Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Evaluating and Enhancing 3D Object Detection Algorithms in Autonomous Systems by Utilizing Stereo/Pseudo-LiDAR and LiDAR Data

Master's Thesis

by

Om Tiwari

August 13, 2025

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Didier Stricker
Dr. Rene Schuster

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Evaluating and Enhancing 3D Object Detection Algorithms in Autonomous Systems by Utilizing Stereo/Pseudo-LiDAR and LiDAR Data“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 13.8.2025

Om Tiwari

Abstract

The advancement of autonomous systems, particularly in autonomous driving, critically depends on robust and accurate 3D object detection. This thesis investigates the evolution of 3D object detection methodologies, transitioning from classical geometry-driven techniques to modern deep learning-based approaches. The research was motivated by the need to modernize an initial experimental setup provided by Astemo, which relied on traditional geometric methods. This thesis was conducted in collaboration with Astemo (formerly Hitachi Astemo), Munich, Germany, and the German Research Center for Artificial Intelligence (DFKI), Kaiserslautern, Germany.

A key contribution of this work is the development and comprehensive evaluation of advanced deep neural networks for 3D object detection using Velodyne LiDAR point clouds. The detection backbone was implemented using the OpenPCDet framework, integrating and benchmarking prominent models such as PV-RCNN, PointPillars, and PointRCNN. This evaluation was conducted on both proprietary LiDAR data from Astemo (formatted as ROS bags) and the public Kitti dataset. The Robot Operating System (ROS) was utilized for real-time visualization across both datasets and enabled frame-accurate mAP calculation for the Astemo dataset through synchronized timestamp alignment.

Furthermore, this research explores image-based 3D object detection through the generation and utilization of pseudo-LiDAR representations. Dense depth maps were generated from stereo image pairs using PSMNet, subsequently transformed into 3D pseudo-point clouds, and employed for end-to-end training based on the methodology proposed by Qian et al. 2020. The pseudo-LiDAR detection system was retrained and validated on the Kitti dataset.

Quantitative evaluation of all approaches was performed using the mean Average Precision (mAP) metric. This thesis presents a full-stack comparison, analyzing the performance of classical versus learning-based methods, and real LiDAR versus pseudo-LiDAR modalities. The strengths and limitations of these approaches are assessed within the context of real-time perception systems, with particular focus on the current potential and inherent constraints of stereo vision for 3D object perception. The findings provide valuable insights into the efficacy of various 3D detection strategies, informing future development in autonomous system perception.

Zusammenfassung

Der Fortschritt autonomer Systeme, insbesondere im Bereich des autonomen Fahrens, hängt entscheidend von robusten und präzisen 3D-Objekterkennungssystemen ab. Diese Arbeit untersucht die Entwicklung von Methoden zur 3D-Objekterkennung, die sich von klassischen geometriebasierten Techniken hin zu modernen Deep-Learning-Ansätzen wandeln. Die Forschung wurde durch die Notwendigkeit motiviert, eine initiale experimentelle Plattform von Astemo zu modernisieren, die auf traditionellen geometrischen Methoden basierte.

Ein wesentlicher Beitrag dieser Arbeit ist die Entwicklung und umfassende Evaluierung fortgeschrittener Deep Neural Networks für die 3D-Objekterkennung mittels Velodyne-LiDAR-Punktwolken. Das Detektions-Backbone wurde mit dem OpenPCDet-Framework implementiert, das prominente Modelle wie PV-RCNN, PointPillars und PointRCNN integriert und vergleicht. Diese Evaluierung erfolgte sowohl mit proprietären LiDAR-Daten von Astemo (im ROS-Bag-Format) als auch mit dem öffentlichen Kitti-Datensatz. Das Robot Operating System (ROS) wurde für die Echtzeit-Visualisierung über beide Datensätze hinweg genutzt und ermöglichte rahmengenaue mAP-Berechnungen für den Astemo-Datensatz durch synchronisierte Zeitstempel-Abgleichung.

Darüber hinaus erforscht diese Arbeit bildbasierte 3D-Objekterkennung durch Generierung und Nutzung von Pseudo-LiDAR-Repräsentationen. Dichte Tiefenkarten wurden aus Stereobildpaaren mit PSMNet erzeugt, anschließend in 3D-Pseudo-Punktwolken transformiert und für End-to-End-Training nach der Methodik von Qian et al. 2020 eingesetzt. Das Pseudo-LiDAR-Erkennungssystem wurde auf dem Kitti-Datensatz neu trainiert und validiert.

Die quantitative Bewertung aller Ansätze erfolgte mittels der Mean Average Precision (mAP)-Metrik. Diese Arbeit präsentiert einen Full-Stack-Vergleich, der die Leistung klassischer versus lernbasierter Methoden sowie echter LiDAR- versus Pseudo-LiDAR-Modalitäten analysiert. Die Stärken und Grenzen dieser Ansätze werden im Kontext von Echtzeit-Wahrnehmungssystemen bewertet, mit besonderem Fokus auf das aktuelle Potenzial und die inhärenten Beschränkungen der Stereobildverarbeitung für die 3D-Objekterkennung. Die Ergebnisse liefern wertvolle Einblicke in die Wirksamkeit verschiedener 3D-Erkennungsstrategien und leisten damit einen Beitrag zur zukünftigen Entwicklung der Wahrnehmung autonomer Systeme.

Contents

1. Introduction

1.1. Motivation and Problem Statement

The capacity for robust and accurate 3D object detection is a cornerstone for the safe and efficient operation of autonomous systems, with autonomous driving serving as a prominent example Xu et al. 2022. These systems must perceive their surroundings with high fidelity, localizing and classifying objects such as vehicles, pedestrians, and cyclists to navigate complex, dynamic environments and make critical decisions to avoid collisions Qian et al. 2020. Historically, 3D perception, particularly from Light Detection and Ranging (LiDAR) sensors, relied heavily on classical geometric methods Maskeliūnas et al. 2025. These foundational techniques, while instrumental in early developments, often exhibit limitations in scalability, robustness to diverse environmental conditions, and the inherent complexity of real-world scenarios Trigka et al. 2025. The intricate nature of urban environments, with varying object densities, occlusions, and sensor noise, poses significant challenges to the efficacy of these traditional approaches.

The advent of deep learning (DL) has instigated a paradigm shift across numerous computer vision tasks, and its application to 3D perception is rapidly maturing Sapkota et al. 2025. Deep Learning models have demonstrated a superior ability to learn hierarchical features directly from raw sensor data, leading to enhanced generalization capabilities and higher accuracy in object detection tasks Trigka et al. 2025. This data-driven approach contrasts with classical methods that often depend on manually engineered features and explicit model assumptions. The transition from these classical methodologies to DL-driven solutions is not merely an academic pursuit but is propelled by compelling industrial demands for perception systems that offer heightened accuracy, improved robustness against environmental variations, and the capacity to effectively interpret complex and dynamic scenes—capabilities where classical methods often fall short.

This thesis is motivated by the need to modernize an existing 3D object detection workflow. The initial experimental setup, provided by Astemo, an automotive industry partner, was based on traditional geometry-driven techniques applied to proprietary LiDAR data. The primary problem addressed herein is the development, implementation, and rigorous evaluation of a comprehensive 3D object detection pipeline leveraging contemporary deep learning models. This endeavor encompasses the processing of both proprietary LiDAR data from Astemo and data from the publicly available Kitti benchmark, a standard in autonomous driving research.

A critical constraint in the deployment of perception systems for autonomous vehicles is the necessity for real-time processing Peng et al. 2015. The dynamic nature of driving scenarios demands that perception and decision-making occur with minimal latency. Consequently, this research emphasizes the development of efficient model architectures and pipeline designs capable of meeting these stringent real-time requirements. The choice of the Robot Operating System 2 (ROS2) as the foundational framework for the developed pipeline underscores this focus on real-world deployability and integration within established robotics ecosystems Hong et al. 2024. ROS2 provides the necessary tools and communication protocols for building complex,

distributed, and real-time robotics applications.

Furthermore, the significant cost associated with high-resolution LiDAR sensors has spurred research into alternative or complementary sensing modalities Qian et al. 2020. This thesis explores one such avenue: image-based 3D object detection facilitated by pseudo-LiDAR techniques Qian et al. 2020. By estimating depth from stereo camera images and converting this information into a 3D point cloud representation, pseudo-LiDAR offers a potentially more cost-effective approach to 3D perception. This dual exploration of direct LiDAR sensing and image-derived pseudo-LiDAR reflects a pragmatic engineering approach, aiming to balance the high performance achievable with dedicated 3D sensors against the economic and redundancy benefits offered by camera-based solutions. Such a balance is crucial for the widespread adoption and commercial viability of autonomous systems.

1.2. Research Objectives and Questions

- **Objective 1:** To implement and benchmark a selection of state-of-the-art deep learning models for LiDAR-based 3D object detection (PointRCNN) on both proprietary (Astemo) and public (KITTI) datasets. This involves a rigorous empirical evaluation of different architectural approaches to understand their performance characteristics across varied data sources.
- **Objective 2:** To investigate and implement an image-based 3D object detection approach using pseudo-LiDAR generation from stereo images, and evaluate its performance. This objective explores the viability of a lower-cost alternative to direct LiDAR sensing, focusing on a specific methodology for generating and utilizing pseudo-LiDAR data, and to answer whether pseudo-LiDAR can outperform or match real LiDAR in 3D detection tasks.
- **Objective 3:** To conduct a comprehensive comparative analysis of classical geometric methods versus deep learning approaches, and real LiDAR versus pseudo-LiDAR modalities, in the context of 3D object detection for autonomous systems. This overarching objective aims to synthesize the findings from the previous objectives into a holistic understanding of the trade-offs involved in different perception strategies.

The systematic inquiry posed by these research questions aims to deconstruct the multifaceted problem of 3D object detection. By progressing from direct LiDAR enhancements to the exploration of alternative modalities and culminating in a holistic comparison, this research endeavors to provide a thorough scientific investigation.

The use of both proprietary (Astemo) and public (KITTI) datasets, coupled with standardized frameworks like OpenPCDet for model implementation and ROS2 for pipeline deployment, implicitly targets the development of a transferable and adaptable methodology. This approach not only allows for benchmarking against the broader research community using public data but also demonstrates the applicability of these modern techniques to specific industrial requirements and potentially unique sensor characteristics encountered in proprietary systems.

1.3. Key Contributions of the Thesis

This thesis makes several key contributions to the field of 3D object detection for autonomous systems:

- **Development and Implementation of a ROS2 Pipeline:** A functional and efficient ROS2 pipeline was developed for ingesting Velodyne LiDAR point clouds and performing 3D object detection using deep neural networks. This pipeline serves as a practical platform for benchmarking and deploying various detection models in a real-time context.

- **Comprehensive Benchmarking on Diverse LiDAR Datasets:** Leading deep learning models for 3D object detection on PointRCNN were systematically implemented and benchmarked on both the public KITTI dataset and a proprietary LiDAR dataset from Astemo. This dual evaluation provides novel insights into the performance characteristics and generalization capabilities of these models across different sensor configurations and environmental conditions. The results on the proprietary Astemo dataset are particularly valuable as they offer insights not currently available in public literature, potentially highlighting performance nuances specific to industrial sensor setups.

- **Implementation and Evaluation of an End-to-End Pseudo-LiDAR Pipeline:** An image-based 3D object detection pipeline was implemented using pseudo-LiDAR generation. This involved leveraging PSMNet for dense depth map creation from stereo images and employing the end-to-end training methodology proposed by Qian et al. (2020) Qian et al. 2020. The performance of this pipeline was evaluated on the KITTI dataset.

- **Full-Stack Quantitative Comparative Analysis:** The research provides a holistic, full-stack quantitative comparison between:
  - Classical geometry-driven 3D object detection techniques (Astemo's initial setup) and modern deep learning–based approaches.
  - Real LiDAR data and pseudo-LiDAR data modalities for 3D object detection.

  This comparative analysis extends beyond isolated component evaluations to assess entire perception strategies, considering accuracy, computational cost, and real-time capability within the ROS2 framework. Such a comprehensive view offers a more complete picture for informed decision-making in the design of autonomous perception systems.

- **Evaluation of Stereo Vision for 3D Perception:** The thesis offers an assessment of the strengths, current limitations, and potential of stereo vision as a sensor modality for 3D object perception, particularly when its output is transformed into pseudo-LiDAR representations. This analysis considers its performance relative to direct LiDAR sensing under real-time operational constraints.

1.4. Thesis Outline

The remainder of this thesis is structured as follows:

- **Chapter 2: Background and Related Work** provides a comprehensive review of the literature, covering the evolution of 3D object detection from classical geometric methods to deep learning approaches, foundational point cloud processing architectures, specific 3D detection models, image-based detection techniques including pseudo-LiDAR, and relevant software frameworks.

- **Chapter 3: System Architecture and Implementation** details the design and development of the experimental systems used in this research, including the classical baseline, the real-time ROS2 pipeline, the LiDAR-based deep learning detection module, and the image-based pseudo-LiDAR detection module.
- **Chapter 4: Experimental Setup** describes the datasets utilized (proprietary Astemo and public KITTI), the evaluation metrics employed for performance assessment, the training procedures for the deep learning models, and the overall benchmarking strategy.
- **Chapter 5: Results and Discussion** presents the quantitative and qualitative results obtained from the experiments, followed by a thorough analysis and interpretation of these findings, including comparative performance evaluations.
- **Chapter 6: Conclusion and Future Work** summarizes the key findings and contributions of the thesis, answers the posed research questions, discusses the limitations of the current work, and suggests potential directions for future research in this domain.

2. Theory and Related Work

This chapter provides a comprehensive review of the literature pertinent to 3D object detection, establishing the theoretical and practical context for the research presented in this thesis. It traces the evolution of detection techniques, delves into foundational deep learning architectures for point cloud processing, discusses specific 3D object detection models, explores image-based approaches including pseudo-LiDAR, and examines relevant software frameworks and tools.

2.1. Evolution of 3D Object Detection

The endeavor to enable machines to perceive and understand the three-dimensional world has a rich history, with techniques evolving significantly over time, particularly in response to the demands of applications like autonomous navigation and robotics.

2.1.1. Classical Geometric Methods for 3D Object Detection

Before the widespread adoption of deep learning, 3D object detection from point cloud data predominantly relied on classical geometric methods. These approaches typically involve multi-stage pipelines that apply geometric reasoning and handcrafted feature extraction directly to the 3D point data Maskeliūnas et al. 2025. Common techniques included:

- Point Cloud Clustering: Algorithms such as Euclidean clustering or Density-Based Spatial Clustering of Applications with Noise (DBSCAN) were employed to group spatially proximate points. This step aimed to segment potential objects from background clutter or differentiate distinct object instances from one another.

- Segmentation: Beyond simple clustering, more sophisticated segmentation techniques like region growing were utilized. Region growing iteratively groups neighboring points based on shared properties, such as surface normal consistency or planarity, to form coherent segments representing parts of objects or entire objects.

- Shape Fitting and Model Matching: Once point clusters or segments were obtained, geometric primitives (e.g., cuboids, cylinders) or more complex predefined object models were fitted to these point sets. Algorithms like Random Sample Consensus (RANSAC) were instrumental in robustly fitting models in the presence of outliers, while the Hough Transform could be adapted for detecting specific shapes. These methods provided estimates of object pose and dimensions.

- Feature-Based Approaches: Some methods focused on extracting discriminative geometric features from point clouds, such as Point Feature Histograms (PFH), Fast Point Feature Histograms (FPFH), or spin images. These features would then be fed into traditional machine learning classifiers (e.g., Support Vector Machines) to identify object classes.

Classical geometric methods possess certain strengths, including inherent interpretability, as their operations are based on explicit geometric principles. For simpler scenes or well-defined objects, they could offer computationally lighter solutions and did not require the extensive labeled datasets characteristic of deep learning Trigka et al. 2025. However, these methods also exhibit significant limitations. They are often sensitive to sensor noise, varying point densities, and occlusions. Their performance can degrade considerably in complex, cluttered environments, and they typically struggle to generalize across a wide diversity of object shapes and appearances without extensive parameter tuning for specific scenarios Trigka et al. 2025. The initial experimental setup provided by Astemo, which this thesis aims to modernize, falls within this category of traditional geometry-driven techniques. These classical approaches, despite their limitations, established fundamental concepts in 3D perception, such as segmentation and shape fitting, which are conceptually mirrored in modern deep learning paradigms, albeit addressed with more powerful, data-driven techniques.

2.1.2. The Rise of Deep Learning in 3D Perception

The remarkable success of deep learning, particularly Convolutional Neural Networks (CNNs), in a multitude of 2D computer vision tasks, has naturally spurred its application to 3D perception Trigka et al. 2025. DL models offer the advantage of learning hierarchical feature representations directly from data, obviating the need for manual feature engineering and often leading to superior performance in terms of accuracy and robustness Trigka et al. 2025. However, the direct application of standard CNN architectures, which excel on regularly structured grid data like images, to raw 3D point clouds presents unique challenges. As stated in the user's introductory text, images are dense, structured arrays where spatial adjacency in the array corresponds to spatial adjacency in the scene, and operations like convolution exploit this regular neighborhood structure. Conversely, point clouds present a set of unordered and irregularly distributed points in 3D space, lacking the structured arrangement found in images Luo et al. 2024. This irregularity means that standard convolution operations, which rely on a fixed kernel and a consistent neighborhood definition, cannot be directly applied. The unordered nature implies that the network's output should be invariant to the permutation of input points. Consequently, specialized architectures have been developed to learn features from point clouds, accommodating their unique characteristics and enabling effective processing of 3D spatial data Luo et al. 2024. This "irregularity" of point clouds served as a primary catalyst for the development of novel DL architectures distinct from their 2D image-based counterparts, leading to innovative ways of defining operations like convolution and pooling for sparse, unordered 3D data.

2.2. Foundational Deep Learning Architectures for Point Cloud Processing

Addressing the unique nature of point cloud data required the development of new deep learning paradigms. Several foundational architectures have emerged, forming the building blocks for many advanced 3D object detection models. The evolution of these architectures reflects a progressive refinement in capturing both fine-grained local details and broader spatial context, driven consistently by the critical need for computational and memory efficiency, especially for real-world applications like autonomous driving.

2.2.1. PointNet Architecture

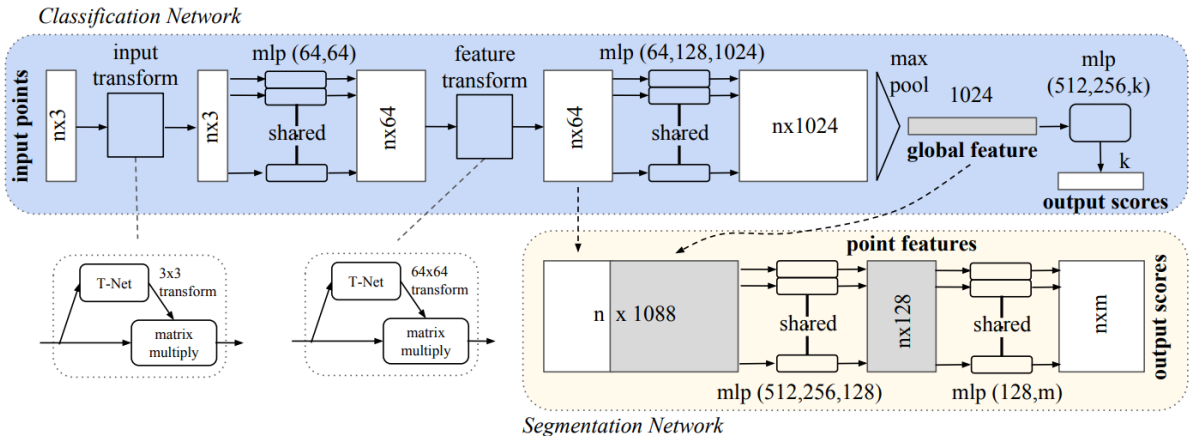

Figure 2.1.: *Hierarchical architecture of PointNet.(Adapted from Qi, Su, et al. 2017a)*

The Figure 2.1: Architecture of the original PointNet (classification and segmentation networks) Qi, Su, et al. 2017b. PointNet is a groundbreaking deep learning architecture designed to process raw, unordered 3D point clouds directly for tasks like classification and segmentation. Its core structure begins with an input transformation network (T-Net), which predicts a 3×3 matrix to align input points into a canonical orientation, ensuring rotational invariance. Aligned points then pass through shared multi-layer perceptrons (MLPs)—implemented as 1D convolutions—to extract per-point features. A second feature transformation network (T-Net) further aligns these features in high-dimensional space, regularized by an orthogonal loss to preserve geometric relationships. Critical to handling point order ambiguity, a symmetric max pooling layer aggregates all per-point features into a single global descriptor (1×1024 vector). For classification, this global feature feeds into MLPs to predict object categories. For segmentation, local features (from early MLPs) are fused with the global descriptor and processed by pointwise MLPs to assign per-point labels. By combining transformation invariance, shared feature learning, and max-pooling symmetry, PointNet achieves robustness to point permutations and rigid transformations while preserving geometric integrity, establishing a foundation for direct point cloud processing.

2.2.2. PointNet++ Architecture

PointNet++ builds on PointNet by introducing a hierarchical feature learning strategy that respects spatial locality in the point set Qi, Yi, et al. 2017. It breaks the point cloud into multiple scales of local regions and applies PointNet-based modules recursively on these subsets, analogous to how CNNs use hierarchies of local receptive fields Qi, Yi, et al. 2017. As shown in Figure 2.2 (illustrated with 2D points for simplicity), PointNet++ first partitions the input set of n points into many overlapping neighborhoods (local regions) using a distance metric (e.g. selecting region centroids by iterative farthest point sampling, then grouping points within a radius) Qi, Yi, et al. 2017. Each local region of points is then processed by a small PointNet (a set of shared MLP layers plus max-pooling, sometimes called a "mini-PointNet") to compute a local feature vector describing that region Qi, Yi, et al. 2017.

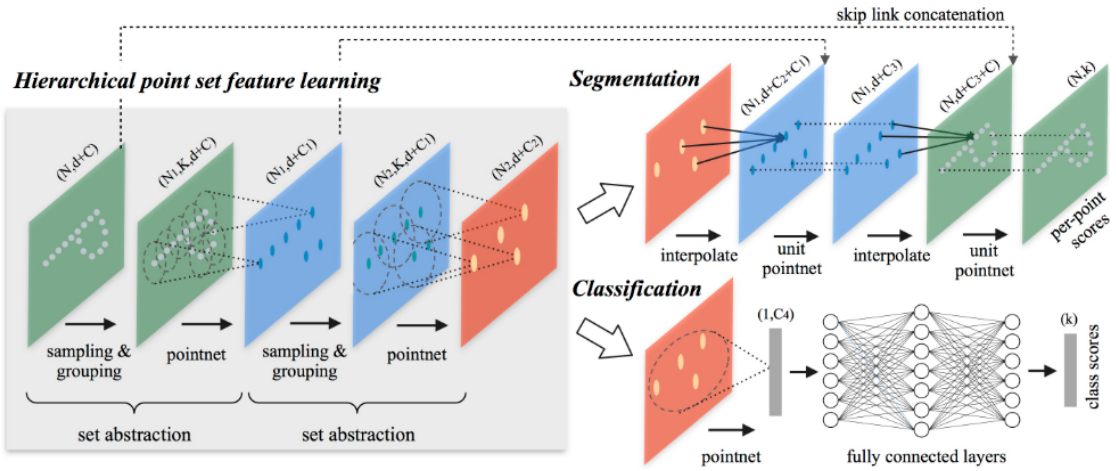

Figure 2.2.: *Hierarchical architecture of PointNet++ for point set feature learning. (Adapted from Qi, Yi, et al. 2017)*

This constitutes one Set Abstraction Level: a Sampling layer picks representative points (centroids), a Grouping layer gathers neighboring points for each centroid, and a PointNet layer extracts a feature for that neighborhood Qi, Yi, et al. 2017. The output of one set abstraction level is a new reduced set of points (the centroids) each with a higher-dimensional feature descriptor summarizing its local patch. Stacking multiple set abstraction levels (as in Figure 2, going from green to blue to red layers) yields a hierarchical network that captures local features at increasing scales.

