CHALMERS
UNIVERSITY OF TECHNOLOGY

Training end-to-end planners in sensor-level simulation

Master's thesis in Complex Adaptive Systems

MIGUEL ÁLVAREZ GUINARTE

DEPARTAMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Training end-to-end planners in sensor-level simulation

MIGUEL ÁLVAREZ GUINARTE

CHALMERS
UNIVERSITY OF TECHNOLOGY

Computer Vision Group
Chalmers University of Technology
Gothenburg, Sweden 2026

Training end-to-end planners in sensor-level simulation
MIGUEL ÁLVAREZ GUINARTE

Supervisor: Joakim Johnander Faxén, Zenseact
Supervisor: Bernardo Taveira, Zenseact
Examiner: Fredrik Kahl, Department of Electrical Engineering

Master's Thesis 2026
Department of Electrical Engineering
Computer Vision Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +34 648 43 36 65

Cover: Reconstructed scene running on the simulator with a trajectory output from a trained planner.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Training end-to-end planners in sensor-level simulation
MIGUEL ÁLVAREZ GUINARTE
Department of Electrical Engineering
Chalmers University of Technology

Abstract

End-to-end autonomous driving planners are commonly trained with imitation learning on offline expert demonstrations, an approach that can achieve strong open-loop performance. However, planners following this paradigm usually suffer during closed-loop deployment, since they have not been exposed to the consequences of their own actions during training. Addressing this requires closed-loop training, which has typically involved high-level representations, forgoing the benefits of sensor-level end-to-end planning.

This work investigates whether recent advances in efficient 3D Gaussian Splatting can make closed-loop training of sensor-level end-to-end planners feasible. A closed-loop simulator is developed by reconstructing driving sequences and integrating them with a trajectory tracker and a kinematic vehicle model capable of executing the planner's predicted trajectories. This makes it possible to render new views that deviate from the original logged trajectory. The resulting framework is used to fine-tune a pretrained Latent TransFuser planner through reinforcement learning and closed-loop imitation learning strategies.

The simulator shows that closed-loop execution reveals failure modes that are not visible from open-loop evaluation, highlighting the importance of closed-loop training and evaluation. The explored training strategies produced mixed but some modest improvements over the baseline, supporting the feasibility of Gaussian Splatting-based sensor-level simulation as a training platform and laying the groundwork for future work on scalable closed-loop learning for end-to-end autonomous driving.

Keywords: autonomous driving, end-to-end planning, closed-loop training, sensor-level simulation, 3D Gaussian Splatting, imitation learning, reinforcement learning.

Acknowledgements

I really must thank my family, especially my mom and dad. Without their continuous and unconditional support, I would never have had the opportunity to be abroad, experiencing all the amazing things that I have been fortunate to do so far in Sweden.
I would also like to thank my friends, both from home and those I have met along the way during my studies here, for their support and companionship. Special thanks go to my supervisors, Joakim Johnander and Bernardo Taveira, for guiding me throughout this thesis, answering my questions, and giving valuable feedback on my ideas.
Finally, a last thank you is owed to the Chalmers Formula Student team, who not only made me feel at home during this time but also provided a needed distraction whenever the thesis work felt unclear or overwhelming.

Miguel Álvarez Guianrte, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

3DGS	3D Gaussian Splatting
BEV	Bird's-Eye View
CL	Closed Loop
CL-IL	Closed-Loop Imitation Learning
CNN	Convolutional Neural Network
COG	Center of Gravity
CUDA	Compute Unified Device Architecture
DDP	Distributed Data Parallel
D-SSIM	Differentiable Structural Similarity Index Measure
E2E	End to End
EPDMS	Extended Predictive Driver Model Score
GAE	Generalized Advantage Estimation
IL	Imitation Learning
LQR	Linear Quadratic Regulator
LTF	Latent TransFuser
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
PID	Proportional-Integral-Derivative
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SE(2)	Special Euclidean Group in Two Dimensions
SE(3)	Special Euclidean Group in Three Dimensions
SfM	Structure from Motion
SSIM	Structural Similarity Index Measure

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Autonomous driving has been an active field of research for several decades, with approaches and system architectures that have been evolving significantly over time. Early autonomous driving systems [10, 11] were essentially based on rule-based methods, relying on deterministic algorithms to interpret sensor data and predefined logic and geometric reasoning to generate driving commands or control outputs.
As the field started developing, and computational capabilities improved, autonomous driving systems gradually adopted more structured architectures. The paradigm shifted towards modular pipelines where the overall driving task was decomposed in different blocks, such as perception, localization, prediction, planning, or control. Having different modules in charge of specific sub-tasks improved the overall debuggability and maintainability. Moreover, it made the field aligned with the classical robotics engineering principles.
The rapid development of machine learning, and in particular deep learning, has had a huge impact on the field. Data-driven models began to outperform classical algorithms, especially in perception tasks such as object detection, semantic segmentation, and sensor fusion [12, 13, 14, 15, 16, 17], as well as in tracking [18, 19]. As a consequence, these learning-based approaches became increasingly integrated into the modular autonomous driving stacks, while the higher-level decision-making and planning components often remained algorithmic and model-based. Although more recently, researchers have also explored using deep learning to directly map object-level perception outputs to trajectories, as seen in works like ChauffeurNet [20], or UrbanDriver [21].
Despite their widespread adoption, modular architectures also exhibit inherent challenges and limitations. Errors can propagate across module boundaries [1], and intermediate representations can possibly discard relevant information for downstream modules [22]. In response to these issues, recent research has explored alternative paradigms that aim to go from this modularized idea into a holistic pipeline directly mapping raw sensor inputs into driving actions [23, 1, 22].
Beyond architectural changes, there is also a paradigm shift in the way these learning models are trained. A large fraction of learning-based autonomous driving systems are trained using Imitation Learning (IL), in which the model learns to mimic expert behavior from recorded human driving data. While this perspective benefits from readily available datasets, it has some inherent limitations. In particular, due to the open-loop nature of IL, the model is not exposed to the consequences of its

own actions during training and therefore does not learn how errors can compound over time [24]. This leads to a significant mismatch between open-loop training and closed-loop deployment, as shown in works like NAVSIM [25].
Several extensions of imitation learning, such as dataset aggregation [24], aim to partially alleviate this issue by iteratively collecting data under the learner's policy. However, these approaches require expert labels for the states visited by the learner, which is usually very difficult to obtain in autonomous driving when using only logged data. Moreover, they still rely on supervised imitation and therefore do not directly optimize closed-loop objectives either.
To address these issues, recent work has increasingly explored closed-loop training strategies, in which the learning agent interacts directly with an environment and adapts its behavior based on the observed outcomes of its actions. Such approaches have been investigated both at the object level [2] and at the sensor level [26] predominantly in simulation environments such as CARLA [27]. Although this idea has shown promising results and is expected to be a major focus of future research, significant challenges remain. End-to-end planners cannot be fully evaluated in object-level simulators, since these do not generate raw sensor observations. And sensor-level simulators such as CARLA still struggle to close the sim-to-real gap.
Beyond the technical challenges, autonomous driving research is closely linked to broader societal and ecological considerations. The development of autonomous driving technologies has the potential to improve road safety, increase accessibility to mobility, and reduce traffic congestion through more efficient driving strategies. In addition, autonomous driving may contribute to ecological sustainability by enabling smoother traffic flow and more energy-efficient driving, potentially reducing overall energy consumption and emissions. These broader considerations provide important context for the continued development of autonomous driving systems and motivate ongoing research in this field.

1.2 Problem Statement

Modern end-to-end driving approaches are still commonly trained using imitation learning on offline expert demonstrations. This training paradigm can lead to strong open-loop performance, where the model is evaluated on states from the expert data distribution. However, during closed-loop deployment, the predicted actions affect the future state of the vehicle and consequently the subsequent observations. Even small prediction errors can shift the planner away from the training distribution, causing errors to compound over time and completely degrade the reliability under closed-loop execution [28, 29].
At the same time, most existing closed-loop training approaches that aimed to address this issue, operated on structured object-level, state-level, or bird's-eye-view representations rather than directly on raw sensor observations [2, 30]. These representations enable efficient simulation and large-scale policy optimization, but they abstract away the sensor-rich information that end-to-end planners exploit to maximize performance [1].
Closed-loop training at the sensor level has also been explored, for example in synthetic simulation environments such as CARLA [31]. While such environments

make interaction possible, their visual realism, sensor characteristics, and scenario diversity remain limited compared to real-world driving data. More recent neural-rendering-based simulators improve sensor realism by reconstructing scenes from logged driving data, and have been used for photorealistic closed-loop testing of autonomous driving systems [32]. Nevertheless, these methods introduce their own limitations, since rendering can be computationally expensive and the fidelity of the generated observations may degrade as the ego vehicle deviates from the original data-collection trajectory.
This work investigates whether recent advances in efficient neural rendering, such as SplatAD [8], can make closed-loop training of sensor-level end-to-end planners feasible. In particular, the question is to determine whether the limitations of open-loop imitation-learned end-to-end planners can be reduced through exposure and interaction with a closed-loop realistic sensor-level simulator.

1.3 Motivation

End-to-end planners are ultimately deployed in closed loop, where their predictions affect future vehicle states and sensor observations. Therefore, evaluating and training them only in open loop provides an incomplete picture of their real performance. Developing methods for closed-loop training at the sensor level is necessary because it addresses this mismatch and helps study how the planner interacts with the environment when the model is exposed to the consequences of its own actions.
From a practical perspective, realistic sensor-level simulation could improve robustness and reduce the dependence on expensive and safety-critical real-world testing. If this kind of neural-rendered environments can provide sufficient realism while remaining computationally feasible, they may serve as a bridge between offline imitation learning and real-world deployment.

1.4 Aim

Building on the problem statement and motivation presented above, the aim of this thesis has been to:

- Develop a realistic closed-loop simulation environment for end-to-end planning models, using neural rendering of logged driving scenes.
- Enable the generation of lightweight, traversable environments where planners can deviate from original trajectories while maintaining closed-loop behavior.
- Train planners under closed-loop interaction, allowing them to learn from feedback generated by their own actions.
- Investigate closed-loop training directly at the sensor level, leveraging rich sensory observations, which aligns with recent trends in end-to-end planning approaches [1, 33, 34].

In particular this work aims to answer the following research questions:
To what extent can sensor-level simulators be used to feasibly train end-to-end planners in a closed-loop setting?

While closed-loop training has demonstrated significant advantages over classical imitation-learning and open-loop approaches, its application to sensor-level planners introduces unique challenges. Unlike planners operating at an object or state-level representation, sensor-level planners require raw sensor inputs, which increases the computational and simulation complexity of both training and evaluation. This project investigates whether the benefits of closed-loop learning can be effectively extended to end-to-end planning at the sensor level, despite these challenges.

How does closed-loop training in sensor-level simulation compare to open-loop or imitation learning in terms of performance, robustness, and generalization for end-to-end planners?
This question investigates whether closed-loop training in sensor-level simulation can be used to improve end-to-end planners by refining models initially trained in open-loop or imitation learning settings. Prior work, such as NeuroNCAP [32], has shown that IL–based planners can suffer from error accumulation and trajectory drift when executed in closed-loop at the sensor level. Consequently, the focus is on assessing whether exposure to closed-loop training leads to improved robustness against these effects, as well as enhanced generalization to scenarios that differ from those encountered during training.

Which training and interaction strategies most effectively leverage closed-loop sensor-level simulators under their practical constraints?
Despite recent advances in sensor-level rendering, closed-loop simulators are often constructed from scenes logged during human driving. As a result, the fidelity of rendered sensor observations may degrade as the planner deviates significantly from the original data collection trajectory, limiting the extent to which the environment can be freely explored. This research question analyzes how these limitations affect closed-loop training of end-to-end planners and investigates training and interaction strategies, such as constrained exploration or selective closed-loop rollouts, which can mitigate these effects and enable effective scaling of learning within these types of simulators.

1.5 Scope and Limitations

The intent of this thesis has not been to propose a new end-to-end planning architecture or to train such a model from scratch. Instead, the work has used existing planner architectures, initialized from pretrained checkpoints, for fine-tuning within the proposed closed-loop simulation framework. The focus lies on the effects of closed-loop sensor-level simulation, not on architectural comparisons or benchmarks. The simulation environment relies on neural rendering techniques applied to an open-source driving dataset, in particular the Argoverse 2 Sensor Dataset [35]. Therefore, the scale, diversity, and variability of rendered environments, as well as the number of scenarios explored, are bounded by the content available in this dataset. In addition, the simulation inherits limitations from the underlying neural rendering representations, which may affect visual fidelity and scene consistency when the ego vehicle deviates from the original data-collection trajectories.

Finally, the scope of this thesis is limited to isolated planning behavior within simulated environments. Broader global aspects, such as large-scale traffic interactions, multi-agent coordination, and the impact of the planners on overall traffic flow, might not have been explicitly addressed and remain outside the scope of this work.

1.6 Contributions

The main contributions of this thesis can be summarized in the following list:

- Development of a closed-loop sensor-level simulator built from neural-rendered reconstructions of logged driving sequences.
- Implementation and evaluation of several closed-loop fine-tuning strategies, including PPO-based reinforcement learning, as well as several approaches to closed-loop imitation learning.
- Design of a complementary evaluation procedure inside the simulator, measuring closed-loop metrics such as completion rate, collision rate, off-road rate, lateral error, longitudinal error, and velocity error.

2
Background

This chapter introduces the prerequisites required to understand the work carried out in this thesis. It begins by presenting planning in autonomous driving, with a particular focus on end-to-end planners and the representations they use. It then explains the distinction between open-loop and closed-loop learning, and why this difference is important when training and evaluating planners.
The chapter further discusses different simulation paradigms for autonomous driving, including object-level and sensor-level simulation. Neural rendering is introduced as a means of constructing realistic sensor-level environments from logged data. Finally, the chapter presents the basic concepts needed to understand the vehicle motion model and the learning schemes used to adapt planners within the proposed simulation framework.

2.1 End-to-End Planning for Autonomous Driving

There are many different approaches and architectural solutions when it comes to end-to-end planning. However, all of them stem from the same premise, learning a direct mapping from sensor observations to driving outputs such as trajectories or control commands. In contrast to modular pipelines, on where perception and planning are often designed and trained as separate components.
This holistic formulation allows the model to learn task-relevant representations directly from the data, preserving information that could be lost when using the intermediate representations. In this sense, what characterizes an end-to-end planner is not only the use of raw sensor inputs, but the fact that the perception and decision-making components are trained together with respect to the final planning objective.

2.1.1 Modular vs End-to-End Architectures

The important aspect to make a proper disinction between both paradigms, is to look at how the information flows through them, as well as how they are trained.

2.1.1.1 Modular Architectures

In modular architectures, the autonomous driving task is decomposed into a set of specialized submodules, such as perception, localization and mapping, decision

making, trajectory planning, and control. Each module is designed and trained to solve a particular subproblem, and information is passed between modules through predefined intermediate representations. Figure 2.1 shows a common example of such a pipeline.

The sensor inputs are first processed by the perception and localization and mapping modules. The perception module is typically responsible for extracting a description of the surrounding environment, for example by detecting relevant agents and estimating their classes and the positions and orientations of their bounding boxes. Notice that this module is trained with perception-specific objectives, such as classification and bounding-box regression losses. These objectives evaluate how accurately the module detects and localizes objects, but they are not directly related to the quality of the trajectory that will eventually be planned by the following modules.

Similarly, the localization and mapping module estimates quantities such as the ego pose, local map, lane or track boundaries, and the geometric relation between the vehicle and its driving environment. These outputs provide the spatial context required by the planner, but the module is again optimized according to its own objective, such as localization accuracy and map consistency, rather than the final driving performance.

The outputs of perception and localization and mapping are then passed to a decision-making module. This module produces a high-level driving directive, such as following the lane, yielding, overtaking, avoiding an obstacle, or stopping. In a modular setting, the decision-making module is usually trained or designed using objectives related to the correctness of this high-level decision. However, this objective is still decoupled from the quality of the final trajectory generated by the trajectory planner. For example, a decision labeled as correct at the behavioral level may still lead to a poor or infeasible trajectory depending on how the subsequent planning module interprets it.

Finally, the trajectory planning module uses the selected behavior, the estimated ego state, the perceived objects, and the available map information to generate a feasible reference trajectory. This trajectory is usually optimized with respect to geometric and dynamic criteria, such as obstacle avoidance, smoothness, curvature, comfort, and vehicle constraints. Therefore, the trajectory planner depends strongly on the quality and structure of the intermediate outputs provided by previous modules.

The central characteristic of modular pipelines is therefore that the system is not optimized end-to-end with respect to the final trajectory or driving objective. Instead, each submodule is optimized independently using task-specific losses or design criteria. This has some favorable points, such as that it improves interpretability and allows each component to be developed, tested, and replaced separately. However, it also introduces a mismatch between local module performance and global driving performance. For example, good object detection and correct high-level decisions do not necessarily guarantee an optimal final trajectory.

It is worth noting that, in a complete autonomous driving stack, an additional control module is typically required to translate the planned trajectory into low-level actuator commands, such as steering, throttle, and braking. However, since this thesis is focused on trajectory-based planners, the control module is omitted

from Figure 2.1.

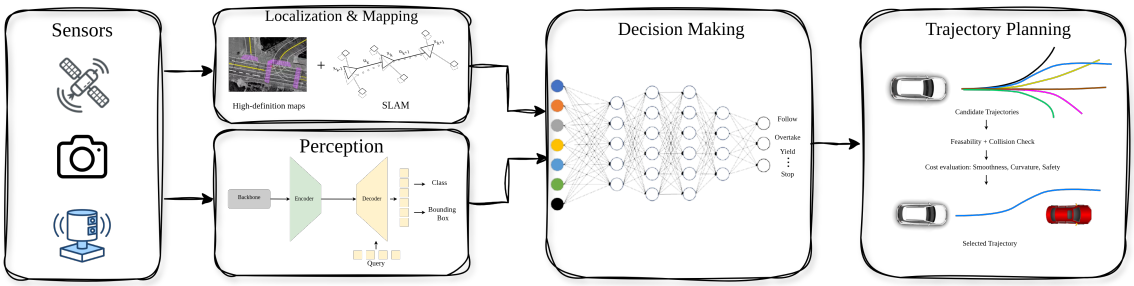


Figure 2.1: Basic modular pipeline showing where each task is executed by a different submodule.

2.1.1.2 End-to-End Architectures

In contrast to modular pipelines, end-to-end architectures aim to coordinate their modules within a unified learning framework. Rather than optimizing each component independently, these methods are designed so that intermediate representations are learned for the final driving objective. In this sense, the architecture is planning-oriented, where perception and prediction are not only evaluated as standalone tasks, but are trained to provide information that is useful for generating the future ego trajectories.
One of the first examples of this philosophy is UniAD [1], whose overall pipeline is shown in Figure 2.2. UniAD still contains recognizable components for perception, mapping, motion, occupancy prediction, and planning. However, these components are connected within a single model and are trained jointly, so that information can flow across tasks without fixed hand-designed interfaces.

