Development of an autonomous driving environment model visualization based on object list level

Christoph Zach, Denis Rösler, Dominik Knauer, Maximilian Pfaller,
Maximilian Haindl, Philipp Korn, Stephan Schweigard and Tobias Wagner

***Abstract*— Simulating vehicle test scenarios, detecting objects with a camera sensor and analyzing the resulting object detections afterwards - in this paper it is described how this can be performed using several applications. Within the scope of a students' project the development environment is created and the quality of object detections by a simulated camera sensor is evaluated with a generated post-processing application.**

I. INTRODUCTION

Camera data and its camera-based algorithms are increasingly being used to make people, vehicles, objects and buildings visible for automated vehicles. With the help of these algorithms, the raw camera data can be evaluated regarding to various criteria. Important distinctions are the classification, dimensions, distance, alignment and the relative velocity of the detected objects in relation to the ego-vehicle [1]. You Only Look Once (YOLO) is one of the most effective real-time object detection algorithms and offers all of these functions [2]. Due to the fact that new driving functions usually are validated on a proving ground under controlled conditions, environment simulation software such as CARLA are used instead for developing, training and validating driving systems [3]. CARLA is an open-source urban driving simulator for autonomous driving research which supports flexible sensor suites, user scripts and full control of all static and dynamic actors and maps [4]. This offers the possibility to validate the camera data sets using different kinds of camera-based algorithms for object detection or in addition to train them. To evaluate the Ground-Truth (GT) data with the calculated algorithm objects, Robot Operating System (ROS) offers the possibility of sending the respective data streams using objects lists and evaluating them with a 3D visualization tool (RVIZ) [5]. In this work, an autonomous driving environment model visualization based on an objects lists level is presented by using a NCAP test scenario for testing automated driving systems. An analysis is made based on the YOLO camera object detection algorithm compared to the GT Data directly generated from the environment simulation software model. Figure 2 illustrates the basic process flows for generating all data, visualizing and validating the results.

II. RELATED WORKS

For detecting objects in an environment an object description model is necessary. This work is based on the model introduced in [1] which depicts an object through vectors and values shown in fig. 1. To compare objects lists with each other and to evaluate detected objects the authors of [12] developed a test method. It presumes the existence of an objects list and the associated GT dataset, therefore this work concentrates on creating these lists. To investigate the quality of the created objects lists, metrics of [12] are used.

III. MATERIALS AND METHODS

A. Creating NCAP test scenario CPNC-50

A major part of the project deals with the used software to create the test environment and the test scenario itself. With its variable sensor suites and full controllable static and dynamic actors an maps CARLA is especially suitable for setting up an own test scenario [4]. The adaptable application programming interface (API) and the ROS integration provides a lot of flexibility and the possibility to extract the GT data directly from the scenario. The work is based on CARLA 0.9.8 release combined with ROS melodic which uses python3 packages and an API based on python3.5. The test case is derived from the Euro NCAP autonomous emergency braking scenario which uses the Car-to-Pedestrian Nearside Child 50 % (CPNC-50) test case from the Insurance Institute for Highway Safety (IIHS) test protocol [6], [7]. Table I presents the elementary test conditions for the created test scenario.

TABLE I
TEST CONDITIONS PEDESTRIAN AUTONOMOUS EMERGENCY BRAKING (P-AEB) [7]

Parameter	CPCN-50 Scenario Child
Test vehicle speed	40 km/h
Pedestrian target speed	5 km/h
Target direction	Crossing from R-to-L
Target path	Perpendicular
Pedestrian dummy size	Child
Overlap	50 %

- An object list with N objects:
  $\mathcal{O} = \{O_1, O_2, \ldots, O_N\}$
- A single object consists of:
  $O_i = \{\hat{\mathbf{x}}, \mathbf{P}, \hat{\mathbf{d}}, \mathbf{d}_{\sigma^2}, p(\exists \mathbf{x}), \mathbf{c}, \mathbf{f}\}$
  - ▶ State vector $\mathbf{x} = [\ x \quad y \quad v_x \quad v_y \quad a_x \quad a_y \quad \psi \quad \dot{\psi}\ ]'$
  - ▶ Dimension vector $\mathbf{d} = [\ l \quad w\ ]'$
  - ▶ Classification vector $\mathbf{c} = [\ C_{\text{Car}} \quad C_{\text{Truck}} \quad C_{\text{Motorcycle}} \quad C_{\text{Bicycle}} \quad C_{\text{Pedestrian}} \quad C_{\text{Stationary}} \quad C_{\text{Other}}\ ]'$
  - ▶ Feature vector $\mathbf{f} = [\ FL \quad FR \quad RL \quad RR \quad FM \quad RM \quad ML \quad MR\ ]'$

