Degree Project in Electrical Engineering, specializing in Systems, Control and Robotics

Second cycle, 30 credits

Design of a Modular Platooning and Traffic Coordination Simulator

KEVIN JAMSAHAR

Design of a Modular Platooning and Traffic Coordination Simulator

KEVIN JAMSAHAR

Master’s Programme, Systems, Control and Robotics, 120 credits Date:
November 19, 2025

Supervisors: Adrian Wiltz, Maria Charitidou
Examiner: Dimos Dimarogonas
School of Electrical Engineering and Computer Science Swedish title: Design av en modulär simulator för platooning och trafikkoordinering

© 2025 Kevin Jamsahar

Abstract

This thesis presents a modular and extensible simulation framework designed to facilitate the assessment and comparative analysis of different vehicle control algorithms and dynamics models. The simulator supports the creation of realistic and configurable road networks, including straight roads, curved sections, and intersections. It features a modular system architecture that allows users to integrate custom vehicle dynamics, controllers and road designs without changing core system components such as the road and vehicle managers. A step-by-step tutorial is included to guide users through the road network creation, vehicle configuration setup and simulation execution. Example scenarios to demonstrate the capabilities of the framework, such as the automatic connection of open ends, coordinated platoon merging maneuvers and vehicle re-entry from the virtual parking lot have been included. The proposed framework facilitates the development and testing of advanced control algorithms enabling the implementation of theoretical results on multi-vehicle planning and control to realistic multi-vehicle scenarios. Its flexibility and extensibility makes it a valuable tool for advancing real-world applications of vehicle platooning including coordinated truck convoys, advanced intersection handling and collaborative maneuvering using Vehicle-to-Vehicle (V2V) communication.

Keywords

Vehicle Platooning, Modular Simulation Framework, Closed-Loop Control, Trajectory Generation, Vehicle Dynamics, Controllers

Sammanfattning

Den här avhandlingen presenterar ett modulärt och utbyggbart simuleringsramverk, utformat för att underlätta bedömning och jämförande analys av olika fordonskontrollalgoritmer och dynamikmodeller. Simulatorn stöder skapandet av realistiska och konfigurerbara vägnät, inklusive raka vägar, kurvor och korsningar. Ramverket har en modulär systemarkitektur som gör det möjligt för användaren att integrera anpassad fordonsdynamik, kontroller och vägdesign utan att ändra på huvudkomponenter som exempelvis väg och fordons "managers". En steg-för-steg handledning ingår för att vägleda användaren genom skapandet av vägnät, inställning av fordonskonfiguration samt simuleringkörning. Exempelscenarier för att demonstrera ramverkets funktioner, som automatisk anslutning av öppna ändar, koordinerande sammanslagningsmanövrar för platooning och återinträde av fordon från en virtuell parkeringsplats har inkluderats. Det föreslagna ramverket underlättar utveckligen och testningen av avancerande kontrollalgoritmer och möjliggör implementering av teoretiska resultat inom flerfordonsplanering och styrning i realistiska scenarier. Dess flexibilitet och utbyggbarhet för det till ett värdefullt verktyg för att främja verkliga tillämpningar av fordonsplatooning, inklusive koordinerade lastbilskonvojer, avancerad korsningshantering och samarbets manövrering med hjälp av fordon-till-fordon (V2V) kommunikation

Nyckelord

Fordonskonvojkörning, Modulärt simuleringssystem, Sluten återkopplad styrning, Generering av referensbanor, Fordonsdynamik, Styralgoritmer

Acknowledgments

I would like to thank Adrian Wiltz for continually supporting me during my time working on this project. I would also like to thank Maria Charitidou for helping me for as long as she could.

Stockholm, November 2025
Kevin Jamsahar

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 2
1.3 Problem Formulation . . . 5
1.4 Features . . . 6
1.5 Delimitations . . . 7
1.6 Outline . . . 7

2 Background **9**
2.1 Object-Oriented Methodology . . . 9
2.2 United Modeling Language . . . 11
2.3 Design Patterns . . . 13
2.4 Control Objectives . . . 14
2.4.1 Lane Tracking . . . 14
2.4.2 Lane Switching . . . 14
2.4.3 Platooning With Merging and Splitting . . . 15
2.5 Control in Platooning and Traffic Coordination . . . 15
2.5.1 Control Strategies for Platooning and Coordination . . 16
2.5.1.1 Longitudinal control . . . 16
2.5.1.2 Lateral control . . . 17
2.5.2 Communication Types and Protocols . . . 18
2.5.3 Information Flow Topologies . . . 19
2.5.4 Infrastructure Involvement in Control . . . 20

3 System Design and Architecture **21**
3.1 Overview of System Modules . . . 21
3.1.1 TrafficEnvironment . . . 22
3.1.2 SimulationProject . . . 23
3.2 Manager Classes and Responsibilities . . . 24

3.2.1 RoadSegmentManager 24
3.2.2 Responsibilites and Core Operations 25
3.2.2.1 Segment creation 26
3.2.2.2 Connection Points 28
3.2.2.3 Segment Connection Management 30
3.2.2.4 Network Updates 32
3.2.2.5 Lane Boundaries 32
3.2.2.6 Virtual Parking Lot and Re-entry 32
3.2.2.7 Graph-Based Road Network Representation 33
3.2.2.8 Automatic Road Generation of Open Ends . 33
3.2.3 VehicleManager 40
3.2.3.1 Responsibilities and Orchestration 41
3.2.3.2 Vehicle Data Model 43
3.2.3.3 VirtualParkingLot and VirtualEntryPoint . . 45
3.3 Vehicle Subsystems 47
3.3.1 Reference Trajectory System 48
3.3.2 Sensor and Perception 52
3.3.3 Closed-Loop Control System 54
3.3.3.1 Lateral Control (steering) 54
3.3.3.2 Longitudinal Control (speed/spacing) 54
3.3.3.3 Vehicle Dynamics 55
3.3.3.4 Simulation Loop 55
3.4 Visualization 56
3.4.0.1 Lane trajectories 57
3.4.0.2 Coordinate Systems and World Screen Transform 58
3.5 Conventions and Terminology 58
3.5.1 Coordinate Frames 58
3.5.1.1 Global Coordinate System 58
3.5.1.2 Local Coordinate System 59
3.5.2 Coordinate Transformations 60
3.5.3 Application of the Transforms 62
3.5.4 Segment Conventions 62
3.5.5 Lane Identification 63
3.5.6 Driving Direction and Segment Orientation 64
3.6 Design Principles and Architectural Structure 64
3.6.1 Layered composition 64
3.6.2 Design patterns applied 65
3.6.3 Simulator structure: Motivation and Benefits 66

3.6.4 Trade-offs . . . 67
3.7 Technologies and Tools Used . . . 67

4 Multi-vehicle simulator: A Tutorial 69
4.1 Project Structure Overview . . . 69
4.2 Configuration and Customization . . . 70
4.2.1 Road Network Configuration . . . 70
4.2.2 Road Network Connection Configuration . . . 71
4.2.3 Automatic road network open end connection . . . 72
4.2.4 Implementation of Controllers and Dynamics . . . 77
4.2.4.1 Extending the Inputs of the Controller modules . . . 79
4.2.5 Vehicle Creation . . . 80
4.2.6 Vehicle Placement . . . 81
4.2.7 Virtual Parking Lot and Virtual Entry Points . . . 82
4.2.8 Saving/Loading and Logging with Visualization . . . 83
4.2.8.1 Project Saving and Loading . . . 83
4.2.8.2 Route Planning Configuration . . . 84
4.2.8.3 Logging Configuration . . . 85
4.2.8.4 Example Configuration To Save Project . . . 86
4.2.8.5 Example Configuration To Load Project . . . 87
4.3 Running the simulator . . . 87

5 Demonstration 89
5.1 Scenario Setup . . . 89
5.1.1 Road Network Topology . . . 92
5.1.2 Simulation Parameters . . . 93
5.1.3 Simulator Execution . . . 95
5.1.4 Key Observations . . . 96
5.1.5 Visual Results . . . 97
5.1.6 Summary . . . 101

6 Conclusions and Future work 103
6.1 Conclusions . . . 103
6.2 Outlook . . . 104

References 107

List of Figures

2.1 Standard notation for Unified Modeling Language (UML) class relation arrows [16]. Image by Yanpas, licensed under CC BY-SA 4.0. 12
2.2 Common information flow topologies: (a) Predecessor-Following (PF), (b) Two-Predecessor Following (TPF), (c) Predecessor-Leader-Following (PLF), (d) Two Predecessor-Leader-Following (TPLF), (e) Bidirectional (BD). 20

3.1 UML class diagram of the core system components. 21
3.2 UML class diagram illustrating the architecture of the RoadSegmentManager and its collaborating components. . . . 24
3.3 UML class diagram illustrating the relationship between the abstract RoadSegment superclass and its subclasses 26
3.4 Illustration of how curved road segments are defined, where θ is the central angle and r is the radius. 27
3.5 Straight (a), Curved (b) and Intersection segment (c) with respective connection points (red dots). 28
3.6 Orientation θ (green arrow) with respect to the global frame W and direction vector $\mathbf{v}$ (black arrow) at the connection points 29
3.7 UML class diagram illustrating the relationship between the `SegmentConnectionManager` and its helper classes, `Connect` and `NetworkUpdater` 30
3.8 Straight-segment connection ($\mathbf{o}_m$ at p_m): (A) before alignment with endpoints p_f, p_m and tangents vectors $\mathbf{t}_f, \mathbf{t}_m$ (both shown as arrows at the connecting connection points), (B) rotation by $\Delta\theta$ so $\mathbf{t}'_m$ is anti-parallel to $\mathbf{t}_f$, (C) translation by $\mathbf{t}_v$ so endpoints coincide, yielding the aligned configuration. 31

3.9 Schematic definition of left/right turning circles relative to a heading $\mathbf{v}$. The circle centers are offset by a distance r along the directions obtained by rotating the heading by $\pm 90°$. . . . 35
3.10 Turning circle center for start point p_s with heading v_s and end point p_e with heading v_e. At each position there is two possible circle centers exits, corresponding to left (s=+1) and right (s=-1) turns. The displacement vector $\Delta C = C_2 - C_1$ and its perpendicular $\hat{u}_\perp$ are shown for one configuration. . . . 36
3.11 External tangent construction Left-Straight-Left (LSL) (shown for $s_1 = s_2 = +1$). Two equal-radius turning circles with the same orientation with two external tangents. The bottom branch (thick blue) is chosen, and the alternate (gray) is discarded. 37
3.12 Internal tangent construction Right-Straight-Left (RSL) (shown for $s_1 = -1$, $s_2 = 1$). Two internal tangents arise from $\theta = \alpha \pm s_1\delta$ with $\delta = \arccos(2r/D)$. The chosen branch (blue) uses $\hat{t} = (\cos\theta_A, \sin\theta_A)$, which gives $T_1 = C_1 + r\hat{t}$ and $T_2 = C_2 - r\hat{t}$. The alternate branch (gray) uses θ_B. The straight segment length is $L = \|T_2 - T_1\| = \sqrt{D^2 - (2r)^2}$. . . . 38
3.13 UML showing the relationship of the `VehicleManager` and its associate components. 40
3.14 UML of the overall `Vehicle` architecture. 43
3.15 A vehicle driving as usual in (a), and a vehicle that has crashed in due to high velocity in (b). 44
3.16 The vehicle before entering the virtual parking lot from a unconnected connection point. 45
3.17 The vehicle after exiting the road network and entering the virtual parking lot. 46
3.18 Generated reference trajectory from the vehicle visualized as a red line. 48
3.19 Information flow in the reference trajectory generation and route planning system. 49
3.20 Block diagram of the closed-loop update at each time step. . . 54
3.21 One straight, curved and intersection segment connected sequentially with blue lane trajectories along each lane center. 57
3.22 The straight segment and intersection segment was created with an orientation of $0°$ while the curved segment has a $270°$ orientation. 59

3.23 Visual representation of lane indexing on lanes relative to the road segments local coordinate system 63

4.1 Road network graph before automatic road generation to connect open ends. 74
4.2 Physical road network before automatic road generation to connect open ends. 75
4.3 Network graph after automatic road generation connecting the chosen open ends. 76
4.4 Physical road network after automatic road generation to connect open ends. 77

5.1 Layout of the road network used in the demonstration scenario. 92
5.2 Layout of the road network used in the demonstration scenario. 93
5.3 Vehicles with their respective names in the road network . . . 94
5.4 Vehicle with their respective names in the road network 95
5.5 Vehicles entering a curved road from a straight segment (seen in the upper portion of the image). 98
5.6 Vehicle 1 executing a left turn in the left image, Vehicle 2 doing a right turn in the intersection in the right image. 98
5.7 Multiple vehicles navigating parallel lanes. 99
5.8 Vehicle before performing a lane changing maneuver. 99
5.9 Vehicle during a lane change maneuver being executed. 100
5.10 Vehicle after completing a lane changing maneuver successfully. 100
5.11 Vehicle just before entering the virtual parking lot (right image), Vehicle after entering the virtual parking lot (left image). 101
5.12 Vehicle just before re-entering the road network 101

List of Tables

4.1 Relevant Parameters by Segment Type 71

5.1 Road segment configuration for the simple scenario. 89
5.2 Vehicle configuration for the simple scenario. 90
5.3 Road segment configuration for the complex scenario. 90
5.4 Vehicle configuration for the complex scenario. 91

List of acronyms and abbreviations

ACC	Adaptive Cruise Control
BD	Bidirectional
C	Curve
CACC	Cooperative Adaptive Cruise Control
CC	Cruise Control
CCW	Counter-Clockwise
CS	Curve-Straight
CSC	curve-straight-curve
CW	Clockwise
FOV	Field-of-view
GUI	Graphical User Interface
LSL	Left-Straight-Left
LSR	Left-Straight-Right
MPC	Model Predictive Control
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
PF	Predecessor Following
PID	Proportional-Integral-Derivative
PLF	Predecessor Leader Following
RSL	Right-Straight-Left
RSR	Right-Straight-Right
RSU	Road Side Unit
S	Straight
SC	Straight-Curve

TPF	Two-Predecessor Following
TPLF	Two-Predecessor Leader Following
UML	Unified Modeling Language
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle

Chapter 1

Introduction

1.1 Motivation

The development of autonomous vehicles has received significant attention over the past years. As technology advances, improvements in hardware such as sensors, processors, and communication units enable more sophisticated transportation solutions that are less constrained by technological bottlenecks allowing for solutions that can improve efficiency, safety and traffic flow. This has led to the implementation of more advanced vehicle coordination approaches. Vehicle coordination refers to the cooperative operation of multiple vehicles dynamically adjusted based on shared information or observed behavior to achieve a common transportation goal. This includes applications such as coordinated merging and splitting, intersection handling, and vehicle platooning, to name a few.

Platooning, in particular refers to the coordination of multiple vehicles that travel in close formation, often maintained through Vehicle-to-Vehicle (V2V) communication, advanced sensing, and automated control. This coordination of vehicles has been shown to reduce fuel consumption by up to 10% through a decrease in aerodynamic drag [1]. Platooning also has the potential to improve safety, by reducing the reliance on human drivers, who are responsible for more than 90% of road accidents [2], making platooning an important area of research for future intelligent transportation systems. However, achieving these benefits through the usage of control strategies requires accurate and robust control algorithms together with realistic vehicle dynamics models. These approaches must be validated under realistic road and traffic conditions to ensure their effectiveness and reliability.

Although several studies ([3] - [4]) have proposed custom simulation tools,

these often feature tightly coupled designs with hard-coded dynamics, road geometries, and control logic created for specific use cases. This limits their flexibility and reusability and makes it difficult to compare different control algorithms or reuse components across different scenarios.

In this thesis a modular simulation framework is presented which allows independent configuration of road segments and plug-and-play integration of vehicle dynamics and control algorithms. The framework is designed to facilitate testing and evaluation of numerous vehicle coordination strategies in diverse traffic environments.

1.2 Related Work

Research in vehicle coordination and platooning often relies on custom simulation environments to validate control strategies such as overtaking, merging, splitting, lane changing and maintaining inter-vehicle gaps. These studies (e.g., [4], [5] and [6]) provide valuable insight into specific aspects of cooperative vehicle behavior, however, their simulation setups are often scenario-specific and integrate tightly coupled control logic, vehicle models, and road layouts that limit extensibility, reusability, and maintainability. Here, control logic refers to the algorithms implemented in the controllers that determine vehicle behavior (e.g., how they accelerate, brake, or steer). When this logic is tightly coupled to specific vehicle models or road layouts, it is hard-coded with fixed dynamics or geometry, which makes it difficult to reuse in different scenarios.

Many existing studies propose control architectures created for specific vehicle coordination challenges. For example, [3] introduces robust distributed control protocols for large platoons and validate them using simulation. The simulation evaluates the performance and robustness of the proposed control protocols under second-order nonlinear vehicle dynamics and external disturbances incorporated into the control design by verifying that inter-vehicle spacing and connectivity are maintained.

Similarly, [7] presents a framework, where the purpose is to optimize the timing in which vehicles merge and the spacing for cooperative merging on highway on-ramps to reduce fuel consumption and travel time. Their method uses a centralized control approach that computes optimal merging times based on vehicle positions and velocities, assuming reliable V2V communication. The proposed framework is validated through a simulation using custom-designed scenarios that represent on-ramp roads and traffic conditions.

Furthermore, [5] presents a cooperative overtaking algorithm for vehicle platoons on freeways using V2V communication. The proposed approach is validated using simulated traffic scenarios specifically designed for freeway environments.

The work in [8] proposes a framework **COMPANION**, for coordination and management of heavy-duty vehicle platoons through a hierarchical architecture. The framework divides the hierarchy into strategic, tactical, and operational layers, which handle the planning of control of platoon operations. It is validated through both simulations and real-world trials, which highlights the broader ecosystem required for cooperative platooning. While such works emphasize large-scale coordination strategies, they rely on specialized architectures and are less focused on providing modular simulation environments for controller experimentation.

Although these studies [3, 5, 7] successfully validate their control strategies, they rely on custom simulation environments that are made for specific tasks, controllers, or road configurations. In each case, the simulator is tightly integrated with the proposed control logic and underlying assumptions. For example, the simulator in [3] assumes the same fixed dynamics for all vehicles and integrates controller-specific logic. This makes it difficult to adapt the setup to alternative control strategies or vehicle behaviors.

Similarly, [7] validates their centralized merging strategy using simulation scenarios specifically configured for highway on-ramps, with fixed vehicle behaviors, merging geometry, and communication assumptions. The freeway scenarios in the simulation used in [5] have also constructed their cooperative overtaking algorithm around V2V communication assumptions. The control logic and communication assumptions are embedded into the simulation setup, which makes it difficult to decouple components or generalize the framework for broader coordination and platooning use cases.

Additionally, most of these works model road networks exclusively as straight road segments or in more advanced cases as on-ramp segments merging straight roads. This is because the implementation of curves, intersections, and roundabouts is generally more complex and computationally demanding. This severely limits the ability to evaluate the coordination strategies in the papers since they don't allow more realistic and diverse traffic scenarios.

In contrast, the framework developed in this thesis supports these complex road geometries in a modular and extensible way. Other studies have progressed towards more generalized control logic, but still rely on constrained simulation environments. For example, [6] proposes decentralized lane-

switching coordination without switching controllers, but their method is implemented within fixed simulation scenarios specific to their algorithm. Similarly, [4] uses Signal Temporal Logic to control platoon splitting and merging behaviors but their simulation platform is constructed primarily for verifying logical specifications and lacks extensibility for general purpose control evaluations.

Studies such as [9] recognize these limitations and propose a modular simulation framework for platooning coordination. The framework integrates the network and traffic simulation capabilities of existing tools to enable the analysis of cooperative vehicle behavior in user-defined scenarios through an intuitive Graphical User Interface (GUI). The framework also supports switching between predefined controllers such as Adaptive Cruise Control (ACC) and Cooperative Adaptive Cruise Control (CACC), however, it lacks standardized plug-and-play abstractions for seamless integration of new controllers or vehicle dynamics, which limits extensibility and reusability.

Another relevant initiative is [10], which introduces **CommonRoad**, a framework that provides a large collection of standardized and composable benchmarks for motion planning on roads, where each benchmark specifies a road network represented as lanelets, together with initial state, goal regions, static or dynamic obstacles, and the model of the ego vehicle. The road networks are mostly provided as predefined scenarios, which are recorded or hand-crafted, although, users can also create new lanelets through the CommonRoad scenario designer. The primary focus of CommonRoad is on reproducibility and comparability of motion planning algorithms across research groups. However, it does not support plug-and-play integration of closed-loop controllers or vehicle dynamics, which limits flexibility compared to frameworks designed for modular integration of custom models and controllers.

Another widely used platform is **CARLA** [11], an open-source urban driving simulator that provides high-fidelity 3D environments for testing autonomous driving tasks such as perception, planning and control. **CARLA** is able to provide realistic renderings, detailed traffic scenarios and sensor simulation (e.g., cameras, LiDAR, radar) which makes it a powerful tool for autonomous driving research. However, its architecture is less focused on modular abstractions for vehicle dynamics or controller integration, which limits its sustainability for systematic controller benchmarking and comparative evaluation of coordination strategies.

Complementary to these efforts, **Duckietown** [12] provides an open, low-cost, and modular hardware/software platform for autonomy education and

research. Road networks are created from interchangeable tiles, including straights, curves, and intersections, which enable reproducible closed-loop experiments with multiple vehicles. Despite Duckietown's demonstration and advantages of modular infrastructure and standardized software stacks, its main focus is on education and small-scale robotics. The platform also relies on differential-drive robots with simplified dynamics and front facing cameras as its only onboard sensor, with road layouts that are constrained to a tile-based design. These factors limit scalability and generalization to more diverse road geometries such as roundabouts or parametrically defined curves.

These studies and platforms highlight important advances in vehicle coordination research including platooning, motion planning benchmarks, and modular experimental setups. However, they also reveal gaps that make it difficult to extend existing systems, fairly compare competing approaches, or replicate results across different research contexts.

This thesis addresses these gaps by introducing a modular simulation framework designed for vehicle platooning research. The framework decouples the creation of different roads, vehicle dynamics, and controller logic into independently configurable modules. It provides abstractions for road layout, vehicle instantiation, and control and dynamic model integration.

These features collectively enable reproducible experiments, comparative benchmarking of controllers, and the construction of flexible scenarios.

1.3 Problem Formulation

Many simulation-based studies on vehicle coordination already exist, however, they are often tightly coupled to specific experimental setups. These implementations frequently use fixed road layouts, hard-coded vehicle dynamics, and controller logic designed for specific use cases. As a result, they lack modularity and extensibility, which makes it difficult to reuse components, adapt scenarios, or compare different control strategies under standardized conditions.

To address these issues, this thesis contributes a modular simulation platform. The framework allows vehicle dynamics models, road network structures, and control strategies to be configured independently. It supports the configuration of different road segments such as straight roads, curves and intersections. Furthermore, it supports the implementation of different control algorithms and dynamic models that can be interchanged through a defined plug-and-play interface that supports evaluation and comparison.

Moreover, the thesis provides a comprehensive tutorial accompanied

by example scenarios that demonstrate the frameworks main features. These include the automatic connection of open road ends as well as the implementation of coordinated platoon merging maneuvers, and vehicle re-entry from a central virtual parking lot. This makes the simulator a flexible and suitable choice as a research tool that supports experimentation with different vehicle coordination strategies across various realistic traffic scenarios.

1.4 Features

The developed simulation framework is designed to be modular, extensible, and reproducible at its core. It supports the creation of realistic traffic scenarios, the interchange of simulation components, and flexible configurations for vehicle behavior and control.

In this context:

- **Modular** means that the simulator is built from independent, interchangeable components such as road segments, vehicle dynamics and controllers that can be used without modifying other parts.
- **Extensible** means that new components can be added, for example new road geometries or control algorithms by implementing a standard interface without requiring changes to the existing code base.
- **Reproducible** means that experiments can be repeated under identical conditions, with deterministic scenario setup to ensure consistent results.

Users can construct road networks from modular configurable road segments, including straight segments, curves and intersections. Each of these has configurable attributes that change the properties of the segment. Segments can be connected at defined connection points, which allows the formation of complex road topologies that preserve lane continuity and allows vehicles to transition seamlessly between segments. The framework also supports road network exit points through which vehicles can enter the virtual parking lot or re-enter the network.

Vehicle dynamics and controllers follow a plug-and-play design, which allows arbitrary models and controllers to be integrated without modifying other components. Each vehicle operates in a closed-loop, following dynamically generated reference trajectories that are aligned with the road topology and updated as the vehicle progresses through the network.

A visualization component is included to support real-time observation and post-simulation analysis. The visualization renders the road network layout and movement of the vehicles as their states are updated throughout the simulation. This enables both network-level observation of traffic flow and detailed inspection of individual vehicle maneuvers.

1.5 Delimitations

The following delimitations have been applied to define the scope of this work:

- **Simulation Constraints:** The simulator does not yet support multi-lane merges or on-ramps/off-ramps. The simulation is deterministic, so lane changes are predictable to ensure repeatability and testability.
- **Routing and Lane Management:** Route instructions are predefined and followed deterministically. If no instructions remain, a default fallback instruction of `straight` is applied. Dynamic rerouting or adaptation to traffic conditions is not supported. The simulator provides the structure for implementing a high-level controller for route planning or lane management and its development is left as an interesting topic for future research.
- **Road Segment Handling:** This work supports the modular segment types: `Straight`, `Curved`, and `Intersection`, each handled independently. Road segments do not have inherent directionality, which means that any vehicle can drive in any lane.
- **Parking Lot:** Vehicles that leave the road network are stored in a virtual parking lot.
- **Visualization and Coordinate Systems:** The simulator uses a global coordinate system for logic and a screen coordinate system for rendering. Visualization is 2D, implemented using Pygame integrated with Cairo. Minor visual artifacts may appear at connected boundaries.