At the final level, a small number of points (or just one single point for the whole object) with rich descriptors are obtained – these can be aggregated to form a global representation for classification, similarly to the original PointNet (e.g. one could max-pool or fully-connect to k class scores as done in the paper) Qi, Yi, et al. 2017.

2.2.3. Graph-based Feature Extraction

Dynamic Graph Convolutional Neural Network (DGCNN), proposed by Yue Wang et al. 2019, is a graph-based architecture tailored for learning from point cloud data. In DGCNN, the point cloud is represented as a graph where each point is a vertex, and edges are formed by connecting each point to its k-nearest neighbors. A significant aspect of DGCNN is that this graph is dynamic; it is recomputed at each layer of the network based on the evolving feature representations of the points, allowing the network to capture changing local structure. A key component of DGCNN is the EdgeConv operation Yue Wang et al. 2019. For each point, EdgeConv computes edge features by applying a function (typically an MLP) to the features of the central point and the features of its neighbors, often incorporating the difference between the central point's features and its neighbors' features. These edge features, which describe the relationships between a point and its local neighborhood, are then aggregated (e.g., via max pooling) to produce new features for the central point.

This process effectively captures local geometric information while maintaining permutation invariance with respect to the ordering of neighbors Yue Wang et al. 2019. By stacking EdgeConv

layers, DGCNN can learn increasingly abstract representations that integrate both local and global geometric information. Global features in DGCNN are extracted similarly to PointNet by applying shared MLPs followed by a symmetric aggregation function, such as max pooling, to ensure permutation invariance. This approach addresses the limitations of earlier methods like PointNet, which lacked explicit mechanisms to model local neighborhood structures, thereby enhancing the network's ability to learn intricate patterns within point cloud data.

2.2.4. Voxel-based Feature Extraction

Another prominent approach to point cloud learning involves voxelization, with VoxelNet Shi, C. Wang, Z. Zhang, et al. 2022 being a representative model in this category. VoxelNet partitions the 3D space of the point cloud into a regular grid of 3D voxels. For each non-empty voxel, it employs a Voxel Feature Encoding (VFE) layer to extract features from the points contained within that voxel. The VFE layer typically uses a simplified PointNet-like structure (point-wise MLPs followed by max pooling among points in the voxel) to learn a unified feature vector representing the geometric properties of the points within the voxel Shi, C. Wang, Z. Zhang, et al. 2022. This process transforms the irregular point cloud into a sparse volumetric tensor representation.

Subsequently, 3D convolutional layers are applied to these voxel-wise features. These 3D convolutions aggregate information from neighboring voxels, enabling the network to learn higher-level spatial representations and understand contextual relationships across larger regions of the scene, drawing inspiration from the success of 2D convolutional operations in image processing. While PointNet is lauded for its simplicity and strong representational capabilities, utilizing point-wise MLPs as its core component—which are also integral to graph-based and voxel-based feature extraction methods—voxel-based 3D convolutional neural networks often incur higher computational costs due to the additional dimension introduced in 3D convolutions Yue Wang et al. 2019. However, VoxelNet mitigates this by processing only non-empty voxels and utilizing a sparse tensor representation, significantly reducing memory usage and computation, especially since over 90 percent of voxels are typically empty in a LiDAR scan. Graph-based feature extraction methods, while more complex, build upon the principles of PointNet and are inherently adept at handling irregular data structures, garnering increasing attention in recent years for their effectiveness in processing point cloud data.

2.2.5. Sparse Convolutional Networks

The high computational and memory demands of standard 3D convolutions on voxelized point clouds, which are often very sparse, led to the development of Sparse Convolutional Networks (SCNs) Qi, Su, et al. 2017b. These networks are designed to efficiently process sparse 3D data by restricting computations only to "active" sites, i.e., non-empty voxels, and their neighborhoods. A particularly influential type is Submanifold Sparse Convolution, introduced by Graham et al..adam-abed-abud 2025 In submanifold sparse convolution, a convolutional operation is performed only when the center of the convolution kernel aligns with an active input site. Crucially, an output site is activated only if its corresponding input site (the one at the kernel's center) was active. This ensures that the sparsity pattern of the feature maps is preserved or maintained throughout the network layers, preventing the "dilation" or densification of active sites that would occur with standard convolutions. This property is vital for building deep 3D CNNs that can process large-scale point clouds without prohibitive computational costs.

Sparse convolutions, especially submanifold sparse convolutions, have become a cornerstone of many modern high-performance voxel-based 3D object detectors, such as SECOND Shi, C. Wang, Li, et al. 2020 and the voxel-based backbone of PV-RCNN Ren et al. 2017. They enable the use of powerful 3D CNN architectures by effectively managing the sparsity inherent in LiDAR data, thus achieving a favorable balance between representational power and computational efficiency.

2.3. Deep Learning-based 3D Object Detection Models

Building upon these foundational point cloud processing architectures, numerous 3D object detection models have been developed. These can be broadly categorized into single-stage and two-stage detectors, each representing a different trade-off between inference speed and detection accuracy—a critical consideration for real-time autonomous systems.

2.3.1. Single-Stage Detectors

Single-stage object detection frameworks for point clouds typically transform the raw data into a compact encoded format, which is then processed by a neural network specifically designed to interpret this structured representation and extract salient features. Subsequently, a detection head utilizes these features to perform object classification and predict the corresponding bounding boxes for each identified object in a single pass. These methods are generally favored for applications requiring high inference speed.

2.3.2. VoxelNet

Introduced by Zhou et al. (2018), VoxelNet is among the pioneering single-stage, end-to-end trainable frameworks for 3D object detection using point cloud data. The architecture is composed of three primary components: the feature learning network, the convolutional middle layers, and the region proposal network (RPN). (While it uses an RPN—often associated with two-stage detectors like Faster R-CNN Ren et al. 2017—VoxelNet's RPN is tightly integrated and trained end-to-end as part of a single network, leading to its common classification as single-stage in the 3D detection literature.) In the initial stage, the feature learning network processes raw point cloud data by partitioning the 3D space into equally spaced voxels. Each voxel contains a set of points, which are transformed into a unified feature representation through a Voxel Feature Encoding (VFE) layer.

This process results in a sparse tensor representation that includes only non-empty voxels, optimizing computational efficiency Zhou et al. 2018. The convolutional middle layers then apply a sequence of 3D sparse convolutions, batch normalization, and ReLU activations to aggregate the voxel-wise features, enhancing the contextual information across the spatial domain Zhou et al. 2018. Subsequently, the RPN utilizes the aggregated features to generate object proposals, outputting a probability score and a regression map for each detected object. A notable limitation of VoxelNet is its relatively high inference time, recorded at approximately 220 milliseconds on a TITAN X GPU.

2.3.3. Sparsely Embedded Convolutional Detection (SECOND)

To address VoxelNet's inference speed limitations, SECOND, proposed by Shi, C. Wang, Li, et al. 2020, adopts similar principles but incorporates a sparse convolutional middle extractor.

By leveraging submanifold sparse convolutions adam-abed-abud 2025, SECOND significantly reduces computation and memory usage, leading to faster inference times while maintaining competitive accuracy Shi, C. Wang, Li, et al. 2020. It has become a widely used baseline and a component in more advanced detectors.

2.3.4. PointPillars

Developed by Lang et al. 2019, PointPillars is a single-stage object detection framework that further enhances inference speed by structuring point cloud data into vertical columns known as "pillars," rather than 3D voxels. The architecture comprises three primary components: the Pillar Feature Network (PFN), a 2D convolutional backbone, and a detection head Lang et al. 2019. The PFN processes raw point cloud inputs by discretizing the data into a grid along the x–y plane, effectively creating a set of pillars. Each pillar's features are encoded using a simplified version of PointNet, resulting in a pseudo-image representation. This pseudo-image allows the subsequent backbone network to apply efficient 2D convolutions for feature extraction, leveraging the maturity and speed of 2D CNN operations. The detection head, often implemented using architectures like Single Shot Detector (SSD) H. Zhang et al. 2023, takes the features from the backbone to predict the bounding boxes of detected objects. Notably, PointPillars achieves an inference time of approximately 24 milliseconds on a GTX 1080 Ti GPU (62 Hz on KITTI), demonstrating its suitability for real-time applications Lang et al. 2019.

2.3.5. Structure Aware Single-Stage Detector (SA-SSD)

Proposed by He et al. H. Zhang et al. 2023, SA-SSD is a single-stage 3D object detection framework that builds upon the VoxelNet architecture by explicitly incorporating structural information from point cloud data. In this approach, the raw point cloud is partitioned into a voxel grid, where each voxel's feature is computed as the mean of the positions and intensities of the points it contains. This voxel grid serves as the input to the backbone network, which employs sparse 3D convolutional layers to efficiently extract features. The extracted features are then processed by the detection network, which utilizes 2D convolutional operations (after projecting features to Bird's Eye View) to predict object bounding boxes. A distinctive component of SA-SSD is its auxiliary network, designed to enhance the backbone's sensitivity to the structural details inherent in point cloud data H. Zhang et al. 2023.

This is achieved through point-wise supervision, where features from the convolutional layers are associated with the original points, enabling the network to better capture spatial relationships and part locations, thereby improving localization accuracy and addressing the common misalignment between classification confidence and localization quality H. Zhang et al. 2023. Performance evaluations indicate that SA-SSD achieves a runtime of approximately 40 milliseconds on an NVIDIA GTX 2080 Ti GPU.

2.3.6. Point-based 3D Single Stage Object Detector (3DSSD)

Introduced by Yang et al. 2020, 3DSSD presents an efficient, anchor-free framework for 3D object detection that operates directly on raw point cloud data, eliminating the need for voxelization or pillarization. This design choice aims to reduce computational overhead and enhance inference speed while preserving fine-grained point information Yang et al. 2020. 3DSSD employs Set Abstraction (SA) layers (from PointNet++) to downsample the point cloud and extract contextual

features. To further optimize performance, it introduces a fusion sampling strategy that combines Distance-based Farthest Point Sampling (D-FPS) and Feature-based Farthest Point Sampling (F-FPS) Yang et al. 2020. This hybrid approach ensures the preservation of semantically significant foreground points while discarding less informative background points. A key component of the architecture is the Candidate Generation (CG) layer, which shifts representative points closer to their respective object centers, enhancing localization accuracy. These candidate points, along with their neighboring points identified through D-FPS and F-FPS, are processed using Multi-Layer Perceptrons (MLPs) to extract discriminative features. The anchor-free prediction head then utilizes these features to regress object dimensions, positions, and orientations. 3DSSD achieves real-time inference speeds exceeding 25 frames per second (FPS), outperforming previous point-based methods and demonstrating its suitability for applications requiring rapid processing of 3D dataYang et al. 2020.

2.3.7. Two-Stage Detectors

Two-stage object detection frameworks divide the detection process into two sequential phases. Initially, the model identifies potential object locations by generating Regions of Interest (RoIs) from the input data, which may consist of raw point clouds or their compact representations. These RoIs, typically represented as bounding boxes, are then passed to the second phase, where detailed features are extracted for each region. This phase considers all points or refined features within each RoI to generate comprehensive feature representations. The features from both stages are combined to refine the bounding boxes and assign class labels to the detected objects. Compared to single-stage detectors, two-stage models generally achieve higher accuracy due to this refined processing of proposals. However, this typically comes at the cost of increased inference time, making them potentially less suitable for applications with extremely stringent real-time constraints. The field has seen continuous evolution in this category, with methods improving both accuracy and, to some extent, efficiency.

2.3.8. PointRCNN

The PointRCNN architecture for 3D object detection from point clouds, introduced by Shi et al. Shi, X. Wang, et al. 2019, is a two-stage framework that directly processes raw point cloud data. As depicted in Figure 2.3, which shows the core architectural diagram divided into parts (a) and (b) representing the two stages, PointRCNN first generates bottom-up proposals then refines them in canonical space.

The first stage, **Bottom-up 3D Proposal Generation**, processes raw point clouds through a PointNet++ backbone comprising an encoder that hierarchically abstracts features at multiple scales and a decoder that propagates features back to original points, producing comprehensive point-wise feature vectors. These vectors simultaneously feed two specialized sub-networks: a foreground segmentation network that classifies points as belonging to objects or background, and a bin-based proposal generator that predicts initial 3D bounding boxes exclusively from foreground points using discrete-continuous coordinate parameterization. The resulting 3D Regions of Interest (RoIs) preserve geometric details while establishing preliminary detection hypotheses.

The second stage, **Canonical 3D Box Refinement**, enhances proposal accuracy through viewpoint-invariant processing. For each initial RoI, the raw points within its volume are pooled and transformed into a canonical coordinate system where the proposal center becomes the origin and axes align with the predicted orientation. These canonical points are merged with

semantic features from the first stage and processed by a dedicated PointNet-like encoder. The refined features feed parallel prediction heads: a bin-based refinement network that adjusts box parameters (center, orientation, dimensions) using residual offsets, and a confidence network that predicts final detection scores. This two-stage approach with canonical transformation enables high-precision 3D detection while maintaining computational efficiency, as demonstrated on KITTI benchmark datasets Shi, X. Wang, et al. 2019.

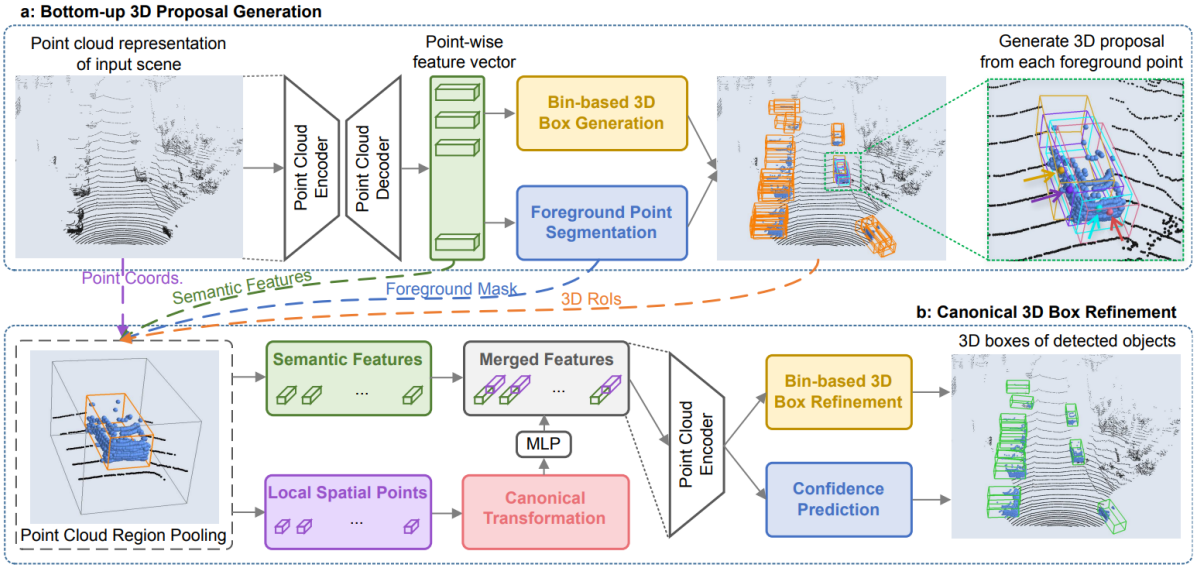

Figure 2.3.: *PointRCNN architecture for 3D object detection. (Adapted from Shi, X. Wang, et al. 2019)*

Figure 2.3 , therefore, provides a comprehensive visual summary of how raw point cloud data is processed through two distinct stages to generate and then refine 3D object detections.

2.3.9. PV-RCNN (PointVoxel-RCNN)

PV-RCNN (PointVoxel-RCNN): Developed by Shi et al. Z. Liu et al. 2019, PV-RCNN is a two-stage framework designed for accurate 3D object detection that deeply integrates both a 3D voxel CNN and PointNet-based set abstraction modules.

- The architecture begins with a 3D voxel CNN backbone, built upon sparse convolution layers, which processes the voxelized point cloud to extract multi-scale semantic features and generate initial 3D object proposals via a Region Proposal Network (RPN) Z. Liu et al. 2019.

- To bridge the voxel and point representations and enhance proposal accuracy, Farthest Point Sampling (FPS) selects a set of keypoints from the raw point cloud. These keypoints are then enriched with semantic information through a novel Voxel Set Abstraction (VSA) module, which encodes multi-scale voxel-CNN features at the keypoint level Z. Liu et al. 2019.

- Following this, keypoint-to-grid RoI feature abstraction modules are employed to generate highly accurate, proposal-aligned features from the keypoints for each RoI. These refined RoI features are subsequently fed into a refinement network, which predicts the precise size and spatial location of each 3D proposal box Z. Liu et al. 2019.
- Finally, two sibling sub-networks are introduced at the last stage: one responsible for confidence prediction and the other for refining the proposal outputs Z. Liu et al. 2019. PV-RCNN effectively combines the efficiency and high-quality proposal generation of voxel CNNs with the flexible receptive fields and accurate localization capabilities of PointNet-based networks.

PV-RCNN++: An advancement over PV-RCNN by Shi, C. Wang, Z. Zhang, et al. 2022, PV-RCNN++ introduces key improvements aimed at enhancing both detection accuracy and computational efficiency.

- The upgraded model proposes two major innovations: the Sectorized Proposal-Centric (SPC) strategy for keypoint sampling and the VectorPool aggregation module for more effective feature aggregation. These modules replace the earlier voxel-to-keypoint scene encoding and keypoint-to-grid RoI feature abstraction modules used in PV-RCNN Shi, C. Wang, Z. Zhang, et al. 2022.
- Recognizing that Farthest Point Sampling (FPS) can be time-consuming and may sample many unnecessary background keypoints, PV-RCNN++ implements the SPC keypoint sampling approach. This method strategically focuses on important neighboring regions surrounding the 3D proposals, thereby significantly reducing the number of sampled keypoints while retaining critical information relevant to objects Shi, C. Wang, Z. Zhang, et al. 2022.
- To further enhance the richness of extracted features with lower resource consumption, the VectorPool aggregation module is introduced. It preserves the spatial arrangement of points within local neighborhoods by splitting the local space into regular voxels and concatenating their features, ensuring that position-sensitive local features are captured both during voxel set abstraction and within the RoI-grid pooling module Shi, C. Wang, Z. Zhang, et al. 2022.
- These refinements lead to PV-RCNN++ being more than twice as fast as the original PV-RCNN while also achieving better performance, particularly on large-scale datasets like the Waymo Open Dataset Shi, C. Wang, Z. Zhang, et al. 2022. This iterative improvement within a successful architectural family highlights an ongoing refinement process in the field, addressing specific bottlenecks like the computational cost of FPS.

2.4. Image-Based 3D Object Detection

While LiDAR provides direct 3D measurements, its cost and sometimes lower resolution compared to cameras have motivated research into image-based 3D object detection. Cameras are ubiquitous, less expensive, and provide rich texture and color information, which can be complementary to LiDAR's geometric data.Qian et al. 2020 The development of pseudo-LiDAR techniques, especially end-to-end trainable versions, represents a significant effort to leverage these advantages by using camera data to emulate LiDAR input for 3D detectors.

2.4.1. Stereo Vision and Depth Estimation

Stereo camera systems, comprising two or more cameras with known relative poses, can perceive depth by triangulating corresponding points identified in their respective image planes. The accuracy of this depth estimation is fundamental to subsequent 3D object detection.

- **PSMNet (Pyramid Stereo Matching Network):** Proposed by Chang et al. 2018, PSMNet is a deep learning architecture designed for high-accuracy stereo depth estimation. It has demonstrated strong performance on benchmarks like KITTI.
  - **Spatial Pyramid Pooling (SPP):** This module aggregates contextual information from the input image features at multiple scales and locations. It helps disambiguate matches in ill-posed regions (e.g., textureless surfaces, repetitive patterns) by incorporating global context Chang et al. 2018.
  - **3D CNN for Cost Volume Regularization:** Left and right image features (enhanced by SPP) are used to construct a 4D cost volume representing matching costs across different disparity levels. A stacked hourglass 3D CNN then processes this cost volume to regularize it and regress the final disparity map Yu Wang et al. 2024.

  In the context of this thesis, PSMNet is utilized to generate dense depth maps from stereo image pairs provided by the Kitti dataset.

- **Challenges in Stereo Depth Estimation:** The accuracy of stereo-derived depth can degrade significantly with increasing distance to objects, in regions with uniform or repetitive textures (making correspondence matching difficult), and in the presence of occlusions or adverse weather conditions Yu Wang et al. 2024. These inaccuracies directly impact the quality of any pseudo-LiDAR data generated from them.

2.4.2. Pseudo-LiDAR Techniques

Pseudo-LiDAR is a paradigm that aims to bridge the gap between image-based perception and LiDAR-based 3D detection pipelines. It involves converting depth maps, estimated from monocular or stereo images, into 3D point cloud representations that mimic the output of a LiDAR sensor. These "pseudo-point clouds" can then be fed into existing LiDAR-based 3D object detectors Qian et al. 2020.

2.4.3. Transformation to Pseudo-Point Clouds

Once a dense depth map $Z(u, v)$ is obtained for an image, where (u, v) are pixel coordinates, it is transformed into a 3D pseudo-point cloud. Each pixel (u, v) with a valid depth value Z_{uv} is projected into a 3D point (X, Y, Z) in the camera's coordinate system using the camera intrinsic parameters (focal lengths f_x, f_y and principal point c_x, c_y):

$$X = \frac{(u - c_x) \cdot Z_{uv}}{f_x}$$

$$Y = \frac{(v - c_y) \cdot Z_{uv}}{f_y}$$

$$Z = Z_{uv}$$

If necessary, these 3D points in the camera coordinate system can be further transformed into the LiDAR coordinate system or a common vehicle coordinate system using the known extrinsic calibration parameters between the camera and the target frame. This results in a pseudo-LiDAR point cloud that can be processed by 3D object detectors designed for LiDAR input. End-to-End Pseudo-LiDAR for Image-Based 3D Object Detection (Qian et al., 2020) introduces a framework that enables end-to-end training of the entire pipeline, from the depth estimation network to the final 3D object detector.

- **Methodology:** This is achieved by introducing differentiable Change of Representation (CoR) modules between the depth estimator and the 3D detector. These CoR modules, such as differentiable subsampling for point-based detectors or soft quantization for voxel-based detectors, allow gradients from the 3D detection loss to be backpropagated through to the depth estimation network Qian et al. 2020.
- **Benefit:** By training the system end-to-end with a joint loss function (combining depth estimation loss and 3D detection loss), the depth estimation network is guided to produce depth maps that are not just generally accurate but are specifically optimized for the downstream task of 3D object detection. This often leads to improved depth estimates around object boundaries, which are crucial for accurate localization, resulting in better overall detection performance compared to pipelines where depth estimation and detection are trained independently Qian et al. 2020.
- **Implementation in this thesis:** This thesis implements and retrains a pipeline on the KITTI dataset based on this end-to-end methodology.
- **Advantages of Pseudo-LiDAR:** The primary advantage is cost-effectiveness, as it relies on cameras which are significantly cheaper than LiDAR sensors Qian et al. 2020. It also allows leveraging the extensive research and mature architectures developed for LiDAR-based 3D detection.
- **Challenges of Pseudo-LiDAR:** The performance of pseudo-LiDAR systems is fundamentally limited by the accuracy of the input depth estimation Qian et al. 2020. Generated pseudo-point clouds can be sparser, noisier, and less accurate in terms of depth precision compared to real LiDAR data, especially for distant, small, or occluded objects. Furthermore, a "domain gap" or distribution shift can exist between real LiDAR point clouds (on which many detectors are pre-trained) and generated pseudo-LiDAR point clouds, potentially degrading detection performance if not properly addressed.

  The quality of pseudo-LiDAR is thus intrinsically linked to the capabilities of the depth estimator and the effectiveness of the end-to-end training in refining depth for the specific task. The concept of differentiable CoR modules is a key enabler, allowing task-specific optimization of earlier stages in multi-stage, multi-modal pipelines.