Fig. 1. Object list vector [1]

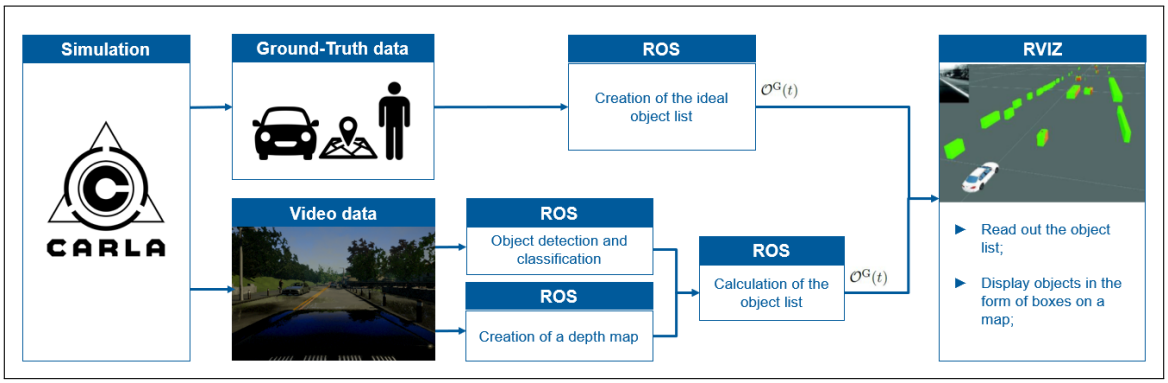

Fig. 2. Process diagram for data generation, illustration and evaluation

The test procedure starts with launching the test scenario by first spawning the three vehicles and the pedestrian to their initial positions into the map. This state consists of an Audi TT (1) in front and an Audi e-tron (2) arranged behind it. The left edges of both cars are parked 0.2 m away from the right edge of the test lane. The longitudinal distance between the cars and between the front car and pedestrian is 1.0 m, each. At the beginning of the simulation, the child is positioned 7.0 m laterally from the center of the ego-vehicle, which is centered in its lane 200 m behind the pedestrian and portrayed as an Audi e-tron. A few seconds after spawning, the ego-vehicle starts accelerating quickly to 40 km/h and the child starts moving with constant speed of 5 km/h from right-to-left to cross the street. The pedestrian becomes visible for the ego-vehicle after he passed the Audi e-tron (2). Figure 3 illustrates the positioning of all vehicles and the pedestrian at this time.

The ego-vehicle immediately engages an emergency brake and comes to a standstill in front of the child. At this point the pedestrian is at the 50 % overlapping point. The child continues crossing the street and the scenario ends as soon as the child completely passed the ego-vehicle.

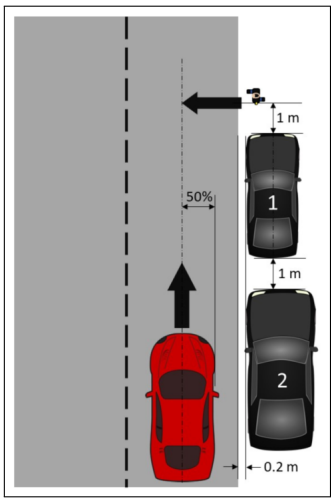

Fig. 3. Target placement based on CPNC-50 Scenario Child [7]

