INSTITUTE FOR MEASUREMENT SYSTEMS
AND SENSOR TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Electrical Engineering and Information Technology

Enhancing LiDAR-based 3D Object Detection through Simulation

Aydin Uzun

Matriculation number: 03734883

INSTITUTE FOR MEASUREMENT SYSTEMS AND SENSOR TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Electrical Engineering and Information Technology

Enhancing LiDAR-based 3D Object Detection through Simulation

Author:	Aydin Uzun
Contributor:	M.Eng. Arsalan Haider
Assessor:	Prof. Dr.-Ing. habil. Dr. h.c. Alexander W. Koch
External supervisor:	Dr. Petr Fomin
External supervisor:	Günther Hasna
Company:	Ansys Germany GmbH
Submission Date:	20.06.2023

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Munich, 20.06.2023 Aydin Uzun

Acknowledgments

First and foremost, I want to give a heartfelt thank you to M.Eng. Arsalan Haider. His consistent help and expert advice throughout this research project have been invaluable.

My sincere appreciation also goes to Dr. Petr Fomin and Günther Hasna from Ansys Germany GmbH. They kindly agreed to act as my external supervisors. Their deep knowledge in this area was extremely beneficial and their helpful advice significantly improved my work. Their guidance provided me with practical skills and experiences that expanded my academic learning.

I am also grateful for the opportunity to work with the team at Ansys Germany GmbH. I would like to thank all team members who contributed to my work in any way.

I cannot express enough thanks to my family for their ongoing love and support. To my parents, who have always stood by me, I cannot thank you enough. Your belief in me has always inspired me and pushed me to achieve more.

I would like to give special thanks to my sister, whose ongoing faith in me and continuous support have been a pillar of strength. The person who has given me unending happiness and motivation in this venture is my little niece, Ipek. Her innocent charm and contagious joy have been the silver lining in the most challenging of days, illuminating the path with hope and determination. Her laughter was the melody that soothed stress and her unassuming wisdom was a reminder of the purity and curiosity that is the foundation of all knowledge.

In conclusion, the completion of this thesis marks not only an academic achievement, but also a personal journey of growth and development, for which I am grateful to everyone mentioned above.

List of Symbols

Section 2.2

Δt time difference

c speed of light in a vacuum

Section 2.2.3

ϕ azimuth angle measured in the XY-plane from the positive x-axis

θ elevation angle in the YZ-plane measured from the positive z-axis

r distance of the point from origin

Section 2.5.1.1

C number of output features of encoder network

D dimensionality of augmented LiDAR point

H height of the pseudo-image

l a point in a point cloud

N maximum number of points per pillar

P maximum number of non-empty pillars in a point cloud

W width of the pseudo-image

x, y, z coordinates of l in Cartesian coordinate system

x_c, y_c, z_c distance to the arithmetic mean of all points in the pillar

x_p, y_p offset from the pillar's x and y center

Section 2.5.1.2

Block(S, L, F) a block operates at stride S, has L 3x3 2D convolution layers with F output channels

Up(S_{in}, S_{out}, F) an upsampling block, transitions from stride S_{in} to S_{out} via a transposed 2D convolution producing F output features

Section 3.1.1.4

a accuracy of the distance measurement

c speed of light in a vacuum

f sampling frequency of the ADC

Section 3.3.3

$ds.x$,$ds.y$,$ds.z$ relative distances from the Object Sensor

$r_{zyx}.x$, $r_{zyx}.y$, $r_{zyx}.z$ rotational orientation (Euler angles in the ZYX convention) of the traffic objects, as interpreted from the Object Sensor

Section 3.4.1

θ yaw angle of the 3D bounding box

w, l, h width, length and height of the 3D bounding box

x,y,z center point coordinates of the 3D bounding box

Section 3.4.2.2

imagesize dimensions of the camera-generated image in pixels. It is a matrix of size 1×2.

p_{2D} a 3D point in the 2D image plane of the camera in homogeneous coordinates

$P_{\text{rect}}^{(2)}$ projection matrix post-rectification for camera 2 (the left color camera). This matrix transforms 3D points in the rectified camera coordinate system to 2D points in the image plane. It is a matrix of size 3×4.

p_{cam} a 3D point in the camera coordinate system in homogeneous coordinates

p_{lidar} a 3D point in the LiDAR coordinate system in homogeneous coordinates

p_{rect} a 3D point in the rectified camera coordinate system in homogeneous coordinates

$R_{\text{rect}}^{(0)}$ rectifying rotation matrix. This matrix aligns the 3D points in the camera coordinate system with the axes of the rectified coordinate system. It is a matrix of size 3×3.

$T_{\text{velo}}^{\text{cam}}$ transformation matrix, which converts 3D points from the LiDAR coordinate system to the camera coordinate system. It is a matrix of size 3×4.

x, y coordinates on the image plane

Section 3.4.3.6

x_{min},y_{min},z_{min},x_{max},y_{max},z_{max} x,y,z range for point cloud filtration

Section 4.2

C number of output features of encoder network

D dimensionality of augmented LiDAR point

H height of the pseudo-image

N maximum number of points per pillar

P maximum number of non-empty pillars in a point cloud

W width of the pseudo-image

x_{min},y_{min},z_{min},x_{max},y_{max},z_{max} x,y,z range for point cloud filtration

Block(S, L, F) a block operates at stride S, has L 3x3 2D convolution layers with F output channels

Up(S_{in}, S_{out}, F) an upsampling block, transitions from stride S_{in} to S_{out} via a transposed 2D convolution producing F output features

Section 4.3

AP_{2D} Average Precision for two-dimensional object detection

AP_{3D} Average Precision for three-dimensional object detection

AP_{BEV} Average Precision for bird's-eye view object detection

$IoU_{2d}(d_{2d}, g_{2d})$ Intersection over Union in two-dimensional representation

$IoU_{3d}(d_{3d}, g_{3d})$ Intersection over Union in three-dimensional representation

$IoU_{bev}(d_{bev}, g_{bev})$ Intersection over Union in bird's-eye view representation

δ_i 1 if detection i has been assigned to a ground truth bounding box, 0 otherwise

Δ_θ^i angular discrepancy between the ground truth and predicted orientations for the i^{th} detection

τ detection threshold for Intersection over Union

AOS average orientation similarity

AP Average Precision

$conf(d)$ confidence of a detection

d detection

$D(r)$ all object detections at recall rate r

d_{2d} detection in two-dimensional representation

d_{3d} detection in three-dimensional representation

d_{bev} detection in bird's-eye view representation

g ground truth box

g_{2d} ground truth box in two-dimensional representation

g_{3d} ground truth box in three-dimensional representation

g_{bev} ground truth box in bird's-eye view representation

$P_{\text{interpolate}}(r)$ interpolated precision at recall level r

$Precision(t)$ precision for a confidence threshold

$Recall(t)$ recall for a confidence threshold

$s(r)$ orientation similarity

$TP(d, g)$ true positive, a detection d is classified as a true positive if its IoU with a ground truth box g surpasses a certain threshold τ.

Glossary

Ansys AVxcelerate CarMaker Co-Simulation This is a collaborative simulation process that involves Ansys AVxcelerate Sensors Simulator and CarMaker software, aimed at providing a comprehensive testing environment for autonomous vehicle systems. AVX simulates sensor outputs, while CarMaker emulates vehicle dynamics and traffic scenarios. In this co-simulation, CarMaker provides high-fidelity models of the vehicle, the road, and the surrounding traffic, while AVxcelerate, with its physics-based sensor models, simulates the sensor data for the virtual vehicle..

Ansys AVxcelerate Sensors Simulator Ansys AVxcelerate Sensors Simulator is a high-fidelity simulation tool designed for autonomous vehicle (AV) applications. The software allows for accurate modeling of diverse environmental conditions and scenarios that AVs could encounter. It is capable of simulating various types of sensors used in AVs such as LiDAR, RADAR, and cameras. .

fine-tuning Fine-tuning in the context of machine learning is a process where a pre-trained model, typically on a large-scale data set, is further adjusted or 'tuned' on a specific data set. This method capitalizes on the generic features learned from the initial broad training phase and adapts them to the specific data set, often improving model performance with less data or computational resources than training from scratch..

KITTI data set The KITTI data set is a comprehensive dataset from the Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. It was

developed to aid the research in the area of autonomous driving and contains a vast range of sensor data collected from real-world driving scenarios..

KITTI Evaluation Metrics These are specific standards used for measuring the performance of computer vision models in the KITTI data set. The metrics for 3D object detection predominantly involve the Average Precision (AP) calculated at various Intersection over Union (IoU) thresholds. The AP is the area under the precision-recall curve, summarizing a model's accuracy across different confidence thresholds..

LiDAR-based 3D Object Detection LiDAR-based 3D Object Detection refers to the process of identifying and locating objects in a 3D space using data generated by Light Detection and Ranging (LiDAR) sensors. LiDAR sensors create high-resolution 3D maps of the environment by emitting pulses of light and measuring the time taken for the light to bounce back after hitting an object. This process generates a point cloud, a large set of data points that represent the 3D coordinates of individual points on the detected objects. Algorithms and machine learning techniques are then applied to this point cloud data to identify and classify the objects, and determine their exact locations and dimensions in the 3D space..

PointPillars PointPillars is a type of neural network architecture designed for object detection in 3D point clouds. PointPillars employs a novel 'pillar' feature encoding scheme. This encodes the 3D point cloud data into a 2D pseudo-image, enabling the use of efficient convolutional neural networks. This approach allows for fast processing speeds and high detection accuracy..

Acronyms

2D two-dimensional.

3D three-dimensional.

ADAS Advanced Driver Assistance Systems.

ADC Analogue to Digital Converter.

AOS Average Orientation Similarity.

AP Average Precision.

AP_{2D} Average Precision for 2D object detection.

AP_{3D} Average Precision for 3D object detection.

AP_{bev} Average Precision for BEV object detection.

AV autonomous vehicle.

AVX Ansys AVxcelerate Sensors Simulator.

BatchNorm Batch Normalization.

BEV bird's-eye view.

BRDF Bidirectional Reflectance Distribution Function.

CAD Computer Aided Design.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

EntityID Entity Identification Number.

FV Front View.

GCC GNU Compiler Collection.

GPS Global Positioning System.

GPU Graphics Processing Unit.

GTA-V Grand Theft Auto V.

IMU Inertial Measurement Unit.

IoU Intersection over Union.

JSON JavaScript Object Notation.

KITTI Karlsruhe Institute of Technology and Toyota Technological Institute.

LiDAR Light Detection and Ranging.

MEMS Micro Electronic Mechanical Systems.

NMS Non-maximum Suppression.

OPA Optical Phased Array.

PR Precision-Recall.

RADAR Radio Detection and Ranging.

ReLU Rectified Linear Unit.

SSD Single Shot Detector.

ToF Time of Flight.

Abstract

This Master's thesis investigates the enhancement of LiDAR-based 3D Object Detection algorithms for autonomous vehicles, using synthetic point cloud data generated from the Ansys AVxcelerate CarMaker Co-Simulation process. The study focuses on integrating and aligning synthetic and real-world data, and applying fine-tuning techniques within the PointPillars network to optimize the model. The research reveals challenges in ensuring model generalization across different data types, especially when identifying complex entities like pedestrians. The study indicates that a balanced combination of synthetic and real-world data yields promising results. Additionally, a hybrid training approach, consisting of initial pre-training with synthetic data followed by fine-tuning with real-world data, exhibits potential, particularly under conditions of real-world data scarcity. This study thus provides valuable insights to guide future improvements in the training and testing methodologies for autonomous driving systems.

Contents

1. Introduction

1.1. Problem Statement

Autonomous driving technology, which has the potential to transform transportation systems, relies on advanced perception systems for safe navigation [1]. One key element of these systems is three-dimensional (3D) object detection [2], which significantly benefits from Light Detection and Ranging (LiDAR) sensors. These sensors provide detailed 3D models, a wide detection range, and high accuracy, creating point cloud representations of the environment that are invaluable for depth perception [3, 4].

However, using LiDAR technology comes with several challenges. Training deep learning algorithms for LiDAR-based object detection requires a large amount of labeled data [5, 6]. Given the limited availability of suitable training samples for each object, classification becomes difficult [7]. Unlike image or text-based applications of deep learning that have abundant training data, point clouds present a unique challenge due to their discrete nature, variations in point density, and incomplete point segments [8, 5, 9, 10]. This makes manual annotation complex, leading to a scarcity of 3D training data.

Given the lack of large-scale annotated data, simulation software for synthetic data generation is a potential solution [8, 6, 15, 16, 5, 4, 17, 18, 19, 11, 12, 13, 14]. Simulations allow for testing in closed-loop environments and generating data for rare events. However, these often fail to accurately replicate real-world sensory data due to their reliance on handcrafted 3D assets and simplified physics [17, 19, 14]. This leads to a 'synthetic-to-real gap' [20], a significant difference between simulated and actual

environments.

Additionally, models trained exclusively on synthetic data may not perform well in real-world scenarios due to differences in data distributions [14]. This calls for improved methods to bridge the gap between simulations and real-world scenarios. Potential solutions could involve combining real and synthetic data [6], using fine-tuning techniques [14, 19], or improving the quality of simulated data [16].

Ansys AVxcelerate Sensors Simulator (AVX) [21], is a tool designed to address these challenges. It provides a virtual environment for testing and analyzing the performance of sensors used in self-driving cars and Advanced Driver Assistance Systems (ADAS), thus reducing the need for costly physical prototypes.

In the context of this study, AVX could potentially help bridge the synthetic-to-real gap by providing a realistic testing environment for autonomous vehicles' sensors. However, it's crucial to investigate how accurately the simulator can replicate real-world data quality and how this impacts the performance of the resulting algorithms.

1.2. Objectives and Research Questions

Given the significant findings of this study, the main objectives and research questions that guided this Master's thesis were:

1.2.1. Objectives

- Generating a replica of the renowned Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) data set [22] using synthetic data crafted with the Velodyne HDL-64E [23] LiDAR instrument model in the AVX, in co-simulation with the CarMaker software [24].
- Applying bounding box extraction algorithms to synthetic point clouds and creating KITTI-compatible labels for extensive evaluation.
- Investigating the potential of synthetic data in enhancing the performance of

object detection algorithms.

- Comparing the performance metrics of models trained on diverse data types (synthetic versus real-world).
- Evaluating the influence of modifying the ratio of synthetic to real-world data on the performance.
- Assessing the viability and efficacy of a hybrid training strategy involving pre-training on synthetic data with subsequent fine-tuning on real-world data.
- Analyzing the impact of pre-training duration on the optimization of model parameters.
- Conducting a detailed qualitative analysis of the trained networks.

1.2.2. Research Questions

- How does the use of synthetic data affect the performance of object detection algorithms in autonomous vehicles?
- What are the challenges in applying models trained on a certain data type (synthetic or real-world) to another type and what are potential strategies to address these challenges?
- What is the effect of changing the ratio of synthetic to real-world data in a training set on the performance of the resultant model?
- Is a hybrid training approach, combining pre-training on synthetic data with fine-tuning on real-world data, effective in enhancing model performance, particularly when real-world data is limited?
- How does the duration of pre-training impact the optimization of model parameters and the overall performance of the model?

- How does the balance of data types within the training set influence the object detection capabilities of the trained networks?
- How can synthetic data, produced by the AVX, be used to provide robust performance metrics for validating new solutions in the automotive industry?

By answering these research questions, this study offers valuable insights into the benefits and challenges of employing synthetic data in training object detection algorithms for autonomous vehicles.

1.3. Thesis Outline

The structure of this thesis is as follows:

- **Chapter 2: Literature Review**
  This chapter delves into the current body of knowledge on 3D object detection techniques and LiDAR technology. It reviews sensor modalities in autonomous vehicles, explores different aspects of LiDAR including its classifications, applications, point cloud data characteristics, and challenges. This chapter also investigates autonomous driving data sets, 3D object detection methods using LiDAR, and the PointPillars Network. It concludes with an overview of related research.
- **Chapter 3: Methodology**
  In this chapter, the research methodology is outlined. Emphasis is placed on the modeling of the LiDAR sensor. It offers a detailed account of the Ansys AVxcelerate CarMaker Co-Simulation process, elaborates on the processing of simulation outputs, and outlines how simulated scenarios are scaled.
- **Chapter 4: Experimental Design**
  The fourth chapter describes the experimental design. It specifies the data sets used, explains the settings of the network, the evaluation metrics, and how KITTI

difficulty levels are adapted for synthetic data set evaluation. It also presents the different experiments carried out.

- **Chapter 5: Results and Discussion**

  This chapter delves into the results from the experiments. A quantitative analysis of the results from each experiment is provided, along with an impact assessment of pre-training and training duration on the Average Precision for 3D object detection (AP_{3D}) scores. A qualitative analysis on the AVX test set and KITTI test set is also included.

- **Chapter 6: Conclusion**

  The final chapter summarizes the findings of the research, offers recommendations for future work, and provides a concluding reflection on the entire research.

2. Literature Review

2.1. Sensor Modalities in Autonomous Vehicles

Advancements in autonomous vehicle technology are deeply intertwined with the development of robust perception systems, effectively functioning as the 'eyes and ears' [25] of self-driving vehicles [1]. A cornerstone of this evolution is the application of machine learning models, particularly those designed for object detection [2, 26, 27, 28]. The efficacy of these models predominantly hinges on the quality and volume of the training data, underscoring the significance of the process of high-quality data set acquisition, which often proves to be labor-intensive and resource-demanding [29, 15, 10]. Various sensors equipped in an autonomous vehicle contribute to the generation of this training data, each capturing distinct yet complementary information about the vehicle's surrounding environment [1].

- **Light Detection and Ranging (LiDAR):** LiDAR sensors, paramount to autonomous vehicle perception systems, generate three-dimensional point cloud data, offering intricate details about the vehicle's surroundings [4, 30, 1, 10]. These sensors are proficient in calculating the distance to an object and identifying its shape, proving invaluable in detecting vehicles, pedestrians, and potential road obstacles, notably in cluttered urban settings [1]. However, LiDAR sensors confront challenges, including their substantial cost and potential limitations in detecting objects at extreme distances [4, 10]. Moreover, the data produced by LiDAR sensors is inherently sparse and unstructured, thereby necessitating complex processing steps [10].

- **Radio Detection and Ranging (RADAR):** RADAR sensors [31, 32], which emit radio waves and measure their reflection off objects, demonstrate resilience under diverse environmental conditions and are effective at long distances, making them essential for high-speed scenarios like highway driving. Additionally, they can measure the speed of other vehicles, contributing to functionalities such as adaptive cruise control and other safety systems. However, a significant drawback lies in their relatively lower resolution compared to other sensor types [10].

- **Cameras:** Cameras [31, 32, 4] capture high-resolution visual data, essential for tasks like traffic sign recognition, lane detection, and pedestrian detection. They can provide rich visual details but may suffer performance degradation under adverse lighting conditions or extreme weather.

- **Ultrasonic Sensors:** Ultrasonic sensors [31] are primarily employed for low-speed, close-proximity tasks like parking or maneuvering in tight spaces. They use sound waves to detect the range and direction of nearby objects.

- **Inertial Measurement Unit (IMU) and Global Positioning System (GPS):** IMUs and GPS systems [22, 31] are vital for tracking the vehicle's position and orientation. IMUs provide data about the vehicle's velocity and angular rate, while GPS offers absolute positioning data.

These sensor types bring unique strengths and constraints to the table. Sensor fusion, a technique that consolidates data from multiple sensors, is utilized to enhance the collective capabilities of these sensors while counteracting their individual limitations [9, 31, 32, 4]. This approach ensures a comprehensive and reliable understanding of the environment.

For instance, consider the fusion of data from cameras and LiDAR sensors. While cameras offer rich visual details, they may falter under poor lighting conditions. Conversely, LiDAR sensors provide detailed 3D point clouds representing the vehicle's surroundings irrespective of lighting conditions but lack the capacity to gather color

information. The combination of data from these two sensors can yield a robust and accurate understanding of the environment.

This thesis will predominantly concentrate on LiDAR sensors and their critical contribution to 3D object detection in autonomous vehicle systems.

2.2. LiDAR

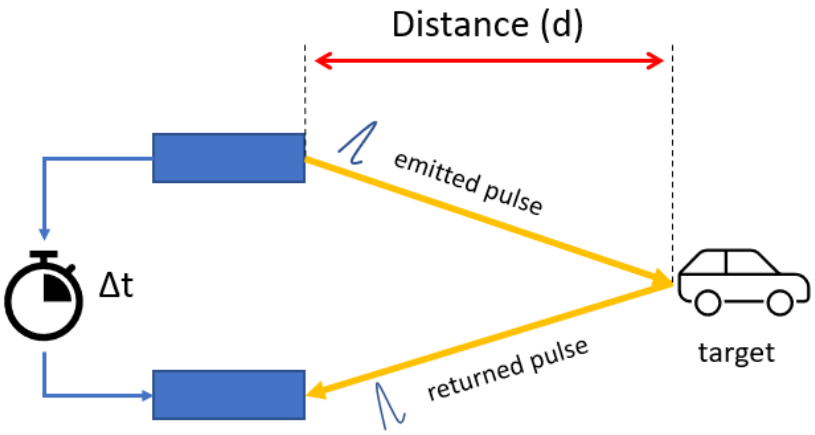


Figure 2.1.: Diagrammatic representation of the Time of Flight (ToF) principle incorporated by LiDAR technology.

$$\text{Distance} = \frac{c \cdot \Delta t}{2} \tag{2.1}$$

LiDAR operates under the principle of Time of Flight (ToF), an approach that involves the propagation of laser pulses and the subsequent monitoring of the time taken for these pulses to be reflected back to the sensor after interacting with various objects [4]. This process is graphically represented in Figure 2.1. The speed of light plays a pivotal role in this process, enabling the calculation of the distance between the sensor and the reflected object as indicated in Equation 2.1. Each return pulse generates a distinct data

point in three-dimensional space, cumulatively creating a 3D point cloud [1], which provides detailed spatial and depth information about the scanned environment.

2.2.1. Classification of LiDAR Sensors

LiDAR sensors can be primarily classified into three categories based on their scanning methodologies [4]:

- **Mechanical LiDAR:** Mechanical LiDAR sensors, typified by the Velodyne HDL-64E [23], operate by physically rotating an array of lasers to scan the environment, achieving a complete 360-degree field of view. Despite their high measurement accuracy and extensive application across autonomous driving and 3D mapping fields, these LiDAR sensors tend to be bulky and expensive [4], owing to the complexity of their mechanical components and assembly process.

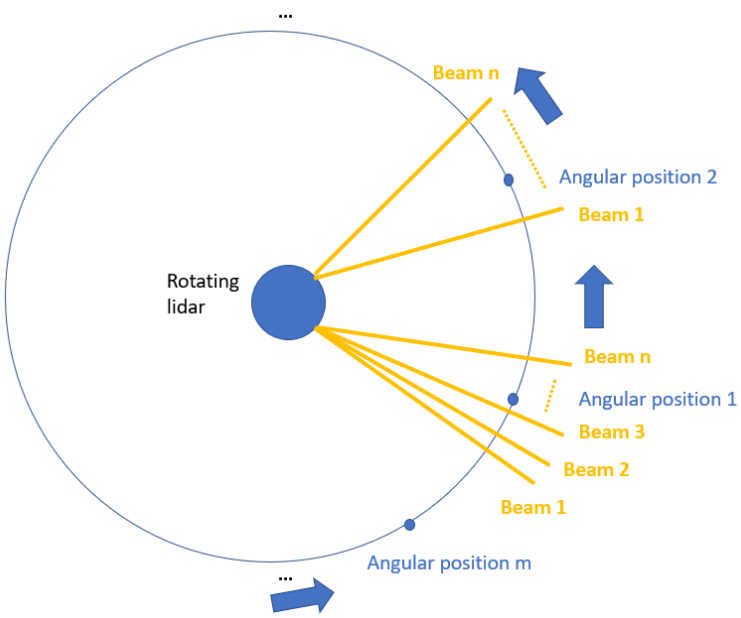


Figure 2.2.: Illustrative Model of a Rotating LiDAR Sensor.

In a typical rotating LiDAR system, each laser diode emits a pulse. The sensor measures the time elapsed for this pulse to return after its reflection from an

object. Using the ToF principle, the sensor then computes the distance to the object [33]. Combining this computed distance with the specific orientation of the emitting laser diode, the 3D coordinates of the reflection point are established.

The system shown in Figure 2.2 illustrates how a rotating LiDAR controls the direction of the beam across its rotation. This method ensures that each angular position adheres to a consistent firing sequence, preventing cross talk. Specifically, as represented in Figure 2.2, the same set of beams (beam 1 to beam n) are fired at each angular position from 1 to m. This synchronized firing sequence enables the LiDAR system to achieve a comprehensive scan as the angular positions from 1 to m collectively span a full 360 degrees. The refresh rate of the LiDAR depends on its rotational speed, with each complete rotation corresponding to a new frame of data.

- **Hybrid LiDAR:** These sensors employ Micro Electronic Mechanical Systems (MEMS) for scanning [4]. They consist of a single laser component and utilize a micro-mirror to reflect the laser pulses in various directions. Although Hybrid LiDARs are more compact and less costly than mechanical LiDARs, their field of view tends to be more limited [33].

- **Solid-State LiDAR:** These sensors use electronic scanning and have no moving parts, leading to smaller form factors and increased reliability. There are two primary types of Solid-State LiDAR: Optical Phased Array (OPA) LiDAR, which uses a multi-light source array for scanning, and Flash LiDAR, which operates similarly to a camera, emitting a wide beam and capturing the returned signals to construct an image [33].

Each LiDAR type has its unique strengths and drawbacks, and substantial research is being conducted to further optimize these technologies and mitigate their limitations.

2.2.2. Implications and Applications of LiDAR Technology

LiDAR's unique capability of creating precise, high-resolution 3D mappings of environments has found invaluable applications across various fields, with its most prominent deployment being in the domain of autonomous vehicles. Here, LiDAR's detailed spatial information significantly enhances object detection, localization, and navigation [34].

Beyond autonomous vehicles, LiDAR has been utilized for a range of activities such as topographical profiling, lunar explorations, and docking procedures [4]. Recent advancements in sensor technology, coupled with reductions in size and cost, have broadened LiDAR's adoption in civilian applications. This includes robotics for obstacle avoidance and navigation, and even integration into consumer electronics such as tablets for applications like 3D face recognition [4].

Despite its myriad advantages, LiDAR does pose certain challenges. Notably, the cost of LiDAR sensors, particularly for high-end models, remains prohibitive [4, 1]. Additionally, LiDAR data, being high-dimensional and unstructured, presents complexities in processing [9], necessitating sophisticated algorithms for its utilization.

2.2.3. Characteristics of Point Cloud Data

LiDAR sensors utilize a combination of angular measurements and ToF computations to generate a data set known as a point cloud. Specifically, the orientation of a return signal is determined by two crucial angular measurements —— azimuth and elevation —— while the ToF principle enables the calculation of the distance from the sensor to the object that reflected the signal.

In the LiDAR sensor context, spherical coordinates (r, θ, ϕ) define each point. Here, r denotes the radial distance from the origin (or the sensor center) to the point, ϕ signifies the azimuth angle measured in the XY-plane from the positive x-axis, and θ represents the elevation angle in the YZ-plane measured from the positive z-axis.

These spherical coordinates are subsequently converted into Cartesian coordinates

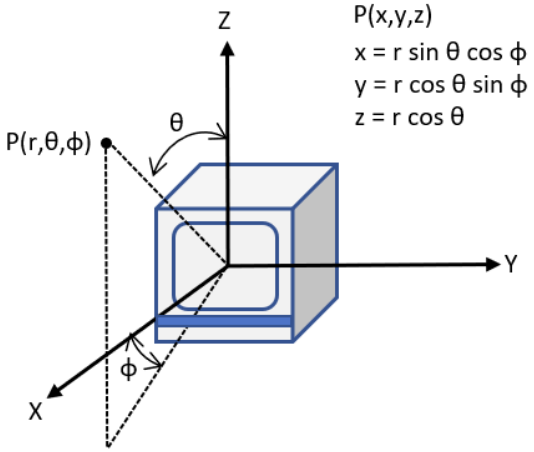


Figure 2.3.: Depiction of the LiDAR Coordinate System. Figure in courtesy of [35].

(x, y, z) through mathematical transformation, as depicted in Figure 2.3.

Each pulse reflection from an object yields a single three-dimensional data point. Over time, these data points cumulatively create a comprehensive point cloud. This assembled point cloud delivers detailed spatial and depth representation of the scanned environment, offering a solid foundation for further analyses and applications, such as 3D object detection, segmentation, and classification.

2.2.4. Challenges of Point Cloud Data

Applying deep learning to LiDAR-based point clouds presents four significant challenges:

- **Absence of Structure:** Raw point cloud data is unstructured and lacks a mesh, making it unsuitable for operations like convolution [9, 18]. Hence, the data must undergo preprocessing before being subjected to deep learning processes.
- **Disorganization:** Various sorting methods for point clouds do not affect the distribution of point clouds in three-dimensional space but impose additional computational burden on the deep learning network [9].

- **Variability in Point Number:** Contrary to images that have a fixed number of pixels, the representation of the same object in point cloud data can significantly vary due to factors such as secondary echoes, spatial occlusions, and sensor discrepancies [9].
- **Sparse Point Cloud Data:** The resolution of LiDAR is generally lower than optical cameras, resulting in substantial loss of information. As the distance between the objects and the LiDAR sensor increases, the density of the point cloud markedly diminishes [9, 18].

Despite these considerations, point clouds exhibit scale invariance [36], meaning that while the target size may vary in an optical image, it remains constant in the point cloud.

2.3. Autonomous Driving data sets

The field of autonomous driving has benefitted significantly from the availability of diverse and comprehensive data sets, including those incorporating LiDAR-generated point cloud data. Various resources such as the Waymo Open data set [37], the Lyft Level 5 data set [38], and the nuScenes data set [39], offer a wealth of multimodal sensor data. However, this work will predominantly focus on the KITTI Vision Benchmark Suite [22], esteemed for its abundant high-resolution stereo and LiDAR data coupled with detailed annotations. For the purpose of this thesis, the term 'KITTI' will refer to this specific data set.

2.3.1. KITTI Database

The value of real-world data in progressing autonomous driving technologies cannot be overstated, given its vital role in training, testing, and evaluation of diverse algorithms. The KITTI data set [22], a result of a collaborative effort between the Karlsruhe Institute

of Technology and Toyota Technological Institute, has risen to prominence as a widely utilized and recognized resource in this sector.

This data set was designed to stimulate research on computer vision and robotic algorithms relevant to autonomous driving [22]. Data was collected in Karlsruhe, Germany, using a sensor setup on a mobile platform that comprised two grayscale and two color cameras, a rotating 3D laser scanner (Velodyne HDL-64E), and a GPS/IMU navigation system [22]. Capturing high-resolution images, laser scans, GPS measurements, and IMU accelerations, the KITTI data set provides an expansive view of various environments categorized as 'Road,' 'City,' 'Residential,' 'Campus,' and 'Person,' allowing algorithmic refinement and validation [22].

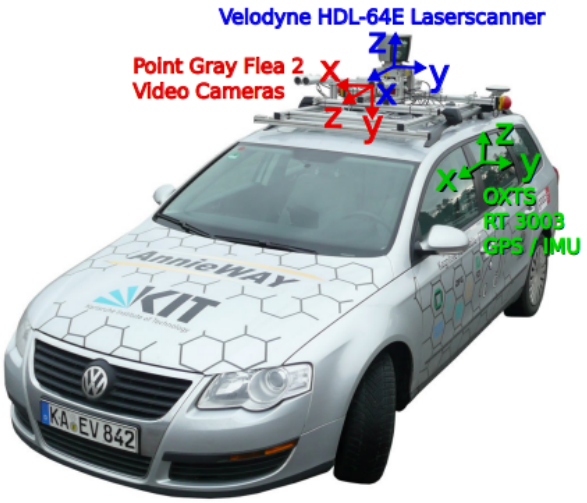


Figure 2.4.: KITTI data set Recording Platform: The VW Passat station wagon, fitted with a suite of sensors, including four video cameras, a rotating 3D laser scanner, and a GPS/IMU inertial navigation system. Figure in courtesy of [22].

As illustrated in Figure 2.4, the KITTI data set reflects the integration of multiple sensor technologies on a recording platform. It includes 7481 labeled frames with 3D coordinates of entities like cars, pedestrians, and trams, plus an additional 7518 unlabeled

frames for performance evaluation through participation in the KITTI competition.

Despite its comprehensive nature, the KITTI data set is not without limitations [9, 1, 18]. The data was primarily collected under daylight conditions [18] with a consistent sensor setup, leading to a lack of diversity in lighting conditions and sensor configurations. The data set's class frequency is skewed towards car objects, and the majority of the objects are oriented towards the ego-vehicle, restricting the variety of object orientations [1]. While these issues may limit the data set's full potential in representing diverse real-world scenarios, the KITTI data set remains an instrumental tool in autonomous driving research and development.

In this thesis, 'real-world data' and 'KITTI data set' and 'KITTI' will be used interchangeably. As the objective is to generate a simulated counterpart of the KITTI data set, the 'Velodyne HDL-64E' [23] laser scanner used in KITTI will be modeled first in the simulation section, 3.1.1.

2.3.2. Artificial Data

The success of supervised learning is often tied to the diversity and quality of training data sets, typically consisting of labeled data frames collected from vehicle sensors like LiDAR, RADAR, and cameras [14]. However, manually labeling this data can be expensive and time-consuming [6].

In response to these challenges, researchers have explored the generation of synthetic data sets. Initial approaches involved data augmentation [14], but these methods were limited by the quality of the original data. An innovative solution is the creation of entirely artificial data by capturing sensor readings within simulated environments. This approach offers key benefits, including the ability to generate nearly unlimited data samples, independence from real-world data, and the capacity to simulate rare scenarios for safety analysis. However, the quality of these artificial data sets largely depends on the realism of the simulation environments [14].

Here are some significant tools used to create synthetic point cloud data for autonomous driving applications:

- **Nvidia Drive Sim** is a simulator developed by Nvidia for testing and developing autonomous vehicle technologies [40]. It can simulate a wide variety of driving environments, weather, and lighting conditions, making it an ideal tool for creating diverse scenarios for autonomous vehicles. However, it is not open-source, and free-to-use licenses are not available.

- **AirSim**, an open-source simulator by Microsoft for drones and cars, offers realistic environments and detailed vehicle dynamics [41]. While it can simulate sensor data, including LiDAR point clouds, it lacks detailed control over each agent.

- **Sim4CV**, a tool designed for generating synthetic data for computer vision applications, includes ray-casting LiDAR models for creating synthetic point clouds [42]. Its capability to generate ground-truth bounding boxes for object detection is a significant feature.

- **Grand Theft Auto V (GTA-V)** is a popular video game used for synthetic data generation due to its detailed and dynamic virtual environment [43]. In GTA-V, a virtual LiDAR scanner is created within the game, and as the 'ego car' navigates the game environment, LiDAR point clouds are collected and images are captured simultaneously [5]. However, GTA-V has certain limitations, including simplified physical models and limited object categories [5, 9].

- **CARLA** is an open-source simulator developed by Intel Labs and Toyota Research Institute [44]. Built on the Unreal Engine, it provides a realistic and customizable simulation environment. CARLA's unique feature is its ability to simulate sensor data, such as LiDAR point cloud data, using ray casting [6]. This feature allows CARLA to generate labeled point cloud data, a critical resource for supervised learning.

In order to produce synthetic point cloud data, this thesis primarily employs the Ansys AVxcelerate Sensors Simulator. The following section will provide an overview about its capabilities.

2.3.2.1. Ansys AVxcelerate Sensors Simulator

The Ansys AVxcelerate Sensors Simulator [21], serves as a pivotal tool in this study due to its high-fidelity, physics-based simulation capabilities, specifically optimized for LiDAR systems. For a detailed overview, the reader is directed to AVX's whitepaper and datasheet [21].

A paramount feature of Ansys AVxcelerate Sensors Simulator (AVX) lies in its in-depth consideration of physical parameters, maintaining rigorous attention to the environmental context — including terrain, structures, vehicles, and pedestrians — and the LiDAR sensor, which encompasses the laser emitter and receiver. The representation of these components within AVX simulations ensures comprehensive and realistic modeling of LiDAR systems.

AVX offers a parametric model of LiDAR, effectively simulating a variety of sensor devices, including Flash, Rotating, and Scanning LiDAR. This model faithfully reproduces all LiDAR device components, generating an array of output types such as raw waveforms, intensity maps, point clouds, and ground truth outputs.

A defining trait of AVX is its simulation capability, which facilitates the generation of sensor-realistic point clouds. This robust LiDAR simulation enables users to evaluate supplier performance, integrate LiDAR systems at different design stages, validate LiDAR-based control laws, and explore a wide array of 'what-if' scenarios. AVX utilizes Bidirectional Reflectance Distribution Function (BRDF) [45] to simulate environmental physical interactions and incorporates physical models of the sky and sun, yielding a comprehensive environmental definition.

As illustrated in Figure 2.5, the architecture of LiDAR simulations in AVX includes distinct modules such as 'Emitter', 'Receiver', and 'Motion'. The 'Motion' entity sets the dynamics of the system, influencing both the 'Emitter' and 'Receiver'.

The 'Emitter' module plays a crucial role in defining the laser pulse properties. Subsequent to pulse emission, there is interaction with the targeted object. Post-interaction, the software utilizes Bidirectional Reflectance Distribution Function (BRDF) [45]. BRDFs articulate the reflection pattern of the emitted pulse upon striking an

opaque surface, offering the ratio of reflected radiance. This contributes to physically based rendering.

The 'Optics' section of the 'Receiver' captures the back-scattered radiance and transmutes it into radiometric power, channeling it to the 'Photodetector'. Following this, the 'Photodetector' situated within the 'Opto-Electronic' compartment transforms this radiometric power into an electrical current.

The 'Opto-Electronic' module, a component of the 'Receiver', encompasses both the 'Photodetector' and the 'Analogue to Digital Converter (ADC)'. The ADC subsequently converts this electrical current into a digital waveform. Then, the 'Default Processing' component within the 'Receiver' transforms this digital waveform into ToF signals and intensities.

The software, as a result, is capable of generating a multitude of outputs, including calculated point clouds, waveform signals, and interaction metrics termed as 'contributions'. These contributions demonstrate the interplay between specific beams and objects present in the scene.

The 'receiver channel', consisting of the 'Default Processing', 'Opto-Electronic', and 'Optics' components, combines these individual segments into a cohesive workflow. This workflow is visually depicted in Figure 2.5, where each component is symbolized through simplified icons, and the input-output dynamics are demonstrated by yellow arrows.

For a detailed exploration of the parameterization and other aspects of AVX's modules, refer to section 3.1.1.

2.4. LiDAR-Based 3D Object Detection Techniques

LiDAR-based 3D Object Detection techniques are central to disciplines such as autonomous navigation and robotics due to their ability to accurately identify and position objects within a three-dimensional space [4]. While varying in specific implementation, these methodologies can be broadly categorized into two primary types based on their

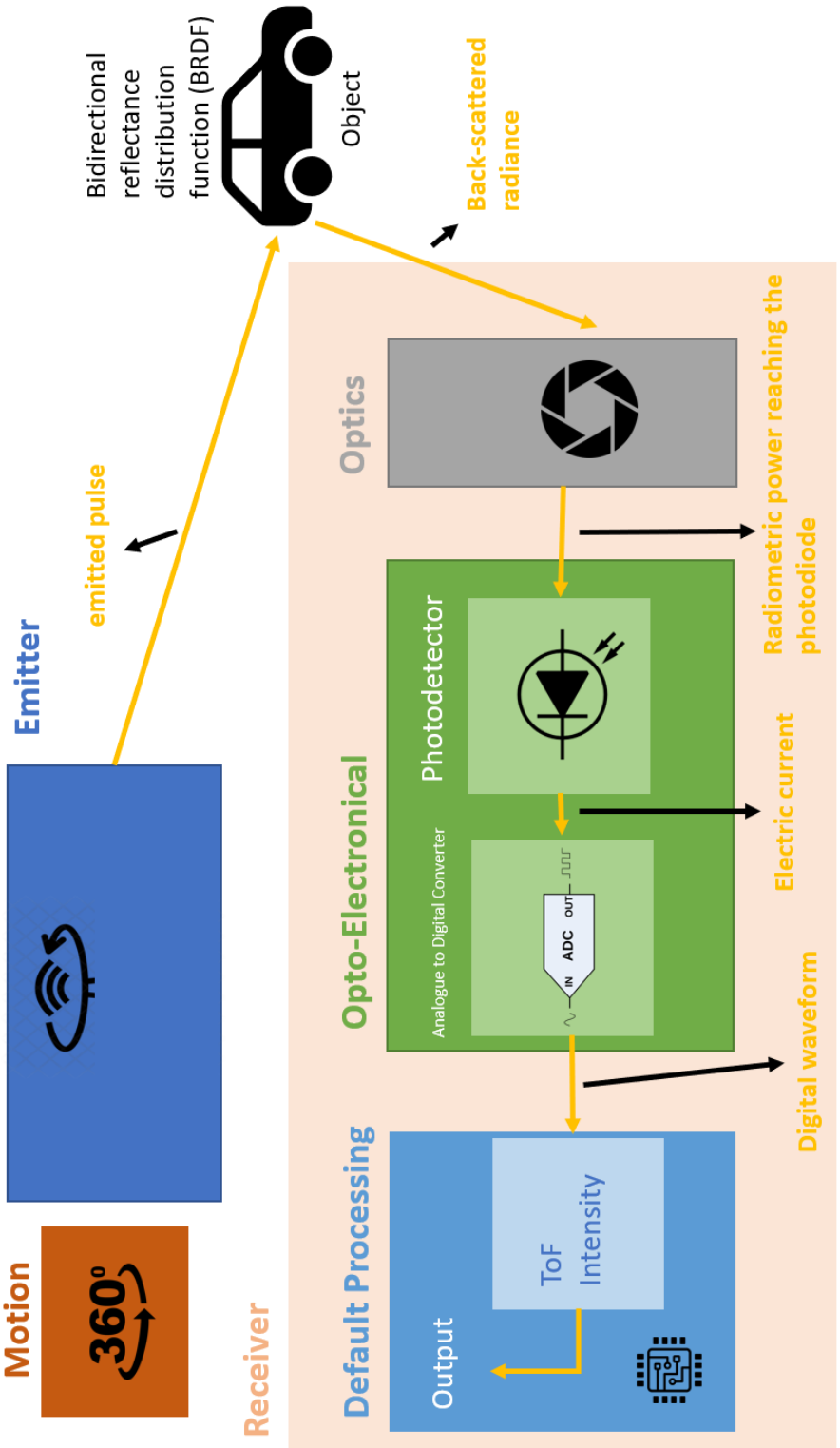


Figure 2.5.: Ansys AVxcelerate Sensors Simulator: Component Interactions and Workflow.

network architecture: single-stage and two-stage methods [9, 27, 26, 10, 4, 30, 2]. The fundamental distinction between these categories resides in the number of processing stages in their respective detection pipelines.