2.5. Comparison of Classical Geometric vs. Deep Learning 3D Object Detection Approaches

This section provides a comparative analysis between classical geometric methods and deep learning methods for 3D object detection, highlighting their key differences in various aspects

as summarized in Table 2.1. Understanding these distinctions is crucial for appreciating the advancements in the field, particularly with the rise of deep learning.

The fundamental differences between traditional geometric methods and modern deep learning approaches for 3D object detection are summarized in Table 2.1. Classical geometric methods rely on manual feature engineering, demanding developers to explicitly design geometric features, making them less dependent on large labeled datasets, and generally yielding lower accuracy, particularly in complex or noisy scenes where they are sensitive to variations. These methods are more interpretable due to their explicit geometric operations, can have lower computational costs for simpler algorithms, but require significant re-engineering for new tasks, often involving lengthy parameter tuning. In contrast, deep learning methods automatically learn hierarchical features directly from raw data, which typically requires extensive labeled datasets for training. These approaches achieve significantly higher accuracy and are more robust to noise, occlusions, and diverse object shapes, generalizing better across various conditions. While often considered "black boxes" due to their complex internal decision-making, deep learning models are more flexible and adaptable to new tasks via transfer learning or fine-tuning, despite their training being computationally expensive. Ultimately, deep learning methods are better equipped to handle high object density and complex scene interactions compared to their classical counterparts.

Table 2.2 presents a comparative overview of key deep learning architectures developed for point cloud processing. These models—such as PointNet, PointNet++, DGCNN, VoxelNet, and SparseConvNets—represent distinct strategies for handling unstructured 3D data. Each architecture introduces unique mechanisms to address challenges like permutation invariance, local feature learning, or computational efficiency. For instance, while PointNet is foundational and efficient, it lacks the ability to capture fine-grained local structures—an issue addressed by PointNet++ and DGCNN. Voxel-based methods like VoxelNet and SparseConvNets leverage volumetric representations and sparse computations for enhanced spatial context and efficiency. This table serves as a foundational reference for selecting suitable architectures based on specific task requirements and computational constraints.

2.6. Frameworks and Tools for 3D Object Detection

The rapid advancement in 3D object detection has been significantly aided by the development of open-source frameworks and tools that provide standardized implementations, pre-trained models, and robust development environments. These resources lower the barrier to entry for researchers and promote reproducibility.

2.6.1. OpenPCDet

- **OpenPCDet:** OpenPCDet is a clear, simple, and self-contained open-source project developed in PyTorch, specifically designed for LiDAR-based 3D object detection OpenMMLab 2020. It serves as the official code release for several influential and state-of-the-art 3D detection models, including PointRCNN open-mmlab 2020, Part-A2-Net, PV-RCNN, Voxel R-CNN, and PV-RCNN++.

Design Principles

The framework is built upon principles of data-model separation, utilizing unified point cloud coordinates and a consistent 3D bounding box definition (*x, y, z, dx, dy, dz, heading*). This promotes flexibility and makes it easier to extend the framework to custom datasets and new models OpenMMLab 2020. Its model structure is designed to be clear and adaptable, accommodating both one-stage and two-stage detection architectures that can be seeing here at OpenMMLab 2020.

This common 7-parameter representation for a 3D bounding box defines its pose and dimensions in a 3D space:

- **(*x, y, z*):** These typically represent the **center coordinates** of the 3D bounding box.
  - **x:** The coordinate along the X-axis (often corresponding to depth or forward direction in a vehicle's coordinate system, but this depends on the sensor setup).
  - **y:** The coordinate along the Y-axis (often corresponding to height or vertical direction).
  - **z:** The coordinate along the Z-axis (often corresponding to width or lateral direction).
  - **Important Note:** While (*x, y, z*) commonly refers to the geometric center, some datasets or frameworks might define it as the bottom center of the box. OpenPCDet standardizes this internally, usually aligning with the geometric center.
- **(*dx, dy, dz*):** These represent the **dimensions** of the 3D bounding box along its local axes.
  - **dx:** The dimension of the box along its local X-axis (often length).
  - **dy:** The dimension of the box along its local Y-axis (often height).
  - **dz:** The dimension of the box along its local Z-axis (often width).
  - The specific assignment of length, width, and height to *dx, dy, dz* depends on the convention adopted by the dataset (e.g., KITTI, Waymo, NuScenes) and how OpenPCDet handles it for each. Generally, *dx* is length, *dy* is height, and *dz* is width for vehicles in KITTI-like setups, but it's crucial to confirm the exact order for your specific dataset.
- **heading:** This parameter represents the **orientation** or yaw angle of the 3D bounding box around the vertical axis (typically the Y-axis in most LiDAR coordinate systems).
  - It's usually an angle in radians, defining the rotation of the box from a reference direction (e.g., positive X-axis) in the ground plane.
  - This allows the box to be arbitrarily rotated in the horizontal plane, capturing the orientation of the .

- **Features:** OpenPCDet supports distributed training and testing across multiple GPUs and machines. It allows for multiple detection heads operating at different scales to detect

various object classes. The framework includes implementations for various data augmentation techniques, RoI pooling methods, and GPU-accelerated 3D IoU calculations and Non-Maximum Suppression (NMS) OpenMMLab 2020. It provides a rich model zoo with pre-trained weights for popular datasets like KITTI, Waymo Open Dataset, and NuScenes.

- **Usage in Thesis:** In this research, OpenPCDet serves as the primary backbone for implementing, training, and benchmarking the selected LiDAR-based deep learning models: PV-RCNN, PointPillars, and PointRCNN. Its comprehensive toolkit facilitates consistent evaluation across these diverse architectures OpenMMLab 2020.

  For experiments on LiDAR and pseudo point clouds, Pointrcnn is utilized. Specifically for pseudo point clouds, the messages originating from the rosbag are modified before being processed by the ros2 node, effectively transforming them into pseudo LiDAR points through an imitation process.

2.6.2. Robot Operating System (ROS 2) for Real-Time Systems

- **ROS 2:** The Robot Operating System (ROS) is a widely adopted, flexible framework for writing robot software. ROS 2, the second generation of ROS, was re-architected to address limitations of ROS 1, particularly concerning real-time performance, security, and use in multi-robot systems and commercial products Open Robotics 2025a.
  - **Nodes:** The fundamental processing units in a ROS 2 system. Each node is typically responsible for a specific task, such as reading sensor data, processing information, or controlling actuators. In this thesis, nodes handle LiDAR data ingestion, point cloud preprocessing, and 3D object detection Open Robotics 2025c.
  - **Topics:** Named communication channels (buses) over which nodes exchange data using a publish-subscribe mechanism. For example, a LiDAR driver node publishes point cloud data to a topic, and a detection node subscribes to that topic to receive the data Open Robotics 2025c.
  - **Messages:** Typed data structures used for communication over topics. A standard message type for LiDAR data is `sensor_msgs/msg/PointCloud2`, which can represent 3D point coordinates, intensity, and other per-point attributes Open Robotics 2025a. Detection results are often published using custom or standard messages like `vision_msgs/msg/Detection3DArray`.
  - **Services:** A request-response communication pattern suitable for remote procedure calls where a client expects a direct reply from a server.
  - **Actions:** Used for long-running, preemptable tasks that provide feedback during execution, such as navigation goals.
  - **Real-Time Capabilities:** ROS 2 is designed with real-time systems in mind, leveraging the Data Distribution Service (DDS) as its underlying communication middleware Open Robotics 2025a. DDS provides features like configurable Quality of Service (QoS) policies, which are crucial for ensuring timely and reliable data delivery in autonomous systems.
  - **Usage in Thesis:** A key component of this research is the development of a real-time ROS 2 pipeline designed to ingest Velodyne LiDAR data (from ROS bags)

and perform 3D object detection using the implemented deep learning models. This allows for the evaluation of models not just in terms of offline accuracy but also their practical performance within a robotics framework Open Robotics 2025c.

- **ROS Bags:** ROS bags are a standard file format (`.bag`) for storing ROS message data that has been published on topics Open Robotics 2025b. They are invaluable for robotics research and development as they allow for the recording of complex sensor streams (like LiDAR scans, camera images, IMU data) and system states during experiments. This recorded data can then be replayed multiple times, providing consistent input for developing, testing, and debugging algorithms offline, without requiring the physical robot or sensors to be active. In this thesis, both the proprietary Astemo LiDAR data and the public KITTI dataset were utilized in the ROS bag format. This standardized format facilitates consistent data handling and allows the developed ROS 2 pipeline to be rigorously tested and validated by replaying identical scenarios for each detection method. The `ros2 bag` command-line tools are used for recording, replaying, and inspecting these bag files Open Robotics 2025b. The use of ROS bags is critical for ensuring reproducible experiments and for systematically evaluating the perception pipeline's performance across different datasets and models.

- **Summary:** The adoption of these open-source frameworks—OpenPCDet for deep learning model development and ROS 2 for system integration and real-time execution—significantly accelerates research by providing robust, well-tested components and standardized communication protocols. This allows researchers to focus on novel contributions rather than re-implementing foundational infrastructure.

2.6.3. Docker for Reproducible Environments

- **Docker:** Docker is a platform that leverages containerization to package software and its dependencies into standardized units called containers, which can run reliably across different computing environments Docker 2020.
  - **Images:** Docker images are read-only templates that contain the application code, runtime, system tools, libraries, and configurations needed to run a container. In this research, custom Docker images were built to include specific versions of CUDA, PyTorch, OpenPCDet, and ROS 2, ensuring consistency across development, testing, and deployment environments Docker 2020.
  - **Containers:** Containers are runtime instances of Docker images. They provide isolated environments where applications can run without interference from the host system. By running the 3D detection pipelines inside Docker containers, the same code and dependencies produce identical results on any machine, simplifying collaborative development and deployment Docker 2020.
  - **Dockerfile:** A `Dockerfile` is a text file that contains instructions for building a Docker image. In this thesis, separate Dockerfiles were created for:
    - **Model Development Container:** Includes PyTorch, OpenPCDet, and necessary Python libraries for training and benchmarking LiDAR detection models.

    - **ROS 2 Runtime Container:** Based on an official ROS 2 Docker image, extended with the required LiDAR drivers, custom ROS 2 nodes, and OpenPCDet inference modules for real-time detection Docker 2020.

- **Docker Compose:** Docker Compose Docker 2020is a tool for defining and running multi-container Docker applications. A `docker-compose.yml` file was created to orchestrate multiple services, including:
    - **LiDAR Data Publisher:** A ROS 2 node that reads ROS bags and publishes point cloud data.
    - **Detection Service:** A separate container running the deep learning inference engine using OpenPCDet.
    - **Visualization Node:** A container that runs RViz2 for real-time visualization of detection results.

  Using Docker Compose ensures that all services start in the correct order, with proper network configurations and shared volumes for ROS bag replay.

- **Benefits:**
    - **Reproducibility:** By encapsulating the entire software stack (OS libraries, frameworks, and custom code) in Docker images, experiments can be reproduced exactly on any machine that supports Docker Docker 2020.
    - **Portability:** Docker containers can run on local workstations, cloud instances, or high-performance computing clusters without modification Docker 2020.
    - **Isolation:** Dependencies and environment settings for LiDAR detection and ROS 2 do not conflict with host system libraries or other projects Docker 2020.
    - **Continuous Integration/Deployment (CI/CD):** Docker images can be built and tested automatically using CI pipelines (e.g., GitHub Actions), ensuring that changes to code do not break the environment or dependencies Docker 2020.

Table 2.1.: *Comparison of Geometric vs. Deep Learning 3D Object Detection Approach*

Aspect	**Classical Geometric Methods**	**Deep Learning Methods**
Feature Engineering	Manual, relies on handcrafted geometric features	Automatic, learns hierarchical features directly from data.Trigka et al. 2025
Data Dependency	Less dependent on large labeled datasets	Typically requires large amounts of labeled data for training.Sapkota et al. 2025
Accuracy	Generally lower, especially in complex scenes	Generally higher, state-of-the-art performance on benchmarks.Trigka et al. 2025
Robustness	Sensitive to noise, occlusions, and diverse object shapes	More robust to variations, better generalization capabilities.Trigka et al. 2025
Interpretability	More interpretable, operations based on explicit geometry	Often considered "black boxes," harder to interpret internal decision-making.Vinodkumar et al. 2024
Computational Cost	Can be lower for simpler algorithms or less complex scenes	Training is computationally expensive; inference can be optimized but often higher.
Adaptability to New Tasks	Requires significant re-engineering or parameter tuning	Can be adapted via transfer learning or fine-tuning, more flexible.Lang et al. 2019
Development Time	Can involve lengthy tuning of heuristic parameters	Model development and training can be time-consuming, but frameworks accelerate this.
Handling Complexity	Struggles with high object density and complex interactions	Better equipped to handle complex scenes and subtle object cues.

Table 2.2.: *Summary of Key Deep Learning Architectures for Point Cloud Processing*

Architecture	**Key Idea/Methodology**	**Strengths**	**Limitations**	**Key References**
PointNet	Processes raw points individually with shared MLPs; uses max pooling for permutation invariance and global feature aggregation.	Pioneer for direct point cloud processing; efficient; learns global shape features; permutation invariant.Qi, Su, et al. 2017b	Does not capture local geometric structures effectively.Qi, Su, et al. 2017b	Qi, Su, et al. 2017b
PointNet++	Hierarchically applies PointNet on local regions (Set Abstraction layers: sampling, grouping, PointNet) to capture multi-scale local features.	Captures local and global features at multiple scales; robust to non-uniform sampling.Shi, C. Wang, Z. Zhang, et al. 2022	More complex than PointNet; computation can increase with hierarchy depth.	Shi, C. Wang, Z. Zhang, et al. 2022
DGCNN	Constructs dynamic graphs (k-NN in feature space) at each layer; EdgeConv operation learns edge features between points and their neighbors.	Explicitly models local neighborhood relationships; captures evolving local structures dynamically.Phan et al. 2018	Graph construction can be computationally intensive; k-NN search can be a bottleneck.	Phan et al. 2018
VoxelNet	Partitions point cloud into 3D voxels; VFE layers learn features within each voxel; 3D CNNs aggregate voxel features.	Leverages 3D CNNs for contextual reasoning; end-to-end trainable for detection.Zhou et al. 2018	Can be computationally expensive with fine voxel resolution; potential information loss from voxelization.	Zhou et al. 2018
SparseConvNets	Performs convolutions only on active (non-empty) sites in sparse data (e.g., voxelized point clouds); Submanifold Sparse Conv preserves sparsity.	Greatly improves computational and memory efficiency for 3D CNNs on sparse data.B. Liu et al. 2015	Implementation can be more complex than dense convolutions.	B. Liu et al. 2015

3. System Architecture and Implementation

This chapter details the design and development of the experimental systems employed in this thesis. It begins with an overview of the comprehensive 3D object detection system, followed by descriptions of the classical geometric baseline, the real-time ROS2 pipeline for LiDAR and Pseudo-LiDAR data processing, the implementation of LiDAR-based deep learning detection models, and also image-based pseudo-LiDAR detection module. The chapter concludes with specifications of the software and hardware environment used for development and experimentation.

The overall system is intentionally designed with modularity to facilitate the comparative benchmarking that forms a core component of this research.

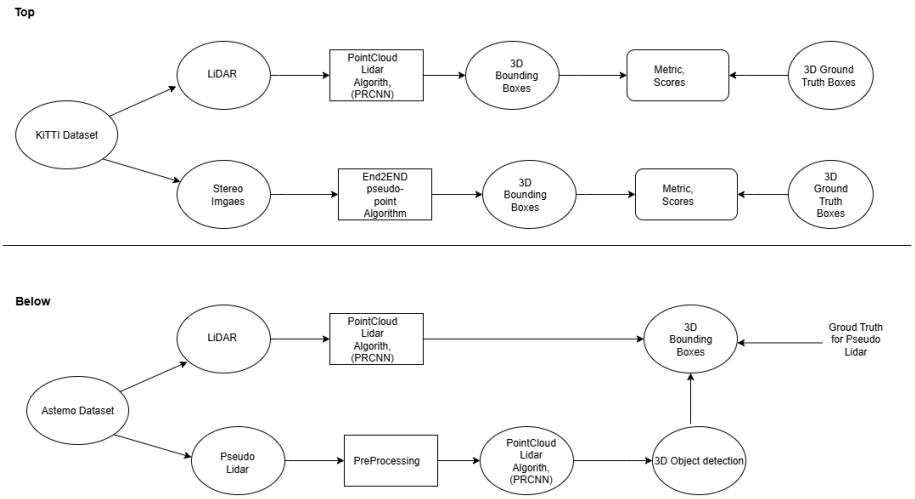

Figure 3.1.: *Overview of the architecture.*

3.1. Methodology Overview

This research employs a structured methodology to evaluate the performance of LiDAR-based versus pseudo-LiDAR-based 3D object detection across distinct datasets. For the KITTI benchmark, the methodology bifurcates: (1) native LiDAR point clouds processed via OpenPCDet framework using PointRCNN for 3D bounding box generation, and (2) stereo images processed through an end-to-end pseudo-point cloud algorithm before PointRCNN detection. Both pipelines are evaluated against KITTI's 3D ground truth using mean Average Precision (mAP) metrics, enabling direct comparison between sensing modalities. Visualization is implemented

with OpenPCDet by integration in ROS environment and evaluation was also done with OpenPCDet. For the proprietary ASTEMO dataset lacking ground truth annotations, native LiDAR detections (via PointRCNN) serve as reference baseline. Concurrently, pseudo-LiDAR data extracted from ROS bags undergoes structural preprocessing to approximate authentic point clouds before being processed by the same PointRCNN model. Frame-by-frame evaluation computes mAP scores by comparing pseudo-LiDAR detections against LiDAR-derived proxy ground truth, establishing relative performance metrics on this dataset. PointRCNN serves as the consistent detection architecture throughout all experimental workflows.

3.2. Classical Geometric Methods: Initial Setup (Astemo)

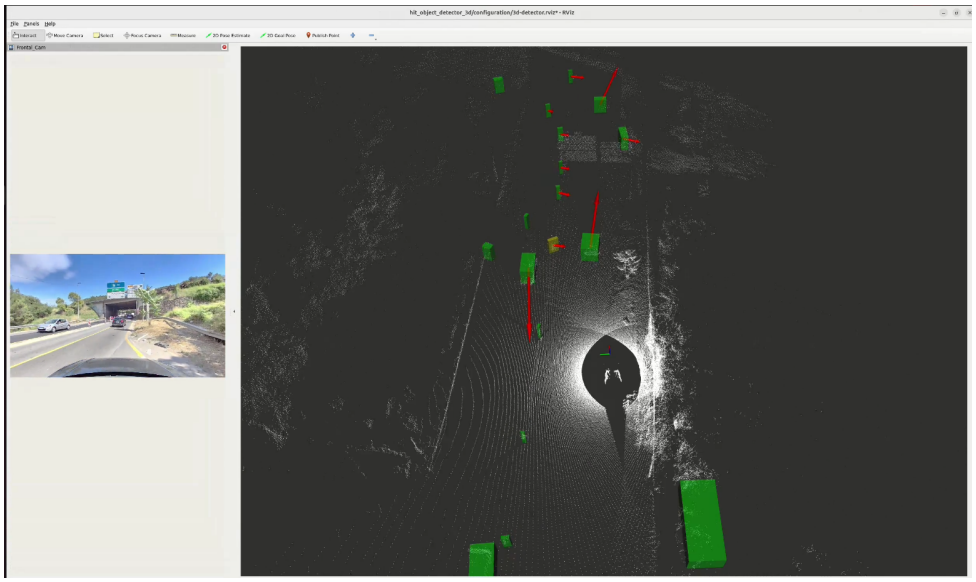

Figure 3.2.: *Astemo Dataset Baseline.*

The initial experimental setup provided by Astemo depicted in Figure 3.2 , which served as the impetus for modernizing the 3D object detection workflow, was based on traditional geometry-driven techniques. This classical pipeline processed LiDAR data from the proprietary Astemo dataset, which was provided in ROS bag format. While specific algorithmic details of Astemo's proprietary implementation were not fully disclosed, the approach is characteristic of classical methods prevalent before the dominance of deep learning. Generally, such pipelines involve a sequence of steps:

- **Preprocessing:** Raw point clouds might undergo filtering to remove noise or irrelevant points (e.g., points too far or too close, or outside a defined region of interest).
- **Ground Segmentation:** Points belonging to the ground plane are often removed to simplify the scene and isolate potential objects.
- **Clustering/Segmentation:** The remaining non-ground points are clustered based on spatial proximity (e.g., using Euclidean clustering or DBSCAN as prosped by Deng 2020) to group points that likely belong to individual objects.

- **Feature Extraction & Shape Fitting:** For each cluster, geometric features (e.g., size, elongatedness, point density) might be extracted. Subsequently, predefined shapes (typically 3D bounding boxes or cuboids) are fitted to these clusters to represent detected objects. This fitting process might involve techniques like principal component analysis (PCA) for orientation estimation or RANSAC for robust fitting. Derpanis 2010

- **Classification (Optional):** Simple rule-based classifiers or traditional machine learning models (trained on limited features) might be used to assign class labels to the fitted shapes.

The outputs from this classical method, when applied to the Astemo dataset, served as a baseline. This allows for a direct quantitative comparison against the deep learning-based methods developed in this thesis, thereby concretely measuring the improvements achieved through the modernization effort on this specific industrial dataset. This baseline is crucial for contextualizing the performance gains offered by the more contemporary approaches.

3.3. Real-Time ROS2 Pipeline for LiDAR Data Processing

To facilitate the implementation and evaluation of modern deep learning-based 3D object detection models in a manner relevant to autonomous systems, a real-time pipeline was developed using the Robot Operating System 2 (ROS2). ROS2 was chosen for its suitability for complex robotics applications, its support for real-time communication, and its growing adoption in the automotive and research communities.

3.4. LiDAR-based 3D Object Detection Module

This module forms the core of the LiDAR-based perception system, integrating state-of-the-art deep learning models for 3D object detection. The OpenPCDet framework was chosen as the primary library for implementing and training these models due to its comprehensive nature, support for numerous leading architectures, and its open-source availability. The selection of PV-RCNN, PointPillars, and PointRCNN covers a diverse range of architectural philosophies in LiDAR-based 3D detection—specifically, voxel-based two-stage, pillar-based single-stage, and point-based two-stage approaches, respectively. This diversity allows for a robust and comprehensive comparison of their capabilities.

3.4.1. Integration with OpenPCDet Framework

The OpenPCDet framework was set up according to its official documentation on the docker environment I built. This involved installing necessary dependencies, including PyTorch, and specific versions of libraries like `spconv` for sparse convolutions.

- **Configuration:** OpenPCDet uses YAML configuration files as shown here in appendix A.4 to define dataset paths, model architectures, training parameters, and evaluation settings. For each model (PointRCNN) and each dataset (Kitti, Astemo), specific configuration files were created or adapted from the defaults provided by OpenPCDet. This included specifying data loading parameters, data augmentation strategies, model-specific hyperparameters, and loss function configurations.

- **Data Loading:** OpenPCDet provides flexible data loaders that can be customized for different dataset formats. For Kitti, the standard data loader was used as shown here in this appendix A.9

- **Training and Evaluation Scripts:** OpenPCDet includes scripts for training models from scratch or fine-tuning pre-trained models, as well as for evaluating trained models on test or validation sets. These scripts were utilized for the experimental procedures described in Chapter 4.

Leveraging OpenPCDet's pre-trained models, especially for the KITTI dataset, and its well-structured training infrastructure significantly reduced development time. This ensured that the model implementations were benchmarked against established standards and allowed the research to focus on the comparative analysis and integration aspects rather than re-implementing complex architectures from scratch, which can be error-prone and time-intensive.

3.4.2. Inference Workflow

Testing for Kitti dataset

- The models were trained using the testing scripts and configuration system provided by OpenPCDet as shown here in appendix A.7. For the KITTI dataset, testing often started from pre-trained model weights available in the OpenPCDet model zoo, used in this thesis.