B. Creating objects list of GT data

The GT Objects List includes four message vectors for every spawned object-,Classification, Dimension, Features and Geometric shown in fig. 1. These messages are used to classify the objects, send their geometrical dimensions, location, acceleration, angles and visible edges [1]. The Classification parameters indicate the type of the spawned objects and differentiate between the relevant classes car and pedestrian. In addition, the Features vector contains all visible and invisible edges of the objects. The evaluation of both messages is statically generated for this scenario referring to the bounding box data of every spawned object. The third message represents the length, width and height of the object. This Dimensions vector receives the information as well from the bounding box. Furthermore, the Geometric message is used to represent the coordinates, velocity, yaw angle and acceleration of the objects relative to the ego-vehicle. The GT data will be published via ROS in two different topics. Every topic includes a header with timestamp, object identifier (ID) as well as an Objects List message. This message includes all four messages vectors mentioned before for the pedestrian and both parked vehicles in topic one and is referenced to the center of the objects. Figure 4 illustrates the coordinate system used for the ego-vehicle (bottom right) and the two parked vehicles and the pedestrian (center).

Topic two includes only the Objects List message with the Geometric data of the ego-vehicle based on the camera position at the front middle of the ego-vehicle. The test scenario offers two options for publishing different data in topic one.

- Publishing only objects in the field of view of the camera (200 m and a total opening angle of 60°)
- Publishing all spawned objects over the whole test period

C. Creating objects list of camera data

1) Object detection and preparation: The first problem to deal with is to detect any possible object in every given frame. For that PyImageSearch published an useful version of a detection algorithm called YOLO [8]. Changes in the code had to be made to use multiple frames instead

of just one saved image stored on disk. These frames of the simulation environment can be generated by in-game sensors like RGB and depth camera. It is necessary to place those cameras at the same spot on the ego-vehicle to collect comparable images without any errors by considering angular misalignment. As a result, the camera sensors will create images of 32-bit BGRA colors to work with. Because the YOLO algorithm needs at least 0.35 seconds on every tested hardware the tick rate has to be synchronized and reduced to 0.5 seconds for both sensors.

To generate predominantly True Positive (TP) cases, the confidence value of YOLO is set to 0.7 and the threshold value to 0.6. Furthermore, tests in CARLA have resulted in False Positive (FP) cases where objects such as umbrellas were detected. To exclude these cases, all irrelevant object classes are filtered out in advance. The bounding boxes are not used as usual to display them in the frame, but the pixel coordinates of the bounding boxes are used. These are composed of a x and y coordinate, as well as a width w and height h to determine the location of the detected object in the frame. In addition, the confidence and name of the detected objects are also used.

2) Data processing: The resulting pixel coordinates of the bounding boxes can be used to cut out a frame of the specific object from the image. This is necessary to filter unsuitable pixel values and general noise in the image with an adaptive threshold filter. This filter also generates a black and white image that makes it possible to detect the silhouette of the given object. An evaluation of the blackened pixels and their resulting distances gives an approximate idea of the location in the simulation environment. Therefore, the CARLA development team provides a formula (eq. (1)) to calculate the distance by using the color values of certain pixels of the depth image seen in fig. 5 [9].

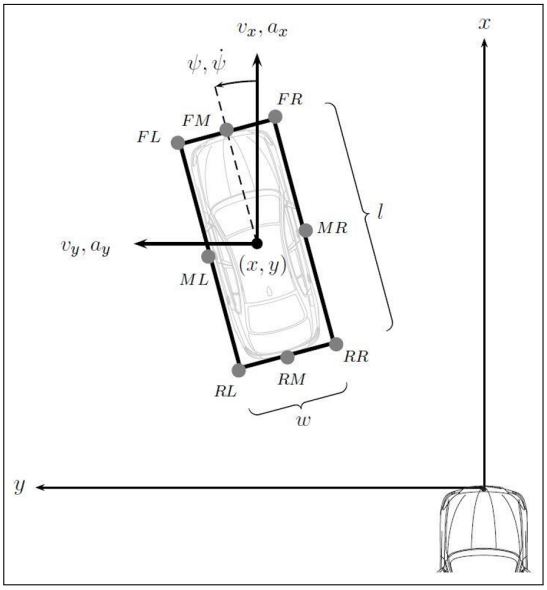

Fig. 4. Vehicle and pedestrian coordinate system [1]

$$
\begin{aligned}
&(R,G,B) = v_{Pixel} \\
&d_{norm} = (R + G*256 + B*256*256) \,/\, (256*256*256 - 1) \\
&d_{direct} = 1000 * d_{norm}
\end{aligned}
\tag{1}
$$

v_{Pixel}	=	color values of specific pixel
d_{norm}	=	normalized displacement
d_{disp}	=	displacement of camera sensor and object