1.6 Outline

Chapter 2 presents preliminaries on Unified Modeling Language (UML), design patterns, and Object-Oriented Programming (OOP). Chapter 3 describes the design and architecture that went into the development of the

simulation framework. Chapter 4 explains how to use the simulator for testing and visualization. Chapter 5 demonstrates the effectiveness of the simulator and its functionality in a plethora of simulation examples. Chapter 6 provides a summary of the thesis contributions and discusses different directions for future research.

Chapter 2

Background

This chapter provides both theoretical and technical background information necessary to understand the design principles and scope of the proposed simulation framework. It covers UML, design patterns, and object-oriented methodologies which were found to be effective tools to create a modular vehicle platooning simulator.

2.1 Object-Oriented Methodology

Object-oriented methodology can be referred to as the structured approach of developing software systems. It encompasses three distinct but related phases, called Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), and Object-Oriented Programming (OOP) [13]. These phases guide the development process from an idea to a practical implementation.

- OOA: This is the first phase of the development process where the purpose is to analyze a problem domain or a task and identifying the system's functional requirements. First, conceptual objects are determined along with their respective attributes and behaviors, and then the interactions and relationships between the objects are introduced. This process forms a conceptual model that focuses on the required functionality rather than the implementation specifics.

- OOD: This phase is responsible for turning the conceptual model in the OOA into implementable software design specifications with classes, their responsibilities, and interactions. In OOD, these specifications describe the classes to be implemented, their responsibilities and interactions. They also define their class hierarchies and relationships

using object-oriented principles such as encapsulation (grouping data and the methods that operate on it within a class), inheritance (allowing classes to reuse attributes and behaviors from other classes), abstraction (hiding implementation details behind an interface) and polymorphism (allowing different objects to be used according to their own specification).

- OOP: This is the last phase in the process, where the focus is on transforming the software design in the OOD into a design model using an object-oriented programming language. The classes in the design are translated into code, and the system is developed by instantiating and composing objects that represent real-world entities and operations.

This linear development approach could be appropriate for smaller systems, but it would be inefficient and significantly difficult to manage in large-scale systems which often require an iterative development model [13]. In such models, analysis, design, and implementation are iterated repeatedly in development cycles to better handle increasingly complex and expanding systems with evolving requirements.

A central aspect of this methodology is OOP itself. OOP is a programming paradigm in which the core concept is *objects*. Objects represent the class that they are instantiated from. They encapsulate their data and behavior, which allows this structure to model real-world entities, thus improving modularity, testability, and readability, all of which are derived from the four core principles of OOP as follows [13]:

- **Encapsulation:** Encapsulation refers to the grouping of data and the methods that operate on it within a class. It restricts direct access to an object's internal state by only exposing controlled interfaces improving security, maintainability, and modularity. This is enforced by many programming languages through access modifiers, however, languages like Python use naming conventions and design practices to perform encapsulation.

- **Inheritance:** Inheritance allows a subclass to obtain attributes and methods from a superclass. This makes the code more reusable and simplifies the creation of class hierarchies. For example, a vehicle superclass could define common attributes and methods that are shared by subclasses such as car, truck, and bicycle by extending the superclass. The attributes that are shared by these subclasses could be properties

such as the name of the manufacturer or the top speed of the vehicle, and the methods could be start or stop as a vehicle action.

- **Abstraction:** Abstraction is the hiding of complex internal implementation details needed for a task. For example, a simulator could define abstract interfaces for vehicle dynamics, making different models such as unicycle or bicycle models integrable and interchangeable without changing the logic that ensures this plug-and-play capability.
- **Polymorphism:** Polymorphism allows different objects to be treated uniformly using a common interface, which makes the code more reusable, flexible, and maintainable. It also gives subclasses the ability to override or extend functionalities to required specification.

These principles are fundamental in designing scalable and maintainable software systems.

2.2 United Modeling Language

UML is a general modeling language that is designed to graphically represent the architecture, design and behavior of software systems [14]. It gives developers the ability to describe the static structures and dynamic behavior of systems through many different types of diagrams, which are selected depending on the modeling objectives and system requirements.

By providing a common visual language, UML bridges the gap between abstract software design and concrete implementation by simplifying communication and providing a shared understanding among people with diverse technical backgrounds through graphical intuitive notations. This shared representation reduces the need for complex explanations.

Furthermore, UML has many different diagrams, but the most common diagram used within software development is the class diagram. This diagram provides a view of a system's structure by detailing its class, attributes, methods and their relationships. Because UML itself is object-oriented, it integrates well with OOP principles and supports the creation of modular, reusable and maintainable software components. This makes UML especially valuable in systems designed to use an object-oriented structure.

Additionally, UML supports both forward and reverse engineering, which means that the source code can be generated from design models or the design models can be reconstructed from existing code [15]. This flexibility allows

UML to be integrated seamlessly across all phases of software developments lifecycle.

In this thesis, UML class diagrams are mainly used to describe the architecture of the system and the relationships between the simulator's core software components. To ensure a consistent interpretation of all diagrams, a standardized notation is followed, as shown in Figure 2.1.

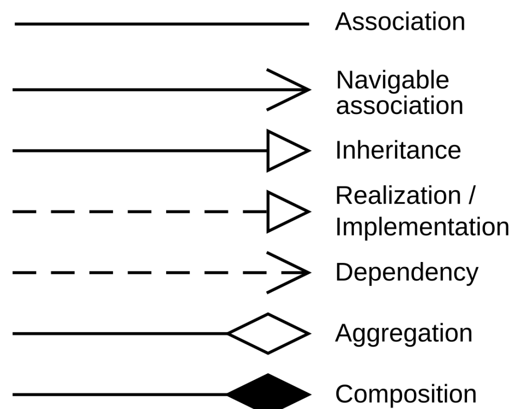

Figure 2.1: Standard notation for UML class relation arrows [16]. Image by Yanpas, licensed under CC BY-SA 4.0.

The figure shows the most common UML relationships, where most of them were used in this work:

- **Association**: Represents a link between two classes, indicating that objects of one class are connected to objects of another.
- **Navigable association**: A directed association where one class can access another, but not necessarily the other way around.
- **Inheritance**: A relationship where one class derives from another and inherits its attributes and methods.
- **Realization/Implementation**: Represents a relationship between an interface and a class that implements its operations.
- **Dependency**: Denotes a temporary relationship where one class uses another through parameters, local variables, or method calls but does not own it.

- **Aggregation**: A relationship in which the contained object can exist independently of the container.
- **Composition**: Similar to aggregation but with stronger ownership, where the contained object lifetimes depend on the container.

These conventions are applied uniformly to all UML figures in the thesis.

2.3 Design Patterns

Design patterns are established solutions to commonly occurring software design problems that arise in object-oriented systems. They are adaptable templates that emphasize reusable design by promoting best practices for solving problems through time tested and proved solutions [17]. These patterns improve the maintainability, flexibility and scalability of software systems through their extensible design [18].

Design patterns are commonly divided into three pattern groups [18]:

- **Creational Patterns** focus on object creation by abstracting the instantiation process. They help make the system independent by decoupling it from how its objects are created, composed, and represented, which results in greater flexibility in the system.
- **Structural Patterns** focus on the composition of classes and objects to form larger and more functional structures. These patterns simplify relationships between objects and classes, which makes the system more flexible and easier to maintain.
- **Behavioral Patterns** are concerned with the interaction and communication between objects or classes and the way responsibilities are distributed.

Many design patterns exist, however, for this thesis, only a subset of these patterns are of relevance. The patterns are as follows:

- **Factory Method (Creational)**: Defines an interface for creating objects, but lets subclasses decide which concrete class to instantiate.
- **Composite (Structural)**: Allows individual objects and compositions of objects to be treated uniformly.

- **Façade (Structural)**: Provides a simplified interface to a larger subsystem by wrapping multiple internal components behind one unified access point, thereby hiding their internal complexity from the user.
- **Adapter (Structural)**: Converts the interface of a class into another interface the user/client expects so that otherwise incompatible components can work together.
- **Strategy (Behavioral)**: Encapsulates interchangeable algorithms behind a common interface so they can be swapped without modifying the client code.
- **State (Behavioral)**: Represents different states of an object explicitly, allowing its behavior to change when its internal state changes.
- **Template Method (Behavioral)**: Defines the skeleton of an algorithm in a base class while allowing subclasses to override specific steps.

These design patterns play a crucial role in building modular and extensible modern software systems. They promote good design and code reuse, which reduce the need for frequent refactoring when new requirements or expansions are needed later in the development. This contributes to the system's long-term maintainability.

2.4 Control Objectives

2.4.1 Lane Tracking

Lane tracking is a fundamental objective in autonomous driving and requires a vehicle to follow the center of a lane. The reference path is derived from the road geometry and expressed in global coordinates. However, for control, the vehicle's deviation from this path is typically computed in the vehicle's local frame, where cross-track and heading errors can be defined consistently.

2.4.2 Lane Switching

Lane switching or lane changing maneuvers are a crucial coordination problem for autonomous vehicles. In this problem, vehicles transition from one lane to another, often in response to route planning decisions, obstacle avoidance

or cooperative driving behavior. These maneuvers are handled through coordinated lateral and longitudinal actions and situational awareness.

2.4.3 Platooning With Merging and Splitting

Vehicle platooning is the result of coordinated movement of multiple vehicles traveling closely together in the same lane with desirable inter-vehicle gaps. Merging occurs when a vehicle joins an existing platoon from a neighboring lane or, a standstill position or from a highway on-ramp. This requires synchronized deceleration or acceleration of the platoon members to create a suitable gap. The merging vehicle must align both its speed and lane position to integrate seamlessly, often requiring coordination through communication protocols such as V2V.

Splitting refers to the process where one or more vehicles exit the platoon, either to change lanes, take an off-ramp, or operate independently. This process must be handled in a way that maintains safety and stability of the remaining platoon. Departing from the platoon involves temporarily relaxing the inter-vehicle spacing constraints and handoff of leader-follower roles within the platoon.

Operations such as adaptive control and real-time decision-making ensure smooth transitions. The control architecture must support dynamic reassignment of leader and follower roles, robust gap management, and trajectory replanning. Communication-based strategies such as CACC significantly improve the responsiveness and reliability of merging and splitting maneuvers compared to sensor-based approaches alone.

2.5 Control in Platooning and Traffic Coordination

Coordinated control of autonomous vehicles plays a significant role in enabling platooning behavior and improving overall traffic efficiency. Effective coordination requires appropriate control strategies, reliable communication technologies, and well-defined communication topologies between vehicles and infrastructure. The control strategy determines how vehicle actions are computed, communication protocols define how information is exchanged, and the topology specifies which vehicles communicate with one another.

2.5.1 Control Strategies for Platooning and Coordination

Vehicle coordination relies on a combination of longitudinal and lateral control strategies, where both are defined with respect to the vehicle's local coordinate frame. Longitudinal controllers regulate vehicle speed and inter-vehicle gaps along the forward x-axis to ensure safe and stable platoon formation. Lateral controllers manage lane tracking, trajectory following and maneuvers such as lane changes or intersection turns with respect to the vehicle's local upward y-axis frame. The reference path that vehicles are expected to follow is derived from the road geometry and expressed in global coordinates, however, deviations such as cross-track and heading errors are computed in the vehicle's frame to ensure consistent control performance. the choice of control strategy depends on factors such as the the vehicle dynamics and the control objectives [19]. Together, these controllers enable individual vehicles to behave cohesively as a part of a larger system [19].

2.5.1.1 Longitudinal control

Longitudinal controllers regulate vehicle acceleration and deceleration to maintain a desired velocity while complying with traffic flow or keeping a safety distance from other vehicles. In platooning, longitudinal control is crucial for maintaining inter-vehicle gaps and ensuring coordinated movement of all vehicles within a platoon or convoy.

Mathematically, these objectives can be formulated as follows [6]:

The vehicle tracks a desired reference velocity v_{ref} by enforcing

$$v_E - v_{\text{ref}} = 0, \tag{2.1}$$

where v_E denotes the velocity of the ego-vehicle (the vehicle whose motion is being controlled relative to surrounding vehicles). To ensure safety, a minimum time headway is enforced to the vehicle ahead through the condition

$$b_1(x) = x_{0F} - x_E - \tau_D v_E \geq 0, \tag{2.2}$$

where x_{0F} is the longitudinal position of the preceding vehicle, x_E the ego-vehicle position, v_E the ego-vehicle velocity, and τ_D a headway parameter. Equation (2.1) expresses a control objective, where the ego vehicle's velocity

should converge asymptotically to the desired reference velocity, while Equation (2.2) represents a hard safety constraint that must be satisfied at every time step to ensure that ego-vehicle maintains a distance proportional to its velocity, thereby avoiding collisions.

Some common longitudinal control strategies include:

- **Cruise Control (CC):** Maintains a constant desired velocity. It is simple and does not account for other vehicles or traffic conditions.
- **ACC:** Uses onboard sensors to dynamically adjust acceleration and by either braking or accelerating to maintain a safety distance to the vehicle ahead. ACC reacts to sensor input and is therefore limited by sensor noise and latency, and lacks forward planning.
- **CACC:** Extends ACC by incorporating V2V or Vehicle-to-Infrastructure (V2I) communication to receive velocity, acceleration and driving intent data from one or more vehicles. This improves string stability and reduces reaction delay.

2.5.1.2 Lateral control

Lateral controllers regulate vehicle steering by ensuring that the vehicle maintains its lane, follows the roads accurately or performs lane changes and turns. It ensures that the vehicle follows the intended path while maintaining lateral stability, which is especially important during high-speed maneuvers or when navigating complex roads.

Mathematically, for lateral control two common error measures are defined. These are the *heading error* (e_ψ) that represents the difference between vehicle orientation and the reference trajectory, and the *cross-track error* (e_y), the lateral deviation from the reference path. The vehicle state (x, y, ψ) is measured in global coordinates, while the error measures are computed in the vehicle's local frame for consistency. They are typically formulated as

$$e_\psi = \psi_{\text{ref}} - \psi, \tag{2.3}$$

$$e_y = y_{\text{ref}} - y, \tag{2.4}$$

where (x, y, ψ) denotes the vehicle state and $(x_{\text{ref}}, y_{\text{ref}}, \psi_{\text{ref}})$ a reference trajectory point in global coordinates. Controllers then act to minimize e_ψ and e_y through steering inputs.

Some common lateral control strategies include:

- **Proportional-Integral-Derivative (PID) Controller:** A feedback control method that computes steering angles based on proportional, integral, and derivative terms of the lateral error. It is easy to implement, but requires careful tuning.
- **Model Predictive Control (MPC) Controller:** A predictive control strategy that optimizes future steering actions over a time horizon by solving a constrained optimization problem. MPC can handle dynamic constraints and future reference points, making it suitable for complex scenarios.
- **Pure Pursuit Controller:** A geometric path-tracking algorithm that calculates the required steering angle to reach a look ahead point on the path by assuming a circular arc. It is easy to implement and performs well at low to moderate speeds.
- **Stanley Controller:** A control strategy that minimizes both the heading error and cross-track error between the vehicle and the reference path. It is robust at high speeds and commonly used in highway driving scenarios.

Among these control strategies, the PID controller is generally the simplest to implement and is widely used in practice especially in the early stages of development.

2.5.2 Communication Types and Protocols

Communication protocols enable vehicles to exchange information with each other or with external systems. Some common types are:

- **V2V:** Enables direct communication with vehicles that are within the sensing range of the vehicle under consideration. Vehicles share state information such as speed, acceleration, position, heading and control information including braking status, path prediction, lane change intent and desired inter-vehicle gap [20].

- **V2I:** Enables vehicles to receive information from road infrastructure such as traffic signals, traffic cameras or Road Side Unit (RSU)s. V2I is essential for maintaining safety by coordinating traffic at intersections and managing large-scale traffic operations [21].

These technologies form the foundation for V2X (Vehicle-to-Everything) systems and are typically implemented using DSRC or C-V2X protocols [22].

2.5.3 Information Flow Topologies

Information flow topologies define how vehicles communicate in a connected vehicle system. They dictate which vehicle receives data from whom and directly impact control performance, stability, safety and fuel consumption [23]. Typical communication topologies include [23]:

- **Predecessor Following (PF):** Each vehicle receives data only from its immediate predecessor. This is simple and scalable but prone to error propagation.
- **Two-Predecessor Following (TPF):** Vehicles communicate with two vehicles ahead. This increases robustness but introduces more communication overhead.
- **Predecessor Leader Following (PLF):** Vehicles receive data from both the immediate predecessor and the platoon leader [24]. This improves disturbance rejection and string stability.
- **Two-Predecessor Leader Following (TPLF):** Each vehicle receives information from the two vehicles directly ahead of it.
- **Bidirectional (BD):** Include communication with vehicles behind which is useful in some formation control contexts but rare in platooning.

The choice of topology depends on the goals of the system, such as responsiveness, safety margins, and available bandwidth. The different topologies can be seen in Figure 2.2.

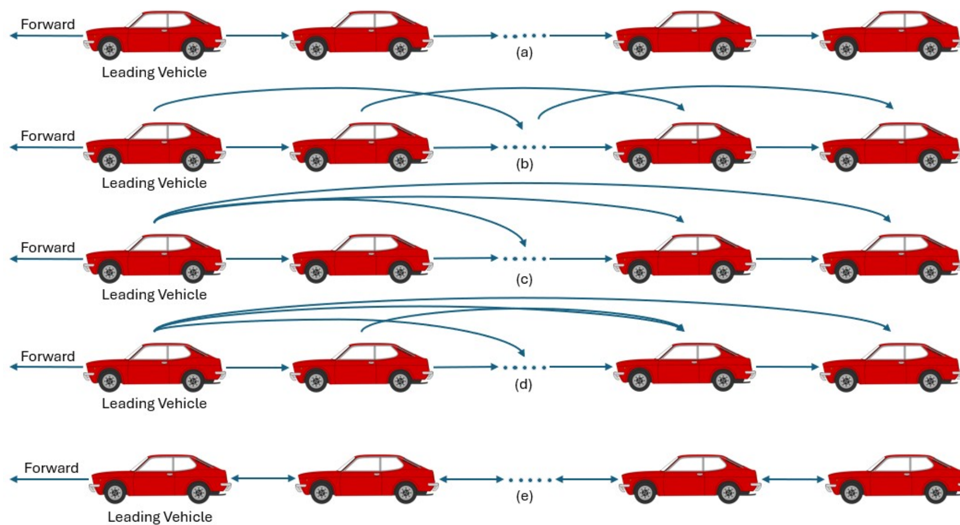

Figure 2.2: Common information flow topologies: (a) Predecessor-Following (PF), (b) Two-Predecessor Following (TPF), (c) Predecessor-Leader-Following (PLF), (d) Two Predecessor-Leader-Following (TPLF), (e) Bidirectional (BD).

2.5.4 Infrastructure Involvement in Control

Infrastructure-based control strategies use external systems to coordinate vehicle behavior:

- **Centralized Control:** A central unit, such as a traffic management server or cloud controller collects global state information from all vehicles and distributes control commands [7]. This approach enables global optimization, but suffers as it has scalability limitations and is vulnerable to single points of failure.

- **Decentralized Control:** Each vehicle independently determines control actions based on local observations using sensor data or information received from V2V communication [6]. This method is scalable and robust, but limited by situational awareness.

Chapter 3

System Design and Architecture

This chapter introduces the fundamental components of the modular vehicle coordination and platooning simulator. The system is designed with flexibility and extensibility at its core to support various road structures, dynamic vehicle behavior and customizable control logic.

3.1 Overview of System Modules

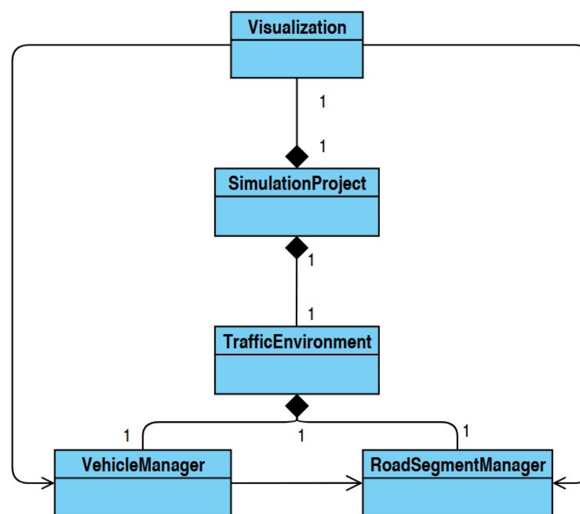

Figure 3.1: UML class diagram of the core system components.

The simulator architecture is organized into two categories of components.

1. Core managers, which maintain the road network, vehicles, and visualization.
2. Orchestration and execution of classes, which integrate the managers into a cohesive system and control the simulation flow.

At the core of the architecture are three managers:

- `RoadSegmentManager`: Responsible for creating, connecting, and maintaining road segments.
- `VehicleManager`: Responsible for vehicle creation, placement and updates during the simulation. It ensures that vehicles interact correctly with the road network, coordinates their perception and control modules, and manages route progress and segment transitions.
- `Visualization`: Responsible for rendering the road network and vehicles. Vehicles are rendered for each simulation step, while the road network and trajectories are only updated when a zoom/panning/resizing action is detected.

The manager classes are integrated by the `TrafficEnvironment`, which coordinates their executions. Its structure ensures that all updates to roads and vehicles are synchronized at each simulation step.

Lastly, the `SimulationProject` (Section 3.1.2) provides the entry point for controlled experiments. It instantiates the `TrafficEnvironment`, runs the simulation loop, and manages functionality such as vehicle data logging and optional video recording.

The relationships between these modules are shown in Figure 3.1.

3.1.1 TrafficEnvironment

The simulation framework is coordinated by the `TrafficEnvironment` class, which acts as the central integration layer between the core system components. It instantiates and owns the `RoadSegmentManager` and `VehicleManager`, and provides a unified interface for operating them.

The `TrafficEnvironment` delegates logic to managers through high-level methods. These methods include:

- **Road network management**: creation and connection of segments (`create_road_segment()`, `connect_road_segments()`,

`automatically_generate_road_to_connect_open_end_-segments()`, trajectory generation (`generate_trajectories()`), and cleanup of unused or single segments (`cleanup_network()`).

- **Vehicle management**: creation (`create_vehicles()`), placement on segments (`add_vehicle_to_segment()`), removal of inactive (not placed on the network) vehicles (`cleanup_vehicles()`), and simulation updates (`update_simulation`).

By forwarding operations to the underlying managers, the `TrafficEnvironment` keeps the road network and vehicles in sync during the simulation. It acts as a coordinator, while each manager is responsible for its own domain-specific tasks.

3.1.2 SimulationProject

The `SimulationProject` class represents the entry point for running experiments with the simulator. It instantiates the `TrafficEnvironment` and manages the simulation loop by ensuring that the time is advancing, vehicles and roads are updated, and that the visualization is refreshed. Its main responsibilities are:

- Initializing the `TrafficEnvironment` with a given configuration
- Executing the simulation loop, including event handling, updates, and time progression.
- Integrating `Visualization` to render the road network and vehicles.
- Supporting optional video recording of the simulation run.
- Collecting vehicle data through an enabling post-simulation plotting.

The central flow is defined in the `start_simulation()` method which at each simulation step performs the following tasks:

1. Handles user input events (panning, zooming, resizing).
2. Calls `update_simulation()` on the `TrafficEnvironment` to update road and vehicle states.
3. Updates visualization and optionally records a video.

4. Logs vehicle states for analysis.

5. Increments the global simulation clock and enforces the configured time step.

3.2 Manager Classes and Responsibilities

The simulator implements a modular design where responsibilities are separated into dedicated manager classes. Each manager focuses on coordination tasks, while specialized functionality is delegated to supporting modules. This section describes the main manager classes in detail and highlights their associated modules.

3.2.1 RoadSegmentManager

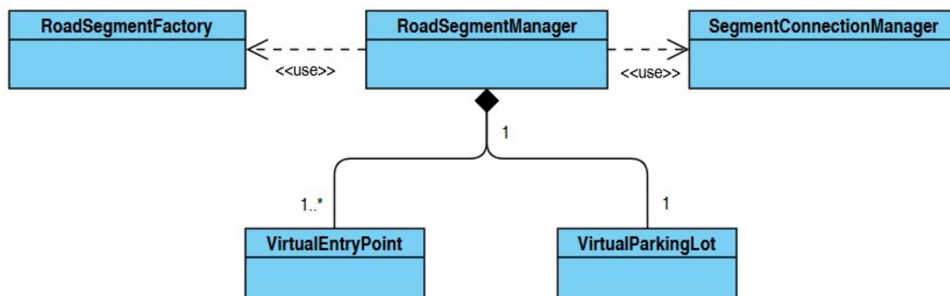

Figure 3.2: UML class diagram illustrating the architecture of the RoadSegmentManager and its collaborating components.

The `RoadSegmentManager` is responsible for creating, connecting, and maintaining road segments. It also specifically differentiates `fixed_segments`, which are segments that have been committed to the network, and `moving_segments`, which are segments that have not yet been connected, thereby not committed. The managers, structure provides a centralized interface for road operations and ensures that simple and complex road networks are managed consistently. It collaborates with the `RoadSegmentFactory`, `VirtualEntryPoint`, `VirtualParkingLot` and `SegmentConnectionManager` in order to handle segment creation, geometric alignment, and integration with vehicle entry and re-entry points (see Figure 3.2).

3.2.2 Responsibilites and Core Operations

During scenario setup, the manager builds segments via a factory, assigns unique identifiers to them, and records connection points together with their states (`CONNECTABLE`/`CONNECTED`). When two segments are connected, the required rotation and translation is performed by the `connect_segments()` method in the `SegmentConnectionManager` so that the selected connection points coincide, and updates both connection-point states accordingly.