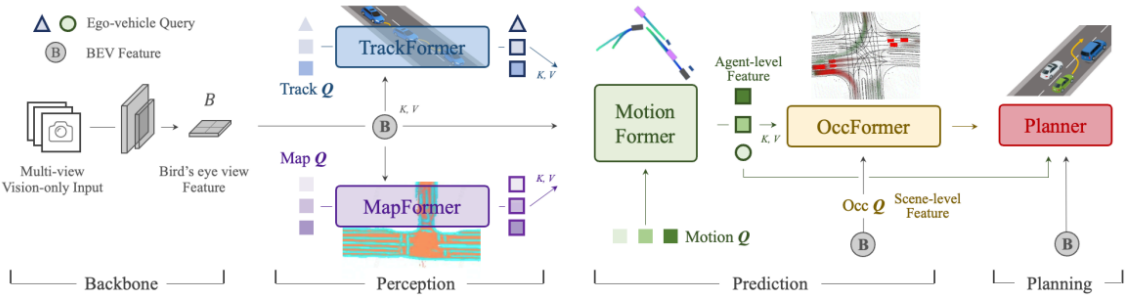


Figure 2.2: Unified Autonomous Driving (UniAD) end-to-end pipeline. Figure reproduced from [1], © 2023 IEEE.

A key mechanism in UniAD is its query-based design. In transformer-based models, queries can be interpreted as learned representations that attend to relevant information in a set of encoded scene features. UniAD uses different types of queries to represent different driving-relevant tasks. For example, tracking queries are used to reason about dynamic agents, map queries represent static road elements, motion queries capture future agent behavior, and an ego-vehicle query is used by the

planner to generate the final planning output. The important detail is that these queries provide a flexible interface between modules. Which, now, do not produce a completely independent output but a learned representation that can be reused and refined by later modules.
As we can see from this example, end-to-end architectures do not necessarily remove all intermediate tasks or representations. Instead, they make these representations differentiable, learned, and optimized together with the final planning objective. For instance, in UniAD, this is reflected in its two-stage training strategy: the perception-related components are first jointly trained for a few epochs. And only after that, the full network is trained end-to-end for longer.

2.1.2 Object-Level, BEV-Level, and Sensor-Level Representations

This subsection aims to illustrate the different forms of scene representation that autonomous driving planners usually operate on. These representations differ in how much processing has already been applied to the raw sensor data before it reaches the planner. At one extreme, object-level representations describe the environment using structured entities such as vehicles, pedestrians, and traffic elements. At the other extreme, sensor-level representations provide the planner with raw or minimally processed sensor observations, typically camera images or LiDAR point clouds. Between these two cases, bird's-eye-view (BEV) representations provide a spatially organized intermediate representation of the scene.
Object-level representations are among the most classical and widely used inputs. In this formulation, the state of the environment is described by a set of discrete objects and road elements. Dynamic agents are typically represented by bounding boxes, class labels, positions, headings, and sometimes predicted future trajectories. Static scene elements like crosswalks, and traffic lights, may also be included in a structured map-like form.
These object-level representations can appear either in two-dimensional form as shown in Figure 2.3, or in full three-dimensional form as in Figure 2.4.

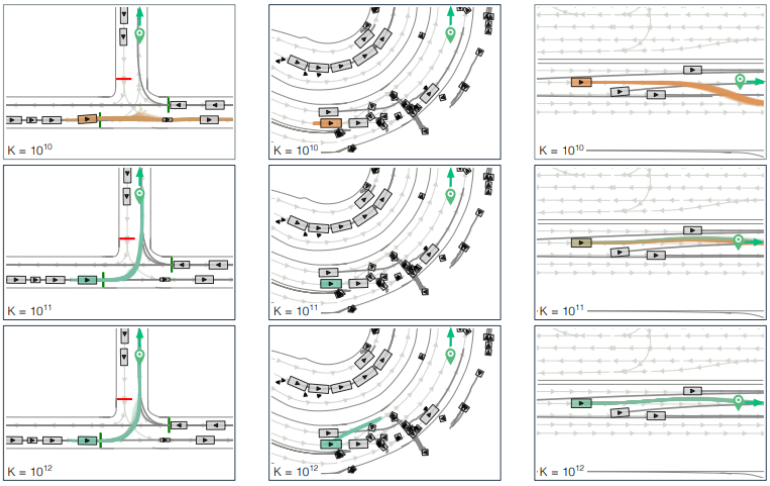


Figure 2.3: Object-level representation used in GIGAFLOW. Taken from *Robust Autonomy Emerges from Self-Play* by Cusumano-Towner et al. [2], published in the Proceedings of the 42nd International Conference on Machine Learning, PMLR 267. Available from PMLR: `https://proceedings.mlr.press/v267/cusumano-towner25a.html`. Licensed under CC BY 4.0.

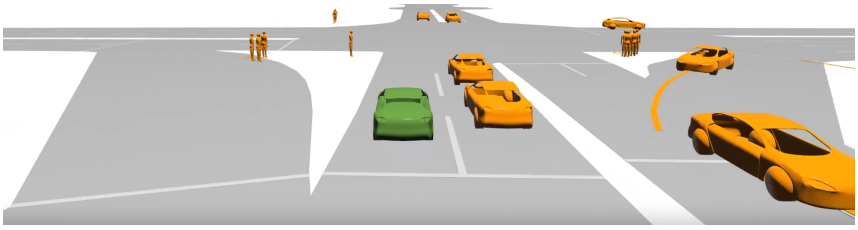


Figure 2.4: 3D object level representation from ViP3D. Figure taken from [3], © 2023 IEEE.

The 2D formulation is often sufficient for motion planning and interaction reasoning, since most driving decisions depend primarily on the position and motion of agents on the road surface. In contrast, 3D object-level representations are more closely connected to perception and tracking, where object height, sensor visibility, and spatial localization are important.
The main advantage of these object-level representations is that they provide a compact and interpretable description of the driving scene. The planner does not need to process the raw data directly and simulations can be made computationally efficient. That is why we usually see them on large-scale closed-loop training or reinforcement learning as in [2, 30, 36]
On the other side, this abstraction also introduces limitations. Since the planner only receives the objects and map elements provided by the perception or simulation system, the representation is less feature-rich than the original representation. Some details like occlusions, unusual objects, and sensor-specific effects are lost. Additionaly, when object-level simulators provide ground-truth objects directly, they may bypass the perception problem and therefore fail to expose the planner to errors that would occur in the real system.

A related intermediate representation is the BEV representation, depicted in Figure 2.5. In BEV, the scene is represented from a top-down perspective, usually in a coordinate frame attached to the ego vehicle or world frame. In comparison to, for example, perspective camera images, BEV representations make distances, relative positions, and trajectories easier to reason about geometrically. The typical approach is that the sensor information is represented in BEV and from that, features are directly extracted.

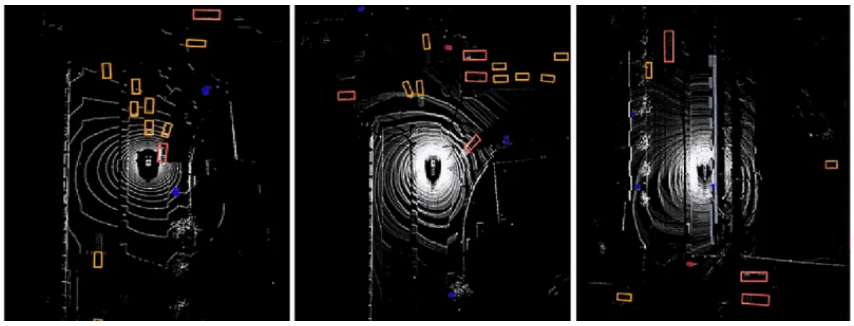


Figure 2.5: BEV representation achieved by BEVFusion [4], © 2023 IEEE.

BEV have been widely used in LiDAR-based perception, where point clouds can be discretized into top-down grids before being processed by other models. For example, PointPillars [37] encodes a point cloud into vertical pillars and applies a 2D convolutional detection network on the resulting pseudo-image representation. Additionally, BEV representations have also become common in camera-based perception. For example Lift,Splat,Shoot [38] project multi-view image features into a single cohesive BEV frame. Another remarkable example is BEVFormer[13] which uses learnable BEV queries and attention mechanisms to aggregate spatial and temporal information from multiple camera views.
The advantage with BEV is that it preserves more spatial structure than a pure object-level representation while remaining more compact and planning-friendly than raw sensor data.
However, BEV is still an abstraction of the original sensor observations and therefore its quality depends on aspects like the perception model, depth estimation, and the resolution of the BEV grid. And there are still details that are difficult to project into the top-down frame, such as vertical structure.
Finally, sensor-level representations are gaining a lot of importance recently. In a context of a difficult task as autonomous driving, it is sensible to leverage data at its maximum and extract as much usable information for the end goal as possible.
A sensor-level representation implies that the raw sensor data is available for the model, in the sense that even if middle representations or data transformations occur, the planner is still allowed to learn from the complete sensors picture and use the information directly in the planning if needed.
Finally, sensor-level representations provide the planner with raw or minimally processed sensor observations, such as camera images, LiDAR point clouds, radar measurements, or combinations of them. The motivation for this representation is that autonomous driving is a visually and geometrically complex task, where relevant information is not always captured by a predefined object list or semantic map.

Therefore, allowing the model to process the sensor observations directly can, in principle, make the planner exploit appearance information that may be lost in higher-level representations.
It is important to note that sensor-level planning does not imply that the model avoids intermediate features. In fact, in practice, the most common thing is still to transform the raw observations into learned internal representations, such as image features, attention tokens, or latent scene embeddings. The difference is that these representations are learned as part of the planner, rather than being provided as a fixed object-level interface. As a consequence, the planner is allowed to decide which parts of the sensor information are relevant for the final planning objective.
Early end-to-end works like [39] already presented this idea of the internal components of the planner, self-optimizing themselves to maximize the overall planner performance. Naturally, this type of approach was relatively simple compared to modern planners, yet it introduced an important principle that is today followed by many recent end-to-end planners, but with richer inputs and larger and more modern architectures.

2.2 Open-Loop and Closed-Loop Learning

This section introduces the distinction between open-loop and closed-loop learning in autonomous driving, and explains why this distinction is important for planner training and evaluation. While open-loop settings use logged expert data, closed-loop settings expose the model to the consequences of its own predictions. As a result, closed-loop performance can reveal failures that are not visible from open-loop behaviour alone.

2.2.1 Imitation Learning and Open-Loop Training

In open-loop training, the model observes data collected by an expert.

$$\mathcal{D} = \{(o_t, a_t^*)\}_{t=0}^{N}, \tag{2.1}$$

where o_t is the observation at time t, and a_t^* is the corresponding expert action or trajectory.
For trajectory-based planners, at each time step, the model predicts a future trajectory given the current observation and state:

$$\pi_\theta(o_t, s_t) \rightarrow \hat{\tau}_t \tag{2.2}$$

where this future trajectory is usually just a sequence of predicted future ego poses:

$$\hat{\tau}_t = \left[(\hat{x}_{t+1}, \hat{y}_{t+1}, \hat{\psi}_{t+1}), \ldots, (\hat{x}_{t+H}, \hat{y}_{t+H}, \hat{\psi}_{t+H})\right] \tag{2.3}$$

on where H is the planner's prediction horizon. This prediction is then compared against the expert target, producing an imitation-learning loss of the form:

$$\mathcal{L}_{IL} = \sum_t \ell(\pi_\theta(o_t, s_t), a_t^*) \tag{2.4}$$

which, for trajectory prediction, can be implemented as the L2 loss between the expert's and the planner's trajectories:

$$\mathcal{L}_{IL} = \sum_t \|\hat{\tau}_t - \tau_t^*\|^2 \quad . \tag{2.5}$$

Some other approaches extend this naive objective with additional loss terms. For example, UniAD [1] combines the trajectory imitation loss with a collision loss term defined as:

$$\mathcal{L}_{\text{col}}(\hat{\tau}, \delta) = \sum_{i,t} \text{IoU}\left(\text{box}(\hat{\tau}_t, w+\delta, l+\delta), b_{i,t}\right) \quad , \tag{2.6}$$

where IoU denotes the intersection over union function. $\text{box}(\hat{\tau}_t, w+\delta, l+\delta)$ denotes the ego-vehicle bounding box placed at the planned pose $\hat{\tau}_t$, with width w and length l enlarged by a safety margin δ. The term $b_{i,t}$ denotes the bounding box of agent i at time t.
However, the important clarification is that the model's prediction is not executed in the environment. Instead, the next input is still the next observation from the recorded expert trajectory. The attractiveness in this setup is that it just needs existing driving logs, which are now reasonably abundant and accessible. It is also simple, stable to train, and does not require computationally expensive interaction with a simulated environment. As a consequence, open-loop imitation learning is highly scalable.
On the other side, the counterpoint is that this approach does not expose the model to its own mistakes. And it measures prediction accuracy on expert states, rather than the ability to recover from model-induced states. Which is a desired characteristic when it comes to real deployment.

2.2.2 Distribution Shift and Compounding Errors

The main limitation of open-loop training is the mismatch between the state distribution seen during training and the one encountered during real rollout. During training, the model observes states generated by the expert policy:

$$s_t \sim d^{\pi^*} \quad .$$

However, during deployment the model is going to experience states that are induced by its own policy:

$$s_t \sim d^{\pi_\theta} \quad .$$

And the issue is that in general, these two distributions are different:

$$d^{\pi^*} \neq d^{\pi_\theta} \quad .$$

And this distribution shift can lead to compounding errors. For example, a planner may initially predict a trajectory that is only slightly shifted laterally from what the expert would have done. In open-loop evaluation, this would appear as a small prediction error and may still result in a good score. In closed-loop execution, however, the shifted trajectory is actually followed by the vehicle. Therefore, the

next scene observation is obtained from a different ego pose than the one visited by the expert. This means that the model is now outside its familiar training distribution, which increases the chances of larger prediction errors at the following step. Even if the next error still looks small, these errors can accumulate over time, and eventually lead to trajectory drift, off-road behavior, or a collision. A comprehensive comparison of this difference can be seen in Figure 2.6.

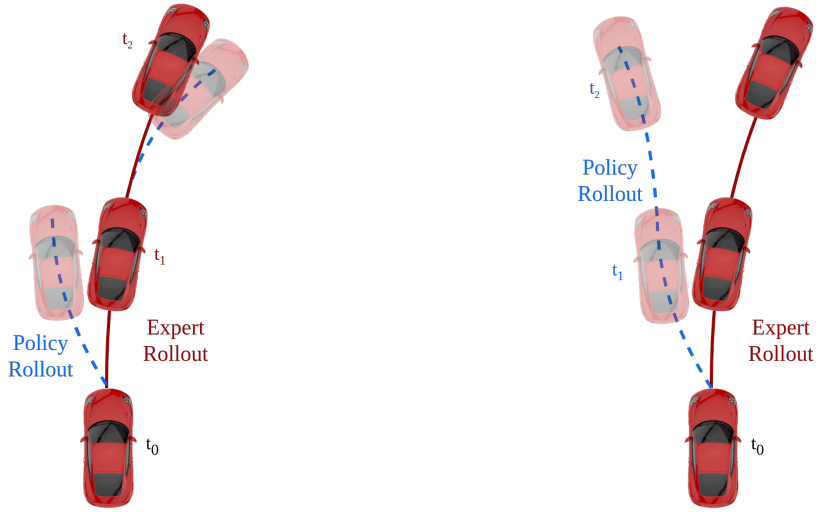


Figure 2.6: Open loop versus closed loop comparison. On the left, we can see that the model is producing trajectories that are never followed. Next samples are instead taken from the expert's rollout. On the right, we can appreciate how the policy is actually executed and sampled from. The first error compounds and becomes much larger in the end when compared to the left version.

Note as well that these effects are even more relevant for sensor-level planners. A small change in ego pose not only corresponds to small object shifts, but it also changes the perceived camera viewpoint, visibility, occlusions, and other geometric relationships in the scene.

2.2.3 Closed-Loop Training and Evaluation

To mitigate the problems caused by distribution shift, the planner must be exposed to states that arise from its own behavior.
In practice, real-world closed-loop training is difficult because unrestricted interaction with a real vehicle is costly, time-consuming, and usually unsafe. Real-world closed-loop evaluation is also limited, since safety-critical or rare scenarios are difficult to reproduce systematically.
For this reason, prior work has explored alternatives such as data augmentation on top of imitation learning. For example, in [40], the model is exposed to perturbed observations and off-trajectory states in order to improve robustness beyond the nominal expert trajectory. Other approaches attempt to close the loop more directly through online imitation learning, reinforcement learning, or supervised fine-tuning in interactive environments such as simulators or game engines.

Evaluation in a closed loop is also something necessary because open-loop prediction error does not fairly characterize driving performance. As discussed before, a planner may obtain low open-loop error while still failing when its outputs are executed. Conversely, it may deviate from the expert trajectory while still producing a safe and valid alternative maneuver.
Therefore, this motivates the use of closed-loop simulation, where planner outputs can be executed, and their consequences can be observed in a controlled and repeatable environment. The main challenge then becomes designing simulators that are both scalable and realistic enough.

2.3 Closed-Loop Simulation for Autonomous Driving

Having established why closed-loop interaction is needed, this section discusses the main simulation paradigms used to obtain such behaviour. Distinguishing between object-level simulation, synthetic sensor-level simulation, and more realistic sensor-level approaches, highlighting the trade-off between scalability, controllability, and sensor realism.
Before going deeper into the section, let us clarify that a simulator for autonomous driving is understood as a framework capable of generating driving states, and also able to provide action ground truths.

2.3.1 Object-Level Simulation

Object-level representations were introduced in Section 2.1.2, and from this perspective, a natural way to obtain closed-loop interaction is to simulate the environment directly at the object level. On where the simulation just requires representing the scenes via quantities, such as agent positions, headings, velocities, and map elements. This type of simulation is attractive since it is compact, efficient, and easy to scale. Since the simulator does not need to generate raw sensor observations, large numbers of rollouts can be produced at relatively low computational cost. This makes object-level simulation particularly suitable for reinforcement learning, or large-scale closed-loop evaluation, as in [36, 2] or in frameworks as Waymax [30]. An illustration of an objecl-level setup is shown in Figure 2.7.

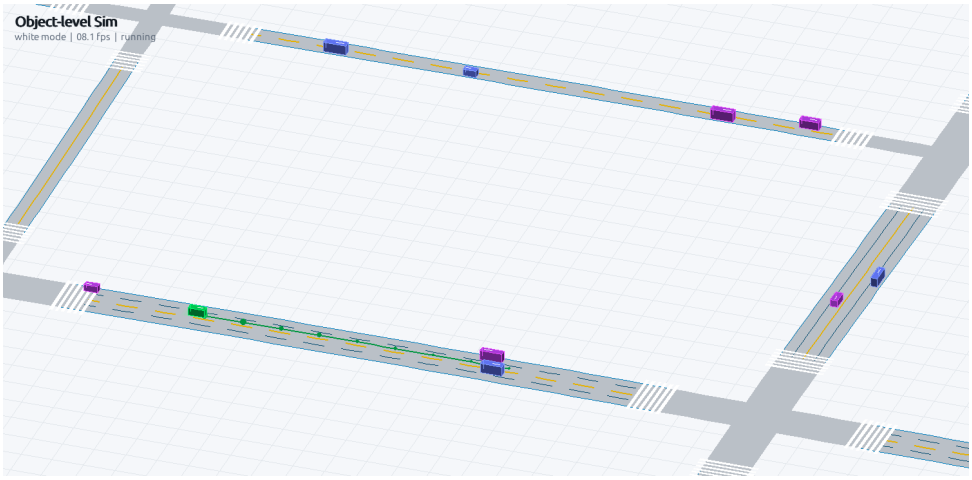


Figure 2.7: Object-level simulator example.