This makes it possible to determine not only the direct distance of any visible object, but also the length of the object itself. By processing the calculated distances in a frame it is possible to estimate the length. To get accurate results at this early stage the object has to be fully visible. Otherwise the length is an approximated estimation. This value is added to the previously calculated distance to obtain the center of an object. To get the final position in the simulation environment taking into account the rotation of an object, the yaw angle must be considered, too.

3) Object Tracker: Over time, new objects become visible and some disappear. For now, the detection algorithm cannot track them. In addition, depending on the movement, objects are not detected in the same order. To keep track of all objects it is necessary to identify already detected ones in the following frames. To do so, all bounding boxes must get connected with an unique ID. For this purpose a function made available by PyImageSearch is used [10]. This tracker was chosen for the reason that it is just necessary to provide coordinates of the bounding boxes. The code recognizes the movement and links an ID to a specific object in the bounding box.

4) Generating an objects list: From the detected coordinates x and y and the corresponding distance, the distance, velocity, acceleration, yaw angle and yaw rate of the object can be calculated. For this, previous values are stored and assigned to an ID. With the proportionality of the object

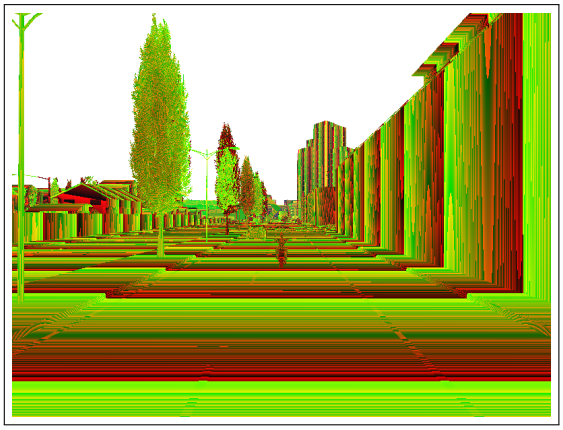

Fig. 5. Image codifying depth value via RGB color space

distance from the vertical center line and the camera angle, the script scales in metric distance. Equation (2) divides the image into two segments (left and right segment) and calculates the distance d_{Pixel} of the object to this center axis.

$$d_{Pixel} = y_{Pixel} - \frac{w_{Pixel}}{2} \tag{2}$$

d_{Pixel} = distance from center line to object
y_{Pixel} = y-coordinate of the object
w_{Pixel} = pixel format of the video horizontal

In eq. (3) a factor k_{angle} is calculated which corresponds to a value of -1...0 or 0...1. This factor indicates the object position relative to the total segment width.

$$k_{angle} = \frac{d_{Pixel}}{w_{Segment,Pixel}} \tag{3}$$

k_{angle} = proportionality factor for distance and angle
$w_{Segment,Pixel}$ = pixel width of segment

The field angle of view of the camera which corresponds to 90°, is used. In relation to a segment, it would be 45°. This is calculated with the factor, to get an angle ϕ related to the vertical axis (see eq. (4)).

$$\phi = k_{angle} * \alpha_{Segment} \tag{4}$$

ϕ = angle from center line to object
$\alpha_{Segment}$ = angle of view for a segment (here: 45°)

By using the angle and the direct distance, the trigonometric theorems can be used to calculate the x- and y-distance in meters (shown in fig. 6) which are the basis for the calculation of velocity, acceleration, yaw angle and yaw rate. The object attributes are stored over several frames to calculate time-dependent values. Velocity is calculated by the change of the x- and y-distances, the acceleration, by changing the x- and y-velocities. The yaw angle with the

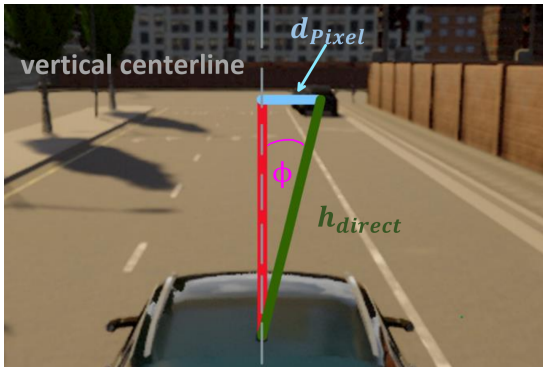

Fig. 6. Calculation of angle

velocity in x- and y-direction and the yaw rate is calculated by time changing angles. The calculated values are related to the ego-vehicle. The dimension of the object is based on eqs. (2) to (4) as well. Angle ϕ is used to calculate the distance y in meters from the center vertical axis for the position of the vehicle on the y-axis in eq. (5).

$$y = h_{direct} * sin(\frac{\phi * \pi}{180}) \tag{5}$$

y = distance from center line to object in meters
h_{direct} = direct distance to object in meters

After calculating the distance of the object to the center axis of the image, it is divided by the pixel distance of the object according to the center axis. This gives a meter per pixel value k_{MpP} to convert the detected object dimensions in pixels to meter (eq. (6)).

$$k_{MpP} = \frac{y}{d_{Pixel}} \tag{6}$$

k_{MpP} = factor meter per pixel for scaling

With the ROS framework, the object data list is transferred via the ROS node /data_play in form of a ROS message (see fig. 7).

D. Visualization of objects lists

The published topics of GT data and Camera-Calculation data are subscribed by ROS node *Object_Visualization*. Each topic contains the ego-vehicle data and the specifically generated objects list. In RVIZ the objects are represented by primitive figures with the help of marker messages. Figure 7 shows the used topics and nodes. Rectangles represent topics and ellipses the called nodes. Moreover, tf is a package which controls the coordinate relationship of the ego-vehicle. Marker messages are described with specific properties such as position, scale, type, color, orientation. Each object classification is assigned to one specific shape and color so that they can be differentiated in RVIZ. The display variants for the possible object classes are shown in table II.

TABLE II
CLASSIFICATION ASSIGNMENT

Classification	Shape	Color [RGB]
car	cube	[1, 0, 0]
truck	cube	[0, 1, 0]
pedestrian	cylinder	[0, 0, 1]
motorcyle	cube	[1, 0, 1]
bicycle	cylinder	[1, 1, 0]
stacionary	sphere	[0, 1, 1]
other	sphere	[1, 1, 1]

In addition, the yaw angle of the objects has to be transformed into a quaternion for the visualization in RVIZ.

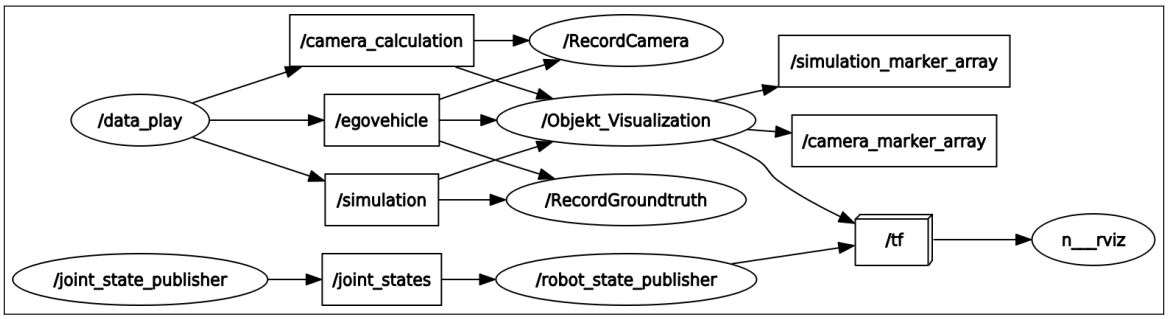

Fig. 7. Nodes (ellipses)/ Topics (rectangles) in ROS

The RGB alpha value of all markers for the calculated camera data is set to 0.5 so that the difference between camera data and GT data is visually recognizable. The highest detection probability of an object indicates the classification so that the marker message properties can be set to the corresponding values of table II. Furthermore, each detection position is mirrored on the Y-axis, because the vehicle coordinate system does not match the RVIZ coordinate system. Finally, the generated markers are combined into a marker array and published. The ego-vehicle is described as URDF model according to [11]. Furthermore, the model can be moved and rotated in the RVIZ coordinate system by tf messages. The published topics of GT data, Camera-Calculation data and ego-vehicle data are also saved in a Rosbag file. Each file contains the published ego data and the corresponding objects lists. In the following, these files are used for post-processing.