Each segment encapsulates its own local coordinate frame and geometry. Coordinate transforms such as translation and rotation between local and global frames (local_to_global, global_to_local), or polar to cartesian conversions (polar_to_cartesian and cartesian_to_polar) for curved and intersection roads are delegated to the segment classes. This ensures that trajectories and poses are expressed consistently in the global frame.

At runtime, other modules query the manager, typically retrieving a segment by an identifier to obtain lane metadata (e.g., lane boundaries, width and speed limit), check if a position is within a segment or a lane, and to detect whether a position is inside an overlap region, i.e., a small spatial zone between two connecting segments. This prevents edge cases where a vehicle transitions between segments and gets a point between the segments due to rounding errors. The manager also provides utilities through the segments connection points, such as selecting the next segment reached from a given point, mapping lane continuity across a connection, and computing the intended driving direction relative to an entry point. A graph view of the network is maintained to support visualization of the connections, where the nodes represent connection points and edges represent segments.

The `VehicleManager` also needs a reference to the `RoadSegmentManager` to confirm lane membership as a vehicle moves, to resolve transitions when leaving a segment or entering an overlap region, and to fetch the receiving segment and lane for the next simulation step.

Furthermore, the manager provides methods such as `find_open_end_segments()`, which find all segments with open ends in the road network, `connect_nearby_segments()` that automatically connects segment connection points that align and are sufficiently close to each other, and `automatically_generate_road_to_connect_open_end_segments`, which connects two open ends with larger gaps using curve-straight-curve (CSC) construction (see Section 4.2.3).

3.2.2.1 Segment creation

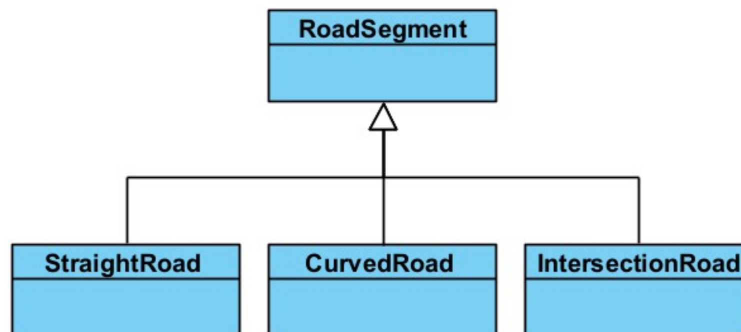

Figure 3.3: UML class diagram illustrating the relationship between the abstract RoadSegment superclass and its subclasses

Segment creation is handled by the `RoadSegmentFactory` through `create_road_segment()` method, which instantiates the appropriate segment type (`StraightRoad`, `CurvedRoad` or `IntersectionRoad`) based on parameters such as orientation, lane count, and geometric configuration. Each created segment is an instance of the abstract base class (superclass) `RoadSegment`, which defines the shared interface and common functionality for all road segment types (see Figure 3.3). The `RoadSegment` superclass defines the generic attributes and methods common to all road segments including: `segment_id`, `length`, `orientation`, `lanes`, `road_type`, `lane_width`, `local_origin` and `road_overlap` (see Section 4.2.1).

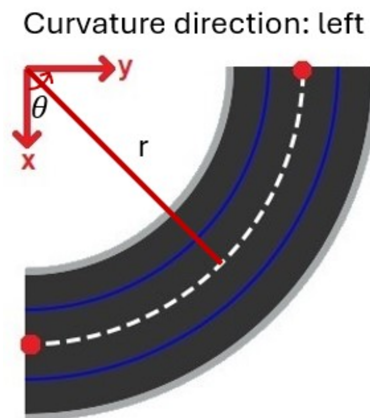

Figure 3.4: Illustration of how curved road segments are defined, where θ is the central angle and r is the radius.

In the current implementation, the following subclasses are defined to represent different types of segments:

- **StraightRoad**: Represents a straight road as a linear road segment. These segments can be used for highways and urban roads.
- **CurvedRoad**: Models a curved road as a modular arc segment, which allows transitions between directions. Each curved segment is defined by its radius, central angle, and curvature direction (left or right) (see Figure 3.4).
- **IntersectionRoad**: A segment that includes four entry and exit points (start, end, left, and right), each internally represented as a subsegment, where each subsegment extends the abstract base class `RoadSegment`.

3.2.2.2 Connection Points

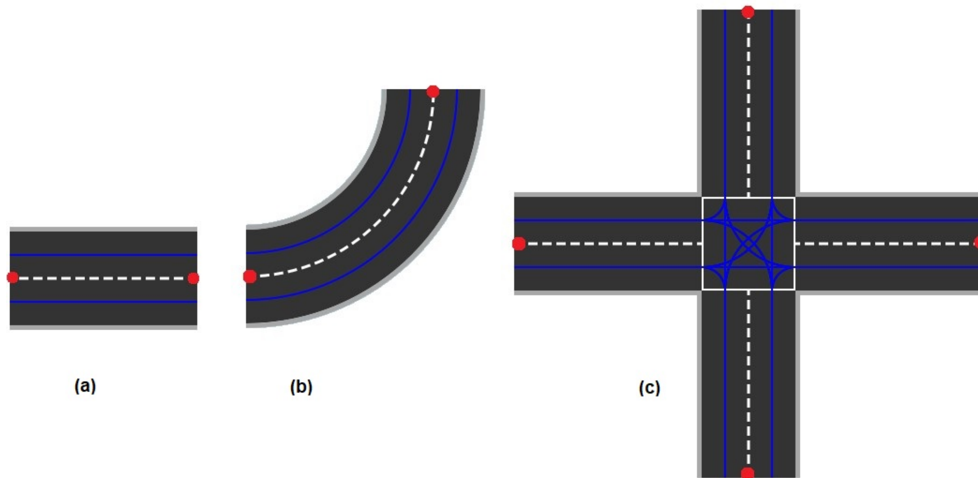

Figure 3.5: Straight (a), Curved (b) and Intersection segment (c) with respective connection points (red dots).

Each road segment has two or more connection points that represent the point of contact where other segments can be connected (see Figure 3.5). In the implementation, a connection point is represented by `ConnectionPoint` class and encapsulates geometric attributes and connectivity relationships. Connection points are characterized by their identity and geometry, where each point has a `point_id` (for example, `"start"`, `"end"`, or for intersections `"left"/"right"`) and `segment_id` of the road segment to which it belongs. Positions are stored in the segment's local frame and in the global frame. An `orientation` specifies the heading angle of the road tangent at the connection point, expressed in the global frame and a `direction` vector which represents the tangent of the road centerline pointing outwards at the connection point (see Figure 3.6).

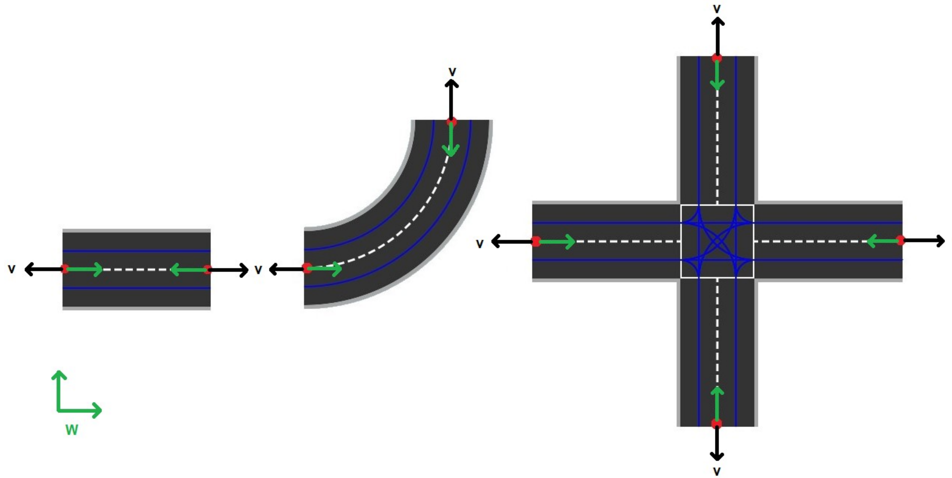

Figure 3.6: Orientation θ (green arrow) with respect to the global frame W and direction vector **v** (black arrow) at the connection points

To represent the road segments, each connection point stores the segment's `lane_width`, the number of lanes (`lanes`), and an `overlap_length` which defines a short overlap region that extends out from the connection (see Section 4.2.1). This overlap region is used to ensure that the segment transition is completed after the vehicle has left the overlap zone. In that way, we can avoid cases where the vehicle's position is not inside an area between two segments that may potentially exist due to numerical errors or inaccuracies. The method `is_position_within_overlap()` checks whether a given vehicle position falls inside the zone and returns `True`, which triggers a delay in the transitions between segments to ensure that the vehicle has completely transitioned over to the next segment.

Connection points also track their `state` which is represented as an `Enum` type with two values:

- **CONNECTABLE**: available for connection.
- **CONNECTED**: already occupied by another segment.

This ensures that the points are only connected once. If a connection is established between two segments, the fields `connected_road_id` and `connected_point_id` identify the segment and endpoint. In this way, the connectivity between two segment connection points is established. However, to ensure lane continuity between connected segments, the `connected_lanes` defines a mapping from each local lane in the current

segment to the corresponding lane at the connected segment's connection point. Since lane identifiers (`lane_id`) may not align directly due to differences in orientation or connection geometry (see Section 3.5.1.2 and 3.5.5), this mapping resolves the correspondence and guarantees consistent lane continuity across the connection. This mapping is established through the method `connect_lanes()` and queried by `get_next_lane()`. These lane connectivity links are essential for trajectory generation and for longitudinal objectives such as leader selection.

Together, these mechanisms allow the `SegmentConnectionManager` to perform geometric alignment, the `RoadSegmentManager` to maintain connectivity consistency, and the `VehicleManager` to detect overlap entry, determine the correct receiving lane, and carry out smooth transitions across connected segments.

3.2.2.3 Segment Connection Management

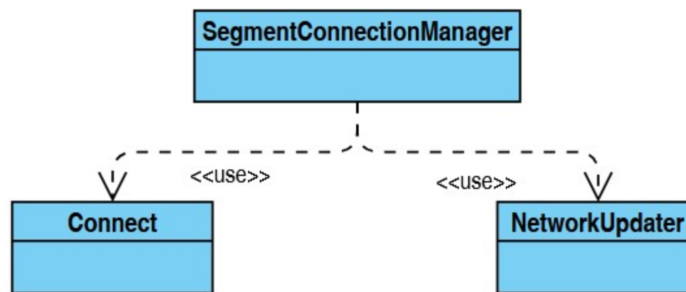

Figure 3.7: UML class diagram illustrating the relationship between the `SegmentConnectionManager` and its helper classes, `Connect` and `NetworkUpdater`

Connections are managed by the `SegmentConnectionManager` class, specifically through the `connect_segments()` method. This method delegates geometric alignment of segment connection points to the `Connect` class and network graph updates to the `NetworkUpdater` class (see figure 3.7. The `Connect` class enforces compatibility and alignment through:

- `is_lane_compatible()`: checks lane count and lane width consistency.
- `_connect_segments()`: rotates and translates the moving segment so its connection point aligns with the fixed one.

The geometric alignment between two segments can be described as follows. $\mathbf{t}_f = (t_{fx}, t_{fy})$ and $\mathbf{t}_m = (t_{mx}, t_{my})$ denote the tangent direction vectors of the fixed and moving segment connection points. $\mathbf{p}_f$ and $\mathbf{p}_m$ are the connection points expressed in the global coordinate frame (see Figure 3.8).

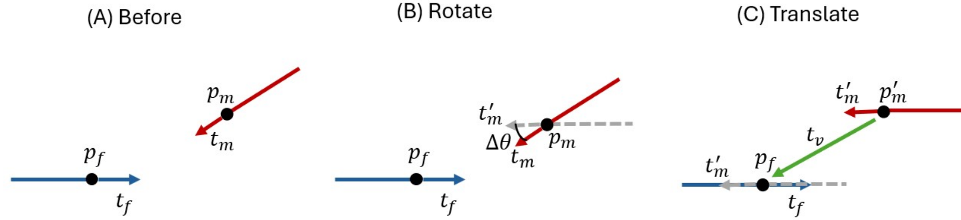

Figure 3.8: Straight-segment connection ($\mathbf{o}_m$ at p_m): (A) before alignment with endpoints p_f, p_m and tangents vectors $\mathbf{t}_f, \mathbf{t}_m$ (both shown as arrows at the connecting connection points), (B) rotation by $\Delta\theta$ so $\mathbf{t}'_m$ is anti-parallel to $\mathbf{t}_f$, (C) translation by $\mathbf{t}_v$ so endpoints coincide, yielding the aligned configuration.

The angles can be computed with the following formulas:

$$\theta_f = \text{atan2}(t_{fy}, t_{fx}), \quad \theta_m = \text{atan2}(t_{my}, t_{mx}) \tag{3.1}$$

The required rotation is:

$$\Delta\theta = \theta_f - \theta_m + \pi \tag{3.2}$$

with rotation matrix:

$$R(\Delta\theta) = \begin{bmatrix} cos\Delta\theta & -sin\Delta\theta \\ sin\Delta\theta & cos\Delta\theta \end{bmatrix} \tag{3.3}$$

After applying the rotation, the point becomes:

$$\mathbf{p}'_m = \mathbf{o}_m + R(\Delta\theta)(\mathbf{p}_m - \mathbf{o}_m), \tag{3.4}$$

where, $\mathbf{o}_m$ is the local origin and the translation vector is defined as: $\mathbf{t}_v = \mathbf{p}_f - \mathbf{p}'_m$. Finally, the resulting transformation that aligns the moving segment with the fixed one can be expressed as:

$$\mathbf{x}' = R(\Delta\theta)\mathbf{x} + \mathbf{t}_v \tag{3.5}$$

where, $\mathbf{x}$ denotes any position vector belonging to the moving segment (for

example, a connection point). The transformed point x' is the corresponding global position after rotation and translation. Equation (3.5) generalizes the alignment step to the entire geometry of the moving segment, not just the specific connection point.

3.2.2.4 Network Updates

After the alignment is complete, the road network and graph are updated. The `NetworkUpdater` refreshes the connection point positions through the `refresh_connection_points()` method, which in turn invokes `segment.update_connection_points`. The `_handle_connection()` updates the network graph for proper visualization.

3.2.2.5 Lane Boundaries

To determine the lane boundaries, each segment computes their lane boundaries based on its number of lanes and lane width. Since each segment type has its own class, it calculates its lane boundaries according to its geometry. This ensures that vehicle positioning, lane tracking, and trajectory generation across all segment types are consistent, while keeping the logic encapsulated within each road own class.

Additionally, lane directions have not been defined, which allows vehicles to have freedom of movement. This supports bidirectional road behavior and can be used to test collision avoidance behavior when vehicles drive in opposite directions (see Section 6.2).

3.2.2.6 Virtual Parking Lot and Re-entry

To ensure that road networks with unconnected ends are usable, the `RoadSegmentManager` creates and maintains a `VirtualParkingLot`. The parking lot is not a physical segment, but an abstraction for managing vehicles that leave the network through open connection points. When the network is constructed, the manager creates a parking lot instance via `create_virtual_parking_lot()`. This method initializes:

- A `VirtualParkingLot` object that defines the platoon size, departure distribution, and each vehicle's timing interval (via `ProbabilityDistribution` and `TimeSequence`) (see Section 4.2.7).

- A set of `VirtualEntryPoints`, created through the `assign_-virtual_entry_points()` method, which are automatically bound to all open ends in the road network, where the open ends are found by `find_open_end_segments()`.
- A configurable list of exit points (`exit_points`, which represents where vehicles re-enter the network after departure. If no exit points are provided, all entry points allow re-entry.

The `VirtualParkingLot` is responsible for grouping the incoming vehicles into platoons and scheduling their re-entry times, while the `RoadSegmentManager` ensures that the parking lot is correctly connected to the road network. The manager also advances the parking lot's internal time via `advance_time()` to keep track of the time globally.

3.2.2.7 Graph-Based Road Network Representation

The simulator maintains a graph-based representation of the road network inside the `RoadSegmentManager`. This graph is implemented as a MultiDiGraph from the NetworkX library [25]. NetworkX provides multiple alternatives for representing a network, such as `Graph`, `DiGraph`, and `MultiDiGraph`. However, for this simulator `MultiDiGraph` is most suitable, because it allows multiple distinct directed edges between the same pair of nodes. In this representation:

- Nodes correspond to the segment connection points and store their global positions.
- Edges represent road segments with its associated metadata.

This graph implementation is used to verify that connections have been established correctly. It also provides the names of connection points, which simplifies the configuration process easier by allowing them to be visualized as the simulation is started.

3.2.2.8 Automatic Road Generation of Open Ends

The simulator supports automatic road generation to connect pairs of open connection points. This feature is useful for maintaining network continuity (forming loops in the network), especially since manually aligning every segment would be impractical. The functionality is implemented in the `create_connection()` method of the `RoadSegmentManager`. The

method constructs a drivable connecting path between two connection points by instantiating a sequence of curve–straight–curve CSC road segments, following the principles of a Dubins path. This approach is inspired by the Dubins path, which is a known method for shortest-path planning for vehicles with a minimum radius.

There are four possible CSC configurations associated with Dubins path:

- Left-Straight-Left (LSL)
- Right-Straight-Right (RSR)
- Left-Straight-Right (LSR)
- Right-Straight-Left (RSL)

In addition to these configurations, this method accounts for degenerate cases where one or both arcs have a central angle that is close to zero, when the radius r is a large enough value and when connections align:

- If one arc's central angle approaches zero, the CSC reduces to a Curve-Straight (CS) or Straight-Curve (SC) connection.
- If both arcs central angles approaches zero, the construction reduces to a pure straight line, since the connection points are already aligned.
- If the radius is exactly large enough for the entry and exit arcs to coincide, the construction is reduced to a single Curve (C) connection.
- If the two connecting points are already aligned, the construction is reduced to a single Straight (S) line connection.

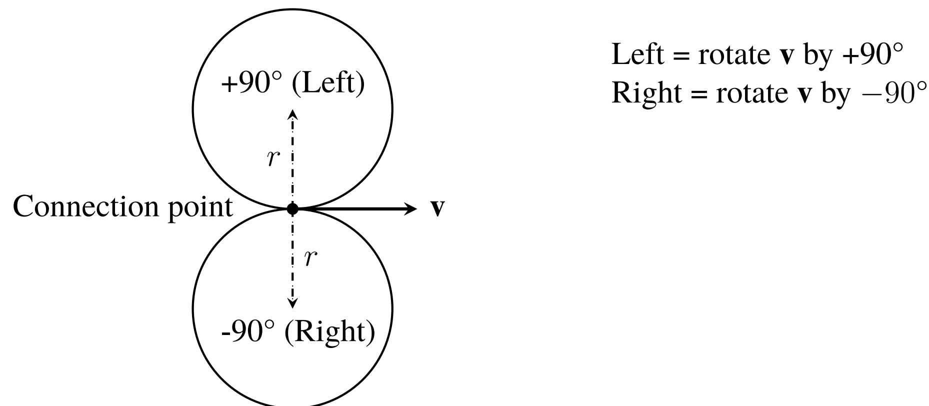

Figure 3.9: Schematic definition of left/right turning circles relative to a heading **v**. The circle centers are offset by a distance r along the directions obtained by rotating the heading by $\pm 90°$.

The algorithm for this method can be divided into three main stages:

1. **Turning circles**: Given the global positions of the connection points and their tangent headings, circles of radius r (user-specified, see Section 4.2.3) are placed at lateral offsets relative to the heading vector. The left circle is obtained by rotating the heading vector by +90° (counter-clockwise), while the right circle is obtained by rotating it by –90° (clockwise). Figure 3.9 illustrates this construction.

2. **Tangent computation**: The external or internal tangent lines between the two circles are computed, which gives the possible straight segments that link the arcs. If the two circles coincide, the tangent reduces to a shared arc, which produces a single curve connection.

3. **Feasibility check**: The tangent points define their sweep/central angles of the entry and exit arcs. Each possible CSC configuration is evaluated to ensure that both arcs are feasible and the resulting path is drivable. If not a degenerate configuration such as SC, CS, C or S cases are selected.

If a valid configuration exists, the method instantiates the corresponding CSC (or a deconstructed version of this) sequence with road segments then connects them to the fixed network using `connect_road_segments()`. The process ensures both geometric alignment and tangent continuity at every connection.

This process can be formalized mathematically as follows:

Given two connection points with poses $p_s, p_e \in R^2$ and a minimum radius $r > 0$, a drivable connection of roads can be constructed. It contains at most two curved segments and one straight segment. The outbound unit heading are written as v_s, v_e and the $90°$ Counter-Clockwise (CCW) rotation matrix as $J = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$.

The turn directions are denoted as $s \in \{+1, -1\}$, where $s = +1$ is for *left*-turning curves CCW and $s = -1$ is for *right*-turning curves (Clockwise (CW)). For a possible start turn s_1 and end turn s_2, the turning-circle centers of radius r are:

$$C_1 = p_s + s_1\, r\, Jv_s \quad \text{(outbound at } p_s\text{)}, \quad C_2 = p_e + s_2\, r\, Jv_e \quad \text{(outbound at } p_e\text{)}. \tag{3.6}$$

The derived quantities are defined as:

$$\Delta C = C_2 - C_1, \qquad D = \|\Delta C\| \qquad \hat{u} = \frac{\Delta C}{D}, \qquad \hat{u}_\perp = J\hat{u} \tag{3.7}$$

where, $\hat{u}$ is the unit vector from C_1 to C_2 and $\hat{u}_\perp$ its CCW perpendicular. The geometry of the turning circles, including ΔC and $\hat{u}_\perp$, is illustrated in Figure 3.10.

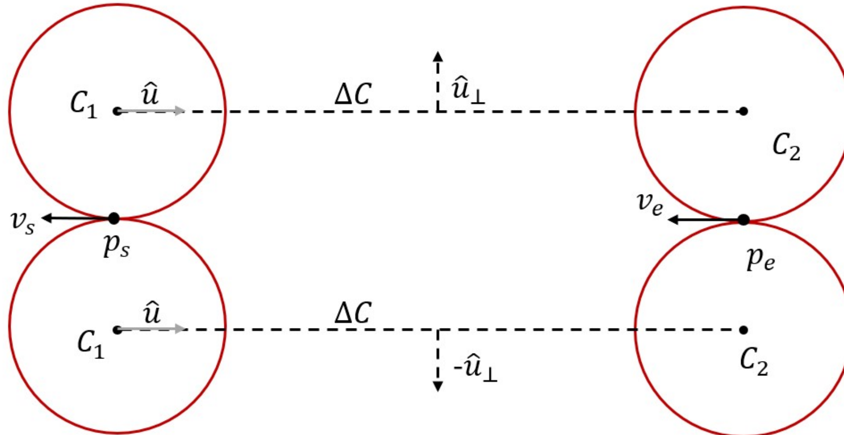

Figure 3.10: Turning circle center for start point p_s with heading v_s and end point p_e with heading v_e. At each position there is two possible circle centers exits, corresponding to left (s=+1) and right (s=-1) turns. The displacement vector $\Delta C = C_2 - C_1$ and its perpendicular $\hat{u}_\perp$ are shown for one configuration.

These defined variables and quantities can then be used to construct the

tangent lines between the circles. However, there are two possible tangents depending on if it is an external tangent or an internal tangent. **External tangents** ($s_1 = s_2$), corresponding to LSL or RSR, occur when both circles have the same orientation. The tangent line is then parallel to ΔC. One branch is

$$T_1 = C_1 + s_1\, r\, \hat{u}_\perp, \qquad\qquad T_2 = C_2 + s_2\, r\, \hat{u}_\perp, \tag{3.8}$$

and the other branch is obtained by replacing $\hat{u}_\perp$ with $-\hat{u}_\perp$. This construction is shown in Figure 3.11, which illustrates the two possible tangents. The straight segment of the CSC path is simply the line between T_1 and T_2, whose length reduces to $L = \|T_2 - T_1\| = D$.

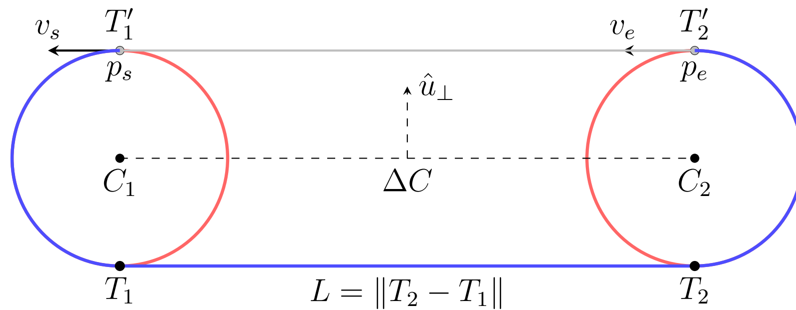

Figure 3.11: External tangent construction LSL (shown for $s_1 = s_2 = +1$). Two equal-radius turning circles with the same orientation with two external tangents. The bottom branch (thick blue) is chosen, and the alternate (gray) is discarded.

If $D > 0$, all external tangents exist, however, when $D \approx 0$ (the circle centers coincide), the construction degenerates to a single circular arc (C) configuration about that center.

Internal tangents ($s_1 = -s_2$), corresponding to LSR or RSL, occur when the circles have opposite orientations. Internal tangents exist only if

$$D \geq 2r. \tag{3.9}$$

Let, $\alpha = \text{atan2}(\hat{u}_y, \hat{u}_x)$ be the heading of $\hat{u}$, and

$$\delta = \arccos\!\left(\frac{2r}{D}\right) \in [0, \pi/2]. \tag{3.10}$$

where u_x, u_y is the directional components of $\hat{u}$ and δ is the angle between $\hat{u}$ and the tangent direction $\hat{t}$. There are two symmetric branches. One branch is

given by

$$\theta = \alpha + s_1\,\delta, \quad \hat{t} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}, \quad T_1 = C_1 + r\,\hat{t}, \quad T_2 = C_2 - r\,\hat{t}. \quad (3.11)$$

The other is obtained by $\theta = \alpha - s_1\,\delta$, which is equivalent to replacing $\hat{t}$ by $-\hat{t}$. This construction is illustrated in Figure 3.12, which shows the two possible internal tangents (blue and gray) between the turning circles. The chosen branch corresponds to $\theta = \alpha + s_1\delta$, while the other branch corresponds to $\theta = \alpha - s_1\delta$.

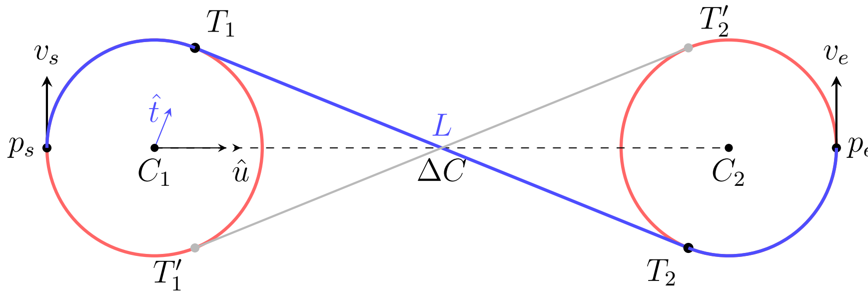

Figure 3.12: Internal tangent construction RSL (shown for $s_1 = -1$, $s_2 = 1$). Two internal tangents arise from $\theta = \alpha \pm s_1\delta$ with $\delta = \arccos(2r/D)$. The chosen branch (blue) uses $\hat{t} = (\cos\theta_A, \sin\theta_A)$, which gives $T_1 = C_1 + r\hat{t}$ and $T_2 = C_2 - r\hat{t}$. The alternate branch (gray) uses θ_B. The straight segment length is $L = \|T_2 - T_1\| = \sqrt{D^2 - (2r)^2}$.

The corresponding straight segment length is

$$L = \|T_2 - T_1\| = \sqrt{D^2 - (2r)^2}. \quad (3.12)$$

Each case (external/internal) has two tangent branches. The algorithm must select the branch that ensures forward tangency at the both tangent points T_1 and T_2. The unit tangent of a circle at P for turn s is defined by

$$\hat{\tau}(C, P, s) = \frac{s\,J(P - C)}{\|P - C\|}. \quad (3.13)$$

Let $\hat{s} = \frac{T_2 - T_1}{L}$ be the straight direction. If

$$\left\langle \hat{\tau}(C_1, T_1, s_1), \hat{s} \right\rangle > 0 \qquad \left\langle \hat{\tau}(C_2, T_2, s_2), \hat{s} \right\rangle > 0 \quad (3.14)$$

If either condition fails, the algorithm switches to the other tangent branch (by

replacing $\hat{u}_\perp$ with $-\hat{u}_\perp$ for external, or $\theta = \alpha - s_1\delta$ for the internal case) and recomputes T_1, T_2. To measure the arc length along a circle, the signed sweep angle has to be defined. Given a circle with center C, the signed sweep angle from a point P_{in} to a point P_{out} on the circle, with turn direction $s \in \{+1, -1\}$, is computed as

$$\theta(C, P_{\text{in}}, P_{\text{out}}, s) = \text{atan2}\big(\det(u_{\text{in}}, u_{\text{out}}),\ \langle u_{\text{in}}, u_{\text{out}}\rangle\big), \tag{3.15}$$

$$\text{adjusted so that} \begin{cases} \theta \geq 0, & s = +1 \text{ (left)} \\ \theta \leq 0, & s = -1 \text{ (right)} \end{cases} \tag{3.16}$$

Here

$$u_{\text{in}} = \frac{P_{\text{in}} - C}{\|P_{\text{in}} - C\|}, \qquad u_{\text{out}} = \frac{P_{\text{out}} - C}{\|P_{\text{out}} - C\|}, \tag{3.17}$$

are the normalized spokes from the circle center to the two points. The determinant is defined as $\det(u_A, u_B) = u_{Ax}u_{By} - u_{Ay}u_{Bx}$, and $\langle \cdot, \cdot \rangle$ is the dot product. This angle is in $(-\pi, \pi]$ and is then adjusted to enforce consistency with the chosen turn direction.

$$\theta > 0 \quad \text{for left turns } (s = +1), \quad \theta < 0 \quad \text{for right turns } (s = -1). \tag{3.18}$$

For a CSC path the two arc sweeps are then

$$\theta_1 = \theta(C_1, p_s, T_1, s_1) \qquad \theta_2 = \theta(C_2, T_2, p_e, s_2), \tag{3.19}$$

corresponding to the entry arc (from the start point to the first tangent point) and the exit arc (from the second tangent point to the end). The total length of the path is therefore

$$L_{\text{CSC}} = r\,|\theta_1| \ + \ L \ + \ r\,|\theta_2|. \tag{3.20}$$

The construction naturally reduces to simpler forms:

1. If $D \approx 0$ and $s_1 = s_2$, the path is a single circular arc: $L = r|\theta(C_1, p_s, p_e, s_1)|$.
2. If $D \geq 2r$ but one sweep $|\theta_i| \approx 0$, the CSC reduces to CS or SC.
3. If both $|\theta_i| \approx 0$, the solution is S connection.

Lastly, the four Dubins CSC types are evaluated

$$\text{LSL}: (s_1 = +1, s_2 = +1), \quad \text{RSR}: (-1, -1), \text{LSR}: (+1, -1), \quad \text{RSL}: (-1, +1),$$

The algorithm proceeds as follows:

1. Compute tangent points (T_1, T_2) from equation (3.8) or (3.11).
2. Apply the branch test using equation (3.15) to ensure forward tangency.
3. Evaluate arc sweeps using equation (3.15) - (3.19).
4. Compute the total length using equation (3.20).

The method returns the minimizer

$$(s_1^*, s_2^*) = \arg \min_{\text{LSL,RSR,LSR,RSL}} L_{\text{CSC}},$$

and instantiates the corresponding sequence of road segments CSC, or a degenerate reduction (C, CS, SC, or S) when appropriate.

3.2.3 VehicleManager

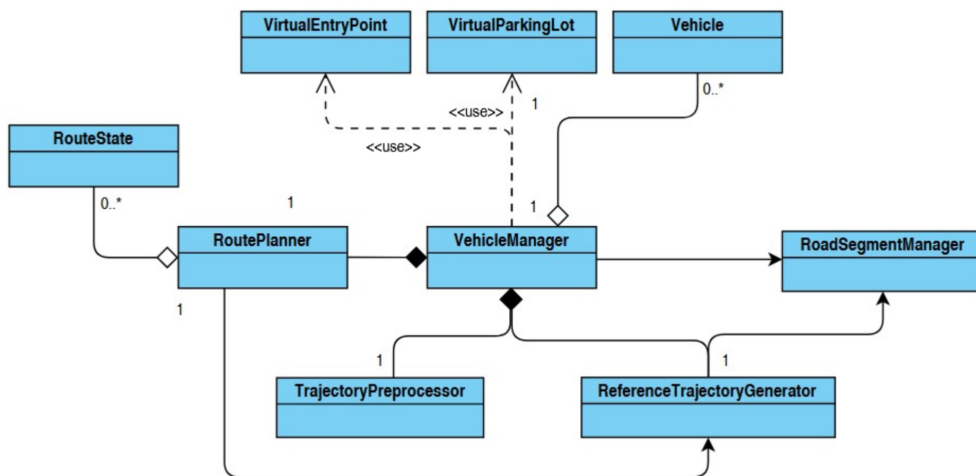

Figure 3.13: UML showing the relationship of the `VehicleManager` and its associate components.

The simulator's vehicle system is centered around the `VehicleManager`, which manages the life cycle and coordination of all vehicles in the

environment. It maintains a registry of all vehicles in `self.vehicles`, and together with its associated components is responsible for creating, placing, updating, and tracking each vehicle throughout the simulation's runtime. The `VehicleManager` holds a reference to the `RoadSegmentManager` in its attributes, as well as `ReferenceTrajectoryGenerator`, `RoutePlanner`, and `TrajectoryPreprocessor` (as shown in Figure 3.13).

3.2.3.1 Responsibilities and Orchestration

The `VehicleManager` coordinates the lifecycle and step by step execution of vehicles. Its role is orchestration, which means calling modules and passing the required inputs, collecting the outputs and updating the vehicle state through the dynamics. During initialization, vehicles are created and registered with concrete modules, (dynamics, lateral and longitudinal controllers and a sensor model), associated with the route planner, and assigned route instructions from configuration. The manager then coordinates module calls for trajectory preprocessing, control, and dynamics (the detailed interfaces for the controllers and dynamics are specified in Section 3.3.3.1-3.3.3.3).

Vehicles are created through the `create_vehicle()` method, which receives the concrete modules (dynamics, lateral and longitudinal controllers, sensor model) as input. Each simulation step is handled by the `update_vehicles()` method, which iterates over active vehicles in deterministic order and performs three actions:

1. Computes a trimmed forward reference trajectory in the `TrajectoryPreprocessor`. This involves locating the closest waypoint on the vehicle's trajectory relative to its global position, discarding all preceding points and retaining only the forward-looking slice of the trajectory within a configurable look-ahead horizon (an array of (x, y) waypoints).

2. Calls the vehicle's `update()` method with the current time, timestep, the registry of vehicles, the trimmed reference trajectory, the route state and a reference to `RoadSegmentManager`. The vehicle's trajectory is always defined while it is active in the network, since it is generated during initialization and updated accordingly. If a vehicle leaves the network (e.g., entering the `VirtualParkingLot` (see Section 4.2.7)), it temporarily lacks a trajectory until it is reintroduced with a new route.

The reference to the `RoadSegmentManager` provides each vehicle with access to the global road network during updates. Although vehicles do not directly query this manager, the reference is passed into the `PerceptionService` (see Section 3.7) which relies on it to resolve lane identifiers, map neighbors on rotated or connected segments and query speed limits. This ensures that perception outputs (e.g., detected vehicles ahead in the same logical lane) are stable and consistent with the overall network representation.

3. Performs route state tracking and segment logic. Segment logic includes lane tracking while the vehicle remains inside a segment or in an overlap region, transition to a connected segment when the vehicle exits the current one, and exit to the `VirtualParkingLot` when reaching an open connection. The route planner is notified on transitions so that movement instructions such as left/right/straight and left/right_turn at intersections are resolved and future references can be regenerated.

When a transition is detected, the manager determines the receiving segment and entry orientation, queries the segment for the appropriate lane and entry point, converts the vehicle's current global pose into the segment's local frame, and updates the dynamics and relevant information that each vehicle stores (segment, lane, positions, orientation). The route planner is then updated (with regeneration enabled) to reflect the new segment context. Initial placement of vehicles follows the same structure, where lane and progress are computed from the entry information, local and global positions are derived, containment in the chosen lane is verified, dynamics are initialized, and a reference trajectory is generated for the controllers.

3.2.3.2 Vehicle Data Model

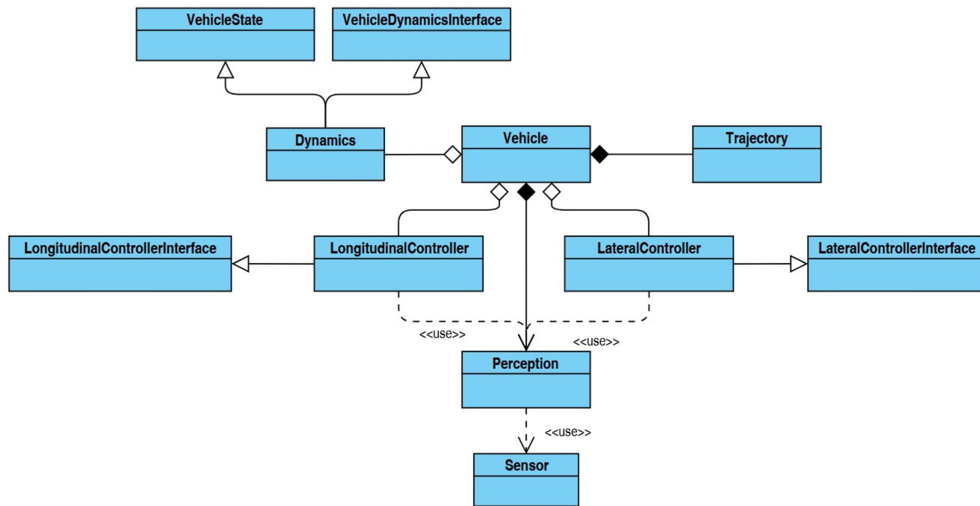

Figure 3.14: UML of the overall `Vehicle` architecture.

The `Vehicle` class represents a single vehicle and encapsulates the data and module references needed for simulation. Each instance stores the physical state in global coordinates and holds references to a dynamics model, a lateral controller, a longitudinal controller, a perception module and a sensor model. The dynamics model is a concrete object (e.g., bicycle or unicycle) that integrates control commands into updated state variables such as position, velocity and heading.

The sensor model abstracts on-board sensing as a configurable range and Field-of-view (FOV) detector, placed at the vehicle's center. Sensors are parametrized by maximum range and angular field of view, and optional delay and noise. Default values are provided, but the parameters can be changed via the `Sensor` class constructor. The current implementation provides a single general purpose sensor model.

The perception module processes the raw sensor output into structured `PerceptionData`. This includes a list of detected vehicles within the sensor's range and FOV, with attributes such as relative position, velocity, distance and lane association. This information can then be used by the controllers, for example, to identify a leader vehicle in the same logical lane.

Controllers compute steering and longitudinal commands from the vehicle's current state and its trimmed reference trajectory, while the dynamics model integrates these commands to update the state. The interaction between

these components is illustrated in Figure 3.14, and the detailed controller and dynamics are specified in Sections 3.3.3.1-3.3.3.3.

The class is concrete and generic, where different vehicle types are obtained by configuring different models at construction (for example, bicycle or unicycle dynamics, PID for lateral control, and ACC for longitudinal control). During each simulation step the `update()` method obtains a sensor reading and assembles a perception state. Depending on the configured controller, this may include identifying a leader vehicle and forming a desired speed consistent with speed limits and leader constraints. The longitudinal controller then computes an acceleration or braking command, and the lateral controller computes a steering command from the current state, trimmed reference trajectory and perception (if needed). Finally, the dynamics model integrates these commands to update the vehicle's position, heading and speed.

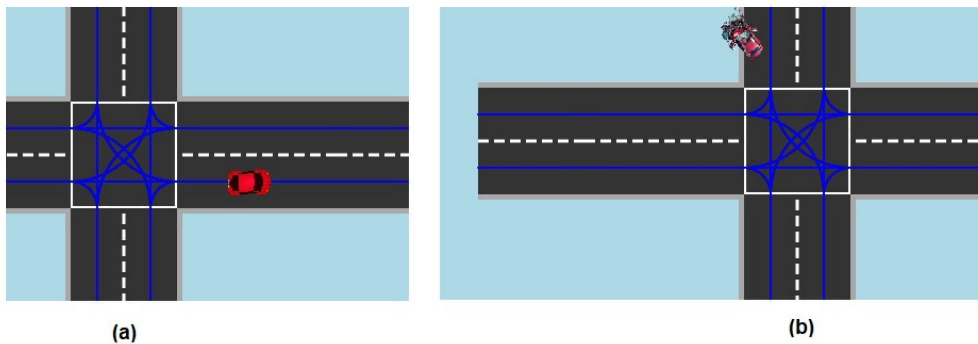

Figure 3.15: A vehicle driving as usual in (a), and a vehicle that has crashed in due to high velocity in (b).

The vehicle class also utilizes helper methods, where one of the helper methods computes the orientation (`get_vehicle_orientation()`) from a direction vector with optional fixed offset (set during the configuration process). Another helper method computes the direction (`get_vehicle_direction()`) that returns a unit direction vector in global coordinates. For straight segments the vector follows the segment orientation, however for curved segments it uses the tangent at the current position. The vector is flipped if needed so that the driving direction convention is respected (POSITIVE points into the right half-plane, NEGATIVE into the left half-plane. see Section 4.2.6). There is also a visualization helper method used for debugging (`draw()`), it uses Pygame to render the vehicle and current reference. Furthermore, as the vehicle drives along roads its status is `"active"`, however, when it crashes (handled by the `VehicleManager`)

the status switches to "crashed" which is detected and visualized by the helper (See Figure 3.15).

3.2.3.3 VirtualParkingLot and VirtualEntryPoint

The VirtualParkingLot is a dedicated class that acts as a temporary holding area for vehicles that exit the road network at open (unconnected) connection points. Vehicles that are transferred into the parking lot are grouped into platoons, and when the platoon becomes full, the virtual parking lot schedules the platoon with a departure time and interval departure times for its vehicles. The ProbabilityDistribution provides a relative platoon departure time, and a TimeSequence provides an interval time to each vehicle within the platoon. The lot tracks the simulation time via advance_time(), a list of active platoons, and an optional list of re-entry points (exit_points) (the exit_points are configurable, discussed in Section 4.2.7). Each platoon also stores its departure_time, and a map of every vehicles' departure time in the platoon (vehicle_departure_times).

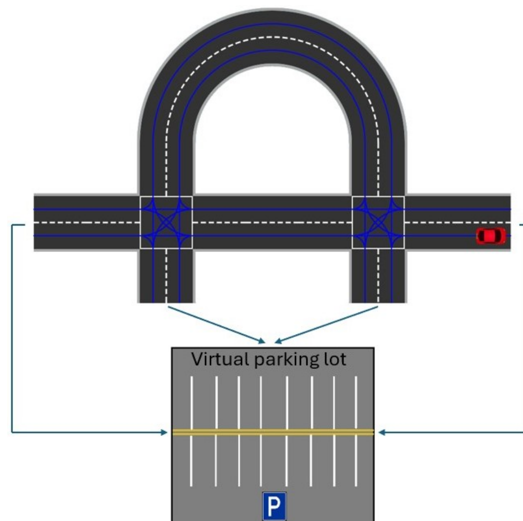

Figure 3.16: The vehicle before entering the virtual parking lot from a unconnected connection point.

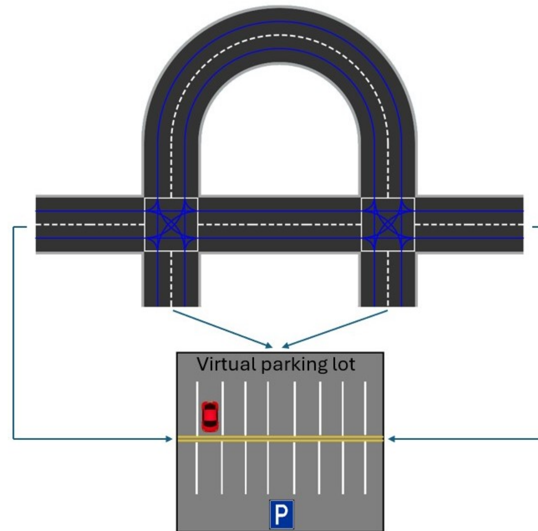

Figure 3.17: The vehicle after exiting the road network and entering the virtual parking lot.

The `VirtualEntryPoint` is basically what binds the virtual parking lot to the road network at the open ended segments connection points. When the `VehicleManager` detects that a vehicle has reached a segment end whose connection state is `CONNECTABLE` (see Section 3.2.2.2), the vehicle's status is switched from `"active"` to `"parked"` and it is transferred to the parking lot by calling the `transition_to_virtual_parking_lot()` in the `VirtualEntryPoint`, which in turn invokes `add_vehicle_to_platoon()` in the `VirtualParkingLot`. The transfer detaches the vehicle from the road state and places it into a platoon (see Figure 3.16 and 3.17).

At every simulation step, the `VehicleManager` calls the `dispatch_ready_platoons()` method, which queries `get_ready_platoons()` in the `VirtualParkingLot` to find platoons whose scheduled `departure_time` has elapsed. When a platoon finally reaches its departure time the `VehicleManager` coordinates the re-entry, and dispatches vehicles one after another as their individual departure times are reached.

On dispatch, the manager does the following:

1. changes the vehicle's status from `"parked"` to `"active`,

2. randomly selects a configured entry point and assigns it to all vehicles in the platoon,

3. randomly selects a lane and assigns it to the vehicle in the platoon and computes the driving direction from the connection (`segment.compute_driving_direction()`),
4. initializes progress along the lane (set to zero),
5. updates the dynamics state (segment, lane, positions, orientation),
6. resets and populates the route planner,
7. generates a new reference trajectory from the new pose and lane context.

When vehicles re-enter the road network they are given a simple `["straight"]` instruction and the route state index is reset to zero to ensure consistent downstream lane and intersection decisions and enabling a theoretically infinite looping simulation. Also, even though each vehicle is given the same entry point, they might not get the same lane, since it is randomized.

Platoons have a fixed `capacity` (configured by the user) and can accumulate vehicles until it is full (see Section 4.2.7). All platoons have the same configured capacity and allow one to many vehicles in the platoon. When a vehicle is added to a platoon, the vehicle clears road-related state on the vehicle (`reset_road_information`) so it no longer references any segment or lane while parked. When the capacity is reached, the platoon records its `fill_time`, is marked `closed`, and subsequently receives its `departure_time` together with each vehicles `vehicle_departure_times` generated from the configured distribution and time sequence.

3.3 Vehicle Subsystems

This section describes the components that run inside each simulated vehicle and collectively produce its behavior on the road. While the `VehicleManager` schedules updates and coordinates vehicles in the environment, each vehicle encapsulates its own subsystems for sensing (perception), decision making (lateral and longitudinal control) and motion (state propagation and trajectory handling).

3.3.1 Reference Trajectory System

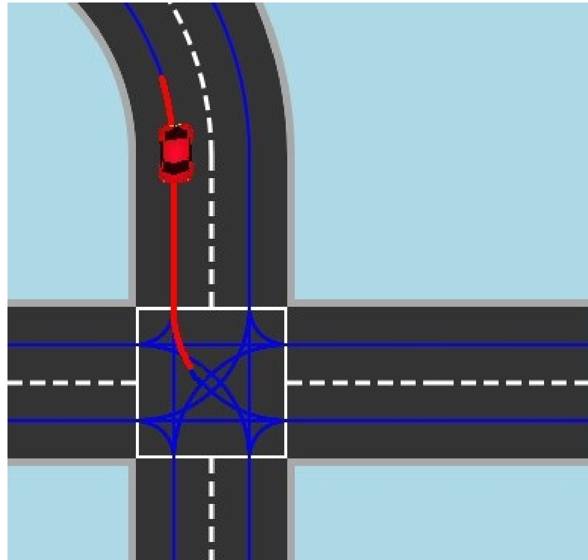

Figure 3.18: Generated reference trajectory from the vehicle visualized as a red line.

Each vehicle in the simulator follows a reference trajectory, which is an array of 2-D waypoints (x, y) in global coordinates that represents the intended route through the road network (see Figure 3.18). The trajectory is regenerated on demand over a fixed look-ahead horizon and at runtime it is trimmed so that only a portion ahead of the vehicle's current position is retained.

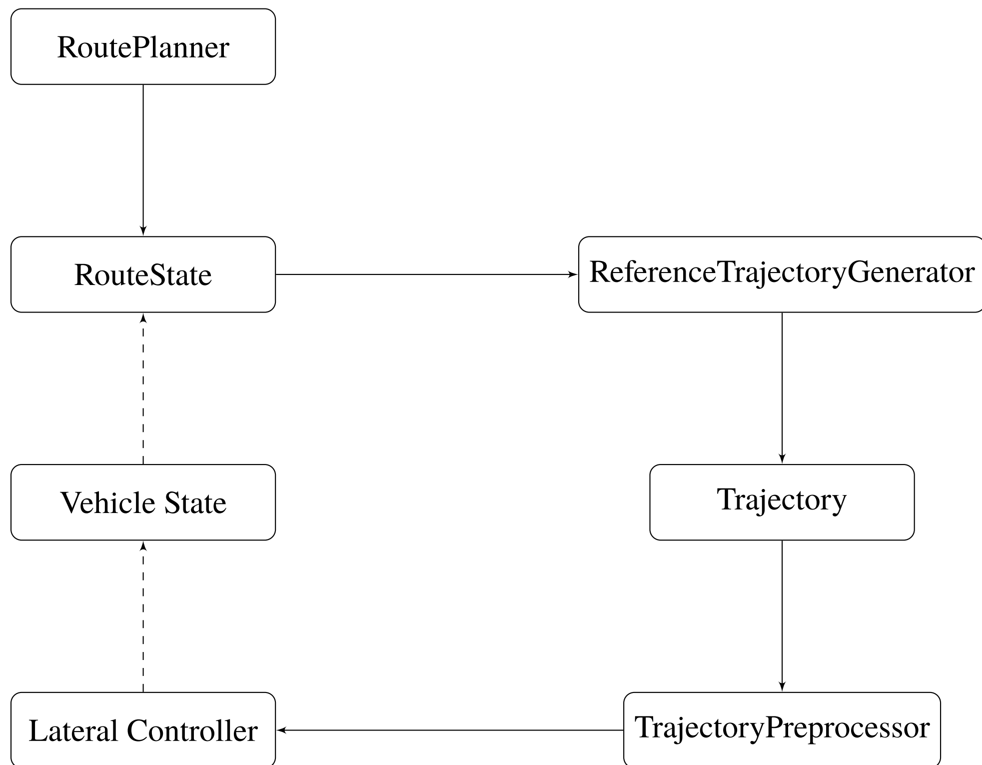

Figure 3.19: Information flow in the reference trajectory generation and route planning system.

The generation process is coordinated by five modules (see Figure 3.19.):

- **`RoutePlanner`**: Maintains a `RouteState` for each vehicle. A route is expressed as a sequence of movement instructions (e.g., `"straight"`, `"left"`, `"right_turn"`, lane changes) that specify how the vehicle should traverse the road network. The planner advances the active instruction by incrementing the route index whenever the vehicle transitions to a new segment. If the remaining look-ahead distance in the trajectory falls below a velocity-based threshold, the trajectory is regenerated using the current instruction, but the route index is not advanced.

- **`RouteState`**: Tracks and maintains all navigation related contexts for a single vehicle. It stores instructions (`route_instructions`), the current instruction index (`instruction_index`), the most recent entry connection into the current segment (`last_entry_connection`) and the most recent intersection entry direction (`last_intersection_entry_point`). It also manages staged

lane changes (pending_lane_change), records which lane change instructions have been executed per segment (executed_lane_-change_instruction) and logs which movement was used when eaveing each segment (entry_movements). Additionally, it keeps the last entry point used when the vehicle was placed or re-entered the network (last_entry_point). Based on the remaining trajectory length and vehicle speed, the RouteState determines when reference trajectory regeneration is required.

- **ReferenceTrajectoryGenerator**: Constructs a forward trajectory over the look-ahead horizon, using the vehicle state and route state. It returns a list of entries, each entry as [command, segment_id, lane_id, points], where command is the applied route instruction (e.g., "straight", "left", "2_right"), segment_id is the identifier of the segment providing the path, lane_id is the selected lane within that segment, points $(x_k, y_k)_{k=0}^{N}$ is the geometric path which is represented as an ordered list of 2-D waypoints in global coordinates.
- **Trajectory**: Stores the generator's output and performs a flattening step, which produces a continuous waypoint list (current_-trajectory) suitable for control.
- **TrajectoryPreprocessor**: Trims the flattened trajectory at runtime, and discards outdated waypoints behind the vehicle and keeps only the forward-looking slice.

The construction of a trajectory follows this structured process.

1. **Instruction parsing**: Route instructions are provided as strings such as straight, left, right, left_turn, or right_turn. Compounded instructions are also possible, for example 2_left, however these are split into a base instruction and a lane change count.
2. **Relative left/right resolution**: Lane change instructions are given from the vehicle's perspective. Since lane indices are ordered relative to a segment's local orientation (Section 3.5), a vehicle entering from the end connection must swap left and right, while entry from the start keeps them unchanged.
3. **Instruction normalization**: Instructions are adapted to the segment type: intersection specific commands such as "left_turn" or "right_turn" fall back to "straight" on non-intersection

segments, while lane-change instructions such as "left" or "right" fall back to straight inside intersections.

4. **Lane target and staging**: Valid lane changes are staged in pending_lane_change and applied exactly once while the vehicle remains on the segment. To prevent repetition, executed lane changes are saved in executed_lane_change_instruction.

5. **Connection selection**: For intersections, get_exit_connection_point() maps the entry direction and movement to the correct exit, avoiding backtracking. For non-intersections, connections are scanned and validated using is_valid_connection_to().

6. **Lane-centerline retrieval**: Each segment provides its lane path through get_trajectory(), with intersections requiring the chosen movement and logical direction.

7. **Direction alignment and trimming**: If the vehicle entered via the end connection, or if a heading–tangent check indicates misalignment, the trajectory is reversed. On the first segment, the trajectory is also trimmed so that only points ahead of the vehicle's current position are included.

8. **Horizon accumulation**: Trajectories are concatenated until the look-ahead horizon is satisfied. If the final step overshoots the horizon, the endpoint is interpolated to match the desired distance exactly.

9. **Record keeping**: For each visited segment, a record [command, segment_id, lane_id, points] is created. The next entry connection is tracked, and the working segment/lane is updated using the connection's lane mapping.

At this stage, the generator has produced a list of trajectory entries, each containing the command, segment identifier, lane identifier, and sequence of waypoints for that lane. While this representation is useful for preserving the full information and history of the route, it cannot be used directly by the controller. To provide a usable path, the simulator applies two key transformations:

- **Flattened trajectory**: The Trajectory class removes meta data such as command, segment id, lane id, while concatenating the waypoint sequences from successive segments into a single ordered sequence

of waypoints, called `current_trajectory`. This produces a continuous array of waypoints across the network that the controller can follow directly.

- **Trimmed trajectory**: At runtime, the `TrajectoryPreprocessor` discards waypoints behind the vehicle, to keep only the forward-looking slice from the vehicle's closest waypoint. Regeneration is triggered if the remaining trimmed path falls below a velocity-scaled threshold or after a segment transition.

The separation of responsibilities provides several advantages. The `Trajectory` class stores and maintains the full planned horizon as output by the generator which can be safely extended with additional metadata, and used to plot or save the history of that data (see Section 4.2.8), while the preprocessor ensures the controller always receives a concise, up-to-date array of trajectory points aligned with the vehicle's actual position. This modular design reduces controller complexity, since controllers operate on a uniform list of trajectory points without handling segment boundaries, trimming logic, or regeneration policies.

The resulting forward trajectory is used directly by the lateral controller (Section 3.3.3.1).

3.3.2 Sensor and Perception

Perception is organized as a pipeline that constructs a perception object with sensing information, which is then used by the longitudinal controller. Each `Vehicle` is equipped with an idealized `Sensor` that models finite reach (range and FOV) without delay or noise to enable evaluation under perfect perception (See Section 3.6).

At each simulation step, the perception object is assembled by a dedicated `PerceptionService` using the `build()` method, which decouples sensing from control and keeps the `Vehicle` class `update()` method clean. The service executes the following steps:

1. **Sensing** (`Sensor`): scans all vehicles' current positions and velocities, excluding the ego and those with status = `parked`. Only vehicles within the configured range and FOV are kept. The sensor measurements themselves (relative positions and velocities) are expressed in the ego's local frame (Section 3.5 for frame conventions) while global state information (e.g., absolute position, road segment ID) is appended as metadata, but not used by the controllers).

2. **Normalization**: convert raw detections to a data class object `DetectedVehicle` with global pose/velocity, local (ego-frame) relative position/velocity, absolute distance, and lane/segment identifiers. This is then assembled into a larger data class called `PerceptionData` which exposes controller friendly package.

The output is a stable data-transfer object `PerceptionData` that controllers read each step. This design allows adding new signals without changing controller interfaces (Section 3.3.3).

The local relative position is computed by rotating the global displacement from the detected vehicle into the ego frame:

$$\mathbf{r}_{\text{rel}}^{\text{local}} = R(-\theta_{\text{ego}}) \cdot (\mathbf{p}_{\text{det}} - \mathbf{p}_{\text{ego}}), \quad R(-\theta_{\text{ego}}) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, \tag{3.21}$$

where, $\mathbf{p_{det}}$,$\mathbf{p_{ego}}$ $\in \mathrm{R}^2$ are the global positions and θ_{ego} is the ego heading. The local relative velocity is computed in a similar way:

$$\mathbf{v}_{\text{rel}}^{\text{local}} = R(-\theta_{\text{ego}}) \cdot (\mathbf{v}_{\text{detected}} - \mathbf{v}_{\text{ego}}) \tag{3.22}$$

where, $\mathbf{v_{det}}$, $\mathbf{v_{ego}}$ are the global velocities. In addition to these local quantities, each detection includes sufficient metadata for downstream reasoning. These metadata are the absolute distance, the detected vehicle's global pose/velocity, and its lane/segment identifiers. Detections are then passed to the longitudinal controller which handles leader identification. The module reduces the number of detected vehicles to one single "leader ahead" candidate, which enables control objectives such as inter-vehicle distance keeping between vehicles in traffic.

This provides the longitudinal controller enough information to implement behaviors such as safe inter-vehicle distance keeping.

3.3.3 Closed-Loop Control System

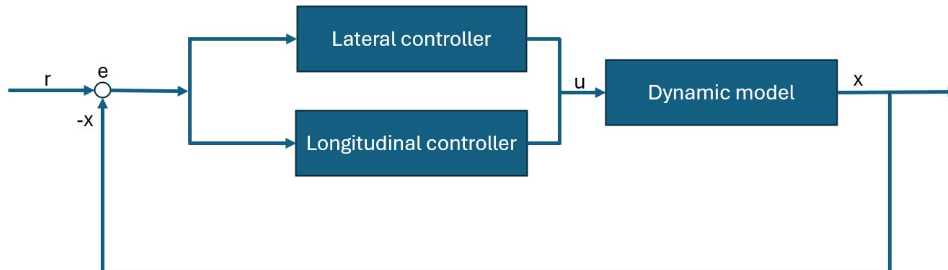

Figure 3.20: Block diagram of the closed-loop update at each time step.

Each vehicle in the simulator executes a closed-loop update at every simulation timestep as illustrated in Figure 3.20. The vehicle receives a trimmed reference trajectory and constructs a perception output of the surroundings from its sensor(s). A lateral (steering) controller and a longitudinal (speed/spacing) controller then compute commands, which are passed to the dynamics model. The dynamics integrates the model and advances the vehicle state.

3.3.3.1 Lateral Control (steering)

A **lateral controller** regulates the vehicle's direction by computing a steering command that keeps the vehicle stable on the reference path. Typical goals include minimizing heading and cross-track errors, smooth steering, and look-ahead tracking.

The **lateral controllers** in the implementation have access to the current time and timestep $(t, \Delta t)$, the trimmed reference trajectory, the perception data object (`PerceptionData`) and a compact vehicle state object (`EgoState`: $(x, y, \theta, v, v_{max})$ which can be extended with necessary states required by the controller (see Section 4.2.4). Using these inputs, the controller computes and returns a single steering angle δ in radians. Any controller can be implemented and used as long as it produces a steering angle δ from these inputs and follows the abstract lateral interface `LateralControllerInterface`.

3.3.3.2 Longitudinal Control (speed/spacing)

A **longitudinal controller** regulates the vehicle's forward motion by computing the appropriate acceleration or deceleration (a) to satisfy high-level

objectives such as maintaining speed, ensuring safety or platooning strategies.

The **longitudinal controller** has access to the current time and timestep $(t, \Delta t)$ as well as the vehicle state object `EgoState`. It also has the perception data object `PerceptionData` that contains all detected vehicle neighbors in the ego frame with relative measurements (relative position, relative speeds) which also can be extended with (see Section 4.2.4). When the controller is unconstrained by traffic, the controller could track a desired speed, and when the controller is constrained by traffic, the controller can use neighbor data to get the nearest vehicle ahead. The exact signature and implementation are discussed in Section 4.2.4.

3.3.3.3 Vehicle Dynamics

The dynamics model updates the vehicle state by using the received commands (δ, a). In general form, let $\mathbf{s} \in R^n$ be the state and $\mathbf{u} \in R^m$ the control:

$$\dot{\mathbf{s}} = f(\mathbf{s}, \mathbf{u}, t),$$

where $\mathbf{u}$ is the control input (δ, a) computed by the controllers at time t:

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_{\text{lat}} \\ \mathbf{u}_{\text{long}} \end{bmatrix} = \begin{bmatrix} f_{\text{lat}}(t, \Delta t, \texttt{EgoState}, \texttt{trajectory}) \\ f_{\text{long}}(t, \Delta t, \texttt{EgoState}, \texttt{PerceptionData}) \end{bmatrix}.$$

The solver then integrates the ODE over the short horizon $[0, \Delta t]$ from the current state $\mathbf{s}(t)$:

$$\mathbf{s}(t + \Delta t) = \texttt{odeint}\Big(f,\ \mathbf{s}(t),\ [0, \Delta t],\ \text{args} = (\mathbf{u},)\Big)_{\text{final}}.$$

This formulation is general and based on the choice of state dimension n and control dimension m. Different dynamics models (kinematic bicycle, dynamic bicycle, etc.) are interchangeable as long as they implement $f(\cdot)$ and the same integration structure.

3.3.3.4 Simulation Loop

At each simulation timestep, the following process is executed for every vehicle:

1. Receives the trimmed reference trajectory from its current pose.

2. Builds a perception object (`PerceptionData`) that normalizes detected vehicles into `neighbors`.

3. Computes the steering according to:
   $\delta \leftarrow f_{\text{lat}}(t, \Delta t, \texttt{EgoState}, \texttt{reference_trajectory})$

4. Computes the acceleration according to:
   $a \leftarrow f_{\text{long}}(t, \Delta t, \texttt{EgoState}, \texttt{PerceptionData})$.

5. Integrates the dynamics with (δ, a) and produces a new *EgoState* for $t + \Delta t$.

3.4 Visualization

The simulator's rendering is handled by the `Visualization` class. It encapsulates attributes and methods to visualize the simulation, and provides a modular interface for managing components that can be visualized and rendered in a Pygame window (simulation window). The class also include methods for centering the road network within the created window and pan/zoom interaction.

The rendering is split into a static layer (roads and lane trajectories) that is drawn only once as a cached background surface, and a dynamic layer (vehicles) that is redrawn every simulation step. At initialization the `update_-background()` is called, which executes `road_visualization()` and `road_trajectory_visualization()`. `road_visualization()` draws the roads and `road_trajectory_visualization()` draws the trajectories/center-line along each lane. Then, when the simulator is running, at each simulation step `vehicle_update_visualization()` is called, where each vehicle is redrawn, as their state changes. However, when parameters change (pan/zoom, window resize) the background is regenerated.

3.4.0.1 Lane trajectories

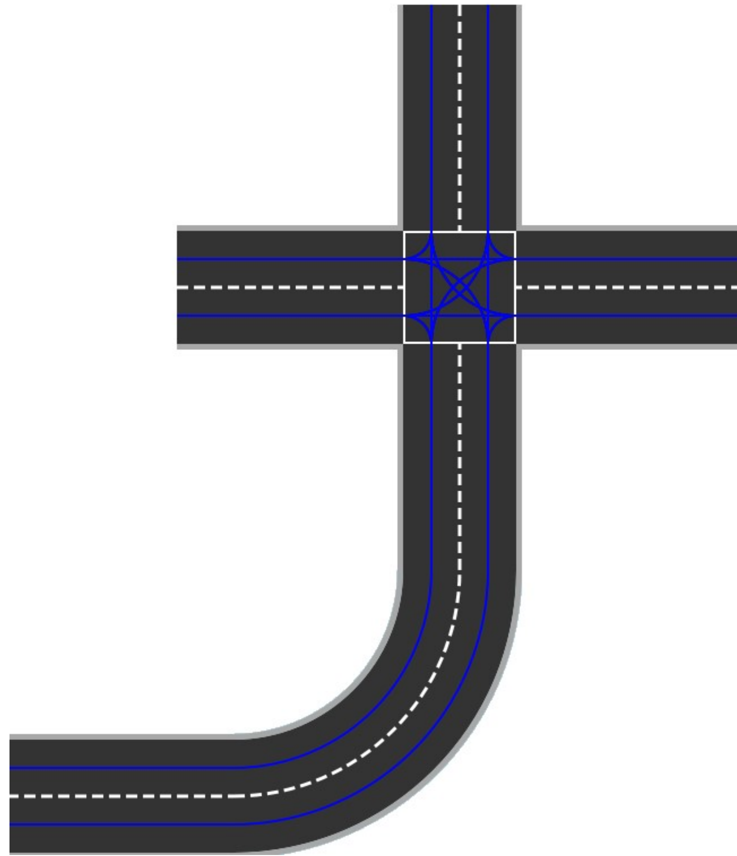

Figure 3.21: One straight, curved and intersection segment connected sequentially with blue lane trajectories along each lane center.

Lane centerline/trajectories are rendered per segment via `segment.draw_-trajectory()`. The visualization collects both fixed segments and detailed segments (`road_segment_manager.fixed_segments` and `road_segment_manager.all_segments`) and draws trajectories for the union. This provides a consistent global view of the lanes used by the controllers and for debugging. The generated lane trajectories across the segments are illustrated in Figure 3.21 and 3.5 as the blue lines. The blue lines are generated along the lane centers which the vehicles will follow.

3.4.0.2 Coordinate Systems and World Screen Transform

The simulator uses a global world frame W with the positive x-axis pointing to the right and the positive y-axis upward. Pygame's screen frame S is different, it uses positive x-axis to the right but positive y-axis downward (top-left of the screen is (0,0)). Rendering therefore applies an affine transform with scale s (pixels per meter), offsets (o_x, o_y) (pixels) and the current window height H (pixels):

$$x_S = o_x + s x_W, \qquad y_S = (H - o_y) - s y_W$$

The inverse transform from screen to world coordinates is primarily used when zooming around the mouse cursor. By first converting the cursor's screen position into world coordinates, the system can update the scale and offsets such that the same world point remains under the cursor after zooming. This gives an intuitive zoom behavior where the user zooms in at the point of interest, rather than shifting the whole scene. The inverse transform is as follows:

$$x_W = \frac{x_s - o_x}{s}, \qquad y_W = \frac{H - y_S - o_y}{s}.$$

3.5 Conventions and Terminology

3.5.1 Coordinate Frames

The simulator supports multiple different types of road segments, which are diverse in their geometry. To ensure correct functionality, these road segments need to be accurately represented. This is done by implementing a dual coordinate system. These are local coordinates for segment related computations and global coordinates for absolute positioning in the road network.

3.5.1.1 Global Coordinate System

The global coordinate system W is a fixed Cartesian reference frame, where all absolute positions, orientations and trajectories are represented. The x-axis points to the right and the positive y-axis points upward (see Figure 3.22). All road segments, vehicle positions, visualization, sensor readings, and inter-segment computations are expressed in this frame. Vehicle dynamics, sensor models, and global trajectory tracking also rely on global coordinates.

3.5.1.2 Local Coordinate System

Each road segment defines its own local coordinate system, where the origin of the local coordinate system is typically placed in a meaningful geometric reference, such as the start of a straight segment or in the center of a curved arc segment. Figure 3.22 shows how the local coordinate systems have been defined for the different road segment types.

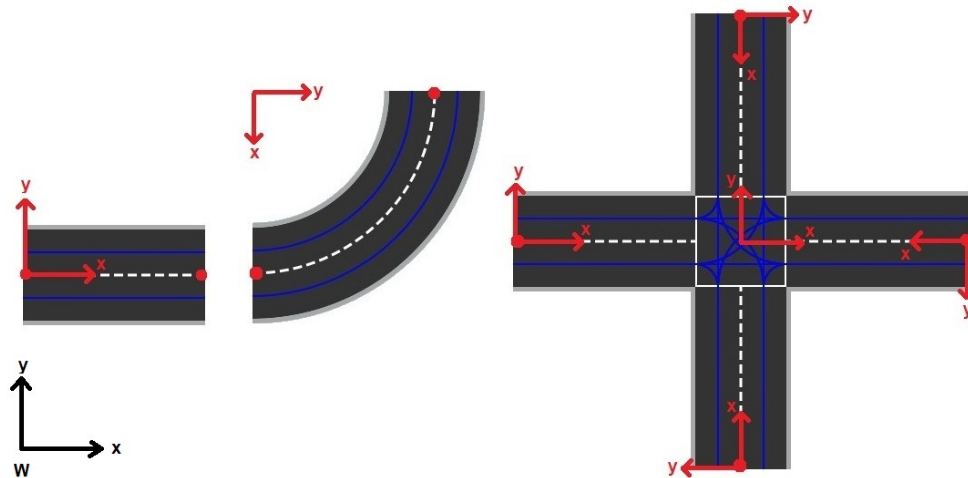

Figure 3.22: The straight segment and intersection segment was created with an orientation of $0°$ while the curved segment has a $270°$ orientation.

The local coordinate system allows each road segment to define its geometry independent of its placement in the global network.

In the local coordinate system:

- Straight segments extend along the local x-axis, with lanes and boundaries defined relative to this axis.
- Curved segments are defined using polar coordinates with respect to a local origin at the center of the curvature.
- Intersections define their own internal layouts relative to a central reference point.

Intersections are composed of multiple segments (one intersection road and four straight sub segments) with one main local coordinate system defined with its origin at the intersection center and multiple derived local coordinate systems, one at each subsegment. Each subsegment is implemented as a

straight subsegment that reuses the straight road conventions (its local x-axis runs along the road). The sub segments local coordinate systems are helpful for initializing vehicles on the intersection sub segments relative to their local coordinate system.

3.5.2 Coordinate Transformations

To support seamless alignment when connecting segments and accurate trajectory generation, the simulator performs a set of transformations:

- `local_to_global`: Transforms a point from the segment's local coordinate system to global coordinates. It is used to express road geometries, connection points, lane boundaries and trajectories in global coordinates so they can be visualized and followed by vehicles. This method is placed in the `RoadSegment` superclass.
- `global_to_local`: Transforms a point from global coordinates to the segment's local coordinate system. This is primarily used to express the vehicle states relative to the road segment, which simplifies computations such as, cross-track error, heading error, and checking whether the vehicle remains within the bounds of a segment.
- `polar_to_cartesian`: Converts polar coordinates (r, θ) to Cartesian coordinates (x, y) in the segment's local frame. This is used in `CurvedRoad` class, where lane boundaries, connection points, and reference trajectories are expressed in polar form relative to the curve center. Transforming these into Cartesian coordinates, makes the simulator represent arc geometries in a form suitable for trajectory generation, vehicle placement, and visualization. Some example methods that utilize this conversion is `setup_lane_boundaries()` and `setup_lane_connection_points()`.
- `cartesian_to_polar`: Converts Cartesian coordinates (x, y) to polar coordinates (r, θ) relative to the curved segment's center or arc centers within intersection center turns. This is used to evaluate vehicle states on curved and intersection roads. For example, it checks whether a position is within the arc's bounds and determines which lane the vehicle is on. It is also used to compute the progress along the curve. Methods that utilize this conversion include `is_position_within_segment()` and `get_lane()` in `CurvedRoad`.

These transformations are crucial for computing connection point positions, aligning segments with arbitrary orientations, generating lane trajectories, defining lane boundaries, and transforming vehicle state information during segment transitions. The transformation from local to global coordinates is defined as follows:

$$\begin{bmatrix} x_{global} \\ y_{global} \end{bmatrix} = R(\theta) \begin{bmatrix} x_{local} \\ y_{local} \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{3.23}$$

where (x_0, y_0) is the global origin of the segment, θ is the segments orientation in the global coordinate system relative to the global x-axis given in radians, and R(θ) is the rotation matrix:

$$R(\theta) = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{3.24}$$

The inverse transformation from global to local coordinates is defined as follows:

$$\begin{bmatrix} x_{local} \\ y_{local} \end{bmatrix} = R(\theta)^\top \left(\begin{bmatrix} x_{global} \\ y_{global} \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \right) \tag{3.25}$$

An example where `local_to_global` mapping is applied is when connecting the end of one segment to the start of another:

1. Compute the orientation difference $\Delta\theta$ between the connection point directions, expressed in global coordinates.

2. Rotate the moving segment by $\Delta\theta$, giving the updated orientation $\theta' = \theta + \Delta\theta$.

3. Translate the moving segment so the connection points coincide. If $\mathbf{p}_A^{\text{glob}}$ is the fixed connection point in global coordinates and $\mathbf{p}_B^{\text{loc}}$ is the corresponding local connection point on the moving segment, the new translation vector is

$$\mathbf{t}' = \mathbf{t} + \Big(\mathbf{p}_A^{\text{glob}} - \big(R(\theta')\,\mathbf{p}_B^{\text{loc}} + \mathbf{t}\big)\Big).$$

4. Recompute all global positions (connection points, lane boundaries, trajectories) by applying

$$\mathbf{q}^{\text{glob,new}} = R(\theta')\,\mathbf{q}^{\text{loc}} + \mathbf{t}'.$$

In this example, only the `local_to_global` mapping is applied during the connection step. In contrast, the `global_to_local` mapping is applied to vehicle states during simulation to compute cross-track error and heading error once vehicles are moving on the road network.

3.5.3 Application of the Transforms

Although vehicles operate in a global coordinate frame, many perception and planning tasks are performed in a local frame tied to the current road segment. This dual-frame approach offers modularity and geometric clarity. Some crucial use cases include:

- **Trajectory planning**: Reference trajectories are generated in the local frame for geometric simplicity and are then transformed into global coordinates for execution.
- **Lane boundary interpretation**: For tasks like lane-keeping the vehicle's global position is transformed into the local frame of the current road segment, where boundary checking and lateral offset calculations are easier and straightforward.
- **Sensor perception**: In the simulation, detected vehicles are first identified using global positions for efficiency. The measurements are then expressed in the ego vehicle's local frame, consistent with how real sensors operate. Relative positions and velocities are obtained by rotating the global displacement and velocity vectors into this local frame, while attributes such as lane affiliation and road segment ID are appended from the detected vehicle's state. This ensures that perception data are geometrically consistent and usable for tasks such as collision checking and leader selection.

This separation between global and local frames enables consistent geometric reasoning while supporting modular road segment construction and scalable global navigation.

3.5.4 Segment Conventions

- Pygame uses a different coordinate system for visualization. The x-axis points to the right and the y-axis points downward.
- Each segment defines connection points where other segments can attach:

  - Straight and curved segments have two connection points start and end.
  - Intersection segments have four connection points: start, end, left, and right

- The local coordinate system for straight road segments is positioned at the ”start” connection point.
- The local origin of all segment types is placed at (0,0). For straight segments, this corresponds to the start connection point, for curved segments, it corresponds to the circle center and for intersections, it corresponds to the geometric center (see Figure 3.23).
- Each segment’s local coordinate system is defined relative to the global coordinate system. The local x-axis points towards the end connection point for all segments except for the left-directional curves for which it points towards the start connection point.
- Positive rotation is counterclockwise.

3.5.5 Lane Identification

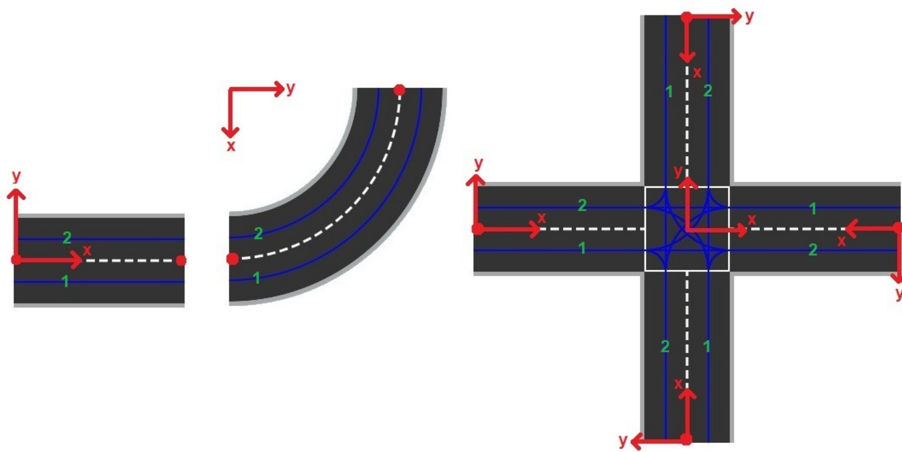

Figure 3.23: Visual representation of lane indexing on lanes relative to the road segments local coordinate system

- Lanes are indexed starting from 1.

- Lanes are numbered bottom to top along the positive local y-axis.
- Lane 1 always refers to the bottommost lane relative to the segment's local coordinate system.
- Lane width is uniform for all lanes within the road segments.

Figure 3.23 illustrates these for the different segment types.

3.5.6 Driving Direction and Segment Orientation

The simulator also supports reasoning about vehicle driving direction relative to the segment orientation:

- A vehicle driving in the global right half-plane (between $-90°$ and $90°$) is considered to have a `POSITIVE` driving direction.
- A vehicle driving in the global left half-plane (between $90°$ and $270°$) is considered to have a `NEGATIVE` driving direction.

This is how the user should reason when initializing vehicles on the road network.

3.6 Design Principles and Architectural Structure

This section discusses the decisions and their motivation for the architectural design choices. The simulator has been designed according to established OOP such that modularity, extensibility, and reusability is applied across the architecture. These principles ensure that individual components have a clear responsibility, while still enabling flexibility, extension and maintenance. The overall design is also structured around the use of established design patterns (see Section 2.3). These patterns provide structure for recurring problems in the system architecture:

3.6.1 Layered composition

The architecture is composed in layers with clear dependency direction:

- Entry layer (`SimulationProject`): Provides the experiment entry point and drives the simulation loop through time stepping, updates, rendering and logging.
- Integration layer (`TrafficEnvironment`): Provides a facade to the simulation while mediating synchronized interactions between road and vehicle managers to keep them decoupled.
- Manager layer:
  - `RoadSegmentManager`: handles segment creation, connection, coordinate transforms, and network maintenance including connection points states and geometry operations.
  - `VehicleManager`: manages each vehicles lifecycle and orchestrates them).
  - Implements the control-loop behaviors of each vehicle by combining sensing perception with controllers and dynamics. These subsystems operate at every simulation step by processing inputs and producing control outputs that drive the vehicle motion.

This decomposition isolates concerns into layers that never touches the internal controller logic and dynamics, and vice versa.

3.6.2 Design patterns applied

- **Factory Method (Creational)**: The instantiation of road segments is decoupled from their usage. The `RoadSegmentFactory` generates appropriate subclass instances based on parameters configured by the user, which allows new road geometries to be added without modifying existing code.
- **Composite (Structural)**: The `IntersectionRoad` is much more complex compared to `StraightRoad` and `CurvedRoad`, since its composed of multiple subsegments. Using the composite design pattern allows the segment to be treated as a single uniform road segment.
- **Façade (Structural)**: The `TrafficEnvironment` acts as a façade that exposes a simplified, unified interface to the simulation subsystems. Rather than interacting directly with the `VehicleManager` and `RoadSegmentManager`, user-level code and simulation scripts interact only with the `TrafficEnvironment`. This reduces complexity, improves readability.

- **Strategy (Behavioral)**: Controllers and vehicle dynamics are encapsulated behind interchangeable interfaces. This allows for the implementation of multiple plug-and-playable algorithms without having to change the algorithm logic. During the configuration process the user only needs to choose the name of the class, which is implemented in the interface. This implementation ensures that the user configuration code remains the same, while the addition of new dynamics and controllers are implemented.

- **State (Behavioral)**: Connection points for each segment maintain explicit states (`CONNECTABLE`, `CONNECTED`), that determine whether or not a road segments connection point is connected to another segment. This improves consistency in the road network, and is an intuitive convention in the code.

- **Template Method (Behavioral)**: The `RoadSegment` superclass defines a general interface with attributes and methods for setting up connection points, lane boundaries, and coordinate transformations. Subclasses such as `StraightRoad`, `CurvedRoad`, and `IntersectionRoad` inherits (attributes and methods) from the `RoadSegment` and overrides them if necessary. Geometrically different road segments such as `CurvedRoad` and `IntersectionRoad` override most of the methods in the `RoadSegment` to implement their unique geometry and geometric specific logic.

- **Adapter (Structural)**: The `SeparatedControllerAdapter` adapts two independent controllers (`LateralController` and `LongitudinalController`) to the `CombinedController` interface used by the vehicle control loop. This enables plug-and-play composition without modifying existing controller classes and cleanly unifies their outputs (e.g., steering and acceleration) for the dynamics layer.

3.6.3 Simulator structure: Motivation and Benefits

The structure was deliberate to enable modularity, extensibility and flexibility. This allows:

- **Plug-and-play experimentation.** Swapping controllers or changing the dynamic model, for example a bicycle model interchanged by a unicycle requires no edits outside the corresponding module.

- **Safe extensibility without interface churn.** New perception signals flow through *PerceptionService* into `PerceptionData` as optional fields, where existing controllers remain valid.
- **Separation of concerns.** Geometry belongs to segments and perception, assembly belongs to the service, control logic belongs in controllers. This eliminates hidden dependencies and duplicated math.
- **Performance by construction.** Controllers use compact objects (`EgoState`, trimmed reference, `PerceptionData`) instead of querying global state, this keeps the control loop small and clean.
- **Traceability.** Serialization make experiments auditable and results reproducible.

3.6.4 Trade-offs

The framework prefers more explicit types and small modules such as the `PerceptionService`, over larger classes. This slightly increases the amount of files, however it significantly improves readability, local reasoning, testing, and future extensions. Controller interfaces are intentionally narrow such that controllers with additional input requirements need to extend the perception and vehicle state data objects, rather than changing method signatures, which preserves backward compatibility and comparability of results.

3.7 Technologies and Tools Used

- **Programming Language**: Python 3.11
- **Visualization**: Pygame, Cairo, NetworkX
- **Libraries**: numpy, matplotlib

Chapter 4

Multi-vehicle simulator: A Tutorial

This chapter provides an overview of how to use the developed simulation framework. It outlines the essential steps required to use the simulator, configure the road network and vehicle setup, as well as manage the simulation through saving and loading functionalities. These sections are organized in a logical sequence to ensure that the simulator is correctly setup from start to finish.

4.1 Project Structure Overview

The process of setting up the simulator is organized into modular components, where each step of the configuration is handled by a dedicated module. This modular structure facilitates navigation, modification, and extension of the code base. The main components are:

- `main.py` - Entry point for running simulations.
- `road_setup.py` - Defines and configures all road segments.
- `connection_setup.py` - Handles how road segments are connected.
- `vehicle_creation.py` - Defines and configures all vehicles.
- `vehicle_placement.py` - Handles the placement of vehicle on the road network.

- `dynamics.py` - Modular framework that stores implementations for vehicle dynamic models.
- `controllers.py` - Modular controller frameworks supporting multiple interchangeable control strategies.
- `virtual_parking_lot_setup.py` - Manages vehicle re-entry setup and platooning size when entering the virtual parking lot.
- `config.py` - Configuration parameters for simulations such as route instructions, simulation duration, time step, save/load project and video.
- `vehicle_data_logging.py` - Configuration parameters for logging and plotting vehicle attributes.

4.2 Configuration and Customization

Users can configure or customize all major elements of the simulation.

4.2.1 Road Network Configuration

To create road segments, navigate to toolbox/initialization/road_setup.py. This is where all road segments need to be configured in order to create a desirable road network. The function used to create a road segment has the following function signature:

```
def create_road_segment(
    segment_type: int,
    length: float = None,
    orientation: float,
    lane_width: float,
    lanes: int,
    radius: float = None,
    direction_of_curvature: str = None,
    central_angle: float = None,
    road_overlap: float,
    speed_limit: float
)
```

Some parameters are common for all segments, however, these segments have different geometric properties, which means that some attributes are specific

to some segments. For example, straight segments have a length, which curved segments do not have. Curved segments instead have parameters such as radius and central angle which are specific to curved segments. Table 4.1 shows the required parameters for each segment type to be properly instantiated.

Table 4.1: Relevant Parameters by Segment Type

Segment Type	**Relevant Parameters**
1 (Straight)	`segment_type, length, orientation, lane_width, lanes, road_overlap, speed_limit`
2 (Curved)	`segment_type, orientation, lane_width, lanes, radius, direction_of_curvature, central_angle, road_overlap, speed_limit`
3 (Intersection)	`segment_type, length, orientation, lane_width, lanes, road_overlap, speed_limit`

As shown in Table 4.1, the parameter `segment_type` takes values from 1 to 3, where 1 corresponds to a straight segment, 2 to curved segment, and 3 to an intersection. The orientation represents how much the segment is oriented in degrees in the counter-clockwise direction from the positive x-axis. The `lane_width` defines the width of individual lanes, and `lanes` specifies the total number of lanes within the segment. The road_overlap extends the current segment onto the connected segment, thereby preventing small discontinuities in the road network that can arise from rounding errors involving π. This ensures that vehicles remain on roads and do not appear in a state between two connected roads with discontinuity, causing the simulator to crash . `speed_limit` enforces the maximum speed that a vehicle can have on the segment.

Curved segments require additional parameters, such as `radius`, `central_angle` and `direction_of_curvature`. Radius determines the size of the road rather than a length, central angle determines the curvature and overall length of the arc, where larger central angle results in a longer curve. Lastly, direction of curvature defines whether the road curves to the left or right from its initial position.

4.2.2 Road Network Connection Configuration

The previous section described how to create road segments. This section explains how to connect them. To connect the created road segments, navigate to toolbox/initialization/connection_setup.py. In this module, the user specifies which segments and connection points should be connected. To connect two segments, use the following method signature:

```python
def connect_road_segments(
    self,
    fixed_segment_index: int,
    connection_point_1: str,
    moving_segment_index: int,
    connection_point_2: str,
    automatic_connection: bool = False
    connecting_open_ends: bool = False
)
```

The fixed_segment_index refers to the segment that is fixed in place. The moving_segment_index is the segment that will be translated and rotated to form the seamless connection with the fixed segment. The connection_point_1 is the connection point on the fixed segment and the connection_point_2 is the connection point on the moving segment.

As mentioned in Section 3.5, all segments are initially placed at the same origin. To determine the correct segment indices, refer back to `road_setup.py`, where each segment is created in order. For example, if ten segments are created, their indices range from 1 to 10, corresponding to the order of creation.

The naming convention distinguishes between the *fixed segment* (which remains stationary) and the *moving segment* (which aligns itself with the fixed one). After specifying the appropriate indices and connection points, the *moving segment* is translated and rotated appropriately and the connection is formed.

Lastly, the automatic_connection and connecting_open_ends are initially set to False in the method and are used as flags to automatically connect selected open ends on the road network. The next section will explain this further..

4.2.3 Automatic road network open end connection

When the configuration and connection of all the road segments has been completed, there can be open ended roads that are not directly connected. To avoid manual alignment in such cases, the simulator provides a feature for automatic road generation that connects two open ends with a geometrically feasible path.

This functionality is implemented in `toolbox/initialization/open_end_automatic_connection.py`. The user specifies two connection points and a turning radius, and the system computes a path composed of a

CSC sequence, following the principles of Dubins path generation.

The function signature to achieve this functionality is:

```
def automatically_generate_road_to_connect_open
_end_segments(
self,
segment_index_1: int,
subsegment_1: str,
connection_point_1: str,
segment_index_2: int,
subsegment_2: str,
connection_point_2: str,
radius: float
)
```

where:

- `segment_index_1`, `segment_index_2` is the two segments that are chosen to be connected.
- `subsegment_1`, `subsegment_2` specify the subsegments if applicable (only works for intersections)
- `connection_point_1`, `connection_point_2` select the open ends to be connected.
- `radius` defines the curvature constraint for the generated arcs.

Example: Assume that segment 1 is a straight road oriented east (0°), segment 2 is a curved road with a central angle of 90° and segment 3 is an intersection. The straight segment's `end` connection point is connected to the curved road's `start` connection point and the intersection's `start` connection point is connected to the curved road's `end` connection point. This configuration is shown in the network graph in Figure 4.1 and the corresponding physical road network in Figure 4.2.

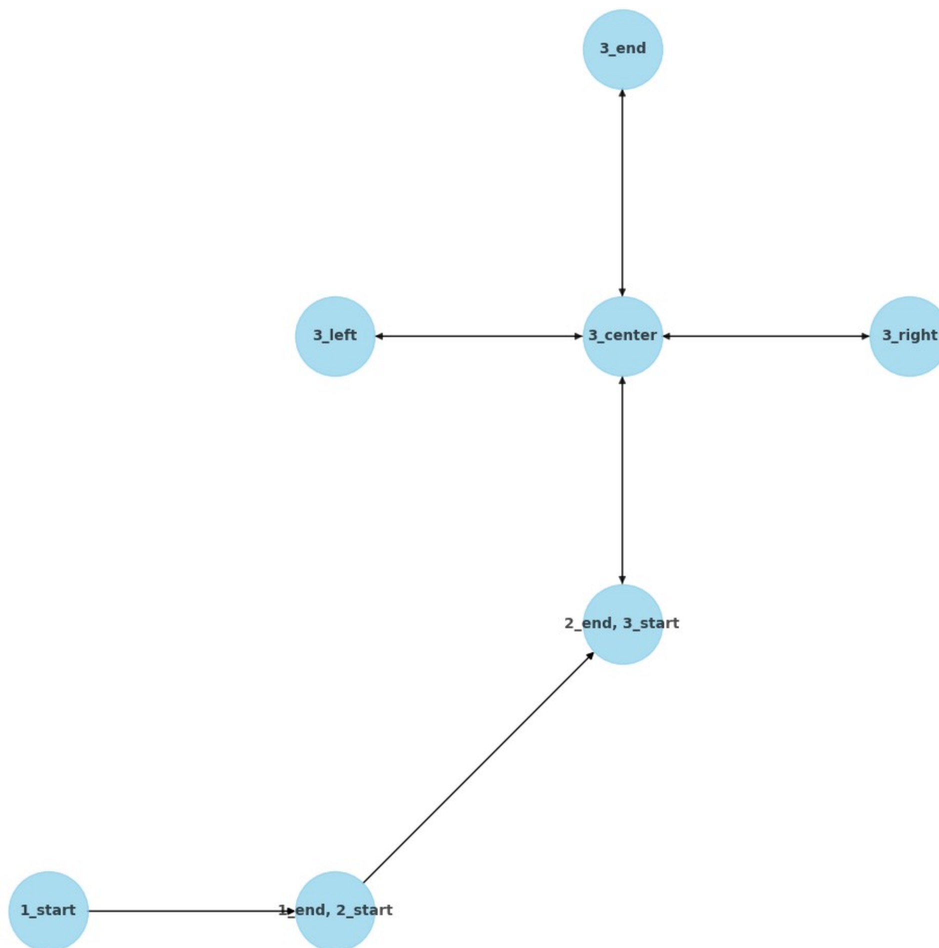

Figure 4.1: Road network graph before automatic road generation to connect open ends.

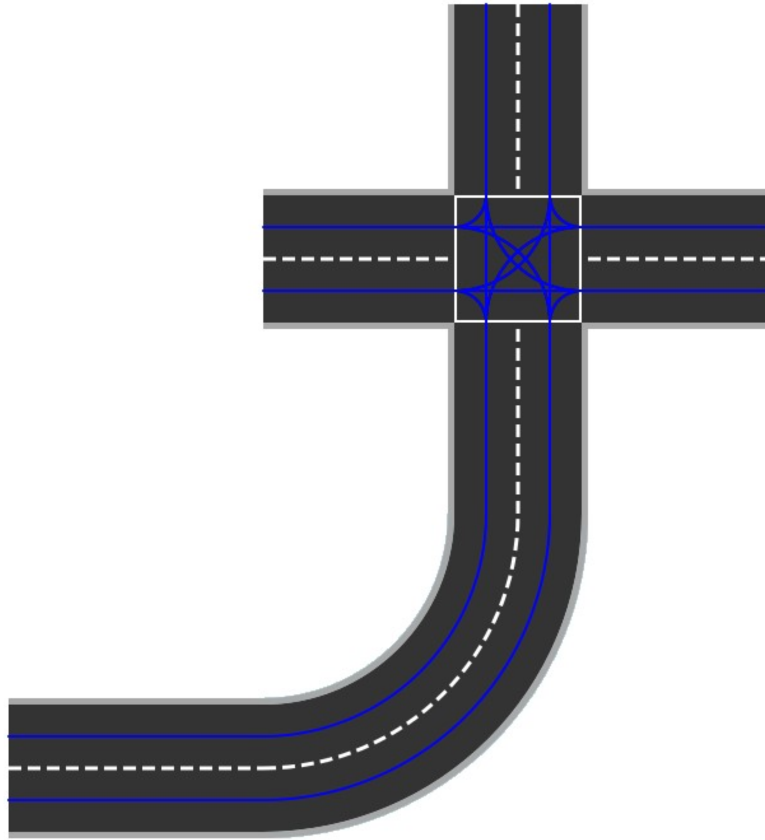

Figure 4.2: Physical road network before automatic road generation to connect open ends.

To connect segment 1 (`segment_index_1=1`) from connection point `1_-start` (`connection_point_1=start`) to segment 3 connection point `3_left` (`segment_index_2=3` (`subsegment_2=left`), connection point `connection_point_2=start`) with a radius of 100 for both generate curves, the call is:

```
automatically_generate_road_to_connect_open
_end_segments(
self,
segment_index_1 = 1,
subsegment_1: None,
connection_point_1 = "start",
segment_index_2 = 3,
subsegment_2 = "left",
```

```
    connection_point_2 = "start",
    radius = 100
)
```

After execution, the simulator generates a RSR sequence that smoothly connects the two open ends. The updated network graph is shown in Figure 4.3 while the resulting physical road network is presented in Figure 4.4.

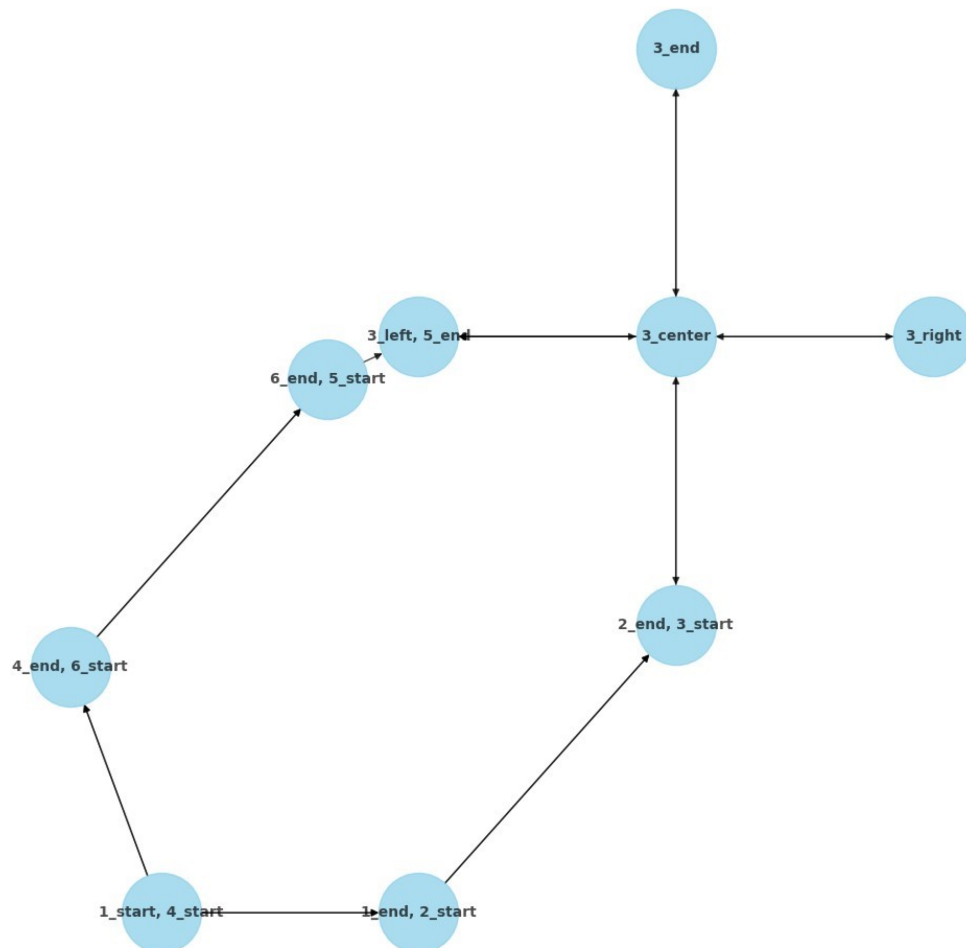

Figure 4.3: Network graph after automatic road generation connecting the chosen open ends.

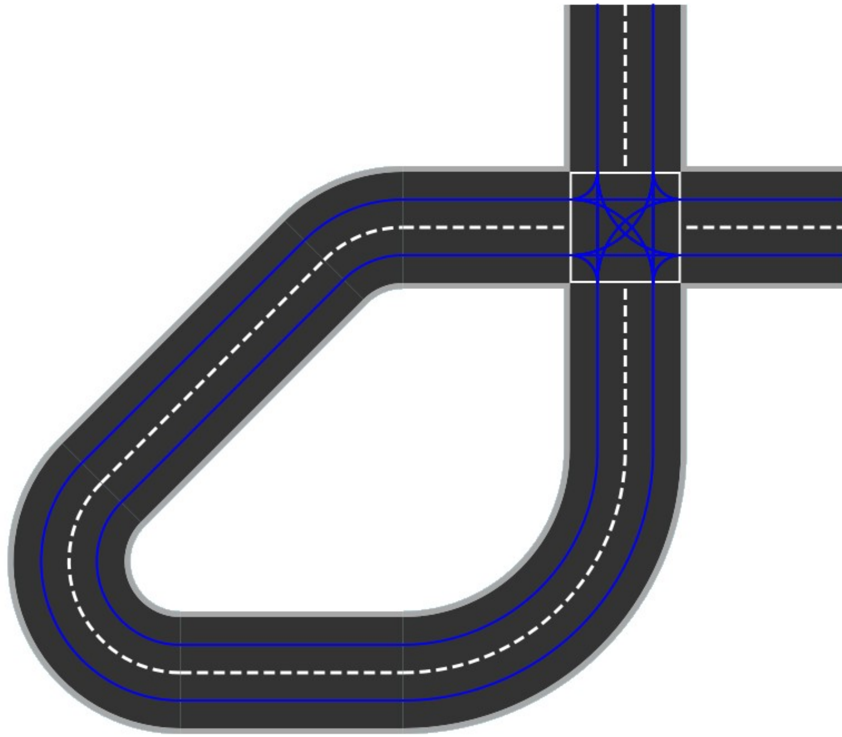

Figure 4.4: Physical road network after automatic road generation to connect open ends.

4.2.4 Implementation of Controllers and Dynamics

The simulator is designed with plug-and-play structure for both controllers and dynamics (see Section 3.3.3) which allows users to easily implement standard algorithms or introduce new ones without modifying the surrounding system. Each controller must adhere to a dedicated interface that prescribes a common function signature, ensuring consistency across implementations. All controller classes are defined in the module `vehicle/controllers.py`. The design follows an object-oriented hierarchy consisting of three key abstract base classes:

- `CombinedController`: A general interface for controllers that compute both lateral and longitudinal control actions within a single class.
- `LateralController`: An abstract base class for controllers that handle only steering (e.g., a PID-based lateral controller).

- `LongitudinalController`: An abstract base class for controllers that handle acceleration and braking (e.g., a PID-based Adaptive Cruise Control).

The required method signature for all controller classes is:

```
@abstractmethod
def compute_control_input(
    self,
    t: float,
    dt: float,
    ego_state: EgoState,
    perception: PerceptionData,
    reference: list[Point2D]
):
    pass
```

To combine separate lateral and longitudinal controllers into a unified control module, the simulator provides an adapter class called `SeparatedControllerAdapter`, which implements the `CombinedController` interface. This design follows the Adapter pattern, allowing independent controllers to operate together within the same control loop. The adapter delegates control to each sub-controller and merges their outputs:

```
class SeparatedControllerAdapter(CombinedController):
    def __init__(self, lateral_controller, longitudinal_controller):
        self.lateral_controller = lateral_controller
        self.longitudinal_controller = longitudinal_controller

    def compute_control_input(self, t, dt, ego_state, perception, reference):
        lat_cmd = self.lateral_controller.compute_control_input(
            t, dt, ego_state, perception, reference)
        long_cmd = self.longitudinal_controller.compute_control_input(
            t, dt, ego_state, perception, reference)
        return {"lateral": lat_cmd, "longitudinal": long_cmd}
```

This structure ensures that users can either:

1. Implement a unified `CombinedController` that handles both control tasks within one class (e.g., `CombinedPID`)

2. Alternatively combine separate `LateralController` and `LongitudinalController` implementations through the `SeparatedControllerAdapter`.

A typical lateral controller, such as a PID-based steering controller, inherits from `LateralController` and computes steering commands based on heading and cross-track error:

```
class PIDController(LateralController):
    def compute_control_input(self, t, dt, state, perception, reference):
        # Compute heading and cross-track errors
        # Apply PID control law with smoothing and steering limits
        return u_clipped
```

Similarly, a longitudinal controller such as a PID-based Adaptive Cruise Control (ACC) inherits from `LongitudinalController` and computes acceleration commands based on relative distance and relative velocity to the leading vehicle:

```
class PIDACC(LongitudinalController):
    def compute_control_input(self, t, dt, ego_state, perception, reference):
        leader = self._select_leader(perception)
        if leader is not None:
            # Maintain safe following distance
            ...
        else:
            # Cruise at desired speed
            ...
        return acc_cmd
```

Both controllers are plug-and-play compatible with the adapter and return control inputs that can be used directly by the vehicle dynamics model.

This modular interface makes it easy to introduce new control algorithms by simply implementing the corresponding abstract base class.

4.2.4.1 Extending the Inputs of the Controller modules

If a controller requires information that is not already exposed, the correct approach is to extend the `PerceptionData` and `EgoState` data classes. This ensures that the existing controllers remain valid while new controllers can access the additional data.

To extend the data entries, the following process has to be followed:

1. Add a new optional field to `PerceptionData` (or `EgoState` for plant/dynamics limitations), with a safe default.
2. Populate this field in the vehicle's perception/dynamics builder (controllers that do not use it remain unchanged).
3. Keep the controller method signatures identical, since controllers ignore the fields they do not need/use.

This preserves plug-and-playability while allowing inputs to be extended.

For example, if a longitudinal controller requires an estimate of time-to-collision (TTC), an additional field can be introduced in `PerceptionData` (navigate to `vehicles/perception_data.py`):

Listing 4.1: Extending PerceptionData with a TTC field.

```
@dataclass
class PerceptionData:
    neighbors: list = field(default_factory=list)
    ttc_leader: Optional[float] = None  # new optional field
```

This field is then populated in the perception builder `PerceptionService` using the `build()` method so that controllers can access it when needed. Similarly, geometry parameters such as `wheelbase` can be added to `EgoState` (navigate to `vehicles/ego_state.py`) if required by advanced lateral controllers.

4.2.5 Vehicle Creation

Vehicle creation is handled in `toolbox/initialization/vehicle_-creation.py`. To create a vehicle use the following method:

```
def create_vehicle(
    self,
    vehicle_id: int,
    lateral_controller: LateralController,
    dynamic_model: DynamicModel,
    longitudinal_controller: LongitudinalController
)
```

Each vehicle has to be assigned a unique `vehicle_id` manually which is later used to place vehicles on the road network. A lateral controller and a longitudinal controller from the `toolbox/vehicles/controllers.py` have to be specified. Lastly, a dynamic model has to be chosen, located in `toolbox/vehicles/dynamics.py`. To implement the controllers and the dynamic model, use the corresponding class names.

The dynamic model requires two inputs, a `ProbalilityDistribution` object and a `max_speed`. The `max_speed` is the top speed the vehicle can reach. The `ProbabilityDistribution` is used to assign the vehicle's initial speed at the start of the simulation. It takes in two integer parameters, the mean and the variance, which define the normal distribution of initial speeds. For example, `ProbabilityDistribution(mean=0, variance=0)`. The first parameter sets the initial mean speed of the vehicle, and the second defines the variance in the speed when the vehicle is placed on the road network. This ensures that vehicles are initialized with a normally distributed speed profile, rather than a fixed value.

4.2.6 Vehicle Placement

Vehicle placement is managed in the `vehicle_placement` module, located in `toolbox/initialization/vehicle_placement.py`. This module allows users to define the initial placement of vehicles on designated lanes and segments. In addition, parameters control the vehicle's offset from the lane center, longitudinal progress along the segment, and orientation relative to the road segments local coordinate systems x-axis.

The method signature for placing a vehicle on the road network is as follows:

```
def add_vehicle_to_segment(
        self,
        vehicle_id: int,
        segment_id: int,
        subsection: str,
        initial_entry_point_id: str,
        lane_id: int,
        driving_direction: str,
        offset_from_lane_center: float,
        offset_orientation: float,
        progress_along_lane: float,
        entry_point_id: str | None = None
) -> None:
```

The `vehicle_id` is the identifier assigned to the vehicle at vehicle creation (see Section 4.2.5). The `segment_id` is the id of the road segment on which the vehicle will be placed on (see road_setup.py). The `subsection` is used to place vehicles on the subsegments (arms) of intersections. Valid inputs to place a vehicle on the subsegments include `"start"`, `"end"`, `"left"`, and `"right"`, which correspond to the four intersection branches. Note that while intersection placement is supported, it is not generally recommended due to geometric complexity.

`initial_entry_point_id` specifies which connection point the vehicle first enters the segment from. This information is important during vehicle initialization, because together with the driving direction it the correct parametrization of the lane (i.e., how progress along the lane is computed).

The `lane_id` is the lane index (starting from 1) which the vehicle will be placed on. See full lane definition in Section 3.5.5. The `driving_direction` specifies the vehicles direction of travel. As described in Section 3.5.6, `POSITIVE` indicates forward motion along the

global $-90°$ to $90°$ direction, while NEGATIVE corresponds to travel in the $90°$ to $270°$ direction. The offset_from_lane_center defines how much lateral offset from the center of the lane the vehicle should be placed. A negative value moves the vehicle along the local negative y-axis direction, while a positive value moves the vehicle along the positive y-axis relative to the local coordinate system. An offset of 0 places the vehicle in the lane center. Note that the value that is inserted is in meters and not in percentage, and can be interpreted in the context of the lane width defined in the road setup. The offset_orientation specifies the vehicle's angular deviation (in degrees) from the segment's local x-axis (positive values rotate counterclockwise). Finally, the progress_along_lane indicates how far along the lane the vehicle should be placed. A value of 0 places the vehicle on the "start" connection point and a value of 100 places the vehicle on the "end" connection point. This value represents how far along from the "start" connection point the vehicle is placed. Lastly, entry_point_id is not part of the user configuration, it is assigned internally when a vehicle is dispatched from the virtual parking lot and is used to determine the correct entry direction during trajectory generation.

4.2.7 Virtual Parking Lot and Virtual Entry Points

To setup a virtual parking lot, locate virtual_parking_lot_setup in toolbox/initialization/virtual_parking_lot_setup.py. This module allows the user to choose from which exit points the vehicle should re-enter the road network and how large the platoon capacity should be. The function signature is written as follows:

```
def create_virtual_parking_lot(
    platoon_size: int
    exit_points: List[Dict{}]
    time_sequence_interval: float,
    time_mean: float,
    time_variance: float,
)
```

The platoon_size determines the size of every platoon before a new one is created. Only when the platoon is full the re-entry logic will be executed. The time_sequence_interval is the time between each consecutive vehicle departing while time_mean and time_variance define the mean and variance of the departure time for the first vehicle of each platoon, lastly exit_points are the points from which the platoons can re-enter the road

network. If the `exit_points` list is left empty, all open ends will be considered as valid re-entry points.

Internally, the parking lot relies on two abstractions:

- **`ProbabilityDistribution`**: Models the stochastic delay before the first vehicle in a platoon departs. For example, with `time_mean = 10.0` and `time_variance = 4.0`, the first vehicle will depart around $t = 10$ s with a Gaussian spread.
- **TimeSequence**: Generates the deterministic offsets between vehicles within the same platoon. For instance, with `time_sequence_-interval = 0.5`, each subsequent vehicle departs 0.5 s after the previous one.

To setup the `virtual_parking_lot_setup` module, the `platoon_-size` has to be chosen. The desired re-entry points need to be set according to the form {`'segment_id: int, 'connection_point': str`} if it is a straight or curved road segment or {`'segment_id: str, 'connection_point': str`} if it is a intersection road. For example, {`'segment_id': 2, 'connection_point': 'start'`} sets the *start* connection point of segment 2 (which can be a straight or curved segment) to a re-entry point. To set intersections with multiple connection points to a re-entry point, set `'segment_id' = "2_start"` if it is the start branch or {`'segment_id' = "2_left"`} if it is the left branch.

4.2.8 Saving/Loading and Logging with Visualization

The final configuration step before running the simulator involves the save, loading and logging with visualization feature.

4.2.8.1 Project Saving and Loading

The simulator is configured through two files. These are :

- `toolbox/config/config.py` - general simulation parameters.
- `toolbox/config/vehicle_data_logger.py` - parameters for logging vehicle data.

The file that is responsible for the simulator configurations is `config.py`, which always internally loads `config.py` which can not be overridden or specified as an external argument. To change the simulation setup, the user edits this file directly. The key parameters include:

- SIMULATION_ID - unique identifier (int) that names the simulation project.
- LOAD_PROJECT - is a boolean, if True, loads the saved project with the specified SIMULATION_ID. If the SIMULATION_ID is incorrect or does not exist a value error is raised.
- SIMULATION_DURATION - the time horizon over which the traffic environment is simulated.
- TIME_STEP - time increment per simulation step.
- HORIZON_LENGTH - distance that vehicles look ahead when following route instructions.
- SAVE_VIDEO - is a boolean, if True, saves a compressed video of the simulation. These videos are saved in the simulation_videos folder under the corresponding SIMULATION_ID.

For example, to save and load a simulation project, set SIMULATION_ID = 1 and LOAD_PROJECT = False. This will save the project to the simulation_projects folder with the SIMULATION_ID as the unique identifier. Now, to load the saved project, set SIMULATION_ID = 1 and LOAD_PROJECT = True.

4.2.8.2 Route Planning Configuration

Users may also want to configure route instructions globally or per vehicle. Global route instructions are only meant for vehicles on the same lane and segment. This ensures that every vehicle in the simulation follows the same route instructions. The vehicle route instructions allow each vehicle to individually follow the given route instructions. To apply the same route to all vehicles on the same segment and lane, set USE_UNIVERSAL_ROUTE = True and define UNIVERSAL_ROUTE_INSTRUCTIONS as a list of movement instructions. For example, UNIVERSAL_ROUTE_INSTRUCTIONS = ["straight", "left", "left_turn", "straight"]. If individual vehicle routing instructions are desired, set USE_UNIVERSAL_ROUTE = False and define the INDIVIDUAL_ROUTE_INSTRUCTIONS dictionary, where each key is a vehicle_id and the value is a list of instructions for that specific vehicle. For example, individual route instructions for 4 different vehicles is written on the following form:

```
INDIVIDUAL_ROUTE_INSTRUCTIONS =
{1: ["straight", "left", "left_turn", "straight"],
2: ["straight", "left", "left_turn", "straight"],
3: ["straight", "straight", "right_turn", "straight"],
4: ["straight", "right", "right_turn", "straight"]}
```

The available route instruction strings are:

- "straight": Remain on the current segment.
- "left": Perform a lane change to the left.
- "right": Perform a lane change to the right.
- "left_turn": Turn left in an intersection.
- "right_turn": Turn right in an intersection.

Instructions like `left_turn` and `right_turn` are only valid within the intersection segment (the sub-segments are excluded, since they don't have turns). Outside of intersections, these instructions default to `straight`. Furthermore, compound instructions such as `2_left` and `2_right` can be used to change multiple lanes at once. However, the simulator does not generate sufficiently smooth trajectories for such cases, causing the vehicle to fail.

4.2.8.3 Logging Configuration

The `vehicle_data_logging.py` module is located in `toolbox/config/vehicle_-data_logging.py`. It provides configurable vehicle attributes, which can later be logged and visualized for analysis. The module defines options for logging states and control data to allow systematic evaluation through visualization. Key configuration options include:

- `LOG_VEHICLE_DATA`: Enables logging of vehicle state and control data.
- `VEHICLE_IDS`: List of vehicle IDs to log.
- `ATTRIBUTES`: Select which attributes to log, for example, "position", "velocity", "control_input", "vehicle_path").
- `START_TIME`, `END_TIME`: Logging interval in simulation time.

- `INTERVAL`: Time step for logging, for example, every 0.1 seconds.
- `SHOW_PLOTS`: Displays plots after runtime or when the simulation is closed.
- `PLOT_ATTRIBUTES`: Attributes to visualize for example, `"control_input"`.
- `LOG_DIR`: Directory where log files and plots are saved.

The log directory is created automatically if it does not exist. All plots are saved to `logs/plot` by default. These plots provide insight into vehicle behavior and controller performance over the simulation horizon.

4.2.8.4 Example Configuration To Save Project

This example shows how to save a project with individual route instructions for vehicles with no video of the simulator and a horizon length of 200 meters.

```
# config.py
SIMULATION_ID = 1
LOAD_PROJECT = False
SIMULATION_DURATION = 1000.0
TIME_STEP = 0.1
USE_UNIVERSAL_ROUTE = False
UNIVERSAL_ROUTE_INSTRUCTIONS = []
INDIVIDUAL_ROUTE_INSTRUCTIONS = {
    1: ["straight", "left", "left_turn","right"],
    2: ["straight", "left", "left_turn","right"]
}
HORIZON_LENGTH = 200.0
SAVE_VIDEO = False
```

This code shows how to log vehicle attributes for `vehicle_id` 1 and 2. Also, how to show the control input as a function of time in a plot.

```
# vehicle_data_logging.py
LOG_VEHICLE_DATA = True
SHOW_PLOTS = True
VEHICLE_IDS = [1, 2]
ATTRIBUTES=["position","velocity","control_input"]
START_TIME = 0
END_TIME = float('inf')
```

```
INTERVAL = 0.1
LOG_DIR = "logs"
PLOT_ATTRIBUTES = ["control_input"]
PLOT_FILE = os.path.join(LOG_DIR, "vehicle_plot.png")
```

Together, these configurations provide a robust way to save, manage and analyze vehicle behavior during and after a simulation run.

4.2.8.5 Example Configuration To Load Project

This example shows how to load the simulation project configurations. The only parameter relevant here is the `SIMULATION_ID` and `LOAD_-PROJECT`. To load the saved simulation project configuration ensure that the `SIMULATION_ID` matches the saved `SIMULATION_ID` and that `LOAD_PROJECT = True` in code 4.2.8.4.

4.3 Running the simulator

After all the previous steps have been followed and all the configurations have been set up, the final step is to run the simulator. The simulation can be executed using the following command:

```
python main.py
```

Executing this command will build and connect all defined road segments, instantiate and place all vehicles according to the specified configurations and launch the simulation visualization as well as the road network graph.

The simulation window will visualize vehicle movement, platooning behavior and lane changes in real time.

Chapter 5

Demonstration

This chapter presents a demonstration of the simulator in action. The goal is to validate the functionality of the simulator and illustrate its features in a representative scenario.

5.1 Scenario Setup

To validate the functionality of the simulator, two representative scenarios were constructed that include multiple interconnected road segments and vehicles with varying configurations. The first and more simple simulation environment consists of seven connected road segments and four vehicles with their configurations shown in Table 5.1 and 5.2. The second and more complex simulation environment consists of 16 total road segments and nine vehicles with their configurations shown in Table 5.3 and 5.4.

Table 5.1: Road segment configuration for the simple scenario.

Segment Type	**Length** (m)	**Orientation** (deg)	**Lane Width** (m)	**Lanes**	**Radius** (m)	**Curve Direction**	**Central Ang.** (deg)
1	100	0	50	2	–	–	–
3	150	0	50	2	–	–	–
1	100	0	50	2	–	–	–
3	150	0	50	2	–	–	–
1	100	0	50	2	–	–	–
2	–	90	50	2	250	right	180
2	–	180	50	2	250	left	180

Table 5.2: Vehicle configuration for the simple scenario.

Vehicle ID	**Dynamic Model**	**Controllers Lat/Long**	**Route Instructions Straight(S), Left(L), Right(R), Turn(T)**
1	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, LT, S, S
2	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, RT, S, S, S, S, S, S, S
3	Bicycle	PID/PIDBasedACC	S, S, S, S, S, S, S, S, S
4	Bicycle	PID/PIDBasedACC	S, S, S, S, S, S, S, S, S

Table 5.3: Road segment configuration for the complex scenario.

Segment Type	**Length** (m)	**Orientation** (deg)	**Lane Width** (m)	**Lanes**	**Radius** (m)	**Curve Direction**	**Central Ang.** (deg)
1	300	200	50	2	–	–	–
3	150	180	50	2	–	–	–
2	-	0	50	2	200	left	90
2	-	30	50	2	200	right	90
2	-	60	50	2	200	right	70
1	100	180	50	2	–	–	–
3	150	280	50	2	–	–	–
2	-	250	50	2	400	left	50
2	-	90	50	2	400	left	60
3	150	280	50	2	–	–	–
2	-	0	50	2	50	right	97.2
1	811.6	0	50	2	–	–	–
2	-	0	50	2	50	right	32.8
2	-	0	50	2	50	right	100.0
1	404.3	0	50	2	–	–	–
2	-	0	50	2	50	right	20.0

Table 5.4: Vehicle configuration for the complex scenario.

Vehicle ID	**Dynamic Model**	**Controllers Lat/Long**	**Route Instructions Straight(S), Left(L), Right(R), Turn(T)**
1	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT
2	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT
3	Bicycle	PID/PIDBasedACC	S, L, LT, S, R, S, RT
4	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT
5	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT, LT, LT, LT
6	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT, LT, LT, LT
7	Bicycle	PID/PIDBasedACC	S, L, LT, S, S, S, LT, LT, LT, LT
8	Bicycle	PID/PIDBasedACC	S, L, R, S, S, S, LT, LT, LT, LT
9	Bicycle	PID/PIDBasedACC	LT, LT, S, S, S, S, LT, LT, LT, LT

These demonstrated scenarios were designed to validate the cohesive operation of the simulator's most crucial and major components. Specifically, we test all segment types, including straight, curved, and intersection roads and vehicle features such as lane tracking, lane changing, and safe inter vehicle gap keeping. By assigning different dynamic models and/or controllers to each vehicle, the scenario also verifies the correctness of control responses and ensures that perception, control, routing, and network transitions function reliably under realistic conditions.

5.1.1 Road Network Topology

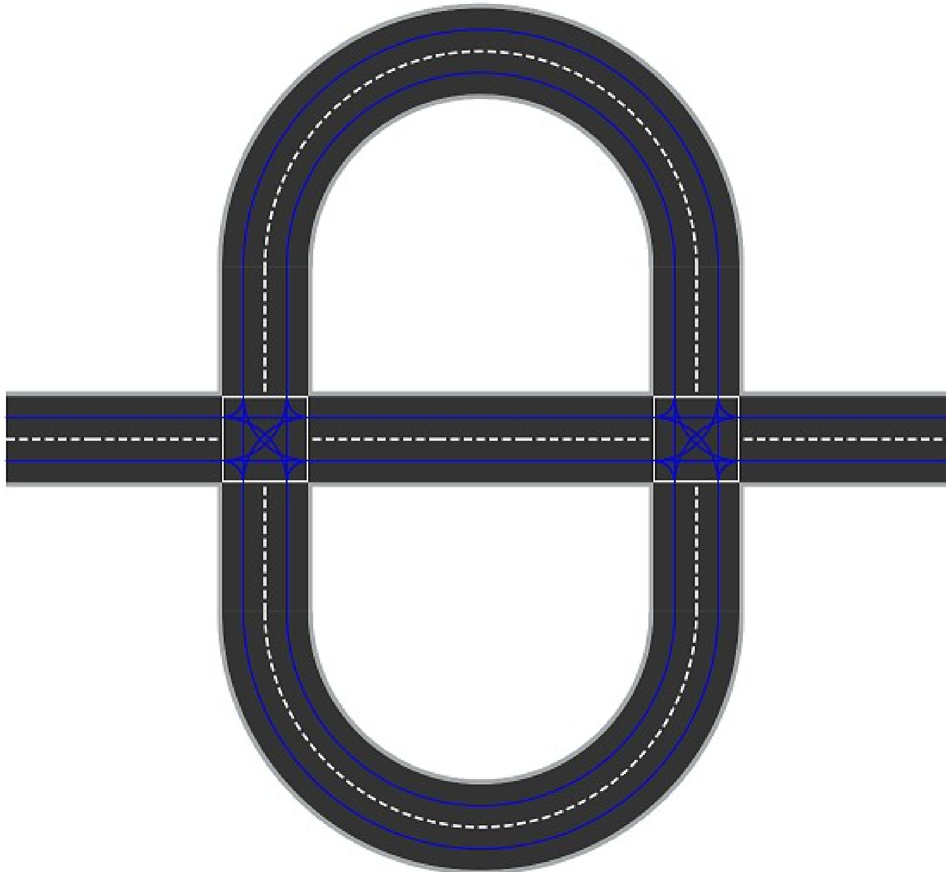

Figure 5.1: Layout of the road network used in the demonstration scenario.

The complete road network used in the first demonstration is composed of seven connected segments. The first five rows of Table 5.1 are connected manually through the segment connection interface, while the last two was generated and connected using the automatic road generation for open ends (Section 4.2.3). Two additional curved segments were appended (last two rows in Table 5.1) to form a continuous path with loops which allows various vehicle behaviors such as turning or traversing from segment to segment on different geometric road layouts.

Figure 5.1 illustrates the layout of the full road network used in the first simulation scenario. The network was manually constructed using the simulator's modular road creation and connection interface ensuring that each segment's orientation, position and connection point were correctly aligned to

enable smooth transitions between them.

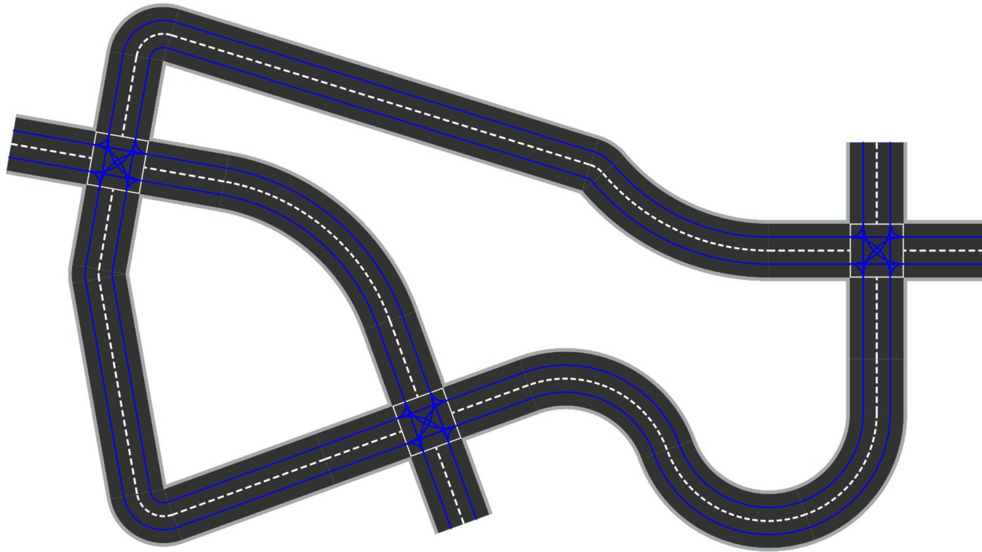

Figure 5.2: Layout of the road network used in the demonstration scenario.

The more complex road network used in the second demonstration is composed of sixteen connected segments, as shown in Table 5.3. The first ten segments are created using the road creation interface and the remaining 6 are automatically created and connected for four open ends. The connection of the open ends are configured by the user (as discussed in Section 4.2.3) to form a continuous path with loops.

Figure 5.2 illustrates the layout of the complex road network that better represents the capabilities of the simulator and demonstrates a better road scenario.

5.1.2 Simulation Parameters

Both simulations were executed with a discrete time step of 0.1 seconds and had a lookahead horizon of 200 meters for the reference trajectory generation. The maximum vehicle speed in the simple scenario was limited to 30m/s for vehicle 1 and vehicle 3 and 20m/s for vehicles 2 and 4. The maximum vehicle speed in the complex scenario was set to 35m/s for all vehicles except vehicles 1 and 5 which were limited to 25m/s and 30m/s.

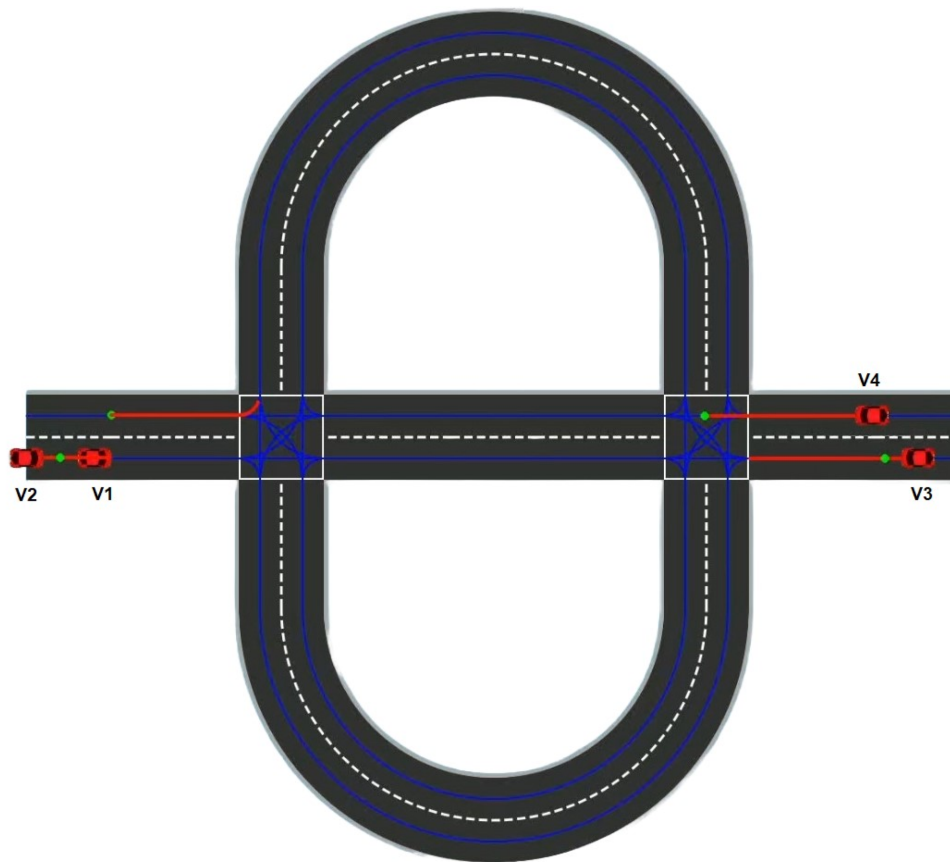

Figure 5.3: Vehicles with their respective names in the road network

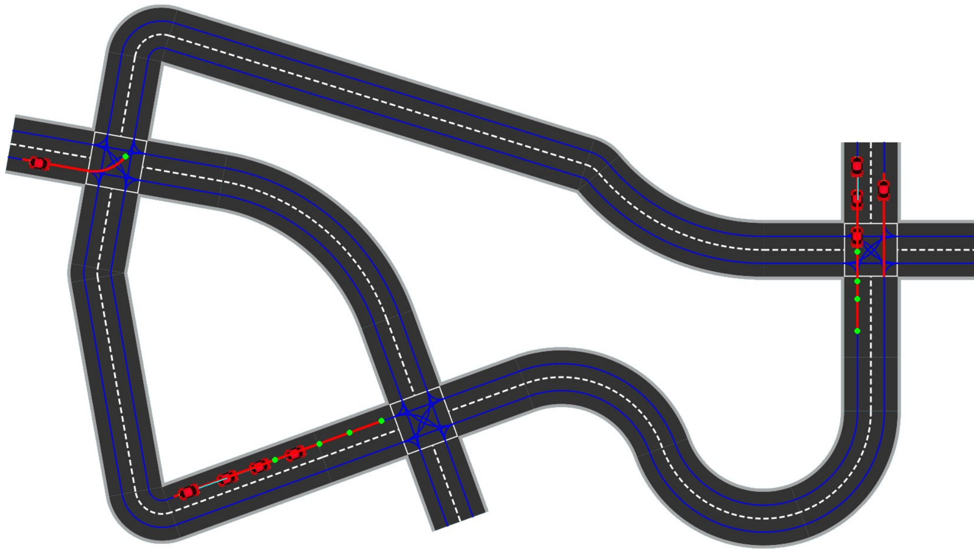

Figure 5.4: Vehicle with their respective names in the road network

In the simple scenario, vehicles were placed according to Figure 5.3 and in the complex scenario they where placed according to Figure 5.4.

5.1.3 Simulator Execution

Once the scenario was initialized, the simulation proceeded with vehicles navigating the connected road segments in accordance with their assigned dynamics, controllers and route instructions. The simulator is executed in discrete time steps, during which the following operations were performed for each vehicle:

- **State Update:** Each vehicle's position, velocity, and heading were updated using its assigned dynamic model (bicycle or unicycle). The controller computed the required control inputs (steering angle and acceleration) based on the reference trajectory and sensor feedback.
- **Lane Tracking:** The vehicles followed a reference trajectory generated by the `ReferenceTrajectoryGenerator`, which continuously gave updated paths based on the current position, route instruction, and lookahead horizon.
- **Segment Transitions:** Upon approaching the end of a road segment, the `VehicleManager` checks for available outbound connections. If a valid connection was found, the vehicle transitioned to the connected

segment, and its reference trajectory was regenerated to reflect the new topology.

- **Lane Change Execution:** Vehicles with pending lane change instructions initiate lateral movement based on spatial feasibility and segment geometry. When the lane change was completed, the vehicle's lane information was updated, and the trajectory recalculated accordingly.
- **Virtual Parking Logic:** If the vehicle reached an unconnected segment end (a dead-end/open-end), it was removed from the active road network and placed into a virtual parking lot. This mechanism supports reentry and platoon-based dispatching.
- **Road Network Re-entry Logic:** When a desired amount of vehicles (see Section 4.2.7) has entered the platoon from the road network and the platoons has no capacity for extra vehicles, the platoon is scheduled to leave the virtual parking lot and re-enter the road network.

Throughout the simulation, the system maintained deterministic behavior by ensuring that all updates and trajectory regenerations were based on observable vehicle states and route instructions, independent of any stochastic decision-making. The simulation results, including vehicle positions, orientations and control inputs can be logged for analysis and visualization as stated in Section 4.2.8.

5.1.4 Key Observations

The simulation confirmed that vehicles followed their assigned trajectories reliably across all segment types. Vehicles accurately executed turns and lane changes based on their predefined route instructions with no observed discontinuities or orientation mismatches.

The controllers performed as expected i.e., the PID-based lateral controller maintained lane alignment while the PID-based ACC longitudinal controller preserved safe following distances during transitions and turns. Lane changes were completed deterministically without major oscillations or instability.

One notable observation was the accuracy of trajectory regeneration during dynamic changes in position and heading. The system adapted to each vehicle's context seamlessly, reaffirming the robustness of the reference trajectory generation.

5.1.5 Visual Results

To complement the analysis, several screenshots were taken at critical moments during the simulation. These include:

- Vehicles entering the curved road from a straight road (transition) (Figure 5.5).
- Vehicles making a left and right turn at the intersection (Figure 5.6).
- Multiple vehicles navigating parallel lanes (Figure 5.7).
- A vehicle performing a lane change maneuver (Figure 5.8, 5.9, 5.10).
- Vehicles just before and after entering the virtual parking lot (Figure 5.11).
- Vehicles re-entering the road network as scheduled (Figure 5.12) after entering the virtual parking lot (previously seen in Figure 5.11).

These visual outputs provide clear confirmation that vehicle behavior matched expectations, segment transitions were successful, and control systems operated as designed. A video demonstration of the simulator, showcasing the vehicle dynamics and controller behavior during various scenarios, is available online.*

*https://www.youtube.com/watch?v=7Ef_6DNhcoE

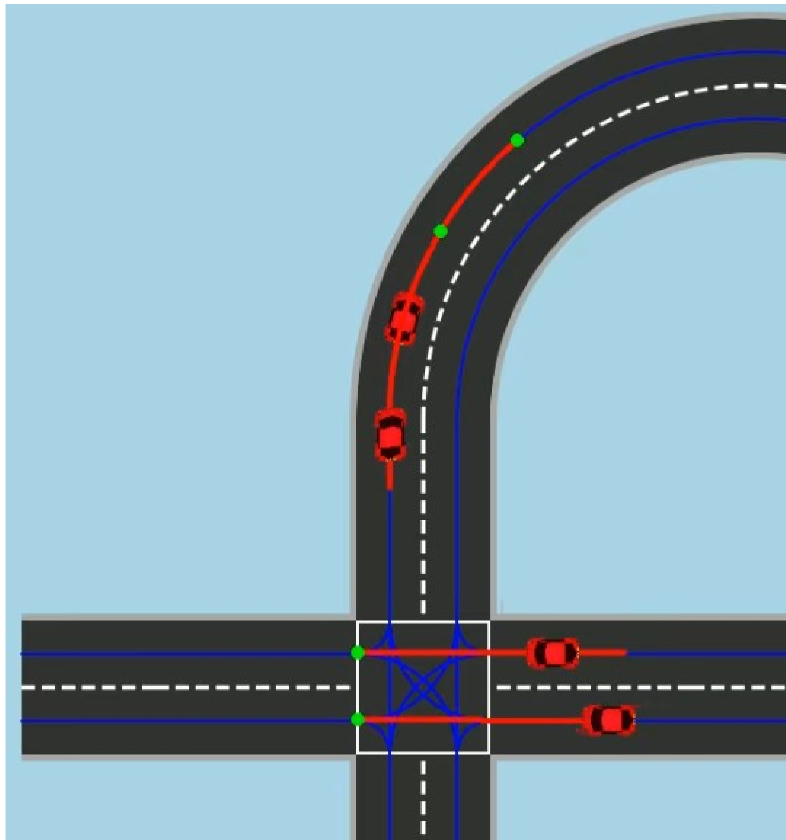

Figure 5.5: Vehicles entering a curved road from a straight segment (seen in the upper portion of the image).

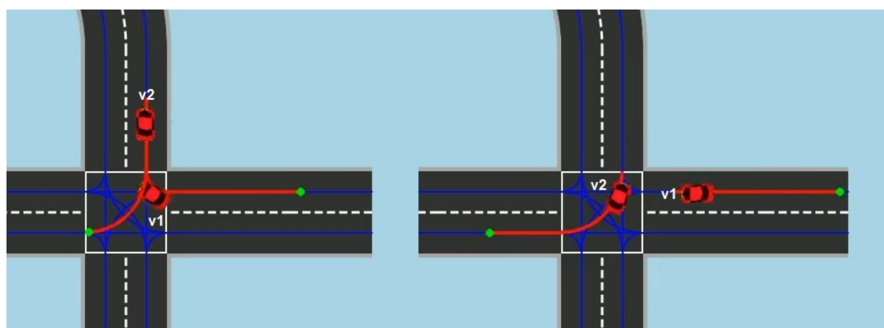

Figure 5.6: Vehicle 1 executing a left turn in the left image, Vehicle 2 doing a right turn in the intersection in the right image.

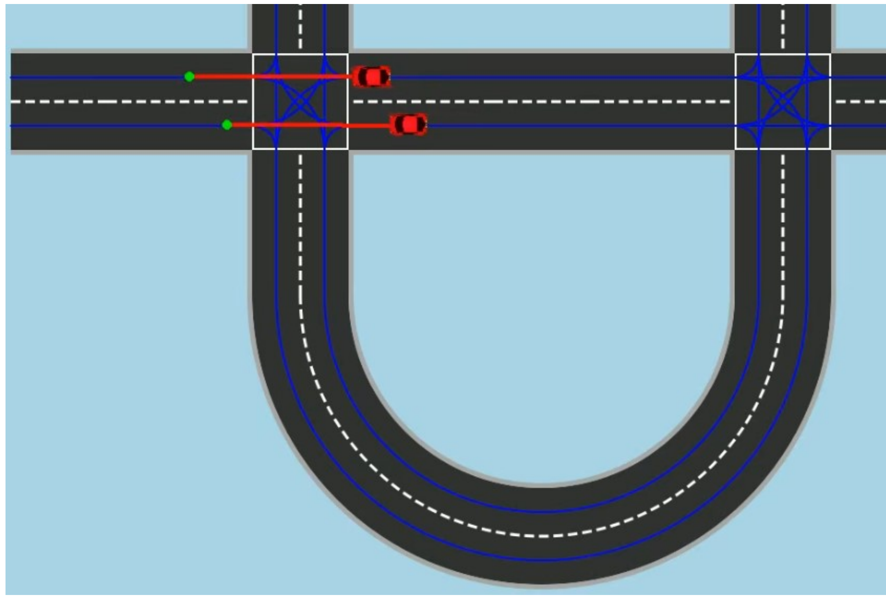

Figure 5.7: Multiple vehicles navigating parallel lanes.

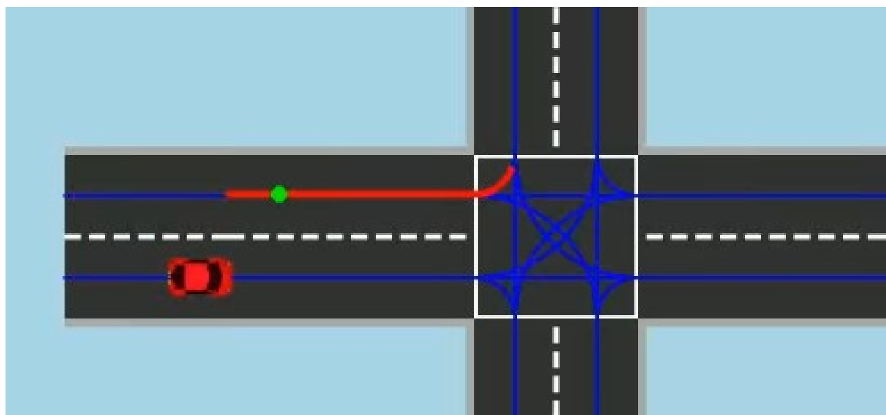

Figure 5.8: Vehicle before performing a lane changing maneuver.

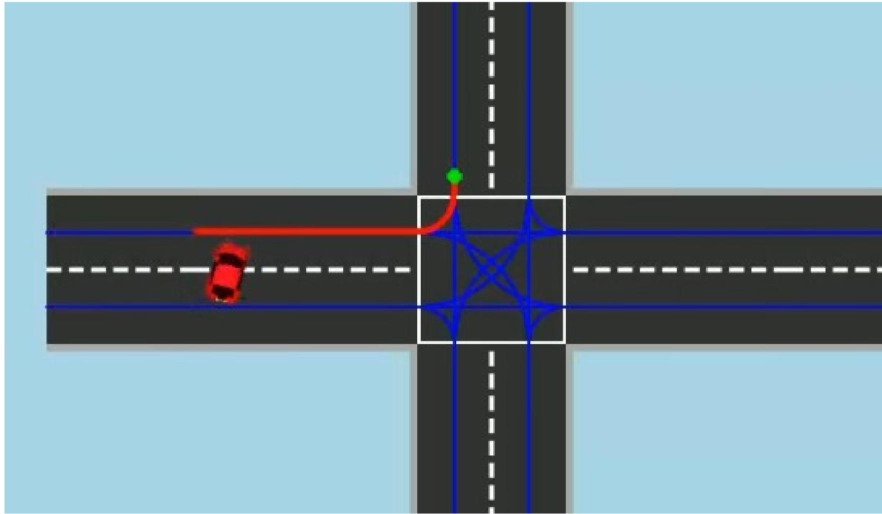

Figure 5.9: Vehicle during a lane change maneuver being executed.

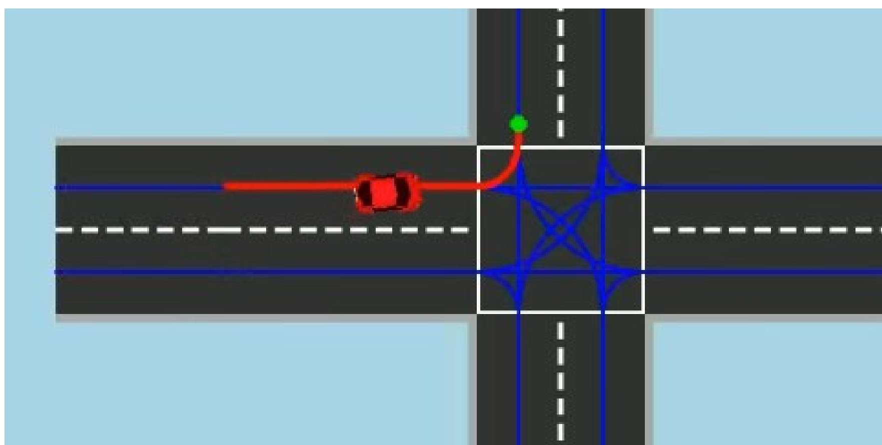

Figure 5.10: Vehicle after completing a lane changing maneuver successfully.

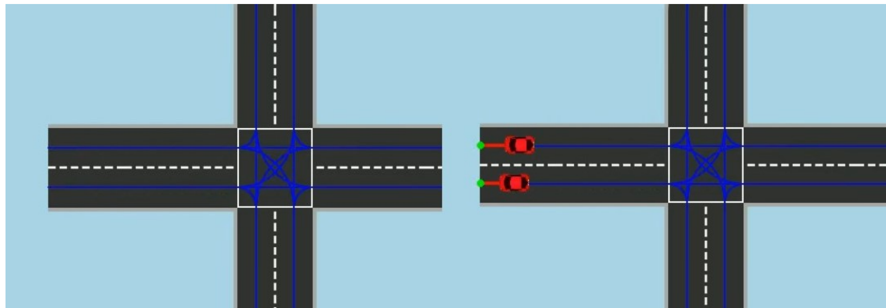

Figure 5.11: Vehicle just before entering the virtual parking lot (right image), Vehicle after entering the virtual parking lot (left image).

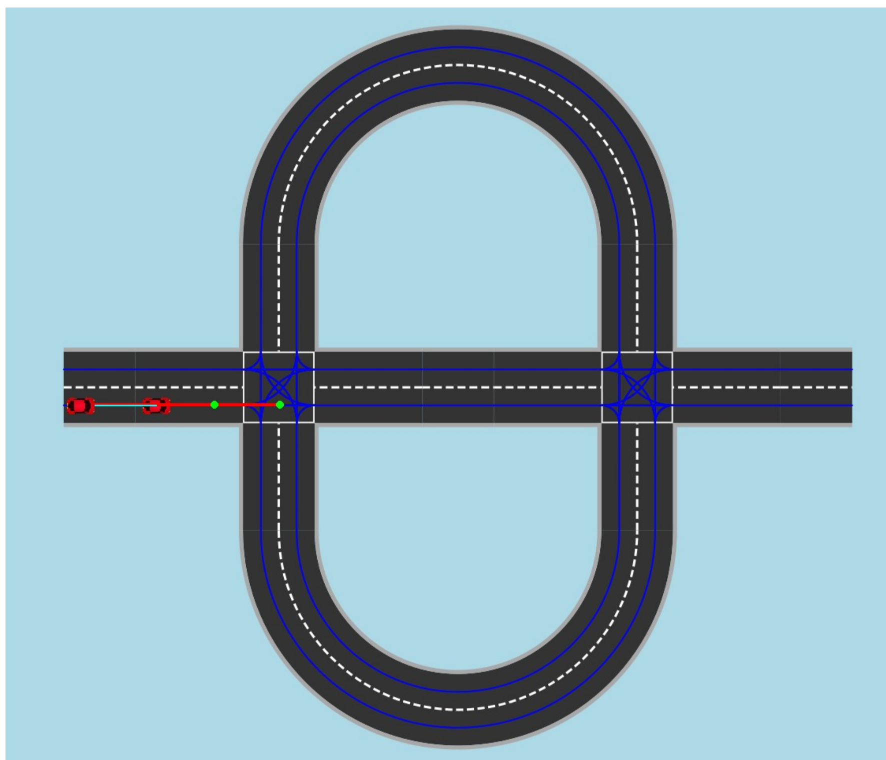

Figure 5.12: Vehicle just before re-entering the road network

5.1.6 Summary

The demonstrated scenarios verified the simulator's ability to perform all essential tasks required for autonomous vehicle coordination. They have illustrated the successful road creation and connection, the automatic road generation of open ends, vehicle instantiation, route instruction following,

dynamic trajectory regeneration, and seamless cross-segment transitions. The results affirm the simulator’s robustness and suitability for evaluating advanced vehicle coordination strategies in controlled virtual environments.

Chapter 6

Conclusions and Future work

6.1 Conclusions

This thesis presents a modular vehicle platooning simulator, which was designed, implemented and validated, with the aim to evaluate autonomous vehicle behavior in structured traffic environments. The simulator supports extensible road networks, dynamic vehicle instantiation, and flexible integration of control and dynamic models.

Chapter 3 detailed the architectural design of key components such as the `RoadSegmentManager`, which handles the creation and connection of modular road segments using factory and strategy patterns. In addition, the `VehicleManager` handles the configuration and control of autonomous vehicles equipped with various dynamics and controller models.

Chapter 4 provides a practical step-by-step guide on how to configure and use the simulator. It covers the configuration process for the setup of a road network and vehicles as well as how the simulation can be executed, saved, and reloaded.

The functionality of the simulator was demonstrated in Chapter 5, where vehicles navigated a connected road network composed by straight, curved and intersection segments. The simulation confirmed reliable trajectory following, seamless segment transitions and robust handling of lane changes, vehicle virtual parking lot entry and road network re-entry. Observations showed that the modular and extensible design allowed for deterministic and predictable vehicle behavior under a variety of conditions.

The simulator meets its main objectives of modularity, testability, and extensibility. It provides a controlled environment for validating control strategies and dynamics while maintaining a clear separation of concerns

between road network logic, vehicle control, and simulation flow.

6.2 Outlook

There are several areas that remain open for future development and refinement, these include:

- **Smooth Lane Change Trajectories:** The current lane change behavior is implemented as a discrete lateral shift, lacking a continuous trajectory model when lane changes are introduced. While it's functional for simple maneuvers, this can result in harsh dramatic turns and can lead to unpredictable vehicle behavior when performing multi-lane changes, specifically when operating at high speeds. Future implementations should incorporate trajectory smoothing techniques such as smoothing the trajectories with Bézier curves or cubic splines to generate dynamically feasible and kinematically consistent lane change paths.
- **Modeling of Sensor Noise and Control Delays:** To increase realism, the system could incorporate sensor noise, delay, and communication latency. This would enable better evaluation of control algorithms under real-world uncertainties where perception is critical.
- **Larger selection of Road Segments** A larger selection of road segments such as different types of intersection roads, roundabouts and on- and off-ramp merging segments.
- **On-ramp/off-ramp merging:** The current implementation does not support complex merge scenarios. Extending the simulator to handle multi-entry ramps or highway merge logic would improve its real-world applicability.
- **V2V and V2I communication:** Incorporating vehicle-to-vehicle and vehicle-to-infrastructure communication models would enable cooperative maneuvers and decentralized control schemes.
- **Scenario configuration interface:** A graphical user interface (GUI) or scriptable scenario definition layer would simplify the process of creating and testing new road layouts and vehicle behaviors.

- **Lane-based graphs**: Implementing a graph where each node corresponds to a lane, and each edge represents a permissible lane-to-lane transition (either a successor or an adjacent lane) would enable higher-level route planners to dynamically find a goal position using a graph search algorithm.
- **Collision avoidance**: Implementing a local planner that works with a higher-level route planner could extend this simulator to simulate collision avoidance objectives. This would allow vehicle's to re-route around crashed vehicles and overtake slow vehicles.

References

[1] S. Tsugawa, S. Jeschke, and S. E. Shladover, “A review of truck platooning projects for energy savings,” *IEEE Transactions on Intelligent Vehicles*, 2016.

[2] Singh Santokh, “Critical reasons for crashes investigated in the national motor vehicle crash causation survey,” National Highway Traffic Safety Administration, Traffic Safety Facts Crash•Stats DOT HS 812 115, 2015.

[3] C. K. Verginis, C. P. Bechlioulis, D. V. Dimarogonas, and K. J. Kyriakopoulos, “Robust distributed control protocols for large vehicular platoons with prescribed transient and steady-state performance,” *IEEE Transactions on Control Systems Technology*, 2018.

[4] M. Charitidou and D. V. Dimarogonas, “Splitting and merging control of multiple platoons with signal temporal logic,” in *2022 IEEE Conference on Control Technology and Applications (CCTA)*, 2022.

[5] M. Strunz, J. Heinovski, and F. Dressler, “CoOP: V2v-based cooperative overtaking for platoons on freeways,” in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021.

[6] A. Frauenfelder, A. Wiltz, and D. V. Dimarogonas, “Decentralized vehicle coordination and lane switching without switching of controllers,” *IFAC-PapersOnLine*, 22nd IFAC World Congress, 2023.

[7] J. Rios-Torres and A. A. Malikopoulos, “Automated and cooperative vehicle merging at highway on-ramps,” *IEEE Transactions on Intelligent Transportation Systems*, 2017.

[8] S. Eilers *et al.*, “COMPANION – towards co-operative platoon management of heavy-duty vehicles,” in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, 2015.

[9] C. Krupitzer, V. Lesch, M. Pfannemüller, C. Becker, and M. Segata, "A modular simulation framework for analyzing platooning coordination," in *Proceedings of the 1st ACM MobiHoc Workshop on Technologies, MOdels, and Protocols for Cooperative Connected Cars*, Association for Computing Machinery, 2019.

[10] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017.

[11] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, *CARLA: An open urban driving simulator*, 2017. arXiv: 1711.03938[cs].

[12] J. Tani *et al.*, "Integrated benchmarking and design for reproducible and accessible evaluation of robotic agents," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[13] S. F.Lott and D. Phillips, *Python Object-Oriented Programming - Fourth Edition*. Packt Publishing, 2021.

[14] E. Semenova, V. Tynchenko, S. Chashchina, V. Suetin, and A. Stashkevich, "Using UML to describe the development of software products using an object approach," in *2022 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, 2022.

[15] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer International Publishing, 2015.

[16] Yanpas, *UML class relations (english)*, 2016. [Online]. Available: https://commons.wikimedia.org/wiki/File:Uml_classes_en.svg.

[17] C. Giridhar and G. Zlobin, *Learning Python Design Patterns - Second Edition*, 2nd ed. Packt Publishing, 2016.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, Mar. 2016.

[20] A. Demba and D. P. F. Möller, "Vehicle-to-vehicle communication technology," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018.

[21] R. Q. Malik, K. N. Ramli, Z. H. Kareem, M. I. Habelalmatee, and H. Abbas, "A review on vehicle-to-infrastructure communication system: Requirement and applications," in *2020 3rd International Conference on Engineering Technology and its Applications (IICETA)*, 2020.

[22] J. Choi, V. Marojevic, C. B. Dietrich, J. H. Reed, and S. Ahn, "Survey of spectrum regulation for intelligent transportation systems," *IEEE Access*, 2020.

[23] Y. Dai, Y. Yang, H. Zhong, H. Zuo, and Q. Zhang, "Stability and safety of cooperative adaptive cruise control vehicular platoon under diverse information flow topologies," *Wireless Communications and Mobile Computing*, 2022.

[24] J. Wang, Y. Zheng, C. Chen, Q. Xu, and K. Li, "Leading cruise control in mixed traffic flow: System modeling, controllability, and string stability," *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[25] A. Hagberg, P. Swart, and D. Chult, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, 2008.

TRITA-EECS-EX-2025:954
Stockholm, Sverige 2024

www.kth.se