Detailed analyses of these architectures will be expounded in subsequent sections. The examination will commence with a focus on the distinct processing stages (section 2.4.1), followed by a comprehensive assessment from a featurization perspective (section 2.4.2).

2.4.1. A Comparison of Single-Stage and Two-Stage Approaches

The dichotomy of single-stage and two-stage network architectures is visually represented in Figure 2.6. In the ensuing section, a comprehensive examination of LiDAR-based 3D object detection methodologies and their categorization based on the number of processing stages is presented.

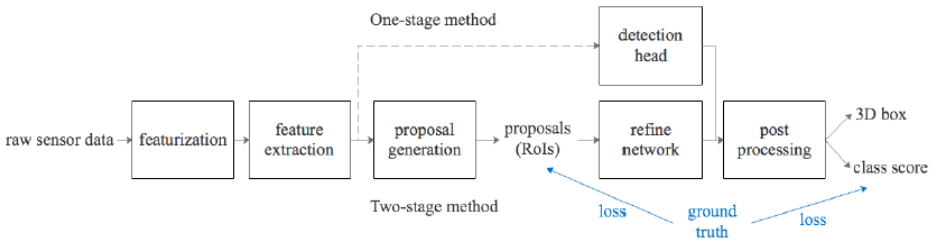


Figure 2.6.: Structural comparison of single-stage and two-stage 3D object detection methodologies. Figure in courtesy of [4].

2.4.1.1. Single-Stage Detection Methods

Single-stage 3D LiDAR-based object detection algorithms are designed to process the input data in a single pass. They eradicate the necessity for an intermediary region proposal phase, a key attribute of two-stage detectors. As single-stage detectors are typically faster and more efficient than their two-stage counterparts, they are often favored in applications requiring real-time performance, such as autonomous driving

[28, 2, 27, 26].

Single-stage detectors begin with raw LiDAR point cloud data and output bounding boxes and class labels for the detected objects directly [46, 47]. The sequence of operations include:

- **Preprocessing:** The preliminary stage entails formatting the raw point cloud data to facilitate subsequent detection processes. This usually involves data conversion into a manageable format like voxels [28] or a bird's-eye view (BEV) representation [48]. However, certain detectors can operate directly on the raw point cloud data [49].

- **Feature Extraction:** Once the data has been transformed into an appropriate format, it is subsequently directed through a sequence of convolutional layers. The extraction methodology can differ based on the type of detector: 3D convolutional layers are typically leveraged in voxel-based detectors, whereas two-dimensional (2D) convolutional layers are utilized in BEV-based detectors. For detectors operating directly on point cloud data, PointNet-like architectures [50] are adopted. These architectures use the 3D coordinates of the point cloud as inputs, process them to extract key point features, and ultimately yield outputs in the form of either object class labels or per-point segmentation labels.

- **Bounding Box Regression and Classification:** The isolated features are then passed through fully connected layers to predict the bounding boxes and class labels of the detected objects. Typically, this includes a regression layer for bounding box coordinates and a classification layer for the class labels [46].

- **Postprocessing:** The final detector output may require additional refinement to fine-tune the bounding boxes and eliminate duplicates. This could include Non-maximum Suppression (NMS) for removing overlapping boxes and score thresholding to discard low-confidence detections [2, 30, 9].

A notable implementation of a single-stage 3D LiDAR-based object detector is

PointPillars [2]. This innovative method operates on a pseudo-image representation of the point cloud data, encoding the points within vertical columns, or 'pillars'. This allows the network to leverage efficient 2D convolution operations while retaining the 3D information intrinsic to the point cloud. The network subsequently applies bounding box regression and classification directly to the pseudo-image for object detection.

Although single-stage 3D LiDAR-based object detectors provide a rapid and efficient approach to object detection within point cloud data, they might occasionally compromise detection accuracy for speed, particularly when compared to two-stage detectors [26].

2.4.1.2. Two-Stage Detection Methods

Two-stage detectors, in contrast to their single-stage counterparts, engage an intricate two-step procedure. They initially generate preliminary region proposals, which are further refined in the second stage to enhance prediction accuracy. This refinement stage adjusts the original proposals using feature alignment to achieve a superior level of precision [12].

Dual-stage detectors, such as PointRCNN [51], utilize PointNet [50] to integrate raw points and semantic features extracted from region proposals, followed by a refinement phase in the second stage. Similarly, Part-A^2 [52] is aware of which part of the object a point belongs to, leveraging this intra-object part location information to attain higher performance. STD [53] adopts spherical anchors to create proposals and includes a segmentation branch to reduce the number of positive anchors, thus boosting performance.

Despite the improved accuracy generally delivered by two-stage detectors, they require a more significant computational overhead due to their two-step process. Hence, these may not be the optimal choice for applications where real-time performance is crucial.

2.4.2. LiDAR-based 3D Object Detection Methods from Featurization Perspective

This section provides an in-depth examination of several methodologies for LiDAR-based 3D object detection, with an emphasis on the process of featurization. Mainly, the techniques explored are the projection-based, voxel-based, and point-based methods [9, 4, 30, 1, 10]. Each method presents distinctive advantages and trade-offs when dealing with the intricacies of 3D point cloud data. The intention of this section is to shed light on these techniques and accentuate their specific strengths and limitations.

2.4.2.1. Projection-Based Techniques

Projection-based methods have gained considerable traction in the field of LiDAR-based 3D object detection. The underlying principle of these methods involves transforming the 3D point cloud data into 2D representations, applying various projection techniques such as plane (image), spherical, cylindrical, or BEV [10]. The resulting 2D data is processed using traditional 2D methods before being converted back into 3D to construct the bounding boxes for the detected objects.

Two main variants of projection-based methods are Front View (FV) based-methods and bird's-eye view (BEV)-based methods. Each comes with unique benefits and challenges, depending on the context and objectives of the object detection task.

- **Front View (FV) Based-Methods:** FV-based methods convert the 3D point cloud data into a 2D plane, similar to a front view from the LiDAR sensor's perspective. In this approach, each point's depth value is transformed into a grayscale image, where the pixel intensity represents the distance from the sensor. The depth information retained in the FV map can capture dependencies across different views, such as raw point cloud and camera images, thereby augmenting sensor fusion capabilities.

  However, FV-based methods have significant drawbacks. Due to the perspective nature of this projection, objects' perceived scale varies with their distance from

the sensor, potentially causing detection problems [4]. Furthermore, objects may become occluded in the FV map, and the projection may cause loss of detail and depth information [4].

- **BEV Based-Methods:** BEV-based methods convert the 3D point cloud data into a 2D plane from a top-down perspective, giving a BEV of the scene. The BEV representation, typically encoded by height, intensity (reflectance value), and density, overcomes issues related to object occlusion and scale changes common in other 2D plane representations [10]. In a BEV representation, objects occupy distinct spaces on the map, reducing the chances of occlusion.

  Despite its benefits, the BEV method has certain limitations. Notably, the BEV representation may cause height compression, leading to potential semantic ambiguity [10]. Furthermore, challenges like the small effective area, sparse representation, and difficulty handling vertical space occlusion can limit the overall accuracy of object detection [4]. However, BEV-based methods are still widely used due to their efficiency and precise object localization.

A crucial element for the success of both FV and BEV-based methods is the use of Convolutional Neural Network (CNN). The compact and ordered 2D maps produced from the projection process pave the way for leveraging well-established 2D CNNs for 3D tasks [12].

However, these methods' effectiveness can be restricted due to their heavy dependence on 2D detection algorithms' evolution and manually crafted features to maintain the spatial attributes of the 2D projection map [4]. The resolution of the projection map and the number of feature channels can influence computational efficiency, and the irreversible information loss caused by projection can further impair object detection accuracy [4].

2.4.2.2. Voxel-Based Techniques

Voxel-based methods have become prominent in the field of 3D LiDAR object detection by offering a unique solution to manage point cloud data's complex nature. These methods pivot on converting point cloud data, usually sparse and unstructured, into a structured format known as voxels, suitable for convolution filter operations.

The voxelization process entails segmenting the 3D space into a grid of fixed-size voxels, each containing a collection of unstructured points. This technique maintains the 3D structural integrity of the original point cloud data by accounting for three-dimensional space division. However, a major challenge is the inherent sparsity of LiDAR point clouds, which leads to a substantial number of empty voxels [2, 10]. The computational demand considerably increases with finer voxel resolution due to this sparsity [10].

Voxel-based methods have significantly evolved over time, moving from basic statistical feature extraction to advanced machine learning models that capture 3D geometric characteristics within voxels [4]. Despite these advancements, the sparsity of the voxel representation and the resulting computational demand can limit real-time performance [4].

PointPillars [2], a more efficient variant of voxel-based methods, addresses these limitations. Instead of using full 3D voxels, PointPillars converts point cloud data into vertical columns or pillars, avoiding the computational load associated with full 3D convolutions. The pillars are then processed using a neural network similar to PointNet [50], to extract point-wise features. The resulting pseudo BEV map preserves the scene's 3D structure and is further processed by a 2D detector, striking a balance between computational efficiency and precision in 3D object detection.

In the context of this thesis, the focus will primarily be on the PointPillars architecture. This model was chosen for its robust performance, computational efficiency, and ability to manage the complexities of 3D object detection tasks, striking a balance between the different methodologies' strengths and weaknesses. Section 2.5 will explain its structural components.

2.4.2.3. Point-Based Techniques

In the context of 3D object detection using LiDAR, point-based methods take a unique stance by working directly with raw point cloud data. This direct approach deviates from the traditional methodologies like projection-based or voxel-based techniques that necessitate the transformation of data into other formats such as 2D images or 3D voxels. One of the principal advantages of this direct processing technique is the preservation of the original structure and traits of the 3D data, which could potentially lead to superior results in object detection.

The core methodology in point-based techniques involves processing the irregular point cloud directly. Pointwise features are extracted by aggregating the features from neighboring local points, thereby facilitating the generation of predictions on a per-point basis. This characteristic distinguishes point-based methods from other techniques.

Despite the challenges inherent to point-based methods, they have proven to be competitive in 3D object detection tasks, primarily due to their ability to fully leverage the 3D geometric and shape characteristics of point cloud data without significant information loss [4]. Additionally, they demonstrate higher memory efficiency during sparse point processing because of lower Graphics Processing Unit (GPU) memory requirements [4].

However, point-based methods are not devoid of limitations. They require random memory access for generating point-wise features, and the production of object proposals is contingent on either image-based object generators or computationally expensive per-point foreground segmentation networks [4]. Moreover, despite providing a direct pathway to 3D object detection, point-based methods can require substantial computational resources due to the need to process large quantities of points and the complexity of the networks used for feature extraction and proposal generation [10].

2.5. PointPillars Network

The PointPillars Network [2], featuring exclusively 2D convolutional layers, proposes an effective strategy for handling LiDAR point cloud data. This network employs an innovative encoder that learns features on vertical 'pillars' within the point cloud, facilitating prediction of 3D oriented boxes for objects.

Distinct advantages of PointPillars include harnessing the full information within the point cloud by relying on learned features rather than fixed encoders [2]. Additionally, the pillar-centric approach negates the need for binning in the vertical direction, a required step in voxel-based methods [2]. Furthermore, the PointPillars' architecture enables all crucial operations to be transposed into 2D convolutions, rendering it highly efficient for GPU computations [2]. With an operation speed of 62 Hz, it surpasses previous solutions, thus presenting an effective trade-off between speed and accuracy [2].

The focus of this thesis lies in the exploration of the PointPillar network, given its high-performance and streamlined architecture. The study aims to derive insights that may be extrapolated to other networks.

2.5.1. Network Architecture

The PointPillars network (Figure 2.7) is designed to take point cloud data as input and generate predictions of 3D bounding boxes for objects such as vehicles, pedestrians, and cyclists. It comprises three key interconnected components:

- **The Pillar Feature Network:** This functions as the initial encoder, transforming the point cloud data into a pseudo-image for subsequent convolutional processing.
- **The Convolutional Backbone:** This intermediate layer applies 2D convolutional operations to the pseudo-image, yielding a more abstract and feature-rich representation that aids object detection.
- **The Detection Head:** This final component leverages the high-level features from

the backbone to identify and regress 3D bounding boxes around detected objects.

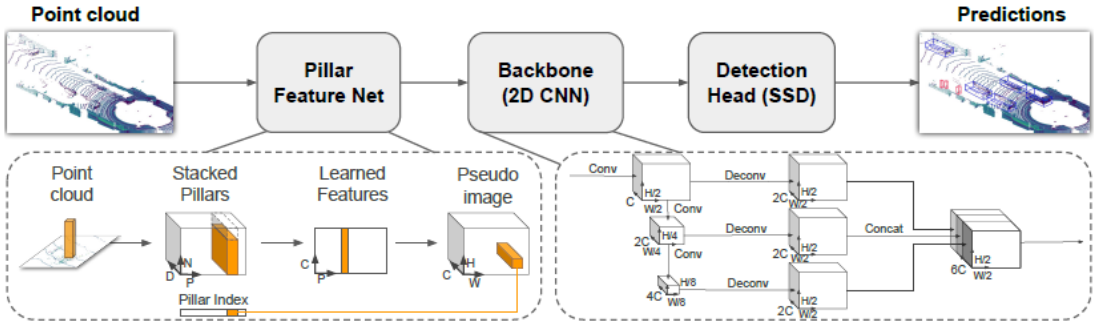


Figure 2.7.: Illustration of the PointPillars network architecture, specifically tailored for car detection. Figure in courtesy of [2].

2.5.1.1. The Pillar Feature Network

In the PointPillars architecture, the Pillar Feature Network facilitates the translation of 3D point cloud data into a pseudo-image, rendering it suitable for 2D convolutional processing. Each point within the point cloud, denoted by l, with its corresponding 3D coordinates x, y, z and reflectance r, is initially segmented into a series of pillars, P, on the $x-y$ plane, yielding a total of $|P| = B$ pillars. In a departure from voxel-based methodologies, PointPillars strategically omits the need for binning along the z dimension.

Each pillar houses points that are enhanced with the following set of features: x_c, y_c, z_c, x_p, and y_p. Here, subscript c denotes the deviation from the pillar's mean, whereas subscript p refers to the displacement from the center (x, y) of the pillar, yielding a LiDAR point l with a dimensionality of $D = 9$.

Taking advantage of the characteristic sparsity of point clouds, PointPillars imposes an upper limit on the quantity of non-empty pillars per sample (P) and points per pillar (N), thereby generating a dense tensor of dimensions (D, P, N). Any excess or insufficient data within a given sample or pillar is rectified through random sampling or zero padding, respectively.

Subsequently, a PointNet-like operation [50] is employed, wherein a linear layer is applied to each point, followed by Batch Normalization (BatchNorm) [54] and Rectified Linear Unit (ReLU) [55], thereby generating a (C, P, N) tensor. A max operation conducted across the points within each pillar condenses this to a (C, P) tensor. The application of a 1x1 convolution on the linear layer facilitates efficient computation. The resulting features are then remapped to their original pillar positions, producing a pseudo-image of dimensions (C, H, W), where H and W represent the height and width of the pseudo-image, respectively.

2.5.1.2. Backbone

The backbone, depicted in Figure 2.7, is an integral component of the PointPillars network. It is primarily responsible for extracting a high-dimensional feature map from the pseudo-image input, providing a foundation for subsequent object detection tasks.

The backbone architecture is divided into two sub-networks: a top-down sub-network, which progressively reduces the spatial resolution of features, and a subsequent network responsible for upsampling and feature concatenation.

The top-down sub-network consists of several blocks, each designated as Block(S, L, F), where S represents the stride relative to the original pseudo-image, L corresponds to the number of 3x3 2D convolutional layers within a block, and F denotes output channels from each layer. To preserve a stride of S, the initial convolution in each block operates at a stride of $\frac{S}{S_{in}}$, whereas subsequent convolutions within the block operate at a stride of 1. Each convolution is succeeded by BatchNorm and ReLU functions.

Following the processing in the top-down sub-network, the features undergo upsampling and are then concatenated. This upsampling operation, represented as Up(S_{in}, S_{out}, F), transitions from stride S_{in} to S_{out} via a transposed 2D convolution, ultimately producing F output features. After upsampling, these features are subjected to BatchNorm and ReLU functions. The final output incorporates a composite of features extracted at multiple strides through concatenation.

2.5.1.3. Detection Head

The detection head forms a pivotal part of the PointPillars network, specializing in object detection and classification. This segment harnesses the high-level features distilled by the backbone network to discern objects and infer their properties, including position, size, and orientation.

The setup for the detection head in PointPillars parallels the framework of the Single Shot Detector (SSD) [46]. This framework utilizes a series of predefined boxes, or 'prior boxes' or 'default boxes', with varying aspect ratios at each location on the feature map. The network predicts a class and four offsets for each box, enabling the box to be adjusted to closely align with the object's correct bounding box. PointPillars, akin to SSD, employs 2D Intersection over Union (IoU) [56] for matching these prior boxes with the ground truth boxes. Notably, the height and elevation of the 3D bounding boxes serve as additional regression targets, rather than being utilized in the matching process.

The incorporation of the SSD framework aligns with PointPillars' design objective, which is to consolidate the benefits of 2D convolutional networks while addressing the requirement for 3D object detection.

2.6. Related Work

The demand for large and accurate data sets has grown with the rise of object detection algorithms. Researchers have started to use synthetic data as a solution. This approach allows for control over the data and can create unusual or hard-to-find real-world situations. This section will review the important studies that have used synthetic data to improve object detection models.

- Dworak et. al [14] effectively exploited the CARLA simulator to supplement the training sets required for neural network algorithms. The generated artificial data proved to be advantageous, capturing a variety of edge-case scenarios. The

research employed VoxelNet, YOLO3D, and PointPillars — three state-of-the-art neural network architectures — during the validation process. The researchers made several crucial discoveries. Synthetic data proved invaluable for validating novel solutions as the related key performance indicators could be mapped onto real-world scenarios post pre-training with real data. However, merging synthetic and real data into a single data set did not yield improvement; it instead resulted in a performance decline. The research, thus, advised the model fine-tuning approach for enhancing real-world performance using synthetic data, especially when the synthetic data set is significantly larger than the real-world counterpart.

- Chitnis et al. [13] addressed the absence of semantic information in raw LiDAR point clouds, which is critical for several applications. Noting the low accuracies of existing object classification methods in mobile LiDAR data due to the shortage of 3D training samples, they proposed a generative model for the creation of synthetic 3D point segments. Their 3D Adversarial Autoencoder was trained to produce synthetic point segments that mirror real ones in appearance and geometric features. Evaluations using a PointNet-like classifier demonstrated that supplementing training data with synthetic samples significantly improved classification performance.

- Addressing the dearth of publicly accessible annotated data, Neri et al. [12] engineered a virtual railway environment to generate a synthetic annotated railway point cloud data set. This innovative approach facilitated the modeling of targeted landmarks like traffic lights and rail tracks, using the integration between Unreal Engine and Matlab. The results indicated promising outcomes for the proposed detection scheme using the synthetic data set.

- Hahner et al. [19] dealt with 3D object detection in foggy conditions using LiDAR. They introduced a method to simulate physically accurate fog in clear-weather scenes, which enabled the generation of large-scale foggy training data without additional costs. Experiments demonstrated that their fog simulation technique

substantially improved 3D object detection in foggy situations.

- In a related study, Hahner et al. [11] tackled 3D object detection in snowy weather conditions. They proposed a method to simulate the impact of snowfall on clear-weather LiDAR point clouds, leading to the creation of partially synthetic snowy LiDAR data for training 3D object detection models. Extensive testing showed significant performance improvements on real snowy data sets compared to clear-weather baselines and other simulation methods, without compromising performance in clear conditions.

- Motivated by the challenges of human labelling, limited access to advanced sensors, and the future prospect of high-resolution LiDAR sensors, Cheng [18] created a synthetic data set using the CARLA simulator. The generated data showcased the potential and effectiveness of synthetic data for enhancing object detection.

- LiDARsim, introduced by Manivasagam et al. [17], seeks to address the significant sim-to-real domain gap prevalent in the field. This gap often arises from the reliance on handcrafted 3D assets and simplified physics assumptions in traditional simulators like CARLA and Airsim. LiDARsim employs a two-stage approach involving assets creation and sensor simulation. The first stage captures data from vehicles driving through multiple cities, creating 3D static maps and dynamic object meshes, providing a more accurate representation of the real world than traditional artist-designed virtual worlds. In the sensor simulation stage, it combines physics-based and learning-based simulations to generate realistic LiDAR point clouds. This approach has shown effectiveness in enhancing confidence in the testing of full autonomy stacks, particularly in rare and safety-critical scenarios.

- The introduction of SqueezeSegV2 by Wu et al. [29] represents an advancement in point cloud segmentation. It not only exhibits enhanced robustness against dropout noise in LiDAR point clouds but also offers higher accuracy when trained on real data. Recognizing the domain shift issue often caused by synthetic data

generated through simulators like GTA-V, they propose a domain-adaptation training pipeline that successfully boosted test accuracy on real-world data from 29.0 % to 57.4 % when trained on synthetic data.

- In their study, Yue et al. [5] proposed a novel framework leveraging the rich virtual environment of GTA-V to generate simulated LiDAR point clouds and corresponding high-fidelity images. The authors used a simultaneous collection of LiDAR point clouds and game screen captures, ensuring consistency between point clouds and images. Their approach facilitated automatic annotation on the collected data, showing improved validation accuracy for a point cloud segmentation task by 9 %.

- Fang et al. [8] introduced a unique LiDAR simulator that augments real point clouds with synthetic obstacles. Their approach circumvents the need for high-fidelity background Computer Aided Design (CAD) models and reveals that detectors trained with simulated LiDAR point clouds alone can perform almost as well as those trained with real data. The performance gap between detectors trained with real or simulated data was within two percentage points, highlighting the potential of their simulator for generating extensive labeled data without the need for manual labeling.

- Beltran et al. [16] tackled the data limitation problem by introducing a method capable of constructing a 3D representation of a scene using real LiDAR data. This method generates synthetic point clouds mimicking the output of different sensor configurations, enabling the simulation of various sensor models and creating additional data sets derived from existing ones without the need for further labeling.

- Wang et al. [15] contributed an open-source method capable of automatically generating 3D annotated LiDAR point clouds. The authors highlighted the scalability and configurability of their approach, showing that synthetic data can

boost the performance of deep learning models, reduce the need for manually labeled training data, and help mitigate the data set bias problem.

- Jabłoński et. al [6] explored the effectiveness of training a LiDAR pedestrian detection algorithm for autonomous vehicles using synthetic and mixed data. Leveraging the CARLA 3D engine, data was automatically labeled and shaped into range images for deep learning. The detectors' efficiency was assessed using the Waymo open data set and the YOLOv4 neural network architecture for pedestrian detection. Results indicated that synthetic data can enhance detector performance, with a YOLOv4 model trained on mixed data improving precision and recall.

In summary, these studies show the value of synthetic data in improving object detection algorithms. This thesis aims to build on this work by investigating the use of synthetic data generated by AVX [21] with the aim of improving the performance of object detection models.

3. Methodology

3.1. Sensor Modeling

The LiDAR Sensor Modeling section of this thesis delves into the detailed modeling process of a specific LiDAR sensor, the KITTI LiDAR model, using AVX.

3.1.1. LiDAR Sensor Modeling

This section delineates the process of creating the KITTI LiDAR model using the Ansys AVxcelerate Sensors Simulator (AVX). This simulator, through its Sensor Modeler interface, provides an array of sensor models such as cameras, fisheye cameras, rotating LiDARs, flash LiDARs, and radar, allowing for tailored sensor model development.

The focus of this investigation lies with a rotating LiDAR system. As outlined in section 2.3.1, the sensor assembly is affixed to the platform demonstrated in Figure 2.4. This assembly comprises two grayscale and two color cameras, a 3D rotating laser scanner, and a combined GPS/IMU navigation system.

The principal object of interest is the 3D rotating laser scanner, specifically, the Velodyne HDL-64E, as detailed in [22]. The characteristics of this laser scanner are as follows:

- Velodyne HDL-64E rotating 3D laser scanner operating at 10 Hz with 64 beams, 0.09° angular resolution, 2 cm distance accuracy, collecting approximately 1.3 million points per second. It has a field of view of 360° horizontally, 26.8° vertically, and a range of 120 m [22].

Building on this information and additional specifications outlined in the user manual [Refer to A.1], the LiDAR instrument will be modelled.

While the overview of Ansys AVxcelerate Sensors Simulator's component interaction and workflow, depicted in section 2.3.2.1 and Figure 2.5, provides an initial understanding of the system, it is also vital to delve into the parametrization of each component.

Figure 3.1 elucidates the parameters available within each component, namely 'Motion,' 'Emitter,' 'Optics,' 'Opto-Electronical,' and 'Default Processing.' It is effectively an augmented version of Figure 2.5, replacing icons with parameters to offer a comprehensive understanding of the system's parametrization.

The analysis begins with the 'Motion' component.

3.1.1.1. Motion

This section describes the dynamic behavior of the system. The dynamic motion of the rotating LiDAR system can be modeled using two parameters:

- **Rotation Speed:** This measures the rotational speed of the LiDAR, quantified in rotations per second. According to [22], this value is set to 10 Hz.
- **Firing Frequency:** This signifies the number of 'fans' (firing sequences) the LiDAR system emits per second. Based on the data in A.1, there exists a 57.6 µs interval between two successive firings of the same laser, equating to a firing frequency of 17 361.11 Hz. After rounding, this value is set to 17 361 Hz.

Figure 3.2 offers an illustrative depiction of the firing process. For each designated instance t_i, the equivalent 'Fan i', as displayed in Figure 3.2b, is activated. Particularly, for the Velodyne HDL-64E, there is an emission of 64 beams per t_i. Given the rotational dynamics of the emitter and receiver, as demonstrated in Figure 3.2a, the system can record a comprehensive 360° horizontal field of view. Moreover, the computation $1/(t_i - t_{i-1})$, as indicated in Figure 3.2a, yields the firing frequency value.

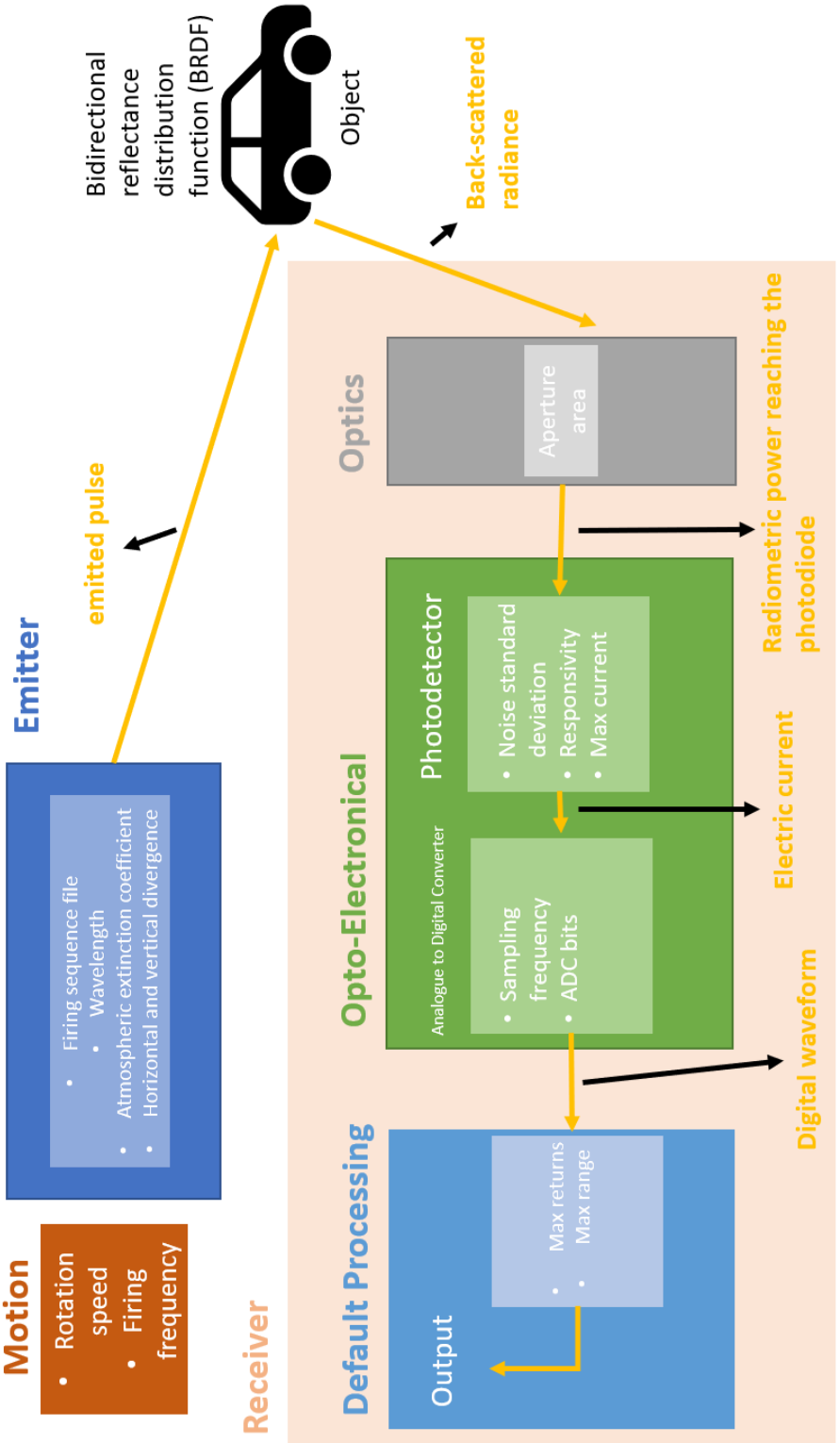
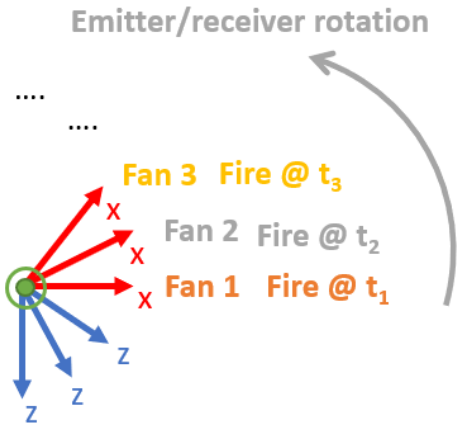
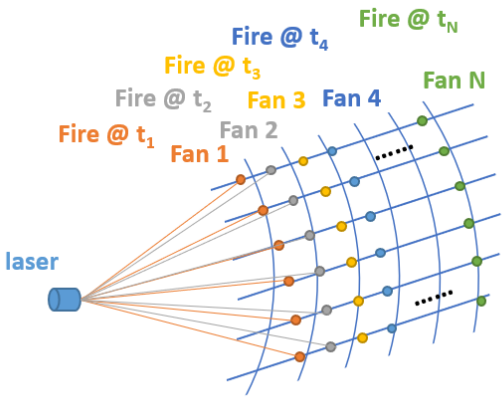


Figure 3.1.: Overview of Parameters Within the Ansys AVxcelerate Sensors Simulator Components.



(a) Visualization of the LiDAR's rotating firings. The same firing pattern is executed at each instance t_i.



(b) Visualization of 'fans', or firing sequences.

Figure 3.2.: Illustrative Depictions of Rotating LiDAR Firing Dynamics and Fan Sequences.

3.1.1.2. Emitter

The Emitter module within the AVX workflow, as displayed in Figure 3.1, includes a set of key parameters:

1. **Firing Sequence File:** The sequential firings or 'flashes' shaping the fan pattern of the emitter are determined by this text file. It is significant to highlight that this firing sequence is not provided by Velodyne, the manufacturer of the LiDAR system, in their user manual. This model instead utilizes a firing sequence file retrieved from the Ansys Database. This file comprises crucial details such as the pulse type, the number of beams (64 specifically for this LiDAR model), the timing of each firing, the energy of every pulse, and the angular data for each individual beam. Figure 3.2b offers a conceptual representation of this process. During each firing instance t_i, the software constructs the fan pattern by leveraging the information available in the firing sequence file.

2. **Wavelength:** This parameter denotes the wavelength of the emitted laser beam. As indicated in A.1, the wavelength for this LiDAR model is set at 905 nm.

3. **Atmospheric Extinction Coefficient:** The atmospheric extinction coefficient [57] accounts for the atmospheric attenuation, which is the loss of energy experienced by a laser beam as it propagates through the atmosphere due to factors such as scattering, absorption by atmospheric gases, particles in the air, and molecular absorption. For this model, the atmospheric extinction coefficient is set at $0.161\,\mathrm{km}^{-1}$, assuming a homogeneous atmosphere.

4. **Horizontal and Vertical Divergence:** The horizontal and vertical divergence terms define the spread of the emitted laser beam in their respective planes. The divergence, measured in radians or degrees, represents the increase in beam diameter with the distance from the optical aperture. Smaller divergence values imply a more focused beam, while larger values denote a broader spread of the beam as it travels. Due to a lack of explicit beam angle measurements in the

LiDAR's datasheet and the unavailability of facilities for direct measurement, default values recommended by the software (0.002 rad for both horizontal and vertical divergence) were used in this model.

3.1.1.3. Optics

As delineated in Figure 3.1, the Optics component primarily functions to convert the back-scattered radiance into radiometric power. This radiance is computed after accounting for interactions with object materials in the scene, guided by their bidirectional reflectance distribution functions [45]. Essentially, this stage emulates the lens system of the receiver.

The parameter set in this context includes:

1. **Aperture Area:** The receiver's aperture area is the surface area of the opening that collects returning laser signals after they have bounced off objects. This parameter is crucial because a larger aperture area can collect more light, thereby enhancing the signal-to-noise ratio and improving the LiDAR system's overall sensitivity. However, since this value couldn't be determined exclusively from the user manual (A.1), and considering the unavailability of the LiDAR instrument for direct measurement, the value for the aperture area suggested by the Ansys Database, which is 100 000 mm^2, was used in this model.

3.1.1.4. Opto-Electronical

Figure 3.1 presents a depiction of the 'Opto-Electronical' module in the AVX system. This component is responsible for the conversion of the radiometric power, received by the 'Photodetector', into an electrical current. Additionally, it includes the transformation of this electrical current into a digital waveform via the 'Analogue to Digital Converter (ADC)'. The parameters established in this component include:

1. **Noise Standard Deviation:** The noise standard deviation parameter reflects the variation or dispersion in the electrical noise associated with the analog

opto-electronics. Measured in nanoamperes (nA), this parameter represents the statistical noise distribution within the analog signal processing components of the LiDAR system. The ability of the LiDAR system to detect weak signals is influenced by this noise factor. However, this information was not available in the datasheets and should ideally be obtained from the LiDAR supplier. Thus, for this model, the default value of 10 nA from the Ansys Database was utilized.

2. **Responsivity:** Responsivity characterizes the efficiency of a photodiode to transform the incident light, measured as radiometric flux, into an electrical current. Responsivity is typically quantified in amperes per watt ($\mathrm{A\,W^{-1}}$), reflecting the electric current generated per unit of light power incident on the photodiode. A higher responsivity signifies a more efficient conversion of light into an electrical signal. Consequently, for an equivalent amount of incident light, the electrical signal generated is stronger, potentially enhancing the sensitivity and performance of the LiDAR system. Unfortunately, this parameter is not provided in A.1. The photodiode's responsivity could potentially be estimated based on typical values corresponding to the type of photodiode and light wavelength used. However, the type of photodiode included is also not specified. Therefore, the default value of 1 $\mathrm{A\,W^{-1}}$ has been utilized for this model.

3. **Max Current:** The maximum or saturation current refers to the upper limit of the output current that the photodiode can generate. When the photodiode is functioning at its saturation current, any additional incident light will not result in an increased output current, which can distort the signal and decrease the accuracy of the measurements. To ensure accurate measurements and a linear response, the photodiode needs to operate below its saturation current. However, as the exact value is not provided in A.1, a high value of 100 000 mA is used in this model to prevent any potential saturation.

4. **Sampling Frequency:** The Sampling Frequency parameter is crucial as it denotes how frequently the analog signal is converted into digital format by the ADC.

This conversion process is integral to determining the accuracy of the LiDAR system, as per the equation $a = \frac{c}{f}$, where 'a' is the accuracy of the measurement, 'c' is the speed of light, and 'f' is the sampling frequency of the ADC. This relationship indicates that a higher sampling frequency would lead to a higher level of accuracy. Given the desired accuracy of 2 cm as per [22], a corresponding sampling frequency of 15 GHz was adopted.

5. **ADC Bits:** The ADC bit resolution is the number of bits used by the ADC to represent the amplitude of the signal. The precision of the signal's digital representation is directly influenced by the bit resolution. For instance, if the bit resolution is 1 bit, the ADC can only represent the signal amplitude as either high or low (2 levels). Each additional bit exponentially increases the number of distinguishable levels (2^n where n is the number of bits. While this information was not provided by the LiDAR manufacturer, the bit resolution was set to 32 for this model.

3.1.1.5. Default Processing

As depicted in Figure 2.5, the 'Default Processing' component primarily transforms digital waveforms into valuable output metrics, specifically Time of Flight and Intensity. This section discusses the essential parameters established within this component:

1. **Max Returns:** In LiDAR systems, 'returns' refer to the reflected light pulses detected by the sensor. A single light beam in a rotating LiDAR system can generate multiple returns if it encounters various surfaces within its path. The 'Max returns' parameter sets the maximum number of detectable returns per laser pulse, influencing the resolution and detail of point cloud data. Given that this value is not specified in A.1, the model defaults to 1, recording only the first detected reflection for each laser pulse.

2. **Max Range:** 'Max range' indicates the furthest distance where a LiDAR sensor can reliably detect and measure objects. This is the maximum operational distance

of the LiDAR system, and during simulations, only objects within this range contribute to the point cloud data. As stated in A.1 and [22], this parameter has been designated a value of 120 m for the present simulation.

3.1.2. LiDAR Simulation Parameters

The AVX simulation engine relies on a parameter file, formatted in JavaScript Object Notation (JSON), to configure the sensors employed within a simulation. This file dictates crucial information to the simulation engine, such as the types of sensors to be deployed, their specific configurations, and the preferred output format.

When it comes to LiDAR sensors, the parameters file empowers the user to determine the format of the output data and optimize the performance of the simulation engine. The parameters file used throughout this thesis can be found in A.2.

A key configuration within the simulation parameters lies in the 'grid' setting, responsible for controlling the subsampling for each laser beam. In a rotating LiDAR setting, this subsampling follows a polar pattern. Here, the 'radialGridPoints' and 'angularGridPoints' parameters define the number of sample points along the radial direction and around each radial circle, respectively. The 'hasCentralPoint' parameter, on the other hand, governs whether an extra ray is fired from the laser beam's center.

In this study, the grid configuration was set to a single sample point for both 'radialGridPoints' and 'angularGridPoints'. In essence, this configuration signifies that each laser beam is represented by a single sample. While increasing sample values could potentially enhance the precision of the simulation, it would also demand greater computational resources. Thus, in the interest of computational efficiency, a streamlined configuration was selected.

Furthermore, the 'contribution' output option was activated, generating ground truth information essential for subsequent training processes. This function allows identification of the objects detected by the LiDAR during the simulation, providing useful insight. A detailed exploration of this function can be found in section 3.3.2.

3.1.3. Sensor Layout and Configuration

This section discusses the sensor translation and orientation process using AVX. The positioning and orientation of the sensor take cues from the sensor setup in the KITTI data set, with the intention to mirror these conditions for consistency.

As displayed in Figure 3.3a, the sensor setup in the KITTI data set situates the laserscanner 1.73 m above ground level and 0.76 m longitudinally from the rear axis of the vehicle. This exact translation was applied to the modeled LiDAR, the updated location of which can be observed in Figure 3.3b. The sensor's orientation aligns with the AVX coordinate system, with the Z-axis represented in blue and the Y-axis in green.