- Data augmentation techniques provided by OpenPCDet (e.g., random flipping, global scaling, global rotation, ground-truth sampling) were employed during training to improve model robustness and generalization.

Inference

- Within the ROS2 3D Object Detection Node, the trained model weights (checkpoint files) were loaded using OpenPCDet's model loading utilities.

- When a `sensor_msgs/msg/PointCloud2` message is received on the subscribed topic, the point cloud data is extracted and converted into the NumPy array format expected by the OpenPCDet model input layer. This typically involves extracting (x, y, z, intensity) values for each point.

- The formatted point cloud is then passed through the loaded model for a forward pass (inference).

- The model outputs detection results, usually as a list of predicted 3D bounding boxes with associated class labels and confidence scores. These predictions are often in a specific coordinate system or format defined by the model.

- Post-processing steps, such as Non-Maximum Suppression (NMS), are applied to filter out redundant or low-confidence detections.

- The final set of 3D bounding boxes is then converted into the chosen ROS2 message format (e.g., a custom array of 3D boxes) and published on the output topic for consumption by other nodes (like RViz2 for visualization or downstream planning modules). The coordinates of the bounding boxes are ensured to be in a consistent frame of reference (e.g., the LiDAR sensor frame or the vehicle's `base_link` frame) using `tf2` transformations if necessary.

3.4.1. Implementation and Integration ROS 2

Figure 3.3.: *Overview of the ROS rqt graph.*

This subsection documents the concrete implementation work carried out for this thesis and clarifies how the different software components were integrated into a single, runnable ROS 2 pipeline. The principal engineering contribution as shown in Figure 3.3 is the development of a ROS 2 node (`/openpcddet`) that subscribes to raw Velodyne scans, converts incoming `sensor_msgs/PointCloud2` messages to the array format required by the OpenPCDet inference pipeline, performs GPU-accelerated inference with a PRCNN model, and republishes the detection results as `vision_msgs/Detection3DArray` messages on the topic `/cloud_detection`. An excerpt of the primary callback implementing point-cloud conversion is provided in Appendix A.3. The complete, reproducible environment (Dockerfile) used to build the container image is given in Appendix A.1, and the container launch script is included in Appendix A.2.

The end-to-end data flow implemented in this work is summarized as follows:

- **Sensor acquisition.** The Velodyne driver node publishes raw point clouds on the topic `/velodyne_sensor` (message type `sensor_msgs/PointCloud2`).
- **Preprocessing & conversion.** The `/openpcddet` node receives the `PointCloud2` message, converts it to a NumPy array using `ros2_numpy`, and reformats the point coordinates and features into the structure expected by OpenPCDet (see Appendix A.3).
- **Inference.** The reformatted point cloud is passed to the PRCNN inference pipeline (loaded once at node startup).The launch file can accessed (Appendix A.6)
- **Postprocessing & publication.** Detection outputs are converted to `Detection3DArray` and published on `/cloud_detection` for visualization and logging.

3.4.2. Processing Pseudo Point Clouds in ROS 2

In addition to processing real Velodyne LiDAR scans, my work implements a dedicated preprocessing pipeline for Pseudo-LiDAR point clouds generated from stereo imagery in the Astemo dataset. The motivation for introducing this preprocessing stage is twofold: (i) to ensure that

pseudo point clouds meet the input format and quality requirements of the PointRCNN model used in OpenPCDet, and (ii) to separate dataset-specific noise handling from the core detection node, thereby improving modularity and maintainability of the ROS 2 pipeline.

The processing is performed by a custom ROS 2 node, `/filtered_PC`, which subscribes to raw pseudo point clouds published from the ROS bag (`/pseudo_lidar_points`) and applies a sequence of preprocessing steps before forwarding the sanitized data to the detection node `/openpcddet`. The complete message flow for this setup is illustrated in Figure 3.4, generated using `rqt_graph`. This node acts as a pre-inference filter in the pipeline, ensuring that only valid, normalized, and size-compliant point sets are passed to the detection stage.

My work implemented preprocessing workflow is shown in Figure 3.5, and comprises the following stages and the code snippet can be found in appendix A.8:

- **Sanitization.** All points with `NaN` coordinates are removed to prevent downstream computational errors and to maintain geometric validity of the input.
- **Feature handling.** For real LiDAR data, the raw intensity channel is preserved. For pseudo-LiDAR data, which lacks true intensity values, an artificial intensity is generated from the Euclidean distance of each point from the sensor origin, normalised to the expected intensity range.
- **Subsampling.** If the number of points exceeds the model's maximum supported input size, random subsampling is applied to reduce the set while preserving spatial distribution.
- **Size normalization.** If the number of points falls below the model's minimum requirement, duplicate points are appended until the minimum is reached, ensuring that the batch dimensions match those expected by the model during inference.
- **Output formatting.** The final point set is represented as an $(N \times \text{num\_features})$ `float32` array, typically containing four features per point: $x, y, z,$ intensity.

After preprocessing, the filtered point cloud is published to topic `/filtered_PC`, from which `/openpcddet` subscribes for inference. Detection results are subsequently published to `/cloud_detection` in the `vision_msgs/Detection3DArray` format for visualization and evaluation.

This modular arrangement is consistent with the principle of separation of concerns in ROS 2 system design, allowing the preprocessing stage to be modified or replaced without altering the detection node implementation. It also improves system transparency, as each transformation stage is explicitly represented in the ROS computation graph.

Figure 3.4.: *`rqt_graph` showing message flow for Pseudo-LiDAR detection: `/pseudo_lidar_points` → `/filtered_PC` → `/openpcddet` → `/cloud_detection`.*

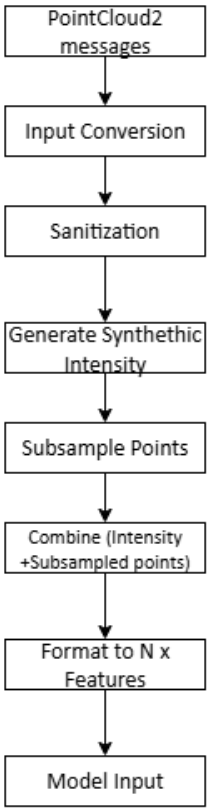

Figure 3.5.: *Flowchart of the implemented pseudo point cloud preprocessing pipeline in the* `/filtered_PC` *node.*

3.5. Image-based 3D Object Detection Module (Pseudo-LiDAR)

The image-based detection pipeline implements the End-to-End Pseudo-LiDAR framework Qian et al. 2020 using the reference implementation from Mileyan's open-source repository[0]. This approach transforms stereo imagery from the KITTI dataset Geiger et al. 2012a into pseudo-LiDAR point clouds, which are subsequently processed by PointRCNN Shi, X. Wang, et al. 2019 for 3D object detection. My deployment of end-to-end pseudo lidar required significant adaptation to contemporary hardware and software environments due to version incompatibilities in the original codebase. The legacy implementation targeted PyTorch 1.1.0 with CUDA 10 dependencies, while the current system operates on CUDA 12.4 with PyTorch 2.8. This discrepancy manifested in three primary challenges: (1) deprecated PyTorch APIs causing forward/backward compatibility failures, (2) Incompatibilities during PointNet++ compilation due to C++ standard mismatches, and (3) broken dependencies for ancillary libraries for example use of *`align_corners=True`, which is the default before PyTorch 0.4.0.* but has to be adopted for newer version of pytorch because we have explicitly make it to True or else we get low recall because of bounding box does not align during evalution thus giving us low recall. .

To resolve these issues, the I conducted the following modifications: (1) Refactored core PyTorch operations to maintain computational graph integrity while complying with modern syntax, (2) Reconfigured PointNet++ compilation with C++17 standard and updated CUDA archi-

tecture flags, and (3) Implemented dependency patching through Docker environment isolation. The solution encapsulates the entire workflow in a containerized environment with custom base images maintaining legacy library support while enabling GPU acceleration. This adaptation represents a significant contribution, as it enables the execution of legacy pseudo-LiDAR frameworks on contemporary hardware without compromising detection accuracy. The joint training of depth map and also pointrcnn training code snippet can be seeing here(see appendix A.10).

3.6. Software and Hardware Environment

The development, training, and evaluation of the systems described in this thesis were conducted using the following software and hardware environment. Precise specification of this environment is essential for ensuring the **reproducibility** of the research, as deep learning models, particularly those involving 3D data and real-time inference, are often sensitive to software versions and hardware capabilities.

3.6.1. Software Environment

- **Operating System:** Ubuntu 20.04 LTS
- **ROS2 Distribution:** ROS2 *Humble Hawksbill*
- **Programming Languages:**
  - Python (used for deep learning pipelines and integration with OpenPCDet)
  - C++ (used for any performance-critical ROS2 nodes or LiDAR drivers)
- **Deep Learning Framework:** PyTorch (e.g., version 1.10, consistent with OpenPCDet compatibility)
- **Key Libraries and Toolkits:**
  - OpenPCDet (e.g., v0.5.0 or v0.6.0) – used for 3D object detection
  - CUDA Toolkit (version 11.x)
  - cuDNN (version 8.x)
  - spconv (sparse convolution library, e.g., v1.2.1 or v2.x)
  - OpenCV (for image processing in the pseudo-LiDAR pipeline)
  - Point Cloud Library (PCL) (for custom point cloud operations if applicable)

Hardware Configuration

– **CPU:** Intel Core i7-10700K @ 3.8GHz

– **GPU:** NVIDIA GeForce RTX 4080 with 16GB VRAM

– **RAM:** 32 GB DDR4 @ 3200 MHz

[0] `https://github.com/mileyan/pseudo-LiDAR_e2e/tree/master/PointRCNN`

4. Experimental Setup

This chapter outlines the methodologies employed for conducting the experiments in this thesis. It provides detailed descriptions of the datasets used, including the proprietary Astemo LiDAR dataset and the public KITTI benchmark.

4.1. Datasets

The empirical evaluation in this thesis relies on two distinct LiDAR datasets: a proprietary dataset provided by Astemo and the widely used public KITTI dataset. The use of both types of datasets allows for a comprehensive assessment of model performance, testing generalizability across different sensor characteristics, environmental conditions, and annotation styles. This dual-dataset approach is a significant strength, as it grounds the research in both industry-specific contexts and publicly verifiable benchmarks.

4.1.1. Proprietary Astemo LiDAR Dataset

The Astemo dataset is a unique collection of LiDAR and pseudo point cloud data provided by Astemo for the purpose of this research. Due to its proprietary nature, specific details are shared to the extent permissible while maintaining confidentiality, focusing on aspects relevant for scientific understanding and reproducibility, as guided by best practices for handling such data.

- **Sensor Configuration:**
  - **LiDAR Model:** The data was collected using Velodyne LiDAR sensor(s). The exact model (e.g., Velodyne VLP-32C or HDL-64E) was not disclosed by the company.
  - **Camera Model:** The data was collected using a stereo camera setup. The exact model information was not disclosed by the company.
  - **Mounting:** The LiDAR sensor and stereo camera were mounted on top of an experimental vehicle.
- **Data Collection Scenarios and Environment:** The dataset comprises sequences recorded in diverse real-world driving scenarios. The scenes primarily consist of urban streets with varying traffic densities, including both parking scenarios and moving traffic conditions.Proprietary Astemo ROS bags containing temporally aligned Velodyne LiDAR scans and pseudo-LiDAR point clouds for the same sequences (company-provided). Each recorded sequence was approximately 1 minute and 15 seconds in duration and included continuous message streams on both the `/velodyne_sensor` and `/pseudo_lidar_points` topics. Ground-truth human annotations were not available for these proprietary sequences; therefore, LiDAR-based PointRCNN outputs are used as the reference labels for evaluation.

– **Ethical Considerations and Data Handling:** All data handling and usage strictly adhered to the confidentiality and data sharing agreements established with Astemo. Results published in this thesis are presented in an aggregated and anonymized manner, focusing on model performance metrics without revealing specific raw data or sensitive contextual details that could compromise Astemo's proprietary interests.

The careful characterization of the Astemo dataset, balancing necessary scientific detail with confidentiality requirements, is crucial. It allows the research community to understand the context of the results derived from this unique dataset and to appreciate the challenges and opportunities it presents compared to public benchmarks.

4.1.2. Public KITTI Dataset

The KITTI Vision Benchmark Suite is a widely adopted public dataset in autonomous driving research, offering data for various tasks including stereo vision, optical flow, visual odometry, and 3D object detection. For this thesis, the 3D object detection benchmark was utilized.

– **Overview:** The KITTI dataset was collected in and around Karlsruhe, Germany, covering diverse scenarios such as urban environments, rural roads, and highways Geiger et al. 2012b.

– **Sensor Setup:** The data collection platform was equipped with a Velodyne HDL-64E LiDAR sensor, color and grayscale stereo cameras, a high-precision GPS/IMU inertial navigation system, and other sensors Geiger et al. 2012b.

– **Relevant Characteristics for 3D Detection:**

  • The 3D object detection benchmark provides 7,481 training samples (images and corresponding point clouds) and 7,518 testing samples Geiger et al. 2012b.

  • 3D bounding box annotations are provided for object classes including *Car*, *Van*, *Truck*, *Pedestrian*, *Person_ sitting*, *Cyclist*, *Tram*, and *Misc* Geiger et al. 2012a. For this thesis, the primary focus is typically on *Car*, *Pedestrian*, and *Cyclist*.

  • Annotations include 3D bounding box dimensions (length, width, height), 3D location (x,y,z), rotation around the y-axis (yaw), object class, truncation level, and occlusion state Geiger et al. 2012b.

  • The data, including point clouds (from the Velodyne HDL-64E), is available in various formats. For this thesis, data was used in or converted to the ROS bag format, containing `sensor_msgs/msg/PointCloud2` messages for LiDAR scans and synchronized image data for the pseudo-LiDAR experiments.

– **Data Preprocessing and Formatting:**

  • The KITTI data was prepared for use with the OpenPCDet framework. This involved ensuring point clouds were in the standard `.bin` format (x, y, z, intensity) and labels were in the KITTI text format.

  • Standard training/validation splits were used to ensure comparability with published results. Commonly, the training data (7481 samples) is split into a train set (e.g., 3712 samples) and a val set (e.g., 3769 samples) as done by many researchers and supported by OpenPCDet. The official test set labels are held out by KITTI for online leaderboard submissions.

4.2. Evaluation Metrics

4.2.1. Evaluation Metrics and Training Details

The performance of the 3D object detection models developed and benchmarked in this thesis was primarily evaluated using mean Average Precision (mAP), a standard metric in object detection tasksBuilt In 2025..

Mean Average Precision (mAP) for 3D Object Detection

- **Definition:** Mean Average Precision (mAP) is the average of Average Precision (AP) values calculated for each object class and often across multiple Intersection over Union (IoU) thresholds Built In 2025.
- **Average Precision (AP):** For a single class, AP summarizes the shape of the precision-recall curve. It is the weighted mean of precisions achieved at each threshold, with the weight being the increase in recall from the prior threshold. Alternatively, it can be calculated as the area under the precision-recall curve Built In 2025.
  - **Precision:** The fraction of correct detections among all detections made by the model for a given class. Calculated as TP/(TP+FP), where TP is the number of True Positives and FP is the number of False Positives Built In 2025.
  - **Recall:** The fraction of actual ground truth objects that were correctly detected by the model. Calculated as TP/(TP+FN), where FN is the number of False Negatives Built In 2025.
- **Calculation Method:** For the KITTI benchmark, AP is typically calculated using 40 recall sampling points (an update from the earlier 11-point interpolation used in PASCAL VOC) to provide a more accurate summary of the precision-recall curve. This method was adopted for evaluations in this thesis *3D Object Detection Evaluation 2017* 2017.

3D Intersection over Union (IoU)

The 3D IoU measures the overlap between a predicted 3D bounding box and a ground truth 3D bounding box. It is defined as the volume of their intersection divided by the volume of their union:

$$\text{IoU}_{3D} = \frac{\text{Volume}(\text{Box}_{\text{predicted}} \cap \text{Box}_{\text{ground truth}})}{\text{Volume}(\text{Box}_{\text{predicted}} \cup \text{Box}_{\text{ground truth}})}$$

A detection is considered a True Positive (TP) if its 3D IoU with a ground truth box of the same class exceeds a certain threshold, and that ground truth box has not already been matched with another detection of higher confidence. Otherwise, it may be considered a False Positive (FP).

- **KITTI IoU Thresholds:**
  - Car: 0.7 (i.e., 70% overlap)
  - Pedestrian: 0.5 (i.e., 50% overlap)
  - Cyclist: 0.5 (i.e., 50% overlap)

– **Astemo IoU Thresholds:** For consistency, similar IoU thresholds were adopted for the Astemo dataset, particularly 0.7 for the 'Car' class, which is the primary focus.

Handling Difficulty Levels (e.g., Kitti's Easy, Moderate, Hard)

The KITTI benchmark defines three difficulty levels for evaluating detections, based on the object's bounding box height in the 2D image (pixels), its occlusion level, and its truncation level:

– Easy: Minimum bounding box height: 40 px, Maximum occlusion: Fully visible, Maximum truncation: 15%.

– Moderate: Minimum bounding box height: 25 px, Maximum occlusion: Partly occluded, Maximum truncation: 30%.

– Hard: Minimum bounding box height: 25 px, Maximum occlusion: Difficult to see (highly occluded), Maximum truncation: 50%.

Results are typically reported for all three difficulty levels. The 'Moderate' difficulty level is the primary criterion for ranking on the official KITTI leaderboard. This multi-level evaluation provides a more nuanced understanding of model performance, revealing robustness to challenging conditions like small or occluded objects. For the Astemo dataset, while formal difficulty levels might not have been predefined in the same way, results were analyzed to understand performance on objects of varying apparent sizes and occlusion states where possible.

4.3. Experimental Design and Goals

The experiments in this chapter were designed to systematically evaluate the performance of the PointRCNN architecture Shi, X. Wang, et al. 2019 in detecting 3D objects from both real Velodyne LiDAR and pseudo-LiDAR point clouds, within the unified ROS 2 pipelines developed in section 3.4. The core aim is to determine the impact of sensor modality and preprocessing on detection accuracy, using mean Average Precision (mAP) at an IoU threshold of 0.70 for the car class, following the KITTI evaluation protocol.

All experiments were conducted within a reproducible GPU-enabled Docker environment (Appendix A.1), ensuring consistent software versions, dependency resolution, and hardware utilization across runs. The only exception is the end-to-end pseudo-LiDAR pipeline described in Section 3.5, which was executed in an isolated Conda environment to accommodate legacy CUDA and PyTorch dependencies required for compatibility with the original framework.

The experimental design isolates the effects of:

- **Sensor modality:** real Velodyne LiDAR vs. pseudo-LiDAR derived from stereo imagery.
- **Preprocessing:** direct input vs. filtered point clouds via the `/filtered_PC` node.
- **Domain shift:** training on one modality (real LiDAR) and testing on another (pseudo-LiDAR).

The message flows and computational graphs for each pipeline configuration are shown in Figures 3.1 and 3.4 in Chapter 3.4.1. The pseudo-LiDAR preprocessing workflow is detailed in Figure 3.5.

4.3.1. Experiment 1: Baseline Detection with Real LiDAR

Hypothesis: The *PointRCNN* model, when applied to high-fidelity Velodyne LiDAR data within our implemented ROS 2 pipeline, is hypothesized to achieve 3D detection accuracy (mAP) on the KITTI "Car" class (IoU = 0.7) comparable to established benchmarks. This will validate the integrity of our baseline implementation and provide a reliable upper-bound performance metric for comparison against the pseudo-LiDAR modality.

Research question link: This experiment forms the baseline for addressing the overarching question of this thesis: *Is pseudo-LiDAR better than LiDAR for 3D object detection?* Establishing the performance of the detection pipeline on real LiDAR data provides the reference point against which all pseudo-LiDAR experiments will be compared.

Objective: Measure the detection accuracy of PointRCNN when operating on high-quality Velodyne LiDAR point clouds without any preprocessing. This quantifies the "best case" performance of the developed ROS 2 detection pipeline.

Pipeline configuration: As implemented in Chapter 3.4.1 and shown in Figure 3.3, raw point clouds published on the `/velodyne_sensor` topic are subscribed to by the `/openpcddet` node. The node converts the incoming `sensor_msgs/PointCloud2` message into the NumPy array format required by OpenPCDet, performs inference with a pre-trained PointRCNN model (see Appendix A.8), and publishes results to the `/cloud_detection` topic.

Execution environment: All steps were run inside the unified GPU-enabled Docker container described in Appendix A.1, ensuring consistent dependencies and reproducibility.

Dataset: KITTI 3D object detection dataset (car class).

Evaluation and visualization: During runtime, detection results were visualized in RViz2 for qualitative inspection of bounding box alignment with the scene geometry. Quantitative performance was obtained by saving detection outputs to disk and applying the official KITTI evaluation script integrated into the OpenPCDet framework, yielding the mAP score for the car class at an IoU threshold of 0.70 and 0.50.

Rationale: This configuration is expected to yield the highest mAP score, as Velodyne point clouds have native intensity values, low noise, and minimal artefacts. The results from this baseline serve as the upper bound for subsequent comparisons with pseudo-LiDAR configurations in later experiments.

4.3.2. Experiment 2: End-to-End Pseudo-LiDAR 3D Object Detection

Hypothesis: It is hypothesized that the end-to-end trained pseudo-LiDAR pipeline can achieve effective, albeit lower, 3D object detection performance compared to real LiDAR. A significant performance gap is predicted, particularly for smaller object classes such as "Pedestrian" and "Cyclist," due to inherent inaccuracies in stereo-derived depth maps.

Research question link: This experiment contributes directly to answering the central question of this thesis *Is pseudo-LiDAR better than LiDAR for 3D object detection?* by implementing and evaluating the End-to-End pseudo-LiDAR 3D detection framework on the KITTI dataset. The results are compared against the real LiDAR baseline established in Experiment 4.3.1.

Objective: Evaluate the performance of stereo-derived pseudo-LiDAR point clouds, jointly trained with depth maps, using the pointrcnn detector as described in Qian et al. 2020. This configuration incorporates the paper's dedicated preprocessing pipeline, enabling a direct assessment of the potential of pseudo-LiDAR for 3D object detection.

Pipeline configuration: Stereo images from the Kitti dataset were converted into pseudo-LiDAR point clouds using the depth estimation and point cloud generation steps defined in the End-to-End pseudo-LiDAR framework. These point clouds underwent preprocessing as described in the original paper including depth map filtering and point cloud normalization — before being jointly used with depth maps for training a Pointrcnn detector. Both training and evaluation were carried out using the scripts provided in the official End-to-End framework implementation.

Execution environment: The entire pseudo-LiDAR generation, preprocessing, training, and evaluation process was executed inside the same GPU-enabled Docker container described in Appendix A.10, with additional dependencies installed for stereo matching, depth map generation, and point cloud processing. CUDA and PyTorch version compatibility were maintained according to the requirements of the End-to-End framework.

Dataset: KITTI 3D object detection dataset, where stereo imagery was used to generate pseudo-LiDAR point clouds and ground-truth annotations were used for training and evaluation (see Section 4.1.2).

Evaluation and visualization: Quantitative evaluation used the official mAP computation script provided in the End-to-End pseudo-LiDAR repository. Results were reported for the car class at an IoU threshold of 0.70. Qualitative inspection of detection outputs was performed using the Open3D visualization library, as the KITTI dataset does not provide pseudo-LiDAR point clouds in ROS bag format and OpenPCDet lacks a native pseudo-LiDAR module for this workflow.

Rationale: This experiment isolates the performance of pseudo-LiDAR generated from stereo imagery under the optimal conditions described in the End-to-End framework. By comparing these results with the real LiDAR baseline (Experiment 4.3.1), the influence of sensor modality on detection performance can be directly quantified.