This simulation paradigm is naturally suited to modular pipelines (Section 2.1.1.1), where upstream components handle the perception and tracking before the planner is executed. And therefore, the planner can be trained or evaluated directly and independently, without requiring the simulator to reproduce the full sensor data. However, as discussed previously, end-to-end planners are beneficial and motivated by the direct exploitation of the raw-sensor information.

2.3.2 Synthetic Simulation and Sim-to-Real Gap

If the aim is, therefore, to simulate closed-loop interaction at the sensor level, the simulator must be able to generate realistic sensor data in addition to updating the state of the environment. A classical approach is to employ game-engine rendering techniques together with manually designed virtual environments.
One of the best-known simulation frameworks is CARLA [27], which uses Unreal Engine 4 (UE4) [41]. CARLA provides urban driving environments composed of static elements such as roads, buildings, vegetation, traffic signs, and road infrastructure, together with dynamic actors such as vehicles and pedestrians. These actors can be controlled by built-in traffic logic, allowing the ego vehicle to interact with the surrounding traffic in closed loop. An example visualization from CARLA is shown in Figure 2.8.

Figure 2.8: Visualization from CARLA. Taken from the CARLA documentation, version 0.9.7, "Getting started" [5].

Remark that CARLA is not only well known by being a simulator and training ground, but also has become one of the main tools used to evaluate and benchmark autonomous driving [31].
There are more game-engine-based simulators that follow similar principles, such as Sim4CV [42], GTA V [43], or AirSim [44]. Others like MetaDrive [6], built using the efficient 3D graphics engine Panda3D [45], trades the visual fidelity for high sample efficiency and extensibility, making it suitable for topics such as safe RL and multi-agent autonomy. Figure 2.9 illustrates MetaDrive's ability to compose diverse driving scenarios from modular components.

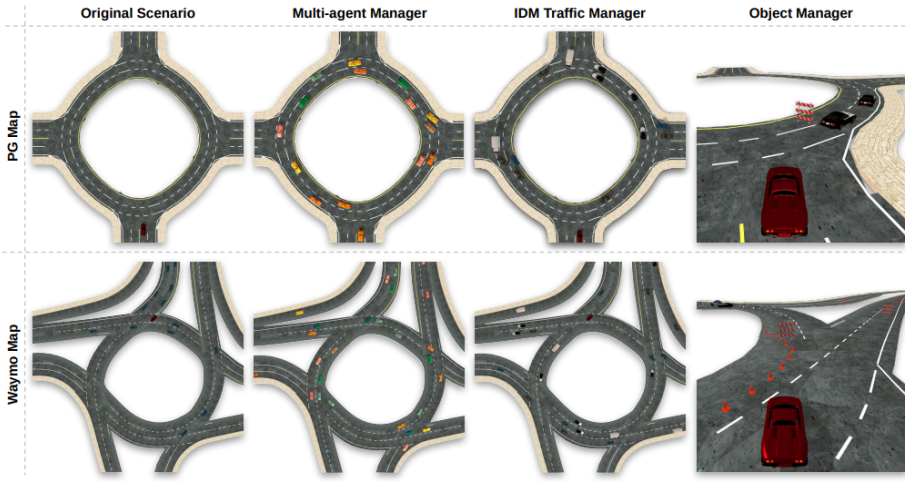


Figure 2.9: New scenarios composition by MetaDrive. Figure taken from [6] © 2022 IEEE. Where they expose the original scenarios to different managers in charge of generating the multi-agent navigation tasks, generating intelligent driving models that respond to traffic, and finally also being able to compose safety-critical environments with an object manager that can randomly add obstacles to the map.

The main advantage of these simulators is their high controllability. They can generate sensor observations while providing full access to the simulation state. Making it possible to generate controlled scenarios, such as rare or safety-critical situations.
However, game engine simulators introduce an important trade-off between realism, flexibility, and scalability. Increasing visual fidelity, traffic diversity and realism requires more complex rendering and more complex behavior models which may cause scalability to be unfeasable. Conversely, lightweight simulators can be easier to scale, but may provide simplified observations and dynamics.
This sim-to-real gap, is even more important for sensor-level planners, since the trajectories depend directly on the rendered observations. Usually, differences in lighting, textures, occlusions and other details, may cause a planner trained on simulation to behave differently in real-world deployment.
These limitations, motivate the need for other simulation approaches, aiming to not only reduce the visual gap between simulation and real-world deployment, but also to provide flexibility to generate sensor observations from different viewpoints while staying lightweight enough to provide scalability.

2.4 Neural Rendering

This section illustrates neural rendering, and in particular, 3D Gaussian Splatting [7] as a different approach capable of reconstructing visually realistic scenarios from logged data.
More traditionally, in 3D computer graphics, explicit surface representations were used to render geometries. For example, polygon meshes composed of interconnected triangles, voxel grids, or parametric surfaces like NURBS[46] are among the most used to render and model graphics. However, using these handcrafted geometric primitives often results in visual artifacts, holes, or high computational costs.
To overcome these limitations, neural rendering emerged as a paradigm shift, popularized by Neural Radiance Fields (NeRF) [47]. Instead of discretizing space into explicit triangles or surfaces, NeRF bypasses traditional geometric primitives entirely by encoding the scene as a continuous volumetric field, i.e., a mathematical function stored within the weights of a neural network that maps a 3D position and viewing direction to color and density. To render a new view, instead of computing the intersection between surfaces, rays are cast from the camera, sampling this function along each ray.

2.4.1 3D Gaussian Splatting

To address the high computational cost of NeRF's ray marching and repeated neural network evaluations, 3D Gaussian Splatting (3DGS) [7] was introduced as a method that combines the benefits of explicit geometric representations with a differentiable rendering pipeline. Unlike NeRF, instead of storing the scene in neural weights, 3DGS represents the scene as a collection of learnable 3D Gaussians.
Each Gaussian i has a mean position $\boldsymbol{\mu}_i \in \mathbb{R}^3$, a covariance matrix $\boldsymbol{\Sigma}_i \in \mathbb{R}^{3\times3}$, an opacity value α_i, and view-dependent color parameters. The spatial influence of a

Gaussian at a specific point x in space is given by:

$$G_i(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T\boldsymbol{\Sigma}_i^{-1}(\mathbf{x}-\boldsymbol{\mu}_i)\right) \quad . \tag{2.7}$$

The covariance matrix determines the size and orientation of the Gaussian. Notice that the optimization of the parameters is done through gradient descent, which is not easy to constrain such that it only produces physically valid (positive semi-definite) covariances. Therefore, the covariance is instead decomposed into a scaling matrix $\mathbf{S}_i$ and a rotation matrix $\mathbf{R}_i$:

$$\boldsymbol{\Sigma}_i = \mathbf{R}_i\mathbf{S}_i\mathbf{S}_i^T\mathbf{R}_i^T \quad , \tag{2.8}$$

On where, $\mathbf{S}_i$ is of the form:

$$\mathbf{S}_i = \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \quad , \tag{2.9}$$

guaranteeing that the eigenvalues remain positive. Additionally, this decomposition of the covariance also allows the Gaussians to be anisotropic, meaning they can be stretched and rotated to align perfectly with thin or flat surfaces in the environment. Then, to render an image, each 3D Gaussian is projected into the camera view. Let $\mathbf{W}$ denote the world-to-camera transformation and let $\Pi(\cdot)$ be the camera projection function. The projected mean of the Gaussian is

$$\mathbf{u}_i = \Pi(\mathbf{W}\boldsymbol{\mu}_i), \tag{2.10}$$

where $\mathbf{u}_i$ is the 2D image location of the projected Gaussian center. The 3D covariance is also projected into image space. Using a local linear approximation of the camera projection, the projected 2D covariance can be written as

$$\boldsymbol{\Sigma}_i' = \mathbf{J}\mathbf{W}\boldsymbol{\Sigma}_i\mathbf{W}^T\mathbf{J}^T \quad , \tag{2.11}$$

where $\mathbf{J}$ is the Jacobian of the projection function evaluated at the Gaussian mean. This produces an elliptical projection in the image plane.
Once the 3D Gaussians are projected into 2D image space, the final image is generated through a process of tile-based rasterization and α-blending. This stage is what allows 3DGS to achieve real-time frame rates while remaining fully differentiable. Instead of sampling many points along every camera ray, as in NeRF-style volume rendering, the tile-based sorting algorithm avoids processing every Gaussian at every pixel. Instead, it divides the image into tiles and performs a culling step, identifying which 2D Gaussians intersect with each tile. Each Gaussian is then assigned a key based on its tile ID and its depth. These keys are sorted using a fast GPU-based sort that guarantees that for any given tile, the Gaussians are processed in a strict front-to-back order.
Continuing with the α-blending. For a given pixel $\mathbf{u}$, the contribution of Gaussian i depends on the distance between the pixel and the projected Gaussian center, measured under the projected covariance $\boldsymbol{\Sigma}_i'$. The effective opacity contribution of Gaussian i at pixel $\mathbf{u}$ is therefore given by

$$\tilde{\alpha}_i(\mathbf{u}) = \alpha_i \exp\left(-\frac{1}{2}(\mathbf{u}-\mathbf{u}_i)^T \mathbf{\Sigma}_i'^{-1}(\mathbf{u}-\mathbf{u}_i)\right) \quad , \tag{2.12}$$

where α_i is the learned opacity of the Gaussian, and $\tilde{\alpha}_i(\mathbf{u})$ is the opacity actually contributed to the pixel. Thus, a Gaussian contributes most strongly near its projected center and fades smoothly towards the boundary of its projected ellipse. To compute the final color $\mathbf{C}(\mathbf{u})$, the renderer aggregates the colors $\mathbf{c}_i$ of the $\mathcal{N}$ sorted Gaussians overlapping the pixel. Using the standard front-to-back formulation, the color is accumulated as:

$$\mathbf{C}(\mathbf{u}) = \sum_{i \in \mathcal{N}(\mathbf{u})} T_i(\mathbf{u}) \tilde{\alpha}_i(\mathbf{u}) \mathbf{c}_i \quad , \tag{2.13}$$

where $T_i(\mathbf{u})$ represents the accumulated transmittance, i.e., the probability that light reaches the camera without being absorbed before reaching Gaussians i:

$$T_i(\mathbf{u}) = \prod_{j<i} (1 - \tilde{\alpha}_j(\mathbf{u})) \quad . \tag{2.14}$$

If the accumulated transmittance drops below a small threshold (i.e., the pixel becomes opaque), the renderer can stop processing further Gaussians for that pixel. Finally, to handle areas with low density, a background color $\mathbf{C}_{\text{bg}}$ is typically blended using the remaining transmittance:

$$\mathbf{C}(\mathbf{u}) = \sum_{i \in \mathcal{N}(\mathbf{u})} T_i(\mathbf{u}) \tilde{\alpha}_i(\mathbf{u}) \mathbf{c}_i + T_{\text{final}}(\mathbf{u}) \mathbf{C}_{\text{bg}} \quad . \tag{2.15}$$

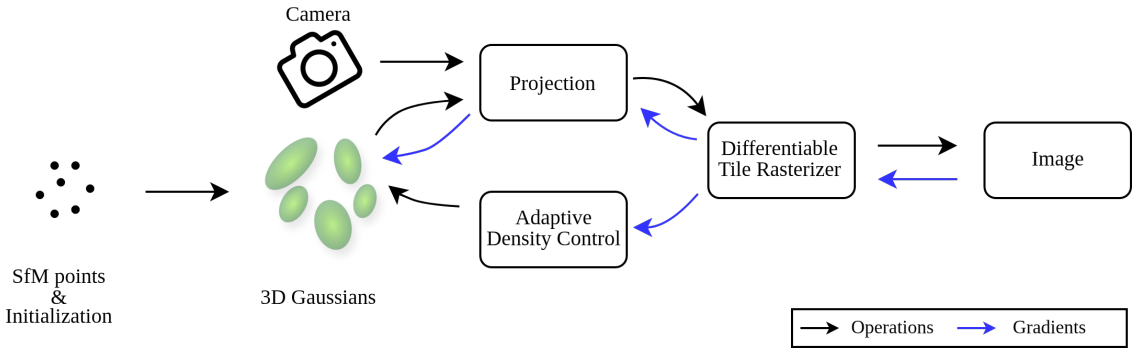


Figure 2.10: Overview of 3DGS [7].

Accurate rendering requires an optimization stage, where the Gaussian parameters are iteratively updated by rendering the scene from the training camera viewpoints and comparing the rendered images against their corresponding ground-truth. In 3DGS, Stochastic Gradient Descent is used for optimization, where the loss function to minimize combines a photometric $\mathcal{L}_1$ term with a structural dissimilarity term:

$$\mathcal{L} = (1-\lambda)\mathcal{L}_1 + \lambda \mathcal{L}_{D-SSIM} \quad . \tag{2.16}$$

The $\mathcal{L}_1$ term measures the average absolute pixel difference between the rendered image $\hat{u}$ and the ground-truth image u:

$$\mathcal{L}_1 = \frac{1}{n}\sum_{i=1}^{n}|u_i - \hat{u}_i| \quad . \tag{2.17}$$

The D-SSIM term is derived from the Structural Similarity Index Measure (SSIM), which compares two images using luminance, contrast, and structural similarity:

$$SSIM(x,y) = l(x,y)^{\alpha} \cdot c(x,y)^{\beta} \cdot s(x,y)^{\gamma} \quad , \tag{2.18}$$

on where each of these terms corresponds to:

$$\begin{aligned} l(x,y) &= \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1} \\ c(x,y) &= \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2} \\ s(x,y) &= \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3} \quad . \end{aligned} \tag{2.19}$$

Where the variables are computed from the image as follows:

- μ_x and μ_y are the pixel sample means.
- σ_x^2 and σ_y^2 are the pixel variances which quantify the contrast.
- σ_{xy} is the covariance between x and y, evaluating structural similarity.
- c_1, c_2, c_3 are stability constants introduced to avoid numerical instability when the denominators are close to zero.

Commonly, these constants are defined using the dynamic range of the pixel values (L) as:

$$c_1 = (K_1 L)^2, \quad c_2 = (K_2 L)^2, \quad c_3 = \frac{c_2}{2} \tag{2.20}$$

where the standard literature values are $K_1 = 0.01$ and $K_2 = 0.03$. The dissimilarity loss is then commonly written as:

$$\mathcal{L}_{D\text{-}SSIM} = \frac{1 - SSIM}{2} \quad . \tag{2.21}$$

On Figure 2.10 we can see an overview of the complete 3DGS method. Appart from what it has already been covered, it includes another key contribution, the Adaptive Densiity Control. During optimization, the number and placement of Gaussians are adjusted depending on the reconstruction quality and the local geometry. Specifically, the system monitors the view-space positional gradients of the Gaussians. If a region has high reconstructed error and significant gradient magnitude, the system can either split a large Gaussian into two smaller ones or clone a small one to fill in missing details. Conversely, Gaussians with low opacity are periodically pruned to maintain computational efficiency. An illustration of this technique is shown in Figure 2.11.

Another important step shown in Figure 2.10 is the initialization of the Gaussian primitives. In the original 3DGS pipeline, this initialization is obtained from Structure from Motion (SfM). SfM is a feature-based technique that extracts geometric

features from a sequence of images, tracking these points to reconstruct a 3D sparse geometry. In 3DGS, these points serve as ideal candidates for initializing the positions of the 3D Gaussians. However, in this thesis the explanation of SfM is secundary, as the specific method used for generating the Gaussian splats in the autonomous driving scenes employs a different initialization strategy, which will be explained in the following section.

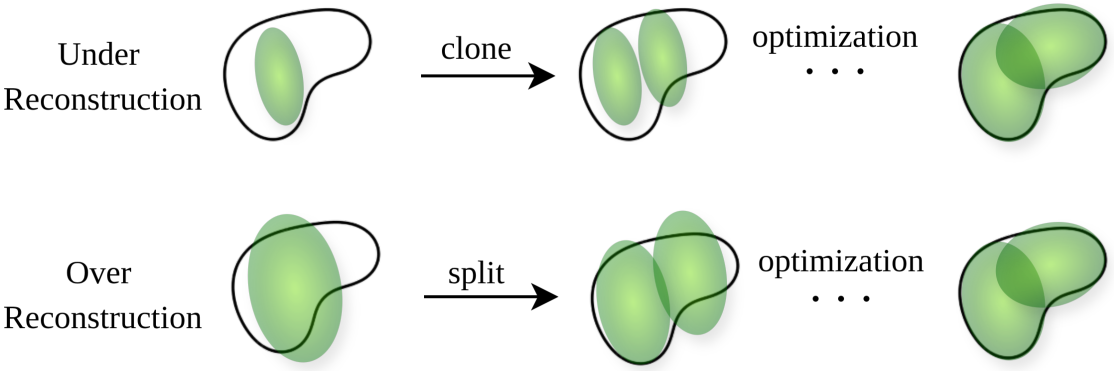


Figure 2.11: Adaptive Gaussian densification technique from 3DGS. When a small-scale geometry is insufficiently covered, the respective Gaussian is cloned. Contrary, if a small-scale geometry is represented by one large splat, it is split into two.

2.4.2 SplatAD

The original 3DGS was primarly designed for achieving efficient and photorealistic rendering of static scenarios. When it comes to reconstructing autonomous driving sequences, different requirements appear due to the larger scale nature of the driving scenes, as well as the presence of dynamic actors like other vehicles or pedestrians. SplatAD [8] addresses these by extending the 3DGS framework into a multi-modal, actor-aware system.

A central contribution of SplatAD is its ability to render both camera images and LiDAR point clouds within a unified 3D Gaussian representation and pipeline. This is important for autonomous driving since LiDAR data is commonly available in driving datasets and provides directly valuable 3D geometric measurements. In fact, SplatAD does not only output LiDAR points as part of the rendering, but utilizes its geometric information for optimization. For example, by supervising the depth accumulated during the α-blending process with respect to the corresponding LiDAR measurement, it can ensure that the Gaussians are placed at the correct physical distances, significantly reducing the floater artifacts common in other purely image-based reconstructions.

Figure 2.12 depicts the rendering method used in splatAD. As we can see, for camera rendering, it preserves the core idea from the 3DGS pipeline. However, spatAD rasterizes learned feature vectors, which are then decoded using a small convolutional network, together with ray directions and sensor embeddings, to model view-dependent and sensor-specific appearance effects.

For LiDAR rendering, SpatAD uses a procedure adapted to the structure of the

LiDAR measurements. Since LiDAR data is represented by azimuth, elevation, and range, the Gaussians are projected into spherical LiDAR coordinates. Then, tiling accounts for the non-uniform vertical spacing of LiDAR beams, and these features, concatenated with ray direction are passed through a small MLP, which predicts LiDAR intensity and ray-drop probability, while the range is obtained through standard α-compositing. Therefore, SplatAD cannot only render the geometry of a LiDAR scan but also the reflective properties of surfaces and even noise patterns. Additionally, as the figure shows, SpatAD includes rolling-shutter compensation for both cameras and LiDAR. Which is important in driving since both the sensors and other actors may move during the acquisition of an image or LiDAR sweep. In this thesis, however, this component is not central to the implemented simulator, since explicit ground-truth velocity information was not available for the scenes.

