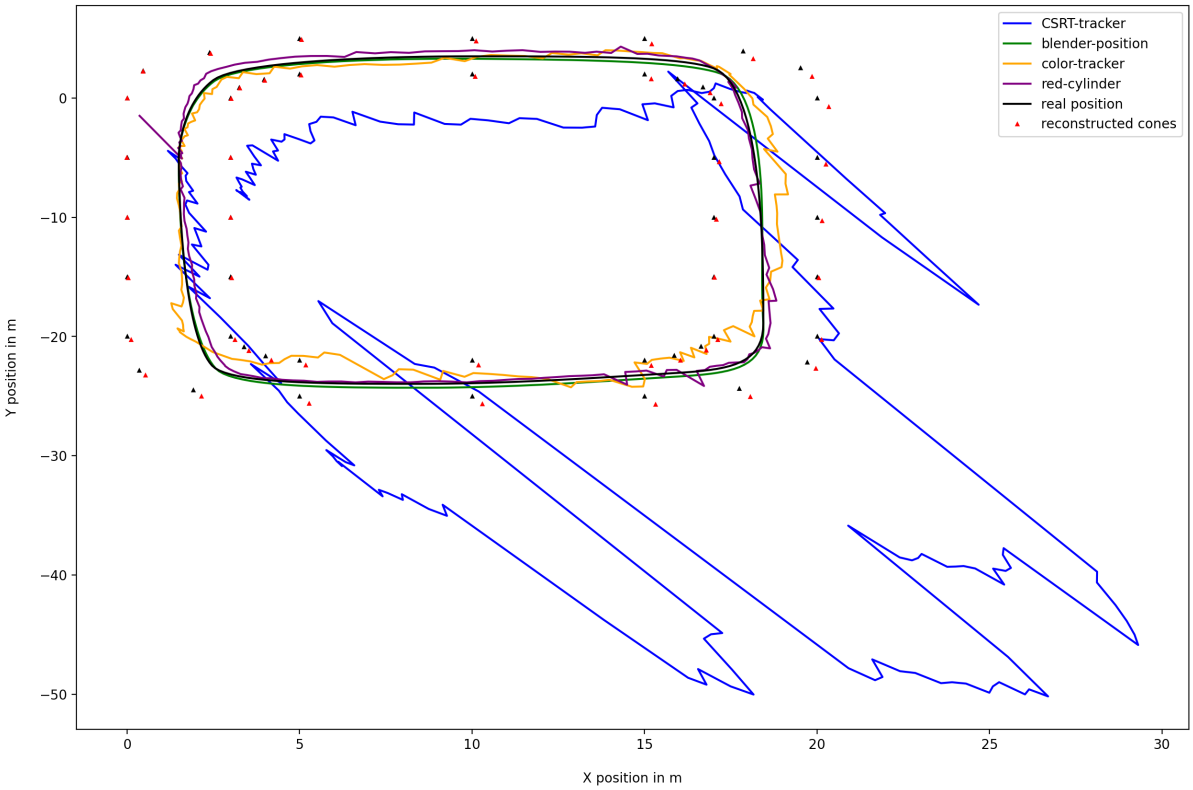

Fig: 2D plot of reconstruction

As we can see the reconstruction using the `blender-position` is performing very well. Also the `reconstructed cones` are very close to the original cones. `CSRT-tracker` is way off from the `real position` of the racecar and is jumping around a lot. `color-tracker` and `red-cylinder` are able to follow the racecar and almost recover a position of the racecar between the cones all the way around the racetrack.

As shown in the next figure the mean squared error confirms those findings. We did not include the high error of `CSRT-tracker` which is `160.54` in the graphic.

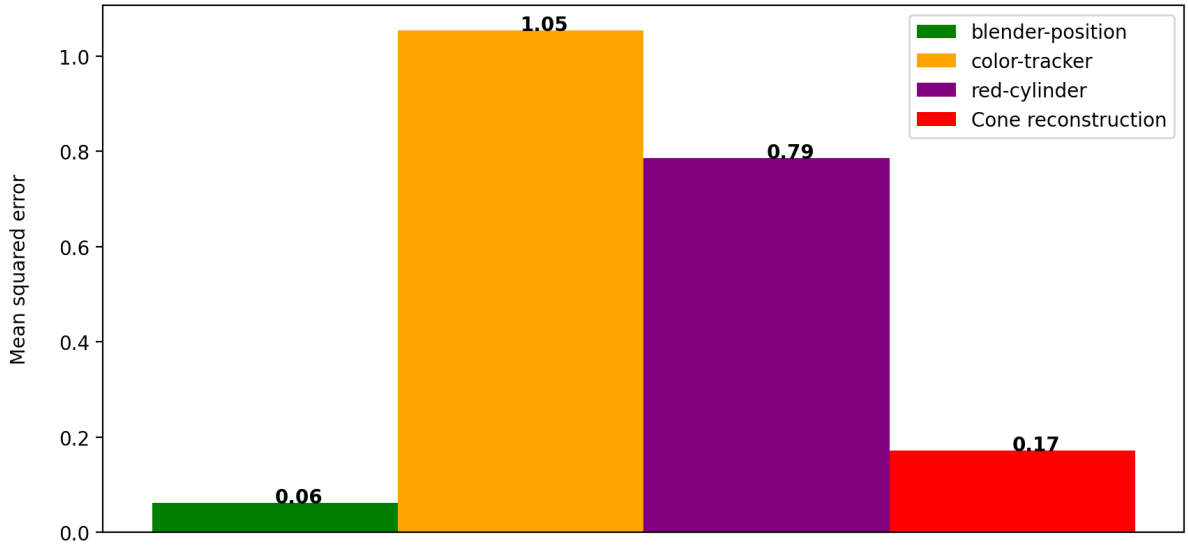

Fig: mean squared error

We found a correlation between the optimization error returned by the `g2o` optimizer and substituted those points. Also we applied a convolution filter to smoothen the result as you can see in the figure below. This did not improve the mean squared error tough. For more details take a look at the jupiter notebook `notebook.ipynb` we used to analyse the reconstruction results.

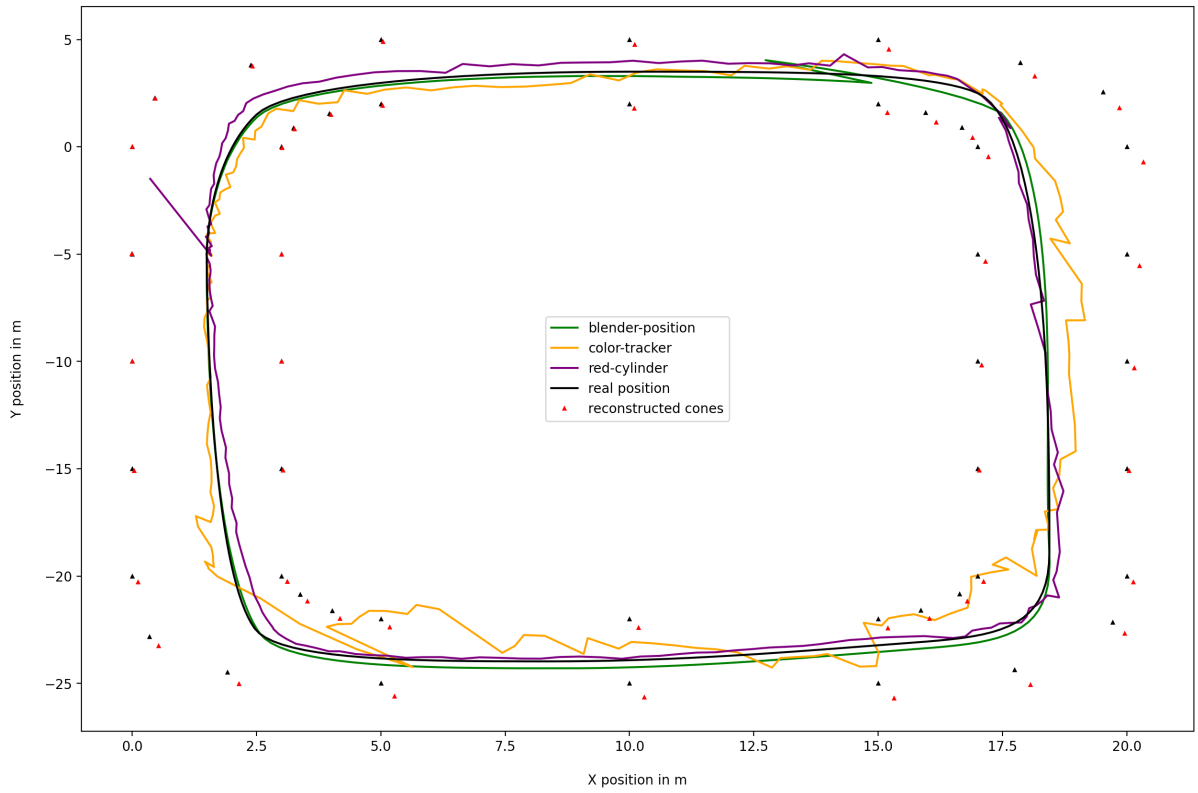

Fig: smoothened 2D plot

Real World Guide

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 group)
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

Conclusion

We where able to reconstruct a racetrack and a car driving around the racetrack by applying a 3D scene reconstruction algorithm. We can see that with "perfect" 2D input points as we get them from blender accuracy around $10cm$ is possible. Therefore our method is in theory suitable to improve the SLAM algorithm driving the car. We experienced that the 3D reconstruction result is highly dependent on valid 2D input points as slight noise results in a high error. That is why precise tracking is important. Also, as our input data is generated by Blender we do not suffer from noise and are using a optically camera with out any distortion or optically imperfections. In the real world those effects will have a negative impact on the accuracy of the reconstruction.

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 locally 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

[8] Richard Szelisk, “Computer Vision: Algorithms and Applications”, March 27 2021 draft, pp. 681-687

[9] Richard Szelisk, “Computer Vision: Algorithms and Applications”, March 27 2021 draft, p. 41