4.3.3. Experiment 3: ROS-based Comparison using Astemo Data (LiDAR-derived Pseudo Ground Truth)

Hypothesis: For the proprietary Astemo dataset, it is hypothesized that the pseudo-LiDAR pipeline will exhibit a quantifiable performance degradation when evaluated against the high-fidelity detections trained on kitti dataset from the real LiDAR stream serving as a reference. This experiment is designed to measure the degree of agreement between the two modalities, predicting that despite preprocessing via the dedicated `/filtered_PC` node, pseudo-LiDAR will underperform in terms of mAP across all object classes.

Research question link: This experiment provides an application-oriented comparison that contributes to the central thesis question *Is pseudo-LiDAR better than LiDAR for 3D object detection?*by evaluating pseudo-LiDAR detections against detections obtained from the real Velodyne LiDAR on the same temporal sequences supplied by the Astemo ROS bags.

Objective: Quantify the agreement between pseudo-LiDAR and real-LiDAR detections when both are produced from the same sensor run. Specifically, PointRCNN is applied to the Velodyne LiDAR stream to produce high-quality detections that are used as *pseudo ground truth* for evaluating the pseudo point cloud detections on the identical sequence. This experimental design isolates modality and preprocessing effects while avoiding the need for manual labelling of proprietary data.

Pipeline configuration: The experiment uses the ROS 2 computation graph implemented in Chapter 3.4.1 (see Figure 3.4). The Astemo ROS bag publishes two synchronized streams for

the same time sequence: the real Velodyne scans on `/velodyne_sensor` and pseudo point clouds on `/pseudo_lidar_points`. The pipeline executed is:

- **LiDAR detection (pseudo-GT generation):** `/velodyne_sensor` → `/openpcddet` (PointRCNN) → `/cloud_detection_lidar`. The detection node produces 3D bounding boxes that are recorded and considered the reference labels for the corresponding timestamps.
- **Pseudo-LiDAR preprocessing and detection:** `/pseudo_lidar_points` → `/filtered_PC` (sanitization, synthetic intensity, subsampling, size-normalization; see Fig. 3.5) → `/openpcddet` (PointRCNN) → `/cloud_detection_pseudo`.
- **Matching & evaluation:** The two detection streams `/cloud_detection_lidar` and `/cloud_detection_pseudo` are timestamp-aligned and paired. Pseudo-LiDAR detections are evaluated against the LiDAR-derived detections as the reference using the same IoU criterion and mAP scoring used elsewhere in this thesis.

Execution environment: All ROS 2 nodes, data playback (rosbag), and the detection pipeline were executed inside the GPU-enabled Docker container described in Appendix A.1. The container ensured that OpenPCDet, PointRCNN, and ROS 2 dependencies were consistent with the implementation described in Chapter 3.4.1.

Dataset: Proprietary Astemo ROS bags containing temporally aligned Velodyne LiDAR scans and pseudo-LiDAR point clouds for the same sequences (company-provided). Ground-truth human annotations were not available for these proprietary sequences; therefore LiDAR-based PointRCNN outputs are used as the reference labels for evaluation (see for more details refer to section 4.1.1).

Evaluation and visualization:

- **Quantitative:** Pseudo-LiDAR detections were compared to LiDAR-derived detections using mean Average Precision (mAP) at IoU = 0.70 for the car class. To ensure fairness, only detections with closely aligned timestamps were considered.
- **Qualitative:** During live playback of the Astemo rosbag, detection streams were visualized in RViz2 to inspect spatial alignment and temporal consistency between LiDAR-derived and pseudo-LiDAR detections (see Figure 3.4 and the RViz screenshots in Figure 5.5). RViz was possible here because the Astemo data were provided as ROS bags, unlike the KITTI pseudo-LiDAR workflow.

Rationale: Using LiDAR-based PointRCNN detections as proxy ground truth enables a direct, implementation-centered comparison between sensor modalities on proprietary, realistic driving sequences without manual annotation. Differences in mAP quantify the extent to which pseudo-LiDAR (after preprocessing) approximates LiDAR-based detections in operational conditions provided by Astemo.

5. Results and Discussion

This chapter presents the empirical findings of the research, detailing the performance of the classical geometric methods and the various deep learning-based 3D object detection models on both the proprietary Astemo dataset and the public KITTI benchmark. The results are analyzed quantitatively using the mAP metric, and qualitatively through visual inspection of detection outputs. A comprehensive discussion interprets these results, comparing the different approaches and modalities, and addressing the research questions posed in Chapter 1.

5.1. Performance of Classical Geometric Methods (Baseline)

The initial experimental setup from Astemo, based on traditional geometry-driven techniques, served as the qualitative baseline for this thesis. A formal quantitative evaluation using mAP was not feasible for this system, as the classical pipeline does not produce the per-detection confidence scores required for a precision-recall analysis.

Instead, performance was assessed through a qualitative analysis of the output, as seen in Figure 5.1. This visual inspection revealed several key limitations that motivated the transition to deep learning:

- **High False Positive Rate:** The geometric methods frequently generated erroneous detections, particularly in cluttered urban environments or when faced with complex, non-vehicular structures.
- **Poor Handling of Occlusion:** The system struggled to identify and place accurate bounding boxes on partially occluded vehicles.
- **Inconsistent Detection:** The detection rate was inconsistent, with objects often being detected in one frame and missed in the next.

These observed failure cases provided clear evidence that the classical approach lacked the robustness required for reliable operation, establishing a performance benchmark that the deep learning models were expected to surpass.

5.2. Performance of LiDAR-based Deep Learning Models on KITTI

To validate the deep learning pipeline(which is also Experiment 4.3.1), the Pointrcnn model was benchmarked on the public Kitti 3D object detection dataset. The performance, measured by Average Precision (AP), confirms the implementation's correctness and aligns with results from established literature. The results, summarized in Table 5.1 and visualized in Figure 5.2, demonstrate the high accuracy of the LiDAR-based approach.

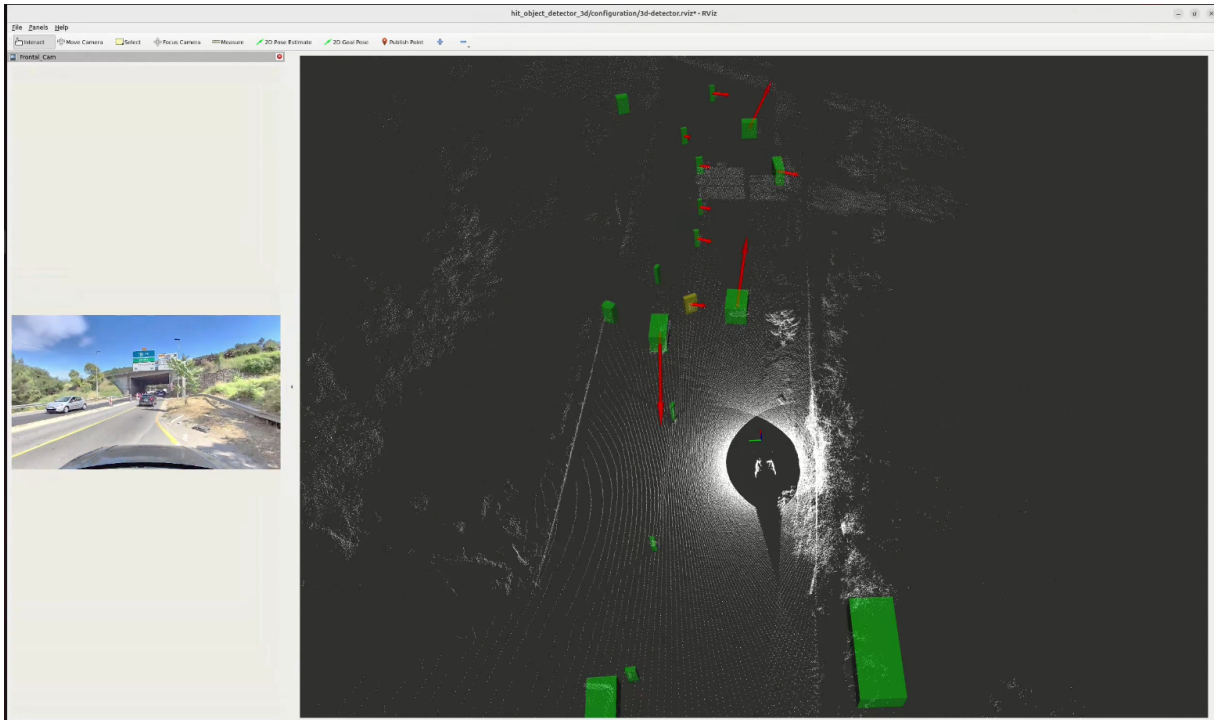

Figure 5.1.: *Visualization of real LiDAR from the ASTEMO Dataset using the classical baseline method.*

Table 5.1.: *Performance of PointRCNN on the Kitti Validation Set (3D AP %). The 'Moderate' category is the primary benchmark.*

Category	IoU Thr.	Easy	Moderate	Hard
Car	0.70	89.08%	78.95%	77.81%
Pedestrian	0.50	65.43%	56.12%	51.34%
Cyclist	0.50	82.11%	65.32%	58.99%

The model achieves a strong AP of **78.95%** for the 'Car' class at the 'Moderate' difficulty level. The high-quality geometric data from the LiDAR sensor enables the model to accurately detect not only cars but also more challenging classes like pedestrians and cyclists, as shown by the detection results in Figure 5.2. These strong results provide a confident, evidence-based benchmark for the subsequent comparative analysis.

5.3. Performance of Image-based Pseudo-LiDAR Detection on KITTI

The end-to-end pseudo-LiDAR pipeline was evaluated on the KITTI dataset to assess the viability of image-based 3D object detection(which is also Experiment 4.3.2). The results, shown in Table 5.2, Figure 5.4 and Figure 5.3, indicate that while this approach is promising, it has clear limitations compared to using real LiDAR.

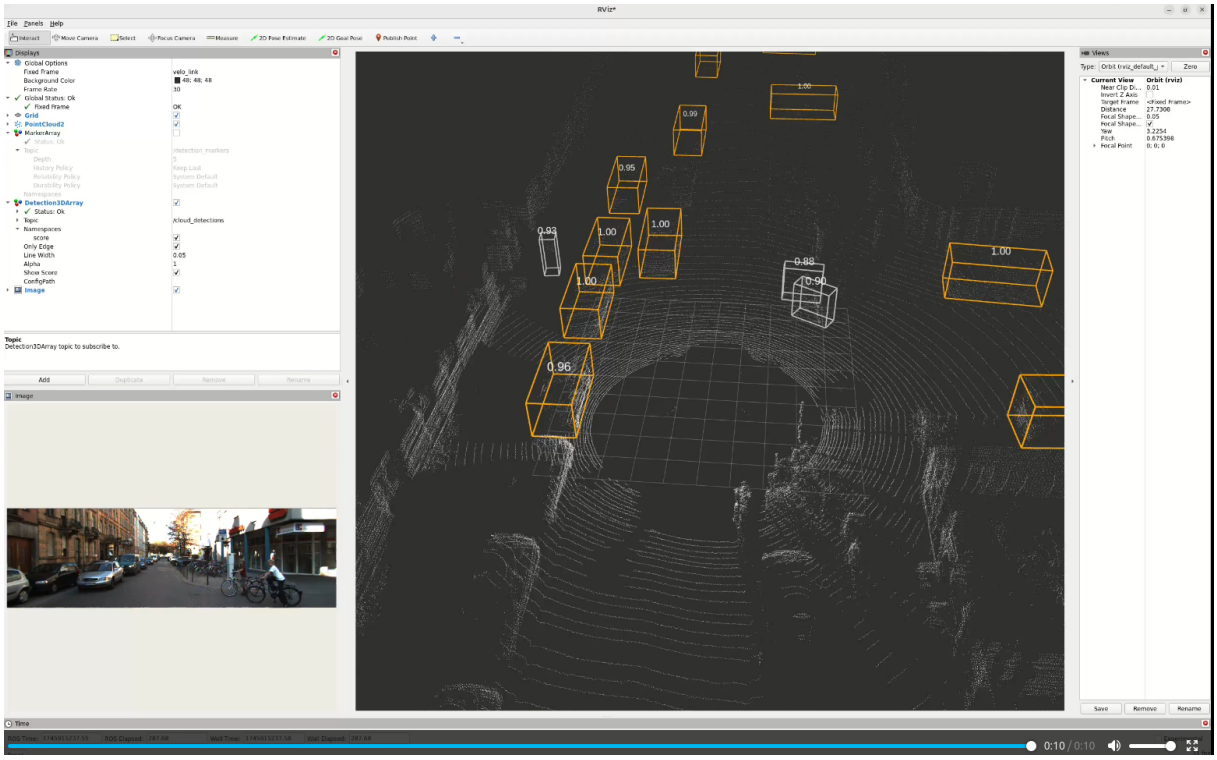

Figure 5.2.: *Visualization of real LiDAR from the ASTEMO Dataset using the classical baseline method.*

Table 5.2.: *Performance of End-to-End Pseudo-LiDAR (E2E-PL) on the KITTI Validation Set (3D AP %).*

Category	IoU Thr.	Easy	Moderate	Hard
Car	0.70	71.7%	53.4%	46.3%
Pedestrian	0.50	38.1%	29.5%	27.8%
Cyclist	0.50	33.6%	22.7%	21.2%

The pipeline performs best on the 'Car' class, achieving **53.4%** AP on the 'Moderate' setting. This demonstrates that pseudo-LiDAR can effectively represent large objects. However, a significant weakness is exposed when detecting smaller classes. The performance for 'Pedestrian' and 'Cyclist' categories is substantially lower, dropping to **29.5%** and **22.7%** respectively for the 'Moderate' difficulty.

5.4. Comparative Analysis of LiDAR and Pseudo-LiDAR on the KITTI Dataset

A direct comparison between the real LiDAR and pseudo-LiDAR pipelines on the KITTI benchmark reveals the significant performance advantage of using direct 3D sensor measurements.

As shown in Table 5.3 and visualized in Figure 5.4, PointRCNN consistently and substantially

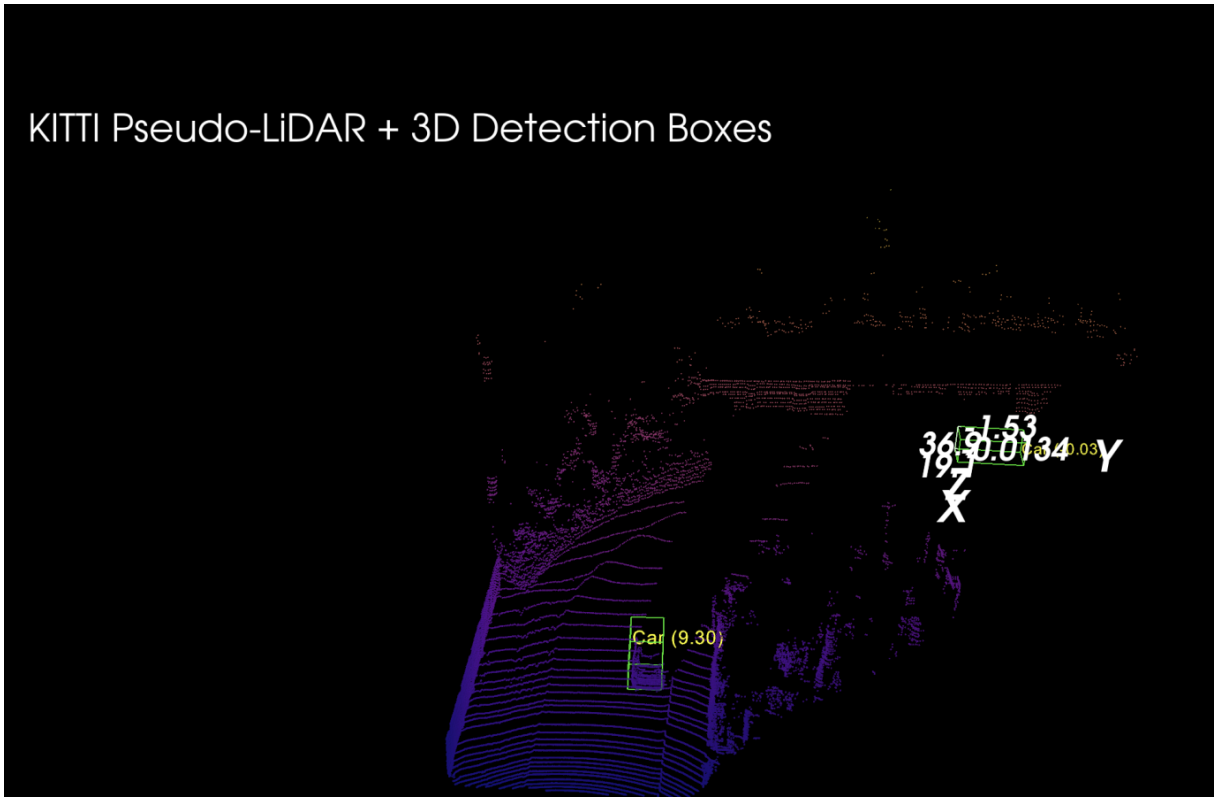

Figure 5.3.: *Visualization of Pseudo-LiDAR from the KITTI Dataset .*

Table 5.3.: *Performance Gap: PointRCNN (LiDAR) vs. E2E-PL (Pseudo-LiDAR) on Moderate Difficulty (3D AP %).*

Category	PointRCNN (LiDAR)	E2E-PL (Pseudo-LiDAR)	Performance Gap
Car	78.95%	53.4%	**-25.55%**
Pedestrian	56.12%	29.5%	**-26.62%**
Cyclist	65.32%	22.7%	**-42.62%**

outperforms the pseudo-LiDAR approach. For the 'Car' class on the 'Moderate' setting, the performance gap is over 25 percentage points. This gap widens dramatically for more challenging classes, exceeding 42 percentage points for 'Cyclist'. These quantitative results strongly support the conclusion that the geometric accuracy of direct LiDAR sensing is superior for robust 3D object detection.

5.5. Performance Analysis on the Astemo Dataset

The final experiments evaluated the pseudo-LiDAR pipeline on the proprietary Astemo dataset (which is also Experiment 4.3.3). Following the recommendation to use a more direct measure of correctness than model confidence, this analysis evaluates detections based on their **Intersection over Union (IoU)** with a reference ground truth.

5.5.1. Methodology: IoU-Based Evaluation

As the Astemo dataset lacks manual ground truth annotations, the high-quality detections from the real LiDAR stream processed by PointRCNN were used as the pseudo ground truth as shown in the two Figures respectively 5.6 and 5.5 for this evaluation. This approach allows for a direct, frame-by-frame comparison of the pseudo-LiDAR's localization accuracy against that of a high-fidelity sensor.

The evaluation process for the 50 selected frames was as follows:

- For each frame, the 3D bounding box predicted by the pseudo-LiDAR pipeline was compared against the corresponding bounding box from the real LiDAR pipeline.
- The 3D Intersection over Union (IoU) was calculated for this pair of boxes.
- A pseudo-LiDAR detection was classified as a **True Positive (TP)** if the IoU was $\geq$ **0.7**, meeting the strict KITTI standard for the 'Car' class.
- If the IoU was $<$ **0.7**, the detection was classified as a **False Positive (FP)**, indicating incorrect localization.
- Frames where the real LiDAR detected an object but the pseudo-LiDAR did not were counted as **False Negatives (FN)**.

5.5.2. Results: Detection Counts

Out of the 50 ground truth objects present in the sequence, the pseudo-LiDAR pipeline produced 44 detections, with 6 instances being missed entirely. The IoU-based analysis of these 44 detections yielded the results summarized in Table 5.4.

Table 5.4.: *Pseudo-LiDAR Detection Performance on 50 Astemo Frames (IoU $\geq$ 0.7).*

Category	Count	Description
True Positives (TP)	12	The detection correctly localized the object.
False Positives (FP)	32	The detection's location was inaccurate (IoU $<$ 0.7).
False Negatives (FN)	6	The model completely failed to detect the object.
Total Objects	**50**	The total number of objects in the evaluated sequence.

5.5.3. mAP Calculation for Pseudo-LiDAR Detections on Pseudo Ground Truth

To quantitatively evaluate the performance of the pseudo-LiDAR pipeline on the Astemo dataset, the mean Average Precision (mAP) was computed using the 40-point interpolation method. The analysis was conducted at an IoU threshold of 0.70 for the *Car* class, consistent with the kitti evaluation protocol.

The pseudo-LiDAR pipeline generated 44 detections across 50 frames. After sorting detections by descending confidence and evaluating against LiDAR-derived ground truth:

- **12 True Positives** (IoU $\geq$ 0.70)

- **32 False Positives** (IoU < 0.70)
- **6 False Negatives** (confidence $= 0.00$)

Performance Metrics: mAP Calculation

To provide a comprehensive, single-figure metric of performance, the **mean Average Precision (mAP)** was calculated. The mAP summarizes the shape of the precision-recall curve and is the standard metric for object detection tasks. The calculation was performed for the 'Car' class at the 0.7 IoU threshold.

Following the Kitti benchmark protocol, the AP was computed using **40 recall sampling points**. The process involves ranking all 44 detections by their confidence scores and incrementally calculating precision and recall. From this data, an interpolated precision-recall curve is generated, and the AP is the mean of the precision values at the 40 standard recall levels.

Results: Detection Performance

The IoU-based analysis of the 50 ground-truth objects present in the evaluated sequence yielded the detection counts summarized in Table 5.4. Out of 44 total detections made by the pseudo-LiDAR system, only 12 as shown in Figure 5.9 were accurately localized, with the remaining 32 being classified as false positives as shown in the Figure 5.11. The system failed to detect 6 as shown in Figure 5.8 entirely. The Detection also failed at places such as parking as shown in the Figure 5.10

Based on this TP/FP/FN distribution, the calculated mean Average Precision (mAP) for the pseudo-LiDAR pipeline on the Astemo dataset is **22.5%** for IoU at 0.70 for the class "car", which is consistent with the Kitti evaluation.

The performance is further illustrated by the precision-recall curve in Figure 5.7. The curve shows that the system achieves high precision only at very low recall levels. After the 12th true positive is detected (at 24% recall), the precision collapses as every subsequent detection is a false positive.

Discussion of Astemo Results

The empirical evidence from the Astemo dataset provides a clear and quantitative assessment of the pseudo-LiDAR system's performance limitations in a realistic operational scenario. The low mAP score of **22.5%** starkly reveals the system's current shortcomings for safety-critical applications.

This poor score is a direct consequence of the model's inability to localize objects accurately. The data in Table 5.4 shows that while the system is capable of detecting the *presence* of an object (only 6 misses out of 50), it fails to determine its correct position and dimensions in the vast majority of cases (32 FPs out of 44 detections) as shown 5.4. This analysis provides stronger evidence than confidence scores alone, confirming that a high confidence score does not guarantee spatial accuracy.

The shape of the precision-recall curve (Figure 5.7) visualizes this failure vividly. The sharp drop in precision after 24% recall indicates that beyond a small number of "easy" detections, the system's output is dominated by localization errors. For autonomous navigation, where precise object location is critical for path planning and collision avoidance, such a high rate of

localization failure renders the system unreliable. These findings, supported by visualizations in your paper, confirm that at its current stage of development, the pseudo-LiDAR modality does not possess the geometric accuracy and reliability of real LiDAR systems.

