Automated Driving in Mixed Traffic Conditions

Academic Institution:
Technische Hochschule Ingolstadt

Course name:
Autonomous Vehicle Engineering

Group:
AVE-6

Project Supervisor:
Maikol Funk Drechsler

Project prepared by:
Carolina Paez
Evgeniia Louwe Kooijmans
Ibrahim Elsayed Fared Ibrahim Khalil
Jemish Ghoghari
Lukas Kramschuster
Ramez Nawras Bashir Alghazawi
Rushan Abdrakhimov
Vishal Balaji

***Abstract*— As automated driving technology continues to advance, the need for comprehensive testing and evaluation of autonomous vehicles becomes increasingly critical. Evaluating the performance of sensors and image recognition algorithms in real-world scenarios poses challenges due to the high costs, safety concerns, and limited repeatability of physical tests. To address these limitations, researchers are turning to virtual environments as a promising alternative. This paper explores the concept of creating virtual environments that can effectively replace the real environment for testing and evaluating autonomous vehicles, with a particular focus on sensor testing and image recognition algorithms.**

INTRODUCTION

The main goal of this project is to retrieve data from both virtual and real-world environments, employ machine learning algorithms to process the collected data, and ultimately define the Operational Design Domain (ODD). The Carla simulator and ROS bridge are essential tools for creating data from virtual environments. The Carla simulator provides a realistic virtual environment for simulating driving scenarios with a high degree of customization to defined requirements. By connecting with the Robot Operating System (ROS) through the ROS bridge, researchers can replicate real-world sensor to a particular extent.

Data generation from the virtual environment involved several key steps to ensure accurate and comprehensive results. Firstly, the virtual environment was created. Next, sensor models were set up to replicate the behavior and characteristics of real-world sensors, ensuring a realistic data collection process. Finally, specific scenarios were defined within the virtual environment. During these scenarios, data recording was initiated. By following these steps in data generation from the virtual environment, the collected data could be effectively compared to real-world data obtained from similar scenarios.

In order to assess the feasibility of replacing the real-world environment with a virtual counterpart, the chosen DD3D model was employed for image recognition. Utilizing machine learning algorithms, the collected data from both the real-world and virtual environments underwent processing and analysis. By conducting a thorough comparison between the two datasets, this evaluation aimed to determine the extent to which the virtual environment could adequately substitute the real-world environment. Additionally, this analysis sought to assess the degree of compatibility and accuracy achieved by the virtual environment in replicating real-world scenarios. Such insights are crucial for understanding the potential of virtual environments in testing and validating sensors and image recognition algorithms, providing valuable guidance for the development and deployment of automated driving systems in mixed scenarios.

Working with the real-world environment enabled the project to accomplish two key objectives. Firstly, it involved gathering crucial data, including images and point cloud data, which served as a benchmark for defining the Operational Design Domain (ODD). This data played a fundamental role in establishing the parameters and criteria for the ODD. Secondly, the real-world environment facilitated the collection of calibration data from various sensors, ensuring accurate and reliable measurements for subsequent analysis and evaluation.

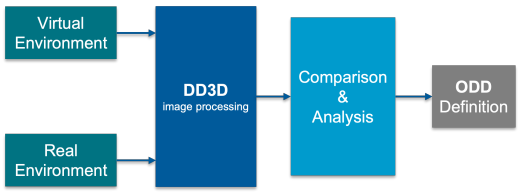


Fig. 1: Project overview.

The product's structure involves obtaining data from two sources: recorded ROS bags from Carla simulations and real-world scenarios. The Carla and ROS bag node interfaces with the Carla server to control it and retrieve the necessary data, which is then visualized in RVIZ. Additionally, the data is sent to the DD3D node for processing, which generates marker arrays that are also visualized in RVIZ. In parallel, real-world ROS bags are sent to RVIZ and the DD3D node for visualization and processing. Ultimately, RVIZ displays the source data from the ROS bags alongside modified images featuring marker arrays, providing a comprehensive visualization of the collected data and the results of the processing steps.

The project work was divided into four distinct parts, each forming a dedicated team with individual responsibilities and tasks.

Team 1 - project management:

- Rushan Abdrakhimov

Team 2 - virtual environment (Carla and ROS bridge):

- Evgeniia Louwe Kooijmans
- Jemish Ghoghari

Team 3 - image recognition (DD3D):

- Vishal Balaji
- Carolina Paez

Team 4 - real-world environment:

- Lukas Kramschuster
- Ramez Nawras Bashir Alghazawi
- Ibrahim. Elsayed Fared Ibrahim Khalil

I. Real-world environment

The team 4 was responsible for real-world data acquisition and sensors calibration. This involved extensive utilization of MATLAB software to perform the necessary activities and tasks outlined in the project.

A. Hardware Assembly

The used sensors must be positioned exactly as being installed in a car; therefore, we built a metal frame.

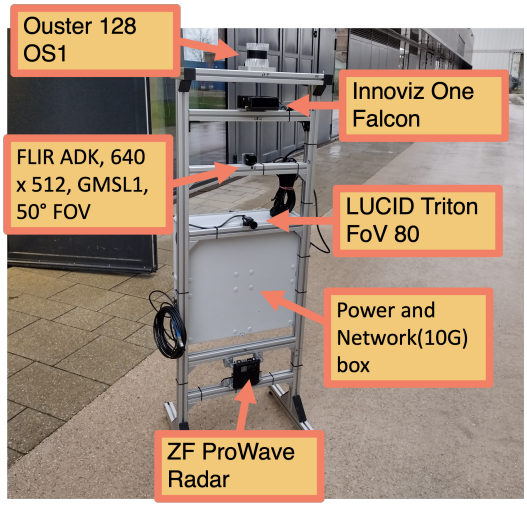


Fig. 2: Sensors rack.

B. The used sensors

Perceiving the surrounding environment of a car requires a combination of sensors that provide different types of information. Here are some common sensors used in modern vehicles:

1) Cameras (FLIR ADK Thermo camera & Lucid Vision Labs - Titron GigE): Vision-based systems, including monocular or stereo cameras, are widely used to capture visual information. Cameras provide high-resolution images that enable object detection, lane detection, traffic sign recognition, and pedestrian detection.

2) LiDAR (InnovizOne Lida MEMS & Ouster OS1 - 128 channels): LiDAR sensors emit laser beams and measure the time it takes for the laser to bounce back after hitting an object. This technology provides precise 3D point cloud data, allowing for accurate object detection, localization, and mapping.

3) Radar (ZF pro Wave Radar): Radar sensors use radio waves to detect objects and measure their distance, speed, and direction. Radar is particularly useful in adverse weather conditions and provides robust detection capabilities for objects such as vehicles, pedestrians, and cyclists.

C. Data aquisition and selection

To initiate the calibration process, a high-quality dataset is required. The acquisition of this data involylves the selection of appropriate samples and the alignment of corresponding images with the point cloud.

1) Data aquisition: The data was recorded in ROSBAG format, necessitating the extraction of both images and the point cloud from the ROSBAG file. To utilize the MATLAB Lidar Camera Calibrator App, the images needed to be in the .PNG format, while the Lidar output required the .PCD format.

2) Image selection: To ensure a high-quality calibration outcome, a selection process was employed to choose 50 images and corresponding PCD files. During the data selection stage, attention must be paid to several factors. For the images, it is crucial to exclude those exhibiting flickering, which often arises due to issues during data collection. Additionally, images should be chosen where the checkerboard is situated at an appropriate distance to ensure high-resolution detection of all the squares on the checkerboard.

Fig. 3: Image with flickering.

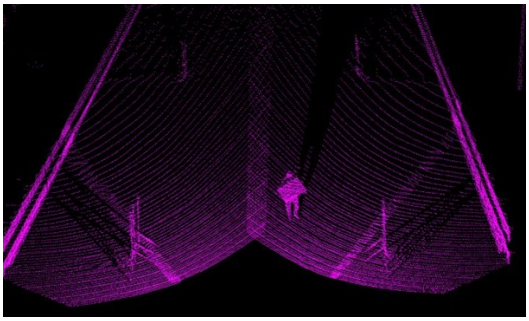


Fig. 4: Point cloud with overlapping beams.