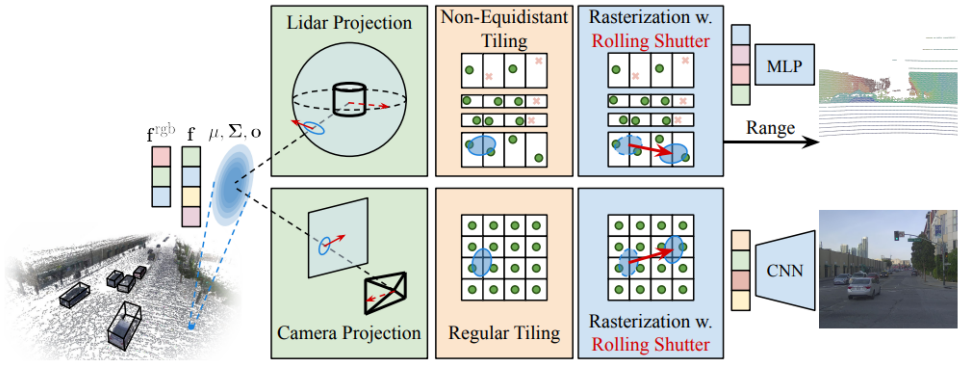


Figure 2.12: Overview of SplatAD rendering method. Figure taken from [8], © 2025 IEEE.

Another important adaptation for autonomous driving sequences is the treatment of dynamic elements. While standard 3DGS assumes a static scene, SplatAD decomposes the scene into a static background and a set of dynamic objects. Each of these dynamic actor is associated with a 3D bounding box and a sequence of poses over time. Gaussians belonging to the static background are represented in the world frame, while Gaussians belonging to an actor are represented in that actor's local coordinate frame. At rendering time, the actor pose is used to transform its Gaussians into the world frame before composing the full scene.
Furthermore, SplatAD also expands the optimization objective beyond the $\mathcal{L}_{depth}$ term explained above. In addition to the standard image reconstruction losses (composed of $\mathcal{L}_1$ and $\mathcal{L}_{SSIM}$), includes LiDAR-specific terms for range, intensity, line-of-sight consistency, and ray-drop prediction. The overall loss can be written as:

$$\begin{aligned}\mathcal{L} = \lambda_r \mathcal{L}_1 + (1-\lambda_r)\mathcal{L}_{SSIM} + \lambda_{\text{depth}}\mathcal{L}_{\text{depth}} + \lambda_{\text{los}}\mathcal{L}_{\text{los}} \\ + \lambda_{\text{intens}}\mathcal{L}_{\text{inten}} + \lambda_{\text{raydrop}}\mathcal{L}_{\text{BCE}} + \lambda_{\text{MCMC}}\mathcal{L}_{\text{MCMC}}.\end{aligned} \tag{2.22}$$

Here, the first two terms, as already mentioned, supervise the rendered camera images. While the remaining terms use LiDAR data and regularization to improve geometric consistency and sensor realism. $\mathcal{L}_{\text{depth}}$ and $\mathcal{L}_{\text{intem}}$ are L2 losses on the rendered expected lidar range and intensity. The line of sight loss $\mathcal{L}_{\text{los}}$ penalizes

opacity closer than the gorund truth LiDAR range. $\mathcal{L}_{\text{BCE}}$ is a binary cross-entropy loss on the predicted ray drop probability from the MLP. Finally, $\mathcal{L}_{\text{MCMC}}$ act as an opacity and scale regulization term.

2.4.3 Limitations of Neural-Rendered Simulation

Although methods like Gaussian Splatting reduce the visual gap associated with fully synthetic environments, they also introduce their own limitations. In particular, neural rendering depends strongly on the coverage provided by the original data collection. Meaning that these methods are performing view interpolation in order to compose new viewpoints, rather than a fully generative, physics-based process.
Therefore, the main challenge comes when rendering views far away from the original logged data. In such cases, previously hidden regions can become visible, object surfaces may be seen from unseen angles, and the geometry behind occluders may be poorly optimized. As a consequence, these regions may contain artifacts, holes, floaters, incorrect textures, or inconsistent geometry.
Driving scenes containing moving actors, amplify this challenge, since it is quite common that from logging, a dynamic actor has only been observed for a short time and from a limited set of perspectives. Thus, during simulation, the agent can end up encountering a view of the dynamic actor with no information to reconstruct. Causing visual holes or black voids, as no Gaussians were ever initialized or optimized in those unobserved volumes.
Additionally, scenes containing rain, snow, or other adverse weather conditions can introduce further limitations. These conditions produce transient and view-dependent effects, such as water droplets, spray, or reduced visibility. As a result, such scenes may lead to degraded reconstruction quality, unstable appearance, or noticeable artifacts.
Finally, it is important to consider as well our selected method, SplatAD, which is currently limited to modeling all dynamic actors as rigid. Meaning that actors like pedestrians might not be perfectly modelled.
For the reader to have a nicer visual comprehension of what artifacts and limitations from neural rendering can look like, Figure 2.13 illustrates an example.

Figure 2.13: Comparison between a ground-truth image and the corresponding 3DGS reconstruction. Left: ground-truth image. Right: reconstruction showing grainy artifacts in the grass region.

2.5 Vehicle Motion

As mentioned on Section 2.2, and illustrated on Figure 2.6, when operating in closed-loop, the learned policy is actually rolled. This requires a mechanism for propagating the ego vehicle state, which usually consists on a vehicle model combined with a tracking controller that allows to follow the planned trajectory and thus, update the ego pose.

2.5.1 Coordinate Frames and Ego Pose

Let us begin with a section dedicated to reviewing geometry and the relationship between various coordinate systems. In autonomous driving, the motion of the ego vehicle and the sensors mounted on is describe using the following coordinate frames:

- **World Frame:** It is a fixed global frame for the environment. The scene and all actors are repesented in this coordinate system, including all the static 3D Gaussians.
- **Ego Frame:** A local frame that attached to the ego vehicle itself, usually with its origin either at the center of the rear axis or the vehicle's center of gravity (COG).
- **Sensor Frame:** A frame attached to a specific sensor, like a camera or a LiDAR. They are related to the ego frame through fixed calibration parameters.

Within this mentioned frames, the pose of the vehicle in the world can be represented as a rigid-body transformation. And since it is defined in $\mathbb{R}^3$, this transformation belongs to the special Euclidean group $SE(3)$, and can be written as:

$$\mathbf{T}_{\text{ego}\to\text{world}} = \begin{bmatrix} \mathbf{R}_{\text{ego}\to\text{world}} & \mathbf{t}_{\text{ego}\to\text{world}} \\ \mathbf{0}^T & 1 \end{bmatrix} , \tag{2.23}$$

where $\mathbf{R}_{\text{ego}\to\text{world}}$ is a rotation matrix and $\mathbf{t}_{\text{ego}\to\text{world}}$ is the translation vector. A point expressed in the ego frame can then be transformed into the world frame as:

$$\begin{bmatrix} \mathbf{p}_{\text{world}} \\ 1 \end{bmatrix} = \mathbf{T}_{\text{ego}\to\text{world}} \begin{bmatrix} \mathbf{p}_{\text{ego}} \\ 1 \end{bmatrix} \quad . \tag{2.24}$$

And the inverse transformation from the world frame back into the ego frame can be achive by just applying instead:

$$\mathbf{T}_{\text{world}\to\text{ego}} = \mathbf{T}^{-1}_{\text{ego}\to\text{world}}. \tag{2.25}$$

However, it is worth mentioning that for most driving tasks, the vehicle's motion can be simplified to a planar $SE(2)$ representation. The state is then defined by the triplet (x, y, ψ), where (x, y) is the position on the ground plane and ψ is the heading angle (yaw). And the corresponding planar homogeneous transformation is

$$\mathbf{T}^{2D}_{\text{ego}\to\text{world}} = \begin{bmatrix} \cos\psi_t & -\sin\psi_t & x_t \\ \sin\psi_t & \cos\psi_t & y_t \\ 0 & 0 & 1 \end{bmatrix} \quad . \tag{2.26}$$

For a better understanding of this work, it is useful to also introduce the pinhole camera model, illustrated in Figure 2.14. In this model, the extrinsic parameters describe the pose of the camera with respect to the scene, while the intrinsic parameters describe how points expressed in the camera frame are projected onto the image plane.

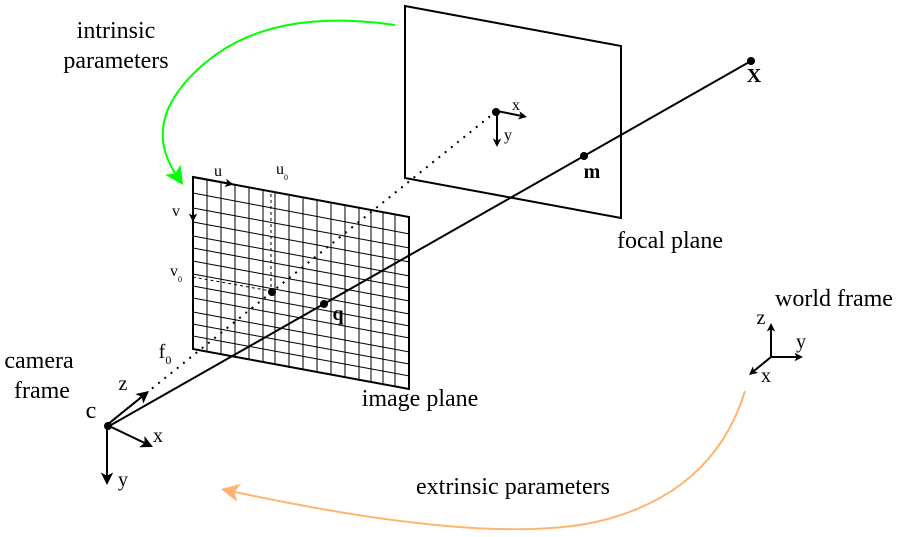


Figure 2.14: The pinhole camera model.

A 3D point expressed in the world frame, $\mathbf{P}_{\mathcal{W}} = [X_{\mathcal{W}}, Y_{\mathcal{W}}, Z_{\mathcal{W}}]^T$, can be first transformed into the camera coordinate frame $\mathcal{C}$ using the camera extrinsics:

$$\mathbf{P}_{\mathcal{C}} = \mathbf{R}_{\mathcal{CW}}\mathbf{P}_{\mathcal{W}} + \mathbf{t}_{\mathcal{CW}} \quad , \tag{2.27}$$

where $\mathbf{R}_{\mathcal{CW}}$ and $\mathbf{t}_{\mathcal{CW}}$ define the rotation and translation from the world frame to the camera frame. Usually, this is found written in homogeneous coordinates as:

$$\mathbf{P}_{\mathcal{C}} = \mathbf{T}_{\mathcal{CW}}\mathbf{P}_{\mathcal{W}} = \begin{bmatrix} \mathbf{R}_{\mathcal{CW}} & \mathbf{t}_{\mathcal{CW}} \\ \mathbf{0}^T & 1 \end{bmatrix} \mathbf{P}_{\mathcal{W}} \quad , \tag{2.28}$$

Once the point is expressed in the camera frame, $\mathbf{P}_{\mathcal{C}} = [X_{\mathcal{C}}, Y_{\mathcal{C}}, Z_{\mathcal{C}}]^T$, it can be projected onto the image plane using the intrinsic matrix $\mathbf{K}$:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} X_{\mathcal{C}} \\ Y_{\mathcal{C}} \\ Z_{\mathcal{C}} \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \ , \tag{2.29}$$

where λ is a scale factor corresponding to the depth of the point $Z_{\mathcal{C}}$. f_x and f_y are the focal lengths in pixel units, and c_x, c_y define the principal point. Finally getting the pixel coordinates:

$$u = f_x \frac{X_{\mathcal{C}}}{Z_{\mathcal{C}}} + c_x, \qquad v = f_y \frac{Y_{\mathcal{C}}}{Z_{\mathcal{C}}} + c_y \ . \tag{2.30}$$

2.5.2 Vehicle Model

There are multiple ways in the literature on how to model a car. From simplistic point-mass models to very complex four-wheel models with tire dynamics. In the case of this work, a kinematic bicycle model is employed to simulate the motion. This model simplifies the four-wheeled vehicle into two wheels (front and rear) and assumes that the vehicle's motion is governed by geometry rather than tire forces, which is a valid assumption for the speeds and low accelerations typically encountered in urban driving. Figure 2.15 shows an illustration of this model.
Within this model, the state of the vehicle is represented as:

$$\mathbf{x}_t = \begin{bmatrix} x_t & y_t & \psi_t & v_t \end{bmatrix}^T, \tag{2.31}$$

where x_t and y_t denote the position of the vehicle reference point (in this case the rear axle) in the world frame, ψ_t is the yaw angle, and v_t is the longitudinal velocity. And its continuous-time dynamics are:

$$\dot{x} = v \cos\psi \ , \tag{2.32}$$

$$\dot{y} = v \sin\psi \ , \tag{2.33}$$

$$\dot{\psi} = \frac{v}{R} = \frac{v}{L} \tan\delta \ . \tag{2.34}$$

where L is the wheelbase, δ is the steering angle of the front wheel, and R is the turning radius.
Since tire slip is not modeled, the front and rear wheels will follow circular arcs around an instantaneous center of rotation (ICR). Thus, the steering angle defines the curvature of the vehicle path through:

$$R = \frac{L}{\tan\delta}. \tag{2.35}$$

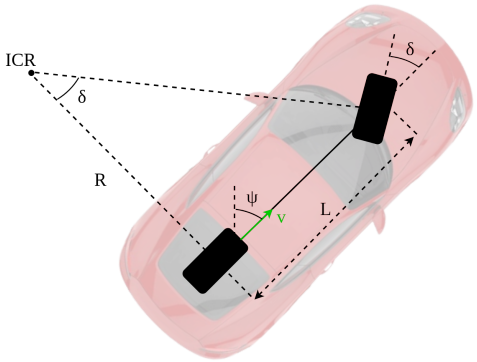


Figure 2.15: Kinematic bicycle model, in which the vehicle is approximated by a front and rear wheel separated by the wheelbase L. The vehicle moves with longitudinal velocity v, heading ψ, and front-wheel steering angle δ. Under the no-slip assumption, the vehicle follows a circular arc around the instantaneous center of rotation (ICR) with turning radius R.

2.5.3 Trajectory Tracking

For the described vehicle model to follow a desired path, it is first required to compute its control inputs, namely steering and longitudinal acceleration or velocity. In this work, the control problem is divided into lateral and longitudinal components in a decoupled fashion.

2.5.3.1 Lateral Control

A common lateral control method is pure pursuit, illustrated in Figure 2.16. The idea is to select a look-ahead point on the desired trajectory and compute a steering command that makes the vehicle follow a circular arc toward that point.

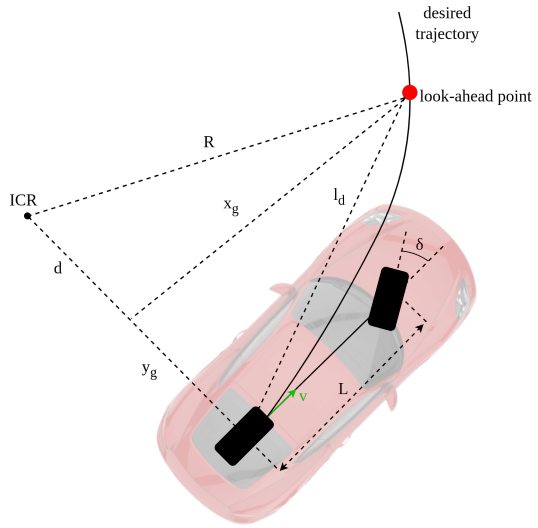


Figure 2.16: Pure pursuit controller. A look-ahead point is selected on the desired trajectory, and the steering command is computed such that the vehicle follows a circular arc toward that point.

Following the convention in Figure 2.16, let the selected look-ahead point be expressed in the ego frame as (x_g, y_g). The look-ahead distance is then:

$$l_d^2 = x_g^2 + y_g^2 \ . \tag{2.36}$$

The radius R of the circular arc that reaches the look-ahead point can be geometrically defined as:

$$R = y_g + d \ , \tag{2.37}$$

and,

$$d^2 + x_g^2 = R^2 \ . \tag{2.38}$$

Substituting $d = R - y_g$ gives

$$(R - y_g)^2 + x_g^2 = R^2 \ . \tag{2.39}$$

Expanding and simplifying,

$$R^2 - 2Ry_g + y_g^2 + x_g^2 = R^2 \ , \tag{2.40}$$

$$2Ry_g = x_g^2 + y_g^2 = l_d^2 \ , \tag{2.41}$$

and therefore:

$$R = \frac{l_d^2}{2y_g} \ . \tag{2.42}$$

The curvature of the circular arc is the inverse of the radius,

$$\kappa = \frac{1}{R} = \frac{2y_g}{l_d^2}. \tag{2.43}$$

Using equation 2.35 from the kinematic bicycle model, the curvature can be related to the steering angle by:

$$\kappa = \frac{1}{R} = \frac{\tan\delta}{L} \ . \tag{2.44}$$

And therefore, solving for the steering angle gives the control law:

$$\delta = \arctan\left(\frac{2Ly_g}{l_d^2}\right) \ . \tag{2.45}$$

2.5.3.2 Longitudinal Control

Longitudinally, it is common to track either velocity or acceleration. This is often handled with a proportional-integral-derivative (PID) controller, which aims to minimize the error between a desired setpoint and the current vehicle state.
When tracking velocity, the error can be computed as

$$e_v(t) = v_{\text{des}}(t) - v(t) \ , \tag{2.46}$$

where $v_{\text{des}}(t)$ is the desired velocity and $v(t)$ is the current longitudinal velocity.
This tracking error can be converted into a longitudinal acceleration command using the standard PID control law:

$$a(t) = K_p e_v(t) + K_i \int e_v(t)\,dt + K_d \frac{de_v(t)}{dt} \ , \tag{2.47}$$

where K_p, K_i, and K_d are controller gains. The proportional term reacts to the current velocity error, the integral term compensates for accumulated steady-state error, and the derivative term damps rapid changes in the error.
Given this acceleration command, the vehicle velocity can be updated in discrete time as

$$v_{t+1} = v_t + a_t \Delta t. \tag{2.48}$$

2.6 Planner Training

This section will depict the underlying theory behind the different training strategies that have been tested during this thesis.

2.6.1 Closed-Loop Imitation Learning

While classical imitation learning suffers from the problems discussed in Section 2.2.2, closed-loop imitation learning (CL-IL) tries to mitigate these by unrolled supervised training. On where scene rollouts are actively executed during the optimization loop, integrating the closed-loop feedback and consequences into the training process.