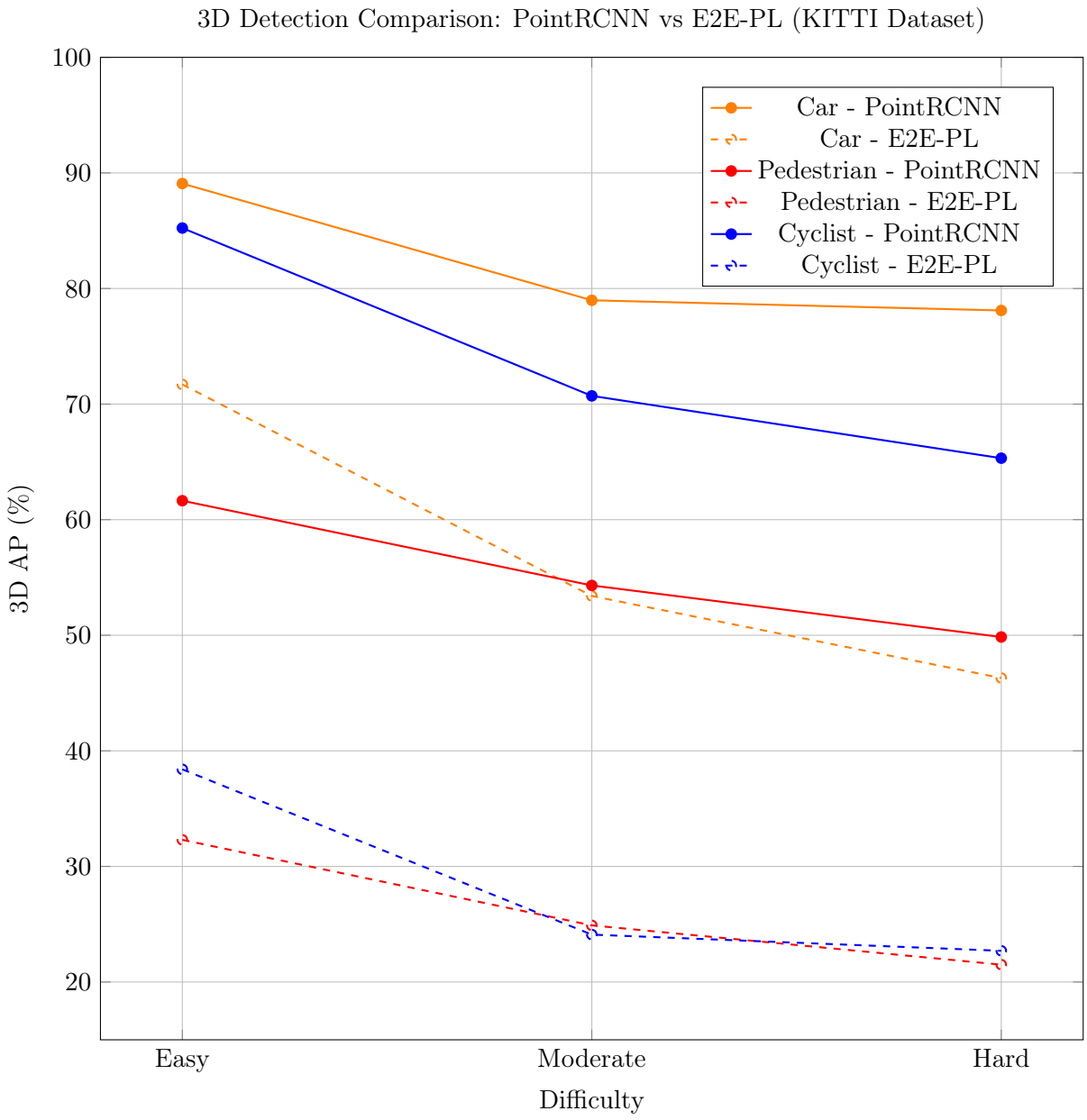

Figure 5.4.: *3D Detection performance of LiDAR and Pseudo-LiDAR across difficulty levels on KITTI dataset*

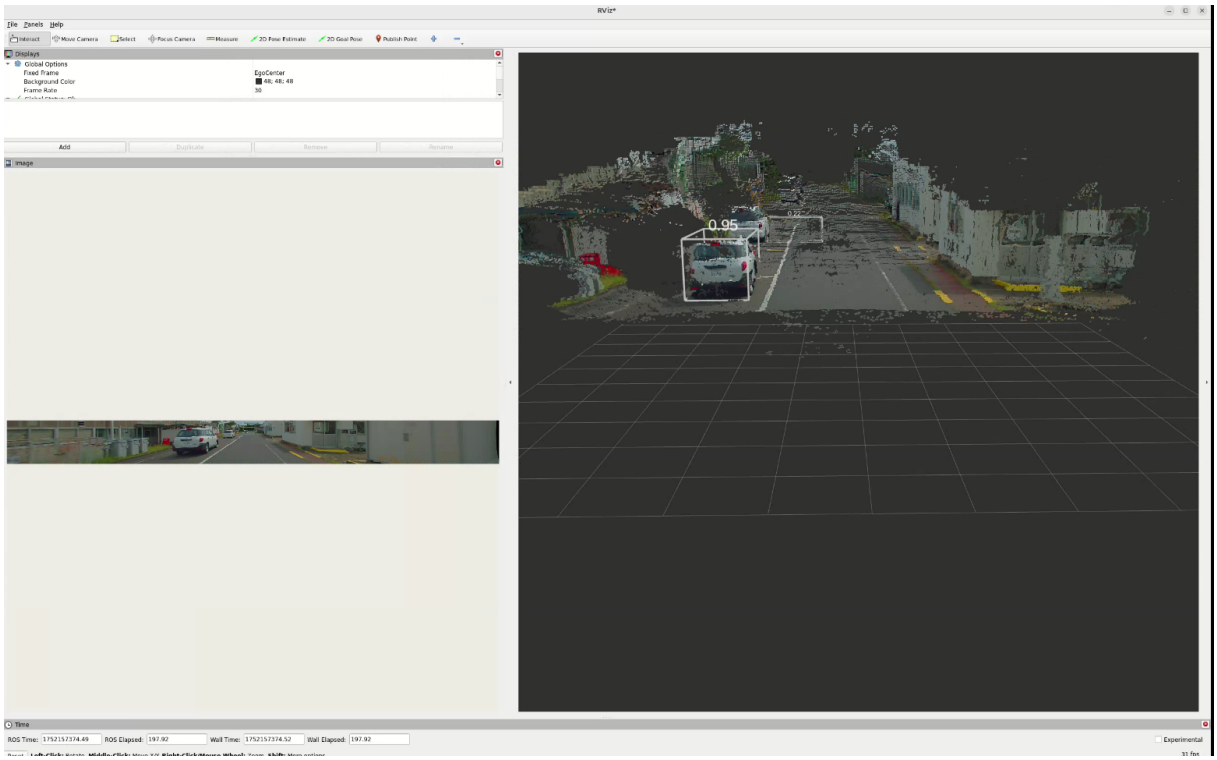

Figure 5.5.: *Visualization of Pseudo-LiDAR from the ASTEMO Dataset .*

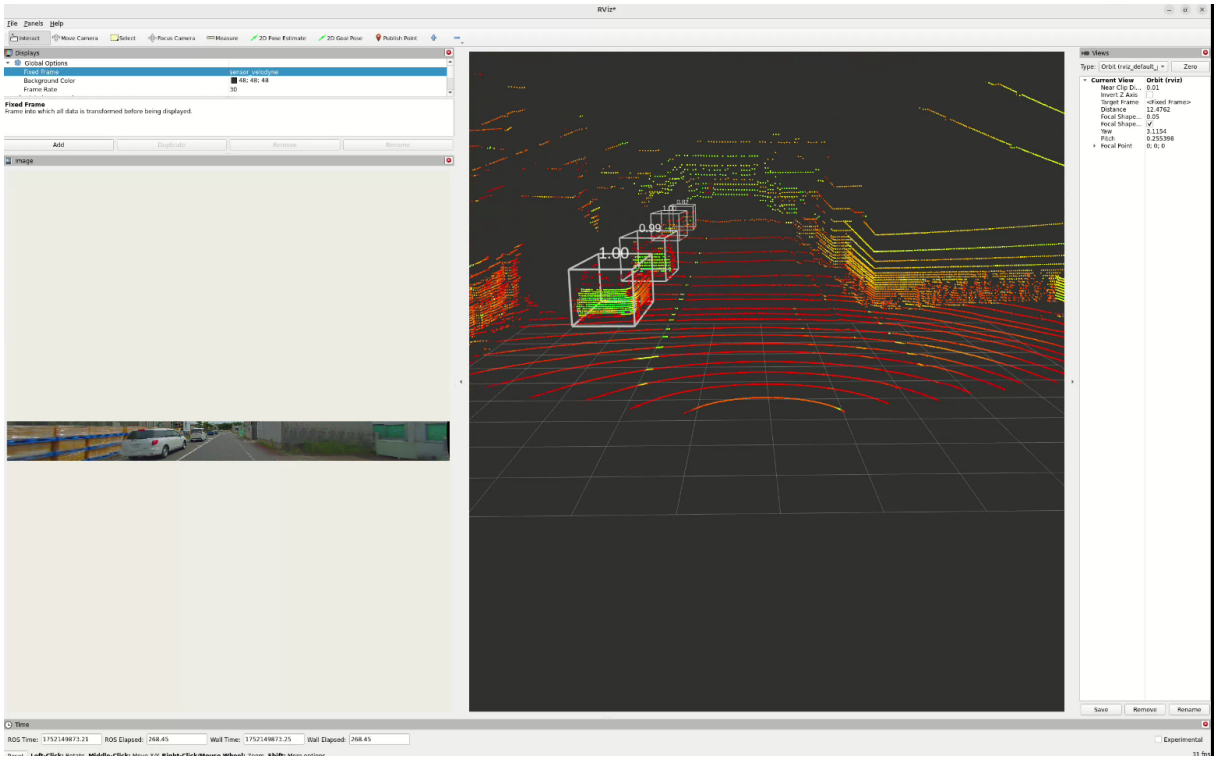

Figure 5.6.: *Visualization of LiDAR from the Astemo Dataset*

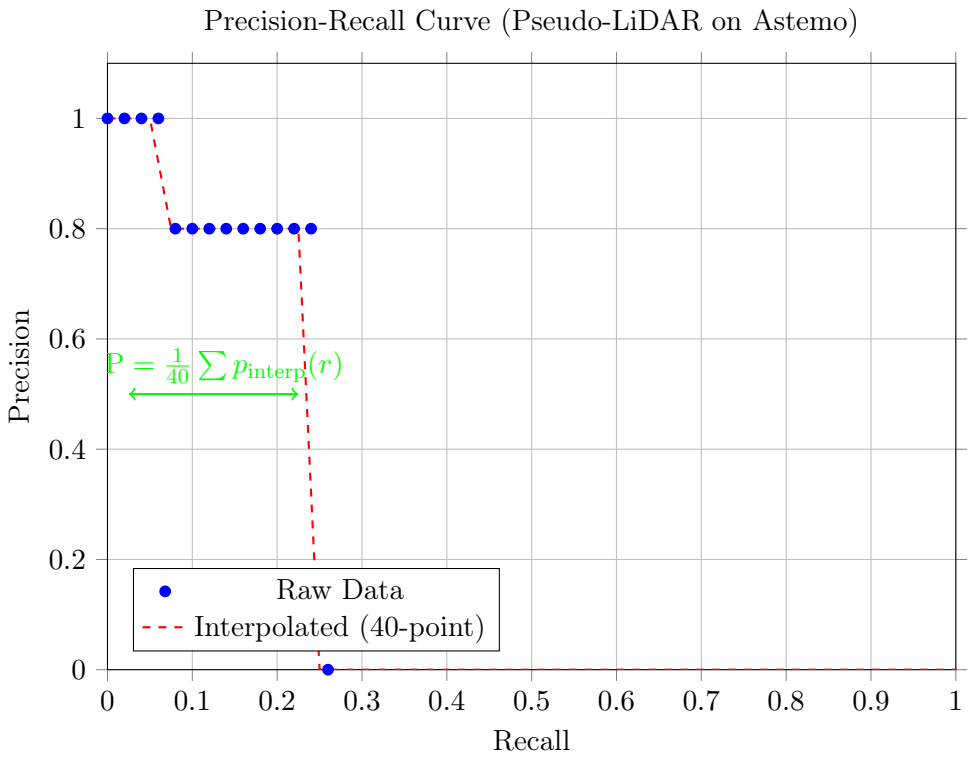

Figure 5.7.: *Precision-Recall curve for pseudo-LiDAR detections on Astemo dataset (IoU threshold = 0.70). The interpolated precision (dashed line) shows maximum recall of 0.24. The Average Precision (AP) of 0.215 is computed as the area under the interpolated curve.*

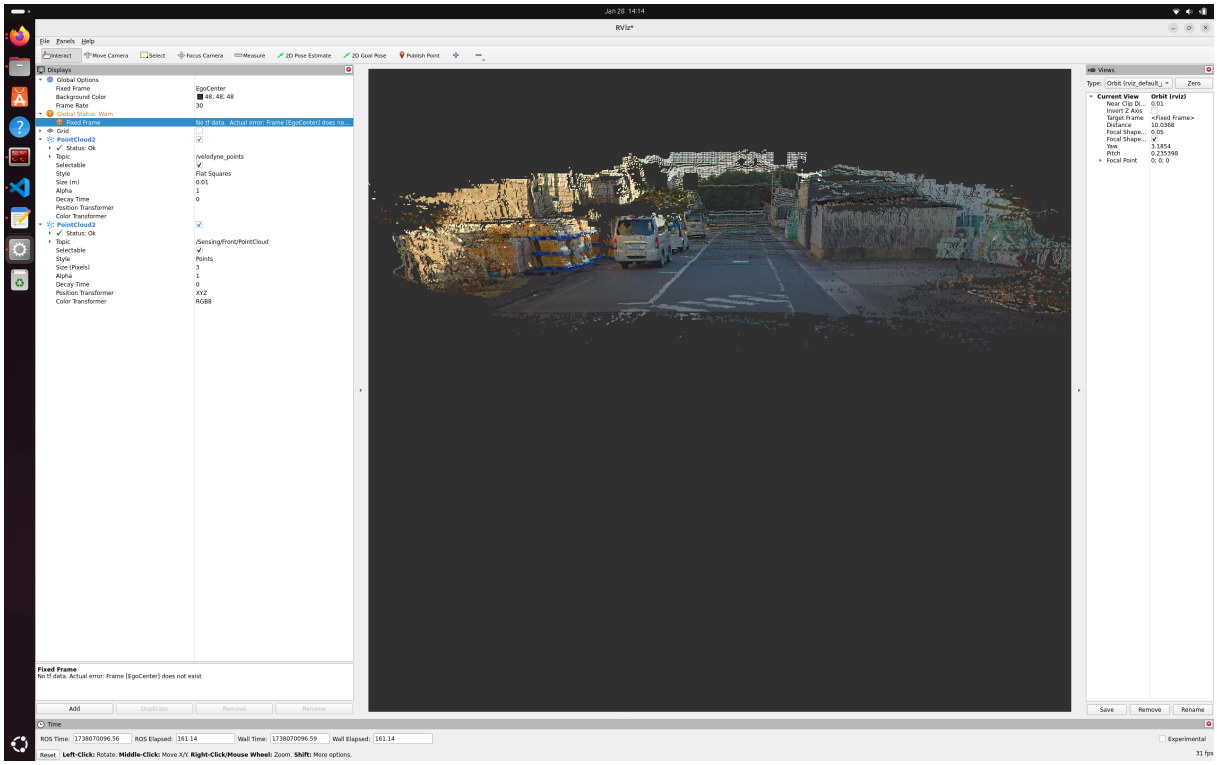

Figure 5.8.: *Visualization of Pseudo-LiDAR from the Astemo Dataset of missed detection.*

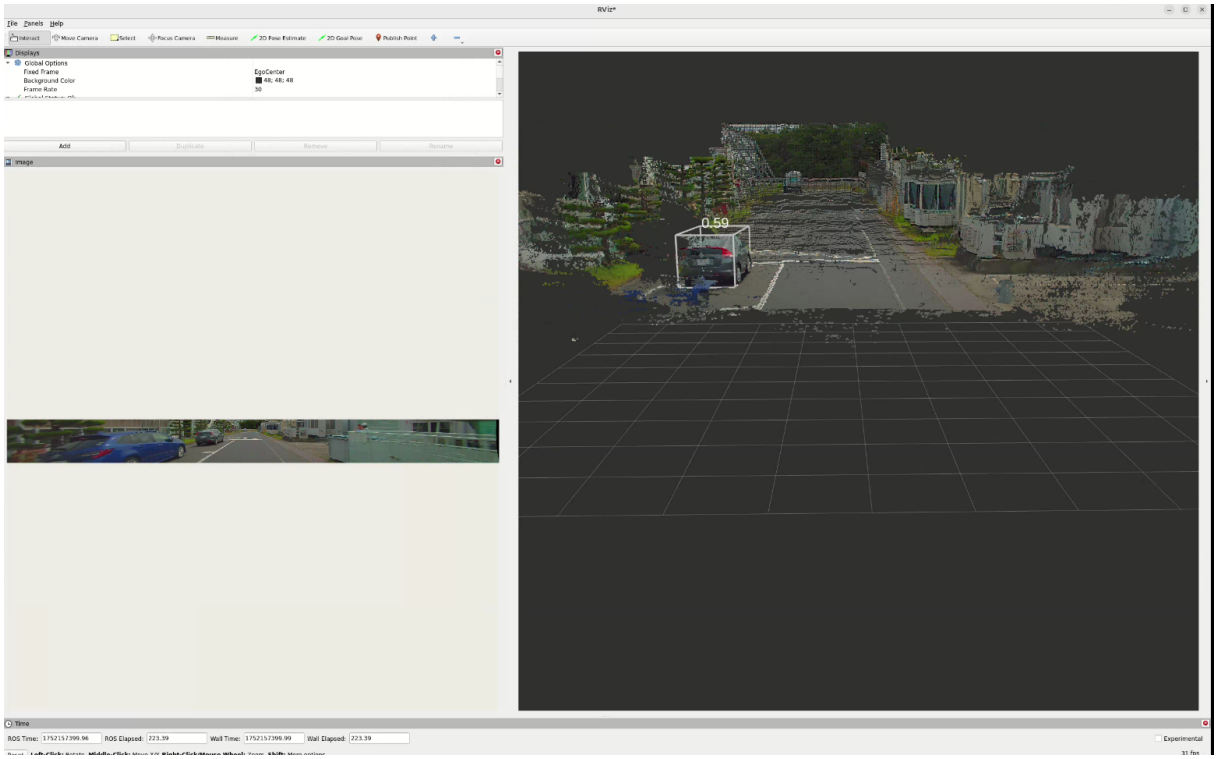

Figure 5.9.: *Visualization of Detection from Pseudo-LiDAR from the Astemo Dataset*

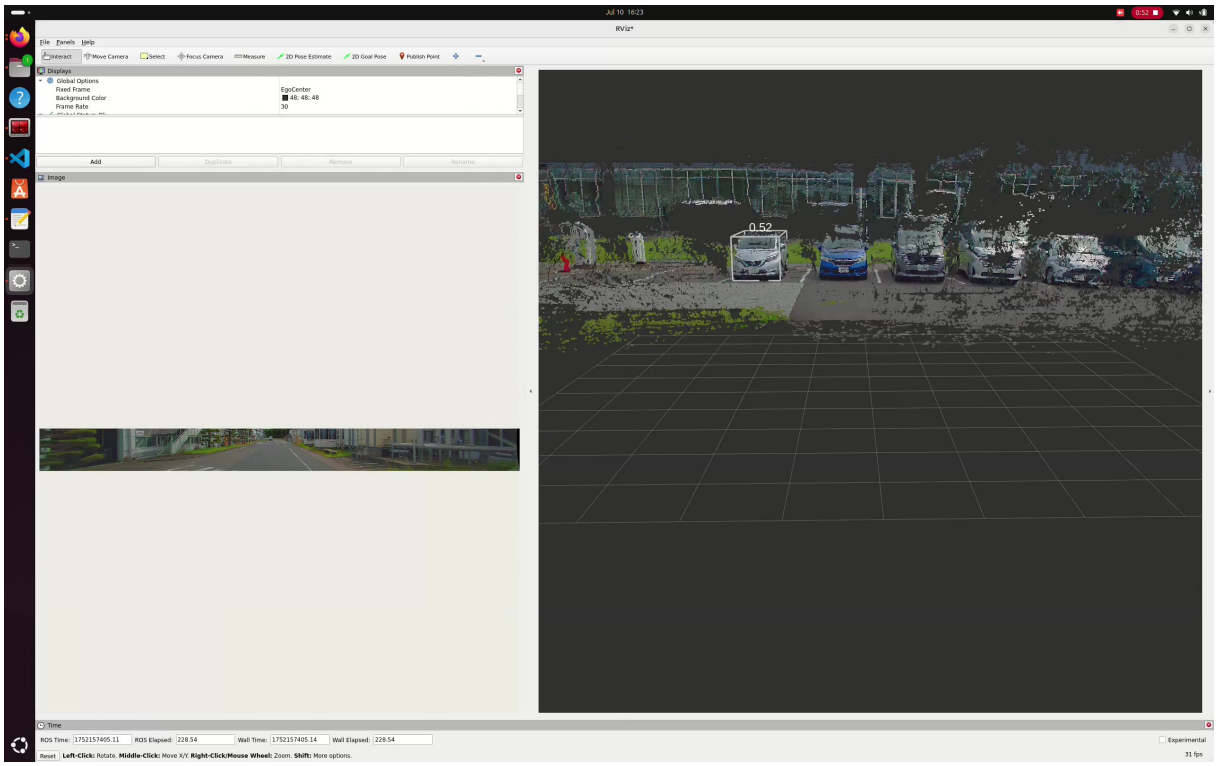

Figure 5.10.: *Visualization of Parking scence of Pseudo-LiDAR from the Astemo Dataset*

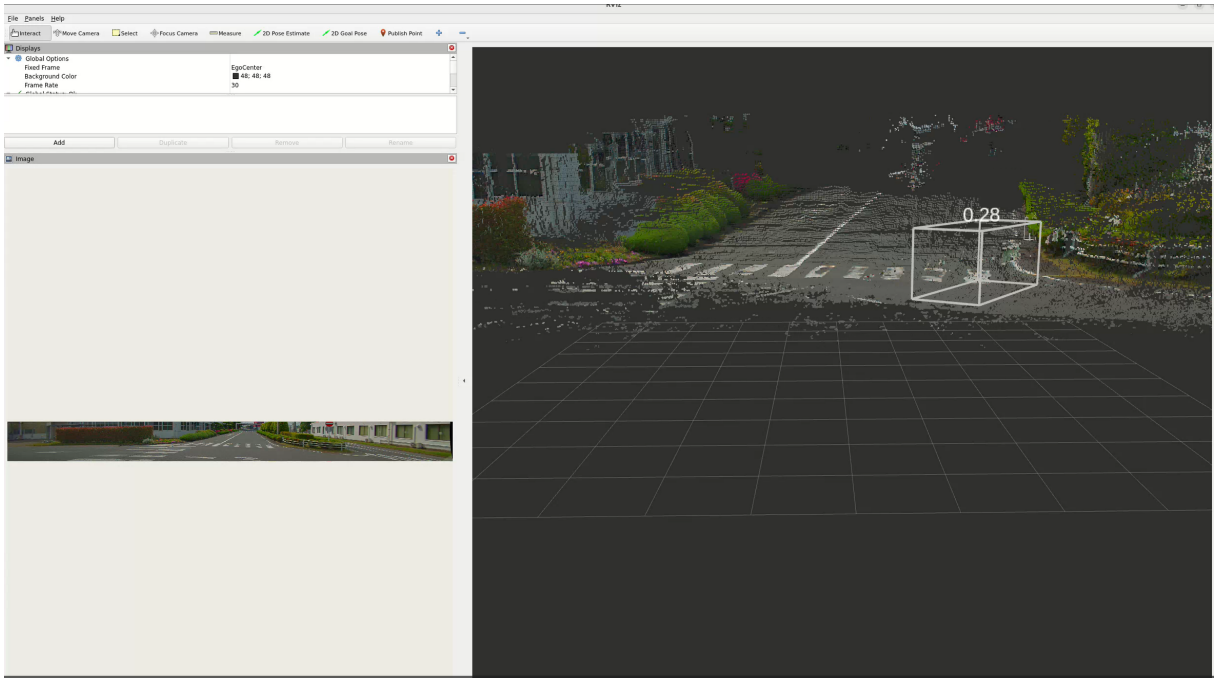

Figure 5.11.: *Visualization of Pseudo-LiDAR from the Astemo Dataset of False Positive Detection*

6. Conclusion

This thesis embarked on a comprehensive investigation into 3D object detection for autonomous systems, motivated by the industrial need to modernize classical geometric workflows with advanced deep learning techniques. The research successfully designed, implemented, and benchmarked deep learning–based 3D object detection models in Docker and ROS environments, providing a robust platform for comparing different sensor modalities and detection architectures. Through a series of structured experiments on both the public Kitti dataset and a proprietary Astemo dataset, this work has generated critical insights into the performance trade-offs between real LiDAR and image-derived pseudo-LiDAR.

The key findings of this research directly address the objectives outlined in the introduction. First, the deep learning models implemented within the OpenPCDet framework, particularly PointRCNN, demonstrated a vast performance improvement over the initial classical methods, confirming the value of the modernization effort. Second, the evaluation on the Kitti benchmark established that while the pseudo-LiDAR pipeline is a viable, low-cost approach for detecting large objects like cars, its performance degrades significantly for smaller, more challenging classes such as pedestrians and cyclists when compared to direct LiDAR sensing.

The central research question of this thesis whether pseudo-LiDAR can match or outperform real LiDAR is answered with a clear conclusion based on the presented evidence.

No, pseudo-LiDAR, in its current implementation, is not better than real LiDAR. The comparative analysis on KITTI revealed a performance gap of over 25 percentage points for cars and more than 40 for cyclists. This conclusion was further solidified by the experiments on the Astemo dataset, where only 12 of 44 pseudo-LiDAR detections met the strict 0.7 IoU threshold for correct localization. This demonstrates that the geometric accuracy provided by direct LiDAR measurements remains the gold standard for robust and reliable 3D perception.

This work contributes a functional ROS2 pipeline for real-time 3D detection, a novel benchmarking of leading models on proprietary industrial data, and a full-stack comparative analysis that grounds its findings in empirical evidence. While this study was limited to a specific set of models and relied on a pseudo ground truth for the proprietary data, the results are clear.

Future research should focus on bridging the performance gap by exploring more advanced depth estimation networks and investigating sensor fusion techniques that combine the strengths of cameras and other low-cost sensors like radar. Furthermore, exploring domain adaptation methods to minimize the distribution shift between real and pseudo-LiDAR data could yield significant performance gains. Ultimately, this thesis confirms that while the pursuit of low-cost 3D perception through stereo vision is a vital research direction, the path to achieving LiDAR-level reliability requires further innovation.

A. Appendix

A.1. Dockerfile for ROS 2 + OpenPCDet Environment

```
# Use NVIDIA's CUDA 11.7.1 base image with Ubuntu 22.04
FROM nvidia/cuda:11.7.1-devel-ubuntu22.04
LABEL maintainer="Om Tiwari"

# Non-interactive apt mode
ENV DEBIAN_FRONTEND=noninteractive

# Install base dependencies
RUN apt update && apt install -y --no-install-recommends \
    git curl wget zsh tmux vim g++ locales gnupg2 lsb-release software-properties-
        ␣ ↪ common \
    && locale-gen en_US.UTF-8

# Set locale environment variables
ENV LANG=en_US.UTF-8
ENV LC_ALL=en_US.UTF-8

# Add ROS 2 GPG key and repository
RUN curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | apt-key
    ␣ ↪  add - && \
    echo "deb [arch=$(dpkg --print-architecture)] http://packages.ros.org/ros2/ubuntu $
        ␣ ↪ (lsb_release -cs) main" \
    > /etc/apt/sources.list.d/ros2-latest.list

# Install ROS 2 Humble Desktop
RUN apt-get update && apt-get install -y ros-humble-desktop

# Install Python and PyTorch with CUDA 11.7
RUN apt update && apt install -y python3.10 python3-pip python3-setuptools sudo
RUN pip install torch==2.0.0+cu117 torchvision==0.15.0+cu117 torchaudio==2.0.0+cu117
    ␣ ↪ --extra-index-url https://download.pytorch.org/whl/cu117

# Install spconv for CUDA 11.7
RUN pip3 install spconv-cu117

# Set default shell
ENV SHELL=/bin/bash
```

Listing A.1: *Dockerfile used for building the GPU-enabled ROS 2 + OpenPCDet environment*

A.2. Docker Run Script

```
#!/bin/bash

IMAGE_NAME="pcdet_ros2_om:stable_with_pv_rcnn"
CONTAINER_NAME="ros_humble"
DOCKERFILE_PATH="."
PROJECT_DIR=$(pwd)
PROJECT_NAME=$(basename "$PROJECT_DIR")

# Build image if not found
if ! docker image inspect "$IMAGE_NAME" > /dev/null 2>&1; then
    docker build -t "$IMAGE_NAME" "$DOCKERFILE_PATH"
fi

# Attach if container already running
if docker ps --format '{{.Names}}' | grep -q "^$CONTAINER_NAME$"; then
    docker exec -it "$CONTAINER_NAME" bash
    exit 0
fi

xhost +local:docker > /dev/null 2>&1

docker run \
    --rm \
    -it \
    --gpus all \
    --env NVIDIA_DRIVER_CAPABILITIES=all \
    --env TERM="xterm-color" \
    --env DISPLAY="$DISPLAY" \
    --env QT_DEBUG_PLUGINS=1 \
    --env QT_X11_NO_MITSHM=1 \
    --volume /tmp/.X11-unix:/tmp/.X11-unix:rw \
    --volume "$HOME/.Xauthority:/root/.Xauthority:rw" \
    --volume "$HOME:/home:rw" \
    --volume "$PROJECT_DIR:/$PROJECT_NAME" \
    --volume "/mnt:/mnt:rw" \
    --volume "/media:/media:rw" \
    --workdir "/$PROJECT_NAME" \
    --hostname "$HOSTNAME" \
    --name "$CONTAINER_NAME" \
    --privileged \
    --net host \
    "$IMAGE_NAME" bash
```

Listing A.2: *Bash script for launching the ROS 2 + OpenPCDet Docker container*

A.3. Python ROS 2 Node Snippet

```
def __cloudCB__(self, cloud_msg):
    # Prepare output message
    out_msg = Detection3DArray()
    # Convert ROS2 PointCloud2 message to NumPy array
    cloud_array = ros2_numpy.point_cloud2.pointcloud2_to_array(cloud_msg)
    # Transform to format expected by OpenPCDet
    np_points = self.__convertCloudFormat__(cloud_array)
    # (Further steps: model inference and publishing to /cloud_detection)
```

Listing A.3: *Callback function from the custom* `/openpcddet` *node for LiDAR point cloud processing*

A.4. OpenPCDet PointRCNN Configuration File

```
CLASS_NAMES: ['Car', 'Pedestrian', 'Cyclist']

DATA_CONFIG:
  _BASE_CONFIG_: cfgs/dataset_configs/kitti_dataset.yaml

  DATA_PROCESSOR:
    - NAME: mask_points_and_boxes_outside_range
      REMOVE_OUTSIDE_BOXES: True

    - NAME: sample_points
      NUM_POINTS: {
          'train': 16384,
          'test': 16384
      }

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

MODEL:
  NAME: PointRCNN

  BACKBONE_3D:
    NAME: PointNet2MSG
    SA_CONFIG:
      NPOINTS: [4096, 1024, 256, 64]
      RADIUS: [[0.1, 0.5], [0.5, 1.0], [1.0, 2.0], [2.0, 4.0]]
      NSAMPLE: [[16, 32], [16, 32], [16, 32], [16, 32]]
      MLPS: [[[16, 16, 32], [32, 32, 64]],
             [[64, 64, 128], [64, 96, 128]],
             [[128, 196, 256], [128, 196, 256]],
             [[256, 256, 512], [256, 384, 512]]]
    FP_MLPS: [[128, 128], [256, 256], [512, 512], [512, 512]]
```

```
POINT_HEAD:
  NAME: PointHeadBox
  CLS_FC: [256, 256]
  REG_FC: [256, 256]
  CLASS_AGNOSTIC: False
  USE_POINT_FEATURES_BEFORE_FUSION: False
  TARGET_CONFIG:
    GT_EXTRA_WIDTH: [0.2, 0.2, 0.2]
    BOX_CODER: PointResidualCoder
    BOX_CODER_CONFIG: {
        'use_mean_size': True,
        'mean_size': [
            [3.9, 1.6, 1.56],
            [0.8, 0.6, 1.73],
            [1.76, 0.6, 1.73]
        ]
    }

  LOSS_CONFIG:
    LOSS_REG: WeightedSmoothL1Loss
    LOSS_WEIGHTS: {
        'point_cls_weight': 1.0,
        'point_box_weight': 1.0,
        'code_weights': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
    }

ROI_HEAD:
  NAME: PointRCNNHead
  CLASS_AGNOSTIC: True

  ROI_POINT_POOL:
    POOL_EXTRA_WIDTH: [0.0, 0.0, 0.0]
    NUM_SAMPLED_POINTS: 512
    DEPTH_NORMALIZER: 70.0

  XYZ_UP_LAYER: [128, 128]
  CLS_FC: [256, 256]
  REG_FC: [256, 256]
  DP_RATIO: 0.0
  USE_BN: False

  SA_CONFIG:
    NPOINTS: [128, 32, -1]
    RADIUS: [0.2, 0.4, 100]
    NSAMPLE: [16, 16, 16]
    MLPS: [[128, 128, 128],
           [128, 128, 256],
           [256, 256, 512]]

  NMS_CONFIG:
    TRAIN:
      NMS_TYPE: nms_gpu
```

```
      MULTI_CLASSES_NMS: False
      NMS_PRE_MAXSIZE: 9000
      NMS_POST_MAXSIZE: 512
      NMS_THRESH: 0.8
    TEST:
      NMS_TYPE: nms_gpu
      MULTI_CLASSES_NMS: False
      NMS_PRE_MAXSIZE: 9000
      NMS_POST_MAXSIZE: 100
      NMS_THRESH: 0.85

  TARGET_CONFIG:
    BOX_CODER: ResidualCoder
    ROI_PER_IMAGE: 128
    FG_RATIO: 0.5

    SAMPLE_ROI_BY_EACH_CLASS: True
    CLS_SCORE_TYPE: cls

    CLS_FG_THRESH: 0.6
    CLS_BG_THRESH: 0.45
    CLS_BG_THRESH_LO: 0.1
    HARD_BG_RATIO: 0.8

    REG_FG_THRESH: 0.55

  LOSS_CONFIG:
    CLS_LOSS: BinaryCrossEntropy
    REG_LOSS: smooth-l1
    CORNER_LOSS_REGULARIZATION: True
    LOSS_WEIGHTS: {
        'rcnn_cls_weight': 1.0,
        'rcnn_reg_weight': 1.0,
        'rcnn_corner_weight': 1.0,
        '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: 0.1
    NMS_PRE_MAXSIZE: 4096
    NMS_POST_MAXSIZE: 500

OPTIMIZATION:
```

```
  BATCH_SIZE_PER_GPU: 2
  NUM_EPOCHS: 80

  OPTIMIZER: adam_onecycle
  LR: 0.01
  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
```

Listing A.4: *OpenPCDet configuration file for PointRCNN model training*

A.5. OpenPCDet ROS2 Node Configuration

```
pcdet:
  ros__parameters:
    config_file: "cfgs/kitti_models/pointrcnn.yaml"
    model_file: "checkpoints/pv_rcnn_8369.pth"
    allow_memory_fractioning: False
    allow_score_thresholding: True
    num_features: 4
    device_id: 0
    device_memory_fraction: 6.0
    threshold_array: [0.7, 0.35, 0.5] # [Car, Pedestrian, Bicycle]
```

Listing A.5: *ROS2 parameter configuration for the OpenPCDet node*

A.6. ROS2 Launch File for OpenPCDet Node

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, GroupAction, SetEnvironmentVariable
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, PythonExpression
```

```
from launch_ros.actions import Node
from nav2_common.launch import RewrittenYaml

def generate_launch_description():
    package_name = 'pcdet_ros2'
    package_dir = get_package_share_directory(package_name)
    config_file = 'pcdet_prcnn.param.yaml'

    namespace = LaunchConfiguration('namespace')
    params_file = LaunchConfiguration('params_file')
    input_topic = LaunchConfiguration('input_topic')
    output_topic = LaunchConfiguration('output_topic')

    configured_params = RewrittenYaml(
        source_file=params_file,
        root_key=namespace,
        param_rewrites={}
    )

    declare_namespace_cmd = DeclareLaunchArgument(
        'namespace',
        default_value='',
        description='Top-level namespace')

    declare_params_file_cmd = DeclareLaunchArgument(
        'params_file',
        default_value=os.path.join(package_dir, 'config', config_file),
        description='Full path to the ROS 2 parameters file to use for the launched
            ␣ ↪ nodes'
    )

    declare_input_topic_cmd = DeclareLaunchArgument(
        'input_topic',
        default_value='/kitti/point_cloud',
        description='Input Point Cloud'
    )

    declare_output_topic_cmd = DeclareLaunchArgument(
        'output_topic',
        default_value='cloud_detections',
        description='Output Object Detections'
    )

    pcdet = Node(
        package=package_name,
        executable='pcdet',
        name='pcdet',
        output='screen',
        parameters=[configured_params,
                    {'package_folder_path': package_dir}],
        remappings=[("input", input_topic),
                    ("output", output_topic)]
```

```
    )

    ld = LaunchDescription()

    ld.add_action(declare_namespace_cmd)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(declare_input_topic_cmd)
    ld.add_action(declare_output_topic_cmd)
    ld.add_action(pcdet)

    return ld
```

Listing A.6: *ROS2 launch file for starting the OpenPCDet detection node*

A.7. OpenPCDet Evaluation Script

```
import _init_path
import argparse
import datetime
import glob
import os
import re
import time
from pathlib import Path

import numpy as np
import torch
from tensorboardX import SummaryWriter

from eval_utils import eval_utils
from pcdet.config import cfg, cfg_from_list, cfg_from_yaml_file, log_config_to_file
from pcdet.datasets import build_dataloader
from pcdet.models import build_network
from pcdet.utils import common_utils


def parse_config():
    parser = argparse.ArgumentParser(description='arg parser')
    parser.add_argument('--cfg_file', type=str, default=None, help='specify the config
        ␣ ↪ for training')

    parser.add_argument('--batch_size', type=int, default=None, required=False, help='
        ␣ ↪ batch size for training')
    parser.add_argument('--workers', type=int, default=4, help='number of workers for
        ␣ ↪ dataloader')
    parser.add_argument('--extra_tag', type=str, default='default', help='extra tag for
        ␣ ↪  this experiment')
    parser.add_argument('--ckpt', type=str, default=None, help='checkpoint to start
        ␣ ↪ from')
```

```
    parser.add_argument('--pretrained_model', type=str, default=None, help='
        ␣ ↪ pretrained_model')
    parser.add_argument('--launcher', choices=['none', 'pytorch', 'slurm'], default='
        ␣ ↪ none')
    parser.add_argument('--tcp_port', type=int, default=18888, help='tcp port for
        ␣ ↪ distrbuted training')
    parser.add_argument('--local_rank', type=int, default=None, help='local rank for
        ␣ ↪ distributed training')
    parser.add_argument('--set', dest='set_cfgs', default=None, nargs=argparse.
        ␣ ↪ REMAINDER,
                        help='set extra config keys if needed')

    parser.add_argument('--max_waiting_mins', type=int, default=30, help='max waiting
        ␣ ↪ minutes')
    parser.add_argument('--start_epoch', type=int, default=0, help='')
    parser.add_argument('--eval_tag', type=str, default='default', help='eval tag for
        ␣ ↪ this experiment')
    parser.add_argument('--eval_all', action='store_true', default=False, help='whether
        ␣ ↪  to evaluate all checkpoints')
    parser.add_argument('--ckpt_dir', type=str, default=None, help='specify a ckpt
        ␣ ↪ directory to be evaluated if needed')
    parser.add_argument('--save_to_file', action='store_true', default=False, help='')
    parser.add_argument('--infer_time', action='store_true', default=False, help='
        ␣ ↪ calculate inference latency')

    args = parser.parse_args()

    cfg_from_yaml_file(args.cfg_file, cfg)
    cfg.TAG = Path(args.cfg_file).stem
    cfg.EXP_GROUP_PATH = '/'.join(args.cfg_file.split('/')[1:-1]) # remove 'cfgs' and '
        ␣ ↪ xxxx.yaml'

    np.random.seed(1024)

    if args.set_cfgs is not None:
        cfg_from_list(args.set_cfgs, cfg)

    return args, cfg


def eval_single_ckpt(model, test_loader, args, eval_output_dir, logger, epoch_id,
    ␣ ↪ dist_test=False):
    # load checkpoint
    model.load_params_from_file(filename=args.ckpt, logger=logger, to_cpu=dist_test,
                                pre_trained_path=args.pretrained_model)
    model.cuda()

    # start evaluation
    eval_utils.eval_one_epoch(
        cfg, args, model, test_loader, epoch_id, logger, dist_test=dist_test,
        result_dir=eval_output_dir
    )
```

```
def get_no_evaluated_ckpt(ckpt_dir, ckpt_record_file, args):
    ckpt_list = glob.glob(os.path.join(ckpt_dir, '*checkpoint_epoch_*.pth'))
    ckpt_list.sort(key=os.path.getmtime)
    evaluated_ckpt_list = [float(x.strip()) for x in open(ckpt_record_file, 'r').
        ␣ ↪ readlines()]

    for cur_ckpt in ckpt_list:
        num_list = re.findall('checkpoint_epoch_(.*).pth', cur_ckpt)
        if num_list.__len__() == 0:
            continue

        epoch_id = num_list[-1]
        if 'optim' in epoch_id:
            continue
        if float(epoch_id) not in evaluated_ckpt_list and int(float(epoch_id)) >= args.
            ␣ ↪ start_epoch:
            return epoch_id, cur_ckpt
    return -1, None


def repeat_eval_ckpt(model, test_loader, args, eval_output_dir, logger, ckpt_dir,
    ␣ ↪ dist_test=False):
    # evaluated ckpt record
    ckpt_record_file = eval_output_dir / ('eval_list_%s.txt' % cfg.DATA_CONFIG.
        ␣ ↪ DATA_SPLIT['test'])
    with open(ckpt_record_file, 'a'):
        pass

    # tensorboard log
    if cfg.LOCAL_RANK == 0:
        tb_log = SummaryWriter(log_dir=str(eval_output_dir / ('tensorboard_%s' % cfg.
            ␣ ↪ DATA_CONFIG.DATA_SPLIT['test'])))
    total_time = 0
    first_eval = True

    while True:
        # check whether there is checkpoint which is not evaluated
        cur_epoch_id, cur_ckpt = get_no_evaluated_ckpt(ckpt_dir, ckpt_record_file, args
            ␣ ↪ )
        if cur_epoch_id == -1 or int(float(cur_epoch_id)) < args.start_epoch:
            wait_second = 30
            if cfg.LOCAL_RANK == 0:
                print('Wait %s seconds for next check (progress: %.1f / %d minutes): %s
                    ␣ ↪ \r'
                      % (wait_second, total_time * 1.0 / 60, args.max_waiting_mins,
                          ␣ ↪ ckpt_dir), end='', flush=True)
            time.sleep(wait_second)
            total_time += 30
            if total_time > args.max_waiting_mins * 60 and (first_eval is False):
                break
```

```
            continue

        total_time = 0
        first_eval = False

        model.load_params_from_file(filename=cur_ckpt, logger=logger, to_cpu=dist_test)
        model.cuda()

        # start evaluation
        cur_result_dir = eval_output_dir / ('epoch_%s' % cur_epoch_id) / cfg.
            ␣ ↪ DATA_CONFIG.DATA_SPLIT['test']
        tb_dict = eval_utils.eval_one_epoch(
            cfg, args, model, test_loader, cur_epoch_id, logger, dist_test=dist_test,
            result_dir=cur_result_dir
        )

        if cfg.LOCAL_RANK == 0:
            for key, val in tb_dict.items():
                tb_log.add_scalar(key, val, cur_epoch_id)

        # record this epoch which has been evaluated
        with open(ckpt_record_file, 'a') as f:
            print('%s' % cur_epoch_id, file=f)
        logger.info('Epoch %s has been evaluated' % cur_epoch_id)


def main():
    args, cfg = parse_config()

    if args.infer_time:
        os.environ['CUDA_LAUNCH_BLOCKING'] = '1'

    if args.launcher == 'none':
        dist_test = False
        total_gpus = 1
    else:
        if args.local_rank is None:
            args.local_rank = int(os.environ.get('LOCAL_RANK', '0'))

        total_gpus, cfg.LOCAL_RANK = getattr(common_utils, 'init_dist_%s' % args.
            ␣ ↪ launcher)(
            args.tcp_port, args.local_rank, backend='nccl'
        )
        dist_test = True

    if args.batch_size is None:
        args.batch_size = cfg.OPTIMIZATION.BATCH_SIZE_PER_GPU
    else:
        assert args.batch_size % total_gpus == 0, 'Batch size should match the number
            ␣ ↪ of gpus'
        args.batch_size = args.batch_size // total_gpus
```

```
output_dir = cfg.ROOT_DIR / 'output' / cfg.EXP_GROUP_PATH / cfg.TAG / args.
    ␣ ↪ extra_tag
output_dir.mkdir(parents=True, exist_ok=True)

eval_output_dir = output_dir / 'eval'

if not args.eval_all:
    num_list = re.findall(r'\d+', args.ckpt) if args.ckpt is not None else []
    epoch_id = num_list[-1] if num_list.__len__() > 0 else 'no_number'
    eval_output_dir = eval_output_dir / ('epoch_%s' % epoch_id) / cfg.DATA_CONFIG.
        ␣ ↪ DATA_SPLIT['test']
else:
    eval_output_dir = eval_output_dir / 'eval_all_default'

if args.eval_tag is not None:
    eval_output_dir = eval_output_dir / args.eval_tag

eval_output_dir.mkdir(parents=True, exist_ok=True)
log_file = eval_output_dir / ('log_eval_%s.txt' % datetime.datetime.now().strftime(
    ␣ ↪ '%Y%m%d-%H%M%S'))
logger = common_utils.create_logger(log_file, rank=cfg.LOCAL_RANK)

# log to file
logger.info('**********************Start logging**********************')
gpu_list = os.environ['CUDA_VISIBLE_DEVICES'] if 'CUDA_VISIBLE_DEVICES' in os.
    ␣ ↪ environ.keys() else 'ALL'
logger.info('CUDA_VISIBLE_DEVICES=%s' % gpu_list)

if dist_test:
    logger.info('total_batch_size: %d' % (total_gpus * args.batch_size))
for key, val in vars(args).items():
    logger.info('{:16} {}'.format(key, val))
log_config_to_file(cfg, logger=logger)

ckpt_dir = args.ckpt_dir if args.ckpt_dir is not None else output_dir / 'ckpt'

test_set, test_loader, sampler = build_dataloader(
    dataset_cfg=cfg.DATA_CONFIG,
    class_names=cfg.CLASS_NAMES,
    batch_size=args.batch_size,
    dist=dist_test, workers=args.workers, logger=logger, training=False
)

model = build_network(model_cfg=cfg.MODEL, num_class=len(cfg.CLASS_NAMES), dataset=
    ␣ ↪ test_set)
with torch.no_grad():
    if args.eval_all:
        repeat_eval_ckpt(model, test_loader, args, eval_output_dir, logger,
            ␣ ↪ ckpt_dir, dist_test=dist_test)
    else:
        eval_single_ckpt(model, test_loader, args, eval_output_dir, logger,
            ␣ ↪ epoch_id, dist_test=dist_test)
```

```
if __name__ == '__main__':
    main()
```

Listing A.7: *Python script for evaluating OpenPCDet models*

A.8. Python Snippet for Point Cloud Conversion