3) Point cloud selection: Regarding the Lidar data, the checkerboard should not be positioned too closely, as the Lidar utilized is long range and could introduce inaccuracies. Furthermore, the checkerboard should not be aligned directly in front of the Lidar, as this results in overlapping beams, leading to further inaccuracies.

4) Matching files: Due to disparities in the start times of Lidar and camera data recording, as well as individual recording inaccuracies, the files do not align perfectly. Therefore, it becomes necessary to identify the corresponding counterparts for each file. To match the

images with their respective point clouds, consideration must be given to the timestamps. The individual PCD files already possess their timestamps in Unix format. To obtain the timestamps for the images, a list was generated from the ROSBAG file, associating each image's name with its timestamp. Subsequently, the timestamps of the selected images could be utilized to search for the matching PCD file. Although both the Lidar and camera operated at a frequency of 10Hz, slight deviations in the recording frequency caused the timestamps to not align precisely. Therefore, a higher accuracy than 1/10s (10Hz) must be taken into account. The timestamp accuracy was considered up to the second decimal place, resulting in a resolution of 1/100 second or 100Hz.

Example:
(1) 1683272247.297495000.pcd
should be matched with its counterpart (2) or (3).
(2) 683272247.307495000.png
(3) 683272247.207495000.png
If a resolution of 1/10s is used, example (3) deviates significantly more than example (2) but still would be chosen. With a resolution of 1/100s example (2) would be chosen, which is the better solution.

The recordings from both cameras were synchronized and had identical names, eliminating the need for a matching process between the RGB camera and the infrared camera. The synchronization ensured that the captured data from both cameras corresponded to the same timeframe.

5) renaming files: To utilize the Lidar Camera Calibrator App, it is necessary to assign matching images and their corresponding counterparts the same name.
Example:
001.png and 001.pcd

D. Intrinsic Calibration Using the Camera Calibration App in Matlab

In the early stages of our project, after we had secured the RGB camera and the LiDAR onto an aluminium frame, we set out to perform intrinsic calibration. This step was vital because it helped us decipher the unique attributes of our camera, such as its focal length, the principal point, and the lens distortion coefficients. These parameters are the backbone for tasks like correcting image distortion, transforming pixel coordinates to real-world coordinates, and carrying out other image processing tasks.

E. Checkerboard Image Acquisition

We initiated the process by taking a series of images of a checkerboard pattern from a range of angles using the RGB camera. The checkerboard pattern acted as a known geometric structure, offering a reference point for the calibration algorithm. Capturing images from various angles ensured that the calibration algorithm had a wide range of data to work with, thereby enhancing the accuracy of the calibration outcome.

F. Calibration Using Matlab's Camera Calibration App

After we had loaded these images into Matlab's Camera Calibration App, the software took over and automatically pinpointed the checkerboard corners in each image. This detection was made possible through an advanced corner detection algorithm, which spots the intersection points of the checkerboard squares. These intersection points, also known as feature points, are crucial for the calibration algorithm as they offer a reference for the geometric relationship between the 3D world and its 2D representation in the image.

G. Calibration Algorithm and Estimation Process

The calibration algorithm, which is based on the pinhole camera model, then estimated the camera parameters by minimizing the difference between the observed positions of the feature points in the image and the positions predicted by the camera model. This process was carried out in several steps. Initially, the algorithm made an estimate of the camera parameters. These parameters were then iteratively refined to minimize the reprojection error, a process achieved using optimization techniques such as the Levenberg-Marquardt algorithm. This iterative process ensured that the resulting camera parameters were as accurate as possible.

H. Outcome of Calibration

The outcome of this calibration process was a set of intrinsic camera parameters, encapsulated in a matrix. This matrix was extracted for further use in our project, serving as a fundamental building block for our subsequent work.

I. Extrinsic Calibration Using the LiDAR Camera Calibration App

In our project, a primary focus was the process of extrinsic calibration. This step is critical to determine the position and orientation of a camera in the world, which is a requirement for several applications like 3D reconstruction, object tracking, and any tasks requiring an understanding of the camera's spatial relationship with its environment.

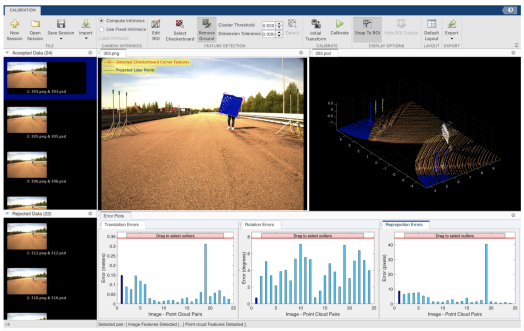


Fig. 5: LiDAR Camera Calibration App in Matlab

J. Extrinsic Calibration Process and Tool

The tool we used to achieve this was the LiDAR Camera Calibration App in Matlab, specifically designed to facilitate this process. To perform the calibration, we prepared a set of LiDAR point cloud data (PCD files) corresponding to a checkerboard pattern. This set was meticulously aligned with images captured by our RGB camera, a process that demanded precision, as even minor misalignments could substantially skew the calibration results.

K. Types of LiDAR Sensors

For our project, we performed extrinsic calibration with two types of LiDAR sensors: 1) the Innoviz LiDAR and 2) the Ouster LiDAR. The same RGB camera was used with both LiDAR sensors, ensuring a consistent imaging setup.

L. Calibration Algorithm and Estimation Process

The LiDAR point cloud gives a 3D representation of the scene, which, when combined with the 2D images from the RGB camera, enabled us to estimate the camera's extrinsic parameters. The calibration algorithm works by establishing a transformation that best aligns the 3D points in the point cloud with their corresponding 2D points in the image. The optimization algorithms, such as the Levenberg-Marquardt algorithm, facilitated the estimation of rotation and translation to minimize the differences between the observed 3D points in the point cloud and the 3D points predicted by the camera model.

M. Outcome of Calibration

The outcome of this process was a set of extrinsic parameters, encapsulated in a matrix form. This matrix is a vital component as it represents the spatial relationship between the sensor and the cameras in our setup. It allows us to understand how our camera perceives the world, thus enabling us to perform tasks that require knowledge of the camera's spatial relationship with the world.

N. Extrinsic Calibration between the Infrared Camera and the RGB Camera

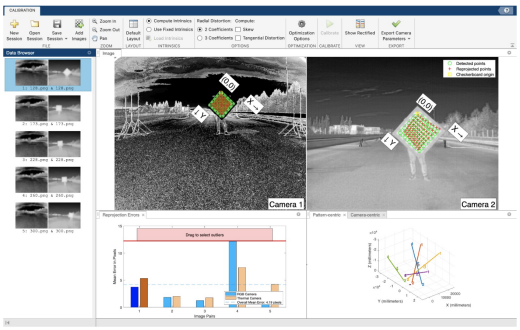


Fig. 6: Stereo Camera Calibrator in Matlab

Furthermore, an extrinsic calibration was also executed between the infrared camera and the RGB camera. To achieve this, we employed the Stereo Camera Calibrator in Matlab, another highly specialized tool designed for this purpose. This additional calibration is essential in our project, providing a comprehensive understanding of the alignment and spatial relationship between the infrared and RGB cameras, each contributing distinct data to our overall sensor suite.

O. Conclusion

In conclusion, the process of extrinsic calibration was a cornerstone in our project. The complexity of these tasks was adeptly managed with the powerful tools provided by Matlab, including the LiDAR Camera Calibration App and the Stereo Camera Calibrator. These applications enabled us to carry out the calibrations accurately and efficiently, enhancing our understanding of how our cameras perceive and interpret their surroundings. This information is invaluable for any tasks that require precise knowledge of the cameras' spatial relationship with the world.