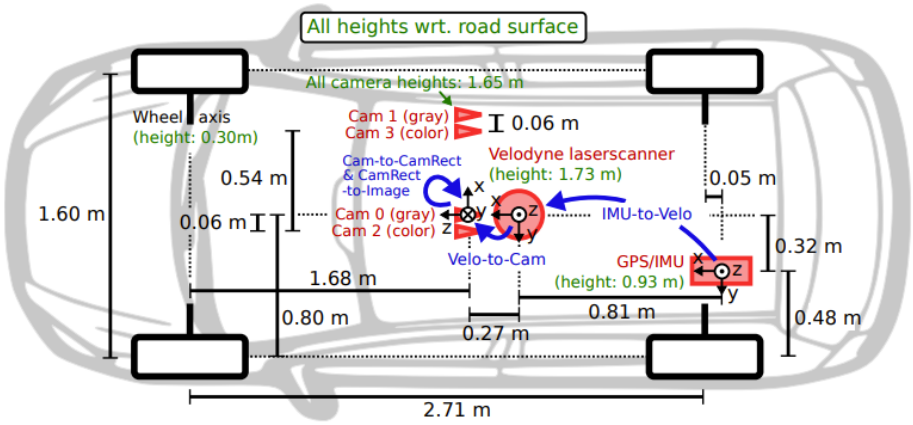
An important point of distinction is that the ego vehicle employed in the simulations differs from the VW Passat station wagon used in the KITTI database, depicted in Figure 2.4. As the asset file specifying the physical attributes and properties of the VW Passat was not accessible for AVX, an alternative vehicle, the 2010 Audi A1 in blue, was chosen to represent the ego vehicle.

However, the selection of the ego vehicle model or color has minimal bearing on the LiDAR sensor simulation. This is because the sensor's primary function is to record its surroundings, and any point cloud data from the ego vehicle is subsequently filtered out during the preprocessing phase prior to being fed into the neural network.

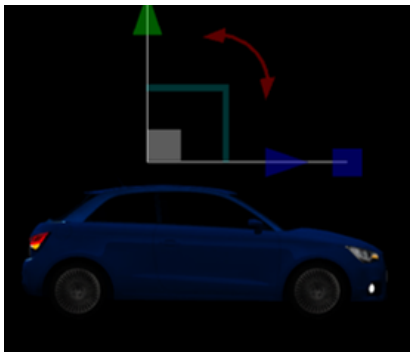
3.2. Ansys AVxcelerate CarMaker Co-simulation

AVX is a powerful tool that offers detailed physics-based simulations of LiDAR models. The software effectively mirrors the real-world functions of LiDAR elements, including the laser emitter's attributes and the receiver's optics and electronics. It also considers the environmental interactions, featuring the optical properties of materials across laser bandwidths from 905 nm to 1550 nm.

AVX is adaptable, supporting various LiDAR technologies, including rotating, solid-state, and flash types. It utilizes its robust, GPU-accelerated ray-tracing capabilities to provide accurate LiDAR sensor simulations.



(a) The sensor setup from the KITTI data set, showcasing sensor (in red) dimensions and mounting positions in relation to the vehicle body. Heights above ground (in green) are measured with reference to the road surface. Sensor transformations are highlighted in blue. Figure in courtesy of [22].



(b) The position and orientation of the LiDAR sensor model post translation and orientation adjustment. Figure in courtesy of [21].

Figure 3.3.: Sensor Positions in the simulation model and KITTI data set.

Co-simulation, the integration of multiple distinct simulation tools, becomes instrumental in this context. This section elaborates on the co-simulation framework involving AVX and CarMaker [24].

CarMaker [24], a product of IPG Automotive, is a software designed for virtual testing of vehicles, from basic vehicle model simulations to complex systems involving driver assistance and integrated powertrains.

These tools, when combined in a co-simulation framework, offer enhanced accuracy and comprehensive understanding. The co-simulation with AVX and CarMaker enables concurrent simulation of vehicle physics and sensor performance.

In this co-simulation setup, CarMaker handles the vehicle dynamics, control systems, and driving environment, while AVX manages the sensor simulations. This integrated approach establishes a dynamic simulation environment where CarMaker introduces real-time driving scenarios and AVX offers precise sensor feedback. This combination paves the way for sensor placement optimization, parameter fine-tuning, and enhanced vehicle performance in real-world conditions.

This co-simulation tool will be leveraged to construct various scenarios and generate the intended synthetic data set for this study.

3.2.1. Mapping the Modelled LiDAR Sensor to CarMaker for Co-simulation

The procedure of associating the modelled sensor, as described in section 3.1, with CarMaker necessitates numerous vital steps.

Primarily, this process leverages the simulation parameter file and the sensor configuration file. The simulation parameter file, discussed in section 3.1.2, provides key details about the sensor's characteristics, while the sensor configuration file, elaborated in section 3.1.3, specifies which sensors to attach to the ego vehicle.

The Co-simulation-Map file is another pivotal component of this mapping procedure. This file plays a crucial role in creating a correspondence between AVX assets and their counterparts within CarMaker. Essentially, the Co-simulation-Map file bridges Ansys' physics-based sensor library with CarMaker's driving tests, environments, and vehicles,

thus streamlining the co-simulation process.

3.2.2. Deployment of CarMaker's Object Sensor

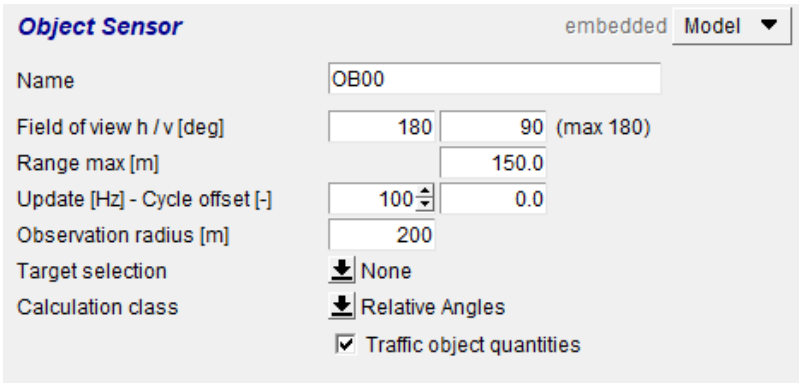
In this section, the use of CarMaker's Object Sensor to calculate the 3D bounding boxes of generated frames is discussed. The choice of the ego car, which was previously referenced in section 3.1.3, is reiterated and utilized in this phase.

The Object Sensor in CarMaker is designed to detect objects categorized as Traffic Objects in the TestRun. Its parameters are graphically depicted in Figure 3.4a. The sensor is configured with a horizontal span of 180°, a vertical span of 90°, and a longitudinal range set to 150 m. An observation radius of 200 m is established to avoid computation for distant objects, ensuring that only Traffic Objects within this area are considered.

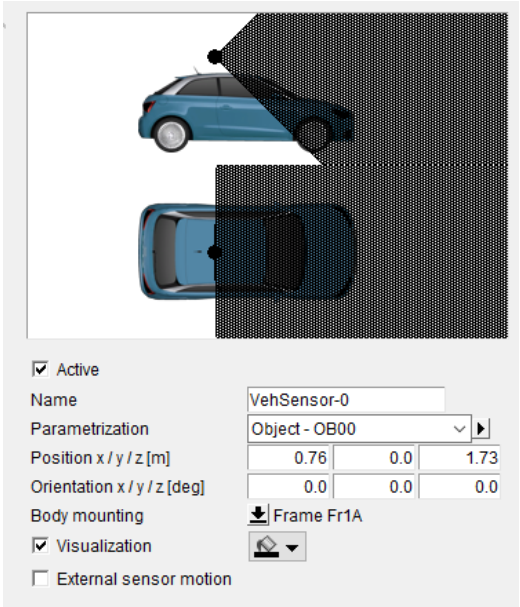
While the Object Sensor's specifications may appear overly comprehensive, potentially detecting objects beyond the range of the modeled Velodyne sensor, it is crucial to clarify the purpose of its application. The goal is to generate 3D bounding boxes specifically for objects that intersect with the Velodyne sensor's beams. Therefore, the broad detection capabilities do not introduce any problems or inaccuracies in our intended outcome. This Object Sensor functions in a hypothetical context exclusively for calculating the 3D bounding boxes. As such, stringent adherence to real-world sensor limitations is not a requirement in this scenario.

The 'Update' parameter sets the frequency at which the environment is scanned by the Object Sensor, configured to be ten times the rotation rate of the Velodyne sensor. This setting allows for ten separate object scans for every generated point cloud, crucial for accommodating dynamic scenes. Matching the 'Update' frequency to the Velodyne LiDAR's rotation rate (10 Hz) could result in discrepancies between the point clouds of objects and their corresponding 3D bounding boxes due to scene changes.

Figure 3.4b visually demonstrates the body mounting of the Object Sensor. It is critical to note that the position and orientation of this sensor match the LiDAR model referenced in section 3.1.3. This alignment ensures the generated 3D bounding boxes



(a) Parameters of the Object Sensor. Figure in courtesy of [24].



(b) Sensor Mounting describing the position, orientation and attachment of the Object Sensor. Figure in courtesy of [24].

Figure 3.4.: CarMaker's Object Sensor. Figures in courtesy of [24].

accurately correspond to the sensor's perspective.

3.2.3. Formulation of Driving Simulation Scenarios

This section provides a detailed explanation of the construction of driving simulation scenarios for synthetic data generation. A scenario, in this context, is defined as a particular succession of conditions or events within the simulation.

A crucial aspect of constructing these scenarios is the creation of road files. A road file is a digital representation of the physical path that the vehicle is expected to navigate during the simulation. The road file details attributes such as the road's length, width, curvature, banking, slope, and surface texture, all contributing to the overall realism of the simulation environment. It is important to note that these road files must also be incorporated into the AVX environment as track files for accurate co-simulation.

For the purpose of this research, pre-existing tracks from the Ansys's Co-simulation database are utilized. The CarMaker Scenario Editor offers functionality beyond road network construction, including the ability to define routes for both the test vehicle and traffic objects. In this instance, the focus will be on generating new routes, trajectories the vehicle is programmed to follow during the simulation.

Figure 3.5 illustrates a sample scenario featuring multiple vehicles, each with defined routes shown as thin red lines. The yellow line represents the route for the ego vehicle. By exploiting the ability to establish different routes on a single track file, a wide range of diverse scenarios can be produced.

3.2.4. Configuring Traffic Objects in CarMaker

Traffic Objects in CarMaker represent interactive entities within the simulation environment, such as vehicles, pedestrians, or cyclists. This section delineates the configuration and implementation of these Traffic Objects in the driving simulation.

Figure 3.6 exhibits the general structure of the traffic dialog, specific to the given scenario. In this instance, five different car objects, one cyclist, and two pedestrians are created and labeled.

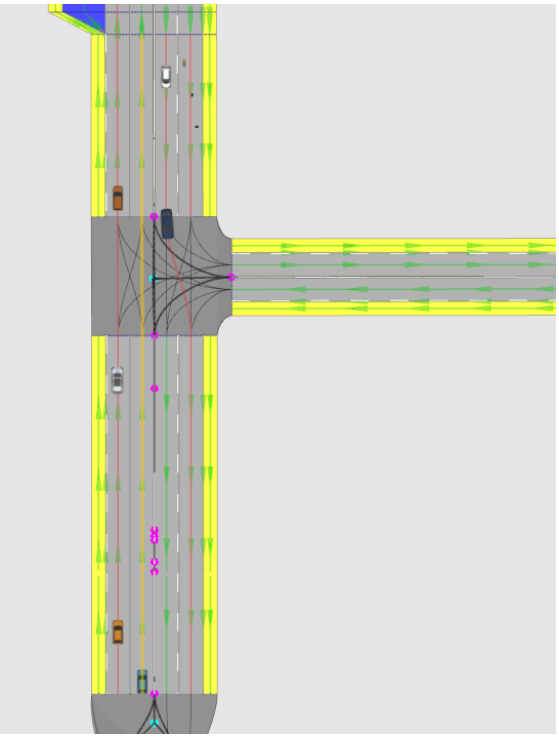


Figure 3.5.: Illustration of a typical scenario depicting the ego car (blue) and its route (yellow line), in conjunction with other vehicles and their routes (red lines). Figure in courtesy of [24].

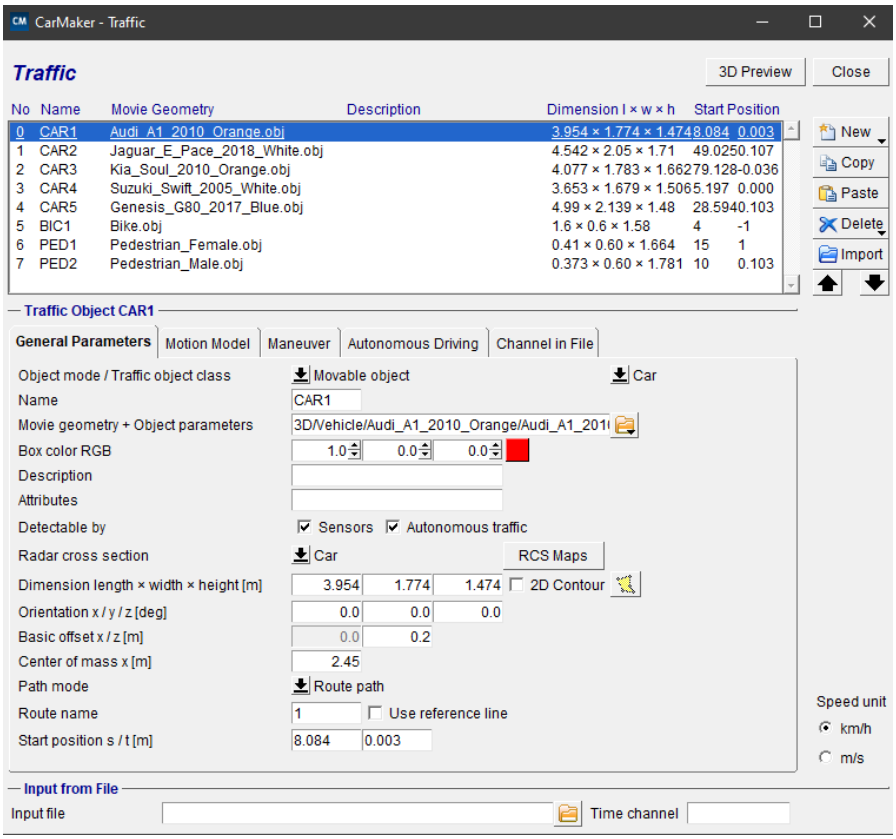


Figure 3.6.: Illustration of the general layout of the traffic dialog. Figure in courtesy of [24].

The configuration process commences with the selection of the 'Object Mode', which distinguishes between movable objects with predefined motion and stationary objects. Subsequently, the 'Object Class' is selected, focusing on 'Car', 'People', and 'Bicycle' for this study.

The traffic object's movie geometry and parameters are defined next. It is important to note that these same details must be included in the AVX environment, aligning the physical models of the involved objects. This requirement somewhat restricts the selection of traffic objects, as the AVX Co-simulation database is not as comprehensive as CarMaker's.

The dimensions of the traffic objects, encompassing length, width, and height, are precisely defined [See Figure 3.6]. The initial orientation of the traffic object relative to the road can be adjusted as well.

Next, the routes specified in section 3.2.3 are assigned to the traffic objects. These route paths will dictate the object's expected trajectory during the simulation. The starting position of the traffic object, in route coordinates, is also defined.

Lastly, a motion model is designated for each traffic object.

3.2.5. Configuration of Ego Vehicle's Maneuver

A maneuver represents a predetermined sequence of actions or events that the vehicle is designed to perform. These actions can include a variety of driving operations such as acceleration, deceleration, turning, or following a specific trajectory. Upon the allocation of a route to the ego vehicle, the following step consists of establishing a maneuver for the ego vehicle, thereby outlining its actions within the simulation.

3.2.6. Predetermination of OutputQuantities for Post-Simulation Analysis

In the approach towards the commencement of the simulation, it is of paramount importance to identify and select specific quantities that will be logged during the course of the Co-simulation. These chosen output quantities, aligned with the requirements of the subsequent analysis, will enable the computation of the 3D bounding boxes.

The key variables to be captured are as follows:

- The relative distances (x, y, and z coordinates) from the Object Sensor, as detailed in section 3.2.2, to the reference points of the traffic objects.
- The rotational orientation (Euler angles in the ZYX convention) of the traffic objects, as interpreted from the Object Sensor's frame at their respective reference points, as detailed in section 3.2.2.

Maintaining these measurements for each traffic object aids in the subsequent derivation of the 3D bounding boxes. Furthermore, the time stamps associated with each sensor signal update will also be recorded. The output of this process will be stored in a '.dat' file.

The sampling frequency for this process is determined to be 100 Hz, as discussed in section 3.2.2. This arrangement aims to produce ten unique bounding box candidates per traffic object in each point cloud.

3.3. Simulation Outputs

Upon setting the necessary parameters and initializing the simulation run, examining the subsequent outputs becomes crucial. This section delves into the specific data generated by the simulation, including the point cloud output, the contribution output, and the OutputQuantities from CarMaker [Refer to section 3.2.6].

3.3.1. LiDAR Point Cloud Output

The LiDAR point cloud output, a text file, consists of a total of 111,104 points, in accordance with the firing frequency set to 17 361 Hz as referenced in section 3.1.1.1. The LiDAR sensor executes a full rotation every 100 ms, generating a frame with each complete turn. This rotation rate was set to 10 Hz, also referenced in section 3.1.1.1. Consequently, with 1,736 firings executed within every 100 ms interval, and each firing comprising 64 beams, a total of 111,104 points are produced.

A small excerpt from a sample point cloud output is available in A.3. The initial line of the output confirms that the LiDAR is of the rotating type. The subsequent lines follow the format:

- float X_x float Y_x float Z_x float I_x Int R_x

Each line in this format signifies a point in the point cloud.

In this format, X_x, Y_x, and Z_x are the coordinates of the return for beam x, indicating the location of the point in a three-dimensional space. I_x represents the normalized amplitude of the current for beam x, a value that ranges from 0 to 1, where 1 corresponds to the maximum current. The variable R_x represents the return number for beam x.

However, as the maximum return was previously set to 1 in section 3.1.1.5, the value of R_x is consistently 1 for all points in the point cloud, rendering this parameter non-informative for the given data set.

3.3.2. LiDAR Contribution Output

An additional output from the simulation run, which was activated as per the steps described in section 3.1.2, is the contribution output. This output consists of a text file containing an equal number of points as the point cloud output, in this case, 111,104 points. A brief excerpt from a sample contribution output is included in A.4.

Each line in this output corresponds to a point in the point cloud, providing a list of contributors per beam. The structure for each row in the text file is as follows:

- Int ID_1 float C_1 ... Int ID_n float C_n

Here, a contributor is a pair made up of an Entity Identification Number (EntityID) ID_i and a contribution ratio C_i.

Recall that the 'Max Return' parameter was set to 1 in section 3.1.1.5, meaning that each LiDAR beam was allowed to interact with a single object. Consequently, each line in the output presents a single EntityID with its associated contribution ratio of 1. The contribution output, in essence, reveals the identities of entities that each LiDAR beam has interacted with throughout the simulation.

This dictionary offers the capacity to identify which EntityIDs are associated with which traffic object in the scene. By integrating this information with the contribution output, it becomes feasible to extract the object-specific point cloud —— essentially, the point cloud of a traffic object —— by filtering the points in the generated point cloud where the contributions stem from these specific EntityIDs.

3.3.3. OutputQuantities from CarMaker

Data from CarMaker's OutputQuantities provide useful information about the objects detected within the simulated environment. These objects are different types of traffic entities, as detailed in section 3.2.4. As previously mentioned in section 3.2.6, each data entry should record the relative position (given by Cartesian coordinates: x, y, z) and orientation (given by Euler angles: roll, pitch, and yaw) of these objects.

In the data, individual objects are marked by specific tags like BIC1, CAR1, CAR2, and others. Each object is represented by six quantities: $ds.x$, $ds.y$, $ds.z$, $r_{zyx}.x$, $r_{zyx}.y$, $r_{zyx}.z$. These quantities were detailed in section 3.2.6. The Time field represents the timestamp of the data measurements, providing a reading every 0.01 seconds. This way, the sensor readings describe the relative position and orientation of each object in relation to the sensor at the exact time.

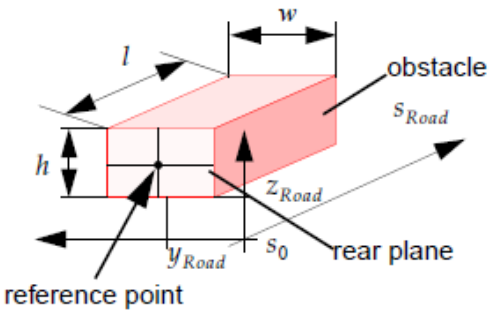


Figure 3.7.: Illustration of a traffic object's reference point within the simulation. Figure in courtesy of [24].

As illustrated in Figure 3.7, the reference point is situated on the rear plane of the traffic object, specifically in the middle of the height-width plane.

3.4. Processing

The last stage of the data generation pipeline is to handle the output point cloud data to generate the ground truth labels needed for neural network training.

Figure 3.8a shows a simulated scenario, created using the Ansys AVxcelerate CarMaker Co-Simulation explained in section 3.2. Figure 3.8b provides a visual representation of the point cloud output from this scene.

After obtaining our point clouds, the next goal is to generate labels using the outputs discussed in section 3.3. But first, the objective should be defined.

3.4.1. Processing objective

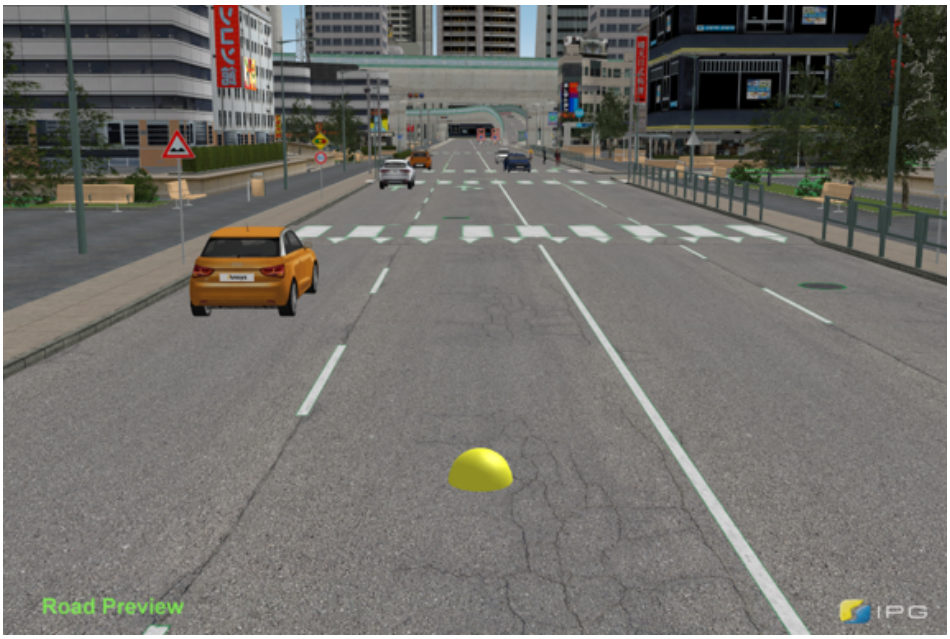
A main goal of this research is to get accurate ground truth 3D bounding boxes for detected traffic participants within a given scenario — namely, 'Car', 'Cyclist', and 'Pedestrian'.

The phrase 'ground truth' refers to the absolute truth that the machine learning model's predictions are compared against. These ground truth boxes are important for supervised learning, serving as training labels that guide the learning of a machine learning model. During its training phase, the model undergoes multiple updates with the goal to minimize the difference between its predictions and the ground truth.

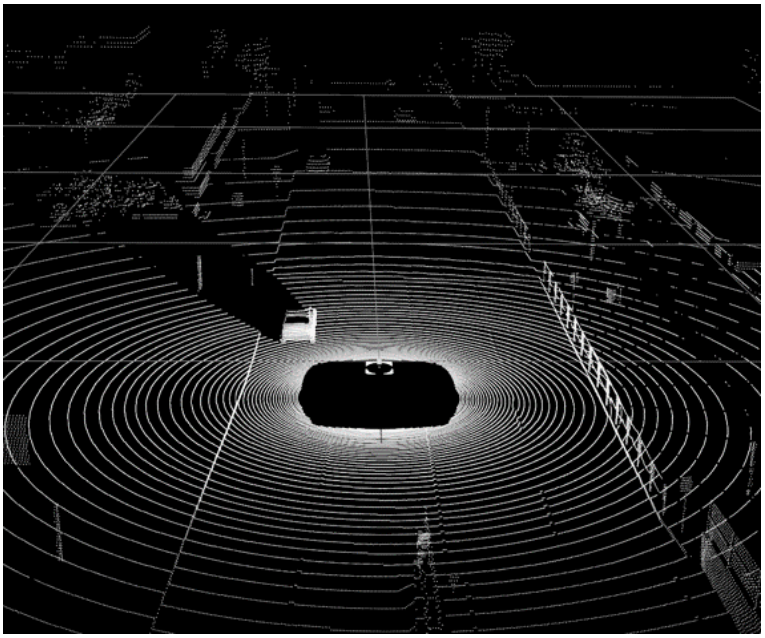
When the model's training is complete, it is important to test its performance on new data. In this step, the model's predictions are compared with ground truth boxes, and the calculated metrics are used to evaluate the model's performance.

As mentioned in [2], ground truth boxes are used to calculate the loss function during training. They, along with the anchors, are described by a seven-dimensional vector that includes the 3D bounding boxes' center point coordinates (x, y, z), dimensions (width - w, length - l, height - h), and rotation angle (yaw angle - θ).

The process of calculating localization regression residuals involves measuring the differences between the predicted (anchors) and the true (ground truth) bounding boxes for the related objects, as described in [2]. The total difference is used to calculate



(a) 3D Road Preview of a sample scenario. Figure in courtesy of [24].



(b) Generated point cloud visualization for the corresponding scene.

Figure 3.8.: Visual comparison of a simulated scenario and its corresponding point cloud output.

the localization loss, a part of the overall loss function used to train the network. This loss represents how far the model's predictions are from the ground truth, with the goal being to minimize this difference during the training process.

In the study by [22], 3D bounding box annotations are provided for each moving object within the camera's field of view, represented in LiDAR coordinates. The identified classes include 'Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', and 'Misc' (e.g., Trailers, Segways). Each object is assigned a class and its 3D size (height, width, length), and each frame offers the object's 3D translation and rotation, as displayed in Figure 3.9. It is worth noting that only the yaw angle is provided, with the other two angles (pitch and roll) approximated to be nearly zero [22].

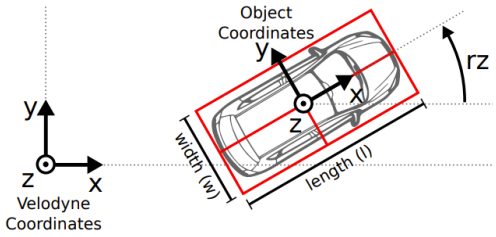


Figure 3.9.: Illustration of the coordinate system for annotated 3D bounding boxes in relation to the coordinate system of the 3D Velodyne laser scanner. Figure in courtesy of [22].

In conclusion, the main purpose here is to generate 3D bounding boxes for traffic objects within a frame. As a result, a ground truth 3D bounding box is defined by the following parameters:

- x: x center, y: y center, z: z center, w: width, l: length, h: height, θ: yaw angle

Subsequent to this, there is a requirement to transform these ground truth 3D bounding boxes into the KITTI label format. This subject will be addressed in section 3.4.2.2.

3.4.2. Challenges

Calculating 3D bounding boxes using the outputs mentioned in section 3.3 brings with it a set of challenges. These can be divided into two categories.

3.4.2.1. Deriving 3D Bounding Boxes from Point Cloud Data

The task of deriving 3D bounding boxes from point cloud data, even when the points corresponding to a particular object are known, is considerably difficult. This is due to several inherent traits of point cloud data and the nature of three-dimensional objects.

1. **Occlusions:** These occur when certain parts of an object are blocked or hidden by other entities within the scene, leading to an incomplete representation in the point cloud data.

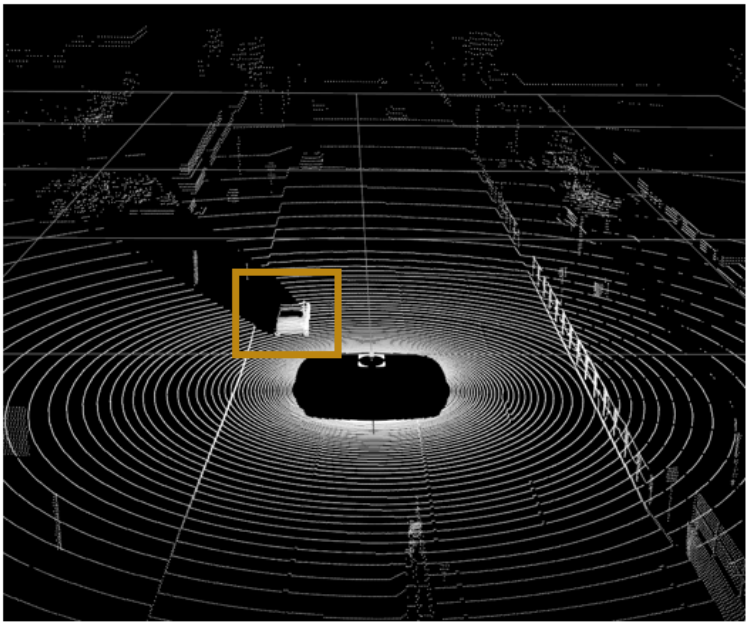
2. **Variations in Point Density:** Inconsistencies often exist in the density of points throughout the structure of an object in point cloud data. These irregularities cause unequal representations of the object, thereby making the exact computation of 3D bounding boxes more difficult. Areas with lower point density, which may contain crucial structural features, add to the challenge.

3. **Orientation Determination:** Figuring out the correct orientation of the bounding box can be challenging. This is particularly difficult for three-dimensional objects that lack a clear 'up' direction or where the primary axis is not clearly visible in the point cloud data.

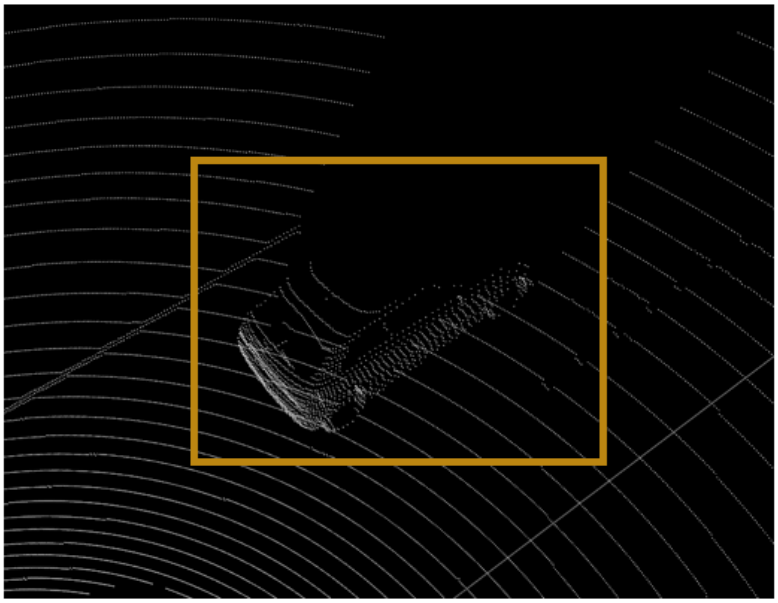
4. **Sparse Distribution of Point Clouds:** The point cloud data corresponding to each object are often sparsely distributed, adding to the difficulty in accurately determining bounding boxes.

To better understand these challenges, consider the following example:

Figure 3.10 and Figure 3.11 demonstrate two distinct car objects within the same scene. For the example given in Figure 3.10, the computation of the bounding box

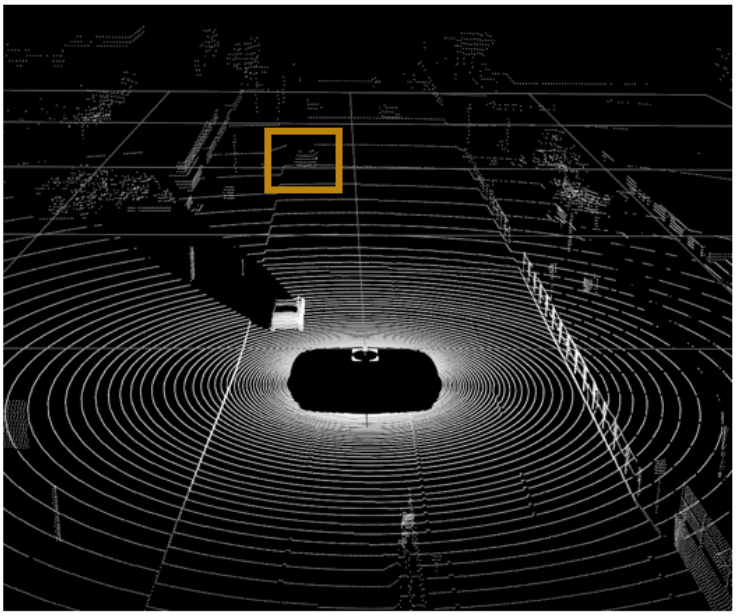


(a) A sample synthetic point cloud; the yellow square indicates the traffic object of interest.



(b) Detailed view of the traffic object of interest in Figure 3.10a.

Figure 3.10.: A point cloud sample illustrating a traffic object in a scene.



(a) A sample synthetic point cloud; the yellow square indicates the traffic object of interest.

(b) Detailed view of the traffic object of interest in Figure 3.11a.

Figure 3.11.: A point cloud sample illustrating a traffic object in a scene.

appears achievable as the object is unoccluded and the point density is relatively high. By visually inspecting the object, it can be identified as a car, and a rectangular bounding box could theoretically be fitted to determine its orientation and dimensions.

Contrarily, consider the case presented in Figure 3.11. Assuming that the object's point cloud is successfully extracted from the scene, determining the bounding box would be extremely difficult with the limited points available. To mitigate this issue, the Object Sensor in CarMaker, as detailed in section 3.2.2, is used. This sensor is capable of detecting objects and recording the relative position and orientation of these objects through Output quantities (discussed in sections 3.2.6 and 3.3.3), thereby aiding in overcoming these challenges.

3.4.2.2. Field of View Calculation and Label Conversion

This research encountered several challenges related to the use and manipulation of the KITTI data set.

Firstly, in the KITTI data set, objects are labeled within the field of view of a reference camera. This created a problem in the simulation setup, where no camera was present. The absence of a camera made the computation of the field of view impossible.

Secondly, the conversion of 3D bounding boxes to the KITTI label format was necessary. This was needed to make use of existing resources made for KITTI, like neural network models trained on KITTI data. Also, using the KITTI format would make it easier to compare results with other KITTI-based research. Using this common format would also help in combining real and synthetic data sets, reducing possible data formatting issues that could affect test results.

To address these issues, the calibration parameters used for conversions from LiDAR coordinates to camera coordinates and onto the camera image plane were explored. Upon observing minimal variations in these parameters, a constant calibration matrix was assumed. This assumption is regarded as valid since the relative pose and orientation of the sensors remain unchanged throughout the data generation process. In this particular setup, the Velodyne LiDAR sensor and the hypothetical camera maintain

their relative positions during the entire simulation.

A set of constant calibration parameters [See Equations 3.1,3.2,3.3,3.4] was subsequently implemented for each synthetic frame, borrowing from the parameters of a real frame from the KITTI data set. These parameters include:

1. $\mathbf{P}_{\text{rect}}^{(2)}$: The projection matrix post-rectification for camera 2 (the left color camera). This matrix transforms 3D points in the rectified camera coordinate system to 2D points in the image plane. It is a matrix of size 3×4.

2. $\mathbf{R}_{\text{rect}}^{(0)}$: The rectifying rotation matrix. This matrix aligns the 3D points in the camera coordinate system with the axes of the rectified coordinate system. It is a matrix of size 3×3.

3. $\mathbf{T}_{\text{velo}}^{\text{cam}}$: The transformation matrix, which converts 3D points from the LiDAR (Velodyne) coordinate system to the camera coordinate system. The first 3x3 segment is the rotation matrix, and the last column represents the translation vector. It is a matrix of size 3×4.

4. **image size**: The dimensions of the camera-generated image in pixels. This 1×2 matrix has the height of the image (375 pixels) as the first element and the width of the image (1242 pixels) as the second element.

Employing this methodology allowed for the calculation and filtering of the field of view prior to creating the 3D bounding boxes and converting these boxes into the KITTI label format. This approach ensured the preservation of the KITTI data set's integrity and structure while supplementing synthetic data.

The calibration parameters utilized in this study are as follows:

$$P_{\text{rect}}^{(2)} = \begin{bmatrix} 7.21537720e+02 & 0.00000000e+00 & 6.09559326e+02 & 4.48572807e+01 \\ 0.00000000e+00 & 7.21537720e+02 & 1.72854004e+02 & 2.16379106e-01 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 & 2.74588400e-03 \end{bmatrix} \tag{3.1}$$

$$R_{\text{rect}}^{(0)} = \begin{bmatrix} 0.9999239 & 0.00983776 & -0.00744505 \\ -0.0098698 & 0.9999421 & -0.00427846 \\ 0.00740253 & 0.00435161 & 0.9999631 \end{bmatrix} \tag{3.2}$$

$$T_{\text{velo}}^{\text{cam}} = \begin{bmatrix} 7.53374491e-03 & -9.99971390e-01 & -6.16602018e-04 & -4.06976603e-03 \\ 1.48024904e-02 & 7.28073297e-04 & -9.99890208e-01 & -7.63161778e-02 \\ 9.99862075e-01 & 7.52379000e-03 & 1.48075502e-02 & -2.71780610e-01 \end{bmatrix} \tag{3.3}$$

$$\text{image size} = \begin{bmatrix} 375 & 1242 \end{bmatrix} \tag{3.4}$$

The calibration parameters are foundational to the process of projecting 3D points from the Velodyne's LiDAR coordinate system onto the 2D image plane of the camera. A detailed walkthrough of the transformation process is provided below.

Taking into account a 3D point in the LiDAR coordinate system in homogeneous coordinates denoted as $p_{lidar} = [X, Y, Z, 1]^T$, the conversion process includes three essential steps:

1. **Transform from Velodyne to Camera Coordinates:** Use the Velodyne-to-camera ($T_{\text{velo}}^{\text{cam}}$) calibration matrix to transform from the LiDAR to the camera coordinate system:

$$p_{cam} = T_{\text{velo}}^{\text{cam}} \times p_{lidar} \tag{3.5}$$

2. **Rectify the Camera Coordinates:** Apply the rectification rotation matrix $R_{\text{rect}}^{(0)}$ to align the coordinates:

$$p_{rect} = R_{\text{rect}}^{(0)} \times p_{cam} \tag{3.6}$$

3. **Project to 2D:** Employ the rectified projection matrix $P_{\text{rect}}^{(2)}$ to convert the 3D point from the rectified camera coordinate system to the 2D image plane of the camera:

$$p_{2D} = P_{\text{rect}}^{(2)} \times p_{rect} \tag{3.7}$$

The resulting P_{2D} will be in homogeneous coordinates. To translate this to Cartesian coordinates, normalize the x and y values by the third coordinate:

$$x = \frac{p_{2D}[0]}{p_{2D}[2]} \tag{3.8}$$

$$y = \frac{p_{2D}[1]}{p_{2D}[2]} \tag{3.9}$$

The final x and y coordinates align with the 2D point on the image plane originating from the 3D point in the LiDAR coordinate system.These calibration parameters establish the foundation for enabling the conversion from LiDAR to camera coordinates, subsequently facilitating projection onto the image plane. This allows for the filtering of the field of view prior to the creation of 3D bounding boxes.

As highlighted in section 3.4.1, the objective is to convert the 3D bounding boxes into the KITTI label format [22], a task that is facilitated by the defined calibration parameters. However, to comprehend the potential benefits of this conversion, a brief overview of the KITTI labels is warranted.

The labels for KITTI's object detection data set are stored within text files, with each line representing an individual object present within the scene. Each point cloud frame corresponds to a specific text file. The fields of each line are as follows:

- **type:** Object class ('Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', 'Misc', or 'DontCare').
- **truncated:** Float (0-1), extent of the object outside image boundaries (0 = fully visible, 1 = invisible).
- **occluded:** Integer (0, 1, 2, or 3), degree of object occlusion (0 = fully visible, 1 = partly occluded, 2 = largely occluded, 3 = unknown).
- **alpha:** Observation angle of object, ranging from $-\pi$ to π.
- **bbox:** Four values defining 2D bounding box in image (x and y of top-left corner, x and y of bottom-right corner).

- **dimensions:** Three values (height, width, length), object dimensions in camera coordinates (meters).
- **location:** Three values (x,y,z), 3D location of object in camera coordinates (meters).
- **rotation_y:** Object rotation around y-axis in camera coordinates, ranging from $-\pi$ to π.

While the computation of all label fields except for 'occluded' was accomplished, the computation of this field is left as a valuable target for future work.

3.4.3. Processing Steps

This section demonstrates the processing of the outputs generated, as discussed in section 3.3.

3.4.3.1. Deletion of Identical Point Clouds

It was noted that in some instances, temporally successive point cloud outputs were identical. This presents a potential challenge, considering the dynamic nature of the scenario being explored, where consecutive point cloud outputs should inherently present variations.

Upon investigation, the duplication of point cloud files appeared to be an outcome of a bug in the AVX software. This issue has been reported to the software developers for resolution. Nevertheless, to maintain data set integrity, it was vital to remove these duplications.

To automate this process, a Python script was employed. This script identifies instances of identical point clouds and subsequently deletes them from the data set, as detailed in A.8.1.