During this procedure, the planner will predict a future ego trajectory, based on the current observations and state:

$$\hat{\tau}_t = \pi_\theta(o_t, s_t) \ . \tag{2.49}$$

After that, the trajectory is converted into control inputs by means of the tracking controllers explained in Section 2.5.3:

$$u_t = \mathcal{C}(\hat{\tau}_t) \ , \tag{2.50}$$

and the vehicle state is propagated by the vehicle model,

$$s_{t+\Delta t} = f(s_t, u_t) \ . \tag{2.51}$$

Notice that this closes the loop, since it makes future observations depend on the planner's previous predictions.
However, the supervised target is still obtained from the expert log. Where, since the ego vehicle may no longer be exactly on the logged trajectory, the expert future trajectory must be expressed relative to the current ego pose. Denoting this target as τ_t^*, the closed-loop imitation-learning objective can be written as:

$$\mathcal{L}_{\text{CL-IL}} = \sum_{t \in \mathcal{T}_{\text{rollout}}} \ell(\hat{\tau}_t, \tau_t^*) \ . \tag{2.52}$$

On where a common choice is an L_2 loss over the predicted future poses:

$$\mathcal{L}_{\text{CL-IL}} = \sum_{t \in \mathcal{T}_{\text{rollout}}} \sum_{k=1}^{H} \left\| \hat{\mathbf{p}}_{t+k} - \mathbf{p}_{t+k}^* \right\|^2 \ , \tag{2.53}$$

where H is the prediction horizon and $\mathbf{p}_{t+k} = (x_{t+k}, y_{t+k})$ is the predicted position at future step k. If heading is also supervised, an additional angular error term can be included:

$$\mathcal{L}_{\text{CL-IL}} = \sum_{t} \sum_{k=1}^{H} \left\| \hat{\mathbf{p}}_{t+k} - \mathbf{p}_{t+k}^* \right\|^2 + \lambda_\psi \left(g_\psi(\hat{\psi}_{t+k} - \psi_{t+k}^*) \right)^2 \ . \tag{2.54}$$

where g_ψ is just an angular distance function to take into account the periodic nature of angles.
The key difference with open-loop imitation learning is that the observations o_t are no longer sampled only from the expert state distribution d^{π^*}. Instead, they are sampled from the distribution induced by the current planner, d^{π_θ}, which aligns with the testing data distribution.
Additionally, if the model commits a minor directional error at step t, the state s_{t+1} will naturally drift away from the expert trajectory. At step $t+1$, the network is fed an observation o_{t+1} that contains this error. Because the loss function continues to penalize the final distance to the expert's ground truth, the optimization gradient will actively force the network to learn recovery behaviors—specifically, such as corrective commands that steer the vehicle back toward the expert path.

2.6.2 Reinforcement Learning

Another possible training strategy is to apply reinforcement learning (RL). Within this framework, instead of matching an expert supervisor, an agent learns to optimize its behavior through active interaction with the environment.
This interaction process is commonly formulated as a Markov Decision Process (MDP), defined by $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- **State Space ($\mathcal{S}$):** The state $s_t \in \mathcal{S}$ represents the environmental and ego conditions at time t.
- **Action Space ($\mathcal{A}$):** A continuous space where the agent selects real-valued actions $a_t \in \mathcal{A}$ at each time step.
- **Transition Dynamics ($\mathcal{P}$):** A transition probability $\mathcal{P}(s_{t+1} \mid s_t, a_t)$ defining the environment's response to an action.
- **Reward Function ($\mathcal{R}$):** The reward $r_t = \mathcal{R}(s_t, a_t)$ is a scalar feedback signal provided at each step by the environment.
- **Discount Factor (γ):** $\gamma \in [0, 1)$ determines the agent's horizon preference, weighting immediate rewards against future returns.

The goal of the agent is to learn an optimal policy $\pi_\theta(a_t \mid s_t)$, that maximizes the expected discounted return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \quad . \tag{2.55}$$

In the context of autonomous driving, the reward can encode objectives such as staying close to a route, producing comfortable motion, or avoiding collisions. Compared to imitation learning, the planner can, in principle, learn any behavior that maximizes the task reward without being constrained to reproduce the expert trajectory exactly.
However, this can also make the training more difficult, since the quality of the learned behavior depends strongly on the design of the reward function and on the stability of the optimization process.
Another consideration is that directly optimizing $J(\theta)$ is challenging because the policy influences not only the selected actions, but also the future states that will be visited. Policy-gradient methods address this by estimating how changes in the probability of selected actions affect the expected return. Let:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k \tag{2.56}$$

denote the discounted return from time t. A basic policy-gradient objective increases the likelihood of actions that lead to high returns and decreases the likelihood of actions that lead to low returns:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) G_t\right] \quad . \tag{2.57}$$

However, using the return G_t directly can lead to high-variance gradients that easily destabilize training. A common solution is to introduce a value function $V_\phi(s_t)$, which estimates the expected return from a given state. The return can then be replaced by an advantage estimate,

$$A_t = G_t - V_\phi(s_t) \ , \tag{2.58}$$

which measures whether the selected action performed better or worse than expected from that state.
This leads to the actor-critic formulation. In this formulation, the actor is the policy $\pi_\theta(a_t \mid s_t)$, responsible for selecting actions, while the critic is a value function $V_\phi(s_t)$, responsible for estimating the expected return. The actor is updated to increase the probability of actions with positive advantage and decrease the probability of actions with negative advantage. While the critic is trained to predict the observed returns, usually through a loss of the form:

$$\mathcal{L}_V = \left(V_\phi(s_t) - G_t\right)^2 \ . \tag{2.59}$$

2.6.2.1 Proximal Policy Optimization

Even within an actor-critic setup, policy gradient steps can still be highly unstable. If a gradient update changes the policy parameters too drastically, the policy can degrade into a sub-optimal state from which it cannot recover. Methods such as Proximal Policy Optimization (PPO) [48] addresses this issue by limiting how much the new policy is allowed to deviate from the policy that generated the data.
At each training iteration, PPO measures the probability ratio $r_t(\theta)$ between the action probabilities under the new, changing policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$ that was used to collect the rollout:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \ . \tag{2.60}$$

To prevent the new policy from drifting too far from the old one, PPO maximizes the clipped surrogate objective:

$$\mathcal{L}_{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \ , \tag{2.61}$$

where $\epsilon \in (0, 1)$ is a hyperparameter that controls the maximum allowed policy change, and $\hat{A}_t$ is the estimated advantage function.
This results in a conservative update mechanism based on the sign of the advantage:

- **Positive advantage ($\hat{A}_t > 0$):** The selected action performed better than expected. The objective encourages increasing the probability of this action, but the improvement is clipped once $r_t(\theta) > 1 + \epsilon$.
- **Negative advantage ($\hat{A}_t < 0$):** The selected action performed worse than expected. The objective encourages decreasing the probability of this action, but the update is clipped once $r_t(\theta) < 1 - \epsilon$.

The full PPO loss is commonly written as a combination of this clipped policy objective, a value-function loss (equation 2.59), and an entropy bonus:

$$\mathcal{L}_{\text{PPO}} = -\mathcal{L}_{\text{CLIP}}(\theta) + c_v\mathcal{L}_V - c_e\mathcal{H}(\pi_\theta) \ , \tag{2.62}$$

where c_v and c_e are weighting coefficients, and $\mathcal{H}(\pi_\theta)$ is the entropy of the policy, which encourages exploration by discouraging the policy from becoming deterministic too early.

3
Methods

This chapter focuses on how the work has been structured and carried out. Starting with the data selection and posterior reconstruction. Continuing with how the simulator has been built, and what kind of planner has been employed. Then the training strategies and experiments will be depicted, as well as the evaluation procedure. Finally, some implementation details about how the work has been scaled up are presented.

3.1 Dataset and Scene Selection

The choice of dataset was strongly conditioned by its compatibility with SplatAD, as well as by the need for rich annotations and map information that could later be used by the simulator and training strategies.
With these requirements in mind, the Argoverse 2 Sensor Dataset [35] was selected. The dataset contains 1000 driving logs, each approximately 15 seconds long. The sensor suite includes 9 cameras and 2 stacked 32-beam LiDAR sensors. LiDAR sweeps are collected at 10 Hz, while images are captured at 20 FPS from 7 ring cameras and 2 stereo cameras. Ego-vehicle poses are provided in a global city coordinate system, and object annotations are given as 3D cuboids in the ego-vehicle reference frame. In addition, each scene includes an HD map composed of vector-map information and a ground-height raster map. An example frame from the dataset is shown in Figure 3.1.
For this work, the following information was extracted from each selected scene:

- Camera images and timestamps.
- Camera intrinsics and extrinsics.
- Ego poses over time.
- LiDAR sweeps.
- Dynamic actor annotations.
- HD map data, including the ground-height raster map.

As discussed in Section 2.4.3, scenes with rain, snow, very dark illumination, or low visibility can introduce artifacts and unstable appearance when rendered with SplatAD. For this reason, such sequences were filtered out during the selection of training scenes. In total, the following scene split was used:

- 641 scenes for training, all taken from the Argoverse 2 training split.
- 150 scenes for evaluation, corresponding to the full Argoverse 2 validation split.

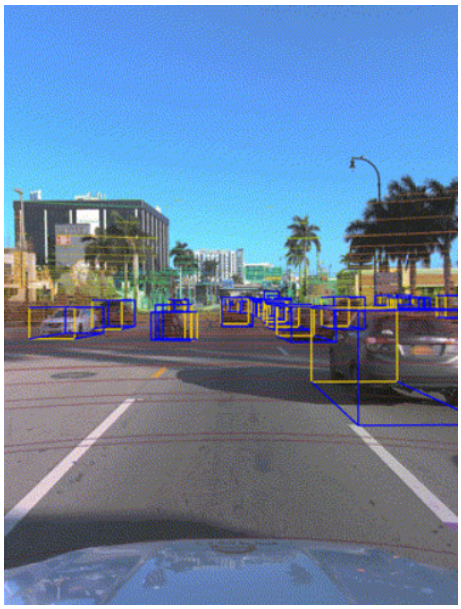


Figure 3.1: Example frame from the Argoverse Sensor dataset User Guide [9].

3.2 Scene Reconstruction

In this work, the public SplatAD implementation [49], built on top of Nerfstudio [50], was used as the basis for scene reconstruction.
The purpose of this stage was to convert each selected Argoverse scene into a self-contained representation that could later be loaded by the closed-loop simulator. For this reason, the reconstruction pipeline had to produce not only a trained Gaussian representation, but also the additional metadata required for simulation.
The original SplatAD repository already included an Argoverse 2 dataparser. This dataparser was extended to load and store the static map associated with each scene through the Argoverse API. The map contains both vector-map information and the rasterized ground-height layer.
An important practical detail is that the internal coordinate system used by SplatAD does not directly match the original Argoverse coordinate system used for ego poses, sensor poses, and HD map elements. Therefore, the dataparser transform was saved for each reconstructed scene, so that poses and map elements could later be converted consistently between both coordinate systems.
Finally, the camera calibration and trajectory information were exported into a separate file. This file contains the intrinsic parameters, image dimensions, camera-to-world poses, timestamps, and sensor identifiers for the training cameras. These quantities are required by the simulator to initialize the ego pose, recover the logged trajectory, select the relevant camera sensors, and render observations from the planner viewpoints.
Overall, each reconstructed scene folder contains the following elements:

- A trained SplatAD checkpoint containing the optimized Gaussian representation, the dynamic actor representations, the appearance embeddings, and the learned image decoder.

- A `dataparser_transforms.json` file, used to relate the Argoverse and reconstruction coordinate systems.
- A copy of the `hd_map` directory, containing map and ground-height information.
- The `train_cameras.pt` file, containing camera intrinsics, camera poses, timestamps, image dimensions, and sensor identifiers.

The reconstruction process was carried out independently for each scene, using one NVIDIA A100 GPU. A typical scene required approximately two to two and a half hours. Figure 3.2 shows an example of the reconstruction achieved with this process.

Figure 3.2: Example of SplatAD reconstruction. The left image shows the original frame from one of the Argoverse 2 cameras, while the right image shows the SplatAD rendering from the same camera, pose, and timestamp.

3.3 Planner Choice

The aim was to find a sensor-level end-to-end planner in order to act as the baseline for this work. Apart from this initial imposition, there were other requirements taken into consideration for selection. First hard constraint was the availability of pretrained checkpoints. Additionally, the compatibility with NavSim [25] (see Section 3.6) was something highly desired to make the evaluation procedure smooth. Finally, sensor modality was also taken into account, since we disposed of camera rendering from different ego positions, as well as the possibility to render LiDAR if required as well. Taking all of those into account, the final decision was to use a Latent TransFuser [33] architecture as the basis of this work. Apart from the above reasons, this planner is widely known and is commonly referred to as a baseline when comparing new planner performances against.
Moreover, using the latent version of Transfuser eased the scene reconstruction process significantly since rendered LiDAR point clouds were not required.
In this work is essentially used as a trajectory generation policy. At each simulation step, it receives a camera observation and an ego-status vector to predict a future ego trajectory. In particular, the camera input is constructed from the front camera triplet, front-left, front-center, and front-right. Following the expected input format, the three rendered camera images are preprocessed into a single stitched image. The side cameras, which have a different resolution compared to the center one, are first resized, and then all three images are cropped vertically, the side views are

horizontally cropped to keep the regions closest to the forward-facing camera, and the resulting images are stitched horizontally. Finally, the stitched image is resized to 1024×256 and converted into a tensor, producing the input feature:

$$I_t \in \mathbb{R}^{3\times256\times1024} \ . \tag{3.1}$$

Additionally, the ego-status feature is composed of the high-level driving command, the current ego velocity, and the current ego acceleration. The driving command is represented as a one-hot vector with four possible values, corresponding to left, straight, right, and unknown. Since both velocity and acceleration are represented by two-dimensional vectors, the complete status feature is:

$$z_t = [c_t,\, v_t,\, a_t] \in \mathbb{R}^8 \ , \tag{3.2}$$

where $c_t \in \mathbb{R}^4$ is the driving command, $v_t \in \mathbb{R}^2$ is the ego velocity, and $a_t \in \mathbb{R}^2$ is the ego acceleration.
Thus, the whole planner input can be summarized as:

$$s_t = (I_t, z_t)\,. \tag{3.3}$$

The planner, after receiving s_t, will output a trajectory composed of future ego poses,

$$\hat{\tau}_t = \left[(\hat{x}_{t+1}, \hat{y}_{t+1}, \hat{\psi}_{t+1}), \ldots, (\hat{x}_{t+H}, \hat{y}_{t+H}, \hat{\psi}_{t+H})\right] \ , \tag{3.4}$$

where each pose is expressed in the current ego frame, at a frequency of 2Hz, or, what is the same, half a second between each pose. And the prediction horizon consists of $H = 8$ future poses, or a total of 4-second prediction.

3.3.1 Drive Command Generation

As mentioned before, the Latent TransFuser expects a high-level driving command as part of its status input. Without any external navigation module in the simulator, this command is generated from the logged ego trajectory. At each simulation step, the closest pose on the logged trajectory is found with respect to the current simulated ego position. A future point is then selected at a fixed time horizon, and the accumulated yaw change over this interval is used to classify the local route direction.
Considering ψ_i as the yaw angle of the closest logged pose and ψ_j the yaw angle at the future horizon point. The total heading change is computed as

$$\Delta\psi = \mathrm{wrap}(\psi_j - \psi_i) \ . \tag{3.5}$$

Then $\Delta\psi$ is just heuristicly used to define the driving command. If it constantly positve and above a certain threshold it is classified as left. If it is consistently negative and below a threshold, the command is classified as right. And if the absolute yaw change is small, the command is classified as straight. Otherwise, the command is marked as unknown.

Particularly, in the implementation, a horizon of 4 s was used. Turns were detected using a yaw-change threshold of 0.25 rad, together with a consistency threshold requiring at least 70% of the intermediate yaw increments to have the same sign. Straight driving was detected when $|\Delta\psi| < 0.12$ rad.

3.4 Closed-Loop Sensor-Level Simulator

3.4.1 Simulator Overview

The closed-loop simulator has been designed to train and evaluate a planner under sensor-level feedback. Thus, at each simulation step, the simulator receives the current ego state, renders the camera observations, passes them to the planner, follows the predicted trajectory using the tracking controllers, and updates the ego pose with the kinematic bicycle model. This produces the next simulator state and closes the loop between the planner outputs and the future observations. Figure 3.3 shows a high-level description of a simulator rollout on which the planner is being trained.

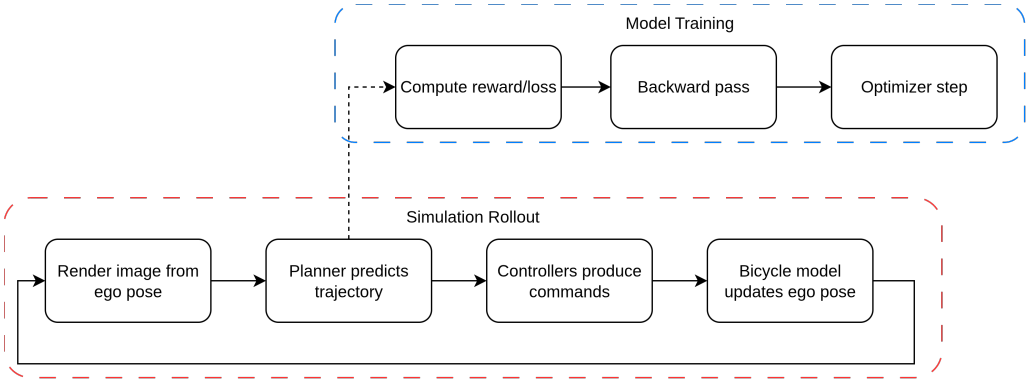


Figure 3.3: Simulator rollout overview while planner is under training.

3.4.2 Scene and Vehicle Initialization

Each simulated episode starts by loading one reconstructed scene folder. Which as mentioned in Section 3.2, should contain the trained SplatAD checkpoint, the exported camera file, the HD map, and the dataparser transform. The simulator then constructs two main objects, a scene object, responsible for rendering and doing map queries, and a vehicle object, responsible for the ego-state propagation.
The vehicle object loads the logged trajectory information for having available as ground truth. As well as the camera calibration for the front-left, front-center, and front-right cameras used by the planner. Then, the initial ego pose is recovered from the first front-camera pose by applying the inverse camera-to-ego transform, and the initial ego velocity is estimated from the first two poses of the logged trajectory.
On the other side, the scene object loads the HD map, the dataparser transform, and the SplatAD checkpoint.

3.4.3 Rendering from Planner Poses

The vehicle state propagated by the simulator is represented in planar form as (x, y, ψ). However, rendering camera observations requires a full ego-to-world transformation in 3D. Therefore, before each rendering call, the simulator lifts the planar ego state into an $SE(3)$ pose.
The vertical position is obtained from the HD map ground-height raster. Since the HD map is defined in the Argoverse city frame, while the reconstructed scene is represented in the SplatAD coordinate frame, the current ego position is first transformed back to the Argoverse frame. The ground height is then queried from the raster map and transformed back into the scene frame. By comparing the initial ego height above the road surface with the current ground height, the simulator gets back the relative sensor height with respect to the road.
Additionally, in order to avoid rendering from a completely flat pose on sloped roads, the local pitch is approximated from the difference between the ground height at the current position and the ground height at a point ahead of the vehicle. A simple suspension model is used as a low-pass filter on this pitch estimate, preventing abrupt changes in the rendered viewpoint. Roll is neglected in the final $SE(3)$ pose composition.
Once the full ego-to-world transform has been constructed, the pose of each camera is obtained by composing it with the corresponding camera-to-ego extrinsic calibration. The resulting camera-to-world pose, together with the camera intrinsics, sensor identifier, and current simulation timestamp, is passed to the SplatAD renderer. The timestamp is used to interpolate the dynamic actors to the correct time, while the intrinsics define the pinhole ray directions for the selected camera. The renderer then rasterizes the Gaussian scene into a feature image, combines it with the sensor-specific learned appearance embedding, and decodes it through the RGB decoder to produce the final camera image.