II. Virtual Environment

Within this project, Team 2 was responsible for developing a simulated version of the test scenarios. To accomplish this, the CARLA simulator [1] and its ROS bridge [2] interface were used, and the activities described in this section were carried out. In addition to the primary tasks associated with the stated objective, some supplementary activities to support Team 3 were conducted.

A. Research and Setup

Due to the fact that initial knowledge required to accomplish the project's objectives was limited, the first weeks of the project were devoted to extensive research and gaining practical experience with essential tools and technologies such as the CARLA simulator, ROS and RVIZ. This period allowed the team to familiarize themselves with these tools and build a solid foundation for the subsequent phases of the project.

Furthermore, during the project's initial phase, the CARLA simulator and the ROS bridge had to be installed on the dedicated system. Afterward, the working principles of the ROS bridge were investigated. Throughout the project, the ROS bridge was not directly modified but rather expanded by incorporating the necessary launch and configuration files.

The team created the main launch file based on the existing file *carla_ros_bridge_with_example_ego_vehicle.launch*. The launch file facilitates the initiation of the ROS bridge node, spawns an ego vehicle with a predefined set of sensors specified in a configuration file, and launches a manual control node. By customizing the existing launch file, the team could efficiently establish the necessary components for the project.

B. Spawning of the sensors

To ensure a realistic replication of the test scenarios in a simulated environment, the virtual sensors must be configured to match the real parameters of the sensors used for the actual tests. In CARLA ROS bridge, the sensors are attached to the ego vehicle during the actor spawning process, which is accomplished by the *carla_spawn_objects* node, utilizing a configuration file defined in JSON format. For this project, a corresponding configuration file was created inside the *carla_spawn_objects* package, and the main launch file was modified to use this file for sensor configuration.

The primary sensor required for data collection in this project was the RGB camera sensor. The sensor was configured with an image resolution of 1936 x 1464 pixel to match the specifications of the Lucid Vision Labs Triton 2.8 MP camera. A camera lens with the field of view of 60° was chosen.

Additionally, a LiDAR sensor was added to enhance the aesthetics of the 3D visualization. The configuration of the sensor was based on the parameters of a real Ouster OS1 LiDAR with 128 channels, which were obtained from a corresponding data sheet [3].

The position of the sensors was chosen in front of the ego vehicle, mimicking the position and the orientation of the real sensors installed on the test frame.

C. Visualization in RVIZ

With the CARLA ROS bridge, the data from CARLA sensors is published into corresponding ROS topics and can be directly visualized with the RVIZ tool. The team configured RVIZ to display the data from the sensors of interest, namely the RGB camera image and the LiDAR point cloud data. Figures 7 and 8 illustrate the visualized data. Both of the sensors were aligned with respect to the ego vehicle coordinate frame. To achieve this, transformations generated by the ROS bridge and published into the */tf* topic were utilized. Subsequently, an additional RVIZ configuration file was created to enable the visualization of the distorted image published by the image distortion node, described in section II-F.

In addition, the team conducted research on publishing *MarkerArray* ROS messages and visualizing these messages in RVIZ. This research aimed to assist Team 3 in visualization of 3D bounding boxes detected by the DD3D model.

Fig. 7: RGB camera image visualized in RVIZ

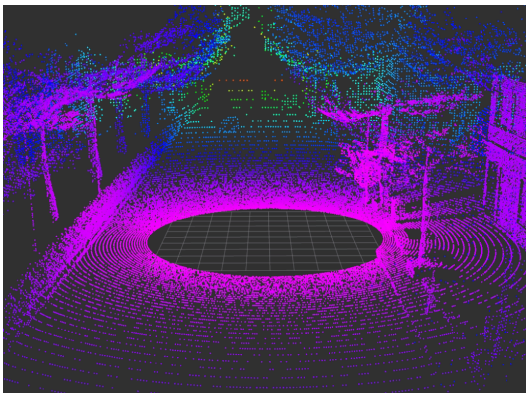


Fig. 8: LiDAR point cloud data visualized in RVIZ

D. Setup CARLA and ROS Bridge on separate Hosts

To avoid potential conflicts and dependencies, for this project, CARLA Simulator and the ROS Bridge were separated by utilizing different host systems. CARLA Simulator was configured on a Windows system equipped with a high-performance GPU, while the ROS Bridge was set up on a Linux system with ample processing power. The goal was to establish communication between the two host systems for seamless data transfer. Two approaches were identified for achieving this communication. The first method involved connecting both host systems via Ethernet and configuring the IP addresses for their respective network interfaces. The second method utilized network-based communication, requiring both hosts to be connected to the same network interface to enable visibility between them. This project adopted the second method since both hosts were on the same network, ensuring a relatively fast data packet transfer rate. Extensive research on network interfaces and IP addresses was conducted to facilitate the setup.

E. Camera calibration in CARLA

In the ROS bridge, the intrinsic camera matrix of an RGB camera sensor is calculated numerically based on the specified field of view, and it is published in a CameraInfo message. Since the object detection algorithm of DD3D model requires a correct intrinsic camera matrix as input, it was decided to calibrate the camera in CARLA to confirm the validity of the published intrinsic parameters.

To begin the calibration process, a model of a chessboard calibration target was created in Blender software using a mesh object of type Grid. The target was colored black and white using the Shading workspace and two materials.

The calibration target was placed inside a CARLA map. CARLA is developed in Unreal Engine 4 (UE4), a Game Engine that provides functionalities for 3D Gaming. The Calibration target was exported from Blender as an FBX model. To bring the Calibration target inside Carla Map, A FBX model can be imported into UE4. After Importing, the Three most essential asset blueprints will be generated automatically, A Skeleton Mesh, A PhysicsAssest, and a Material. Here, The Calibration Target does not have any Physics assigned to it. Skeleton-Mesh is the overall blueprint that shows the final shape of the Object. A Material is an asset that provides body color and texture. The Material is set up for Calibration Target and assigned to it. Since, The Calibration target is the Static Object, A Static Object Blueprint can be generated that is easy to drag and drop into Carla Map. Now, A required position and rotation can set. CARLA needs to be built, and Calibration Target is Visible on the CARLA Map. The camera can detect this Target and can record the Images.

To generate the necessary images for calibration, the CARLA Python API was used. A Python script was developed to spawn a vehicle in front of the calibration target and repeatedly attach a camera sensor to it in various positions and orientations concerning the calibration target. The camera poses were carefully chosen to ensure various locations and orientations of the calibration target within the captured images. The camera sensor's parameters were set to match those used in the CARLA ROS bridge. For each camera position, a calibration image was rendered and saved.

The calibration was performed following the standard camera calibration approach of the OpenCV library [4]. The corners of the chessboard target detected by the *findChessboardCorners()* algorithm are illustrated in Figure 9. Through experimentation, it was observed that removing background objects in the calibration images improved the processing time of OpenCV functions and enhanced corner detection. As a result, an additional background object was added to the CARLA map to conceal the background behind the calibration target as shown in Figure 9.

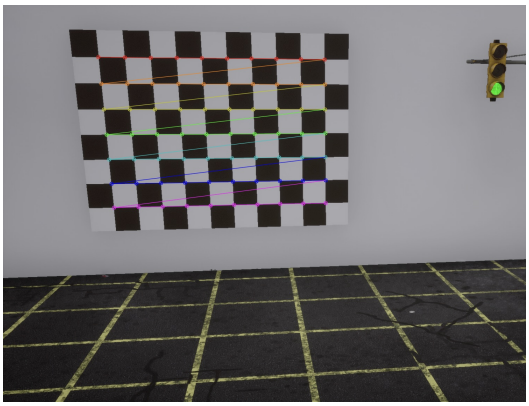


Fig. 9: Chessboard corners of the CARLA calibration target detected by the OpenCV function

The calibration results showed that the RGB camera in CARLA exhibits no lens distortion and that the intrinsic camera matrix is nearly identical to the one numerically calculated by the ROS bridge, with only a slight deviation in the position of the principal point. Overall, it was concluded that the information about camera intrinsic parameters published by the ROS bridge is sufficient, as the differences compared to the parameters determined through calibration were insignificant.