3.4.3.2. Data Transformation

The objective is to map the original AVX environment's coordinate system to that of the KITTI Velodyne LiDAR's coordinate system.

As illustrated in Figure 3.3b, the AVX environment employs a unique coordinate system. In this framework, the 'Z' axis signifies the front, as represented by the blue arrow in Figure 3.3b, the 'X' axis denotes the left direction, and the 'Y' axis corresponds to the upward direction, indicated by the green color in Figure 3.3b.

On the other hand, the KITTI Velodyne LiDAR operates on a different coordinate system, depicted in Figure 3.12.

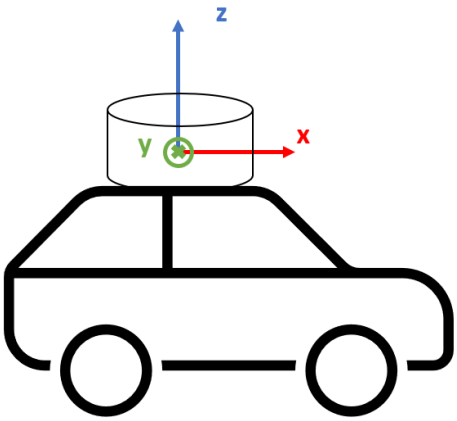


Figure 3.12.: Illustration of the Velodyne laser scanner's coordinate system.

This transformation is automated using a Python script, as referenced in A.8.2. The transformed data, now in the KITTI Velodyne LiDAR's coordinate system, is saved as Numpy array files.

3.4.3.3. Conversion of JSON Files to Text

A straightforward preprocessing step involves transforming the LiDAR contribution dictionary, initially in the JSON format, into a more readable structure. The aim of this transformation is to facilitate easier data interpretation by converting the information into a text file format. The conversion process is enabled through a Python script, found in A.8.3.

3.4.3.4. Extraction of Traffic Object Dimensions

The analysis involves handling unique instances in the contribution dictionary. Each 'instance' corresponds to traffic object names, shown in Figure 3.6. For every unique instance, its associated dimensions are recorded in a text file, shown in Table 3.1. These dimensions come from a manually created list that includes dimensions for different models, covering all dimensions for the AVX traffic object assets. The process of compiling this list involves utilizing the dialog box depicted in Figure 3.6, with the resulting list accessible at A.8.4.

The Python script A.8.4 assists in the execution of this task. A sample output, a text file for a single frame, is presented in the table below:

Table 3.1.: Dimensions of Traffic Objects for a sample frame.

Object	Length	Width	Height
CAR1	3.954	1.774	1.474
CAR2	4.542	2.050	1.710
CAR3	4.077	1.783	1.662
CAR4	3.653	1.679	1.506
CAR5	4.990	2.139	1.480
BIC1	1.600	0.600	1.580
PED1	0.410	0.600	1.664
PED2	0.373	0.600	1.781

3.4.3.5. Mapping Instance Names to Entity Identifiers

This stage involved establishing a connection between instance names (from Table 3.1) and their respective EntityIDs from the contribution dictionary. The result is a file documenting each instance name, in other words traffic objects, coupled with a list of its corresponding EntityID. A sample of the resulting data can be viewed in A.6.

The Python script A.8.5 was developed to execute this task.

3.4.3.6. Label Generation

Proper label generation is vital for LiDAR point cloud data. The algorithm detailed in A.8.6 accomplishes this task, processing LiDAR point cloud data directories and generating a matching set of labels, compliant with the KITTI label format. Each label is subsequently stored in a specific file corresponding to a particular point cloud.

The implementation, though intricate, warrants some discussion of its key features:

The algorithm initially filters out blank lines in the contribution file and subsequently removes the associated points in the point cloud. This step essentially eliminates beams that did not interact with any elements in the scene, resulting in their absence from the point cloud.

The point cloud is then transformed from LiDAR coordinates to rectified camera coordinates, using the parameters and equations outlined in section 3.4.2.2. These rectified points are projected onto an image plane utilizing the camera parameters discussed in section 3.4.2.2. A subsequent check ensures these projected points remain within the image boundaries and that the points' depth remains non-negative.

Then the algorithm applies a specific range for point cloud filtration, defined as:

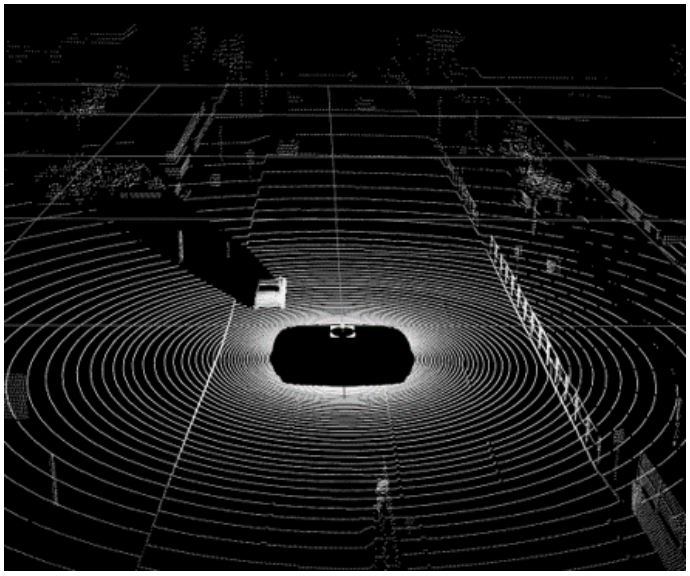
- `Point Cloud Range [m] = [`x_{min} `= 0,` y_{min} `= -39.68,` z_{min} `= -3,` x_{max} `= 69.12,` y_{max} `= 39.68,` z_{max} `= 1]`

This point cloud range is consistent with the parameters used for training and evaluating the network, as suggested in [2].

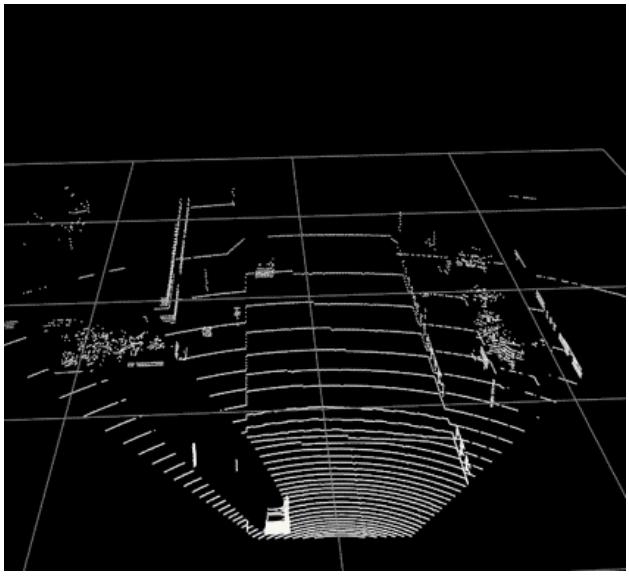
The impact of field of view filtering on point clouds is demonstrated in Figure 3.13. Figure 3.13a displays a representative point cloud in its initial state, while Figure 3.13b shows the same point cloud after field of view filtering application, visually highlighting the significant effect of this filtering method.

The next process begins by extracting object point clouds for each traffic object using the EntityIDs discussed in section 3.4.3.5 and exemplified in A.6.

Assume an object of interest, 'CAR1'; its dimensions can be found in Table 3.1. This object's relative position and orientation, as captured by CarMaker's Object Sensor, are



(a) Illustration of a sample point cloud.



(b) Illustration of a sample point cloud after the application of field of view filtering.

Figure 3.13.: Comparative Illustrations demonstrating the influence of field of view filtering on a sample point cloud.

described in section 3.3.3. Note the presence of ten separate object scans, as detailed in section 3.2.2, indicating a reading frequency of every 0.01 seconds.

Each object is characterized by six distinct parameters: $ds.x$, $ds.y$, $ds.z$, $r_{zyx}.x$, $r_{zyx}.y$, and $r_{zyx}.z$ (as explained in section 3.3.3). However, $r_{zyx}.x$ and $r_{zyx}.y$ are disregarded in line with KITTI labels, which only consider the yaw angle [22], thus represented by $r_{zyx}.z$. The bounding box's center coordinates are determined by shifting $ds.x$ and $ds.y$ along the object's orientation direction (i.e., $r_{zyx}.z$) by half the object's length. This shift is needed due to the reference point of the traffic object being positioned on the rear-plane (refer to Figure 3.7).

With these parameters, 3D bounding boxes can now be defined by their center point, extent, and yaw angle, as detailed in section 3.4.1. However, with the existence of ten bounding box candidates, an iterative process is employed. For each candidate center and yaw angle, an oriented bounding box is constructed at the center with the given extents and yaw angle. Then, the number of points from the object point cloud falling within each oriented bounding box candidate is computed. The oriented bounding box enclosing the maximum number of points is selected.

Figure 3.14 highlights the importance of considering multiple candidate oriented bounding boxes during the selection process. This figure visually demonstrates the 3D bounding box candidates corresponding to a traffic object —— in this case, a car. This examination ensures the most encompassing and representative 3D bounding box is selected for each traffic object.

To improve the representation of object dimensions, particularly for the 'Pedestrian' and 'Cyclist' categories, it is essential to refine the initial oriented bounding box, which is derived from CarMaker's provided positions.

Given the inherent structure and physical attributes of 'Pedestrian' and 'Cyclist', the corresponding point clouds can appear in irregular and scattered patterns, potentially resulting in a mismatch between the initial bounding box and the actual object point cloud. Therefore, it is necessary to verify the dimensional compatibility of the bounding box and the object point cloud along each axis.

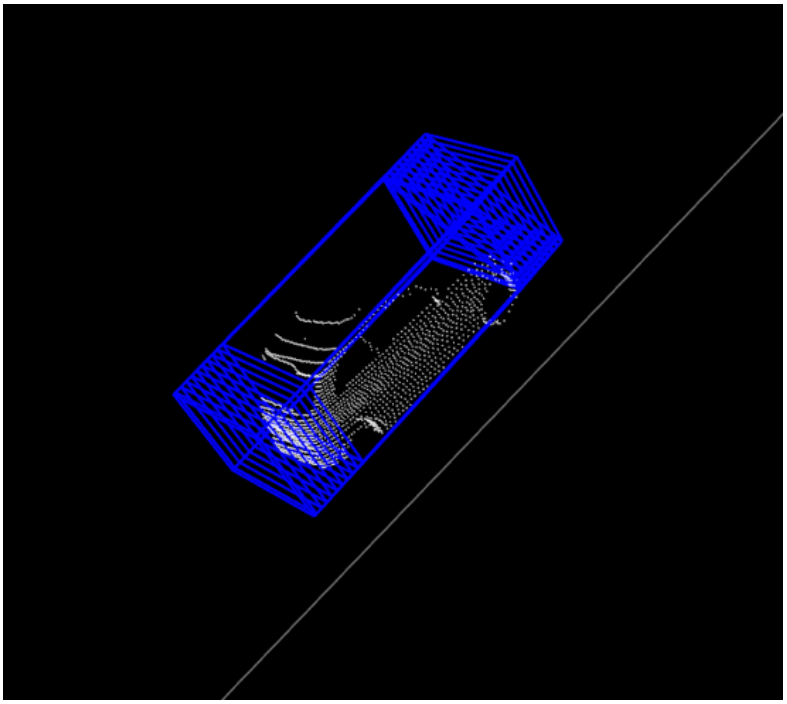
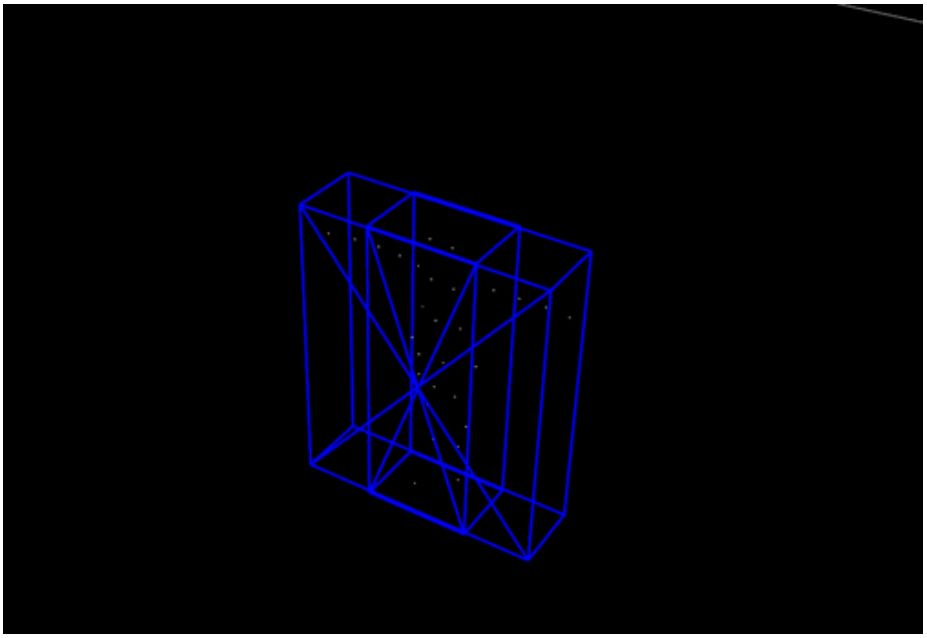


Figure 3.14.: Depiction of 3D bounding box candidates associated with a car.

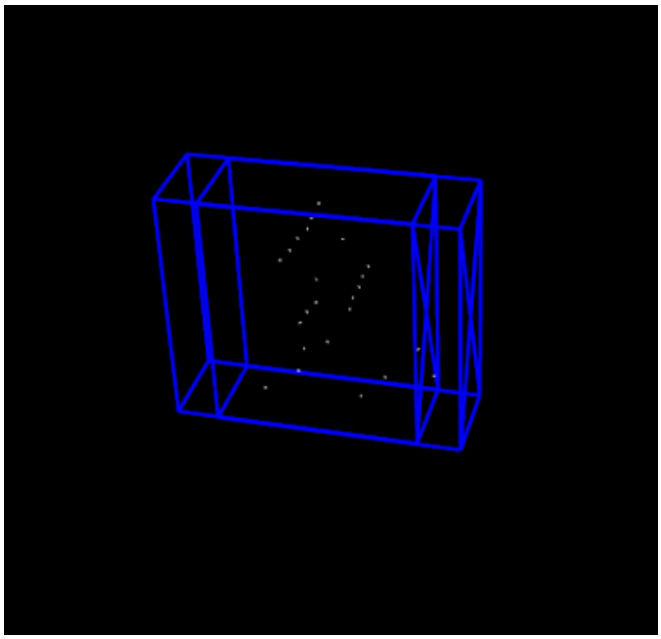
If the bounding box fails to encapsulate the object point cloud on any axis, the box must be expanded to match the object point cloud's dimension on that axis. This expansion requires a recomputation of the box's center.

However, adjustments might still be necessary, even if an expansion is not required. Such cases typically occur when the point cloud lacks symmetry or when the distribution of points is uneven. Under these circumstances, the xy-plane center of the bounding box may not align with the point cloud's center. This necessitates realigning the bounding box's center in the xy-plane with the point cloud's center, while maintaining the same z-coordinate.

Figure 3.15 illustrates the critical nature of these adjustments and expansions. Figure 3.15a shows a T-shaped pedestrian spreading their arms wide. The inner bounding box, derived from CarMaker, does not cover all points of the object, hence requiring expansion. Figure 3.15b displays a cyclist. Here, the left bounding box (initial one) suggests an asymmetric object distribution, which is rectified in the right box (modified



(a) The inner box shows the bounding box before modification, the outer box shows the after expansion.



(b) The left box shows the bounding box before modification, the right box shows after adjustment.

Figure 3.15.: Comparative illustrations of initial and modified bounding boxes.

one). These modifications ensure that the bounding box accurately encapsulates the spatial extent of the object in the LiDAR point cloud, thus enhancing object detection performance.

As detailed in section 3.4.2.2, the 'truncated' parameter is an essential part of the KITTI labels. This floating point value, which ranges from 0 (non-truncated) to 1 (truncated), indicates the object's visibility extent, with truncation correlating to the object's deviation from image boundaries.

This concept is visually exemplified in Figures 3.16a and 3.16b. Figure 3.16a portrays a traffic object, 'Car', represented in a raw point cloud, while Figure 3.16b displays the same object post the application of field of view filtering.

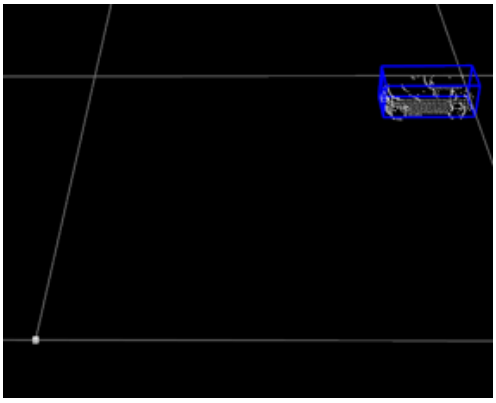
The necessity to compute the truncation value becomes evident upon noticing a discrepancy in the number of points enclosed within the bounding boxes in Figures 3.16a and 3.16b. When such a need arises, a uniformly distributed hypothetical point cloud is generated within the bounding box, as shown in Figure 3.16c. Following this, the hypothetical point cloud undergoes filtering using the identical calibration parameters and methodology detailed in section 3.4.2.2. The outcome, represented in Figure 3.16d, is the filtered version of the hypothetical point cloud.

The truncation value is subsequently derived as one minus the ratio of the number of points in the filtered hypothetical point cloud to the number in the original hypothetical point cloud. Consequently, for the example depicted in Figure 3.16, the computed truncation value is 0.9.

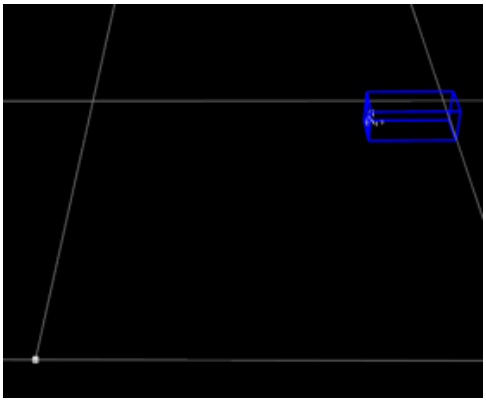
Throughout this study, observations suggest that modifying the z-coordinate (height) of the established bounding boxes can lead to a better match with the genuine 3D object shape present in the point cloud data.

For cars, the modification involved decreasing the z-coordinate by a value of 0.08 m. This downward shift aimed to correct for an inherent offset present in the synthetic data generated by CarMaker, which resulted in the initial bounding boxes being slightly elevated above the road surface.

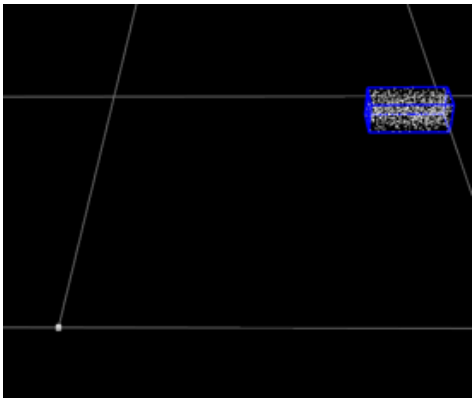
Conversely, for 'Pedestrian' and 'Cyclist' objects, the bounding boxes were adjusted



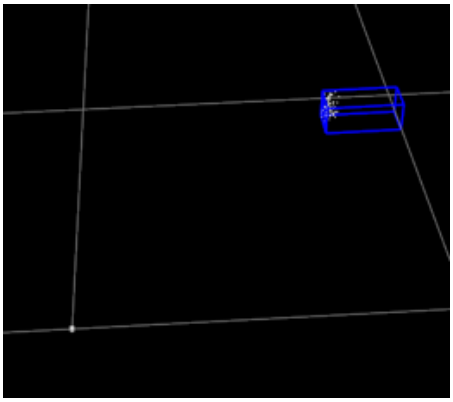
(a) Raw object point cloud and its corresponding bounding box.



(b) Filtered version of Figure 3.16a.



(c) Uniformly distributed hypothetical point cloud generated within the bounding box.



(d) Filtered version of Figure 3.16c.

Figure 3.16.: Sequential stages in the calculation of truncation value.

upward by increasing the z-coordinate by 0.02 m. This modification was intended to exclude road surface points from the bounding boxes, thereby improving the accuracy of object representation.

These adjustments, while based on observed patterns, cater specifically to the unique characteristics of the synthetic data set employed in this study. The adjustment values were chosen through a process of repeated visual inspections of the data, and their application may not be universally suitable for other data sets or simulation scenarios.

In the concluding stage of this process, the oriented bounding box —— characterized by its center, extent, and yaw angle —— along with the truncation value, is converted into a label string following the KITTI label format, as referenced in section 3.4.2.2. This conversion was enabled through the use of the calibration matrix and the associated transformations discussed in the same section.

Nonetheless, a lingering challenge remains: calculating the 'occluded' field in the KITTI label. Due to current project limitations, this aspect has not been addressed, resulting in all objects being labeled as 'fully visible'. This opens up a potential avenue for future refinement to further boost the representation in this synthetic data set.

The label generation procedure concludes with the creation of labels for a synthetic point cloud. The resulting labels, formatted in KITTI style, are displayed in Table 3.2. In this context, 'Tr.' refers to 'truncated', 'Occ.' to 'occluded', and 'Rot. Y' signifies 'rotation_y'. These terms were previously explained in section 3.4.2.2.

Table 3.2 presents the labels for two cars detected in the synthetic point cloud visualized in Figure 3.8b.

Visualizations of labeled bounding boxes on real and synthetic data, in addition to raw and filtered data, can provide a more accessible understanding for readers.

Figures 3.17 and 3.18 demonstrate these visualizations. Figure 3.17 displays the labeled bounding boxes, represented in blue, on a raw real point cloud from the KITTI data set, in conjunction with those on a raw synthetic point cloud. In a similar manner, Figure 3.18 exhibits the labeled bounding boxes on filtered versions of these point clouds, specifically focusing on the camera's field of view.

Table 3.2.: Generated Labels for the Synthetic Point Cloud (Refer to Figure 3.8b), Presented in KITTI Format.

Type	Tr.	Occ.	Alpha	Bounding Box				Dimensions			Location			Rot. Y
				left	top	right	bottom	height	width	length	x	y	z	
Car	0.00	0	-1.14	63.86	185.86	397.10	361.11	1.47	1.77	3.95	-3.99	1.66	8.34	-1.57
Car	0.00	0	-1.49	532.63	178.93	567.90	205.60	1.71	2.05	4.54	-4.08	2.15	49.57	-1.57

Upon examination of these figures, certain differences between synthetic and real point clouds become apparent. Synthetic point clouds, as generated by AVX, possess lower noise levels, owing to their derivation from idealized models. Moreover, they appear more complete, a consequence of being produced from comprehensive 3D scene models.

Conversely, real point clouds are inherently imperfect and tend to include gaps or missing data, a result of occlusions in the real world environment. This distinction is particularly noticeable in Figures 3.17 and 3.18.

3.5. Scaling Simulated Scenarios

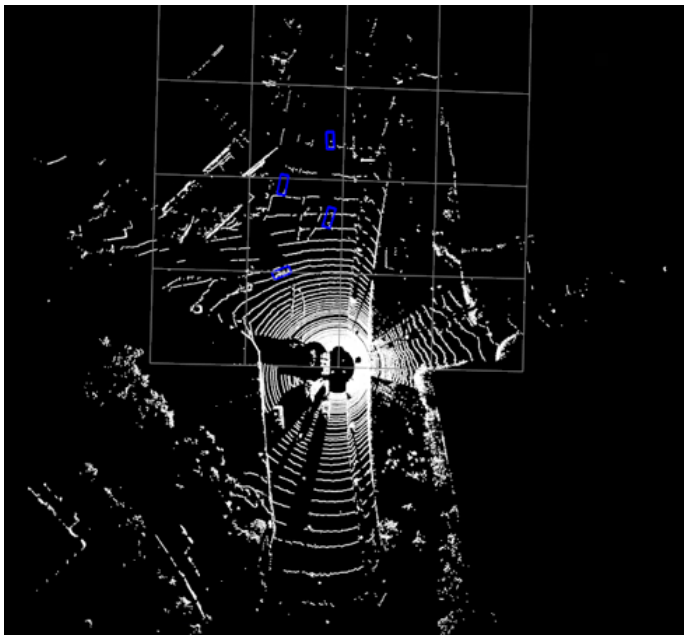
The enlargement of the simulated scenarios was facilitated through the development of several Python scripts, designed to streamline the process. The Python script detailed in A.8.7 undertakes the task of reorganization and renaming. It is worth mentioning that labels without corresponding objects within the camera's field of view, known as empty labels, and their respective point clouds are disregarded during this process.

The script in A.8.8 automates the execution of a sequence of scripts. Upon scenario selection, this script ensures all components are prepared for use.

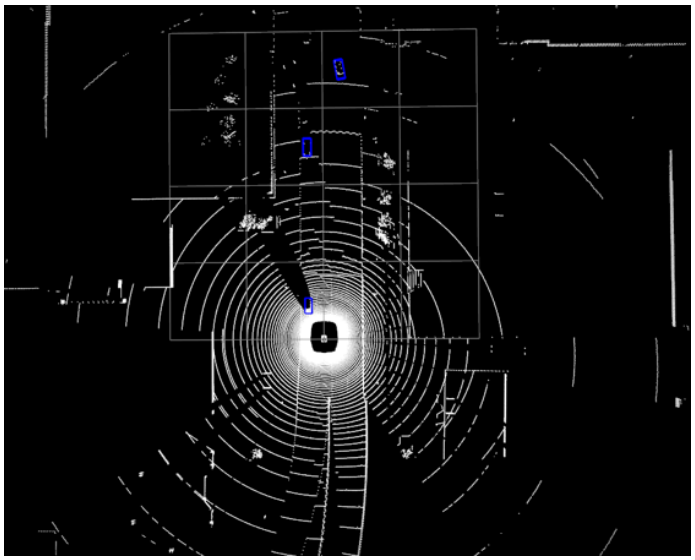
The function of the script defined in A.8.9 is to collect point cloud data and labels from a multitude of source folders, transferring them to a unified destination folder. This transfer grants the data unique identifiers and sets the stage for the establishment of the 'Training' and 'Evaluation' databases.

Lastly, it is crucial to note that the script detailed in A.8.10, sourced from [58], is employed extensively in this study for the purpose of visualizing point clouds and bounding boxes.

The CarMaker Co-simulation Library deployed for this research provided 46 unique car assets, each representing distinct car models. However, it is important to note that these car assets comprise 15 distinct size variations, as some merely represent different color variations of the same model, as detailed in A.8.4.

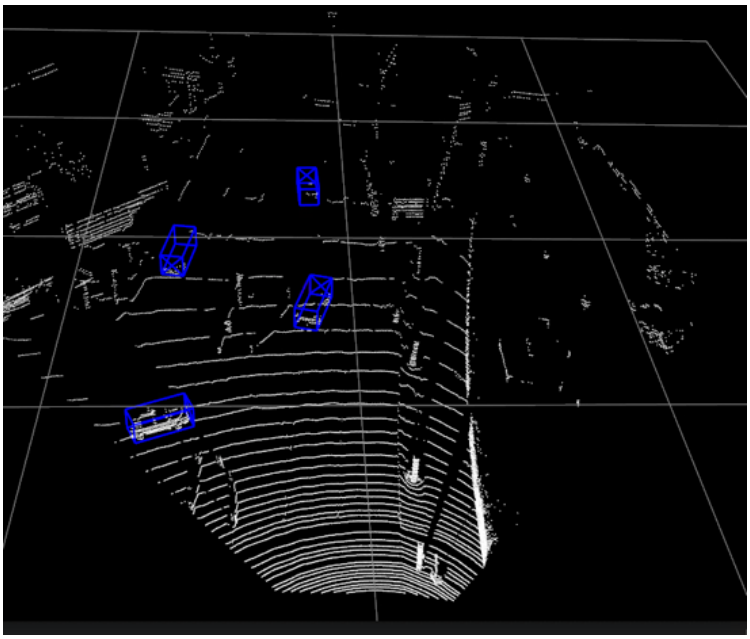


(a) Raw real point cloud from the KITTI data set, with labeled bounding boxes indicated in blue.

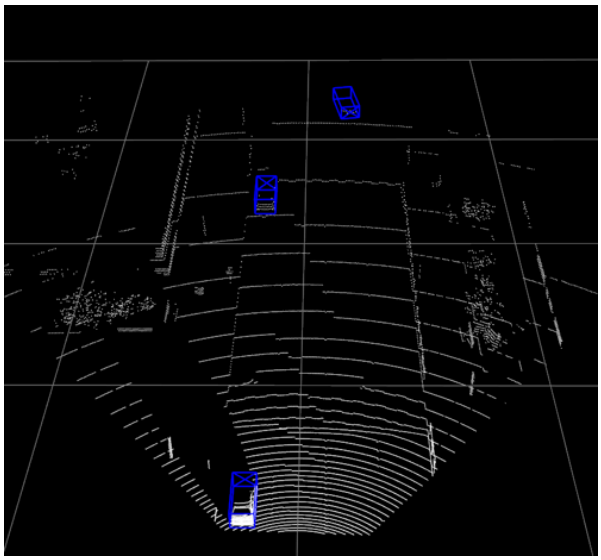


(b) Raw synthetic point cloud, with labeled bounding boxes indicated in blue.

Figure 3.17.: Visualization of labeled bounding boxes in raw real and synthetic point clouds.



(a) Filtered real point cloud from the KITTI data set, with labeled bounding boxes indicated in blue.



(b) Filtered synthetic point cloud, with labeled bounding boxes indicated in blue.

Figure 3.18.: Visualization of labeled bounding boxes in filtered real and synthetic point clouds, focusing on the camera's field of view.

The library incorporates two pedestrian assets (male and female) and one cyclist asset. These assets can be configured as either dynamic or stationary entities.

Figure 3.19 presents a simulation scenario that incorporates all three asset types (Car, Pedestrian, and Cyclist).

Figure 3.19.: Illustration of the diverse assets available in the CarMaker Co-simulation Library.

In this research, a scenario designates a particular sequence of events or driving situation being simulated. Environments, conversely, refer to the virtual contexts within which these vehicle simulations occur, supplying the necessary physical conditions for the simulation.

For validation data collection, 31 unique scenarios across 3 unique environments were employed, yielding a total of 4,464 generated frames. However, only 3,617 of these frames contained objects within the field of view and were hence deemed suitable for further analysis.

For training, 66 unique scenarios across 8 unique environments were utilized, resulting in the generation of 9,872 frames, of which 5,981 frames captured objects within the field of view successfully.

4. Experimental Design

4.1. Data sets

Throughout this thesis, terms such as 'Evaluation', 'Testing', 'Test', 'Validation' will be employed synonymously. Similarly, 'AVX data set' and 'synthetic data set' are used interchangeably, as are 'KITTI data set' and 'real data set'.

The comparison of the KITTI and AVX data sets, in terms of frame count and object classes, is presented in Table 4.1. For both training and testing subsets, the distribution of object classes across frames is enumerated. It is noteworthy that the AVX data set encompasses only three object classes — car, pedestrian, and cyclist — compared to the wider variety of classes in the KITTI data set. This comparative table provides insight into the structure and diversity of the employed data sets.

Table 4.1.: Number of Frames and Object Class Counts in KITTI and AVX data sets.

data set	number of frames	Car	Pedestrian	Cyclist	Tram	Truck	Van	Misc	Person sitting	DontCare
KITTI Train	3712	14357	2207	734	224	488	1297	337	56	5399
KITTI Test	3769	14385	2280	893	287	606	1617	636	166	5896
AVX Train	5981	6880	3519	3140	-	-	-	-	-	-
AVX Test	3617	6192	1475	1058	-	-	-	-	-	-

In the KITTI data set, the 'DontCare' label plays a crucial role in identifying regions that are not relevant for evaluation. Such regions may comprise portions of the image that contain either irrelevant objects or objects too small or ambiguous to be classified distinctly. By specifically tagging these 'DontCare' regions, the algorithm can efficiently concentrate on areas of relevance, reducing false positives and improving detection

performance. Any objects detected within these 'DontCare' regions during the test phase are intentionally disregarded to avoid their misinterpretation as false positives.

In the KITTI data set, a consideration is made for visual resemblances that may exist among specific classes. For instance, detections of 'Vans' are not classified as false positives when the target class is 'Car'. Similarly, the 'Person sitting' instances are not treated as false positives in the case of 'Pedestrian' detections. This consideration allows the system to correctly handle instances where class appearances might be ambiguous or misleading.

The synthetic AVX data set does not incorporate such features, primarily due to the scenarios created that include only three distinct classes: 'Car', 'Pedestrian', and 'Cyclist'. Although the 'DontCare' labels would be beneficial for enhancing object detection precision, they are not implemented within the scope of this study. However, their potential addition offers a promising avenue for future work.

It is essential to clarify that the 'KITTI Train' and 'AVX Train' data sets, as specified in Table 4.1, are solely utilized for training purposes. Similarly, the 'KITTI Test' and 'AVX Test' datasets are reserved solely for processes such as testing, validation, and evaluation. Regardless of the terminology used —– testing, validation, or evaluation —– all refer to the act of assessing model performance.

For instance, when the phrases 'trained on KITTI' or 'train on real-world data' are used, they signify the usage of the 'KITTI Train' data set for training. Similarly, when 'trained on AVX' or 'train on synthetic data' are stated, it is implied that the 'AVX Train' data set is used for training. Further, terms such as 'tested on AVX', 'tested on synthetic data', 'validated on synthetic data', 'evaluation on AVX', or 'evaluation on synthetic data set' denote the use of 'AVX Test' for these assessment processes. Analogously, similar expressions with reference to KITTI correspond to the use of the 'KITTI Test' data set.

4.2. Network Settings

The PointPillars Network, due to its distinctive and beneficial design, was selected as the main architecture for investigation in this study. Its design, characterized by an exclusive deployment of 2D convolutional layers, presents an efficient approach to managing LiDAR point cloud data. This efficient, high-performing, and streamlined architecture deemed the PointPillars Network an appropriate choice for conducting experiments on both real and synthetic data. More details on the network can be found in section 2.5.

The computational experiments conducted in this thesis were implemented using high-performance computing infrastructure. The hardware utilized consisted of a server-grade Intel(R) Xeon(R) Gold 6238R CPU running at 2.20 GHz with 32 cores, coupled with a powerful RTX8000P-8Q virtual GPU equipped with a total memory of 8192MB. This setup ensured computational power and memory capacity to handle the intensive tasks of training and evaluating.

Regarding software, the experiments were executed using Python, specifically version 3.10.6, compiled with GCC 11.3.0. The training and testing of the models employed PyTorch [59], a widely used open-source machine learning library for Python, at version 1.13.1+cu117. NVIDIA's CUDA toolkit was installed to support GPU-accelerated operations in PyTorch, with the version being 11.7.

OpenPCDet [58] was chosen as the primary tool for the implementation of the PointPillars network [2] in this study. OpenPCDet [58] is a robust, open-source PyTorch-based codebase specifically crafted for 3D object detection in point cloud data. It is highly respected in the field for its flexibility and efficiency, offering extensive support for multiple state-of-the-art 3D object detection methods.

For this study, OpenPCDet [58] was utilized with its default training configurations for the PointPillars [2]. This approach ensured that the experiments were conducted in a controlled, reproducible manner and that the results obtained align with established standards in the field.

The experiments center on the detection of three object classes, specifically 'Car', 'Pedestrian', and 'Cyclist', leveraging the PointPillars model. This architecture is consistently employed across all experiments, with its configuration thoroughly detailed in the provided script (refer to A.7).

The data preprocessing stage comprises several steps, including masking points and boxes outside a pre-specified range, shuffling points during the training phase to enhance model generalization, and transforming points into voxels.

The point cloud range is specified in meters as follows:

- $x_{\text{min}} \quad y_{\text{min}} \quad z_{\text{min}} \quad x_{\text{max}} \quad y_{\text{max}} \quad z_{\text{max}}: \quad [0, -39.68, -3, 69.12, 39.68, 1]$

The computation of canvas size involves determining the range difference in each dimension ($x_{\text{max}} - x_{\text{min}}$ for the x-axis and $y_{\text{max}} - y_{\text{min}}$ for the y-axis), followed by division by the grid resolution of 0.16 m, thereby resulting in a canvas size of 432×496.

Furthermore, the training process restricts the maximum number of points per voxel to 100 and the total number of voxels to 12,000. This restriction generates a dense tensor of dimensions $(D = 9, P = 12000, N = 100)$, where D represents the dimensionality of the augmented lidar points. A detailed explanation of these parameters can be found in section 2.5.1.1.

Every point undergoes a linear layer operation, followed by BatchNorm and ReLU, resulting in a tensor of dimensions $(C = 64, P = 12000, N = 100)$. Here, C represents the number of output features of the encoder network. The max operation across pillars yields an output tensor of dimensions $(C = 64, P = 12000)$. After redistributing P pillars back to their original locations, a pseudo-image of dimensions $(C = 64, H = 432, W = 496)$ is created, providing a 3D tensor of size $64 \times 432 \times 496$ as input to the backbone. Figure 2.7 is a visual depiction of this process.

The backbone, which serves as the network's feature extraction component, includes three blocks: Block1($S = 2, L = 4, C = 64$), Block2($2S = 4, L = 6, 2C = 128$), and Block3($4S = 8, L = 6, 4C = 256$). Details on these blocks can be found in section 2.5.1.2. The backbone sequentially processes data, incrementally elevating the quantity of 2D

convolution filters to learn complex features as the spatial dimension reduces.

This is followed by an upsampling process consisting of: Up1($S_{in} = 2, S_{out} = 2, 2C = 128$), Up2($S_{in} = 4, S_{out} = 2, 2C = 128$) and Up3($S_{in} = 8, S_{out} = 2, 2C = 128$). Additional details on the upsampling process can be found in section 2.5.1.2. These deconvolutional blocks serve to upsample the features back to their original resolution.

The features of Up1, Up2, and Up3 are then concatenated to provide $6C = 384$ features for the detection head, as depicted in Figure 2.7.

Following this, the detection segment utilizes a dense layer, which utilizes high-level features derived from the backbone. The anchor-based detection head is designed to form unique sets of anchors for each class.

The training phase spans 80 epochs, utilizing the Adam optimizer with a learning rate of 0.003 and maintaining a batch size of 2. The loss function used is a weighted sum of classification, localization, and direction classification losses [2].

Post-processing of the model's output incorporates NMS [2]. Instead of pre-training, the weights undergo random initialization via a uniform distribution. To maintain reproducibility, a fixed random seed is utilized, ensuring consistent results across different runs.

Lastly, data augmentation enriches the training set's diversity [2]. This process introduces random but realistic transformations, such as rotations and flips, into the data set. The augmentation pipeline also includes ground truth sampling, random world flipping along the x-axis, random rotation within a predefined range, and random scaling.

In order to conduct the training process, a modified version of the training script derived from the OpenPCDet [58] framework was utilized. Key configurations within the script were specifically tailored to meet the demands of each experimental setup.

4.3. Evaluation Metrics

The selection of KITTI Evaluation Metrics [60] in this research for network performance assessment is guided by key considerations:

The primary goal of this research is to generate KITTI-like point clouds by replicating the renowned KITTI data set. The similarity of the produced data to the KITTI set makes its evaluation metrics relevant and suited to this study.

Furthermore, the extensive usage of the KITTI benchmark suite in autonomous driving research highlights its utility for comparative performance analyses.

This suite [60] includes metrics such as Average Precision for 2D object detection (AP_{2D}), Average Precision for 3D object detection (AP_{3D}), Average Precision for BEV object detection (AP_{bev}) and Average Orientation Similarity (AOS).

The understanding and interpretation of the Average Precision (AP) metric form the cornerstone of assessing the performance of object detection tasks. This metric is primarily discussed in relation to the detection of 'Car', 'Pedestrian', and 'Cyclist' classes.

Each detected object is predicted with a bounding box parameterized by the center coordinates (x, y, z), width w, length l, height h, and yaw angle θ. A corresponding confidence score also quantifies the certainty level of the detection.

The following part focuses primarily on AP_{3D}, which classifies detections as either true positives or false positives based on their 3D Intersection over Union (IoU) with the ground truth bounding boxes. For each detection d and the corresponding ground truth box g, the 3D IoU is computed as follows:

$$IoU_{3d}(d_{3d}, g_{3d}) = \frac{Volume(d_{3d} \cap g_{3d})}{Volume(d_{3d} \cup g_{3d})} \tag{4.1}$$

In this equation, d_{3d} and g_{3d} are the 3D representations of the predicted and ground truth bounding boxes, respectively.

To simplify the notation going forward, the terms d_{3d}, g_{3d}, and IoU_{3d} will be denoted as d, g, and IoU, respectively.

A detection d is classified as a true positive if its IoU with a ground truth box g surpasses a certain threshold τ. It's essential to note that this threshold is subject to the type of object: τ is designated as 0.7 for cars and 0.5 for pedestrians and cyclists. This can be mathematically represented as:

$$TP(d,g) = \begin{cases} 1, & \text{if } IoU(d,g) > \tau \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

For each confidence threshold t, corresponding precision and recall are calculated as:

$$Precision(t) = \frac{\sum_{i=1}^{N} TP(d_i, g_i) \cdot \mathbb{1}(conf(d_i) > t)}{\sum_{i=1}^{N} \mathbb{1}(conf(d_i) > t)} \tag{4.3}$$

$$Recall(t) = \frac{\sum_{i=1}^{N} TP(d_i, g_i) \cdot \mathbb{1}(conf(d_i) > t)}{N} \tag{4.4}$$

Here, $conf(d)$ signifies the confidence of detection d, N represents the total number of ground truth boxes, and $\mathbb{1}(.)$ functions as an indicator function that returns 1 if the condition within the parentheses is met, and 0 otherwise.

These calculated pairs create a Precision-Recall (PR) curve when plotted. This PR curve provides a useful visual aid to comprehend the balance between model precision and recall, as depicted in Figure 4.1.

Figure 4.1 illustrates a PR curve for three classes: 'Car' (red), 'Pedestrian' (green), and 'Cyclist' (blue). Notably, different applications prioritize different areas on this curve. For instance, autonomous driving prioritizes high recall, especially for pedestrian detection, to minimize accidents. This scenario corresponds to a low confidence threshold, which generates a significant number of detections, thereby ensuring nearly all positive samples are detected. Another scenario could be face recognition on a mobile phone, where high precision is prioritized to avoid false positives.

A singular metric, the Average Precision (AP), is used to summarize the PR curve. AP is defined as the mean precision at a subset S of N equally spaced recall levels $[q_0, q_0 + \frac{q_1 - q_0}{N-1}, q_0 + \frac{2(q_1 - q_0)}{N-1}, \ldots, q_1]$, beginning at recall level q_0 and ending at level q_1:

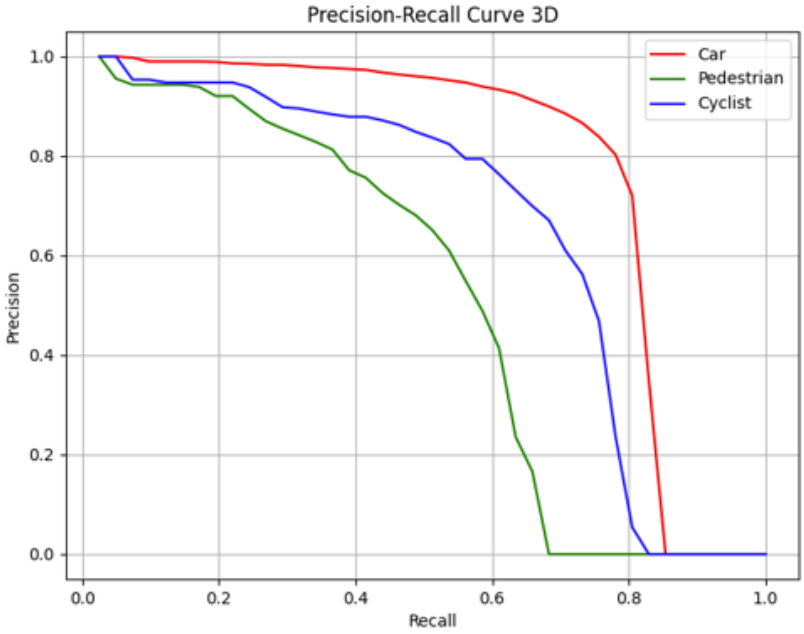


Figure 4.1.: Precision-Recall (PR) Curve for 3D Object Detection.

$$AP = \frac{1}{N} \sum_{r \in S} P_{\text{interpolate}}(r) \tag{4.5}$$

Here, to maintain the monotonicity of the PR curve, the precision at each recall level r is interpolated by the maximum precision at recall values greater or equal to r, defined as:

$$P_{\text{interpolate}}(r) = \max_{r' : r' \geq r} p(r') \tag{4.6}$$

The initial version of the KITTI benchmark [60] utilized the established 11-Point interpolated AP metric, assessing a subset of 11 recall levels at $S_{11} = [0, 0.1, 0.2, \ldots, 1]$. However, a more recent 40-Point interpolated AP metric [61] has been introduced and adopted by the KITTI benchmark as of October 8, 2019. This updated approach encompasses more information for a better approximation of the PR curve and omits the precision calculation at the 0 recall position to enable a fairer evaluation of detection

algorithms. The revised recall levels now stand at $S_{40} = [1/40, 2/40, 3/40, \ldots, 1]$. This thesis adheres to this methodology and calculates APs using the 40-Point interpolation.

The metrics AP_{2D}, AP_{3D}, and AP_{bev} all operate under similar principles, but the computation of the IoU varies according to the respective scenario. For the AP_{2D} metric, the bounding box prediction d_{2d} and the corresponding ground truth g_{2d} are represented in the 2D image plane. Conversely, for the AP_{bev} metric, the detection and ground truth boxes are represented in a top-down bird's-eye view, represented as d_{bev} and g_{bev} respectively. The IoUs for these scenarios are calculated as:

$$IoU_{2d}(d_{2d}, g_{2d}) = \frac{Area(d_{2d} \cap g_{2d})}{Area(d_{2d} \cup g_{2d})} \tag{4.7}$$

$$IoU_{bev}(d_{bev}, g_{bev}) = \frac{Area(d_{bev} \cap g_{bev})}{Area(d_{bev} \cup g_{bev})} \tag{4.8}$$

In these equations, d_{2d}, g_{2d}, d_{bev}, and g_{bev} signify the 2D and BEV representations of the predicted and ground truth bounding boxes, respectively.

In contrast to the previously mentioned metrics, Average Orientation Similarity (AOS) takes into account the difference between the head and tail of objects. The computation for this metric is as follows:

$$AOS = \frac{1}{N} \sum_{r \in S_{40}} \max_{\tilde{r}:\tilde{r} \geq r} s(\tilde{r}) \tag{4.9}$$

In this equation, r signifies the recall, computed using the 2D representations of both detections and ground truths. The notation $s(\tilde{r})$ represents the orientation similarity at a recall rate of $\tilde{r}$.

The orientation similarity, $s(\tilde{r})$, serves as a normalized version of the cosine similarity and can be formulated as:

$$s(r) = \frac{1}{|D(r)|} \sum_{i \in D(r)} \left(1 + \cos \Delta_\theta^i\right) \frac{\delta_i}{2} \tag{4.10}$$

Here, $D(r)$ denotes all object detections at a recall rate r. The term Δ_θ^i signifies the angular discrepancy between the ground truth and predicted orientations for the i^{th}

detection.

Moreover, to account for multiple detections corresponding to a single object, a penalty term, represented as δ_i, is introduced. This term is defined as follows:

$$\delta_i = \begin{cases} 1 & \text{if detection } i \text{ has been assigned to a ground truth bounding box} \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

This penalty term is 1 if a match has been found between detection i and a ground truth bounding box, and 0 in the absence of a match.

4.4. Adapting KITTI Difficulty Levels for Synthetic data set Evaluation

The KITTI data set segments object detection challenges into three separate levels of difficulty, titled 'Easy,' 'Moderate,' and 'Hard' [22]. The assignment of these levels is based on a set of conditions, which include minimum bounding box height, maximum occlusion, and truncation level.

- **Easy:** This level refers to simpler scenarios. The lower limit for bounding box height is 40 pixels, occlusion is denoted as 'Fully Visible' or 'Partly Occluded,' and truncation is limited to a maximum of 15%.
- **Moderate:** This level incorporates more complexity with smaller objects and higher occlusion levels. The minimum bounding box height in this case is 25 pixels. The occlusion can be 'Fully Visible,' 'Partly Occluded,' or 'Largely Occluded,' and the truncation can be up to 30%.
- **Hard:** This level presents the most challenging scenarios, potentially including small, heavily occluded or truncated objects. The bounding box height limit remains at 25 pixels, any occlusion level is allowed, and truncation can be as high as 50%.

However, when evaluating a synthetic data set, certain adjustments to this approach are necessary. As stated in section 3.4.3.6, the 'occluded' field in the KITTI label was not considered in the synthetic data set, resulting in all objects being marked as 'fully visible'. This modification means that object occlusions are not taken into account in the synthetic data set evaluation. Therefore, the revised difficulty levels for the synthetic data set only consider the bounding box height and truncation of the objects.

Also, as mentioned in section 4.1, the 'DontCare' label plays a significant role in the evaluation of real data sets. However, this feature is not included during synthetic data set evaluation.

4.5. Experiments

The experiments designed for this study were grounded on two fundamental principles. First, the weights of the network were initialized randomly to guarantee that no prior knowledge of the data was incorporated. Subsequently, the network structure and parameters are kept consistent across all training and testing stages.

The data sets used in these experiments include both real and synthetic data sets. Further information about these data sets can be found in section 4.1.

For the preparation of training sets, various versions of a preliminary Python script are utilized [See A.8.11].

4.5.1. Experiment 1: Training on a Single Database

The initial experiment aims to assess the effectiveness of the selected neural network architecture in executing detection tasks. Here, the network is trained exclusively on one data set. Subsequently, the performance of these networks is evaluated on both the KITTI and AVX test sets, using the KITTI metrics as the evaluation standard.

The underlying objective is to understand how robustly the models perform when exposed to diverse data distributions. Real-world data includes a wide array of variability and randomness. In contrast, synthetic data, while containing precise,

well-defined features, might lack some of the nuances and unpredictability seen in real-world scenarios.

- **Sub-experiment 1:** KITTI
  - The neural network is trained using real data, followed by validation on both real and synthetic data sets.
  - This procedure evaluates the capacity of a model, trained on real-world data, to successfully extrapolate its learned knowledge to synthetic data, and vice versa.
- **Sub-experiment 2:** AVX
  - The neural network is trained solely on synthetic data, followed by validation on both real and synthetic data sets.
  - The aim of this approach is to measure the capability of a model, trained on synthetic data, to perform effectively when exposed to real-world data, and vice versa.

4.5.2. Experiment 2: Training on Combined Databases

The second experiment of this study is formulated to introduce additional variability by merging data from the KITTI and AVX data sets. The objective is to evaluate the capability of combined data sets to augment model performance. This combination leverages the distinctive strengths of both types of data, with real data offering real-world complexity and variability, and synthetic data providing precise, controllable, and plentiful scenarios.

The merged training data set incorporates 3712 frames, aligning with the size of the KITTI training set, as outlined in section 4.1. This experiment includes several sub-experiments, each utilizing a unique proportion of AVX and KITTI data sets. The overarching aim is to identify an optimal mix of real and synthetic data that boosts model performance across both real and synthetic validation sets.

In each sub-experiment, the total frame count is kept constant at 3712, with the proportion of AVX to KITTI data being varied. For example, in the initial sub-experiment, 90% of the 3712 frames are sourced from the AVX data set, and the remaining 10% from the KITTI data set. In the selection process, the first 371 samples are chosen from the KITTI training set, and the residual frames are sourced from the AVX training set, sorted based on the number of objects each frame contains. This selection strategy prioritizes frames with a higher object count, given that some frames in the AVX data set only contain a few objects.

- **Sub-experiment 1:** 90% AVX, 10% KITTI
  - The first sub-experiment involves training the network on a data set composed of 90% AVX data and 10% KITTI data.
  - Performance evaluation is undertaken on both KITTI and AVX test sets.
- **Sub-experiment 2:** 80% AVX, 20% KITTI
  - The second sub-experiment involves training the network on a data set composed of 80% AVX data and 20% KITTI data.
  - Performance evaluation is undertaken on both KITTI and AVX test sets.
- **Sub-experiment 3:** 50% AVX, 50% KITTI
  - The final sub-experiment utilizes an equal blend of data from both sources, with each contributing 50% of the data set.
  - Performance evaluation is undertaken on both KITTI and AVX test sets.

4.5.3. Experiment 3: Pre-training on Synthetic Data and Fine-Tuning on Real-World Data

The third experiment adopts a fine-tuning approach, wherein the neural network is first pre-trained on synthetic data, followed by subsequent fine-tuning using real-world data. The main aim of this technique is to explore the potential benefits of fine-tuning methods in enhancing neural network performance.

The process begins with the network's pre-training on the AVX data set. Note that the term 'pre-trained model' refers to the network model trained on the AVX data set, as described in section 4.5.1. Subsequently, fine-tuning is carried out using different portions of the KITTI data set, namely 10%, 20%, and 50% of the total train set.

To assess the impact of synthetic data pre-training, the performance of each fine-tuned model is compared with that of models trained solely on the corresponding subsets of the KITTI data set without pre-training. This comparative analysis aims to shed light on the potential benefits of pre-training on synthetic data for real-world object detection tasks.

- **Sub-experiment 1:** Fine-Tuning with 10% of KITTI:
  - In this scenario, the network, initially pre-trained on AVX, is fine-tuned with 10% of the KITTI train set.
  - Performance evaluation is undertaken on both KITTI and AVX test sets.
  - The performance of this fine-tuned network is compared with a network trained solely on a data set comprising 10% of the KITTI train set, using the KITTI test set for evaluation.
- **Sub-experiment 2:** Fine-Tuning with 20% of KITTI:
  - In this scenario, the network, initially pre-trained on AVX, is fine-tuned with 20% of the KITTI train set.
  - Performance evaluation is undertaken on both KITTI and AVX test sets.
  - The performance of this fine-tuned network is compared with a network trained solely on a data set comprising 20% of the KITTI train set, using the KITTI test set for evaluation.
- **Sub-experiment 3:** Fine-Tuning with 50% of KITTI:
  - In this scenario, the network, initially pre-trained on AVX, is fine-tuned with 50% of the KITTI train set.

- Performance evaluation is undertaken on both KITTI and AVX test sets.
- The performance of this fine-tuned network is compared with a network trained solely on a data set comprising 50% of the KITTI train set, using the KITTI test set for evaluation.

5. Results and Discussion

The results presented and discussed in this section refer specifically to the 'Moderate' difficulty setting, as outlined in section 4.4.

5.1. Quantitative Analysis

5.1.1. Results from Experiment 1

Table 5.1 presents the AP performance results of the first experiment. The experiment involved object detection tasks for vehicles (CAR), pedestrians (PEDESTRIAN), and cyclists (CYCLIST), utilizing diverse data sets for training and testing —– the KITTI and AVX sets. In this context, 'Testing' implies the evaluation of the model.

Four evaluation metrics are employed in this experiment: Average Precision for 2D object detection (AP_{2D}), Average Precision for 3D object detection (AP_{3D}), Average Precision for BEV object detection (AP_{bev}), and Average Orientation Similarity (AOS). In Table 5.1, these metrics are signified as **Bbox** for AP_{2D}, **3d** for AP_{3D}, **bev** for AP_{bev}, and **aos** for Average Orientation Similarity. A detailed explanation of these metrics is provided in section 4.3.

The 'Training duration' column signifies the duration of the model's training phase, performed with a batch size of 2 and epoch size of 80, as detailed in section 4.2. The 'Train' and 'Test' columns indicate the data sets employed for training and testing the model, respectively. The subsequent columns report the AP values for different evaluation metrics across the three object categories.

High scores are represented in green color, while low scores are indicated in red in

Table 5.1.: Performance Metrics for Experiment 1.

Training duration	Train	Test	Average Precision											
			CAR				PEDESTRIAN				CYCLIST			
			Bbox	bev	3d	aos	Bbox	bev	3d	aos	Bbox	bev	3d	aos
07:43	KITTI	KITTI	91.84	88.12	76.47	91.61	63.28	56.62	48.42	35.07	72.61	66.92	63.64	68.07
		AVX	83.19	74.57	48.52	79.51	1.27	0.06	0.00	0.57	45.97	42.55	41.09	37.72
11:08	AVX	KITTI	42.88	52.41	10.75	40.62	0.01	0.00	0.00	0.00	1.67	1.48	0.66	0.96
		AVX	91.05	90.27	81.27	90.41	80.46	76.96	74.38	65.27	92.59	91.78	90.99	92.12

Table 5.1.

The first row of Table 5.1 depicts the performance of a model that has been trained and tested on the real-world data set, KITTI. Across all metrics and object categories, this model reports high AP values. This result aligns with the findings of the original study [2].

However, the second row introduces a contrast. When a model trained on the KITTI data set is tested on synthetic AVX data, there is a substantial drop in AP values, especially for the 'Pedestrian' category. This outcome implies that a model proficiently trained on real-world data may struggle when generalized to synthetic data, a problem that becomes more acute with complex subjects such as pedestrians.

This pattern is repeated in the third row, where a model, trained on synthetic AVX data, exhibits lesser performance when tested on the real-world KITTI data set. This finding emphasizes that while synthetic data offers an advantage in generating diverse training scenarios, it may not sufficiently prepare the model to tackle the complexities characteristic of real-world data.

Despite this, the fourth row illuminates the efficacy of synthetic data. It illustrates that a model can perform exceptionally well when the training and testing phases both utilize the synthetic data set. This insight highlights the potential value of synthetic data while underscoring the necessity for effective strategies to combine these data sets for a robust model performance across varied data distributions.

Pedestrian detection manifests distinct challenges evident in the results. The smaller image size of pedestrians and their propensity for occlusion by other objects in the scene create detection difficulties. When synthetic and real-world data present significant disparities, these challenges intensify, possibly leading to the notably low cross-validation scores observed for pedestrian detection.

Another factor contributing to these results is the object distribution disparity across the training sets, as discussed in section 4.1. The imbalance in the number of traffic objects between the real and synthetic databases may provoke biased learning in the model, resulting in over-prediction of more frequent objects in the training data.

The performance variance between the models trained and tested on differing data sets ('Train AVX, Test KITTI' versus 'Train KITTI, Test AVX') brings to light the inherent dissimilarities between synthetic and real-world data. Real-world data sets encapsulate higher complexity, incorporating a more extensive range of scenarios, object classes, and variability relative to synthetic data sets. It is worth noting that model trained on real-world data may have developed more robust and generalizable feature representations. This could have potentially boosted its performance on synthetic data, in contrast to the model trained on synthetic data and tested on real-world data.

The performance disparity across different metrics —- AP_{2D}, AP_{bev}, AP_{3D}, and AOS — can be attributed to the unique characteristics and complexity of the tasks evaluated by these metrics. Of particular note is AP_{3D}, which requires the prediction of the full 3D bounding box of an object, including its pose and dimensions in three-dimensional space. This task's complexity, coupled with limited variance in object instances in synthetic training data, is likely to contribute to lower performance during cross-validation.

The findings underscore the need for innovative training methodologies capable of leveraging the advantages of both real-world and synthetic data. Potential strategies could encompass fine-tuning or data mixing techniques. This would empower the model to learn from the rich complexity of real-world data and the controlled, diverse scenarios offered by synthetic data, thus enhancing its ability to generalize across varied data distributions.

5.1.2. Results from Experiment 2

Table 5.2 demonstrates the effect of training the model using various ratios of synthetic data and real-world data. The model's subsequent evaluation on both data sets yields results expressed using the same metrics as described in section 5.1.1. The final two rows serve as benchmarks, reproducing the conditions of section 5.1.1 wherein the model is trained and evaluated on identical data sets.

Upon examination, a distinct trend emerges across all object categories: the model's

Table 5.2.: Performance Metrics for Experiment 2.

Training duration	Train	Test	Average Precision CAR Bbox	CAR bev	CAR 3d	CAR aos	PEDESTRIAN Bbox	PEDESTRIAN bev	PEDESTRIAN 3d	PEDESTRIAN aos	CYCLIST Bbox	CYCLIST bev	CYCLIST 3d	CYCLIST aos
06:55	AVX 90% KITTI 10%	KITTI	87.53	84.44	69.09	87.08	33.93	28.52	21.43	16.86	56.77	56.03	56.03	52.44
		AVX	91.26	90.54	82.40	90.80	77.41	73.32	68.54	60.79	91.80	90.30	89.67	91.48
07:00	AVX 80% KITTI 20%	KITTI	89.07	87.20	74.16	88.74	40.16	38.44	30.89	18.19	63.24	55.53	53.35	59.63
		AVX	93.16	90.56	84.25	92.74	74.95	69.31	62.70	62.72	90.97	89.45	89.08	90.47
06:56	AVX 50% KITTI 50%	KITTI	91.03	87.54	75.81	90.78	53.62	51.06	42.04	27.57	67.05	59.52	54.91	63.52
		AVX	92.02	88.94	80.02	91.35	67.48	50.81	45.89	45.83	89.71	88.66	88.01	89.20
07:43	100% KITTI	KITTI	91.84	88.12	76.47	91.61	63.28	56.62	48.42	35.07	72.61	66.92	63.64	68.07
11:08	100% AVX	AVX	91.05	90.27	81.27	90.41	80.46	76.96	74.38	65.27	92.59	91.78	90.99	92.12

performance on the KITTI test set improves proportionally to the increase of KITTI data used during training. For example, within the 'Car' detection category, the AP_{2D} on the KITTI test set increases from 87.53 to 91.03 as the training data shifts from a composition of 90% AVX and 10% KITTI to an even 50-50 split. This performance trend is consistent across all categories and metrics, suggesting an increased ability of the model to handle the complexities of real-world scenarios as more real-world data is incorporated into the training set.

Focusing on individual categories yields further insight:

- **CAR:** For the 'Car' category, the model's performance on the AVX test set demonstrates considerable consistency despite the varying proportions of real-world and synthetic data in the training set. As the training set transitions from being predominantly synthetic data to a balanced mix, the AP_{2D} scores fluctuate minimally, ranging between 91.26 and 92.02. This implies the model's robust capability in interpreting synthetic data and its resilience to the incorporation of increasing amounts of real-world data in the training set. These outcomes underscore the model's generalizability and robustness in handling synthetic data sets.

  Notably, the model achieves competitive performance when trained on an equal mix of AVX and KITTI data, effectively rivaling the benchmarks established by the 100% KITTI and 100% AVX data. This suggests that a combination of synthetic and real-world data could yield results parallel to those obtained from training exclusively on either type of data. It also highlights the potential of synthetic data to augment real-world data without compromising the model's capability to navigate the complexities of real-world scenarios.

  These observations underscore the importance and potential of synthetic data, particularly in scenarios with limited real-world data. For instance, in the AVX 80% KITTI 20% experiment, only 742 real frames were available. The use of AVX to generate around 3000 synthetic frames (comprising 80% of the training set) resulted in near-baseline scores when evaluated on real data for the 'Car' category.

To provide a numerical perspective, the 3d baseline for 'Car' was 76.47, while the score for AVX 80% KITTI 20% was 74.16.

The implications of these findings are substantial, suggesting that training on a combined dataset of real and synthetic data can yield competitive results. Consequently, this could diminish the necessity for extensive real-world data collection.

- **PEDESTRIAN:** In the 'Pedestrian' category, the model's performance exhibits a considerable fluctuation, dependent on the blend of synthetic and real-world data applied during training. An enhancement in the model's performance on the KITTI test set correlates with an increased ratio of real-world data in the training set. However, the pedestrian detection scores on the AVX test set tend to decline as the proportion of KITTI data raises in the training set.

  The performance of the model, when trained on a mixture of data sets, falls short of the baselines established by models exclusively trained on 100% KITTI or 100% AVX data. The pedestrian detection scores persistently remain beneath both baselines, indicating substantial variations between the KITTI and AVX data sets concerning pedestrian representation.

  The evident performance gap between real and synthetic data for the 'Pedestrian' class highlights a discrepancy possibly arising from differential representation of pedestrians in the synthetic AVX data set compared to the real-world data. This variation may be linked to the limitations in synthetic data's ability to reproduce the diversity in pedestrian appearances, movements, and environments, or potentially arise from differences in annotation standards between the two data sets. As a direction for future work, efforts could be concentrated on improving the realism and variability of pedestrian scenarios within synthetic data set.

- **CYCLIST:** In the 'Cyclist' category, consistent model performance is evident across the AVX test set, regardless of the proportion of synthetic and real-world data used during training. The sustained high scores on the AVX test set under-

score the model's capability and versatility in detecting cyclists within synthetic environments.

A noteworthy observation is the peak in AP_{3D} of cyclists on the KITTI test set, when the training set is composed of 90% AVX and 10% KITTI data. This highlights the potential benefits of integrating real-world data into the synthetic data set.

Nevertheless, the AP values for 'Cyclist' detection on the KITTI test set consistently fall short when compared to those on the AVX test set across all training compositions. This discrepancy can be potentially attributed to the synthetic data's limited representation of cyclist diversity, as it only includes a single cyclist asset, as detailed in section 3.5. This observation indicates the need for incorporating a wider variety of cyclist assets into the synthetic data set to better replicate the complexities and variations present in real-world scenarios.

5.1.3. Results from Experiment 3

Table 5.3 presents the impact of different training methodologies, specifically highlighting the use of AVX synthetic data for pre-training, supplemented by fine-tuning on diverse subsets of the KITTI data set. This table shows the performance of the model across various training scenarios, both on the AVX and KITTI data sets. The evaluation metrics adopted for this analysis are consistent with those introduced in section 5.1.1. The final two rows in the table serve as a performance benchmark as they replicate the conditions outlined in section 5.1.1, where the training and testing data sets are identical.

Fine-tuning involves further training an already pre-trained model on a specific data set, which capitalizes on the pre-trained model's learned features, potentially reducing the computational resources and time needed for training.

The training scenarios represented by 10%, 20%, and 50% of the KITTI data set depict conditions of limited availability of real-world data. These models, pre-trained and

Table 5.3.: Performance Metrics for Experiment 3.

Training duration	Train	Test	Average Precision CAR Bbox	CAR bev	CAR 3d	CAR aos	PEDESTRIAN Bbox	PEDESTRIAN bev	PEDESTRIAN 3d	PEDESTRIAN aos	CYCLIST Bbox	CYCLIST bev	CYCLIST 3d	CYCLIST aos
00:43	10% KITTI	KITTI	85.92	83.18	66.02	85.50	43.61	38.01	27.79	23.08	57.41	53.43	49.93	48.05
00:43	pretrain on AVX, fine-tuning on 10% KITTI	KITTI	88.03	84.07	69.00	87.47	44.75	38.05	26.00	22.74	64.77	57.55	52.93	57.61
		AVX	80.87	74.09	45.33	79.43	4.24	0.28	0.03	2.14	62.88	59.48	49.64	47.31
01:26	20% KITTI	KITTI	88.92	85.15	72.73	88.46	46.01	43.62	37.36	23.74	63.11	57.23	52.15	55.16
01:25	pretrain on AVX, fine-tuning on 20% KITTI	KITTI	88.95	85.47	72.68	88.57	49.43	48.92	40.91	27.24	68.53	60.39	56.88	61.98
		AVX	81.99	75.41	51.88	80.29	1.96	0.07	0.02	1.22	69.73	61.75	53.40	60.23
03:34	50% KITTI	KITTI	91.12	87.25	75.55	90.81	55.49	54.10	45.73	29.67	71.89	64.12	58.64	69.04
03:29	pretrain on AVX, fine-tuning on 50% KITTI	KITTI	89.60	87.13	74.87	89.34	54.59	51.60	43.70	32.05	73.64	63.97	58.04	67.63
		AVX	82.72	74.63	47.00	79.75	2.07	0.07	0.02	1.13	73.63	68.86	66.05	65.55
07:43	100% KITTI	KITTI	91.84	88.12	76.47	91.61	63.28	56.62	48.42	35.07	72.61	66.92	63.64	68.07
11:08	100% AVX	AVX	91.05	90.27	81.27	90.41	80.46	76.96	74.38	65.27	92.59	91.78	90.99	92.12

subsequently fine-tuned, when evaluated against the KITTI data set, aim to enlighten the implications of using synthetic data in data-limited contexts.

Moreover, the pre-trained models are also evaluated on the AVX data set. This step is intended to explore whether the application of synthetic data could pave the way for new validation solutions, particularly when physical tests may be prohibitive in terms of cost.

An analysis of Table 5.3 reveals that the fine-tuning strategy generally leads to an enhanced performance on the KITTI test set as compared to models exclusively trained on respective subsets of KITTI data. This improvement, apparent across all categories and AP measures, demonstrates the benefits of pre-training on synthetic AVX data, followed by fine-tuning on real-world data from the KITTI data set. However, the performance of these hybrid models does not exceed those trained solely on either 100% KITTI or 100% AVX data, as indicated by the benchmark results. This finding suggests that while the hybrid approach can be beneficial, especially in scenarios with limited real-world data, it does not substitute for training on larger data sets.

- **CAR:** The 'Car' category, when examined, displays a notable trend. The advantages of pre-training using synthetic AVX data become particularly apparent, especially when there is a shortage of real-world KITTI data (e.g., 10% KITTI). The results show that models fine-tuned on a limited KITTI data subset closely approach the baseline performance levels. This demonstrates the potential of this scalable approach where the availability of theoretically limitless synthetic data can be employed to boost detection performance, especially when there is a shortage of labeled real-world data.

  Interestingly, there is a significant drop in the AP_{3D} metric when evaluated on AVX, despite satisfactory scores for the AP_{2D}, AP_{bev}, and AOS metrics. While the AP_{2D} is 82.72, the AP_{bev} is 74.63, and the AOS is 79.75, the AP_{3D} drops significantly to 47. This discrepancy calls for additional investigation.

  Nonetheless, the potential use of synthetic data for validating new solutions is

evident. For instance, in situations where introducing a new perception system, sensor, or technology may render extensive real-world data collection, like KITTI, impractical or undesirable, the use of AVX data for validation emerges as a viable alternative. Evaluation on the AVX data yields relevant scores for AP_{2D}, AP_{bev}, and AOS, although the AP_{3D} scores need improvement. This suggests that synthetic data could be a valuable tool for testing and validating new perception technologies and systems when acquiring real-world data presents challenges.

- **PEDESTRIAN:** Within the 'Pedestrian' category, the results gathered via pre-training and subsequent fine-tuning on 10% or 20% of KITTI data, followed by evaluation on the KITTI test set, seem to fall short of the baseline metrics obtained with 100% KITTI data. Nevertheless, when pre-training is complemented by fine-tuning on 50% of KITTI data, the scores approach those of the baseline (i.e., 100% KITTI data).

  For training sets comprising 10% of KITTI data, the method of pre-training on AVX followed by fine-tuning on KITTI yields only a slight increase in AP_{2D} and AP_{bev} scores. Conversely, there is a minor decline in AP_{3D} and AOS in comparison to models trained solely on 10% of KITTI data. Nevertheless, with training sets composed of 20% of KITTI data, the method of pre-training on AVX and fine-tuning on KITTI results in a significant improvement in performance. This progress underscores the benefits of incorporating a larger portion of real-world KITTI data during fine-tuning following the initial pre-training on synthetic AVX data. When training with 50% KITTI data, the fine-tuning method does not yield further enhancement in performance metrics, except for the AOS measure, when evaluated on the KITTI set.

  Contrarily, the model's performance on the AVX test set exhibits a sharp decline across all metrics following fine-tuning. This suggests that while the model can leverage valuable insights from synthetic AVX data during pre-training for application on real-world KITTI data, the reverse does not seem to be true. Thus,

it can be deduced that synthetic AVX data may not be suitably representative for 'Pedestrian' category for the purpose of validating new solutions. There exists an apparent discrepancy between the representation of pedestrians in synthetic AVX data and their real-world counterparts.

- **CYCLIST:** In the 'Cyclist' category, with a training set comprised of 10% KITTI data, the methodology involving pre-training on AVX data followed by fine-tuning on KITTI data results in superior AP values across the AP_{2D}, AP_{bev}, AP_{3D}, and AOS metrics in comparison to the model trained solely on 10% KITTI data. The findings suggest that even when real-world KITTI data is in short supply (10%), pre-training on synthetic AVX data enhances the model's ability to detect real cyclists. Hence, in situations where labeled real-world cyclist objects are limited, the usage of the AVX to generate abundant synthetic cyclist data could potentially improve the performance of the model.

  As the portion of KITTI data in the training set expands to 20%, the strategy of pre-training on AVX data and subsequent fine-tuning on KITTI data continues to enhance the AP values relative to the model trained solely on 20% KITTI data. This outcome further validates the advantage of synthetic AVX data pre-training.

  In scenarios where the training set contains 50% KITTI data, the fine-tuning approach continues to bring about an improvement in AP_{2D} metric. However, there is a slight dip in performance across other metrics when evaluated on the KITTI data set. This suggests that the benefits of pre-training on synthetic data become more noticeable in scenarios where real-world data is less abundant.

  When the model is evaluated on the AVX data set, the resulting metrics deviate considerably from the AVX benchmark. Nevertheless, in scenarios where the model is pre-trained on AVX data and fine-tuned on either 10% or 20% KITTI data, synthetic data can be utilized to validate new solutions, such as a new sensor, technology, or perception algorithm, especially for 'Cyclist' detection tasks. In such cases, the scores on AVX evaluation approximate the KITTI benchmarks,

implying that the simulated cyclist assets deliver a high degree of representation.

To summarize, the strategy of fine-tuning presents considerable potential for scalable applications, especially in situations where synthetic data is more readily available than its real-world counterpart. This approach allows the model to leverage the extensive synthetic data for pre-training and subsequently fine-tune its performance using a smaller, but more complex, real-world data set. However, the quality of representation in synthetic data remains a crucial element that significantly influences the overall results.

5.1.4. Analysis of Pre-training and Training Duration Impact on 3D AP Scores

The following analysis conducts an in-depth investigation into the effect of pre-training duration on the progressive evolution of performance metrics throughout successive epochs. The primary focus lies on the progression of AP_{3D} scores for the 'Car' and 'Cyclist' categories under particular training scenarios —— namely, training solely on 10% of the KITTI data set, and pre-training on the AVX data set, succeeded by fine-tuning on 10% of the KITTI data set.

Without pre-training, the 'Car' and 'Cyclist' categories achieve AP_{3D} scores of 66.02 and 49.93, respectively, as detailed in Table Table 5.3. Incorporating a pre-training phase boosts these metrics to 69 and 52.93, respectively. This elevation in performance metrics underscores the instrumental role pre-training plays in model performance, as previously highlighted in section 5.1.3.

However, it is crucial to note that these results are a product of 80 epochs of training, considering the epoch size was consistently set to 80 epochs. A thought-provoking proposition arises when considering the model's performance if the parameters were saved after every tenth epoch during the training process and subsequently evaluated on the KITTI data set. This scenario unfolds in Figure 5.1, which is composed of four sub-figures, wherein Figures 5.1a and 5.1b correspond to the 'Car' category, and Figures

5.1c and 5.1d relate to the 'Cyclist' category. Each sub-figure presents the AP_{3D} scores, depicted on the y-axis, across the number of epochs, represented on the x-axis. The discontinuity observed in the curves results from the parameters being saved every tenth epoch.

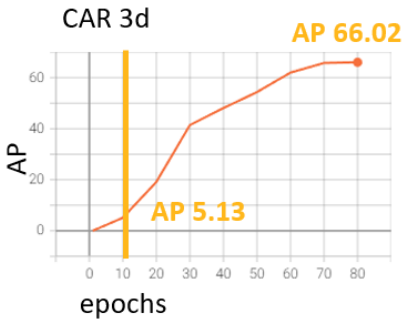
The horizontal yellow lines in Figures 5.1a, 5.1b, 5.1c, and 5.1d denote the performance metrics at the tenth epoch, with respective AP_{3D} values of 5.13, 68.71, 20.40, and 48. The final AP_{3D} scores obtained after the 80^{th} epoch are reported as 66.02, 69, 49.93, and 52.93, for each sub-figure respectively, reflecting the state of the network after 80 epochs.

These charts clearly show that pre-training followed by fine-tuning, as shown in Figure 5.1b and 5.1d, yields competitive scores as early as the tenth epoch. A potential critique may arise concerning the fluctuating nature of the performance in Figures 5.1b and 5.1d, yet it is important to note that even during these fluctuations, the AP values remain higher than the corresponding figures without pre-training at each epoch.

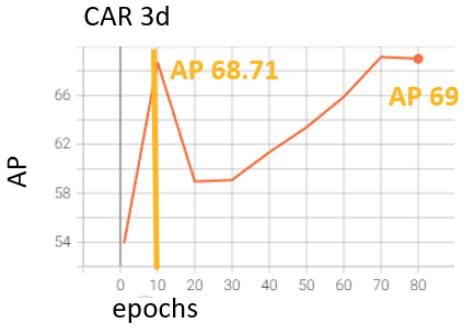
This early improvement in performance goes beyond simply better numbers. It highlights the important role synthetic data can play in quickly optimizing parameters, which is critical when the goal is to adapt perception systems for real-time use. Additionally, the efficiency of training time is highlighted, with competitive scores achieved within less than 10 minutes for 10 epochs, compared to 43 minutes for 80 epochs, as demonstrated in Table 5.3.

5.2. Qualitative Analysis

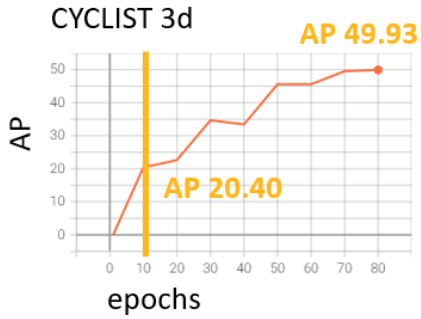
This section presents a qualitative analysis of the performance of the trained networks. In conjunction with the quantitative analysis, qualitative analysis is essential to provide an in-depth understanding of the model's performance and potential shortcomings. This analysis can reveal particular patterns, trends, or consistent errors, thereby illuminating the behavior of models under varying scenarios or with distinct object classes.



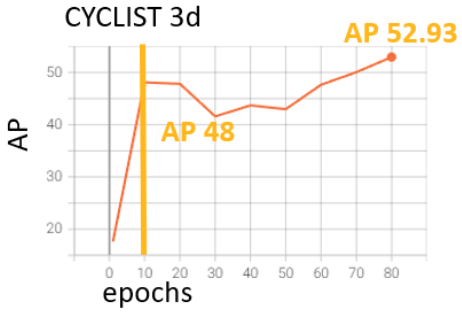
(a) Training with 10% KITTI, Evaluation on KITTI, showcasing CAR AP_{3D}.



(b) Training approach involves pre-training on AVX, followed by fine-tuning with 10% KITTI, Evaluation on KITTI, showcasing CAR AP_{3D}.



(c) Training with 10% KITTI, Evaluation on KITTI, showcasing CYCLIST AP_{3D}.



(d) Training approach involves pre-training on AVX, followed by fine-tuning with 10% KITTI, Evaluation on KITTI, showcasing CYCLIST AP_{3D}.

Figure 5.1.: Performance comparison of different training approaches on the AP_{3D} scores for 'Car' and 'Cyclist'

For a thorough understanding, this analysis will concentrate on two specific tests. The first test will involve detections from synthetic database, discussed in section 5.2.1, while the second will involve real database detections, detailed in section 5.2.2.

5.2.1. Analysis on AVX test set

This subsection conducts an investigation of a subset from the AVX test set, as mentioned in section 4.1. The analysis involves two networks: the AVX benchmark network (section 5.1.1) and another network pre-trained on AVX and subsequently fine-tuned with 20% of the KITTI data set (section 5.1.3).

While a video demonstration was initially planned, constraints of the PDF format prevent this. Nevertheless, visual representations of detection results are available in the form of video demonstrations assembled from screenshots at:

https://github.com/aydnzn/MasterThesis

These videos were generated using the Python script detailed in A.8.12.

Figure 5.2 depicts a synthetic scene that corresponds to the point clouds displayed in Figure 5.3. The color-coded objects, as seen in Figures 5.3a and 5.3b, consist of ground truth objects (blue), car detections (green), cyclist detections (yellow), and pedestrian detections (cyan). To achieve high recall, a confidence threshold of 0.1 was established, which resulted in numerous false positives and is consistent with the lower-right area of the precision-recall curve (Figure 4.1).

The first video, named video_1.mp4, aligns with Figure 5.3a and employs the network that underwent pre-training on AVX and subsequent fine-tuning with 20% of the KITTI data set. The second video, denoted as video_2.mp4, corresponds to Figure 5.3b and utilizes the AVX benchmark network (section 5.1.1). Despite the LiDAR's rotation rate of 10 Hz equating to an actual frame rate of 10 frames per second, both videos maintain a frame rate of 0.5 frames per second to enhance visualization.

Upon examining Figure 5.3, the presence of false positives is evident, primarily due to the low confidence threshold. Despite this, the intended objective — detecting

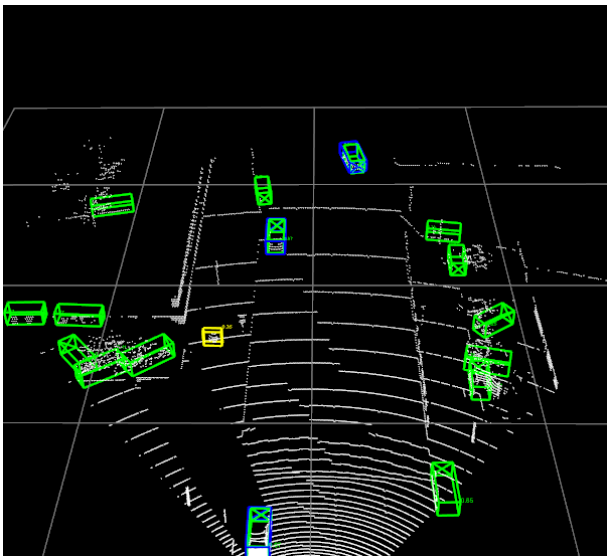
Figure 5.2.: Synthetic scene from Co-simulation. Figure in courtesy of [24].

all positive samples within the frames — has been achieved. It is worth noting that data imbalance can have substantial implications on detection results, evidenced by instances where short traffic poles are incorrectly classified as pedestrians. As indicated in section 4.1, the AVX train set, in comparison to the KITTI train set, contains a larger proportion of cyclists and pedestrians. This disparity is mirrored in the detection results of the network solely trained on the AVX data set (Figure 5.3b). In contrast, Figure 5.3a presents a multitude of car detections, an expected outcome considering the car-heavy composition of the KITTI train set.

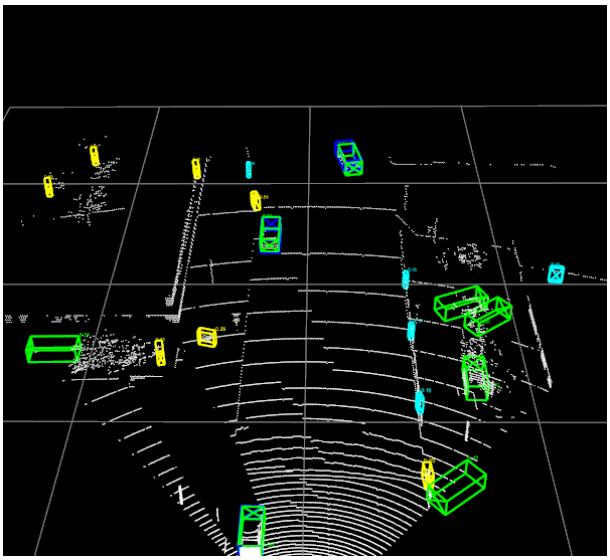
The AP_{3D} scores for the network trained and evaluated on AVX (Figure 5.3b) were reported as 81.27 for cars, 74.38 for pedestrians, and 90.99 for cyclists, as detailed in section 5.1.1. On the other hand, for the network that was pre-trained on AVX and subsequently fine-tuned on 20% KITTI (Figure 5.3a), the AP_{3D} scores on the AVX evaluation were 51.88 for cars, 0.02 for pedestrians, and 53.40 for cyclists, as indicated in section 5.1.3.

5.2.2. Analysis on KITTI test set

This subsection presents an evaluation of a subset derived from the KITTI test set, referenced in section 4.1. The examination involves two different networks: the bench-



(a) Pre-trained on AVX, fine-tuned on 20% KITTI, Tested on AVX.



(b) Trained and Tested on AVX.

Figure 5.3.: Color-coded detection results from different networks at a 0.1 confidence threshold. Key: ground truth objects (blue), cars (green), cyclists (yellow), pedestrians (cyan).

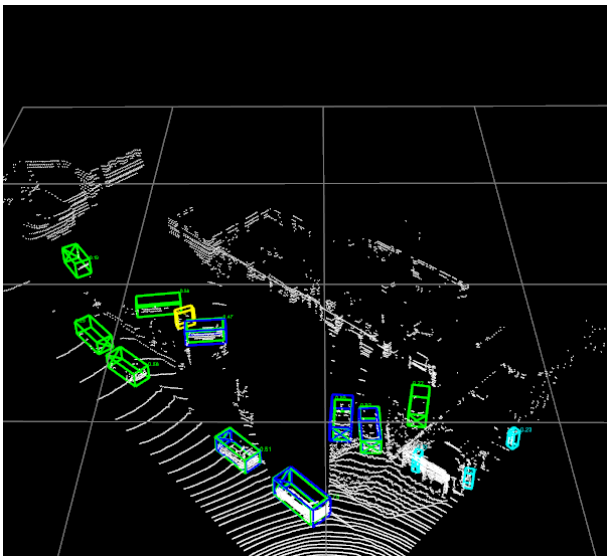
mark network trained on the KITTI data set (section 5.1.1), and another network that underwent pre-training on the AVX data set, followed by fine-tuning with 20% of the KITTI data set (section 5.1.3).

Figure 5.4 presents a real-world scene that matches the point clouds shown in Figure 5.5. Objects in the scene are color-coded for differentiation: blue signifies ground truth objects, green indicates car detections, yellow represents cyclist detections, and cyan marks pedestrian detections. A confidence threshold of 0.1 was established. This, however, resulted in an increased number of false positives.

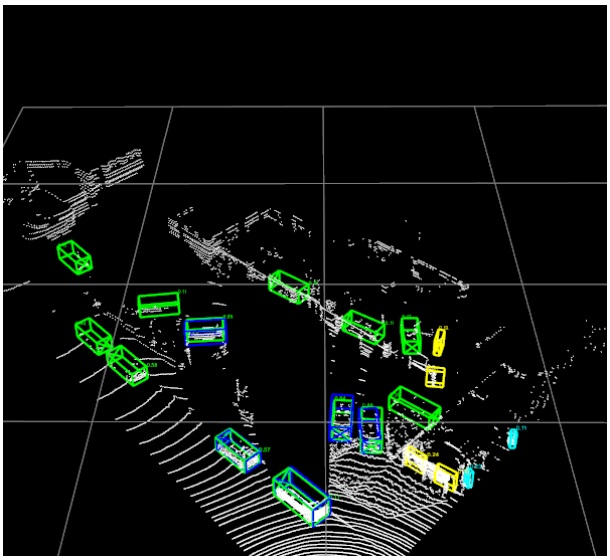
The corresponding videos, video_3.mp4 for Figure 5.5a and video_4.mp4 for Figure 5.5b, are available. The former employs the network pre-trained on AVX and fine-tuned with a 20% KITTI data set, while the latter utilizes the KITTI benchmark network (section 5.1.1). To facilitate a more comprehensive visualization, the videos are presented at a frame rate of 0.5 frames per second, despite the LiDAR's 10 Hz rotation rate theoretically equating to a frame rate of 10 frames per second. It is also worth noting that the video frames are not temporally consecutive.

Figure 5.4.: Image from KITTI data set [22].

Consistent with the results in section 5.2.1, Figure 5.5 exhibits a high number of false positives, attributed to the low confidence threshold. Yet, all labeled ground truths (blue) are successfully detected by both networks. This analysis also highlights the impact of 'DontCare' labels, as discussed in section 4.1. In Figure 5.4, two parked cars situated far from the camera are not labelled as cars in Figure 5.5, yet both networks successfully detected them. Importantly, these detections do not contribute to false positive calculations in the AP score computation. The network pre-trained and



(a) Pre-trained on AVX, fine-tuned on 20% KITTI, Tested on KITTI.



(b) Trained and Tested on KITTI.

Figure 5.5.: Color-coded detection results from different networks at a 0.1 confidence threshold. Key: ground truth objects (blue), cars (green), cyclists (yellow), pedestrians (cyan).

fine-tuned on 20% KITTI even demonstrated fewer false positives.

In terms of the AP_{3D} scores, the network trained and evaluated on the KITTI data set (Figure 5.5b) yielded scores of 76.47 for cars, 48.42 for pedestrians, and 63.64 for cyclists, as outlined in section 5.1.1. Conversely, the network that was pre-trained on the AVX data set and fine-tuned on 20% of the KITTI data set (Figure 5.5a) yielded AP_{3D} scores of 72.68 for cars, 40.91 for pedestrians, and 56.88 for cyclists during the KITTI evaluation, as described in section 5.1.3.

Although the insights derived from this initial analysis provide valuable information, it is important to view this as a starting point for a more comprehensive, extensive investigation. The manual comparison and examination of thousands of frames might not be a practical strategy. However, the insights gained from this type of evaluation can be essential for guiding potential future modifications and enhancements to the network.

6. Conclusion

6.1. Summary of findings

This thesis undertakes a comprehensive study of the use of both synthetic and real-world data for the training of object detection models, providing crucial insights into the performance of such models across various evaluation metrics.

The first experiment underscores the inherent challenges in generalizing models trained on one data type (synthetic or real-world) to another, with notable performance discrepancies when the model training and testing data types differ. These challenges become especially pronounced for complex entities such as pedestrians. This experiment thus underscores the need to develop innovative training methodologies capable of effectively utilizing both synthetic and real-world data.

The second experiment evaluates the impact of various ratios of synthetic to real-world training data on model performance. The findings suggest that increasing the proportion of real-world data (KITTI) in the training set tends to improve performance on the KITTI test set across all object categories. However, the 'Car' category shows remarkable resilience to changes in the composition of training data. Notably, a balanced combination of synthetic and real-world data provides competitive results, emphasizing the potential benefits of synthetic data utilization to augment real-world data.

The third experiment investigates a hybrid training approach, which involves synthetic data pre-training followed by fine-tuning on real-world data. This approach shows promise, especially under constraints of real-world data availability, although

the quality of synthetic data representation emerges as a crucial factor influencing model performance.

An additional examination of pre-training duration reveals the advantages of synthetic data pre-training before fine-tuning on real-world data. Such an approach notably accelerates the parameter optimization process, suggesting the potential role of synthetic data in facilitating the training process and boosting model performance, especially when real-world data is in short supply.

The qualitative analysis of trained networks provided a deeper understanding of how the models operated. The primary finding was that an imbalance in training data affects detection results. This analysis, though insightful, marks the beginning of a more comprehensive exploration into the performance and optimization of object detection models trained on varying compositions of synthetic and real-world data.

6.2. Recommendations for Future Work

This study has opened multiple avenues for future work that could further improve the performance and understanding of object detection models trained on synthetic and real-world data.

- **Enhancing Synthetic data sets:** The expansion of the synthetic data set, including an increase in object variety, could help to enhance the variability of data and improve model learning. Particular attention could be given to including more 'Car' objects, as the current AVX train set comprises only half of the 'Car' objects found in the KITTI train set.

- **Diversifying Scenarios:** There is an opportunity to broaden the scope of scenes used in training models and to employ a more diverse array of CarMaker scenarios. This could involve enhancing the Co-Simulation Library to facilitate more diverse scenarios.

- **Automating Scenario Creation:** The automation of scenario creation would sim-

plify the process of expanding the synthetic data set, making it more manageable and efficient.

- **Camera Simulation:** Future studies could explore the inclusion of actual camera simulation in training scenarios, rather than solely relying on hypothetical camera parameters.
- **Safe Driving Relation:** Investigating how detection performances relate to safe driving could yield interesting insights, linking object detection capabilities with practical, real-world outcomes.
- **Edge Case Testing and Safety Evaluation:** As part of a comprehensive evaluation, future research could include edge case testing and safety assessments.
- **Inclusion of Challenging Conditions:** Incorporating challenging conditions, such as nighttime or adverse weather, into simulations could improve the robustness of models and address the difficulty of collecting real-world data in such circumstances.
- **Exploring 'Car' Detection Discrepancies:** The significant drop in the AP_{3D} metric for 'Car' detection on the AVX test set [See section 5.1.3], despite high consistency in 'Car' detection overall, needs further investigation. Understanding the reasons behind this drop could provide insights for potential improvements.
- **Comprehensive Qualitative Analysis:** An in-depth qualitative analysis of detections and ground truths within both real and synthetic data sets could provide a more comprehensive understanding of the model's performance. This analysis could trigger a reevaluation of the existing labeling methodology.
- **Addressing Data Imbalance:** Data imbalance in the training set has been identified as a potential issue. Future work should consider this when combining real and synthetic data sets.

- **Comparing Reflectance Values:** The comparison of reflectance values between the real and synthetic data sets, using a possible similarity metric, could yield insightful results about data set similarities and differences.
- **Randomized Subsets for Percentage Trainings:** Future experiments could involve randomizing the selection process for creating subsets from the original data set. This could be followed by training multiple networks on different subsets and calculating the average performance score. This approach could provide a more reliable measure of the performance for a given percentage of data used in training, such as KITTI 20%.

Through these recommendations, future research could continue to expand our understanding of object detection, leveraging the strengths of both real-world and synthetic data.

6.3. Conclusion

In this Master's thesis, the use of synthetic data was examined to see how it could improve the performance of LiDAR-based 3D object detection algorithms, specifically with the PointPillars network. The main questions tackled included how to combine synthetic and real-world data, whether the performance of models trained on synthetic data would translate well to real-world scenarios, and how to use mixed data sets for training and testing neural networks. The practical use of fine-tuning techniques was also explored.

The work involved an extensive study of how synthetic and real-world data could be used in training 3D object detection models. The results provided crucial insights into the models' performance across various evaluation metrics. Even though it is challenging to generalize models trained on one type of data (synthetic or real-world) to another, new training methods showed promise in effectively using both synthetic and real-world data.

The experiments showed that when the share of real-world data in the training set was increased, performance on the KITTI test set improved across all object categories. Interestingly, the 'Car' category showed strong stability to changes in the composition of training data, keeping stable performance levels. Using a balanced mix of synthetic and real-world data also led to competitive results, highlighting the potential benefits of using synthetic data to supplement real-world data.

The research also showed that a two-step training approach, first training on synthetic data and then fine-tuning on real-world data, could be particularly useful when real-world data is scarce. An additional study into the length of pre-training further emphasized the benefits of starting training with synthetic data, as it helped speed up the optimization process and improved overall model performance.

A qualitative analysis of the trained networks further underlined how an imbalance in training data can affect detection results. However, these initial analyses are just the first step towards more comprehensive studies into how object detection models perform and can be optimized when trained on different mixes of synthetic and real-world data.

In conclusion, this study has made important progress in understanding how synthetic data can be used effectively. It fulfilled its primary goals and provided answers to the initial research questions, thus outlining potential directions for future research. Even though synthetic data has its limitations, this research emphasizes its considerable value, especially when used alongside real-world data within an effective training method.

A. Appendix

A.1. Velodyne HDL-64E User's Manual

This appendix section provides an overview of the essential excerpts from the User's Manual of the Velodyne HDL-64E. Designed to serve as a comprehensive reference point, this material aids in the understanding and effective utilization of the device.

USER'S MANUAL AND PROGRAMMING GUIDE

HDL-64E S3

High Definition LiDAR Sensor

Velodyne®

Principles of Operation

The sensor has 64 lasers fixed mounted on upper and lower laser blocks, each housing 32 lasers. Both laser blocks rotate as a single unit. With this design each of the lasers fires tens of thousands of times per second, providing exponentially more data points/second and a more data-intensive point cloud than a rotating mirror design. The sensor delivers a 360° horizontal Field of View (HFOV) and a 26.8° vertical FOV.

Additionally, state-of-the-art digital signal processing and waveform analysis are employed to provide high accuracy, extended distance sensing and intensity data. The sensor is rated to provide usable returns up to 120 meters. The sensor employs a direct drive motor system with no belts or chains in the drive train.

See the specifications at the end of this manual for more information about sensor operating conditions.

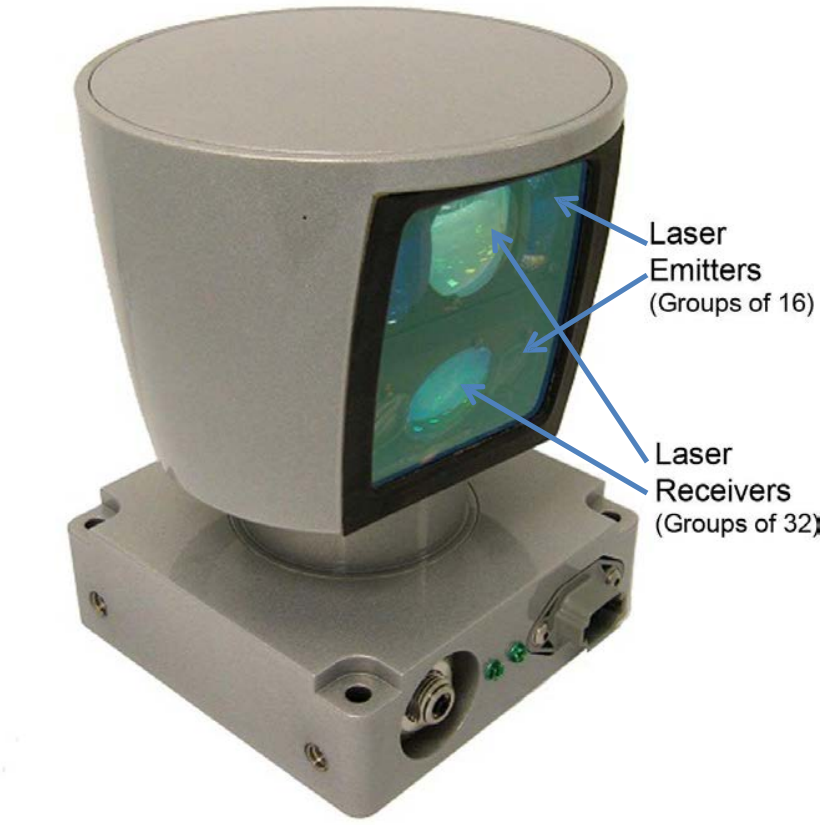


Figure 1: HDL-64E S3 Design Overview

Appendix A: Mechanical Drawings

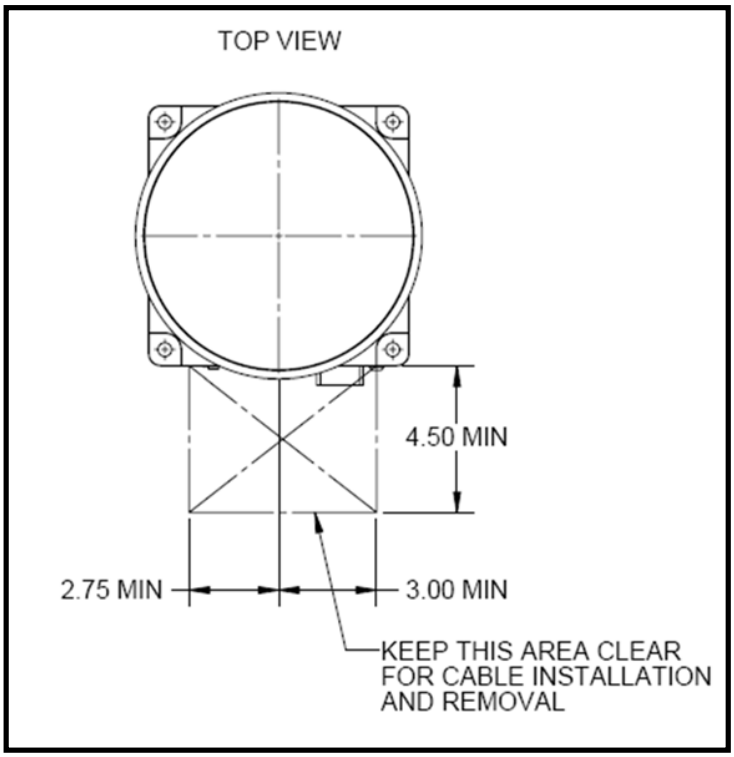


Figure 4: Top View

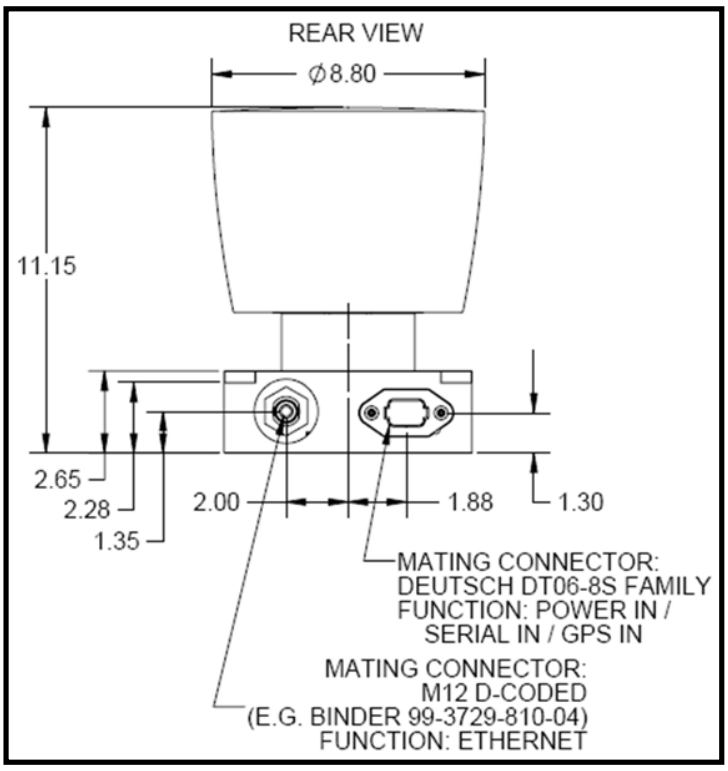


Figure 5: Rear Unit View

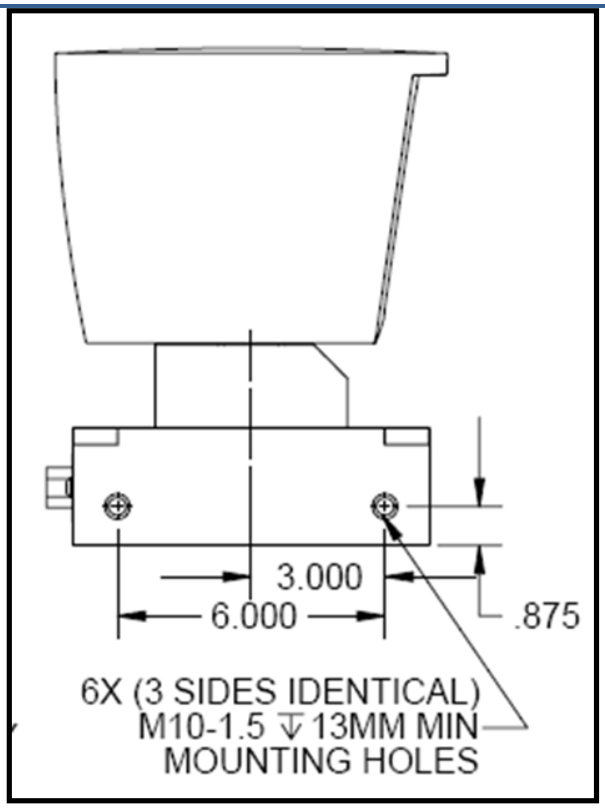


Figure 6: Side and Front Unit View

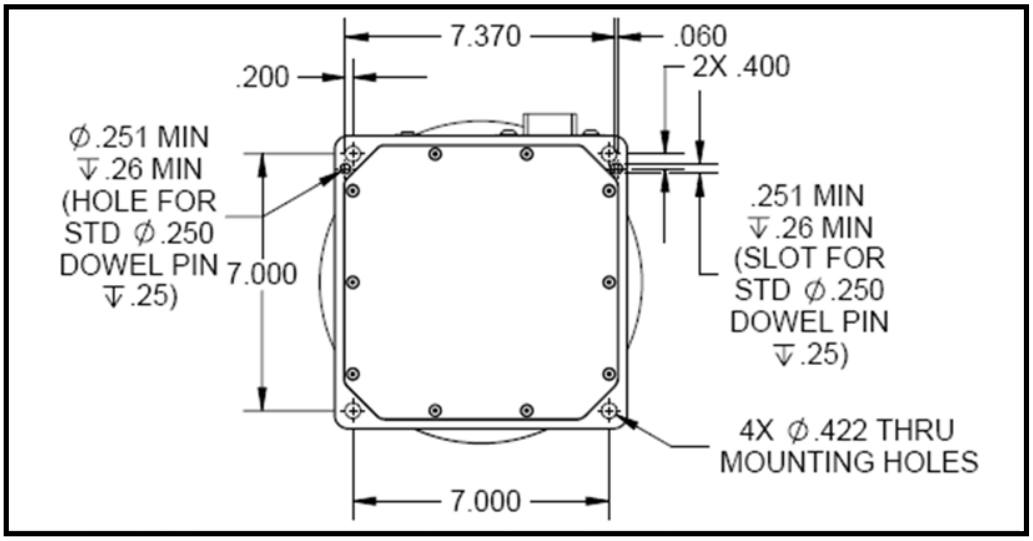


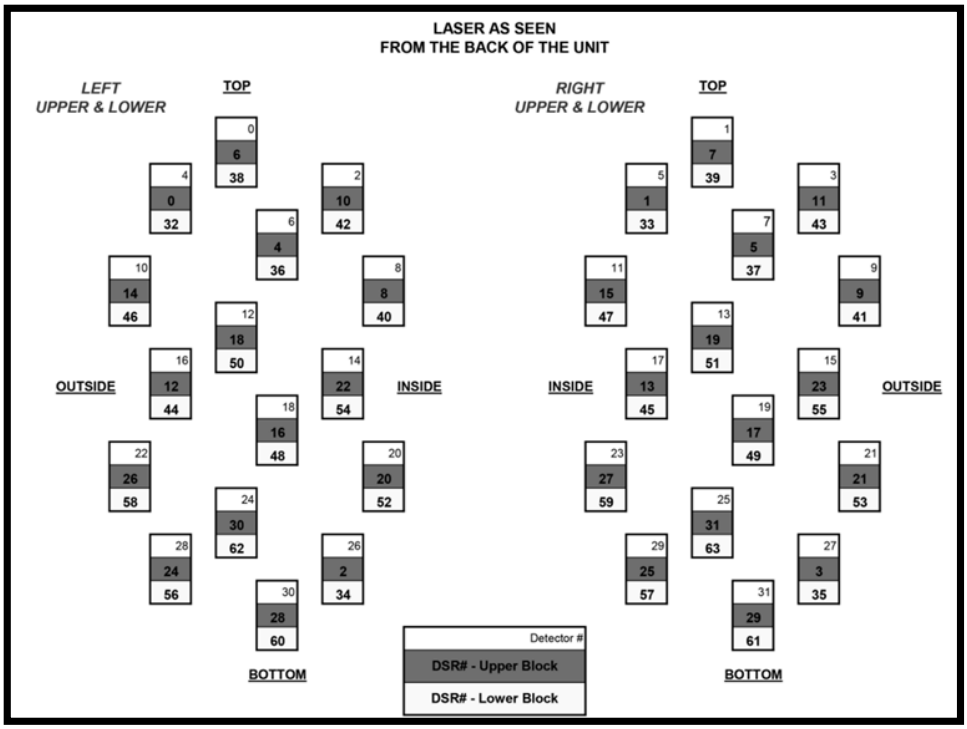
Figure 7: Bottom Unit View

Laser Firing Time Table for HDL64 S3 Firmware F2 Dual Return																	
Data Block	Laser Block	Laser Number 0–15 and 32–47 (Upper, Lower)															
		0, 32	1, 33	2, 34	3, 35	4, 36	5, 37	6, 38	7, 39	8, 40	9,41	10,42	11,43	12, 44	13, 45	14, 46	15, 47
1	Upper	172.8	171.5	170.3	169.1	165.6	164.3	163.1	161.9	158.4	157.1	155.9	154.7	151.2	149.9	148.7	147.5
2	Lower	172.8	171.5	170.3	169.1	165.6	164.3	163.1	161.9	158.4	157.1	155.9	154.7	151.2	149.9	148.7	147.5
3	Upper	172.8	171.5	170.3	169.1	165.6	164.3	163.1	161.9	158.4	157.1	155.9	154.7	151.2	149.9	148.7	147.5
4	Lower	172.8	171.5	170.3	169.1	165.6	164.3	163.1	161.9	158.4	157.1	155.9	154.7	151.2	149.9	148.7	147.5
5	Upper	115.2	113.9	112.7	111.5	108	106.7	105.5	104.3	100.8	99.5	98.3	97.1	93.6	92.3	91.1	89.9
6	Lower	115.2	113.9	112.7	111.5	108	106.7	105.5	104.3	100.8	99.5	98.3	97.1	93.6	92.3	91.1	89.9
7	Upper	115.2	113.9	112.7	111.5	108	106.7	105.5	104.3	100.8	99.5	98.3	97.1	93.6	92.3	91.1	89.9
8	Lower	115.2	113.9	112.7	111.5	108	106.7	105.5	104.3	100.8	99.5	98.3	97.1	93.6	92.3	91.1	89.9
9	Upper	57.6	56.3	55.1	53.9	50.4	49.1	47.9	46.7	43.2	41.9	40.7	39.5	36	34.7	33.5	32.3
10	Lower	57.6	56.3	55.1	53.9	50.4	49.1	47.9	46.7	43.2	41.9	40.7	39.5	36	34.7	33.5	32.3
11	Upper	57.6	56.3	55.1	53.9	50.4	49.1	47.9	46.7	43.2	41.9	40.7	39.5	36	34.7	33.5	32.3
12	Lower	57.6	56.3	55.1	53.9	50.4	49.1	47.9	46.7	43.2	41.9	40.7	39.5	36	34.7	33.5	32.3
Data Block	Laser Block	Laser Number 16—31 and 48—63 (Upper, Lower)															
		16,48	17,49	18,50	19,51	20,52	21,53	22,54	23,55	24,56	25,57	26,58	27,59	28,60	29,61	30,62	31,63
1	Upper	144	142.7	141.5	140.3	136.8	135.5	134.3	133.1	129.6	128.3	127.1	125.9	122.4	121.1	119.9	118.7
2	Lower	144	142.7	141.5	140.3	136.8	135.5	134.3	133.1	129.6	128.3	127.1	125.9	122.4	121.1	119.9	118.7
3	Upper	144	142.7	141.5	140.3	136.8	135.5	134.3	133.1	129.6	128.3	127.1	125.9	122.4	121.1	119.9	118.7
4	Lower	144	142.7	141.5	140.3	136.8	135.5	134.3	133.1	129.6	128.3	127.1	125.9	122.4	121.1	119.9	118.7
5	Upper	86.4	85.1	83.9	82.7	79.2	77.9	76.7	75.5	72	70.7	69.5	68.3	64.8	63.5	62.3	61.1
6	Lower	86.4	85.1	83.9	82.7	79.2	77.9	76.7	75.5	72	70.7	69.5	68.3	64.8	63.5	62.3	61.1
7	Upper	86.4	85.1	83.9	82.7	79.2	77.9	76.7	75.5	72	70.7	69.5	68.3	64.8	63.5	62.3	61.1
8	Lower	86.4	85.1	83.9	82.7	79.2	77.9	76.7	75.5	72	70.7	69.5	68.3	64.8	63.5	62.3	61.1
9	Upper	28.8	27.5	26.3	25.1	21.6	20.3	19.1	17.9	14.4	13.1	11.9	10.7	7.2	5.9	4.7	3.5
10	Lower	28.8	27.5	26.3	25.1	21.6	20.3	19.1	17.9	14.4	13.1	11.9	10.7	7.2	5.9	4.7	3.5
11	Upper	28.8	27.5	26.3	25.1	21.6	20.3	19.1	17.9	14.4	13.1	11.9	10.7	7.2	5.9	4.7	3.5
12	Lower	28.8	27.5	26.3	25.1	21.6	20.3	19.1	17.9	14.4	13.1	11.9	10.7	7.2	5.9	4.7	3.5

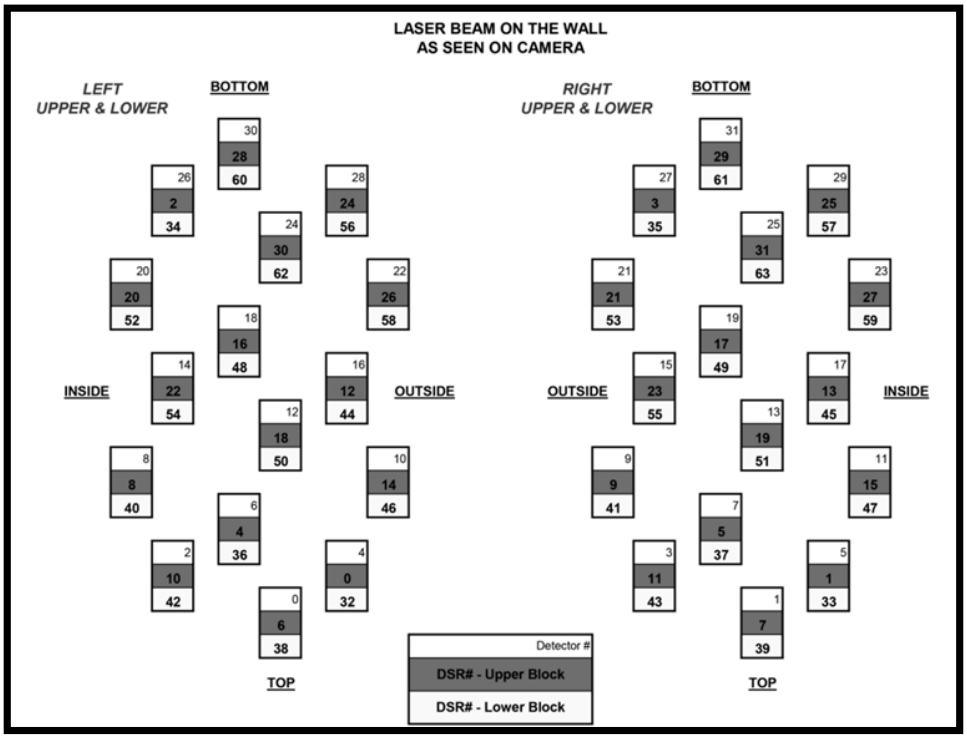
Laser and Detector Arrangement

The images below show the arrangement of the lasers and detectors in the unit.

Lasers as Seen from the Back of the Unit



Laser as Seen on the Wall



Appendix I: Angular Resolution

RPM	RPS (Hz)	Total Laser Points per Revolution	Points Per Laser per Revolution	Angular Resolution (degrees)
0300	5	266,627	4167	0.0864
0600	10	133,333	2083	0.1728
0900	15	88,889	1389	0.2592
1200	20	66,657	1042	0.3456

Appendix L: Specifications

Sensor	. 64 lasers/detectors . 360 degree field of view (azimuth) . 0.09 degree angular resolution (azimuth) . Vertical Field of View (26.8 degrees): o +2 to -8.33 @ 1/3 degree spacing o -8.83 to -24.33 @ 1/2 degree spacing . < 2cm distance accuracy (one sigma) . 5 - 20 Hz field of view update (user selectable) . 50 meter range for pavement (~0.10 reflectivity) . 120 meter range for cars and foliage (~0.80 reflectivity) . **Data rate:** o **Single return:** about 1.333 million points per second o **Dual return:** about 1.00 million points per second . **Operating temperature:** -40° to 85° C . **Storage temperature:** -50° to 90° C . **Vibration:** 0.1 g^2/Hz from 24 to 1000 Hz (equivalent to 9.9 Grms)
Laser	. Class 1 - eye safe . 4 x 16 laser block assemblies . 905 nm wavelength . 5 nanosecond pulse (time of flight) . Adaptive power system for minimizing saturation, extended laser life and enhanced eye safety . EMC capability: EC-42 version
Electrical	. **Input Voltage Range:** 10V to 32 V o At 12V draws 4 amps o At 24V draws 2 amps
Mechanical	. 0300 RPM - 1200 RPM spin rate (user selectable) . Environmental Protection rated to IP67 (+/- 35 kPa)
Output	. 100 MBPS UDP Ethernet packets
Dimensions (H x W x D)	. **Unit:** 11.15” x 8.80” x 9.10” [283mm x 223.5mm x 231.1mm] . **Crate:** 18.8” x 18.8” x 17.375” [477.5mm x 477.5mm x 441.3mm]
Weight (approximate)	. 70 lbs [31.8 Kg] Shipping (Crate + Sensor Unit) . 30 lbs [13.6 Kg] Sensor Unit

A.2. AVX LiDAR Sensor Simulation Parameters

The given JSON file represents the parameter configuration for the AVX. The parameters define the setup and properties of the LiDAR sensor simulation.

```
{
    "sensorSimulationParameters": [{
            "identifier": "lidar",
            "recordingFormat": {
                "lidarRecordingFormat": "TEXT"
            },
            "lidarSimulation": {
                "contribution": true,
                "grid": {
                    "polarGrid": {
                        "angularGridPoints": 1,
                        "hasCentralPoint": false,
                        "radialGridPoints": 1
                    }
                },
                "waveform": false,
                "numberOfBatches":1
            }
        }
    ]
}
```

A.3. Point Cloud Output File

The provided text file represents a subset of the point cloud output generated by AVX, capturing the initial 30 points within the point cloud. A detailed exploration of its structure is available in section 3.3.1.

```
-1 -1 1
0 0 0 0 1
0.1314314753 -1.723893285 8.091629982 7.631024346E-06 1
0 0 0 0 1
-0.3064283729 -1.722250819 7.739273071 8.24406743E-06 1
0 0 0 0 1
```

```
0.4029927254 -1.728716731 10.96961594 4.191184416E-06 1
0 0 0 0 1
-0.1975486577 -1.725240111 10.34804344 4.820525646E-06 1
-4.601933002 -0.8883528709 72.31811523 3.329478204E-08 1
-0.7053152919 -1.723534346 7.404108047 8.911592886E-06 1
-9.939009666 -1.796225667 98.23377228 9.429641068E-08 1
-1.073943019 -1.719437599 7.049202442 9.640585631E-06 1
0 0 0 0 1
-0.7287546992 -1.724409699 9.775464058 5.519250408E-06 1
0 0 0 0 1
-1.212251425 -1.724744201 9.235898018 6.048474461E-06 1
2.530569077 -1.753721595 29.23798752 3.592576832E-07 1
0.7217442989 -1.718410015 5.530332088 1.617823727E-05 1
1.306893826 -1.750718117 26.61198616 4.742760211E-07 1
0.3981485665 -1.719488978 5.394683838 1.72983855E-05 1
4.795693398 -1.157847166 47.75258636 7.310882211E-07 1
1.028606057 -1.719696879 6.785401344 1.044012606E-05 1
3.201788664 -1.533663034 50.92105484 3.771856427E-08 1
0.622361362 -1.720337272 6.585893154 1.131906174E-05 1
0.2853425741 -1.645746231 22.96339417 6.889458746E-07 1
0.09619765729 -1.71772778 5.24103117 1.848535612E-05 1
-0.5208556652 -1.633357882 21.03564262 9.996816516E-06 1
-0.1907826066 -1.719530582 5.090302944 1.97456684E-05 1
1.295488715 -1.785065055 49.57660294 8.312053978E-08 1
0.247970745 -1.721892238 6.382583618 1.22867059E-05 1
```

A.4. Contribution Output File

The provided text file represents a subset of the contribution output generated by AVX, capturing the first 30 points within the point cloud. A detailed exploration of its structure is available in section 3.3.2.

```

34 1

34 1

34 1

```

```
34 1
202 1
34 1
33 1
34 1

34 1

34 1
34 1
34 1
34 1
34 1
159 1
34 1
164 1
34 1
34 1
34 1
87 1
34 1
34 1
34 1
34 1
```

A.5. Contribution Dictionary File

The provided file embodies a fraction of the contribution dictionary generated by AVX. Within this file, the 'instanceName' denotes the traffic object to which the EntityID corresponds. The subset under inspection indicates that all the included EntityIDs correspond to the traffic object 'CAR1'. It also reveals that each entity identification number is connected to a distinctive mesh within the simulated environment. For instance, EntityID 146 correlates with a mesh crafted in the wheel of the car.


```
{'EntityID': 136, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
```

```
    meshPartName': 'vhl_lightpositionback', 'materialPath': '..\\Materials\\vhl_lightpositionback
    '}}
{'EntityID': 137, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightbrake', 'materialPath': '..\\Materials\\vhl_lightbrake'}}
{'EntityID': 138, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_id', 'materialPath': '..\\Materials\\vhl_id'}}
{'EntityID': 139, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightfogfront', 'materialPath': '..\\Materials\\vhl_lightfogfront'}}
{'EntityID': 140, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightblinkingright', 'materialPath': '..\\Materials\\
    vhl_lightblinkingright'}}
{'EntityID': 141, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightpositionfront', 'materialPath': '..\\Materials\\
    vhl_lightpositionfront'}}
{'EntityID': 142, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightbackup', 'materialPath': '..\\Materials\\vhl_lightbackup'}}
{'EntityID': 143, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_lightblinkingleft', 'materialPath': '..\\Materials\\vhl_lightblinkingleft
    '}}
{'EntityID': 144, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
```

```
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_body', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag':
     'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_carpaint', 'materialPath': '..\\Materials\\vhl_carpaint'}}
{'EntityID': 145, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_glass', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'tag
    ': 'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_glass', 'materialPath': '..\\Materials\\vhl_glass'}}
{'EntityID': 146, 'assetDescription': {'instanceName': 'CAR1', 'assetPath': 'C:\\Users\\auzun\\
    AppData\\Local\\Temp\\gwnxocqk.dae\\audi_a1_2010_orange.scenetree'}, 'nodeHierarchy': [{'name
    ': 'highway_japan/CAR1/vhl_wheel01', 'tags': []}, {'name': 'highway_japan/CAR1', 'tags': [{'
    tag': 'Vehicle', 'label': ''}]}, {'name': 'highway_japan', 'tags': []}], 'meshDescription': {'
    meshPartName': 'vhl_wheelgum', 'materialPath': '..\\Materials\\vhl_wheelgum'}}
```

A.6. Traffic Objects and their Entitiy Identifiers

This section presents a compilation of traffic objects, also referred to as instance names, aligned with their respective entity identification numbers. For a recapitulation on the concepts of 'instanceName' and EntityID, refer back to A.5. The list of associations is provided below:

```
CAR1: ['136', '137', '138', '139', '140', '141', '142', '143', '144', '145', '146', '147', '148',
    '149']
CAR2: ['151', '152', '153', '154', '155', '156', '157', '158', '159', '160', '161', '162', '164',
    '166', '169']
CAR3: ['171', '172', '173', '174', '175', '176', '177', '178', '179', '180', '181', '182', '183',
    '184', '185']
CAR4: ['187', '188', '189', '190', '191', '192', '193', '194', '195', '196', '197', '199', '200']
CAR5: ['202', '203', '204', '205', '206', '207', '208', '209', '210', '211', '212', '213', '214',
    '215', '216', '217', '218']
BIC1: ['220', '221']
PED1: ['223', '224', '225', '226', '227', '228', '229']
PED2: ['231', '232', '233', '234', '235', '236', '237']
```

A.7. Pointpillars Network Configuration

The script outlined in this section, sourced from [58], highlights the network configuration implemented throughout all experimental procedures. Further details can be found in section 4.2.

```
# This script is sourced from https://github.com/open-mmlab/OpenPCDet
CLASS_NAMES: ['Car','Pedestrian','Cyclist']

DATA_CONFIG:
    _BASE_CONFIG_: cfgs/dataset_configs/kitti.yaml
    POINT_CLOUD_RANGE: [0, -39.68, -3, 69.12, 39.68, 1]
    DATA_PROCESSOR:
        - NAME: mask_points_and_boxes_outside_range
          REMOVE_OUTSIDE_BOXES: True

        - NAME: shuffle_points
          SHUFFLE_ENABLED: {
            'train': True,
            'test': False
          }

        - NAME: transform_points_to_voxels
          VOXEL_SIZE: [0.16, 0.16, 4]
          MAX_POINTS_PER_VOXEL: 100
          MAX_NUMBER_OF_VOXELS: {
            'train': 12000,
            'test': 12000
          }
    DATA_AUGMENTOR:
        DISABLE_AUG_LIST: ['placeholder']
        AUG_CONFIG_LIST:
            - NAME: gt_sampling
              USE_ROAD_PLANE: False
              DB_INFO_PATH:
                  - kitti_dbinfos_train.pkl
              PREPARE: {
                 filter_by_min_points: ['Car:5','Cyclist:5'],
              }

              SAMPLE_GROUPS: ['Car:15','Cyclist:8']
```

```
            NUM_POINT_FEATURES: 4
            DATABASE_WITH_FAKELIDAR: False
            REMOVE_EXTRA_WIDTH: [0.0, 0.0, 0.0]
            LIMIT_WHOLE_SCENE: True
          - NAME: random_world_flip
            ALONG_AXIS_LIST: ['x']

          - NAME: random_world_rotation
            WORLD_ROT_ANGLE: [-0.78539816, 0.78539816]

          - NAME: random_world_scaling
            WORLD_SCALE_RANGE: [0.95, 1.05]


MODEL:
    NAME: PointPillar

    VFE:
        NAME: PillarVFE
        WITH_DISTANCE: False
        USE_ABSLOTE_XYZ: True
        USE_NORM: True
        NUM_FILTERS: [64]

    MAP_TO_BEV:
        NAME: PointPillarScatter
        NUM_BEV_FEATURES: 64

    BACKBONE_2D:
        NAME: BaseBEVBackbone
        LAYER_NUMS: [3, 5, 5]
        LAYER_STRIDES: [2, 2, 2]
        NUM_FILTERS: [64, 128, 256]
        UPSAMPLE_STRIDES: [1, 2, 4]
        NUM_UPSAMPLE_FILTERS: [128, 128, 128]

    DENSE_HEAD:
        NAME: AnchorHeadSingle
        CLASS_AGNOSTIC: False
        USE_DIRECTION_CLASSIFIER: True
        DIR_OFFSET: 0.78539
        DIR_LIMIT_OFFSET: 0.0
```

```
77        NUM_DIR_BINS: 2
78
79        ANCHOR_GENERATOR_CONFIG: [
80            {
81                'class_name': 'Car',
82                'anchor_sizes': [[3.9, 1.6, 1.56]],
83                'anchor_rotations': [0, 1.57],
84                'anchor_bottom_heights': [-1.78],
85                'align_center': False,
86                'feature_map_stride': 2,
87                'matched_threshold': 0.6,
88                'unmatched_threshold': 0.45
89            },
90            {
91                'class_name': 'Pedestrian',
92                'anchor_sizes': [[0.8, 0.6, 1.73]],
93                'anchor_rotations': [0, 1.57],
94                'anchor_bottom_heights': [-1.465],
95                'align_center': False,
96                'feature_map_stride': 2,
97                'matched_threshold': 0.5,
98                'unmatched_threshold': 0.35
99            },
100           {
101               'class_name': 'Cyclist',
102               'anchor_sizes': [[1.76, 0.6, 1.73]],
103               'anchor_rotations': [0, 1.57],
104               'anchor_bottom_heights': [-1.465],
105               'align_center': False,
106               'feature_map_stride': 2,
107               'matched_threshold': 0.5,
108               'unmatched_threshold': 0.35
109           }
110       ]
111
112       TARGET_ASSIGNER_CONFIG:
113           NAME: AxisAlignedTargetAssigner
114           POS_FRACTION: -1.0
115           SAMPLE_SIZE: 512
116           NORM_BY_NUM_EXAMPLES: False
117           MATCH_HEIGHT: False
```

```
            BOX_CODER: ResidualCoder

        LOSS_CONFIG:
            LOSS_WEIGHTS: {
                'cls_weight': 1.0,
                'loc_weight': 2.0,
                'dir_weight': 0.2,
                'code_weights': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
            }

    POST_PROCESSING:
        RECALL_THRESH_LIST: [0.3, 0.5, 0.7]
        SCORE_THRESH: 0.1
        OUTPUT_RAW_SCORE: False

        EVAL_METRIC: kitti

        NMS_CONFIG:
            MULTI_CLASSES_NMS: False
            NMS_TYPE: nms_gpu
            NMS_THRESH: 0.01
            NMS_PRE_MAXSIZE: 4096
            NMS_POST_MAXSIZE: 500


OPTIMIZATION:
    BATCH_SIZE_PER_GPU: 4
    NUM_EPOCHS: 80

    OPTIMIZER: adam_onecycle
    LR: 0.003
    WEIGHT_DECAY: 0.01
    MOMENTUM: 0.9

    MOMS: [0.95, 0.85]
    PCT_START: 0.4
    DIV_FACTOR: 10
    DECAY_STEP_LIST: [35, 45]
    LR_DECAY: 0.1
    LR_CLIP: 0.0000001

```

```
    LR_WARMUP: False
    WARMUP_EPOCH: 1

    GRAD_NORM_CLIP: 10
```

A.8. Python Scripts

The following are Python scripts deployed during the course of this research. These scripts have been previously discussed in their corresponding sections within the thesis.

A.8.1. Identical Point Cloud Deletion

For a detailed understanding of its application, refer to the section 3.4.3.1.

```
import os
from collections import defaultdict
import filecmp
import sys

# Define paths
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")
modified_dir = os.path.join(chosen_subdir, "Modified")

# Create directory if it doesn't exist
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Function to find identical point cloud files
def find_identical_point_clouds(folder_path):
    # Get all point cloud files
    point_cloud_files = [f for f in os.listdir(folder_path) if f.startswith('lidar') and f.
        endswith('pointcloud.txt')]
    identical_clouds = []

    # Compare each pair of files
    for i in range(len(point_cloud_files)):
        file1 = point_cloud_files[i]
```

```
        for j in range(i + 1, len(point_cloud_files)):
            file2 = point_cloud_files[j]
            # If files are identical, add to the list
            if filecmp.cmp(os.path.join(folder_path, file1), os.path.join(folder_path, file2)):
                identical_clouds.append((file1, file2))

    return identical_clouds

# Function to delete identical point cloud files
def delete_identical_point_clouds(folder_path):
    identical_clouds = find_identical_point_clouds(folder_path)
    deleted_files = []

    # Delete each identical file
    for file1, file2 in identical_clouds:
        file_to_delete = os.path.join(folder_path, file2)
        contribution_file_to_delete = file_to_delete.replace("pointcloud.txt", "contributions.txt")
        # If both files exist, delete them
        if os.path.exists(file_to_delete) and os.path.exists(contribution_file_to_delete):
            os.remove(file_to_delete)
            os.remove(contribution_file_to_delete)
            deleted_files.extend([file_to_delete, contribution_file_to_delete])

    return deleted_files

# Delete identical point clouds
deleted_files = delete_identical_point_clouds(chosen_subdir)

# Create an output file listing the deleted files
output_file_path = os.path.join(modified_dir, "deleted_files.txt")
with open(output_file_path, "w") as output_file:
    output_file.write("\n".join(deleted_files))

# Print out the deleted files and the path to the output file
print("Deleted files:\n", "\n".join(deleted_files))
print("Output file created:", output_file_path)
```

A.8.2. Point Cloud Transformation

For a detailed understanding of its application, refer to the section 3.4.3.2.

```python
import os
import numpy as np
import sys

# Define the absolute path to the parent folder
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
# Extract the name of the subdirectory from the script argument
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
# Form the full path to the chosen subdirectory
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Create a new directory named "Modified" inside chosen_subdir, if it doesn't exist yet
modified_dir = os.path.join(chosen_subdir, "Modified")
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Loop over all .txt files in the chosen subdirectory
for filename in os.listdir(chosen_subdir):
    if filename.endswith("pointcloud.txt"):
        # Load the .txt file into a Numpy array
        file_path = os.path.join(chosen_subdir, filename)
        # Load data from columns (0, 1, 2, 3), skipping the first row
        data = np.loadtxt(file_path, dtype='float', usecols=(0, 1, 2, 3), skiprows=1)
        # Swap columns 0 and 2, and 1 and 0 to achieve the required transformation
        data[:, [0, 1, 2]] = data[:, [2, 0, 1]]
        # Save the transformed Numpy array as a .npy file in the "Modified" directory
        npy_filename = os.path.splitext(filename)[0] + '.npy'
        np.save(os.path.join(modified_dir, npy_filename), data)

# Print a success message
print("Conversion complete!")
```

A.8.3. Read Contribution Dictionary

For a detailed understanding of its application, refer to the section 3.4.3.3.

```python
import os
import json
import sys

```

```
# Define the absolute path to the parent folder
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))

# Extract the name of the subdirectory from the script argument
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
# Form the full path to the chosen subdirectory
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Loop over all .json files in the chosen subdirectory
for filename in os.listdir(chosen_subdir):
    if filename.endswith(".json"):
        json_path = os.path.join(chosen_subdir, filename)
        # Open the .json file and load its contents
        with open(json_path, 'r') as f:
            data = json.load(f)
            # Extract the 'items' from the .json data
            text = data['items']

            # Create a corresponding .txt filename
            text_file = filename.replace(".json", ".txt")
            text_path = os.path.join(chosen_subdir, text_file)

            # Open the new .txt file and write each 'item' on a new line
            with open(text_path, 'w') as f:
                for item in text:
                    f.write(str(item) + '\n')

# Print a success message
print("Conversion complete!")
```

A.8.4. Extract Instance Dimensions

For a detailed understanding of its application, refer to the section 3.4.3.4.

```
import os
import numpy as np
import sys

# Define the path to the folder containing the subdirectories
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
```

```
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Define instance mapping dictionary
instance_mapping = {'CAR1': 'Car', 'CAR2' : 'Car','CAR3': 'Car', 'CAR4' : 'Car','CAR5': 'Car','
    CAR6': 'Car','CAR7': 'Car','CAR8': 'Car', 'PED1': 'Pedestrian','PED2': 'Pedestrian', 'PED3': '
    Pedestrian', 'PED4': 'Pedestrian', 'BIC1': 'Bicycle', 'BIC2': 'Bicycle', 'BIC3': 'Bicycle', '
    BIC4': 'Bicycle'}

# Define model dimensions dictionary
dimensions = {
    'audi_a1_2010_blue':[3.954,1.774,1.474],
    'audi_a1_2010_long_range_blue':[3.954,1.774,1.474],
    'audi_a1_2010_orange':[3.954,1.774,1.474],
    'audi_a1_2010_red':[3.954,1.774,1.474],
    'audi_a1_2010_white':[3.954,1.774,1.474],
    'audi_a3_2013_red' : [4.333,2,1.434],
    'audi_a3_2013_red_engine_running_thermal':[4.333,2,1.434],
    'audi_a3_2013_red_engine_off_thermal':[4.333,2,1.434],
    'audi_a3_sportback_red':[4.3,2.0,1.427],
    'bmw_x5_2009_black':[4.838,2.073,1.896],
    'bmw_x5_2009_blue' :[4.838,2.073,1.896],
    'bmw_x5_2009_grey':[4.838,2.073,1.896],
    'bmw_x5_2009_red':[4.838,2.073,1.896],
    'bmw_x5_2009_white':[4.838,2.073,1.896],
    'citroen_nemo_2007_black':[3.872,1.642,1.715],
    'citroen_nemo_2007_blue':[3.872,1.642,1.715],
    'citroen_nemo_2007_grey':[3.872,1.642,1.715],
    'citroen_nemo_2007_red':[3.872,1.642,1.715],
    'citroen_nemo_2007_white':[3.872,1.642,1.715],
    'dodge_charger_2008_grey':[4.864,1.887,1.515],
    'dodge_charger_2008_red':[4.864,1.887,1.515],
    'dodge_charger_2008_white':[4.864,1.887,1.515],
    'dodge_charger_2008_yellow':[4.864,1.887,1.515],
    'dodge_charger_2018_yellow_engine_running_thermal':[4.864,1.887,1.515],
    'dodge_charger_2018_yellow_engine_off_thermal':[4.864,1.887,1.515],
    'genesis_g80_2017_black': [4.99,2.139, 1.48],
    'genesis_g80_2017_blue': [4.99,2.139, 1.48],
    'jaguar_e_pace_2018_white':[4.542,2.05,1.71],
    'kia_soul_2010_black':[4.077,1.783,1.662],
    'kia_soul_2010_orange':[4.077,1.783,1.662],
    'kia_soul_2010_red':[4.077,1.783,1.662],
```

```
    'kia_soul_2010_white':[4.077,1.783,1.662],
    'kia_soul_2010_yellow':[4.077,1.783,1.662],
    'nissan_juke_2013_black':[4.147,1.94,1.568],
    'skoda_octavia_2020_blue':[4.845,2.03,1.486],
    'suzuki_swift_2005_white':[3.653,1.679,1.506],
    'toyota_chr_2017_grey':[4.356,2.033,1.603],
    'toyota_corolla_2017_brown':[4.646,2.1,1.457],
    'toyota_prius_2012_blue': [4.386, 1.758, 1.496 ],
    'toyota_prius_2012_green':[4.386, 1.758, 1.496],
    'toyota_prius_2012_orange':[4.386, 1.758, 1.496],
    'toyota_prius_2012_red':[4.386, 1.758, 1.496],
    'toyota_prius_2012_white':[4.386, 1.758, 1.496],
    'volkswagen_tiguan_2019_blue':[4.475,2.115,1.586],
    'volvo_xc60_2018_blue':[4.85,1.95,1.805],
    'volvo_xc60_2018_blue_simplified_engine_running_thermal':[4.85,1.95,1.805],
    'pedestrian_male': [0.373,0.6,1.781],
    'pedestrian_female': [0.41,0.6,1.664],
    'pedestrian_male_thermal':[0.373,0.6,1.781],
    'pedestrian_female_thermal':[0.41,0.6,1.664],
    'bike': [1.6,0.6,1.58],
           # Add more asset paths and dimensions as needed
}

# Define the path to a new directory called "Modified" inside chosen_subdir, and create it if it
    doesn't already exist
modified_dir = os.path.join(chosen_subdir, "Modified")
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Check if the 'Extents.txt' already exists in the chosen or modified directory, if it does print
    a message and abort, otherwise process the lidar_ContributionDictionary.txt file
if os.path.exists(os.path.join(chosen_subdir, 'Extents.txt')):
    print(f"The file {os.path.join(chosen_subdir, 'Extents.txt')} already exists. Aborting...")
elif os.path.exists(os.path.join(modified_dir, 'Extents.txt')):
    print(f"The file {os.path.join(modified_dir, 'Extents.txt')} already exists. Aborting...")
else:
    instances_written = set()
    with open(os.path.join(chosen_subdir, 'lidar_ContributionDictionary.txt'), 'r') as f, open(os.
        path.join(chosen_subdir, 'Extents.txt'), 'w') as out:
        # Process each line of the 'lidar_ContributionDictionary.txt' file
        for line in f:
```

```
            data = eval(line) # Convert string to dictionary
            instance_name = data['assetDescription']['instanceName']
            # Check if instance_name is in the mapping and has not been written to output file yet
            if instance_name in instance_mapping and instance_name not in instances_written:
                asset_path = data['assetDescription']['assetPath']
                filename = os.path.basename(asset_path)
                # Find matching keys in the dimensions dictionary
                matching_keys = [key for key in dimensions if filename.startswith(key)]
                # If there are matching keys, write the instance_name and dimensions to the 'Extents
                    .txt' file
                if matching_keys:
                    key = matching_keys[0] # Assuming there's only one matching key
                    dimensions_str = '\t'.join([f'{d:.3f}' for d in dimensions[key]])
                    out.write(f'{instance_name}\t{dimensions_str}\n')
                    instances_written.add(instance_name)

    print("Extents.txt complete!")
```

A.8.5. Generate Instance Entity IDs

For a detailed understanding of its application, refer to the section 3.4.3.5.

```
import os
import numpy as np
import sys

# Define the path to the folder containing the subdirectories
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))

# Get the name of the chosen subdirectory from the command line arguments
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Define the path to a new directory called "Modified" inside chosen_subdir, and create it if it
    doesn't already exist
modified_dir = os.path.join(chosen_subdir, "Modified")
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Check if Extents.txt exists in chosen_subdir, if it does, move it to the Modified directory
extents_path = os.path.join(chosen_subdir, 'Extents.txt')
```

```
if os.path.exists(extents_path):
    os.rename(extents_path, os.path.join(modified_dir, 'Extents.txt'))
else:
    print("Extents.txt does not exist in the chosen subdirectory.")

# Read the contents of the moved Extents.txt file from the Modified directory
with open(os.path.join(modified_dir, 'Extents.txt'), 'r') as f:
    extents_lines = f.read().splitlines()

# Create a dictionary mapping each car name to a list of entity IDs from the Extents.txt file
car_entity_dict = {}
for line in extents_lines:
    tokens = line.split('\t')
    car_name = tokens[0]
    entity_id = tokens[1]
    if car_name in car_entity_dict:
        car_entity_dict[car_name].append(entity_id)
    else:
        car_entity_dict[car_name] = [entity_id]

# Read the contents of the lidar_ContributionDictionary.txt file, convert each line to a
    dictionary, and add it to a list
with open(os.path.join(chosen_subdir, 'lidar_ContributionDictionary.txt'), 'r') as f:
    file_lines = f.read().splitlines()
entity_list = []
for line in file_lines:
    entity_dict = eval(line)
    entity_list.append(entity_dict)

# Create a dictionary mapping each car instance name to a list of entity IDs from the
    lidar_ContributionDictionary.txt file
car_instance_dict = {}
for entity_dict in entity_list:
    instance_name = entity_dict['assetDescription']['instanceName']
    entity_id = str(entity_dict['EntityID'])
    if instance_name in car_entity_dict:
        if instance_name in car_instance_dict:
            car_instance_dict[instance_name].append(entity_id)
        else:
            car_instance_dict[instance_name] = [entity_id]

```

```
# Write the contents of the car_instance_dict to a new IDs.txt file in the Modified directory
txt_file_path = os.path.join(modified_dir, "IDs.txt")
with open(txt_file_path, 'w') as f:
    for car_name, entity_ids in car_instance_dict.items():
        f.write(f"{car_name}: {entity_ids}\n")

print("Conversion complete!")
```

A.8.6. Label Generation

For a detailed understanding of its application, refer to the section 3.4.3.6.

```
import pandas as pd
import numpy as np
import os
import glob
import open3d
import sys
import open3d_vis_utils as V
import visualize_utils as V_mayavi
import mayavi.mlab as mlab
import copy

def expand_or_adjust_obb(obb_test, points, object_name):
    # Get the center, rotation matrix, and extents of the original OBB
    C = np.array(obb_test.center)
    R = np.array(obb_test.R)
    S = np.array(obb_test.extent)

    # Transform points to the coordinate system of the OBB
    P_transformed = (np.linalg.inv(R) @ (points - C).T).T

    # Compute the min and max coordinates of the transformed points
    min_coords = np.min(P_transformed, axis=0)
    max_coords = np.max(P_transformed, axis=0)

    # Compute the center and size of the point cloud in the OBB coordinate system
    P_center = (min_coords + max_coords) / 2
    P_size = max_coords - min_coords

    # Flag to indicate whether the OBB was expanded
```

```
    expanded = False

    # If the object is a pedestrian or bicycle
    if 'PED' in object_name or 'BIC' in object_name:
        # If the OBB is smaller than the point cloud along any axis, expand it
        for i in range(3):
            if S[i] < P_size[i]:
                # The new size is the size of the point cloud along this axis
                S[i] = P_size[i]
                # Flag that the OBB was expanded
                expanded = True

    # If the OBB was expanded, recompute its center
    if expanded:
        C_new = (min_coords + max_coords) / 2
        C = (R @ C_new) + C
    # If the OBB wasn't expanded but the object is a pedestrian or bicycle, adjust its center in
        the xy-plane
    else:
        if 'PED' in object_name or 'BIC' in object_name:
            C_new_ped = (R @ np.array([P_center[0], P_center[1], 0])) + C
            C = np.array([C_new_ped[0], C_new_ped[1], obb_test.center[2]])

    # Update the original OBB with new center and extents
    obb_test.center = C
    obb_test.extent = S

    # Return the updated OBB
    return obb_test



def generate_points_in_obb(obb_fnc, num_points):
    """
    Generate a set of random points within a specified Oriented Bounding Box (OBB).

    :param obb_fnc: The Oriented Bounding Box
    :param num_points: Number of points to generate within the OBB
    :return translated_points: The generated points
    """
    # Generate random points directly within the OBB, considering its extent
```

```
    points = np.random.uniform(-0.5 * obb_fnc.extent, 0.5 * obb_fnc.extent, size=(num_points, 3))

    # Transform the points to align with the orientation of the OBB
    transformed_points = np.dot(points, obb_fnc.R.T)

    # Translate the points to match the center of the OBB
    translated_points = transformed_points + obb_fnc.center[np.newaxis, :]

    return translated_points



def lidar_to_rect(pts_lidar,V2C_array,R0_array):
    """
    Transforms points from LiDAR coordinates to rectified coordinates

    :param pts_lidar: points in LiDAR coordinates (N, 3)
    :return pts_rect: points in rectified coordinates (N, 3)
    """
    pts_lidar_hom = cart_to_hom(pts_lidar) # Convert to homogeneous coordinates
    # Perform coordinate transformation
    pts_rect = np.dot(pts_lidar_hom, np.dot(V2C_array.T, R0_array.T))
    return pts_rect

def cart_to_hom(pts):
    """
    Converts points from Cartesian coordinates to homogeneous coordinates

    :param pts: points in Cartesian coordinates (N, 3 or 2)
    :return pts_hom: points in homogeneous coordinates (N, 4 or 3)
    """
    pts_hom = np.hstack((pts, np.ones((pts.shape[0], 1), dtype=np.float32))) # Append ones along
        the second axis
    return pts_hom

def get_fov_flag(pts_rect, img_shape, P2_array):
    """
    Check if the rectified points fall within the image boundaries

    Args:
       pts_rect: points in rectified coordinates
```

```
        img_shape: shape of the image
        P2_array: Camera matrix

    Returns:
        pts_valid_flag: Boolean array indicating the validity of each point
    """
    pts_img, pts_rect_depth = rect_to_img(pts_rect, P2_array) # Project points onto image plane
    # Check if the points fall within the image boundaries
    val_flag_1 = np.logical_and(pts_img[:, 0] >= 0, pts_img[:, 0] < img_shape[1])
    val_flag_2 = np.logical_and(pts_img[:, 1] >= 0, pts_img[:, 1] < img_shape[0])
    val_flag_merge = np.logical_and(val_flag_1, val_flag_2)
    # Check if the depth of the points is non-negative
    pts_valid_flag = np.logical_and(val_flag_merge, pts_rect_depth >= 0)
    return pts_valid_flag

def rect_to_img(pts_rect, P2_array):
    """
    Project points from rectified coordinates onto an image plane

    :param pts_rect: points in rectified coordinates (N, 3)
    :return pts_img: projected points in image coordinates (N, 2)
    """
    pts_rect_hom = cart_to_hom(pts_rect) # Convert to homogeneous coordinates
    pts_2d_hom = np.dot(pts_rect_hom, P2_array.T) # Project points onto image plane
    pts_img = (pts_2d_hom[:, 0:2].T / pts_rect_hom[:, 2]).T # Normalize by depth
    pts_rect_depth = pts_2d_hom[:, 2] - P2_array.T[3, 2] # Calculate depth in rectified camera
        coordinates
    return pts_img, pts_rect_depth

def filter_point_cloud(point_cloud, range_values):
    """
    Filters the point cloud based on a provided range along the x, y, and z axis

    :param point_cloud: The point cloud to be filtered (N, 3)
    :param range_values: A list of range values [x_min, y_min, z_min, x_max, y_max, z_max]
    :return flag_array: Boolean array indicating whether each point falls within the specified
        range
    """
    x_min, y_min, z_min, x_max, y_max, z_max = range_values
    x, y, z = point_cloud[:, 0], point_cloud[:, 1], point_cloud[:, 2]

```

```
    # Create boolean array indicating whether each point falls within the specified range
    flag_array = np.logical_and.reduce((x >= x_min, x <= x_max, y >= y_min, y <= y_max, z >= z_min,
         z <= z_max))

    return flag_array



def get_optimal_oriented_bounding_box(car_df,extents, object_name,pcd_fov):
    """
    Compute the optimal Oriented Bounding Box for an object given the object's point cloud and
        extents.
    The optimal OBB is the one that contains the maximum number of points from the point cloud's
        FOV.

    :param car_df: DataFrame containing the object's point cloud
    :param extents: Dictionary mapping each object name to its extents
    :param object_name: Name of the object
    :param pcd_fov: Point cloud of the FOV
    :return best_obb: The optimal OBB
    :return az_selected: The azimuth corresponding to the optimal OBB
    """
    x_values = car_df.values[:,0]
    y_values = car_df.values[:,1]
    z_values = car_df.values[:,2]
    az = car_df.values[:,5] # Azimuths of the points

    extent = extents[object_name] # Extents of the object
    distance = extent[0]/2 # Half the length of the object
    x_values_moved = x_values + np.cos(az)*distance # x-coordinates of the OBB centers
    y_values_moved = y_values + np.sin(az)*distance # y-coordinates of the OBB centers

    max_points_inside = 0 # Maximum number of points inside an OBB
    best_obb = None # The optimal OBB
    az_selected = None # Azimuth corresponding to the optimal OBB
    prev_distance = float('inf') # Set an initial value for prev_distance
    for i in range((len(x_values))):
        center = [x_values_moved[i], y_values_moved[i], z_values[i]] # Center of the OBB
        lwh = extent # Length, width, and height of the OBB
        axis_angles = np.array([0, 0, az[i] + 1e-10]) # Orientation of the OBB
        rot = open3d.geometry.get_rotation_matrix_from_axis_angle(axis_angles) # Rotation matrix
```

```
            for the OBB
        obb_10 = open3d.geometry.OrientedBoundingBox(center, rot, lwh) # Construct the OBB
        distance_to_origin = np.sqrt(center[0]**2 + center[1]**2 ) # Distance from the OBB center
            to the origin
        points_inside = obb_10.get_point_indices_within_bounding_box(pcd_fov.points) # Indices of
            points from the FOV that fall within the OBB
        num_points_inside = len(points_inside) # Number of points inside the OBB
        if num_points_inside > max_points_inside: # If the OBB contains more points than the
            current optimal OBB
            max_points_inside = num_points_inside
            best_obb = obb_10
            az_selected = az[i]
        elif num_points_inside == max_points_inside: # If the OBB contains the same number of
            points as the current optimal OBB
            is_moving_towards_origin = distance_to_origin < prev_distance # Check if the OBB center
                is closer to the origin than the current optimal OBB
            if is_moving_towards_origin: # If it is, update the optimal OBB
                max_points_inside = num_points_inside
                best_obb = obb_10
                az_selected = az[i]

        prev_distance = distance_to_origin # Update the distance to the origin

    return best_obb, az_selected


def calculate_truncation_ratio(obb_expanded, num_points,V2C_array,R0_array,image_shape, P2_array):
    """
    Calculate the truncation ratio for an expanded Oriented Bounding Box.

    :param obb_expanded: The expanded Oriented Bounding Box
    :param num_points: Number of points to generate within the OBB
    :param V2C_array: Array for transformation from Velodyne coordinates to camera coordinates
    :param R0_array: Rectification matrix
    :param image_shape: Shape of the image (height, width)
    :param P2_array: Projection matrix
    :return truncation: Truncation ratio
    """
    hypothetical_point_cloud = generate_points_in_obb(obb_expanded, num_points) # Generate a
        hypothetical point cloud within the OBB
    hyp_pts_rect = lidar_to_rect(hypothetical_point_cloud[:, 0:3],V2C_array,R0_array) # Transform
```

```
        the point cloud to rectified camera coordinates
    hyp_fov_flag = get_fov_flag(hyp_pts_rect, image_shape, P2_array) # Determine which points fall
         within the FOV
    hyp_filtered_points = hypothetical_point_cloud[hyp_fov_flag] # Filter the points that fall
        within the FOV
    truncation = 1 - hyp_filtered_points.shape[0]/hypothetical_point_cloud.shape[0] # Calculate
        the truncation ratio

    return truncation



def generate_kitti_label(best_bbox, P2_array, R0_array, V2C_array, image_shape, truncation, Type):
    """
    Generate a label string in the KITTI dataset format for a given Oriented Bounding Box (OBB).

    :param best_bbox: The best bounding box (OBB)
    :param P2_array: Projection matrix
    :param R0_array: Rectification matrix
    :param V2C_array: Transformation from Velodyne coordinates to camera coordinates
    :param image_shape: Shape of the image (height, width)
    :param truncation: Truncation value of the object
    :param Type: The type of the object
    :return line: The generated label string
    """
    # Some lengthy code to transform the OBB to camera coordinates and to compute the 2D and 3D
        bounding boxes in image and camera coordinates
    best_bbox = np.array(best_bbox)
    boxes3d_lidar_copy = copy.deepcopy(best_bbox)
    boxes3d_lidar_copy = np.array(boxes3d_lidar_copy)
    xyz_lidar = boxes3d_lidar_copy[:, 0:3]
    l, w, h = boxes3d_lidar_copy[:, 3:4], boxes3d_lidar_copy[:, 4:5], boxes3d_lidar_copy[:, 5:6]
    r = boxes3d_lidar_copy[:, 6:7]
    xyz_lidar[:, 2] -= h.reshape(-1) / 2
    pts_lidar_hom = cart_to_hom(xyz_lidar)
    xyz_cam = np.dot(pts_lidar_hom, np.dot(V2C_array.T, R0_array.T))
    r = -r - np.pi / 2
    best_boxes_camera = np.concatenate([xyz_cam, l, h, w, r], axis=-1)
    boxes3d = best_boxes_camera
    bottom_center = True
    boxes_num = boxes3d.shape[0]
```

```
    l, h, w = boxes3d[:, 3], boxes3d[:, 4], boxes3d[:, 5]
    x_corners = np.array([l / 2., l / 2., -l / 2., -l / 2., l / 2., l / 2., -l / 2., -l / 2], dtype=np.float32).T
    z_corners = np.array([w / 2., -w / 2., -w / 2., w / 2., w / 2., -w / 2., -w / 2., w / 2.], dtype=np.float32).T
    if bottom_center:
        y_corners = np.zeros((boxes_num, 8), dtype=np.float32)
        y_corners[:, 4:8] = -h.reshape(boxes_num, 1).repeat(4, axis=1) # (N, 8)
    else:
        y_corners = np.array([h / 2., h / 2., h / 2., h / 2., -h / 2., -h / 2., -h / 2., -h / 2.], dtype=np.float32).T

    ry = boxes3d[:, 6]
    zeros, ones = np.zeros(ry.size, dtype=np.float32), np.ones(ry.size, dtype=np.float32)
    rot_list = np.array([[np.cos(ry), zeros, -np.sin(ry)],
                         [zeros, ones, zeros],
                         [np.sin(ry), zeros, np.cos(ry)]]) # (3, 3, N)
    R_list = np.transpose(rot_list, (2, 0, 1)) # (N, 3, 3)

    temp_corners = np.concatenate((x_corners.reshape(-1, 8, 1), y_corners.reshape(-1, 8, 1),
                                   z_corners.reshape(-1, 8, 1)), axis=2) # (N, 8, 3)
    rotated_corners = np.matmul(temp_corners, R_list) # (N, 8, 3)
    x_corners, y_corners, z_corners = rotated_corners[:, :, 0], rotated_corners[:, :, 1], rotated_corners[:, :, 2]

    x_loc, y_loc, z_loc = boxes3d[:, 0], boxes3d[:, 1], boxes3d[:, 2]

    x = x_loc.reshape(-1, 1) + x_corners.reshape(-1, 8)
    y = y_loc.reshape(-1, 1) + y_corners.reshape(-1, 8)
    z = z_loc.reshape(-1, 1) + z_corners.reshape(-1, 8)

    corners = np.concatenate((x.reshape(-1, 8, 1), y.reshape(-1, 8, 1), z.reshape(-1, 8, 1)), axis=2)
    corners3d =corners.astype(np.float32)

    pts_rect_hom = cart_to_hom(corners3d.reshape(-1, 3))
    pts_2d_hom = np.dot(pts_rect_hom, P2_array.T)
    pts_img = (pts_2d_hom[:, 0:2].T / pts_rect_hom[:, 2]).T # (N, 2)
    corners_in_image = pts_img.reshape(-1, 8, 2)
    min_uv = np.min(corners_in_image, axis=1) # (N, 2)
    max_uv = np.max(corners_in_image, axis=1) # (N, 2)
```

```
    boxes2d_image = np.concatenate([min_uv, max_uv], axis=1)
    if image_shape is not None:
        boxes2d_image[:, 0] = np.clip(boxes2d_image[:, 0], a_min=0, a_max=image_shape[1] - 1)
        boxes2d_image[:, 1] = np.clip(boxes2d_image[:, 1], a_min=0, a_max=image_shape[0] - 1)
        boxes2d_image[:, 2] = np.clip(boxes2d_image[:, 2], a_min=0, a_max=image_shape[1] - 1)
        boxes2d_image[:, 3] = np.clip(boxes2d_image[:, 3], a_min=0, a_max=image_shape[0] - 1)

    best_boxes_img=boxes2d_image
    # Compute alpha, the rotation angle in image plane
    alpha = -np.arctan2(-best_bbox[:, 1], best_bbox[:, 0]) + best_boxes_camera[:, 6]

    # Extract relevant values for the KITTI label
    bbox = best_boxes_img
    dimensions = best_boxes_camera[:, 3:6]
    location = best_boxes_camera[:, 0:3]
    rotation = best_boxes_camera[:, 6]
    truncated = truncation
    occluded = np.full((best_bbox.shape[0], 1), 0) # Here, it assumes that the object is not
        occluded

    # Generate the KITTI label string
    line = f"{Type} {truncated:.2f} {int(occluded)} {alpha[0]} " \
        f"{bbox[0][0]} {bbox[0][1]} {bbox[0][2]} {bbox[0][3]} " \
        f"{dimensions[0][1]} {dimensions[0][2]} {dimensions[0][0]} " \
        f"{location[0][0]} {location[0][1]} {location[0][2]} {rotation[0]}\n"

    return line


def process_pointcloud_data(modified_dir,file,P2_array, R0_array,V2C_array,image_shape,extents,df,
    time_carmaker):
    # Initializing red_flag which might be used to signal errors or exceptions during execution
    red_flag = False
    point_cloud_file = file

    # Extracting the timestamp from the file name
    time_ms = int(point_cloud_file.split('_')[1])
    time_s = time_ms / 1000

    # Generating corresponding label and contributions file names
    label_file = file.replace("pointcloud", "label").replace(".npy", ".txt")
```

```
332     contributions_file = file.replace("pointcloud.npy", "contributions.txt")
333
334     # Fetching the row in the dataframe that corresponds to the current timestamp
335     index = time_carmaker[(time_carmaker.iloc[:,0] <= time_s) & (time_carmaker.iloc[:,0] >= (
            time_s-0.1)) ].index
336     row = df.iloc[index]
337
338      # Loading point cloud data
339     points = np.load(os.path.join(modified_dir, point_cloud_file))
340
341     # Loading and processing file contributions
342     with open(os.path.join(chosen_subdir, contributions_file), 'r') as file_2:
343         file_contents = file_2.read()
344
345     # filter out empty lines in the contributions file and removes the corresponding points in the
            point cloud.
346     my_lines = file_contents.splitlines()
347     empty_indexes = np.where(np.array(my_lines) == '')[0]
348     my_lines = np.delete(my_lines, empty_indexes)
349     points = np.delete(points, empty_indexes, axis=0)
350     points = np.array(points)
351
352     # Transforming the point cloud from lidar coordinates to rectified camera coordinates
353     pts_rect = lidar_to_rect(points[:, 0:3],V2C_array,R0_array)
354     # Getting the flags that represent whether each point is in the field of view or not
355     fov_flag = get_fov_flag(pts_rect, image_shape, P2_array)
356
357     # Filtering points and lines based on the field of view and point cloud range
358     filtered_points = points[fov_flag]
359     flag_array = filter_point_cloud(filtered_points, POINT_CLOUD_RANGE)
360     filt_filt_points = filtered_points[flag_array]
361
362     # Note: The same filtering process is done to the lines as well
363     filtered_lines = list(filter(lambda x: x[1], zip(my_lines, fov_flag)))
364     filtered_lines = [line for line, flag in filtered_lines]
365     filt_filt_lines = list(filter(lambda x: x[1], zip(filtered_lines, flag_array)))
366     filt_filt_lines = [line for line, flag in filt_filt_lines]
367
368     # Creating dictionaries to hold point cloud data and lines grouped by objects
369     indices_by_object = {}
370     indices_by_object_raw = {}
```

```
371
372
373     for obj, obj_values in objects.items():
374         obj_values_set = set(obj_values)
375         obj_indices = np.array([index for index, line in enumerate(filt_filt_lines) if int(line.
                split()[0]) in obj_values_set])
376         obj_indices_raw = np.array([index for index, line in enumerate(my_lines) if int(line.split
                ()[0]) in obj_values_set])
377         indices_by_object[obj] = obj_indices
378         indices_by_object_raw[obj] = obj_indices_raw
379
380     points_by_object = {obj: [filt_filt_points[index] for index in indices] for obj, indices in
            indices_by_object.items()}
381
382     points_by_object_raw = {obj: [points[index] for index in indices] for obj, indices in
            indices_by_object_raw.items()}
383     # Note: The same process is applied to the raw lines as well
384
385     lines = []
386     bboxes= []
387
388     # Looping over all objects
389     for object_name, points_of_object in points_by_object.items():
390         best_bbox = None
391         # Process the point cloud data for each object
392         if points_of_object:
393
394             # Generate optimal bounding boxes for each object using Open3D and custom functions
395             my_pointcloud_for_test = points_by_object[object_name]
396             my_pointcloud_for_test_raw = points_by_object_raw[object_name]
397             object_point_cloud_raw = np.array(my_pointcloud_for_test_raw)[:, :3]
398             pcd_raw = open3d.geometry.PointCloud()
399             pcd_raw.points = open3d.utility.Vector3dVector(object_point_cloud_raw)
400             object_point_cloud_fov = np.array(my_pointcloud_for_test)[:, :3]
401             pcd_fov = open3d.geometry.PointCloud()
402             pcd_fov.points = open3d.utility.Vector3dVector(object_point_cloud_fov)
403             car_df = row.filter(like=object_name)
404             best_obb, az_selected = get_optimal_oriented_bounding_box(car_df,extents,object_name,
                    pcd_fov)
405
406
```

```
407             obb =best_obb
408
409             if best_obb is not None:
410                 points_inside_obb = obb.get_point_indices_within_bounding_box(pcd_fov.points)
411                 num_points_inside_obb = len(points_inside_obb)
412                 if num_points_inside_obb != object_point_cloud_fov.shape[0]:
413                     obb_expanded = expand_or_adjust_obb(obb, np.array(my_pointcloud_for_test)[:,
                            :3],object_name )
414                 else:
415                     if 'PED' in object_name or 'BIC' in object_name:
416                         obb_expanded = expand_or_adjust_obb(obb, np.array(my_pointcloud_for_test)[:,
                                :3],object_name )
417                     else:
418                         obb_expanded = obb
419
420
421                 best_bbox = np.array([obb_expanded.center[0], obb_expanded.center[1], obb_expanded.
                        center[2], obb_expanded.extent[0], obb_expanded.extent[1], obb_expanded.extent
                        [2], az_selected])
422                 best_bbox = np.reshape(best_bbox, (1, -1))
423
424                 points_inside_raw = obb_expanded.get_point_indices_within_bounding_box(pcd_raw.
                        points)
425                 num_points_inside_raw = len(points_inside_raw)
426
427
428
429                 points_inside_fov = obb_expanded.get_point_indices_within_bounding_box(pcd_fov.
                        points)
430                 num_points_inside_fov = len(points_inside_fov)
431
432                 # Calculate the truncation ratio
433                 if num_points_inside_fov != num_points_inside_raw:
434                     truncation = calculate_truncation_ratio(obb_expanded, 1000,V2C_array,R0_array,
                            image_shape, P2_array)
435                 else:
436                     truncation = 0
437
438                 Type = None
439                 if 'CAR' in object_name:
440                     Type = 'Car'
```

```
441                 elif 'PED' in object_name:
442                     Type = 'Pedestrian'
443                 elif 'BIC' in object_name:
444                     Type = 'Cyclist'
445                 # Adjust the height of the bounding boxes based on the object type
446                 if 'CAR' in object_name:
447                     best_bbox[0][2] = best_bbox[0][2] - 0.08
448                 elif 'PED' in object_name:
449                     best_bbox[0][2] = best_bbox[0][2] + 0.02
450                 elif 'BIC' in object_name:
451                     best_bbox[0][2] = best_bbox[0][2] + 0.02
452             else:
453                 print(f"No valid OBB found for object: {object_name}")
454                 print(point_cloud_file)
455                 # Note: If an optimal bounding box can't be generated for an object, set the
                        red_flag to True
456                 red_flag = True
457 
458         if best_bbox is not None:
459             if all(x is not None for x in best_bbox):
460                 # Generate labels in KITTI format for each object
461                 line = generate_kitti_label( best_bbox, P2_array, R0_array, V2C_array, image_shape,
                        truncation, Type)
462                 lines.append(line)
463                 bboxes.append(best_bbox)
464             else:
465                 line = ''
466                 lines.append(line)
467                 bboxes.append(best_bbox)
468         else:
469             line = ''
470             lines.append(line)
471             bboxes.append(best_bbox)
472 
473     return filt_filt_points, label_file, point_cloud_file, lines, red_flag, bboxes
474 
475 
476 def save_labels(modified_dir,label_file,lines):
477     # Function to save labels into files
478     with open(os.path.join(modified_dir, label_file), "w") as f:
479         for line in lines:
```

```
            f.write(line)

# These are constants used for processing the lidar data and images
POINT_CLOUD_RANGE= [0, -39.68, -3, 69.12, 39.68, 1]
P2_array = np.array([[7.21537720e+02, 0.00000000e+00, 6.09559326e+02, 4.48572807e+01],
[0.00000000e+00, 7.21537720e+02, 1.72854004e+02, 2.16379106e-01],
[0.00000000e+00, 0.00000000e+00, 1.00000000e+00, 2.74588400e-03]], dtype=np.float32)
R0_array = np.array([[ 0.9999239 , 0.00983776, -0.00744505],
[-0.0098698 , 0.9999421 , -0.00427846],
[ 0.00740253, 0.00435161, 0.9999631 ]], dtype=np.float32)
V2C_array = np.array([[ 7.53374491e-03, -9.99971390e-01, -6.16602018e-04,-4.06976603e-03],
[ 1.48024904e-02, 7.28073297e-04, -9.99890208e-01, -7.63161778e-02],
[ 9.99862075e-01, 7.52379000e-03, 1.48075502e-02, -2.71780610e-01]], dtype=np.float32)
image_shape = np.array([375, 1242],dtype=np.int32)

# Getting the directory path
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Creating the modified directory if it doesn't exist
modified_dir = os.path.join(chosen_subdir, "Modified")
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Creating the output directory if it doesn't exist
out_dir = os.path.join(modified_dir, "Out")
if not os.path.exists(out_dir):
    os.mkdir(out_dir)

# Reading the 'IDs.txt' file
with open(os.path.join(modified_dir, 'IDs.txt'), 'r') as f:
    data = f.readlines()

# Creating a dictionary of objects with their corresponding numbers
objects = {}
for line in data:
    entity, numbers_str = line.strip().split(": ")
    numbers = [int(n.strip("'[]")) for n in numbers_str.split(", ")]
    objects[entity] = numbers