3.4.4 Rollout and Trajectory Following

At each simulation step, after rendering the current observations, the planner predicts a future trajectory in the current ego frame. This trajectory is transformed into control commands using the trajectory tracking strategy introduced in Section 2.5.2. In the implementation, the steering is obtained using the pure pursuit law from equation 2.45, on where the look-ahead target is selected as the fourth predicted pose in the trajectory sequence. And where the wheelbase corresponds to $L = 2.7\,\mathrm{m}$, matching the Ford Fusion Hybrid vehicles used in the Argoverse 2 data collection. An overview of this trajectory-following process is shown in Figure 3.4.

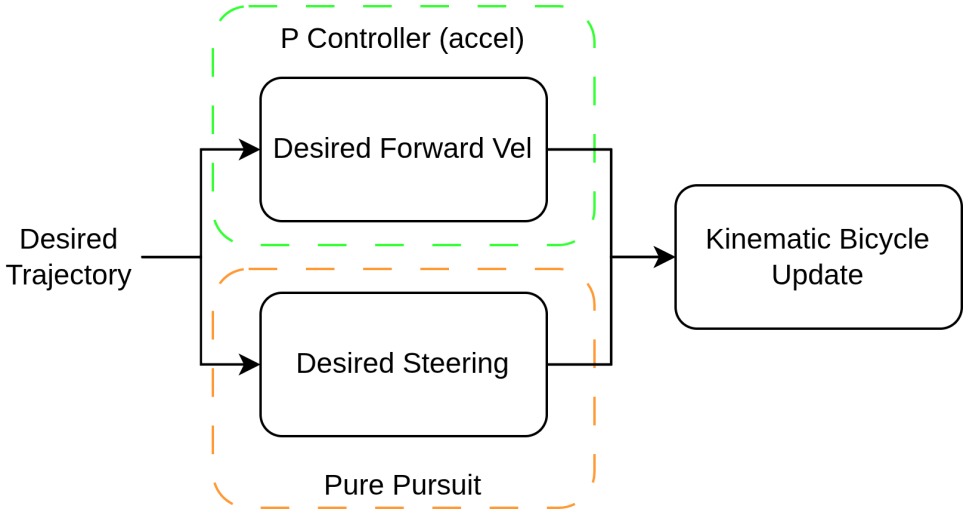


Figure 3.4: Simulated trajectory tracking process.

For the longitudinal control, first, a desired velocity is estimated from the spacing between the first predicted trajectory points. Describing Δt_τ as the time interval between predicted trajectory poses, and using the first k trajectory segments, the desired speed is computed as:

$$v_{\text{des}} = \frac{1}{k} \sum_{i=0}^{k} \frac{\|\hat{\mathbf{p}}_{t+i} - \hat{\mathbf{p}}_{t+i-1}\|}{\Delta t_\tau} \ , \tag{3.6}$$

where $\hat{\mathbf{p}}_{t+i} = [\hat{x}_{t+i}, \hat{y}_{t+i}]^T$. In the implementation, $k = 2$ and $\Delta t_\tau = 0.5\,\text{s}$. The velocity error term can be obtained by simply applying equation 2.46.
For the acceleration command, the PID law 2.47, has been simplified, to use only the proportional term:

$$a_t = K_p \left(v_{\text{des}} - v_t\right), \tag{3.7}$$

which is usually enough for an accurate control for simple dynamics. In the actual implementation the used proportional gain was simply $K_p = 1.0$. Then, to keep the simulated motion within a reasonable range, the acceleration command is clipped according to:

$$a_t \in [-4.0,\ 2.5]\,\text{m/s}^2. \tag{3.8}$$

The command is then smoothed using a first-order low-pass filter,

$$\tilde{a}_t = \alpha a_{t-1} + (1 - \alpha) a_t \ , \tag{3.9}$$

where $\alpha = 0.8$.
Finally, the forward velocity used for the vehicle rollout is simply updated as:

$$v_{t+\Delta t} = v_t + \tilde{a}_t \Delta t \ . \tag{3.10}$$

Both the steering and updated forward velocity are passed to the kinematic bicycle model, which uses Euler integration, to update the planar ego pose as:

$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\cos(\psi_t)\Delta t, \tag{3.11}$$

$$y_{t+\Delta t} = y_t + v_{t+\Delta t}\sin(\psi_t)\Delta t, \tag{3.12}$$

$$\psi_{t+\Delta t} = \psi_t + \frac{v_{t+\Delta t}}{L}\tan(\delta_t)\Delta t \ . \tag{3.13}$$

Once with the new pose, the simulator time is advanced,

$$t \leftarrow t + \Delta t \ , \tag{3.14}$$

and the rollout continues until an episode termination condition is triggered. These termination conditions depend on the training procedure being carried, and will be explained in Section 3.5.

3.4.5 Collision and Off-Road Checks

The simulator can also produce checks that are later used during training or evaluation. Off-road detection is performed by querying the drivable-area layer of the HD map at the current ego position. Once again, the ego position has to be transformed from the SplatAD frame back into the Argoverse city frame before the query.
On the other hand, collision detection is performed using the positions of the dynamic actors stored in the SplatAD scene. At the current simulation timestamp, actor poses are interpolated and converted into 2D bounding boxes in the world frame.
Collision detection is performed using the dynamic actors stored in the SplatAD scene. At the current simulation timestamp, actor poses are interpolated and converted into 2D bounding boxes in the scene frame, which can be essentially thought of as BEV boxes.
For an actor with length l, width w, and box-to-world transform $\mathbf{T}_a$, the local box corners can be written as

$$\mathbf{q}_j = \begin{bmatrix} \pm l/2 \\ \pm w/2 \\ 0 \\ 1 \end{bmatrix}, \qquad j = 0, \ldots, 3 \ . \tag{3.15}$$

The corresponding world-frame corners are just:

$$\mathbf{p}_j = \mathbf{T}_a \mathbf{q}_j, \tag{3.16}$$

and the 2D polygon used for collision checking is obtained from the x and y components of $\mathbf{p}_j$. The ego vehicle is represented in the same way, using its current ego-to-world transform and dimensions matching the Ford Fusion Hybrid vehicles, with $w = 1.8\,\mathrm{m}$ and $l = 4.8\,\mathrm{m}$.
To check whether two oriented boxes overlap, the simulator uses the Separating Axis Theorem. For every edge of both polygons, a perpendicular axis is computed, and

both boxes are projected onto that axis. In our case, with rectangular boxes, only four unique axes need to be tested per pair of boxes.
For a polygon with corners $\mathbf{p}_j$, the projection interval along an axis $\mathbf{n}$ is:

$$\left[\min_j \mathbf{n}^T \mathbf{p}_j,\ \max_j \mathbf{n}^T \mathbf{p}_j\right] \quad . \tag{3.17}$$

If the projection intervals do not overlap for at least one tested axis, i.e.,

$$\max_j \mathbf{n}^T \mathbf{p}_j^{(0)} < \min_j \mathbf{n}^T \mathbf{p}_j^{(1)} \quad \text{or} \quad \max_j \mathbf{n}^T \mathbf{p}_j^{(1)} < \min_j \mathbf{n}^T \mathbf{p}_j^{(0)} \quad , \tag{3.18}$$

the boxes are separated, and no collision is detected. If the intervals overlap for all tested axes, the ego box and actor box are considered to collide. It was consider that performing this check in BEV was sufficient for detecting planar vehicle overlaps in the simulated driving scenes. Figure 3.5 shows a visual example of a non-overlapping projection interval.

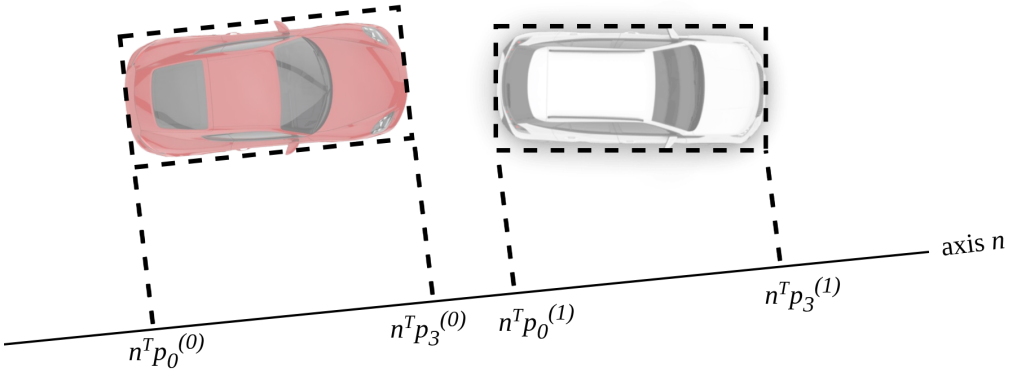


Figure 3.5: Example of a separating axis, on which the projected intervals do not overlap, indicating that the bounding boxes are not colliding.

3.5 Training Strategies

This section will cover the different training strategies and experiments carried out in this work. As discussed in Section 1.5, the goal was not to train an end-to-end planner from scratch, but to adapt an open-loop pretrained planner to overcome the challenges described in Section 2.2.2.
Therefore, all explored training strategies were formulated as fine-tuning procedures on top of the pretrained Latent TransFuser planner. Most of the model parameters were kept frozen, and depending on the experiment, only a small set of output heads was optimized.

3.5.1 PPO-Based Reinforcement Learning

One of the explored strategies was to adapt the planner using reinforcement learning with PPO. Since the theoretical foundation was explained in Section 2.6.2, this subsection will focus on how PPO was implemented in the simulator.

As mentioned before in this work, the base Latent TransFuser model predicts a deterministic trajectory. However, PPO requires a stochastic policy, since the update is based on the probability of the actions sampled during rollout. This is a nice characteristic since it allows the reward itself not to need to be differentiable with respect to the planner parameters.

3.5.1.1 Stochastic Residual Policy

To make the planner compatible with PPO, the deterministic Latent TransFuser was extended with two additional heads, a residual actor head and a value head. Both heads were implemented as two-layer MLPs applied to the trajectory query, following the same MLP structure as the original trajectory head. The pretrained model was kept frozen and used as a base planner, while the residual actor head predicted a probability distribution over trajectory corrections. The value head produced the scalar value estimate required by the PPO critic.
While the original trajectory head produces a base trajectory of the same form as equation 3.4, the residual actor head uses the query to predict a Gaussian distribution over residual corrections. Thus, in the PPO implementation, the action is defined as

$$a_t = \Delta\hat{\tau}_t = [(\Delta\hat{x}_{t+1}, \Delta\hat{y}_{t+1}, \Delta\psi_{t+1}), \ldots, (\Delta\hat{x}_{t+H}, \Delta\hat{y}_{t+H}, \Delta\psi_{t+H})] \quad . \tag{3.19}$$

The residual action is sampled from a Gaussian policy,

$$\Delta\hat{\tau}_t \sim \mathcal{N}\left(\boldsymbol{\mu}_\theta(o_t), \boldsymbol{\sigma}^2_\theta(o_t)\right) \quad , \tag{3.20}$$

where o_t denotes the rendered camera observation together with the ego-status feature. The residual mean is produced by the actor head, while the standard deviation is represented either as a learnable parameter or, in some experiments, as a fixed small value.
The final trajectory passed to the simulator is then

$$\hat{\tau}_t = \hat{\tau}_t^{\text{base}} + \Delta\hat{\tau}_t \quad . \tag{3.21}$$

Several residual formulations were explored during the PPO experiments. These ranged from applying corrections to the full predicted trajectory, including x, y, and yaw at all future poses, to applying only planar x and y corrections, or correcting only the first predicted poses in the trajectory. In the case of the full residual formulation, the corrected trajectory is computed as:

$$\hat{\tau}_{t,xy} = \hat{\tau}^{\text{base}}_{t,xy} + \Delta\hat{\tau}_{t,xy}, \qquad \hat{\tau}_{t,\psi} = \text{wrap}\left(\hat{\tau}^{\text{base}}_{t,\psi} + \Delta\hat{\tau}_{t,\psi}\right) \quad . \tag{3.22}$$

Additionally, the value head outputs a scalar estimate $V_\phi(o_t)$, which is used to compute the value loss and the advantage estimates during PPO training.

3.5.1.2 PPO Rollout and Update

As explained, at each step the planner produces a base trajectory and a residual sampled from the Gaussian policy, resulting in a corrected trajectory that is followed.

After the vehicle pose is updated, the simulator computes a reward and stores the transition in a rollout buffer. Each stored transition contains the camera and ego-status features, the sampled residual action, the action log-probability, the value estimate, the reward, and a termination flag indicating whether the episode has finished.
Depending on the experiment, the termination condition was defined in different ways. In all cases, an episode was considered completed when the simulator time reached the last timestamp of the reconstructed scene. In some variants, the episode could also be terminated earlier if the agent collided with a dynamic actor or left the drivable area. An overview of the rollout and update process is summarized in Figure 3.6. The figure shows, on the left, the data-collection phase, where the policy interacts with the simulator, from the optimization phase, to the right, where the collected information is used to update the policy and the critic.

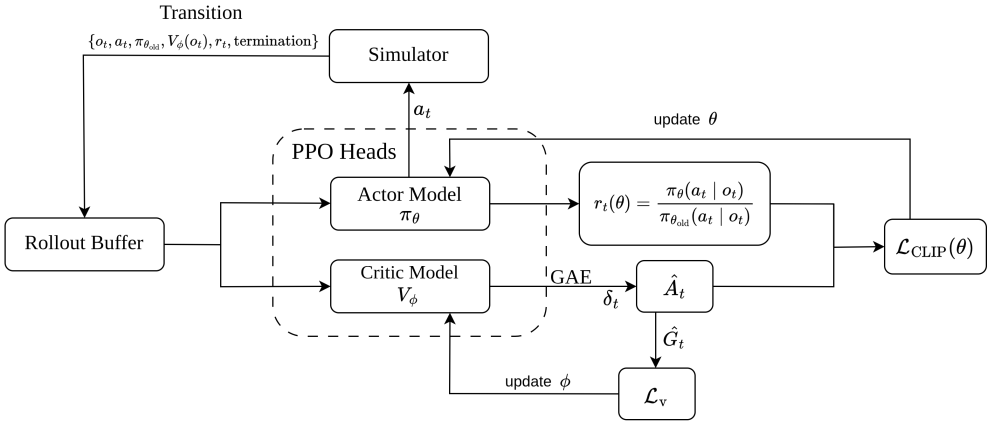


Figure 3.6: Overview of the PPO training loop.

After an episode is collected, the rollout buffer is used to compute the returns and the advantage estimates. In our implementation, equation 2.58 is estimated using Generalized Advantage Estimation (GAE), which computes an exponentially weighted sum of temporal-difference residuals:

$$\delta_t = r_t + \gamma(1 - d_t)V_\phi(o_{t+1}) - V_\phi(o_t) \ , \tag{3.23}$$

where d_t is the termination flag, and this difference is used for computing the advantage estimate as:

$$\hat{A}_t = \sum_{l=0}^{T-t} (\gamma\lambda)^l \delta_{t+l} \ , \tag{3.24}$$

where λ controls the bias–variance trade-off of the advantage estimate. Finally, the return target for the value function is computed as:

$$\hat{G}_t = \hat{A}_t + V_\phi(o_t) \ . \tag{3.25}$$

After the rollout buffer is filled, the collected transitions are used to update the policy. Instead of performing a single gradient step, PPO performs several optimization

epochs or passes over the same rollout data. At each PPO epoch, the stored transitions are shuffled and divided into minibatches. The stored residual actions are then evaluated again under the updated policy, producing new log-probabilities, entropy values, and value estimates.
In practice, the probability ratio is computed as

$$r_t(\theta) = \exp\left(\log \pi_\theta(a_t \mid o_t) - \log \pi_{\theta_{\text{old}}}(a_t \mid o_t)\right) \quad , \tag{3.26}$$

where $\pi_{\theta_{\text{old}}}$ denotes the policy that generated the rollout data. The policy is then optimized using the clipped PPO objective, value loss, and entropy bonus described in Section 2.6.2.

3.5.1.3 Reward Design

Several reward formulations were explored during the PPO experiments. The general objective was to encourage the planner to remain close to the logged route while avoiding invalid behavior such as collisions, off-road driving, and uncomfortable motion.
Initially, the reward design was inspired by GigaFlow [2]. Following the same general structure, the reward was defined as a weighted combination of multiple terms. However, some of the original terms, such as penalties for reversing, running red lights, or the per-step simulation penalty, were removed since they were not directly applicable to our simulator. Other terms were slightly modified to match the available simulator signals. The resulting reward function was:

$$r_t = r_{\text{collision}} + r_{\text{off-road}} + r_{\text{comfort}} + r_{\text{align}} + r_{\text{center}} + r_{\text{velocity}} + r_{\text{goal}} \quad , \tag{3.27}$$

where the individual terms can be interpreted as follows:

- $r_{\text{collision}}$ penalizes the agent for colliding with other dynamic actors. The penalty is bigger for collisions at higher speeds.
- $r_{\text{off-road}}$ penalizes the agent for leaving the drivable area.
- r_{comfort}: is a penalty for exceeding the comfortable limits of acceleration and jerk.
- r_{align} encourages the ego heading to align with the logged trajectory direction.
- r_{center} rewards the agent for having a small lateral deviation.
- r_{velocity} penalizes velocity error with respect to the logged trajectory.
- r_{goal} rewards the agent for reaching its final goal.

The precise mathematical form of these terms is shown in Table 3.1. In the table, $\mathbf{1}$ denotes an indicator function, e_ψ is the yaw error with respect to the logged trajectory, e_{lat} is the lateral error, e_{v_x} is the forward velocity error, and v_x is the ego forward velocity.
The reward weights α were randomized in some experiments following the GigaFlow-style randomization strategy. In those cases, the weights were sampled from the distributions shown in the second column of Table 3.1. In other experiments, the weights were sampled only once at the beginning of training in order to improve training stability.

Additionally, in some experiments, the alignment reward term r_{align} was replaced by a simpler yaw-error penalty of the form:

$$r_{\text{align}} = -\alpha_{\text{align}} \Delta t |e_\psi|. \tag{3.28}$$

Reward term	**Training distribution**
$r_{\text{collision}} = -(\alpha_{\text{collision}} + 0.1\lvert v_x \rvert)\, \mathbf{1}_{\text{collision}}$	$\alpha_{\text{collision}} \sim \mathcal{U}(0.5, 3.0)$
$r_{\text{off-road}} = -\alpha_{\text{off-road}}\, \mathbf{1}_{\text{off-road}}$	$\alpha_{\text{off-road}} \sim \mathcal{U}(1.0, 3.0)$
$r_{\text{comfort}} = -\alpha_{\text{comfort}} \big(\mathbf{1}_{\lvert a_x \rvert > 3} + \mathbf{1}_{\lvert a_y \rvert > 3} + \mathbf{1}_{\lvert j_x \rvert > 5 \vee \lvert j_y \rvert > 5} \big)$	$\alpha_{\text{comfort}} \sim \mathcal{U}(0.0, 0.1)$
$r_{\text{align}} = \alpha_{\text{align}}\, \Delta t \Big(\cos(e_\psi) + \alpha_{\text{vel-align}}\, v_x\, \cos(e_\psi) + 0.0025 \left(1 - \frac{\lvert e_\psi \rvert}{\pi/2} \right) \Big)$	$\alpha_{\text{align}} \sim \mathcal{U}(4.2, 5.5)$ $\alpha_{\text{vel-align}} \sim \mathcal{U}(0.0, 1.0)$
$r_{\text{center}} = -\alpha_{\text{center}}\, \Delta t \Big(\mathbf{1}_{\cos(e_\psi) > 0.5}\, \lvert e_{\text{lat}} - b_{\text{center}} \rvert - \frac{0.05}{\exp(\lvert e_{\text{lat}} - b_{\text{center}} \rvert - 0.5)} \Big)$	$\alpha_{\text{center}} \sim \mathcal{U}(7.5{\times}10^{-2}, 9.0{\times}10^{-1})$ $b_{\text{center}} \sim \mathcal{U}(-0.5, 0.5)$
$r_{\text{velocity}} = -\alpha_{\text{velocity}} \lvert e_{v_x} \rvert$	$\alpha_{\text{velocity}} \sim \mathcal{U}(0.1, 0.5)$
$r_{\text{goal}} = \mathbf{1}_{t > 0.75T}\, \mathbf{1}_{\lVert \mathbf{p} - \mathbf{p}_{\text{goal}} \rVert < \delta_{\text{goal}}}$	$\delta_{\text{goal}} \sim \mathcal{U}(3.0, 12.0)$

Table 3.1: Reward terms used during PPO training.