E. Evaluation of objects lists - Post-processing

To evaluate the quality of calculated sensor data with the aim to improve the object detection algorithm post-processing is necessary. After recording objects lists data streams in Rosbag files there is the possibility to analyze data in different ways by multiple post-processing functions. Either a single Rosbag file can be analyzed or two files can be compared. In both cases the data is evaluated frame by frame.

1) Basic analysis: Considered to one single Rosbag file specific attribute values of single objects which are selected by their object ID can be displayed. The variety of available attribute types is described in section III-B. In addition, the number of detected objects can be visualized.

2) Advanced analysis: Regarding two Rosbag recordings further analysis methods for comparing the streams are provided. In this case a common time base needs to be generated. To avoid errors because of time variation of both recordings following mapping algorithm is executed. Each frame time stamp is handled as time relative to its Rosbag start time in milliseconds. To every frame in the Rosbag file which is provided by sensor data a frame of simulation data is dedicated. The simulation frame to choose is the latest past frame in relative stream time. The principle of frame mapping is shown in figure 8. With this mechanism pairs

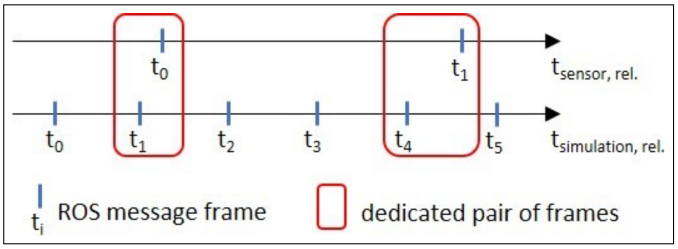

Fig. 8. Principle of frame mapping algorithm

of frames are generated (sensor frame with corresponding simulation frame).

A fundamental use case in analysing the recorded Rosbag files is to evaluate the quality of sensor data in comparison with GT data through metrics introduced in [12]. To determine whether a given camera object is evaluated as TP, FP, False Negative (FN) or a mismatch (mm), the **Intersection over Union (IoU)** value is used. In general there is a list of m camera objects (B_{pr}) and a list of n GT objects (B_{gt}) for each frame. To evaluate a frame, for each combination of GT object and camera object the IoU value is calculated. All those values build a matrix like shown in table III.

TABLE III
IOU-MATRIX FOR A SINGLE FRAME

$IoU(B_{gt,1},B_{pr,1})$	$IoU(B_{gt,2},B_{pr,1})$	...	$IoU(B_{gt,n},B_{pr,1})$
$IoU(B_{gt,1},B_{pr,2})$	$IoU(B_{gt,2},B_{pr,2})$	...	$IoU(B_{gt,n},B_{pr,2})$
...	...	...	...
$IoU(B_{gt,1},B_{pr,m})$	$IoU(B_{gt,2},B_{pr,m})$	...	$IoU(B_{gt,n},B_{pr,m})$

After calculating the matrix the objects in one single frame can be evaluated. A given camera object $B_{pr,i}$ is ...

... **FP** if there is no value

$$IoU(B_{gt,k},B_{pr,i}) > t \quad \text{with} \quad k \in \{1,...,n\} \tag{7}$$

in the according row of the matrix which is greater than the given threshold t.

... **FP** if there is one or more IoU values in the according row greater than the threshold, but for every $B_{gt,k}$, for which

expression eq. (7) is true, there is another $B_{pr,j}$ ($j \neq i$) which matches with $B_{gt,k}$ and $IoU(B_{gt,k}, B_{pr,j}) > IoU(B_{gt,k}, B_{pr,i})$

... a **mismatch (mm)** if it is no FP case, but none of the found possible matching $B_{gt,k}$ has the same class as $B_{pr,i}$.

... **TP**, also called a match, if none of the other mentioned cases are detected. That means, that there is at least one $B_{gt,k}$ which fulfills expression eq. (7) and has the same classification as $B_{pr,i}$ and there is no other $B_{pr,j}$ which matches better with the found $B_{gt,k}$.

Going through the rows of the matrix, for each $B_{pr,i}$ in the given frame it can be decided, whether the case is TP, FP or mm.

The other way round, examining the GT objects $B_{gt,k}$, that means the columns of the calculated matrix, all FN cases can be detected. It is an **FN** if there is no $B_{pr,i}$ for which

$$IoU(B_{gt,k}, B_{pr,i}) > t \quad \text{with} \quad i \in \{1, ..., m\} \tag{8}$$

Going through the columns of the matrix, this decision can be made for every $B_{gt,k}$.
With these steps, a given frame with m camera objects and n GT objects can be investigated.

These functions, one for investigating the camera objects and one for detecting all FN cases, were realized in Python. The calculation of an IoU value is processed with functions of the package *shapely* [13]. First, the given objects which are defined through their properties x, y, *length*, *height*, *yaw* and *classification* like presented in [1] are transformed into bounding boxes. With two of these bounding boxes *shapely* can calculate the intersection area and the union area so that the IoU value can be processed.

As a result of the frame evaluation functions following data quality parameters according to [12] can be calculated:

- recall per frame
- precision per frame
- FPPI per sensor data stream
- MOTA per sensor data stream
- MOTP per sensor data stream

Another feature of the post-processing application is the analysis of deviations by calculating differences of specific attribute values between two recorded data streams. For that matter only TP cases of IoU evaluation are regarded and the concerning object is selected by its object ID in the simulation data record. The difference value results from

$$difference = value_{simulation} - value_{sensor} \tag{9}$$

3) Graphical User Interface (GUI): To investigate the quality of the processed camera object data, a GUI is provided. It is designed with Python's binding package for Qt (PyQt5) [14] and defined as a plugin for *rqt*, a ROS framework for GUI development [15]. With this plugin, the user can import two Rosbag files, one GT data file and one sensor (camera) data file. By using the functions mentioned before, the GUI can show several data graphs to the user like raw data plots, comparing plots with object data of both files or evaluation data. Along with every data set - except from FPPI, MOTA and MOTP - the mean value and the standard deviation for each data set is portrayed in the GUI.
Apart from data plots the interface can also show quality parameters for the whole camera data Rosbag file in an extra widget. For each operation where IoU calculation is needed, the user can set the threshold value for the evaluation.

IV. Results

The performance of processed camera data can be evaluated with metrics presented in [12] which are realized like introduced in section III-E.2. The metrics are validated by comparing two identical Rosbag files. This ensures that the implementation of the evaluation functions are correctly. Considering the objects lists created by data processing it is noticeable that the camera data is quite different from the GT data. Because the simulation is subject to near real-time requirements, a fast processing time is essential. The evaluation and tracking of camera data needs at least 1 second, hence is not fast enough to deliver comparable results. This delay leads to a perceptible offset in the visualization, therefore it is hardly possible to measure the precision of YOLO and the calculation. Figure 9 shows the distance of an object in x direction from GT data and camera data. Moreover, this figure makes the time difference between GT and camera data very clearly. The GT stream ends sending after about nine seconds, because the used sensor suite cannot recognize an object anymore. Certainly, the camera data sends the comparable object list after another two seconds, so that a correct evaluation can not be carried out. Furthermore, minor errors occur with slightly misplaced bounding boxes by YOLO and detecting the object precisely within the given bounding box. This influences following values like calculation of the object length and object height.

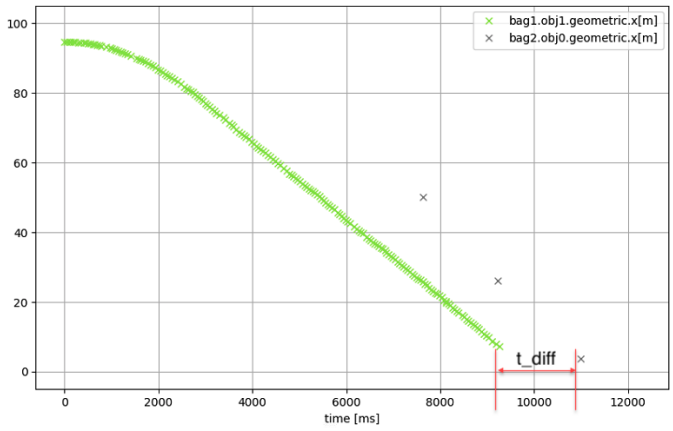

Fig. 9. Post-processing GT (green)/ Camera Data (gray)

V. Conclusions

Nowadays, urban driving simulation software has a major impact on the development process of new autonomous driving functions. Due to the constant further development of the range of functions, the adaptable API, ROS integration and as well the rapidly growing community, open-source simulators like CARLA offer excellent opportunities for testing and evaluating individual sensor suites and functions. Freely adaptable scenarios and weather conditions offer the developer a lot of flexibility when creating their own test cases.

In this specific scenario a real-time object detection system combined with an object tracker are used. As said in section III-C, the data processing time is highly dependent on the hardware on which the calculation runs. The delay mentioned in section IV between the time of recording and visualization leads to outdated results in RVIZ. This is the most urgent problem to be solved. Further improvements can be achieved with training the YOLO algorithm. Synchronizing the published objects lists data by considering the time stamps of the calculated data as well as the GT data should also lead to more comparable data. However, this only will be useful if the acquisition time by YOLO and the code itself will improve.

References

[1] M. Aeberhard, "Object-level fusion for surround environment perception in automated driving applications," Ph.D. dissertation, Technische Universität Dortmund, Dortmund. [Online]. Available: https://d-nb.info/113647157X/34

[2] ODSC - Open Data Science, "Overview of the yolo object detection algorithm." [Online]. Available: https://medium.com/@ODSC/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0

[3] F. Reway, A. Hoffmann, D. Wachtel, W. Huber, A. Knoll, and E. Ribeiro, "Test method for measuring the simulation-to-reality gap of camera-based object detection algorithms for autonomous driving." [Online]. Available: https://www.researchgate.net/publication/341494584_Test_Method_for_Measuring_the_Simulation-to-Reality_Gap_of_Camera-based_Object_Detection_Algorithms_for_Autonomous_Driving

[4] D. Alexey, R. German, C. Felipe, L. Antonio, and K. Vladlen, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[5] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org

[6] EUROPEAN NEW CAR ASSESSMENT PROGRAMME (Euro NCAP), "Aeb vru test protocol v3.0.2," JULY 2019. [Online]. Available: https://cdn.euroncap.com/media/53153/euro-ncap-aeb-vru-test-protocol-v302.pdf

[7] Insurance Institute for Highway Safety, "Pedestrian autonomous emergency braking test protocol (version ii)," FEBRUARY 2019. [Online]. Available: https://www.iihs.org/media/f6a24355-fe4b-4d71-bd19-0aab8b39aa7e/TfEBAA/Ratings/Protocols/current/test_protocol_pedestrian_aeb.pdf

[8] A. Rosebrock, "Yolo object detection with opencv." [Online]. Available: https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

[9] "Depth camera." [Online]. Available: https://carla.readthedocs.io/en/latest/ref_sensors/#depth-camera

[10] A. Rosebrock, "Simple object tracking with opencv." [Online]. Available: https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/

[11] Open Source Robotics Foundation, Inc., "car_demo." [Online]. Available: https://github.com/osrf/car_demo

[12] F. Reway, A. Hoffmann, D. Wachtel, W. Huber, A. Knoll, and E. Riberio, "Test method for measuring the simulation-to-reality gap of camera-based object detection algorithms for autonomous driving."

[13] S. Gillies, "Shapely documentation." [Online]. Available: https://pypi.org/project/Shapely/

[14] Riverbank Computing Limited, "Pyqt5." [Online]. Available: https://pypi.org/project/PyQt5/

[15] D. Thomas, D. Scholz, and A. Blasdel, "Ros rqt." [Online]. Available: http://wiki.ros.org/rqt