```

```
# Reading the 'Extents.txt' file
with open(os.path.join(modified_dir, 'Extents.txt'), 'r') as f:
    extents = {}
    for line in f:
        fields = line.strip().split()
        entity = fields[0]
        extent_here = [float(x) for x in fields[1:]]
        extents[entity] = extent_here

# Looking for a single .dat file in the chosen directory
dat_files = glob.glob(os.path.join(chosen_subdir, "*.dat"))
if len(dat_files) != 1:
    raise ValueError(f"Expected 1 .dat file in {chosen_subdir}, found {len(dat_files)}")
file_name = dat_files[0]

# Reading the .dat file
length_df=len(extents)*6 +1
df = pd.read_csv(file_name, sep='\t', skiprows=2, usecols=range(1, length_df))
time_carmaker = pd.read_csv(file_name, sep='\t', skiprows=2, usecols=[length_df])

# Naming the columns of the DataFrame
with open(file_name, 'r') as f:
    column_names = f.readline().strip().split('\t')[1:length_df]
df.columns = column_names

# Getting all the .npy files that start with 'lidar' in the directory
files = [file for file in os.listdir(modified_dir) if file.startswith("lidar") and file.endswith("
    pointcloud.npy")]

# Processing each file one by one
for file in files:
    filt_filt_points, label_file, point_cloud_file, line, red_flag, bboxes =
        process_pointcloud_data(modified_dir,file,P2_array, R0_array,V2C_array,image_shape,extents
        ,df,time_carmaker)
    if red_flag:
        save_labels(modified_dir,label_file,'')
    else:
        filtered_bboxes = [bbox for bbox in bboxes if bbox is not None]
        save_labels(modified_dir,label_file,line)
```

A.8.7. Prepare LiDAR Data

For a detailed understanding of its application, refer to the section 3.5.

```
import numpy as np # For numerical operations (not used in the script)
import os # For operating system dependent functionalities
import shutil # For high-level file operations
import sys # To use command line arguments

# Get the absolute path to the parent directory of this script file
folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
# Get the name of the subdirectory from command line arguments
subdir_name = os.path.basename(os.path.normpath(sys.argv[1]))
# Join the subdirectory name to the parent directory path
chosen_subdir = os.path.join(folder_path, subdir_name, "lidar")

# Create a new directory called "Modified" inside chosen_subdir, if it doesn't already exist
modified_dir = os.path.join(chosen_subdir, "Modified")
if not os.path.exists(modified_dir):
    os.mkdir(modified_dir)

# Define the path to the new directory inside "Modified"
ready_dir = os.path.join(modified_dir, "Ready")

# If 'Ready' directory already exists, delete its contents
if os.path.exists(ready_dir):
    shutil.rmtree(ready_dir)

# Create the 'Ready' directory
os.mkdir(ready_dir)

# Initialize a counter variable
counter = 0

# Loop through the files in the 'Modified' directory
for file in os.listdir(modified_dir):
    # Check if the file is a Lidar point cloud file
    if file.startswith("lidar") and file.endswith("pointcloud.npy"):
        # Store names of associated 'contributions.txt' and 'label.txt' files
        point_cloud_file = file
        label_file = file.replace("pointcloud", "label").replace(".npy", ".txt")
        label_file_path = os.path.join(modified_dir, label_file)
```

```
39
40        # Check if the label file exists and is not empty
41        if os.path.exists(label_file_path) and os.stat(label_file_path).st_size != 0:
42            # Generate new base name for output files
43            output_file_base = str(counter).zfill(6)
44
45            # Copy the point cloud and label files to the 'Ready' directory with new names
46            shutil.copyfile(os.path.join(modified_dir, point_cloud_file), os.path.join(ready_dir,
                  output_file_base + ".npy"))
47            shutil.copyfile(label_file_path, os.path.join(ready_dir, output_file_base + ".txt"))
48
49            # Increment the counter variable
50            counter += 1
```

A.8.8. Run Sequence Scripts

For a detailed understanding of its application, refer to the section 3.5.

```
1  import os # For operating system dependent functionalities
2  import subprocess # To run shell commands
3  import sys # To use command line arguments
4
5  # Get the directory paths to be processed.
6  # Could be provided as command line argument, but in this case,
7  # it's hardcoded in the script itself as an example.
8  # directory_paths = sys.argv[1:] # Uncomment to get paths from command line
9  directory_paths = ['...'] # Define your directories here
10
11 # Iterate through each directory path provided
12 for directory_path in directory_paths:
13     # Compute the absolute path of the parent directory of this script file
14     folder_path = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
15
16     # Check if the directory exists, exit the script if not
17     if not os.path.exists(os.path.join(folder_path, directory_path)):
18         print("Directory␣does␣not␣exist:␣", directory_path)
19         sys.exit(1)
20
21     # List of scripts to be run sequentially
22     script_names = ['IdenticalPointCloudDeletion.py', 'PointCloudTxtToNpyConverter.py',
23                     'JsonToTxtConverter.py', 'ExtractInstanceDimensions.py',
```

```
                'GenerateInstanceEntityIDMapping.py', 'LidarPointCloudLabelGenerator.py',
                'PrepareLidarData.py']

    # Loop over the script names and run each one using subprocess.run()
    for script_name in script_names:
        # Run the script using subprocess.run and capture the output
        result = subprocess.run(['python', script_name, directory_path], capture_output=True, text=
            True)

        # Print the output of the script
        print(f"Output of {script_name}:")
        print(result.stdout) # Standard output (stdout)
        print(result.stderr) # Standard error (stderr), if any
```

A.8.9. Collect Data

For a detailed understanding of its application, refer to the section 3.5.

```
import os
import shutil
import random

# List of source folders containing point cloud and label data
folders = ['...'] # Replace this with the actual folders

# Base directory relative to this script
src_dir = '..'

# Destination directories for labels and point cloud data
dst_labels_dir = os.path.join(src_dir, 'AVX_DATA\Version_7\Test\labels')
dst_points_dir = os.path.join(src_dir, 'AVX_DATA\Version_7\Test\points')

# Create the destination directories if they don't exist
if not os.path.exists(dst_labels_dir):
    os.mkdir(dst_labels_dir)

if not os.path.exists(dst_points_dir):
    os.mkdir(dst_points_dir)

# Counter to generate unique filenames
j = 0
```

```
24
25 # Iterate over each source folder
26 for folder in folders:
27     # Define the path to the 'Ready' subfolder containing the data
28     ready_dir = os.path.join(src_dir, folder, 'lidar', 'Modified', 'Ready')
29
30     # Get lists of point cloud and label files
31     point_cloud_files = [file for file in os.listdir(ready_dir) if file.endswith('.npy')]
32     label_files = [file for file in os.listdir(ready_dir) if file.endswith('.txt')]
33
34     # Randomize the order of point cloud files
35     random.shuffle(point_cloud_files)
36
37     # For each point cloud file
38     for point_cloud_file in point_cloud_files:
39         # Define source paths for the point cloud file and its corresponding label file
40         src_point_cloud_path = os.path.join(ready_dir, point_cloud_file)
41         src_label_path = os.path.join(ready_dir, os.path.splitext(point_cloud_file)[0] + '.txt')
42
43         # Generate new unique filenames
44         j_str = '{:06d}'.format(j)
45         new_point_cloud_filename = j_str + '.npy'
46         new_label_filename = j_str + '.txt'
47         j += 1 # Increment counter
48
49         # Define destination paths
50         dst_point_cloud_path = os.path.join(dst_points_dir, new_point_cloud_filename)
51         dst_label_path = os.path.join(dst_labels_dir, new_label_filename)
52
53         # Copy point cloud file and label file to the destination directories
54         shutil.copyfile(src_point_cloud_path, dst_point_cloud_path)
55         shutil.copyfile(src_label_path, dst_label_path)
```

A.8.10. Visualization Utilities

The provided script, drawn from [58], serves as a valuable tool for visualizing generated point clouds. Beyond mere visualization, it offers the capability to display ground truth boxes as well as detections.

```python
# This Python script is sourced from https://github.com/open-mmlab/OpenPCDet/blob/master/tools/visual_utils/visualize_utils.py
import mayavi.mlab as mlab
import numpy as np
import torch

box_colormap = [
    [1, 1, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 1, 0],
]


def check_numpy_to_torch(x):
    if isinstance(x, np.ndarray):
        return torch.from_numpy(x).float(), True
    return x, False


def rotate_points_along_z(points, angle):
    """
    Args:
        points: (B, N, 3 + C)
        angle: (B), angle along z-axis, angle increases x ==> y
    Returns:

    """
    points, is_numpy = check_numpy_to_torch(points)
    angle, _ = check_numpy_to_torch(angle)

    cosa = torch.cos(angle)
    sina = torch.sin(angle)
    zeros = angle.new_zeros(points.shape[0])
    ones = angle.new_ones(points.shape[0])
    rot_matrix = torch.stack((
        cosa, sina, zeros,
        -sina, cosa, zeros,
        zeros, zeros, ones
    ), dim=1).view(-1, 3, 3).float()
    points_rot = torch.matmul(points[:, :, 0:3], rot_matrix)
```

```
    points_rot = torch.cat((points_rot, points[:, :, 3:]), dim=-1)
    return points_rot.numpy() if is_numpy else points_rot


def boxes_to_corners_3d(boxes3d):
    """
        7 -------- 4
       /| /|
      6 -------- 5 .
      | | | |
      . 3 -------- 0
      |/ |/
      2 -------- 1
    Args:
        boxes3d: (N, 7) [x, y, z, dx, dy, dz, heading], (x, y, z) is the box center

    Returns:
    """
    boxes3d, is_numpy = check_numpy_to_torch(boxes3d)

    template = boxes3d.new_tensor((
        [1, 1, -1], [1, -1, -1], [-1, -1, -1], [-1, 1, -1],
        [1, 1, 1], [1, -1, 1], [-1, -1, 1], [-1, 1, 1],
    )) / 2

    corners3d = boxes3d[:, None, 3:6].repeat(1, 8, 1) * template[None, :, :]
    corners3d = rotate_points_along_z(corners3d.view(-1, 8, 3), boxes3d[:, 6]).view(-1, 8, 3)
    corners3d += boxes3d[:, None, 0:3]

    return corners3d.numpy() if is_numpy else corners3d


def visualize_pts(pts, fig=None, bgcolor=(0, 0, 0), fgcolor=(1.0, 1.0, 1.0),
                  show_intensity=False, size=(600, 600), draw_origin=True):
    if not isinstance(pts, np.ndarray):
        pts = pts.cpu().numpy()
    if fig is None:
        fig = mlab.figure(figure=None, bgcolor=bgcolor, fgcolor=fgcolor, engine=None, size=size)

    if show_intensity:
        G = mlab.points3d(pts[:, 0], pts[:, 1], pts[:, 2], pts[:, 3], mode='point',
```

```
                        colormap='gnuplot', scale_factor=1, figure=fig)
    else:
        G = mlab.points3d(pts[:, 0], pts[:, 1], pts[:, 2], mode='point',
                        colormap='gnuplot', scale_factor=1, figure=fig)
    if draw_origin:
        mlab.points3d(0, 0, 0, color=(1, 1, 1), mode='cube', scale_factor=0.2)
        mlab.plot3d([0, 3], [0, 0], [0, 0], color=(0, 0, 1), tube_radius=0.1)
        mlab.plot3d([0, 0], [0, 3], [0, 0], color=(0, 1, 0), tube_radius=0.1)
        mlab.plot3d([0, 0], [0, 0], [0, 3], color=(1, 0, 0), tube_radius=0.1)

    return fig


def draw_sphere_pts(pts, color=(0, 1, 0), fig=None, bgcolor=(0, 0, 0), scale_factor=0.2):
    if not isinstance(pts, np.ndarray):
        pts = pts.cpu().numpy()

    if fig is None:
        fig = mlab.figure(figure=None, bgcolor=bgcolor, fgcolor=None, engine=None, size=(600, 600))

    if isinstance(color, np.ndarray) and color.shape[0] == 1:
        color = color[0]
        color = (color[0] / 255.0, color[1] / 255.0, color[2] / 255.0)

    if isinstance(color, np.ndarray):
        pts_color = np.zeros((pts.__len__(), 4), dtype=np.uint8)
        pts_color[:, 0:3] = color
        pts_color[:, 3] = 255
        G = mlab.points3d(pts[:, 0], pts[:, 1], pts[:, 2], np.arange(0, pts_color.__len__()), mode=
            'sphere',
                        scale_factor=scale_factor, figure=fig)
        G.glyph.color_mode = 'color_by_scalar'
        G.glyph.scale_mode = 'scale_by_vector'
        G.module_manager.scalar_lut_manager.lut.table = pts_color
    else:
        mlab.points3d(pts[:, 0], pts[:, 1], pts[:, 2], mode='sphere', color=color,
                    colormap='gnuplot', scale_factor=scale_factor, figure=fig)

    mlab.points3d(0, 0, 0, color=(1, 1, 1), mode='cube', scale_factor=0.2)
    mlab.plot3d([0, 3], [0, 0], [0, 0], color=(0, 0, 1), line_width=3, tube_radius=None, figure=
        fig)
```

```
121     mlab.plot3d([0, 0], [0, 3], [0, 0], color=(0, 1, 0), line_width=3, tube_radius=None, figure=
            fig)
122     mlab.plot3d([0, 0], [0, 0], [0, 3], color=(1, 0, 0), line_width=3, tube_radius=None, figure=
            fig)
123 
124     return fig
125 
126 
127 def draw_grid(x1, y1, x2, y2, fig, tube_radius=None, color=(0.5, 0.5, 0.5)):
128     mlab.plot3d([x1, x1], [y1, y2], [0, 0], color=color, tube_radius=tube_radius, line_width=1,
            figure=fig)
129     mlab.plot3d([x2, x2], [y1, y2], [0, 0], color=color, tube_radius=tube_radius, line_width=1,
            figure=fig)
130     mlab.plot3d([x1, x2], [y1, y1], [0, 0], color=color, tube_radius=tube_radius, line_width=1,
            figure=fig)
131     mlab.plot3d([x1, x2], [y2, y2], [0, 0], color=color, tube_radius=tube_radius, line_width=1,
            figure=fig)
132     return fig
133 
134 
135 def draw_multi_grid_range(fig, grid_size=20, bv_range=(-60, -60, 60, 60)):
136     for x in range(bv_range[0], bv_range[2], grid_size):
137         for y in range(bv_range[1], bv_range[3], grid_size):
138             fig = draw_grid(x, y, x + grid_size, y + grid_size, fig)
139 
140     return fig
141 
142 
143 def draw_scenes(points, gt_boxes=None, ref_boxes=None, ref_scores=None, ref_labels=None):
144     if not isinstance(points, np.ndarray):
145         points = points.cpu().numpy()
146     if ref_boxes is not None and not isinstance(ref_boxes, np.ndarray):
147         ref_boxes = ref_boxes.cpu().numpy()
148     if gt_boxes is not None and not isinstance(gt_boxes, np.ndarray):
149         gt_boxes = gt_boxes.cpu().numpy()
150     if ref_scores is not None and not isinstance(ref_scores, np.ndarray):
151         ref_scores = ref_scores.cpu().numpy()
152     if ref_labels is not None and not isinstance(ref_labels, np.ndarray):
153         ref_labels = ref_labels.cpu().numpy()
154 
155     fig = visualize_pts(points)
```

```
156     fig = draw_multi_grid_range(fig, bv_range=(0, -40, 80, 40))
157     if gt_boxes is not None:
158         corners3d = boxes_to_corners_3d(gt_boxes)
159         fig = draw_corners3d(corners3d, fig=fig, color=(0, 0, 1), max_num=100)
160 
161     if ref_boxes is not None and len(ref_boxes) > 0:
162         ref_corners3d = boxes_to_corners_3d(ref_boxes)
163         if ref_labels is None:
164             fig = draw_corners3d(ref_corners3d, fig=fig, color=(0, 1, 0), cls=ref_scores, max_num
                =100)
165         else:
166             for k in range(ref_labels.min(), ref_labels.max() + 1):
167                 cur_color = tuple(box_colormap[k % len(box_colormap)])
168                 mask = (ref_labels == k)
169                 fig = draw_corners3d(ref_corners3d[mask], fig=fig, color=cur_color, cls=ref_scores[
                    mask], max_num=100)
170     mlab.view(azimuth=-179, elevation=54.0, distance=104.0, roll=90.0)
171     return fig
172 
173 
174 def draw_corners3d(corners3d, fig, color=(1, 1, 1), line_width=2, cls=None, tag='', max_num=500,
     tube_radius=None):
175     """
176     :param corners3d: (N, 8, 3)
177     :param fig:
178     :param color:
179     :param line_width:
180     :param cls:
181     :param tag:
182     :param max_num:
183     :return:
184     """
185     import mayavi.mlab as mlab
186     num = min(max_num, len(corners3d))
187     for n in range(num):
188         b = corners3d[n] # (8, 3)
189 
190         if cls is not None:
191             if isinstance(cls, np.ndarray):
192                 mlab.text3d(b[6, 0], b[6, 1], b[6, 2], '%.2f' % cls[n], scale=(0.3, 0.3, 0.3), color
                    =color, figure=fig)
```

```
193             else:
194                 mlab.text3d(b[6, 0], b[6, 1], b[6, 2], '%s' % cls[n], scale=(0.3, 0.3, 0.3), color=
                        color, figure=fig)
195
196         for k in range(0, 4):
197             i, j = k, (k + 1) % 4
198             mlab.plot3d([b[i, 0], b[j, 0]], [b[i, 1], b[j, 1]], [b[i, 2], b[j, 2]], color=color,
                    tube_radius=tube_radius,
199                         line_width=line_width, figure=fig)
200
201             i, j = k + 4, (k + 1) % 4 + 4
202             mlab.plot3d([b[i, 0], b[j, 0]], [b[i, 1], b[j, 1]], [b[i, 2], b[j, 2]], color=color,
                    tube_radius=tube_radius,
203                         line_width=line_width, figure=fig)
204
205             i, j = k, k + 4
206             mlab.plot3d([b[i, 0], b[j, 0]], [b[i, 1], b[j, 1]], [b[i, 2], b[j, 2]], color=color,
                    tube_radius=tube_radius,
207                         line_width=line_width, figure=fig)
208
209         i, j = 0, 5
210         mlab.plot3d([b[i, 0], b[j, 0]], [b[i, 1], b[j, 1]], [b[i, 2], b[j, 2]], color=color,
                tube_radius=tube_radius,
211                     line_width=line_width, figure=fig)
212         i, j = 1, 4
213         mlab.plot3d([b[i, 0], b[j, 0]], [b[i, 1], b[j, 1]], [b[i, 2], b[j, 2]], color=color,
                tube_radius=tube_radius,
214                     line_width=line_width, figure=fig)
215
216     return fig
```

A.8.11. Organize experiments

For a detailed understanding of its application, refer to the section 4.5.

```
1 import shutil
2 import random
3 import os
4
5 # Function to count the number of objects in each file
6 def count_objects(filename):
```

```
    with open(filename, 'r') as f:
        lines = f.readlines()
    return len(lines)

# Define various file names and dataset parameters
dataset_name = '....'
labels_avx_kitti_format = 'labels_AVX'
points_AVX_database = 'points_AVX'
calib_avx_kitti_format = 'calib_AVX'
################################################################################
my_data_size = int(3712*1) # The size of the data to be used
percent_synthetic = ... # The proportion of synthetic data to be used
# 0.9 for %90 KITTI %10 AVX training
################################################################################
kitti_training_train = 'kitti_training_train' # The directory containing the real training data

# Define the directories for the synthetic data
labels_synthetic = f'data/AVX_DATA/Version_6/Train/{labels_avx_kitti_format}'
points_synthetic = f'data/AVX_DATA/Version_6/Train/{points_AVX_database}'
calib_synthetic = f'data/AVX_DATA/Version_6/Train/{calib_avx_kitti_format}'

# Define the directories for the real data
points_train_real = f'data/{kitti_training_train}'
labels_real = 'data/kitti/training/label_2'
images_real = 'data/kitti/training/image_2'
calib_real = 'data/kitti/training/calib'

# Define the destination directories for the various components of the dataset
points_destination_dir = f'data/{dataset_name}/training/velodyne'
labels_destination_dir = f'data/{dataset_name}/training/label_2'
calib_destination_dir = f'data/{dataset_name}/training/calib'
image_destination_dir = f'data/{dataset_name}/training/image_2'
imagesets = f'data/{dataset_name}/ImageSets'

# Create the destination directories if they do not exist
if not os.path.exists(points_destination_dir):
    os.makedirs(points_destination_dir)
if not os.path.exists(labels_destination_dir):
    os.makedirs(labels_destination_dir)
if not os.path.exists(calib_destination_dir):
    os.makedirs(calib_destination_dir)
```

```
if not os.path.exists(image_destination_dir):
    os.makedirs(image_destination_dir)
if not os.path.exists(imagesets):
    os.makedirs(imagesets)

# Get a list of all the label files in the synthetic data directory
label_files_synth = []
for filename in os.listdir(labels_synthetic):
    if filename.endswith('.txt'):
        label_files_synth.append(os.path.join(labels_synthetic, filename))

# Count the number of objects in each synthetic data file
object_counts = {}
for filename in label_files_synth:
    object_counts[filename] = count_objects(filename)

# Sort the synthetic data files based on their object count
sorted_files = sorted(label_files_synth, key=lambda x: object_counts[x], reverse=True)

# Calculate the size of the synthetic data based on the given percentage
synthetic_data_size = int(my_data_size * percent_synthetic)

# Select the synthetic data files up to the calculated size
selected_avx = sorted_files[:synthetic_data_size]

# Convert the synthetic label files to point cloud files
for i in range(len(selected_avx)):
    selected_avx[i] = selected_avx[i].replace(labels_avx_kitti_format, points_AVX_database).
        replace(".txt", ".npy")

# Get a list of all the point cloud files in the real data directory
files_points_train_real = os.listdir(points_train_real)
files_points_train_real = [f for f in files_points_train_real if f.endswith('.npy')]
files_points_train_real.sort()

# Select the real data files up to the remaining size
files_points_train_real = files_points_train_real[:my_data_size-synthetic_data_size]

# Add the directory path to each real data file
files_points_train_real = [os.path.join(points_train_real, f) for f in files_points_train_real]

```

```
# Merge the synthetic and real data lists and shuffle them
train_database = files_points_train_real + selected_avx
random.shuffle(train_database)

# Keep track of existing files to avoid duplication
existing_files = set(os.listdir(points_destination_dir))

new_filenames = []

# Copy and rename each file in the merged list to the destination directory
for i, filename in enumerate(train_database):
    if filename.endswith('.npy'):
        # Define new filenames based on the index
        new_filename = '{:06d}.npy'.format(i)
        while new_filename in existing_files:
            i += 1
            new_filename = '{:06d}.npy'.format(i)
        shutil.copy(filename, os.path.join(points_destination_dir,new_filename))

        # Copy and rename corresponding label, calibration, and image files
        new_label_filename = new_filename.split('.')[0] + '.txt'
        new_image_filename = new_filename.split('.')[0] + '.png'

        # Handle synthetic data
        if points_AVX_database in filename:
            file_name = os.path.basename(filename)
            new_file_name = file_name[:-4] + '.txt'
            new_file_path = filename.replace(points_AVX_database, labels_avx_kitti_format)
            label_AVX_full_path = new_file_path.replace(file_name, new_file_name)
            shutil.copy(label_AVX_full_path, os.path.join(labels_destination_dir,new_label_filename
                ))
            new_file_path_calib = filename.replace(points_AVX_database, calib_avx_kitti_format)
            calib_AVX_full_path = new_file_path_calib.replace(file_name, new_file_name)
            shutil.copy(calib_AVX_full_path, os.path.join(calib_destination_dir,new_label_filename)
                )

        # Handle real data
        if kitti_training_train in filename:
            file_name = os.path.basename(filename)
            file_name = file_name[:-4]
            label_kitti_train_path = os.path.join(labels_real, file_name + '.txt')
```

```
            shutil.copy(label_kitti_train_path, os.path.join(labels_destination_dir,
                new_label_filename))
            calib_kitti_train_path = os.path.join(calib_real, file_name + '.txt')
            shutil.copy(calib_kitti_train_path, os.path.join(calib_destination_dir,
                new_label_filename))
            image_kitti_train_path = os.path.join(images_real, file_name + '.png')
            shutil.copy(image_kitti_train_path, os.path.join(image_destination_dir,
                new_image_filename))

        # Add the new filename to the existing files set and new filenames list
        existing_files.add(new_filename)
        new_filenames.append(new_filename.split('.')[0])

# Create a train.txt file with the names of the point cloud files to be used for training
with open(os.path.join(imagesets, 'train.txt'), 'w') as f:
    for filename in new_filenames:
        f.write(filename + '\n')
```

A.8.12. Screenshots to video

For a detailed understanding of its application, refer to the section 5.2.1.

```
import cv2
import os

def create_video_from_screenshots(image_folder, output_video_path, fps):
    # Initialize an empty list to store the images
    images = []
    # For each file in the sorted list of files in the image folder
    for filename in sorted(os.listdir(image_folder)):
        # If the file is a PNG image
        if filename.endswith('.png'):
            # Get the path to the image
            img_path = os.path.join(image_folder, filename)
            # Read the image and append it to the images list
            images.append(cv2.imread(img_path))

    # If no images were found, print a message and return
    if len(images) == 0:
        print("No images found in the specified folder.")
        return
```

```
20
21      # Get the dimensions of the images
22      height, width, _ = images[0].shape
23
24      # Define the codec using VideoWriter_fourcc() and create a VideoWriter object
25      fourcc = cv2.VideoWriter_fourcc(*'mp4v')
26      video_writer = cv2.VideoWriter(output_video_path, fourcc, fps, (width, height))
27
28      # For each image, write it to the video file
29      for image in images:
30          video_writer.write(image)
31
32      # After all images have been written, release the VideoWriter
33      video_writer.release()
34
35      # Print a success message
36      print(f"Video␣created␣successfully:␣{output_video_path}")
37
38
39  # Specify the folder containing images, the output path for the video and the frame rate (fps)
40  image_folder = os.path.join('./', 'AVX_demoset_avx_train')
41  output_video_path = os.path.join(image_folder, 'demoset_0_5fps.mp4')
42  fps = 0.5
43
44  # Call the function to create the video
45  create_video_from_screenshots(image_folder, output_video_path, fps)
```

List of Figures

List of Tables

Bibliography

[1] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby, and A. Mouzakitis. "A Survey on 3D Object Detection Methods for Autonomous Driving Applications". In: *IEEE Transactions on Intelligent Transportation Systems* 20 (10 Oct. 2019), pp. 3782–3795. ISSN: 1524-9050. DOI: 10.1109/TITS.2019.2892405.

[2] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom. "PointPillars: Fast Encoders for Object Detection from Point Clouds". In: (Dec. 2018).

[3] F. Moosmann, O. Pink, and C. Stiller. "Segmentation of 3D lidar data in non-flat urban environments using a local convexity criterion". In: IEEE, June 2009, pp. 215–220. ISBN: 978-1-4244-3503-6. DOI: 10.1109/IVS.2009.5164280.

[4] Y. Wu, Y. Wang, S. Zhang, and H. Ogai. "Deep 3D Object Detection Networks Using LiDAR Data: A Review". In: *IEEE Sensors Journal* 21 (2 Jan. 2021), pp. 1152–1171. ISSN: 1530-437X. DOI: 10.1109/JSEN.2020.3020626.

[5] X. Yue, B. Wu, S. A. Seshia, K. Keutzer, and A. L. Sangiovanni-Vincentelli. "A LiDAR Point Cloud Generator: from a Virtual World to Autonomous Driving". In: (Mar. 2018).

[6] P. Jabłoński, J. Iwaniec, and W. Zabierowski. "Comparison of Pedestrian Detectors for LiDAR Sensor Trained on Custom Synthetic, Real and Mixed Datasets". In: *Sensors* 22 (18 Sept. 2022), p. 7014. ISSN: 1424-8220. DOI: 10.3390/s22187014.

[7] H. He, K. Khoshelham, and C. Fraser. "A multiclass TrAdaBoost transfer learning algorithm for the classification of mobile lidar data". In: *ISPRS Journal of Pho-*

togrammetry and Remote Sensing 166 (Aug. 2020), pp. 118–127. ISSN: 09242716. DOI: 10.1016/j.isprsjprs.2020.05.010.

[8] J. Fang, D. Zhou, F. Yan, T. Zhao, F. Zhang, Y. Ma, L. Wang, and R. Yang. "Augmented LiDAR Simulator for Autonomous Driving". In: (Nov. 2018). DOI: 10.1109/LRA.2020.2969927.

[9] S. Huang, L. Liu, X. Fu, J. Dong, F. Huang, and P. Lang. "Overview of LiDAR point cloud target detection methods based on deep learning". In: *Sensor Review* 42 (5 Aug. 2022), pp. 485–502. ISSN: 0260-2288. DOI: 10.1108/SR-01-2022-0022.

[10] S. Y. Alaba and J. E. Ball. "A Survey on Deep-Learning-Based LiDAR 3D Object Detection for Autonomous Driving". In: *Sensors* 22 (24 Dec. 2022), p. 9577. ISSN: 1424-8220. DOI: 10.3390/s22249577.

[11] M. Hahner, C. Sakaridis, M. Bijelic, F. Heide, F. Yu, D. Dai, and L. V. Gool. "LiDAR Snowfall Simulation for Robust 3D Object Detection". In: IEEE, June 2022, pp. 16343–16353. ISBN: 978-1-6654-6946-3. DOI: 10.1109/CVPR52688.2022.01588.

[12] M. Neri and F. Battisti. "3D Object Detection on Synthetic Point Clouds for Railway Applications". In: IEEE, Sept. 2022, pp. 1–6. ISBN: 978-1-6654-6623-3. DOI: 10.1109/EUVIP53989.2022.9922901.

[13] S. A. Chitnis, Z. Huang, and K. Khoshelham. "GENERATING SYNTHETIC 3D POINT SEGMENTS FOR IMPROVED CLASSIFICATION OF MOBILE LIDAR POINT CLOUDS". In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLIII-B2-2021 (June 2021), pp. 139–144. ISSN: 2194-9034. DOI: 10.5194/isprs-archives-XLIII-B2-2021-139-2021.

[14] D. Dworak, F. Ciepiela, J. Derbisz, I. Izzat, M. Komorkiewicz, and M. Wojcik. "Performance of LiDAR object detection deep learning architectures based on artificially generated point cloud data from CARLA simulator". In: IEEE, Aug. 2019, pp. 600–605. ISBN: 978-1-7281-0933-6. DOI: 10.1109/MMAR.2019.8864642.

[15] F. Wang, Y. Zhuang, H. Gu, and H. Hu. "Automatic Generation of Synthetic LiDAR Point Clouds for 3-D Data Analysis". In: *IEEE Transactions on Instrumentation and Measurement* 68 (7 July 2019), pp. 2671–2673. ISSN: 0018-9456. DOI: 10.1109/TIM.2019.2906416.

[16] J. Beltran, I. Cortes, A. Barrera, J. Urdiales, C. Guindel, F. Garcia, and A. de la Escalera. "A Method for Synthetic LiDAR Generation to Create Annotated Datasets for Autonomous Vehicles Perception". In: IEEE, Oct. 2019, pp. 1091–1096. ISBN: 978-1-5386-7024-8. DOI: 10.1109/ITSC.2019.8917176.

[17] S. Manivasagam, S. Wang, K. Wong, W. Zeng, M. Sazanovich, S. Tan, B. Yang, W.-C. Ma, and R. Urtasun. "LiDARsim: Realistic LiDAR Simulation by Leveraging the Real World". In: IEEE, June 2020, pp. 11164–11173. ISBN: 978-1-7281-7168-5. DOI: 10.1109/CVPR42600.2020.01118.

[18] D. Cheng. "3D Object Detection with Enriched Point Cloud". School of Computer Science Carnegie Mellon University, 2020, pp. 1–6.

[19] M. Hahner, C. Sakaridis, D. Dai, and L. V. Gool. "Fog Simulation on Real LiDAR Point Clouds for 3D Object Detection in Adverse Weather". In: IEEE, Oct. 2021, pp. 15263–15272. ISBN: 978-1-6654-2812-5. DOI: 10.1109/ICCV48922.2021.01500.

[20] M. Fabbri, G. Braso, G. Maugeri, O. Cetintas, R. Gasparini, A. Osep, S. Calderara, L. Leal-Taixe, and R. Cucchiara. "MOTSynth: How Can Synthetic Data Help Pedestrian Detection and Tracking?" In: (Aug. 2021).

[21] *Ansys AVxcelerate Sensors | AV Sensor Simulation Software — ansys.com*. https://www.ansys.com/products/av-simulation/ansys-avxcelerate-sensors. [Accessed 15-Jun-2023].

[22] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets robotics: The KITTI dataset". In: *The International Journal of Robotics Research* 32 (11 Sept. 2013), pp. 1231–1237. ISSN: 0278-3649. DOI: 10.1177/0278364913491297.

[23] J. Maynard. *The HDL-64E Lidar Sensor Retires | Velodyne Lidar — velodynelidar.com*. https://velodynelidar.com/blog/hdl-64e-lidar-sensor-retires/. [Accessed 15-Jun-2023].

[24] *CarMaker | IPG Automotive — ipg-automotive.com*. https://ipg-automotive.com/en/products-solutions/software/carmaker/. [Accessed 15-Jun-2023].

[25] C. Nast. *Driverless Cars Need Ears as Well as Eyes — wired.com*. https://www.wired.com/story/driverless-cars-need-ears-as-well-as-eyes/. [Accessed 15-Jun-2023].

[26] W. Zheng, W. Tang, L. Jiang, and C.-W. Fu. "SE-SSD: Self-Ensembling Single-Stage Object Detector From Point Cloud". In: (Apr. 2021).

[27] W. Zheng, W. Tang, S. Chen, L. Jiang, and C.-W. Fu. "CIA-SSD: Confident IoU-Aware Single-Stage Object Detector From Point Cloud". In: (Dec. 2020).

[28] Y. Zhou and O. Tuzel. "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection". In: (Nov. 2017).

[29] B. Wu, X. Zhou, S. Zhao, X. Yue, and K. Keutzer. "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud". In: (Sept. 2018).

[30] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. "Deep Learning for 3D Point Clouds: A Survey". In: (Dec. 2019).

[31] D. J. Yeong, G. Velasco-Hernandez, J. Barry, and J. Walsh. "Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review". In: *Sensors* 21 (6 Mar. 2021), p. 2140. ISSN: 1424-8220. DOI: 10.3390/s21062140.

[32] S. Sivaraman and M. M. Trivedi. "Looking at Vehicles on the Road: A Survey of Vision-Based Vehicle Detection, Tracking, and Behavior Analysis". In: *IEEE Transactions on Intelligent Transportation Systems* 14 (4 Dec. 2013), pp. 1773–1795. ISSN: 1524-9050. DOI: 10.1109/TITS.2013.2266661.

[33] D. Wang, C. Watkins, and H. Xie. "MEMS Mirrors for LiDAR: A Review". In: *Micromachines* 11 (5 Apr. 2020), p. 456. ISSN: 2072-666X. DOI: 10.3390/mi11050456.

[34] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. "Multi-View 3D Object Detection Network for Autonomous Driving". In: (Nov. 2016).

[35] C. S. in Lidar Toolbox - MATLAB & Simulink - MathWorks. https://nl.mathworks.com/help/lidar/ug/lidar-coordinate-systems.html. [Accessed 15-Jun-2023].

[36] Z. Wang, S. Ding, Y. Li, M. Zhao, S. Roychowdhury, A. Wallin, G. Sapiro, and Q. Qiu. "Range Adaptation for 3D Object Detection in LiDAR". In: (Sept. 2019).

[37] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov. "Scalability in Perception for Autonomous Driving: Waymo Open Dataset". In: IEEE, June 2020, pp. 2443–2451. ISBN: 978-1-7281-7168-5. DOI: 10.1109/CVPR42600.2020.00252.

[38] J. Houston, G. Zuidhof, L. Bergamini, Y. Ye, L. Chen, A. Jain, S. Omari, V. Iglovikov, and P. Ondruska. *One Thousand and One Hours: Self-driving Motion Prediction Dataset*. 2020. arXiv: 2006.14480 [cs.CV].

[39] 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: IEEE, June 2020, pp. 11618–11628. ISBN: 978-1-7281-7168-5. DOI: 10.1109/CVPR42600.2020.01164.

[40] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. *End to End Learning for Self-Driving Cars*. 2016. arXiv: 1604.07316 [cs.CV].

[41] S. Shah, D. Dey, C. Lovett, and A. Kapoor. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. 2017. arXiv: 1705.05065 [cs.RO].

[42] M. Müller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem. "Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications". In: *International Journal of Computer Vision* 126.9 (Mar. 2018), pp. 902–919. DOI: 10.1007/s11263-018-1073-7. URL: https://doi.org/10.1007%5C%2Fs11263-018-1073-7.

[43] S. R. Richter, Z. Hayder, and V. Koltun. *Playing for Benchmarks*. 2017. arXiv: 1709.07322 [cs.CV].

[44] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. *CARLA: An Open Urban Driving Simulator*. 2017. arXiv: 1711.03938 [cs.LG].

[45] *Bidirectional reflectance distribution function - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function. [Accessed 17-Jun-2023].

[46] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2. URL: https://doi.org/10.1007%5C%2F978-3-319-46448-0_2.

[47] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: IEEE, June 2016, pp. 779–788. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.91.

[48] M. Simon, S. Milz, K. Amende, and H.-M. Gross. *Complex-YOLO: Real-time 3D Object Detection on Point Clouds*. 2018. arXiv: 1803.06199 [cs.CV].

[49] Z. Yang, Y. Sun, S. Liu, and J. Jia. *3DSSD: Point-based 3D Single Stage Object Detector*. 2020. arXiv: 2002.10187 [cs.CV].

[50] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV].

[51] S. Shi, X. Wang, and H. Li. *PointRCNN: 3D Object Proposal Generation and Detection from Point Cloud*. 2019. arXiv: 1812.04244 [cs.CV].

[52] S. Shi, Z. Wang, J. Shi, X. Wang, and H. Li. *From Points to Parts: 3D Object Detection from Point Cloud with Part-aware and Part-aggregation Network*. 2020. arXiv: 1907.03670 [cs.CV].

[53] Z. Yang, Y. Sun, S. Liu, X. Shen, and J. Jia. *STD: Sparse-to-Dense 3D Object Detector for Point Cloud*. 2019. arXiv: 1907.10471 [cs.CV].

[54] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[55] V. Nair and G. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair". In: vol. 27. June 2010, pp. 807–814.

[56] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88 (2 June 2010), pp. 303–338. ISSN: 0920-5691. DOI: 10.1007/s11263-009-0275-4.

[57] *Attenuation coefficient - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Attenuation_coefficient. [Accessed 17-Jun-2023].

[58] O. D. Team. *OpenPCDet: An Open-source Toolbox for 3D Object Detection from Point Clouds*. https://github.com/open-mmlab/OpenPCDet. 2020.

[59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[60] 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.

[61] A. Simonelli, S. R. R. Bulò, L. Porzi, M. López-Antequera, and P. Kontschieder. *Disentangling Monocular 3D Object Detection*. 2019. arXiv: 1905.12365 [cs.CV].