In practice, tuning a reward function with these many terms proved difficult. Some terms, such as collision and off-road penalties, were sparse, and others, such as alignment and velocity, were more informative but sensitive to scaling. For this reason, simplified rewards were also tested. Some of the simplest ones tested were, for example, the combination of lateral and velocity errors:

$$r_t = -e_{\text{lat}}^2 - \alpha_{v_x}\, e_{v_x}^2 \ , \tag{3.29}$$

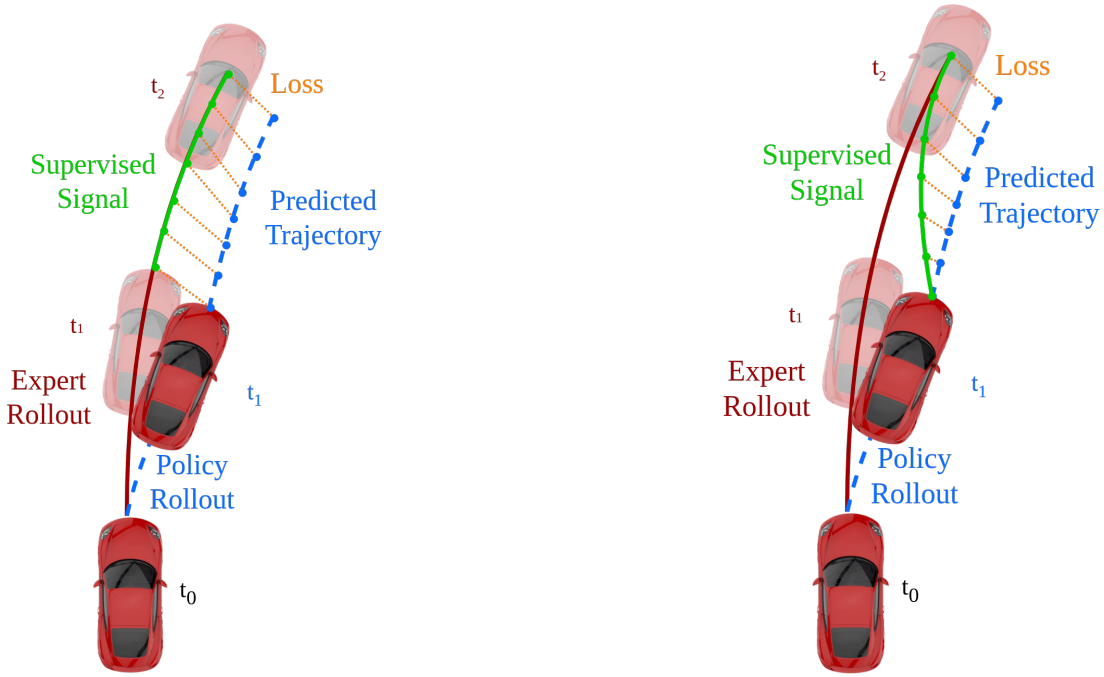
where α_{v_x} controls the relative importance of velocity tracking. An even simpler variant used only the lateral deviation penalty:

$$r_t = -e_{\text{lat}}^2 \ . \tag{3.30}$$

3.5.2 Closed-Loop Imitation Learning

The second type of training approaches explored in this work were based on supervised learning. The predicted trajectory from the planner is still executed inside

the simulator, and all the closed-loop behaviour is maintained. However, the signal used in the function to optimize, is directly obtained from the logged trajectory associated with the reconstructed scene.



(a) Supervised signal fully from logged trajectory.

(b) Artificial supervised signal reconstructed from logged trajectory and ego pose.

Figure 3.7: Different closed-loop imitation learning strategies.

3.5.2.1 Logged Trajectory Supervised Signal

The first supervised approach, as illustrated in Figure 3.7a, used the logged trajectory directly as the ground truth in the loss function from equation 2.54.
To construct this target, the current ego pose is first matched with the closest index on the logged trajectory. In practice, finding the closest logged pose in x, y was made more efficient by only searching within a local window around the previously matched index. Also, preventing big discontinuous jumps along the route:

$$i_t = \arg\min_{i \in \mathcal{W}_t} \|\mathbf{p}_t - \mathbf{p}_i^*\|^2 \quad , \tag{3.31}$$

where $\mathbf{p}_t$ is the current simulated ego position, $\mathbf{p}_i^*$ is the logged position at index i, and $\mathcal{W}_t$ is the local search window around the previously matched index.
After finding i_t, future logged poses were sampled at the same temporal spacing as the planner output, $\Delta t_\tau = 0.5\,\text{s}$, and transformed into the current ego frame.
The idea of this target generation is that it not only encodes the geometric path information but also the expected progression speed along it.

Following the residual formulation explored in PPO, one set of experiments kept the original Latent TransFuser planner frozen and trained an additional residual trajectory head to correct the base prediction $\hat{\tau}_t^{\text{base}}$. As in the PPO experiments, both planar residuals over x, y and full residuals over x, y, ψ were explored.
For the planar residual case, the target residual is:

$$\Delta\tau_{t,xy}^* = \tau_{t,xy}^* - \hat{\tau}_{t,xy}^{\text{base}} \quad , \tag{3.32}$$

and the loss was therefore computed using equation 2.53.When yaw residuals were included, the additional target are:

$$\Delta\tau_{t,\psi}^* = \text{wrap}\left(\tau_{t,\psi}^* - \hat{\tau}_{t,\psi}^{\text{base}}\right) \quad , \tag{3.33}$$

and then the loss followed equation 2.54.
Naturally, as mentioned in the stochastic residual approach, the actual trajectory executed in the simulator corresponds to the one from equation 3.21.
Additionally, another set of experiments removed the residual head and fine-tuned the original trajectory head directly. In this case, the loss was applied between the predicted trajectory $\hat{\tau}_t$ and the target τ_t^*, using equation 2.54.

3.5.3 Interpolated Supervised Signal

In the second supervised approach, the target was not taken directly from the logged trajectory. Instead, the target was artificially constructed by interpolating the current ego pose with the future logged trajectory, as illustrated in Figure 3.7b.
The motivation for this idea is that directly supervising the logged future trajectory can create unrealistic correction targets, especially when the deviation is large. In that case, the target may implicitly ask the planner to return to the logged trajectory too aggressively, almost as if the vehicle could instantly teleport back to the logged path.
As before, the current ego pose is first associated with an index i_t on the logged trajectory. Future logged poses are then sampled at the same temporal spacing as the planner output, $\Delta t_\tau = 0.5\,\text{s}$, and transformed into the current ego frame. Let us denote that trajectory as:

$$\tilde{\tau}_t^* = \left[\tilde{\mathbf{q}}_{t+1}^*, \ldots, \tilde{\mathbf{q}}_{t+H}^*\right] \quad . \tag{3.34}$$

Then, since the ego pose is the origin of this frame, the offset or difference between ego and the matched logged trajectory can be simply expressed as:

$$\mathbf{d}_{\text{offset}} = -\tilde{\mathbf{q}}_t^* \quad . \tag{3.35}$$

Afterwards, for each future step $k = 1, \ldots, H$, a decaying weight

$$w_k = 1 - \frac{k}{H} \tag{3.36}$$

is applied to this offset. Achieving the final interpolated supervised signal points as:

$$\mathbf{q}_{t+k}^* = \tilde{\mathbf{q}}_{t+k}^* + w_k \mathbf{d}_{\text{offset}}, \qquad k = 1, \ldots, H \quad . \tag{3.37}$$

Thus, the first target poses remain closer to the current ego pose, while the later target poses progressively converge back to the logged expert trajectory, as illustrated in Figure 3.7b. Additionally, after this operation, the yaw component is wrapped to preserve a valid angular representation.
In this set of experiments, both the approaches of training a residual head as well as, fine-tuning directly the original trajectory head were applied.

3.6 Evaluation Procedures

This section will explain the two different evluation methods that has been used during this work. First method consists on using NAVSIM v2 [25], as an external benchmark allowing that way comparison with other planners. Second, a fully closed-loop evaluation procedure was implemented inside our simulator, collecting several metrics that were considered central for autonomous driving.

3.6.1 NAVSIM v2 Benchmark Evaluation

The main motivation for including this method is that it provides a standardized evaluation framework, which is widely adopted across the autonomous driving research community. And thus, it allows a direct comparison with other planners under the same metric definitions and dataset split. In particular, in this work, the `navhard_two_stage` split and the Extended Predictive Driver Model Score (EPDMS) were used.
NAVSIM v2 evaluates planners on OpenScene-based data [51] using a two-stage aggregation process. This process aims to approximate closed-loop behaviour while still keeping the setting as a non-reactive simulation. On it, the background actors follow their recorded future trajectories, and the ego vehicle moves using a LQR controller.
Following the official explanation from [52], this two stage process works as follows:

1. First Stage Scoring: An initial scene is evaluated over a fixed horizon of 4 seconds, using the EPDMS metric. The simulator unrolls the ego vehicle pose using the LQR to follow the predicted trajectory.
2. Second Stage Scoring: After this initial scene rollout, multiple potential follow-up scenes are included in the test set to be evaluated. These follow-up scenes were pre-computed by rolling out several simulations starting from the initial scene, each with a different 4-second plan. All of these follow-up scenes will start then, from a different end point of the initial scene. The EPDMS metric is therefore computed again on each follow-up scene over a fixed horizon of 4 seconds.
3. Weighting and Aggregation: To emulate the effects of closed-loop simulation, the relevance of each follow-up scene to the overall score depends on how close its starting position is to where the planner actually ended in the first stage. So using a gaussian-kernel, higher weights are assigned to follow-up scenes that started closer to the planner's end position. Finally, to produce a unique metric, the scores of the first and second stage are aggregated via multiplication.

Now let us clarify the EPDMS metric. It was designed as a hybrid scoring function that penalizes safety-critical failures while rewarding driving progress and comfort. Mathematically, it is formulated as:

$$\text{EPDMS} = \underbrace{\prod_{m \in \mathcal{M}_{\text{pen}}} \text{filter}_m(\text{agent}, \text{human})}_{\text{penalty terms}} \cdot \underbrace{\frac{\sum_{m \in \mathcal{M}_{\text{avg}}} w_m \, \text{filter}_m(\text{agent}, \text{human})}{\sum_{m \in \mathcal{M}_{\text{avg}}} w_m}}_{\text{weighted average terms}} . \tag{3.38}$$

where:

$$\mathcal{M}_{\text{pen}} = \{\text{NC}, \text{DAC}, \text{DDC}, \text{TLC}\}, \qquad \mathcal{M}_{\text{avg}} = \{\text{TTC}, \text{EP}, \text{HC}, \text{LK}, \text{EC}\} \ , \tag{3.39}$$

are summarized in Table 3.2.
And the filter operation is defined as:

$$\text{filter}_m(\text{agent}, \text{human}) = \begin{cases} 1.0 & \text{if } m(\text{human}) = 0 \\ m(\text{agent}) & \text{otherwise} \end{cases} \tag{3.40}$$

Metric	**Role**	**Weight**	**Range**
No at-fault Collisions (NC)	Multiplier	–	$\{0, \frac{1}{2}, 1\}$
Drivable Area Compliance (DAC)	Multiplier	–	$\{0, 1\}$
Driving Direction Compliance (DDC)	Multiplier	–	$\{0, \frac{1}{2}, 1\}$
Traffic Light Compliance (TLC)	Multiplier	–	$\{0, 1\}$
Ego Progress (EP)	Weighted	5	$[0, 1]$
Time to Collision (TTC)	Weighted	5	$\{0, 1\}$
Lane Keeping (LK)	Weighted	2	$\{0, 1\}$
History Comfort (HC)	Weighted	2	$\{0, 1\}$
Extended Comfort (EC)	Weighted	2	$\{0, 1\}$

Table 3.2: Composition of the EPDMS metric. Multiplier terms act as hard penalties for inadmissible behaviors, while weighted terms reward positive driving quality. Metrics introduced in NAVSIM v2 with respect to v1 are DDC, TLC, LK, HC, and EC.

3.6.2 Custom Evaluation Inside the Simulator

In addition to the NAVSIM v2 benchmark, a fully closed-loop evaluation procedure was implemented directly inside the simulator. The purpose was to have a complementary view of planner performance under actual closed-loop execution.
This evaluation is performed on the 150 reconstructed scenes from the Argoverse 2 validation split, as mentioned in Section 3.1.
Each scene is evaluated independently as a single episode. The episode starts from the same initial ego pose as the logged trajectory, and proceeds until one of the following termination conditions is triggered:

- **Completed:** the scene reaches its final timestamp.
- **Collision:** a collision with a dynamic actor is detected, as explained in Section 3.4.5.
- **Off-road:** the ego vehicle leaves the drivable area according to the HD map.
- **Invalid render:** occurs when the ego deviates significantly enough from the original data collection trajectory that the Gaussian representation contains invalid numerical values.

During the rollout, the following error signals are computed by matching the current ego pose against the closest index on the logged ground truth trajectory as done in the training procedure (equation 3.31):

- **Displacement Error (DE):** The Euclidean distance between the current ego position and the matched ground truth position:

$$\mathrm{DE}_t = \sqrt{(x_t - x_t^*)^2 + (y_t - y_t^*)^2} \ . \tag{3.41}$$

- **Lateral Error:** The component of the position error perpendicular to the ground truth heading direction:

$$e_t^{\mathrm{lat}} = -\sin(\psi_t^*)\,(x_t - x_t^*) + \cos(\psi_t^*)\,(y_t - y_t^*) \ . \tag{3.42}$$

- **Longitudinal Error:** The component of the position error along the ground truth heading direction:

$$e_t^{\mathrm{lon}} = \cos(\psi_t^*)\,(x_t - x_t^*) + \sin(\psi_t^*)\,(y_t - y_t^*) \ . \tag{3.43}$$

- **Velocity Error:** The difference between the ego's forward velocity v_t^x and the forward velocity estimated from the ground truth trajectory at the matched index:

$$e_t^v = v_{x_t} - v_{x_t}^* \ . \tag{3.44}$$

All per-step errors are accumulated and averaged over the episode length to produce per-scene mean metrics. These are then aggregated across all evaluated scenes to produce the following final metrics:

- **Collision rate:** Fraction of scenes terminated by a collision.
- **Off-road rate:** Fraction of scenes terminated by leaving the drivable area.
- **Completion rate:** Fraction of scenes that ran until the final timestamp without any early termination.
- **Mean displacement error:** Average per-scene mean of DE_t across all steps.
- **Mean lateral error:** Average per-scene mean of $|e_t^{\mathrm{lat}}|$.
- **Mean longitudinal error:** Average per-scene mean of $|e_t^{\mathrm{lon}}|$.
- **Mean velocity error:** Average per-scene mean of $|e_t^v|$.

On where the termination-based metrics aim to reflect the safety and robustness of the planner under closed-loop, while the error-based metrics simply quantify how well the planner tracks the logged expert trajectory.

3.7 Implementation Details

This section describes some engineering decisions that made the pipeline feasible, as well as scalable.

Since the project depends on several components with compatibility requirements, including PyTorch, CUDA, SplatAD, the Argoverse 2 API, and NAVSIM, all dependencies were packaged into a single container image used across scene reconstruction, training, and evaluation.

As described in Section 3.2, each reconstructed scene folder contains everything needed to run a simulation episode. The SplatAD checkpoint, camera calibration, HD map, and dataparser transform. This self-contained design made it straightforward to distribute scenes independently across workers with no shared in-memory state between processes.

Multi-GPU training was implemented using PyTorch Distributed Data Parallel (DDP), following a scene-level parallelization strategy. At the start of each epoch, the scene list is shuffled and partitioned across ranks, with each rank loading and rolling out only its assigned scenes while the trainable planner parameters are synchronized through DDP during backpropagation. A practical complication is that different scenes produce rollouts of different lengths, either due to varying scene durations or early termination, meaning ranks can accumulate different numbers of optimizer steps within the same epoch. To handle this, a DDP join context was used so that ranks finishing early could wait for the rest without hanging. At the end of each epoch, the rank with the most optimizer steps was selected as source, and its model and optimizer state were broadcast to all other workers. Epoch-level summaries were then computed by reducing scalar statistics across ranks, and checkpoints and evaluation outputs were written exclusively by the main process.

Finally, since each reconstructed SplatAD scene is memory-intensive, scenes were loaded sequentially rather than kept in memory simultaneously. After completing all rollouts for a scene, the scene object was explicitly deleted, and the CUDA cache was cleared before loading the next one, keeping per-GPU memory usage bounded across all scenes.

4

Results

This chapter presents the main results of this work, starting with a qualitative assessment of the closed-loop simulator, and continuing with the results obtained by applying the different training strategies to the pretrained Latent TransFuser planner.

4.1 Qualitative Simulator Results

First, let us compared the visual quality of the simulator. Figure 4.1 illustrates the original logged frames, compared with renders from a simulator rollout on where the ground truth trajectory is executed.

Figure 4.1: Top row: original logged frames. Bottom row: corresponding SplatAD renders from similar poses.

Figure 4.2 illustrates rendering frames that were taken from completely new poses deviated from the logged trajectory. Mainly, noticeable artifacts are encountered on lane marks. Figure4.3 shows the three frontal cameras together, rendered from the same pose as the second column from Figure 4.2. There, it is possible to appreciate how, despite of large artifacts on the lanes, the entire render from the frontal-left camera retains a high rendering quality.

Figure 4.2: Degradation when rendering with lateral deviation from the logged trajectory.

Figure 4.3: Full view of the three frontal cameras, rendered from the deviated pose shown in the second column from Figure 4.2.

Finally, Figure 4.4 illustrates another common encountered artifact, which happens whenever a dynamic actor that has never been seen from the rear in the original trajectory, overtakes the ego vehicle during the simulation.

Figure 4.4: Left: First dynamic actor overtakes the ego vehicle. Right: A second dynamic actor overtakes the ego vehicle.