```
def __convertCloudFormat__(self, cloud_array, remove_nans=True, dtype=float):
    '''
    '''
    if remove_nans:
        mask = np.isfinite(cloud_array['x']) & np.isfinite(cloud_array['y']) & np.
            ␣ ↪ isfinite(cloud_array['z'])
        cloud_array = cloud_array[mask]

    points = np.zeros((cloud_array.shape[0], 3), dtype=dtype)
    points[:, 0] = cloud_array['x']
    points[:, 1] = cloud_array['y']
    points[:, 2] = cloud_array['z']

    if self.__use_pseudo_lidar__:
        points = self.__subsample_points__(points)
        if self.__num_features__ > 3:
            intensity = self.__compute_intensity__(points)
            points = np.hstack((points, intensity[:, None]))
    else:
        if self.__num_features__ > 3 and 'intensity' in cloud_array.dtype.names:
            intensity = cloud_array['intensity'][: len(points)]
            points = np.hstack((points, intensity.reshape(-1, 1)))
    return points

def __compute_intensity__(self, points: np.ndarray) -> np.ndarray:
    """Generate fake intensity values based on range."""
    dist = np.linalg.norm(points[:, :3], axis=1)
    if dist.max() > 0:
        return dist / dist.max()
    return dist

def __subsample_points__(self, points: np.ndarray) -> np.ndarray:
    """Subsample the input point cloud deterministically"""
    if self.__max_subsample_points__ is not None and len(points) > self.
        ␣ ↪ __max_subsample_points__:
        # Deterministic uniform sampling
        step = max(1, len(points) // self.__max_subsample_points__)
        return points[::step][:self.__max_subsample_points__]
    return points
```

Listing A.8: *Python snippet for converting ROS2 PointCloud2 to OpenPCDet format*

A.9. OpenPCDet Kitti Dataset Configuration

```
DATASET: 'KittiDataset'
DATA_PATH: '../data/kitti'

POINT_CLOUD_RANGE: [0, -40, -3, 70.4, 40, 1]

DATA_SPLIT: {
    'train': train,
    'test': val
}

INFO_PATH: {
    'train': [kitti_infos_train.pkl],
    'test': [kitti_infos_val.pkl],
}

GET_ITEM_LIST: ["points"]
FOV_POINTS_ONLY: True

DATA_AUGMENTOR:
    DISABLE_AUG_LIST: ['placeholder']
    AUG_CONFIG_LIST:
        - NAME: gt_sampling
          USE_ROAD_PLANE: True
          DB_INFO_PATH:
              - kitti_dbinfos_train.pkl
          PREPARE: {
              filter_by_min_points: ['Car:5', 'Pedestrian:5', 'Cyclist:5'],
              filter_by_difficulty: [-1],
          }

          SAMPLE_GROUPS: ['Car:20','Pedestrian:15', 'Cyclist:15']
          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]


POINT_FEATURE_ENCODING: {
    encoding_type: absolute_coordinates_encoding,
    used_feature_list: ['x', 'y', 'z', 'intensity'],
```

```
    src_feature_list: ['x', 'y', 'z', 'intensity'],
}


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: [0.05, 0.05, 0.1]
      MAX_POINTS_PER_VOXEL: 5
      MAX_NUMBER_OF_VOXELS: {
        'train': 16000,
        'test': 40000
      }
```

Listing A.9: *OpenPCDet configuration for the KITTI dataset*

A.10. End2End pseudo-Lidar Joint Evaluation Script Snippet

```
def eval_one_epoch_joint(model, depth_model, dataloader, epoch_id, result_dir, logger)
    ␣ ↪ :
    np.random.seed(666)
    MEAN_SIZE = torch.from_numpy(cfg.CLS_MEAN_SIZE[0]).cuda()
    mode = 'TEST' if args.test else 'EVAL'

    final_output_dir = os.path.join(result_dir, 'final_result', 'data')
    os.makedirs(final_output_dir, exist_ok=True)

    if args.save_result:
        roi_output_dir = os.path.join(result_dir, 'roi_result', 'data')
        refine_output_dir = os.path.join(result_dir, 'refine_result', 'data')
        rpn_output_dir = os.path.join(result_dir, 'rpn_result', 'data')
        os.makedirs(rpn_output_dir, exist_ok=True)
        os.makedirs(roi_output_dir, exist_ok=True)
        os.makedirs(refine_output_dir, exist_ok=True)

    logger.info('---- EPOCH %s JOINT EVALUATION ----' % epoch_id)
    logger.info('==> Output file: %s' % result_dir)
    model.eval()
    depth_model.net.eval()

    thresh_list = [0.1, 0.3, 0.5, 0.7, 0.9]
    total_recalled_bbox_list, total_gt_bbox = [0] * 5, 0
```

```
total_roi_recalled_bbox_list = [0] * 5
dataset = dataloader.dataset
cnt = final_total = total_cls_acc = total_cls_acc_refined = total_rpn_iou = 0

progress_bar = tqdm.tqdm(total=len(dataloader), leave=True, desc='eval')
for data in dataloader:
    depth_loss, data = depth_model.eval(data, args.max_high)
    cnt += 1
    sample_id, pts_rect, pts_features, pts_input = \
        data['sample_id'], data['pts_rect'], data['pts_features'], data['pts_input'
            ␣ ↪ ]
    batch_size = len(sample_id)
    inputs = pts_input
    input_data = {'pts_input': pts_input}

    # model inference
    ret_dict = model(input_data)

    roi_scores_raw = ret_dict['roi_scores_raw'] # (B, M)
    roi_boxes3d = ret_dict['rois'] # (B, M, 7)
    seg_result = ret_dict['seg_result'].long() # (B, N)

    rcnn_cls = ret_dict['rcnn_cls'].view(batch_size, -1, ret_dict['rcnn_cls'].shape
        ␣ ↪ [1])
    rcnn_reg = ret_dict['rcnn_reg'].view(batch_size, -1, ret_dict['rcnn_reg'].shape
        ␣ ↪ [1]) # (B, M, C)

    # bounding box regression
    anchor_size = MEAN_SIZE
    if cfg.RCNN.SIZE_RES_ON_ROI:
        assert False

    pred_boxes3d = decode_bbox_target(roi_boxes3d.view(-1, 7), rcnn_reg.view(-1,
        ␣ ↪ rcnn_reg.shape[-1]),
                                      anchor_size=anchor_size,
                                      loc_scope=cfg.RCNN.LOC_SCOPE,
                                      loc_bin_size=cfg.RCNN.LOC_BIN_SIZE,
                                      num_head_bin=cfg.RCNN.NUM_HEAD_BIN,
                                      get_xz_fine=True, get_y_by_bin=cfg.RCNN.
                                          ␣ ↪ LOC_Y_BY_BIN,
                                      loc_y_scope=cfg.RCNN.LOC_Y_SCOPE, loc_y_bin_size
                                          ␣ ↪ =cfg.RCNN.LOC_Y_BIN_SIZE,
                                      get_ry_fine=True).view(batch_size, -1, 7)

    # scoring
    if rcnn_cls.shape[2] == 1:
        raw_scores = rcnn_cls # (B, M, 1)

        norm_scores = torch.sigmoid(raw_scores)
        pred_classes = (norm_scores > cfg.RCNN.SCORE_THRESH).long()
    else:
        pred_classes = torch.argmax(rcnn_cls, dim=1).view(-1)
```

```
            cls_norm_scores = F.softmax(rcnn_cls, dim=1)
            raw_scores = rcnn_cls[:, pred_classes]
            norm_scores = cls_norm_scores[:, pred_classes]

        # evaluation
        recalled_num = gt_num = rpn_iou = 0
        if not args.test:
            if not cfg.RPN.FIXED:
                rpn_cls_label, rpn_reg_label = data['rpn_cls_label'], data['
                    ␣ ↪ rpn_reg_label']
                rpn_cls_label = torch.from_numpy(rpn_cls_label).cuda(non_blocking=True).
                    ␣ ↪ long()

            gt_boxes3d = data['gt_boxes3d']

            for k in range(batch_size):
                # calculate recall
                cur_gt_boxes3d = gt_boxes3d[k]
                tmp_idx = cur_gt_boxes3d.__len__() - 1

                while tmp_idx >= 0 and cur_gt_boxes3d[tmp_idx].sum() == 0:
                    tmp_idx -= 1

                if tmp_idx >= 0:
                    cur_gt_boxes3d = cur_gt_boxes3d[:tmp_idx + 1]

                    cur_gt_boxes3d = torch.from_numpy(cur_gt_boxes3d).cuda(non_blocking=
                        ␣ ↪ True).float()
                    iou3d = iou3d_utils.boxes_iou3d_gpu(pred_boxes3d[k], cur_gt_boxes3d)
                    gt_max_iou, _ = iou3d.max(dim=0)
                    refined_iou, _ = iou3d.max(dim=1)

                    for idx, thresh in enumerate(thresh_list):
                        total_recalled_bbox_list[idx] += (gt_max_iou > thresh).sum().item
                            ␣ ↪ ()
                    recalled_num += (gt_max_iou > 0.7).sum().item()
                    gt_num += cur_gt_boxes3d.shape[0]
                    total_gt_bbox += cur_gt_boxes3d.shape[0]

                    # original recall
                    iou3d_in = iou3d_utils.boxes_iou3d_gpu(roi_boxes3d[k],
                        ␣ ↪ cur_gt_boxes3d)
                    gt_max_iou_in, _ = iou3d_in.max(dim=0)

                    for idx, thresh in enumerate(thresh_list):
                        total_roi_recalled_bbox_list[idx] += (gt_max_iou_in > thresh).sum
                            ␣ ↪ ().item()

                if not cfg.RPN.FIXED:
                    fg_mask = rpn_cls_label > 0
                    correct = ((seg_result == rpn_cls_label) & fg_mask).sum().float()
```

```
                union = fg_mask.sum().float() + (seg_result > 0).sum().float() -
                    ␣ ↪ correct
                rpn_iou = correct / torch.clamp(union, min=1.0)
                total_rpn_iou += rpn_iou.item()

        disp_dict = {'mode': mode, 'recall': '%d/%d (%.4f)' % (total_recalled_bbox_list
            ␣ ↪ [3], total_gt_bbox,
                                                               total_recalled_bbox_list[3]/
                                                                   ␣ ↪ total_gt_bbox)}
        progress_bar.set_postfix(disp_dict)
        progress_bar.update()

        if args.save_result:
            # save roi and refine results
            roi_boxes3d_np = roi_boxes3d.cpu().numpy()
            pred_boxes3d_np = pred_boxes3d.cpu().numpy()
            roi_scores_raw_np = roi_scores_raw.cpu().numpy()
            raw_scores_np = raw_scores.cpu().numpy()

            rpn_cls_np = ret_dict['rpn_cls'].cpu().numpy()
            rpn_xyz_np = ret_dict['backbone_xyz'].cpu().numpy()
            seg_result_np = seg_result.cpu().numpy()
            output_data = np.concatenate((rpn_xyz_np, rpn_cls_np.reshape(batch_size,
                ␣ ↪ -1, 1),
                                          seg_result_np.reshape(batch_size, -1, 1)), axis
                                              ␣ ↪ =2)

            for k in range(batch_size):
                cur_sample_id = sample_id[k]
                calib = dataset.get_calib(cur_sample_id)
                image_shape = dataset.get_image_shape(cur_sample_id)
                save_kitti_format(cur_sample_id, calib, roi_boxes3d_np[k],
                    ␣ ↪ roi_output_dir,
                                  roi_scores_raw_np[k], image_shape)
                save_kitti_format(cur_sample_id, calib, pred_boxes3d_np[k],
                    ␣ ↪ refine_output_dir,
                                  raw_scores_np[k], image_shape)

                output_file = os.path.join(rpn_output_dir, '%06d.npy' % cur_sample_id)
                np.save(output_file, output_data.astype(np.float32))

        # scores thresh
        inds = norm_scores > cfg.RCNN.SCORE_THRESH

        for k in range(batch_size):
            cur_inds = inds[k].view(-1)
            if cur_inds.sum() == 0:
                continue

            pred_boxes3d_selected = pred_boxes3d[k, cur_inds]
            raw_scores_selected = raw_scores[k, cur_inds]
            norm_scores_selected = norm_scores[k, cur_inds]
```

```
        # NMS thresh
        # rotated nms
        boxes_bev_selected = kitti_utils.boxes3d_to_bev_torch(pred_boxes3d_selected
            ␣ ↪ )
        keep_idx = iou3d_utils.nms_gpu(boxes_bev_selected, raw_scores_selected, cfg
            ␣ ↪ .RCNN.NMS_THRESH).view(-1)
        pred_boxes3d_selected = pred_boxes3d_selected[keep_idx]
        scores_selected = raw_scores_selected[keep_idx]
        pred_boxes3d_selected, scores_selected = pred_boxes3d_selected.cpu().numpy
            ␣ ↪ (), scores_selected.cpu().numpy()

        cur_sample_id = sample_id[k]
        calib = dataset.get_calib(cur_sample_id)
        final_total += pred_boxes3d_selected.shape[0]
        image_shape = dataset.get_image_shape(cur_sample_id)
        save_kitti_format(cur_sample_id, calib, pred_boxes3d_selected,
            ␣ ↪ final_output_dir, scores_selected, image_shape)

progress_bar.close()
# dump empty files
split_file = os.path.join(dataset.imageset_dir, '..', '..', 'ImageSets', dataset.
    ␣ ↪ split + '.txt')
split_file = os.path.abspath(split_file)
image_idx_list = [x.strip() for x in open(split_file).readlines()]
empty_cnt = 0
for k in range(image_idx_list.__len__()):
    cur_file = os.path.join(final_output_dir, '%s.txt' % image_idx_list[k])
    if not os.path.exists(cur_file):
        with open(cur_file, 'w') as temp_f:
            pass
        empty_cnt += 1
        logger.info('empty_cnt=%d: dump empty file %s' % (empty_cnt, cur_file))

ret_dict = {'empty_cnt': empty_cnt}

logger.info('-------------------performance of epoch %s---------------------' %
    ␣ ↪ epoch_id)
logger.info(str(datetime.now()))

avg_rpn_iou = (total_rpn_iou / max(cnt, 1.0))
avg_cls_acc = (total_cls_acc / max(cnt, 1.0))
avg_cls_acc_refined = (total_cls_acc_refined / max(cnt, 1.0))
avg_det_num = (final_total / max(len(dataset), 1.0))
logger.info('final average detections: %.3f' % avg_det_num)
logger.info('final average rpn_iou refined: %.3f' % avg_rpn_iou)
logger.info('final average cls acc: %.3f' % avg_cls_acc)
logger.info('final average cls acc refined: %.3f' % avg_cls_acc_refined)
ret_dict['rpn_iou'] = avg_rpn_iou
ret_dict['rcnn_cls_acc'] = avg_cls_acc
ret_dict['rcnn_cls_acc_refined'] = avg_cls_acc_refined
ret_dict['rcnn_avg_num'] = avg_det_num
```

```
    for idx, thresh in enumerate(thresh_list):
        cur_roi_recall = total_roi_recalled_bbox_list[idx] / max(total_gt_bbox, 1.0)
        logger.info('total roi bbox recall(thresh=%.3f): %d / %d = %f' % (thresh,
            ␣ ↪ total_roi_recalled_bbox_list[idx],
                                                                    total_gt_bbox,
                                                                        ␣ ↪ cur_roi_recall
                                                                        ␣ ↪ ))
        ret_dict['rpn_recall(thresh=%.2f)' % thresh] = cur_roi_recall

    for idx, thresh in enumerate(thresh_list):
        cur_recall = total_recalled_bbox_list[idx] / max(total_gt_bbox, 1.0)
        logger.info('total bbox recall(thresh=%.3f): %d / %d = %f' % (thresh,
            ␣ ↪ total_recalled_bbox_list[idx],
                                                               total_gt_bbox,
                                                                  ␣ ↪ cur_recall))
        ret_dict['rcnn_recall(thresh=%.2f)' % thresh] = cur_recall

    if cfg.TEST.SPLIT != 'test':
        logger.info('Averate Precision:')
        name_to_class = {'Car': 0, 'Pedestrian': 1, 'Cyclist': 2}
        ap_result_str, ap_dict = kitti_evaluate(dataset.label_dir, final_output_dir,
            ␣ ↪ label_split_file=split_file,
                                            current_class=name_to_class[cfg.CLASSES])
        logger.info(ap_result_str)
        ret_dict.update(ap_dict)

    logger.info('result is saved to: %s' % result_dir)
    return ret_dict
```

Listing A.10: *Python snippet for evaluating a combined depth and 3D detection model*

Bibliography

3D Object Detection Evaluation 2017 (2017). Online. URL: http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d (visited on 05/25/2025).

adam-abed-abud (2025). *Sparse Convolutional Neural Networks for particle classification*. GitHub. URL: https://github.com/adam-abed-abud/SparseNet_classification_algorithm (visited on 05/25/2025).

Built In (2025). *Mean Average Precision (mAP) Explained*. Online. URL: https://builtin.com/articles/mean-average-precision (visited on 05/25/2025).

Chang, Jia-Ren and Yong-Sheng Chen (2018). "Pyramid stereo matching network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5410–5418.

Deng, Dingsheng (2020). "DBSCAN clustering algorithm based on density". In: *2020 7th international forum on electrical engineering and automation (IFEEA)*. IEEE, pp. 949–953.

Derpanis, Konstantinos G (2010). "Overview of the RANSAC Algorithm". In: *Image Rochester NY* 4.1, pp. 2–3.

Docker, Inc. (2020). *What is Docker?* https://www.docker.com/what-docker. Retrieved on Aug. 12, 2025.

Geiger, Andreas, Philip Lenz, and Raquel Urtasun (2012a). "Are we ready for autonomous driving? The KITTI vision benchmark suite". In: *CVPR*.

– (2012b). "Are we ready for autonomous driving? the kitti vision benchmark suite". In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE, pp. 3354–3361.

Hong, Dongwon and Changjoo Moon (2024). "Autonomous driving system architecture with integrated ROS2 and adaptive AUTOSAR". In: *Electronics* 13.7, p. 1303.

Lang, Alex H, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom (2019). "Pointpillars: Fast encoders for object detection from point clouds". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 12697–12705.

Liu, Baoyuan, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky (2015). "Sparse convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 806–814.

Liu, Zhijian, Haotian Tang, Yujun Lin, and Song Han (2019). "Point-voxel cnn for efficient 3d deep learning". In: *Advances in neural information processing systems* 32.

Luo, Chester and Kevin Lai (2024). *Optimizing Sparse Convolution on GPUs with CUDA for 3D Point Cloud Processing in Embedded Systems*. arXiv: 2402.07710 [cs.LG]. URL: https://arxiv.org/abs/2402.07710 (visited on 05/25/2025).

Maskeliūnas, Rytis, Sarmad Maqsood, Mantas Vaškevičius, and Julius Gelšvartas (2025). "Fusing LiDAR and photogrammetry for accurate 3D data: A hybrid approach". In: *Remote sensing*. 17.3, pp. 1–27.

Open Robotics (2025a). *Concepts — ROS 2 Documentation: Humble*. Online. Retrieved on Aug. 13, 2025. URL: https://docs.ros.org/en/humble/Concepts.html.

Open Robotics (2025b). *Recording and playing back data — ROS 2 Documentation: Foxy*. https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html. Retrieved on May 25, 2025.

– (2025c). *Understanding nodes — ROS 2 Documentation: Humble*. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html. Retrieved on May 25, 2025.

open-mmlab (2020). *OpenPCDet Toolbox for LiDAR-based 3D Object Detection*. GitHub. URL: https://github.com/open-mmlab/OpenPCDet (visited on 05/25/2025).

OpenMMLab (2020). *OpenPCDet Getting Started Guide*. GitHub repository. Retrieved on Aug. 12, 2025. URL: https://github.com/open-mmlab/OpenPCDet/blob/master/docs/GETTING_STARTED.md.

Peng, Yan, Dong Qu, Yuxuan Zhong, Shaorong Xie, Jun Luo, and Jason Gu (2015). "The obstacle detection and obstacle avoidance algorithm based on 2-d lidar". In: *2015 IEEE international conference on information and automation*. IEEE, pp. 1648–1653.

Phan, Anh Viet, Minh Le Nguyen, Yen Lam Hoang Nguyen, and Lam Thu Bui (2018). "Dgcnn: A convolutional neural network over large-scale labeled graphs". In: *Neural Networks* 108, pp. 533–543.

Qi, Charles R., Hao Su, Kaichun Mo, and Leonidas J. Guibas (July 2017a). "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 652–660.

– (2017b). "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 77–85. DOI: 10.1109/CVPR.2017.16. (Visited on 05/25/2025).

Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas (2017). "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space". In: *Advances in Neural Information Processing Systems (NIPS) 30*, pp. 5099–5108.

Qian, Rui, Divyansh Garg, Yan Wang, Yurong You, Serge Belongie, Bharath Hariharan, Mark Campbell, Kilian Q Weinberger, and Wei-Lun Chao (2020). "End-to-end pseudo-lidar for image-based 3d object detection". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 5881–5890.

Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun (2017). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6, pp. 1137–1149. DOI: 10.1109/TPAMI.2016.2577031. URL: https://www.scirp.org/reference/referencespapers?referenceid=2604248 (visited on 05/25/2025).

Sapkota, Ranjan, Konstantinos I Roumeliotis, Rahul Harsha Cheppally, Marco Flores Calero, and Manoj Karkee (2025). "A review of 3d object detection with vision-language models". In: *arXiv preprint arXiv:2504.18738*.

Shi, Shaoshuai, Chaoxu Wang, Jian Li, Zhiqiang Zhu, Anqi Huang, and Xiaogang Wang (2020). "PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10529–10538. URL: https://openaccess.thecvf.com/content_CVPR_2020/papers/Shi_PV-RCNN_Point-Voxel_Feature_Set_Abstraction_for_3D_Object_Detection_CVPR_2020_paper.pdf (visited on 05/25/2025).

Shi, Shaoshuai, Chaoxu Wang, Zhe Zhang, Liwei Li, Jian Zhang, Xiaogang Chu, and Xiaogang Wang (2022). *PV-RCNN++: Semantical Point-Voxel Feature Interaction for 3D Object Detection*. arXiv: 2208.13414. URL: https://arxiv.org/pdf/2208.13414 (visited on 05/25/2025).

Shi, Shaoshuai, Xiaogang Wang, and Hongsheng Li (2019). "Pointrcnn: 3d object proposal generation and detection from point cloud". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 770–779.

Trigka, Maria and Elias Dritsas (2025). "A Comprehensive Survey of Machine Learning Techniques and Models for Object Detection". In: *Sensors* 25.1, p. 214.

Vinodkumar, Prasoon Kumar, Dogus Karabulut, Egils Avots, Cagri Ozcinar, and Gholamreza Anbarjafari (2024). "Deep Learning for 3D Reconstruction, Augmentation, and Registration: A Review Paper". In: *Entropy* 26.3, p. 235. DOI: 10.3390/e26030235. URL: https://doi.org/10.3390/e26030235 (visited on 05/25/2025).

Wang, Yu, Shaohua Wang, Yicheng Li, and Mingchun Liu (2024). "A comprehensive review of 3d object detection in autonomous driving: Technological advances and future directions". In: *arXiv preprint arXiv:2408.16530*.

Wang, Yue, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon (2019). *Dynamic Graph CNN for Learning on Point Clouds*. ResearchGate. URL: https://www.researchgate.net/publication/322694871_Dynamic_Graph_CNN_for_Learning_on_Point_Clouds (visited on 05/25/2025).

Xu, Qiangeng, Yiqi Zhong, and Ulrich Neumann (2022). "Behind the curtain: Learning occluded shapes for 3d object detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 3, pp. 2893–2901.

Yang, Pei, Ying Luo, Wenhan Lin, and Raquel Urtasun (2020). "3DSSD: Point-based 3D Single Stage Object Detector". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11067–11076. URL: https://paperswithcode.com/paper/3dssd-point-based-3d-single-stage-object (visited on 05/25/2025).

Zhang, Hao, Xiaowei Li, Lihua Chen, and Wei Liu (2023). "ADSSD: Improved Single-Shot Detector with Attention Mechanism and Dilated Convolution". In: *Applied Sciences* 13.6, p. 4038. DOI: 10.3390/app13064038. URL: https://www.mdpi.com/2076-3417/13/6/4038 (visited on 05/25/2025).

Zhou, Yin and Oncel Tuzel (2018). "Voxelnet: End-to-end learning for point cloud based 3d object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4490–4499.