F. Image distortion

A lens of a real camera can introduce lens distortion, which can affect the image. To accurately replicate images captured by a real camera, a simulated camera must exhibit the same lens distortion effects.

Initially, attempts were made to introduce lens distortion directly in CARLA by adjusting the camera lens distortion attributes. However, it was discovered that the existing distortion-related functionality in CARLA was insufficient.

Consequently, the distortion had to be added to images as a post-processing step. This involved applying a distortion algorithm to images after rendering. To accomplish this, an image distortion ROS node was developed as a part of the ROS bridge. The node was included in the newly created ROS package called *admt*, and the main launch file was

modified to start the node simultaneously with the ROS bridge.

The image distortion node subscribes to an RGB camera image topic, distorts the received image, and publishes it in a dedicated ROS topic. Additionally, it generates and publishes a new ROS CameraInfo message, which contains the intrinsic matrix of the camera and distortion parameters, matching the distorted image. The inclusion of the additional CameraInfo topic is necessary, as the correct camera intrinsic matrix and distortion coefficients are required by Team 3.

The node performs image distortion by applying a transformation map to the received image. The distortion map is calculated by inverting the undistortion and rectification transformation map generated by the *initUndistortRectifyMap()* algorithm of the OpenCV library. The camera intrinsic matrix parameters and distortion coefficients are read from a configuration file defined in JSON format. The necessary distortion parameters were determined through the calibration of the real RGB camera.

One problem that occurs when adding image distortion in post-processing is the resulting black frame around the image as shown in Figure 10. This effect occurs due to the missing information, as the pixels that should be mapped to the black pixel's positions are outside of the image bounds. To solve this problem, an additional RGB camera sensor was attached to the ego vehicle in CARLA. This camera sensor has an image resolution extended by 300 pixels in height and width (2236 x 1764 pixels), and an extended field of view of 67.4°, chosen in a way to achieve the same focal length in pixels as the focal length of the original camera. The image received from this sensor was used for image distortion and then cropped to the desired image resolution of 1936 x 1464 pixel. This allows obtaining a distorted image that is almost identical to the one acquired from the original image but without missing information. Such an image is illustrated in Figure 11. The intrinsic camera matrix published in CameraInfo is modified accordingly by adjusting the position of the principle point, as shown by experiments in section II-H.

Overall, this approach allows for the replication of lens distortion effects in simulated images and ensures accurate simulation of real-world camera behavior.

G. ROS bags and traffic generation

Throughout this project, it was essential to provide Team 3 with ROS bags recordings to facilitate the development and testing of the object detection algorithm. The ROS bags needed to capture various traffic scenarios, which required populating the CARLA map with a diverse range of traffic participants, including both vehicles and vulnerable road users. To automate this process, the ROS bridge was extended with the *traffic_generator* package. A ROS bridge compatible node, called *carla_traffic_generator.py*, was developed based on the traffic generation example provided with the CARLA Python API [5]. When executed, this node populates the CARLA environment with pedestrians and vehicular traffic, taking inputs such as the desired number of vehicles and pedestrians.

Fig. 10: Image distorted using the 1936 x 1464 pixel image resolution.

Fig. 11: Image distorted using the 2236 x 1764 pixel image resolution and cropped afterwards.

To incorporate the traffic generator node seamlessly with the ROS bridge, modifications were made to the main launch file, enabling simultaneous execution of the traffic generator node and the ROS bridge.

H. Investigation of the effects of image modifications on camera intrinsic and extrinsic parameters

To enhance the performance of the DD3D object detection model of Team 3, optimizing the image resolution and aspect ratio may be necessary. This can involve image modifications, such as cropping, padding, and resizing. However, these image modifications can impact the camera intrinsic matrix, which is required as an accurate input for the model. To assess the effects of image modifications on the camera intrinsic parameters, a series of experiments involving image modifications and subsequent re-calibration were carried out in Blender. Additionally, the experiments provided an opportunity to examine whether the image modifications have any influence on predicted extrinsic parameters, which describe the relative position of the camera and the objects captured in the image. Blender was chosen as the software for these

experiments since it provides the capability to define a camera with precisely known lens parameters. Furthermore, Blender provides access to ground truth data regarding the extrinsic parameters, as in Blender, the relative position of the camera and the calibration target are accurately known.

The camera intrinsic matrix represents the internal properties of the camera and includes the following parameters:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (1)$$

where f_x and f_y are the focal lengths of the camera lens in pixels, and c_x, c_y are the coordinates of the principle point, calculated form the upper left corner.

To conduct the experiments, the same calibration target was used as described in section II-E. The target was animated for 50 frames to introduce positional diversity of the target in rendered images. The camera object was configured to match the parameters of the RGB camera sensor used in CARLA. The camera calibration was performed using the standard OpenCV approach, and several cases were considered:

1) Original rendered images were used for calibration. The calibration produced expected results, with the obtained intrinsic matrix matching the one calculated by the ROS bridge in CARLA.
2) Calibration was performed on images cropped on the top by $n_{cropped,top}$ pixels. The calibration demonstrated a shift in the position of the principal point, given by $c_{y,new} = c_y - n_{cropped,top}$.
3) Calibration was performed on images padded on left side by $n_{padded,left}$ pixels and on the right side by $n_{padded,right}$ pixels. The calibration revealed a shift in the position of the principal point, given by $c_{x,new} = c_x + n_{padded,left}$.
4) Calibration was performed on images resized by a factor $s = \frac{width\_resized}{width\_original}$. The calibration showed that resizing changes all intrinsic parameters by a factor of s as follows:

$$K = \begin{bmatrix} s \cdot f_x & 0 & s \cdot c_x \\ 0 & s \cdot f_y & s \cdot c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2)$$

Finally, it was determined that none of the image modifications described above affect the predicted extrinsic parameters.

I. Reproduction of the real test scenario in CARLA

To achieve the main objective of the project and generate simulated replicas of the real test scenarios, several activities were carried out.

1) Digital Twin: The provided digital twin has two essential components: the test track map and the 3D models of the Test Agents. The digital twin track Map is stored in a ZIP file that needs to be extracted into the Unreal Engine content folder, which contains all necessary assets. After extraction, Test Map should be visible in Unreal Engine content.

2) Coordinate system: The provided test track coordinate system does not align with the CARLA coordinate system. Upon analyzing the issue, it has been discovered that the coordinate system used in the test track is equivalent to the Cartesian system employed in CARLA. For that, a method for transforming coordinate systems between CARLA maps and Unreal Engine has been identified.

$$\begin{bmatrix} x_{unreal} \\ y_{unreal} \\ z_{unreal} \end{bmatrix} = 100 * \begin{bmatrix} x_{carla} \\ y_{carla} \\ z_{carla} \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} x_{carla} \\ y_{carla} \\ z_{carla} \end{bmatrix} = \frac{1}{100} * \begin{bmatrix} x_{unreal} \\ y_{unreal} \\ z_{unreal} \end{bmatrix} \quad (4)$$

3) Target models: To utilize the test target functionality in the CARLA simulator, it is necessary to import 3D models of pedestrians, bicycle riders, cars, and sprinklers via unreal engine. The unreal engine supports various formats, including FBX, OBJ, and Collada, facilitating the import process. subsequently, the unreal engine automatically generates a skeleton mesh blueprint for each model, which requires the application of textures to achieve the desired appearance, including clothing details. The aforementioned approach entailed thorough research to establish the appropriate linking and structure for the models, ensuring a cohesive final result. To incorporate the models into the CARLA map, two methods were employed: direct drag-and-drop placement onto the map within the Unreal Engine interface or registration of the models as CARLA objects for spawning using a Python script. In this specific project, all objects were registered as CARLA walkers. To register the models as CARLA walkers, CARLA provides a designated directory where crucial model details, such as the blueprint's location, the model's name, and various speed values, are specified. Upon completing the registration process, the models became accessible via the Python script, allowing for their seamless spawning and placement within the CARLA map.

4) Target physics: To make targets detectable by CARLA sensors, which employ distance measurements, all objects registered in CARLA must have a physics setup. The provided targets do not have any physics bone setup replicating their body shape. By doing research, it is founded that the unreal engine provides functionality to add physics bones to the body of the model that enables visibility of the model. To add physics bones, at first, a physics asset blueprint of the model, created automatically by the unreal engine, has to be modified. In physics assets, three options are given to add the shape of the bone that is Box, Cylinder, or Sphere. The position of the bone has been adjusted as per the requirement, which is depicted in Figure 12. After that, the blueprint should be saved. In the next step, the 'generate hit events' option should be ticked, which is found in the skeleton mesh blueprint. In this project, physics bones were added for all target models to make them visible for the LiDAR sensor [6].

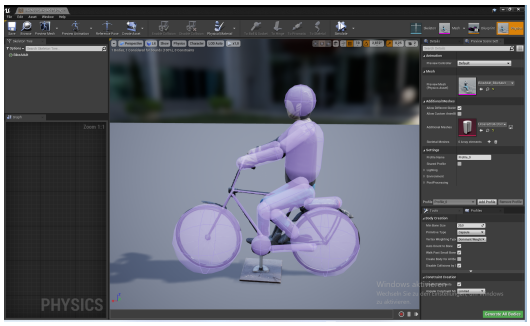


Fig. 12: BikeRider model with Physics bone added.

5) Ego vehicle spawning: The virtual sensors in CARLA needed to be placed at certain coordinates within the digital twin, which correspond to the position of real sensors during the collection of real-world data. To achieve this, the accurate spawn point of the ego vehicle, including its position and yaw rate, was calculated. The calculated spawn point was then incorporated as an argument within the main launch file of the ROS bridge. This enabled the spawning of the ego vehicle and its sensors at the precise coordinates during the initialization of the ROS bridge.

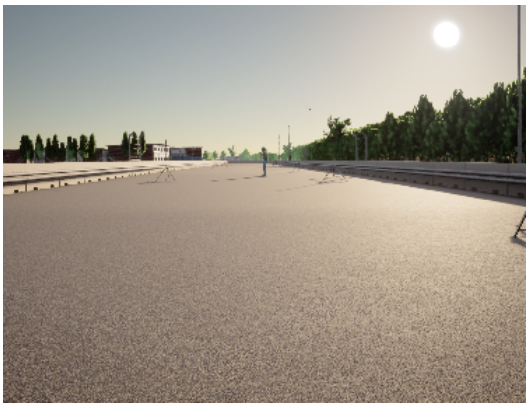
6) Scenario generation: To automate the scenario generation process, a Python script has been developed. This script is capable of locating the ROS bridge ego vehicle and spawning the desired target in front of the vehicle on a straight line at a specified distance. The script allows for adjustment of the target type, target distance, and weather conditions through the use of flags.

It should be noted that the script has not been integrated into the ROS bridge. Instead, it operates through the CARLA Python API and can be executed as needed while the ROS bridge is already running. This approach provides flexibility and allows for generation of new scenarios without the need to restart the ROS bridge.

7) Weather conditions: In the real test scenarios, rainy weather was chosen with a specific rainfall rate of 10 mm/h. To replicate similar weather conditions in the simulated environment, an equivalent amount of precipitation needed to be defined. However, CARLA's existing functionality for rain settings uses a rain intensity scale of 0 to 100 (no rain to heavy rain), which does not correspond to real-world units. Consequently, the rain conditions had to be approximated purely visually in order to achieve the desired rainfall effect.

8) Recording of ROS bags: To store the results of the scenario replication in CARLA, ROS bags were recorded. The results of scenario recreation compared to the real-world test are illustrated in Figure 13. During comparison of the simulated and the real-world scenarios, certain differences were observed.

Firstly, although the RGB camera sensor and the target have been positioned within the digital twin map at the exact coordinates as in reality, in the simulated image the target, sprinkles, and background objects appear closer to the camera. This effect is likely a result of the imperfect model of the RGB camera sensor in CARLA.



(a) Simulated

(b) Real-world

Fig. 13: Simulated and real-world scenario comparison

Secondly, the real image is significantly impacted by camera overexposure effects. However, the current functionality of CARLA does not allow for the replication of the same effect in simulation.

Lastly, even though the simulated image has been distorted based on the distortion parameters obtained from the calibration of the real camera, the distortion of the real image is noticeably stronger than of the simulated one. This presumably stems from the limited accuracy of the camera calibration results.

These observations illustrate the persisting challenges in achieving perfect simulation of the real-world tests in a virtual environment.

III. Object Detection

A. Research

During the initial phase of the project, our team devoted considerable effort to conducting comprehensive research on various tools available for working with image recognition, which were deemed essential for fulfilling our project objectives. These tools included Python, OpenCV, ROS, DD3D, and various models.

Following the completion of the research and learning phase, task allocation was undertaken based on the team members' prior expertise and knowledge in the subject matter.

1) OpenCV: OpenCV is widely recognized as a popular open-source library renowned for its extensive application in image processing tasks [7] [8] [9]. It serves as a versatile tool capable of processing images and videos to detect and analyze objects, faces, and even human handwriting. With its support for multiple programming languages such as Python, C++, Java, and more, OpenCV has established itself as a flexible solution in the field.

In the context of utilizing OpenCV in Python, several fundamental image processing techniques can be employed, which we have tested ourselves to see if they would serve as a possible solution for some of the tasks in our team. These include:

- Image Reading: Employing the imread() method from the cv2 module enables the reading of an image. For instance, one may execute: img = cv2.imread("pyimg.jpg").
- Image Display: The imshow() method from the cv2 module facilitates the display of an image. For example, one may utilize: cv2.imshow("Image", img).
- Image Resizing: The resize() method from the cv2 module allows for image resizing. To illustrate, one can utilize: resized = cv2.resize(img, (width, height), interpolation = cv2.INTER_AREA) [9].
- Image Rotation: By utilizing the getRotationMatrix2D() and warpAffine() methods from the cv2 module, images can be rotated effectively.
- Image Flipping: To flip an image, the flip() method from the cv2 module can be utilized. For example: flipped = cv2.flip(img, 1) [9].
- Grayscale Conversion: To convert an image to grayscale, the cvtColor() method from the cv2 module is used. For instance: gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).
- Applying Masks to Images: To apply a mask to an image, the bitwise_and() method from the cv2 module is employed.

2) ROS: Considering the project's requirement for ROS utilization and our team's limited prior experience with it, an in-depth exploration of relevant documentation and diverse sources became essential to acquire comprehensive knowledge on the subject.

The initial phase involved installing the Noetic version of ROS1 on our Ubuntu distribution, a vital step to ensure compatibility and functionality. This was one of the first challenges that our team had to face, since one of the members was not able to properly install a virtual machine with Ubuntu, due to issues with the processor of the laptop that was supposed to be used for it. The workaround was to directly work in the University's computer, which also allowed us to effectively collaborate, and also facilitated the coordination with other groups from the project.
Once the environment inside Ubuntu was set up, we adopted a multifaceted learning approach to expedite our understanding. While leveraging YouTube video tutorials, which proved highly beneficial, we concurrently pursued a step-by-step guide provided on the official ROS website to solidify our grasp on the subject matter.
To know how to define our part of the project's architecture, first we had to understand how ROS works, which elements it is composed of, and how these communicate with each other. The following elements were key for us to understand in order to succeed in the implementation of the project.

- ROS command lines: ROS command lines refer to the set of command-line tools provided by the ROS framework for executing various tasks and managing ROS components. These tools enable users to interact with the ROS system, launch nodes, inspect topics, visualize data, record and playback data, and perform other essential operations. Proper utilization of ROS command lines is crucial for efficient ROS workflow management.
- ROS Packages: ROS packages are the fundamental organizational units in ROS. They are self-contained directories that encapsulate all the necessary files, libraries, and dependencies for a specific ROS functionality or application. Packages provide a modular structure, facilitating code reusability and maintainability. They are managed using the Catkin build system and enable easy distribution and sharing of ROS software components.
- ROS Nodes: In ROS, nodes are individual computational units that perform specific tasks or functions within the overall system. Nodes are designed to be lightweight and can be distributed across multiple machines to achieve distributed computing. They communicate with each other by publishing and subscribing to topics, allowing for efficient data exchange and coordination in a ROS network.
- ROS Bags: ROS bags are a file format used for recording and playing back ROS message data. They enable the storage and retrieval of published messages, providing a convenient way to log and analyze data collected during ROS system operation. ROS bags are particularly useful for offline analysis, debugging, and testing of ROS applications.
- ROS Topics: ROS topics facilitate communication between ROS nodes by enabling the exchange of mes-

sages. Topics follow a publish-subscribe messaging paradigm, where nodes can publish messages to a specific topic, and other nodes can subscribe to that topic to receive the messages. This decoupled communication mechanism promotes loose coupling and flexibility in connecting different components of a ROS system.

- Publisher Node: In ROS, a publisher node is responsible for sending messages to a specific topic. It creates and publishes messages on a particular topic, allowing other nodes subscribed to that topic to receive and process the messages. Publisher nodes play a vital role in disseminating information or data throughout a ROS network.
- Subscriber Node: A subscriber node in ROS is responsible for receiving and processing messages from a specific topic. It subscribes to a topic of interest and waits for incoming messages. When a message is published on that topic, the subscriber node receives it and performs the necessary actions based on the received data.
- ROS Launch: ROS launch is a powerful tool used for orchestrating the execution of multiple ROS nodes and configuring the ROS environment. It allows users to specify and launch multiple nodes with their respective parameters and remappings in a single command. ROS launch files aid in simplifying the initialization and configuration of complex ROS systems.

3) Object Detection: Object detection is a common computer-vision task to detect and classify multiple objects from sensor data. Since our project is designed to detect road participants from a single camera (also referred as mono-cam), there are two possible types of object detection: 2D or 3D Multi-Object detection. 2D Object detection locates the object in an image while 3D Object detection locates their position in 3D world coordinates along with rotation. Since camera images are composed of millions of RGB pixels and inherently lack depth information, 2D detection is a much easier task and many efficient algorithms like Yolo, SSD among others have been developed for this purpose. On the other hand, 3D object detection is a relatively complex task using a single image and is less accurate in general 14. For autonomous driving tasks, it is crucial to have exact 3D locations of surrounding objects and hence it was decided to use one of the state-of-the-art 3D object detection algorithms, DD3D.

Fig. 14: 3D (DD3D) vs 2D (YoloV5) object detection

DD3D [10] (Dense depth pre-training for 3D detection) is a fully convolutional single-stage network for monocular 3D object detection and depth prediction. The network takes an RGB image as input and computes convolutional features at different scales and each head produces a classification score, 2D bounding box, 3D Box, dense depth map, and 3D confidence score for each detected object. Noisy detected objects are further filtered out by post-processing the model outputs with the 3D Non-Maximum suppression (NMS) algorithm. This model has been researched and evaluated by Toyota Research Institute-Machine Learning (TRI-ML [11]) and they have open-searched their results as well [12].

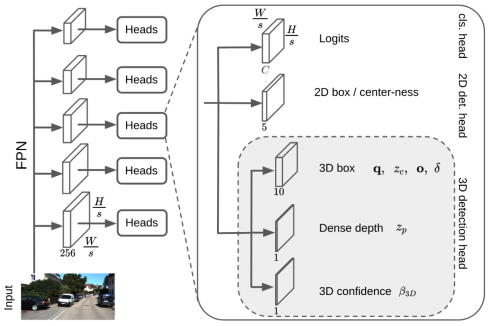


Fig. 15: DD3D neural network structure [10]

Three different variants of backbones are available for DD3D: V2-99, DLA34, and OmniML. They were trained and evaluated on KITTI-3D [13] and NuScenes [14] datasets. The main reason for choosing this model was its reported high accuracy (AP 30.98 for Car, BEV-Easy in KITTI-3D, and AP 41.8 for Car in NuScenes) and close to real-time inference speeds. Due to the inherent lacking of depth information in RGB images, mono-camera object detection network tend to be really big networks with slow inference speed or very fast with bad detection accuracy. DD3D has a good balance between both. Pretrained weights were released by TRI-ML only for V2-99 and DLA34 backbone for KITTI-3D dataset.

The initial days were spent on setting up our development environment for running these models on our local system. The development computer was one of CARISSMA's lab PC with an Intel i7-11700K CPU with 64 GB RAM and Quadro RTX 5000 GPU with 16 GB VRAM. The released code for DD3D had installation instructions only to run in Docker but since it has to run in our local system, the installation steps were extracted from it. DockerFile and executed in our system. The main dependencies were Nvidia drivers for GPU support and Pytorch 1.9.0 deep-learning framework with Python 3.8. After installing the required dependencies, sample scripts from the repository could be successfully run.

B. Implementation

*1) **Improving the model**:* The provided code had scripts to train and evaluate on KITTI and NuScenes datasets but not directly on images and videos. So the next step was to dig deep into the repository and get a basic understanding of its working. This involved reading and debugging through hundreds of lines of TRI-ML's own research code and trying to decouple just the model inference from all the boilerplate code. Furthermore, no straightforward method was provided

to visualize the model detections on the images. Here all the research efforts and our learnings from OpenCV helped to create 3D bounding boxes on image by using projection matrix along with alpha colors to display the orientation and small text on the bottom to indicate the class and confidence score. This effort took 3 weeks of time.

Fig. 16: Real world images with DD3D output boxes visualized

With this custom inference code, we proceeded towards evaluating the performance and accuracy of different variants of DD3D. As mentioned in the original paper, it came to our understanding V2-99 variant is more accurate than DLA34 but significantly slow as well. OmniML variant should be right in-between both these models for both performance and accuracy and was initially chosen to be perfect for our use case. Unfortunately as mentioned before, TRI-ML didn't release any pre-trained weights for it. Config files and training scripts for training the model from scratch were provided and an attempt was made to train our own OmniML model on KITTI dataset. A training run was started on their default OmniML config on a much more powerful CARISSMA Lab PC with Intel i9 and RTX 3090. The entire training lasted 10 full days resulting in an underwhelming accuracy of 18% mAP (Mean Average Precision) in KITTI validation set compared to the 35.5% mAP stated in the repo. On running this model on couple of sample images, it produced lot of incorrect and noisy detections. The training process ran smoothly without any errors and upon further inspection of code for possible mistakes, nothing out of the ordinary could be found. Hence it was decided to not further this approach anymore and stick with the other variants.

Another approach was also explored to improve the accuracy of the models: Training on NuScenes dataset. NuScenes is a much more diverse, rich and higher-resolution dataset compared to KITTI, and according to the DD3D paper, should provide a meaningful improvement in accuracy. Since trained models on NuScenes dataset was not made available for public, we decided to train it from scratch by ourselves. NuScenes dataset required around 1TB of free space (compared to 180 GB of KITTI). So, downloading and storing the dataset was a huge challenge initially. Then minor errors popped in code and significant amount of time went into debugging it.

Then the training run was initialized and according to the time counter during training, 66 days was required to fully complete the training in a single RTX 3090 system. TRI-ML had originally trained the model much faster as they used Multi-GPU clusters but for single-GPU setup like ours, it was too long and was infeasible. Nevertheless, we started with training and allowed to proceed for 14 days. During training, the checkpoint model with all the weights were stored every 2000 training steps and at the end of 14 days, we had already 10 checkpoints. All the checkpoints were evaluated but the improvement in accuracy was too little to justify any further training. Hence, the training was stopped and we had no other option but to use the KITTI pre-trained variants of V2-99 or DLA34.

Since the sole task was to just detect objects from single images, all the unwanted evaluation and other boilerplate code were removed from the forward() statement of the model to improve performance. Also, it was noticed that none of the DD3D variants had support of FP16 (Floating point 16 Bit) inference, even though they were stated to be used in the training. According to my initial test, only the default FP32 (Floating point 32 Bit) was being used. FP16 has the benefit of being twice as small as FP32, resulting in larger models able to fit in small GPU memory. Furthermore, many GPUs, especially Nvidia GPUs like ours, have special tensor cores which are purpose-made for executing FP16 matrices very efficiently. This leads to significant performance improvement as shown in the benchmarking table below:

Image resolution	**V2-99**		
	FP32 (ms)	**FP16 (ms)**	**Speedup (%)**
1936, 1464	-	-	-
1452, 1098	-	156	-
968, 732	165.7	108.45	52.8
640, 484	92.04	63.83	44.2

TABLE I: Benchmark results (Time required per image) with V2-99 backbone

Image resolution	**DLA34**		
	FP32 (ms)	**FP16 (ms)**	**Speedup (%)**
1936, 1464	166.34	113.2	46.9
1452, 1098	106.43	64.71	64.5
968, 732	60.94	44.5	36.9
640, 484	42	44.68	-6

TABLE II: Benchmark results (Time required per image) with DLA34 backbone

As observed in I, for image resolution 1936x1464, V2-99 cannot even run as it is too big to fit in 16 GB GPU VRAM. Same applies to V2-99 (FP32) at 1452x1098 resolution, but since weights occupy signficantly less memory at FP16, the model is small enough to fit in GPU VRAM and was able to run.

For DLA34 backbone results in II, it is clear that the model is small enough to fit even in the maximum resolution and

shows decent speedup across resolutions when converted to FP16. At 640x484, FP16 is slower than FP32 because the time taken to convert weights into FP16 is far greater than total runtime at FP16. This conversion negates the performance gain and this particular case causes a slowdown in comparison to FP32.

In theory, there should be a drop in accuracy when FP32 model is converted to FP16 due to the loss of precision of the bits, but usage of mixed-precision casting in latest deep-learning frameworks like PyTorch results in very negligible loss of accuracy. This was further backed by our testing of FP32 and FP16 on same images, where no degradation of performance was subjectively noticed.

*2) **ROS Nodes**:* The inference code developed and optimized, as mentioned in the previous section, has to be wrapped ín a ROS1 node called *mono3d_det_node.py* in a separate package called *monocam_3D_object_detection*. This node should subscribe to image and camera_info topics, and publish the result as MarkerArray for visualization in RVIZ. Camera_info topic is of message type *sensor_msgs/CameraInfo* and contain crucial information about image geometry and calibration matrices. Model requires the RGB image and the corresponding intrinsic calibration matrix to correctly project the boxes in world coordinates. Each detected object in model output has associated integer classification value [**??**] and 8 corners of bounding box in 3D world coordinates (Important: Not image coordinates) (Matrix of shape: (8, 3), 3 stands for x, y, z coordinates)

Since real world CARISSMA .rosbags only have sensor data (for cameras and lidars) and no tf and cam_info topic, a separate node *CARISSMA_tf_caminfo_broadcaster.py* was created to generate these topics from positions of the sensors and intrinsic calibration of camera samples was done in a similar process as mentioned here II-E. This node subscribed to topics from real sensors and published them once again along with time synchronized /tf and /cam_info topics

Highly parameterized .roslaunch files were generated to make it easy to change important variables and to prevent hard-coding them in the code itself. The sensor data is primarily provided in form of recorded .rosbags and the name of sensor data topics are different in Carla and real-world CARISSMA test track samples. To prevent changing the name of topic everytime, separate .roslaunch files have been generated for Carla and CARISSMA separately. These .roslaunch files also load play the required .rosbags automatically and open the right configuration for rviz visualization as well

Sensor data	ROS topic for Carla
Camera	/carla/ego_vehicle/rgb_front/image
Camera_info	/carla/ego_vehicle/rgb_front/camera_info
Ouster LiDAR	/carla/ego_vehicle/lidar_ouster_os1

TABLE III: ROS topic names for Carla .rosbags

*3) **Publish RVIZ Info**:* These outputs were then projected to image coordinates for the local visualization done before, but this is not necessary for visualization in RVIZ. Given the

Sensor data	ROS Topic for CARISSMA
Camera	/adapted_arena_camera/compressed
Camera_info	/camera_info
Ouster LiDAR	/ouster/points_adapted

TABLE IV: ROS topic names for CARISSMA .rosbags

location/orientation of camera and camera matrices (intrinsic, extrinsic and distortion), , which are specifically present in */tf* and */camera_info* topics, RVIZ will automatically project it in the right position. As mentioned before, outputs to RVIZ should be in RVIZ MarkerArray format and it offers variety of shapes for visualization like Line, Cube, Sphere, Arrows among others. Since we are dealing with boxes around detected objects, RVIZ MarkerArray of CUBE was chosen as the best one. It required center position, orientation and scale (i.e size) of the CUBE along all three axes, and this was calculated to using simple geometry from the 8 corners of the box.

Position and scale calculation was pretty straightforward but the rotation calculation along each axis (yaw, pitch and roll) was tricky. This was further complicated by the fact that Carla and RVIZ used a different coordinate system, with Camera additionally being rotated horizontally in Carla. Finding the right axes to rotate was pretty much evaluated by trial-and-error basis, and an plausible solution was found after weeks of continous testing. Furthermore camera was supposed to have additional pitch towards the ground (5 degrees) but this was missing from the data we evaluated on. Once this was introduced, the visualized RVIZ cubes were not aligning perfectly, indicating a lot more time-consuming testing was required to make it work.

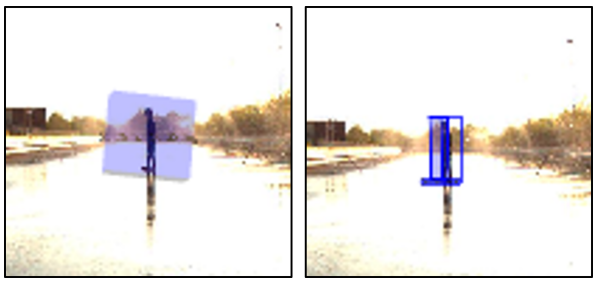
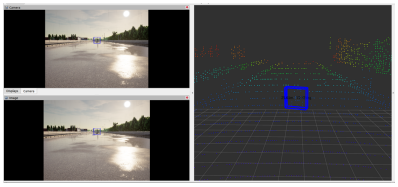


Fig. 17: Wrong rotation with rviz CUBE (left) compared to accurate rviz LINE_LIST (right)

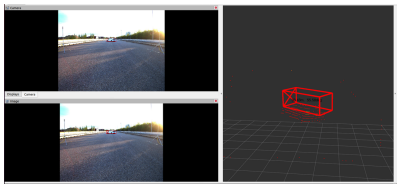
Instead of this, a more straight-forward approach was then adopted: Drawing just edges between corners of the box. This is exactly how the local visualization using OpenCV is done and tends to be precise as well. For this, RVIZ LINE_LIST Marker type was used and to-be connected corners are iterated over in right order to form an edge 17. To indicate the heading direction, the front face of the bounding box was lightly shaded (using alpha) in local visualization. This was unfortunately not possible with RVIZ, as it natively has no support for visualizing planar surfaces. Instead the heading direction is visualized with a simple **X** in the front face. An additional text marker to indicate the distance of

object along with it's orientation is also added to make it easier to collect results for evaluation. Colors of each class is distinct: Car, Cyclist and Pedestrian

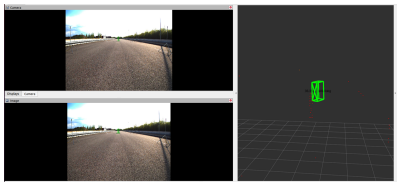
C. Results



(a) Carla: Rain, Bike at 28m

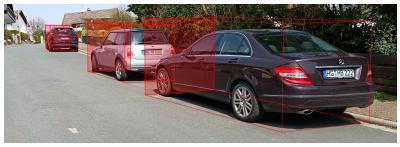


(b) Carissma: Clear weather, Car at 28m



(c) Carissma: Clear weather, Pedestrian at 28m

Fig. 18: RVIZ visualization using DLA-34 backbone at 1600p: RVIZ projected camera image (Top-left), local OpenCV visualization (Bottom-right), and LiDAR point cloud (Right)



(a) Picture from smartphone

(b) Picture from normal Carla map

Fig. 19: Local visualization with OpenCV using V2-99 backbone at 1200p

1) How does the model work?:

Results DLA-34					
Road User	Weather Conditions	Real Values		Model Values	
Pedestrian	Clear Weather	28m	90°		
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	25.29m	10.32°
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
Bike	Clear Weather	28m	90°	31.98m	112.94°
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	20.93m	126.26°
		56m	90°	38.55m	19.56°
		84m	90°	-	-
		112m	90°	-	-
Car	Clear Weather	28m	45°	31.43m	66.3°
		56m	45°	59.36m	56.31°
		84m	45°	80.0m	44.88°
		112m	45°	-	-
	Rain	28m	45°	30.22m	76.99°
		56m	45°	43.67m	45.9°
		84m	45°	77.26m	33.1°
		112m	45°	-	-

TABLE V: Results DLA-34

Results V2-99					
Road User	Weather Conditions	Real Values		Model Values	
Pedestrian	Clear Weather	28m	90°		
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	-	-
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
Bike	Clear Weather	28m	90°	33.02m	63.27°
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	-	-
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
Car	Clear Weather	28m	45°	30.44m	55.31°
		56m	45°	56.93m	56.89°
		84m	45°	77.42m	37.23°
		112m	45°	-	-
	Rain	28m	45°	29.25m	65.79°
		56m	45°	54.96m	51.92°
		84m	45°	73.03m	6.09°
		112m	45°	-	-

TABLE VI: Results V99

How does the model work for the virtual environment data from CARLA:

Results DLA-34 - CARLA					
Road User	Weather Conditions	Real Values		Model Values	
Pedestrian	Clear Weather	28m	90°	28.67	110.8
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	25.88m	162.55°
		56m	90°	-	-
		84m	90°	-	-
		112m	90°	-	-
Bike	Clear Weather	28m	90°	31.98m	112.94°
		56m	90°	54.45	36.46
		84m	90°	-	-
		112m	90°	-	-
	Rain	28m	90°	26.01m	75.04°
		56m	90°	39.84	80.87°
		84m	90°	-	-
		112m	90°	-	-
Car	Clear Weather	28 m	45°	29.43m	69.73°
		56m	45°	54.65m	57.73°
		84m	45°	m	°
		112m	45°	-	-
	Rain	28m	45°	27.97m	62.88°
		56m	45°	54.63m	63.35°
		84m	45°	68.53m	64.39°
		112m	45°	80m	69.14°

TABLE VII: Results DLA-34 with Carla Bags

2) Results conclusion: After running both models for each of the 24 scenarios that we have, both for Carla and Real-World data, we came to the conclusion that the best performance and better accuracy for our usecase was achieved with the DLA-34 backbone. V2-99 was relatively slower and more stabile in it's detections (less flickering of boxes) but had overall less accuracy.

- For the case of Pedestrian with rainy weather with Carissma .rosbags, DLA-34 model is recognizing the pedestrian as a bicycle (most of the times).

With the ROS bags that were created by the Team 2 in CARLA the following issues were encounter, when running the DLA-34 model:

- For the scenario bike, with clear weather at a distance of 56m, the model is detecting the dummy but only for short moments. In our opinion, even though it is recognizing the correct object at an approximated distance, we would not say that this can be classified as a good recognition.
- For the scenario Pedestrian, with rain at a distance of 28m, the DLA-34 model detected both a pedestrian and a bicyclist at the same time, for some periods of times. Rest of the time, the correct classification was done.

3) Possible improvements:

- 8-bit quantization for faster inference: Significant performance gains have been shown by moving from FP32 to FP16 I. Proceeding in the same order to even lower quantization levels like INT8 or FP8, should result in even better performance.
- Converting to Nvidia TensorRT: As of right now, the model is executed in PyTorch. PyTorch is pretty fast and efficient but converting the model to Nvidia's own TensorRT framework will allow the model to make use of Nvidia specific optimizations leading to significant speedups. TensorRT also has support for above mentioned lower quantization levels
- Multi-GPU training: The model couldn't be trained on NuScenes dataset due to extremely long training time. With multi-gpu setup as mentioned in the original DD3D paper, the training speed could be significantly improved.
- Synthetic Dataset: As seen in the results above, the model performs better in Carissma test case than Carla. This could be explained by the fact that the model was exclusively trained on real-life images from KITTI dataset and had no simulation images. Many frameworks online have the functionality to generate synthetic data from CARLA and this could be smartly mixed with KITTI dataset to obtain even higher accuracy.
- Sensor Fusion: From the sensor setup point-of-view, both CARLA and CARISSMA .rosbag have LiDAR data. This LiDAR data could be leveraged to improve object detection.

CONCLUSION

In conclusion, this project focused on creating a virtual environment that serves as a comprehensive tool for scenario creation supported by an image recognition model. The integration of the Carla simulator and ROS bridge allowed for the generation of realistic virtual environments, however, with some limitations discussed in the previous chapters. At the same time, the utilization of DD3D as an image recognition framework is provided with a detailed analysis of underlying models. Through the comparison of data obtained from virtual and real-world scenarios, we gained valuable insights into the potential of virtual environments as a substitute for real-world testing. Furthermore, the calibration of sensors and the acquisition of real-world data played a crucial role in ensuring the accuracy and reliability of our evaluations. By leveraging the continuous advancements in the tools and methods employed throughout the project, there is a significant potential for enhancing performance, and, hence, expanding the operational design domain (ODD).

REFERENCES

[1] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[3] Ouster, Inc. (2021) OS1. Mid-Range High-Resolution Imaging Lidar. [Online]. Available: https://data.ouster.io/downloads/datasheets/datasheet-revd-v2p1-os1.pdf

[4] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[5] CARLA. (2023) Python API - CARLA Simulator. [Online]. Available: https://github.com/carla-simulator/carla/tree/master/PythonAPI

[6] Epic Games, Inc. (2023) Physics Asset Editor — Unreal Engine Documentation. [Online]. Available: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Physics/PhysicsAssetEditor/

[7] geeksforgeeks. (2023) OpenCV Python Tutorial. [Online]. Available: https://www.geeksforgeeks.org/opencv-python-tutorial/

[8] G. Singh. (2023) Image Processing using OpenCV in Python. [Online]. Available: https://python.plainenglish.io/image-processing-using-opencv-in-python-857c8cb21767

[9] A. Jayaswal. (2023) GETTING STARTED WITH PYTHON IMAGE PROCESSING USING OPENCV. [Online]. Available: https://www.youngwonks.com/blog/Getting-started-with-Python-Image-Processing-using-OpenCV

[10] D. Park, R. Ambrus, V. Guizilini, J. Li, and A. Gaidon, "Is pseudo-lidar needed for monocular 3d object detection?" in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[11] "Toyota Research Institute - Machine Learning — github.com," https://github.com/TRI-ML, [Accessed 22-Jun-2023].

[12] "GitHub - TRI-ML/dd3d: Official PyTorch implementation of DD3D: Is Pseudo-Lidar needed for Monocular 3D Object detection? (ICCV 2021), Dennis Park*, Rares Ambrus*, Vitor Guizilini, Jie Li, and Adrien Gaidon. — github.com," https://github.com/TRI-ML/dd3d, 2021, [Accessed 22-Jun-2023].

[13] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[14] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *CVPR*, 2020.