4.2 Training Results

This section presents the results obtained when adapting the base Latent TransFuser planner inside the proposed simulator. It first discusses initial single-scene feasibility experiments, and then presents the training results obtained when scaling the training strategies.
Before presenting the training results, it is of importance to remark the base performance of the Latent Transfuser planenr when deployed on the fully closed loop simulator. What we encounter is that it suffers a lot from compounding errors, which is very noticeable in the longitudinal behavior. A common failure mode was that the predicted trajectory points became progressively less spaced over time. Since the desired velocity is directly infered from the spacing between the predicted trajectory points, this caused the ego velocity to collapse, eventually stopping the vehicle in the middle of the road.

4.2.1 Feasibility Study

In order to assess the feasibility of closed-loop training within the simulator, the first experiments were focused on overfitting to a single reconstructed scene. The premise was simple, if the planner cannot improve its behavior on a single repeated sequence, scaling to hundreds of scenes is pointless.
The first approach attempted was the PPO-based reinforcement learning, as described in Section 3.5.1. This method proved highly unstable, on where frequently gradient explosions and NaN values appeared across different reward formulations and residual parameterizations. When training did not diverge numerically, the resulting driving behavior remained poor, with the planner failing to progress correctly on the scene. Figure 4.5 illustrates a representative rollout, where both speed and direction are mislearned and the vehicle eventually leaves the road.

Figure 4.5: Poor driving behavior resulting from PPO training.

Several factors may have contributed to this instability. The velocity collapse of the base planner is itself a complicating factor, since the residual policy had to simultaneously correct lateral deviations but also the velocity problem. Beyond that, the high-dimensional rendered observation space makes value estimation difficult for

the critic, since small changes in trajectory may alter future observations greatly, which destabilizes even more the policy gradient updates.
Several factors may have contributed to this instability. The velocity collapse observed in the base planner was itself a complicating factor, since the residual policy had to correct not only lateral deviations but also the longitudinal behavior of the vehicle. Beyond that, the high-dimensional rendered observation space makes value estimation difficult for the critic, and the closed-loop setting further aggravates this since small trajectory changes can significantly alter future observations. As a result, the policy-gradient updates became sensitive and difficult to stabilize.
The same single-scene overfitting experiment was then repeated using closed-loop imitation learning. As an initial supervised setup, only residual corrections in the x and y trajectory components were trained, using the expert trajectory as the supervised signal. In contrast to PPO, this approach produced a stable result very quickly. A single training run was sufficient to produce a clear overfit to the sequence. Figure 4.6 shows the fine-tuned planner completing the scene successfully, whereas the base planner collapses in velocity and stops around the fifth frame of the first row, having also drifted onto the left lane.

Figure 4.6: Single-scene overfitting using closed-loop imitation learning. The planner completes the reconstructed driving sequence after learning residual x, y corrections.

These checks confirmed that the simulator and training pipeline were functioning correctly. And given these results, PPO was not pursued further. The remaining experiments focused only on supervised closed-loop training.

4.2.2 Multi-Scene Training

As discussed in Section 3.5, several closed-loop imitation learning variants were explored. Since the simulator and the number of available reconstructed scenes were scaled progressively during the project, the experiments differ not only in the training strategy, but also in the number of training scenes used. This section reports only the most representative experiments, presented in the order they were conducted, since each configuration was usually motivated by observations from the

previous one. A summary of the NAVSIM evaluation results is shown in Table 4.1, while Table 4.2 shows the corresponding custom simulator metrics.

- **Experiment 1: x, y residuals with logged target.** Following directly from the feasibility study, this experiment scaled the residual x, y correction approach to 250 scenes over 72 epochs. The EPDMS score improved slightly over the base Latent TransFuser, but the Extended Comfort (EC) sub-metric degraded noticeably, suggesting that the x, y corrections were introducing heading inconsistencies.
- **Experiment 2: x, y, ψ residuals with logged target.** After scaling the simulator to the full training set, the residual formulation was extended to include yaw corrections. This experiment was trained on the 641 reconstructed training scenes for 160 epochs using 16 GPUs for 72 hours. Despite the larger training set and the additional correction degree of freedom, the EPDMS score collapsed significantly, dropping to approximately 11, well below the base planner.
- **Experiment 3: trajectory-head fine-tuning with logged target.** The third experiment removed the additional residual head and instead fine-tuned the original trajectory head directly, using the logged trajectory as the supervised target. Trained on the full training set for 100 epochs using 16 GPUs for 42 hours. The results were similarly poor, with EPDMS scores comparable to Experiment 2, suggesting that the problem was not specific to the residual formulation.
- **Experiment 4: x, y, ψ residuals with interpolated target.** The fourth experiment kept the residual-head formulation, but replaced the direct logged trajectory target with the interpolated supervised target described in Section 3.5.3. 60 epochs on the 641 scenes with 16 GPUs for 19 hours. The NAVSIM score no longer collapsed, remaining close to the base planner, though without a clear improvement.
- **Experiment 5: trajectory-head fine-tuning with interpolated target.** The final experiment also used the interpolated supervised target, but fine-tuned the original trajectory head instead. Run for 21 hours using 16 GPUs, completing 70 epochs on the full training set. Results were similar to the previous experiment, with no significant degradation but also no clear gain over the base planner.

Method	**Scenes**	**Epochs**	**EPDMS ↑**
Base LTF	–	–	25.00
Experiment 1	250	72	26.64
Experiment 2	641	160	13.60
Experiment 3	641	100	11.22
Experiment 4	641	60	25.40
Experiment 5	641	70	13.76

Table 4.1: NAVSIM v2 EPDMS results for the representative multi-scene training experiments.

Method	**Completion rate** ↑	**Collision rate** ↓	**Off-road rate**↓	**Lat. error** [m] ↓	**Vel. error** [m/s] ↓
Base LTF	0.30	0.58	0.12	2.09	2.13
Experiment 1	0.30	0.44	0.26	1.62	1.96
Experiment 2	0.36	0.44	0.20	2.08	1.90
Experiment 3	0.40	0.43	0.16	2.34	1.82
Experiment 4	0.36	0.45	0.19	2.26	1.86
Experiment 5	0.39	0.47	0.13	2.12	1.71

Table 4.2: Closed-loop simulator evaluation results for the representative multi-scene training experiments.

5
Conclusion

This chapter reflects on the main achievements and observations of this work, discusses its limitations, and outlines directions for further work.

5.1 Discussion

The main outcome of this work is the closed-loop simulator built from reconstructed data using SplatAD. Although the simulator was used in this thesis mainly for planner adaptation inside reconstructed Argoverse 2 scenes, it also provides a platform for future closed-loop experiments. Significant emphasis was placed on making the simulator self-contained and scalable, so that it can be extended beyond what was explored here.

One of the clearest observations from the experiments is that deploying the pretrained Latent TransFuser directly in closed loop reveals failure modes that are not visible from open-loop evaluation alone. In particular, the planner suffered from errors in the spacing of the predicted trajectory points, which compounded into velocity collapse during rollout. This reinforces the importance of closed-loop, either by closed-loop training as a corrective measure, or at least a fully closed-loop evaluation, that can showcase what is not captured during open-loop.

Although the training results were modest, they suggest that the simulator can be used as an environment for adapting an open-loop planner. The experiments also showed that the choice of learning signal is critical. PPO-based reinforcement learning was difficult to stabilize in this setting. The combination of high-dimensional rendered observations, sensitive reward terms, and the need to correct both lateral and longitudinal behaviour made the optimization unstable. Closed-loop imitation learning provided a much more stable signal, although the experiments also showed that naive supervised closed-loop training does not straightforwardly generalize when scaled to hundreds of scenes.

When interpreting the NAVSIM results, it is also important to consider the domain gap between our training and evaluation. The closed-loop training was performed inside reconstructed Argoverse 2 scenes, keeping the planner backbone frozen. NAVSIM evaluates the planner on a different dataset with different camera characteristics and visual appearance. This makes direct transfer to NAVSIM more challenging and may explain why improvements in the simulator did not translate into large gains in EPDMS.

Other limitations should also be considered when interpreting these results. First, the simulator inherits the constraints of the underlying 3DGS reconstruction. Which

is reliable close to the logged trajectory but degrades with large deviations, particularly for road markings and unseen perspectives on dynamic actors. Second, the simulator is non-reactive with respect to other agents, meaning surrounding vehicles follow their logged trajectories regardless of what the ego does. For example, an actor will overtake straight through the ego body if the planner drives too slowly. Finally, the computational cost of reconstructing and rolling out hundreds of scenes is not negligible, which limited the number of training variants and hyperparameter settings that could be explored in this work.

5.2 Future Work

As discussed in the previous section, several limitations remain open and could be addressed in future work. At the same time, the simulator developed opens multiple directions for further closed-loop and sensor-level training and evaluation.

One direction is to improve the visual robustness of the reconstructed scenes. Some of the limitations of Gaussian Splatting could be mitigated by applying diffusion-based refinement techniques, such as the ones explored in [53]. This could be especially useful for obtaining more consistent road markings and reducing artifacts in regions that are poorly observed in the original data. In combination with the ability to transform dynamic actor poses, this could also enable the generation of more challenging scenarios, where planners can be fine-tuned on rare or safety-critical situations.

A related extension would be to introduce a traffic-management layer for dynamic actors. An ideal closed-loop simulator would allow these actors to react to the ego planner's actions, for example, by braking or changing lanes. This would make the generated scenarios more realistic and would prevent inconsistent situations where a logged actor continues through the ego vehicle.

Another natural extension is to complete the sensor-level integration of the simulator. This worked only focused on rendering camera observations, while LiDAR is only used indirectly during the SplatAD reconstruction. Extending the framework to also provide LiDAR renders would make the simulator allow a broader class of end-to-end planners.

From an implementation perspective, rendering could also be further optimized. Currently, the different cameras are rendered sequentially. For example, with the current used planner, parallelizing the front-left, front-center, and front-right images could reduce rollout time significantly, allowing for even larger-scale experiments.

The simulator now provides a platform for exploring a much wider range of training strategies and planner architectures than what was investigated. Regarding PPO, a more stable base planner, one that does not suffer from velocity collapse, would likely make the approach significantly more stable and perhaps is worth revisiting. More broadly, a more systematic exploration of the training strategies is needed, including careful hyperparameter tuning and a more controlled analysis of why certain configurations degrade at scale or simply why some work better than others.

Finally, if the main objective is to improve performance on a specific benchmark, such as NAVSIM, it may be necessary to either reconstruct scenes with similar visual characteristics to eliminate the visual mismatch between training and evalu-

ation entirely. Or unfreezing parts of the backbone during training, aiming for the model to adapt its internal representations to the simulator's visual characteristics, potentially improving generalization to the evaluation domain.

Bibliography

[1] Y. Hu, J. Yang, L. Chen, K. Li, C. Sima, X. Zhu, S. Chai, S. Du, T. Lin, W. Wang, L. Lu, X. Jia, Q. Liu, J. Dai, Y. Qiao, and H. Li, "Planning-oriented autonomous driving," 2023.

[2] M. Cusumano-Towner, D. Hafner, A. Hertzberg, B. Huval, A. Petrenko, E. Vinitsky, E. Wijmans, T. W. Killian, S. Bowers, O. Sener, P. Kraehenbuehl, and V. Koltun, "Robust autonomy emerges from self-play," in *Proceedings of the 42nd International Conference on Machine Learning* (A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, eds.), vol. 267 of *Proceedings of Machine Learning Research*, pp. 11710–11737, PMLR, 13–19 Jul 2025.

[3] J. Gu, C. Hu, T. Zhang, X. Chen, Y. Wang, Y. Wang, and H. Zhao, "Vip3d: End-to-end visual trajectory prediction via 3d agent queries," 2023.

[4] xLAB for Safe Autonomous Systems, "Efficient 3d perception for autonomous vehicles zhijian liu mit," 2023.

[5] CARLA Simulator, "Getting started with CARLA." `https://carla.readthedocs.io/en/0.9.7/getting_started/`. CARLA documentation, version 0.9.7. Accessed: 2026-05-28.

[6] Q. Li, Z. Peng, L. Feng, Q. Zhang, Z. Xue, and B. Zhou, "Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning," 2022.

[7] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," 2023.

[8] G. Hess, C. Lindström, M. Fatemi, C. Petersson, and L. Svensson, "Splatad: Real-time lidar and camera rendering with 3d gaussian splatting for autonomous driving," 2025.

[9] Argoverse, "Argoverse 2 user guide," 2026. [Online; accessed 18-May-2026].

[10] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "Stanley: The robot that won the darpa grand challenge: Research articles," *J. Robot. Syst.*, vol. 23, p. 661–692, Sept. 2006.

[11] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar,

H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, *Autonomous Driving in Urban Environments: Boss and the Urban Challenge*, pp. 1–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[12] A. W. Harley, Z. Fang, J. Li, R. Ambrus, and K. Fragkiadaki, "Simple-bev: What really matters for multi-sensor bev perception?," 2022.

[13] Z. Li, W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Q. Yu, and J. Dai, "Bevformer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers," 2022.

[14] S. Doll, R. Schulz, L. Schneider, V. Benzin, E. Markus, and H. P. Lensch, "Spatialdetr: Robust scalable transformer-based 3d object detection from multi-view camera images with global cross-sensor attention," in *European Conference on Computer Vision(ECCV)*, 2022.

[15] Y. Liu, T. Wang, X. Zhang, and J. Sun, "Petr: Position embedding transformation for multi-view 3d object detection," 2022.

[16] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2722–2730, 2015.

[17] J. Janai, F. Güney, A. Behl, and A. Geiger, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," 2021.

[18] A. Hu, Z. Murez, N. Mohan, S. Dudas, J. Hawke, V. Badrinarayanan, R. Cipolla, and A. Kendall, "Fiery: Future instance prediction in bird's-eye view from surround monocular cameras," 2021.

[19] A. K. Akan and F. Güney, "Stretchbev: Stretching future instance prediction spatially and temporally," 2022.

[20] M. Bansal, A. Krizhevsky, and A. Ogale, "Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst," 2018.

[21] O. Scheel, L. Bergamini, M. Wołczyk, B. Osiński, and P. Ondruska, "Urban driver: Learning to drive from real-world demonstrations using policy gradients," 2021.

[22] B. Jiang, S. Chen, Q. Xu, B. Liao, J. Chen, H. Zhou, Q. Zhang, W. Liu, C. Huang, and X. Wang, "Vad: Vectorized scene representation for efficient autonomous driving," 2023.

[23] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad, "A survey of end-to-end driving: Architectures and training methods," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, p. 1364–1384, Apr. 2022.

[24] S. Ross, G. J. Gordon, and J. A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," 2011.

[25] D. Dauner, M. Hallgarten, T. Li, X. Weng, Z. Huang, Z. Yang, H. Li, I. Gilitschenski, B. Ivanovic, M. Pavone, A. Geiger, and K. Chitta, "Navsim: Data-driven non-reactive autonomous vehicle simulation and benchmarking," 2024.

[26] X. Liang, T. Wang, L. Yang, and E. Xing, "Cirl: Controllable imitative reinforcement learning for vision-based self-driving," 2018.

[27] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," 2017.
[28] S. Ross, G. J. Gordon, and J. A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," 2011.
[29] M. Simchowitz, D. Pfrommer, and A. Jadbabaie, "The pitfalls of imitation learning when actions are continuous," 2025.
[30] C. Gulino, J. Fu, W. Luo, G. Tucker, E. Bronstein, Y. Lu, J. Harb, X. Pan, Y. Wang, X. Chen, J. D. Co-Reyes, R. Agarwal, R. Roelofs, Y. Lu, N. Montali, P. Mougin, Z. Yang, B. White, A. Faust, R. McAllister, D. Anguelov, and B. Sapp, "Waymax: An accelerated, data-driven simulator for large-scale autonomous driving research," 2023.
[31] B. Osiński, P. Miłoś, A. Jakubowski, P. Zięcina, M. Martyniak, C. Galias, A. Breuer, S. Homoceanu, and H. Michalewski, "Carla real traffic scenarios – novel training ground and benchmark for autonomous driving," 2021.
[32] W. Ljungbergh, A. Tonderski, J. Johnander, H. Caesar, K. Åström, M. Felsberg, and C. Petersson, "Neuroncap: Photorealistic closed-loop safety testing for autonomous driving," 2024.
[33] K. Chitta, A. Prakash, B. Jaeger, Z. Yu, K. Renz, and A. Geiger, "Transfuser: Imitation with transformer-based sensor fusion for autonomous driving," 2022.
[34] W. Sun, X. Lin, Y. Shi, C. Zhang, H. Wu, and S. Zheng, "Sparsedrive: End-to-end autonomous driving via sparse scene representation," in *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8795–8801, 2025.
[35] B. Wilson, W. Qi, T. Agarwal, J. Lambert, J. Singh, S. Khandelwal, B. Pan, R. Kumar, A. Hartnett, J. K. Pontes, D. Ramanan, P. Carr, and J. Hays, "Argoverse 2: Next generation datasets for self-driving perception and forecasting," 2023.
[36] E. Vinitsky, N. Lichtlé, X. Yang, B. Amos, and J. Foerster, "Nocturne: a scalable driving benchmark for bringing multi-agent learning one step closer to the real world," 2023.
[37] K. Lis and T. Kryjak, "Pointpillars backbone type selection for fast and accurate lidar object detection," 2022.
[38] J. Philion and S. Fidler, "Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3d," 2020.
[39] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016.
[40] Z. Li, W. Yao, Z. Wang, X. Sun, J. Chen, N. Chang, M. Shen, Z. Wu, S. Lan, and J. M. Alvarez, "Generalized trajectory scoring for end-to-end multimodal planning," 2025.
[41] Epic Games, "Unreal engine 4." `https://www.unrealengine.com`.
[42] M. Müller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, "Sim4cv: A photo-realistic simulator for computer vision applications," *International Journal of Computer Vision*, vol. 126, p. 902–919, Mar. 2018.

[43] M. Martinez, C. Sitawarin, K. Finch, L. Meincke, A. Yablonski, and A. Kornhauser, "Beyond grand theft auto v for training, testing and enhancing deep learning in self driving cars," 2017.

[44] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," 2017.

[45] M. Goslin and M. R. Mine, "The panda3d graphics engine," *Computer*, vol. 37, no. 10, pp. 112–114, 2004.

[46] Wikipedia, "Non-uniform rational b-spline," 2026. [Online; accessed 13-May-2026].

[47] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," 2020.

[48] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[49] C. Lindström, "splatad." `https://github.com/carlinds/splatad`, 2025. GitHub repository.

[50] M. Tancik, E. Weber, E. Ng, R. Li, B. Yi, J. Kerr, T. Wang, A. Kristoffersen, J. Austin, K. Salahi, A. Ahuja, D. McAllister, and A. Kanazawa, "Nerfstudio: A modular framework for neural radiance field development," in *ACM SIGGRAPH 2023 Conference Proceedings*, SIGGRAPH '23, 2023.

[51] O. Contributors, "Openscene: The largest up-to-date 3d occupancy prediction benchmark in autonomous driving." `https://github.com/OpenDriveLab/OpenScene`, 2023.

[52] Autonomous Vision Group, "navsim/docs/metrics.md at main." `https://github.com/autonomousvision/navsim/blob/main/docs/metrics.md`, 2026. Accessed: 2026-05-23.

[53] W. Ljungbergh, B. Taveira, W. Zheng, A. Tonderski, C. Peng, F. Kahl, C. Petersson, M. Felsberg, K. Keutzer, M. Tomizuka, and W. Zhan, "R3d2: Realistic 3d asset insertion via diffusion for autonomous driving simulation," 2026.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY