POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

# AR-Sim: A High-fidelity 3D Simulator for Autonomous Racecars

Supervisors

Prof. Andrea TONOLI

MSc Felix JAHNCKE

MSc Eugenio TRAMACERE

Candidate

Giovanni SCAPICCHI

July 2024

## Abstract

This thesis introduces a versatile simulation environment for autonomous racing vehicles, developed at the TUM Autonomous Vehicle System Lab. Initially created for F1Tenth cars, the simulator is adaptable for any vehicle type, including full-scale models. It feature a C++ API and a custom Unity executable for realistic graphics and sensor simulations. A ROS2 extension package is included for easy software-in-the-loop testing, while a Python wrapper implements a Gymnasium environment for machine learning algorithms.

The project not only addresses the limitations of the current F1Tenth gym simulator, which is confined to 2D environments and lacks capabilities to include camera and 3D LiDAR sensors, but overcome the lack of open source autonomous racing simulators for vehicles. The existing simulators are few and with limited generalizability, resulting in a trade off between customization of vehicle dynamic modeling and 3D realism. On the other side autonomous road vehicle simulation environments present advanced capabilities but are very complex and demand substantial computational resources, in addition none of them allow fully customization of the vehicle dynamic model employed and mostly only rely on the PhysX physics engine.

This new simulator bridges the gap with unique features, offering both PhysX and external physics engines, and provides an easy-to-use API for custom dynamic models. User-friendly YAML configuration files enable seamless setup of the simulation environment. Custom race tracks can be integrated from pre-made 3D models, CSV files, or generated randomly using varied barriers and materials. The simulator also features a LiDAR model and an RGB camera sensor, both of which are fully parameterized to match real sensor specifications. Multiple sensors can be placed on a car, with the option to include more than one of each type.

This open-source simulator is a valuable tool for research and education, especially in the F1Tenth community, and its flexibility makes it suitable for a broad spectrum of autonomous racing vehicle applications.

# Acknowledgements

I am thankful to everyone who supported me throughout my thesis journey. First and foremost, I owe a huge debt of gratitude to Felix Jhancke for welcoming me into the AVS lab at TUM for his constant trust and support, and for giving me the freedom to pursue my ideas on this project. A special thanks also to Professor Johannes Betz, for giving me the opportunity to be part of their research team.

Additionally, I'd like to thank Daniel Gebhart and Yichao Gao, who collaborated with me on the project, for their assistance and teamwork.

I also want to extend my appreciation to Professor Tonoli for allowing me to take on this project and to Eugenio for his advice and assistance.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**AI** Artificial Intelligence

**RL** Reinforcement Learning

**E2E** End-to-End

**IAC** Indy Autonomous Challenge

**AV** Autonomous Vehicles

**AD** Autonomous Driving

**ODE** Ordinary Differential Equation

**GSS** Ground Speed Sensor

**SIL** Software In the Loop

**HIL** Hardware In the Loop

**ML** Machine Learning

# Chapter 1

# Introduction

Autonomous driving technology is revolutionizing the transportation sector, offering significant benefits in terms of safety, efficiency, and accessibility. Autonomous racing competitions have emerged as a driver for advancements in this technology and as a means to increase public acceptance by showcasing its capabilities.

In academia, the field of autonomous driving is expanding with new, accessible platforms like F1Tenth that provide valuable testbeds for prototyping new solutions. In these applications, simulation environments are crucial tools that enable safe, cost-effective development and testing.

In this context, this thesis explores the development of an autonomous racing simulator designed to enhance training and research for autonomous race vehicles.

## 1.1 Background

Autonomous cars are expected to be one of the biggest revolutions in transportation systems, the benefits will concern safety, traffic efficiency, and accessibility to transportation, at the same time there are consequences not yet sure what will be the impact on the job market or from the environmental point of view [1], [2]. Autonomous driving (AD) is a rapidly evolving technology where a lot of effort is displayed by researchers and companies to solve the present questions concerning the complexity of the technology. Anyway, autonomous driving is no longer a fiction, with several companies already providing shared mobility solutions with robo-taxi as Waymo, and Cruise in the U.S. already available, starting to become a real business [3]. There are also cars with up to level 3 autonomous driving, such as the Mercedes EQS, available to the customers. The economic perspective is also very significant with a projected value on the global market of $615 Billion by 2026 [4]. This chapter aims to give an overview of autonomous driving technology and contextualize this thesis in respect to trends in this sector. The last paragraph of this chapter outlines the structure of the rest of this writing.

## 1.2 Level of Autonomous Driving

Autonomous driving is divided according to SAE into 6 levels going from 0-5 based on the level of automation the system provides. Most commercial ADAS (advanced driving assistance systems) on vehicle fall between categories 0-3. The six levels are:

- **Level 0**: No driving automation, automation, if any, is only present for warning purposes.
- **Level 1:** Driver Assistance.The vehicle features a single automated system for example adaptive cruise control. There can't be at the same time later and longitudinal control.
- **Level 2**: Partial driving automation. The vehicle can control both lateral and longitudinal dynamic at the same time. It required active driver supervision. Tesla Autopilot is currently on this level.

- **Level 3**: Conditional Driving Automation.The vehicle can perform all driving functions but only under certain conditions and the system may request the driver to take control when does conditions are no longer met. Some companies providing level 3 autonomy are Audi, Honda, and Mercedes-Benz.

- **Level 4**: High Automation. The vehicle can operate completely autonomously in specific conditions or environments differently from level 3 no request is made to the driver to take control. At this level though vehicles can only operate in specific environments. Examples of this are robotaxi services like Waymo.

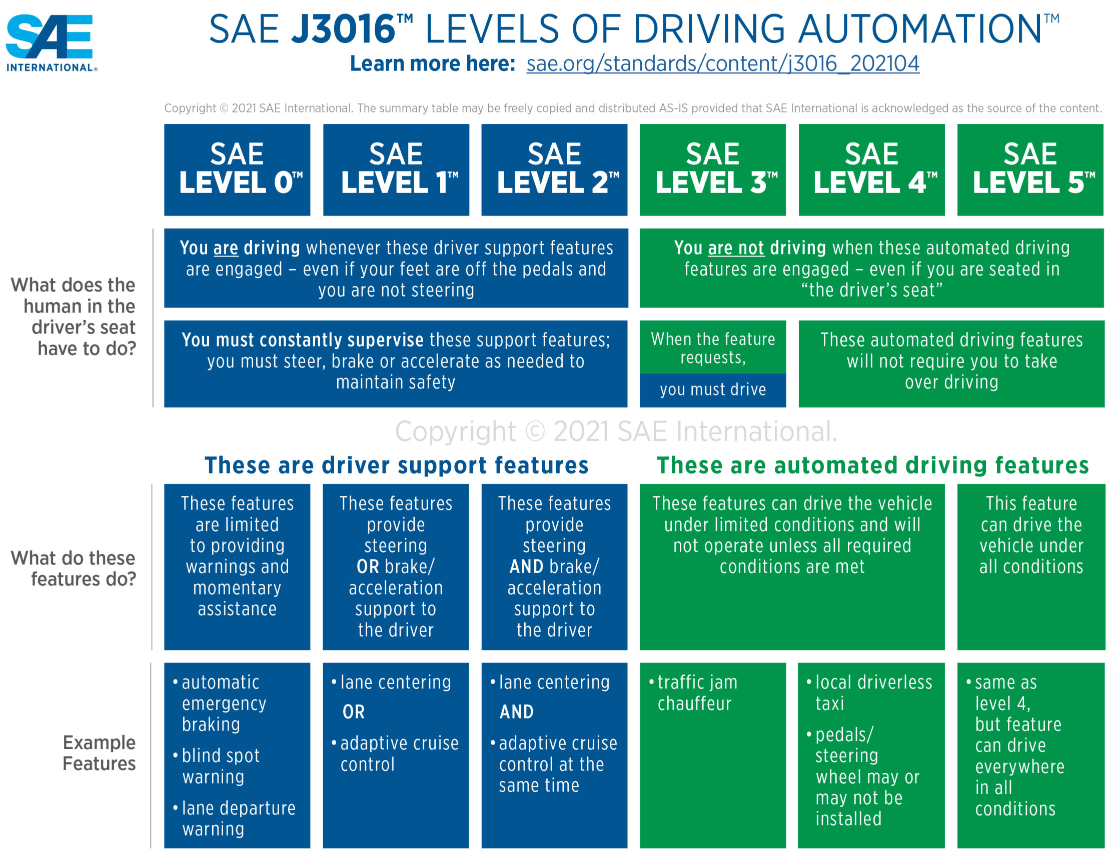
- **Level 5**: Full automation.The vehicle can drive by itself in every scenario or location.

## 1.3 Impact of Autonomous driving

Autonomous Driving will impact several areas of transportation systems solutions in a deeper way but will also affect and shape our society differently, briefly in this chapter are presented some of the most relevant consequences, outlining both positive but also possible negative ones.

**Safety -** Safety will be one of the major positive influences of autonomous driving, every year the causalities due to accident involve millions of people worldwide, and more than 90% of those accidents are due to human errors. More than half of all road accident deaths involve young adults ages 15-44 and are the leading cause of death for young people ages 15-29. [1]

**Accessibility -** Autonomous vehicles can increase the accessibility to transportation to physically disabled people or people who don´t possess a license, giving them more independence.



SAE **J3016**™ LEVELS OF DRIVING AUTOMATION™

**Learn more here:** sae.org/standards/content/j3016_202104

| | SAE LEVEL 0™ | SAE LEVEL 1™ | SAE LEVEL 2™ | SAE LEVEL 3™ | SAE LEVEL 4™ | SAE LEVEL 5™ |
|---|---|---|---|---|---|---|
| What does the human in the driver's seat have to do? | **You are driving** whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering | | | **You are not driving** when these automated driving features are engaged – even if you are seated in "the driver's seat" | | |
| | **You must constantly supervise** these support features; you must steer, brake or accelerate as needed to maintain safety | | | When the feature requests, you must drive | These automated driving features will not require you to take over driving | |
| | **These are driver support features** | | | **These are automated driving features** | | |
| What do these features do? | These features are limited to providing warnings and momentary assistance | These features provide steering **OR** brake/ acceleration support to the driver | These features provide steering **AND** brake/ acceleration support to the driver | These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met | | This feature can drive the vehicle under all conditions |
| Example Features | • automatic emergency braking<br>• blind spot warning<br>• lane departure warning | • lane centering<br>**OR**<br>• adaptive cruise control | • lane centering<br>**AND**<br>• adaptive cruise control at the same time | • traffic jam chauffeur | • local driverless taxi<br>• pedals/ steering wheel may or may not be installed | • same as level 4, but feature can drive everywhere in all conditions |

**Figure 1.1:** SAE Level of Autonomous driving

**Traffic Efficiency -** Traffic congestion can be reduced thanks to the use of autonomous driving since this technology allows cars to better cooperate and make them respect speed limits, safety distance and also the increased use of car sharing and carpooling can increase the per-car vehicle occupancy while reducing the number of vehicles parked on the road. V2X communication could also allow experimenting with technology like platooning, and efficient routing since cars could communicate with each other and with the infrastructure to optimize the path taken for going to the destination based on the current state of the roads.[2]

**Environmental Impact -** It's complex to predict the environmental impact of this, while on one side the functionalities of Autonomous cars can reduce GHG emissions thanks to the more efficient use of the vehicles, on the other side the increased facility and ease of access to transportation could increase the number of people traveling by cars. Also, cars could go faster on highways due to the increased safety causing an increase of fuel consumption as high as 40% or more [2]. On the other side, they can provide a more tailored on-demand service that can substitute public transportation in rural areas [5].

**Employement Impact -** The commercial vehicle sector of taxis and trucks are expected to be using AV technology since this will allow to transport more people and goods at a lower cost, this implies that professional drivers will be among the first to become unemployed [6]. New unexpected jobs will also emerge as it happened with web-based services like Amazon, but this is indeed complex to predict. Anyway [6] found that those who profit from the newly available jobs aren´t usually the same as the ones losing it.

**Mobility on Demand -** Several studies report that privately owned cars can be replaced with a single robotic taxi in cities, but also in rural areas where access to mobility is more restrictive, can be improved with mobility-on-demand services where, for examples, train from point A to B can be replaced with multiple robotaxis.[5]

**Shared Mobility -** Autonomous vehicle technology can enhance shared mobility by offering extensive service coverage without necessitating a large fleet. This is because autonomous vehicles can reach users directly, eliminating the need for users to walk several minutes to access a vehicle [7], [8].

## 1.4 Autonomous racing

Autonomous racing provides a unique and demanding environment that is pushing forward the development of autonomous vehicle (AV) technologies. This field not only advances AV technology but also increases public acceptance by showcasing the capabilities and reliability of autonomous systems in complex environments [7]. Additionally, autonomous racing introduces a new type of motorsport focused on engineering and innovative strategies, attracting a technology-enthusiast audience and inspiring future advancements. [9].

### 1.4.1 Historical development

To advance the research on the AD field autonomous racing competitions emerged starting in the 21 century, as happened with the traditional race competition, which serves as an innovation laboratory for commercial vehicles. In 2004 the first DARPA challenge held place with a prize of $1M for the car able to navigate 142 miles through the Mojave desert. No one of the 107 times enrolled in the competition was able to perform more than 5% of the race. In 2005 the second competition, with a doubled prize of $2M took place, this time Standford's robot "Stanley" finished the race [4],[10]. That challenge was an occasion to push the state of the art of autonomous driving to a new level.

The increased interest in the field in AV, fueled also by companies like Google launching its research on autonomous driving in 2009, and traditional car manufacturing introducing the use of Advance Driver Assistance Systems (ADAS), led to the establishment of new autonomous racing categories. In 2017, the Formula Student Driverless category was introduced, enabling lots of students and researchers to get involved in the field and produce new advanced results, such as the software stack developed by the AMZ team

from Zurich [11]. Roborace, the first global championship for autonomous racing, took place on different occasions from 2016 to 2021. All teams were equipped with the same hardware and the competition was based only on software development. The Roborace was initially tested during Formula E events during the 2016-2018 season, in 2018 it was the first autonomous car to complete the Goodwood Hillclimb. Roborace was fatigued to find success, after trying to integrate some metaverse functionalities to have augmented reality races it ceased to exist in 2022. In 2019 started the Indy Autonomous Challenge competition where 9 teams from 21 universities participated [12]. The first race of the IAC was at Indianapolis, here the scope of the competition was reduced to a time trial event with an obstacle avoidance test, on this occasion the $1M prize was wined by the TUM autonomous Motorsport Team. In 2022 the second IAC took place at Las Vegas Motor Speedway as the final event of the 2022 edition of the Consumer Electronics Show (CES). This was the first time two full scale autonomous racing vehicles raced against each other on track even if the race event was simplified with respect to a normal competition, in this occasion the Polimove team from Milan won the competition. The Indy Autonomous Challenge was held on an oval race track until the new challenge was the 2023 race at the Autodromo Nazionale di Monza, due to the increased complexity of the road circuits this race was once again based on time and only one vehicle at a time run on the track.
With a different intention the the new A2RL (autonomous racing league), aims to have wheel-to-wheel races like normal race car competitions. The first of this event took place in 2024 at the Yas Marina Circuit where multiple autonomous racing cars compete against each other racing on the track at the same time. On another occasion the was an event where for the first time a human-driven car and AI-driven car ran on the same track at the same time. The winner of the multi-agent race was obtained by the TUM autonomous racing team.

## 1.5 Scaled research Test Beds

Full-scale autonomous vehicles are anyway quite expensive and so are outside the reach of most researchers and students. Also testing new algorithms on those vehicles can be quite risky, for this reason, scaled autonomous vehicles are gaining a lot of interest since they allow researchers to exploit new ideas without the risk of damaging costly hardware or having to need large space to test the newly designed software. A different number of examples of this are present in academia.

### 1.5.1 1:43 Vehicles

The ORCA project [13] from ETH Zurich, is an internal 1:43 scaled vehicles testbed used to research advanced controllers for race cars, such as MPC [14], or reinforcement learning application [15]. Though was made using a completely different perception stack with respect to 1:1 vehicle since perception was carried out with external capturing motion systems, which also make them very expensive.

### 1.5.2 Low Budget

Instead, low budget (400-600$) scaled autonomous test-bed with exteroceptive sensors mounted on the car is the Amazon AWS DeepRacer [16] or the Donky car [17]. Both of these cars mount a camera sensor to experiment with reinforcement learning agents able to drive the car on different race tracks and propose extensive documentation and a 3D simulation environment.

### 1.5.3 Scaled Urban Driving

Other small race cars instead focuses more on road applications like the Duckietown [18] or the Autodrive Nighel platform [19]. This car is thought to research the theme of urban autonomous driving using scaled versions of city scenarios like intersections to teach and research.

### 1.5.4 1:10 Advanced Sensor Suite

In the last category are 1:10 scaled vehicles with more high prices that are aiming to develop autonomous racing platforms with more expensive and performing equipment. Different institutions have developed and documented 1:10 scale remote-controller cars converted to autonomous vehicles. Berkeley Autonomous Racecar [20], the MIT Racecar [21], the RoSCAR [22], and the F1TENTH vehicle [23]. These vehicles feature interchangeable sensor setups, allowing the use of monocular cameras (e.g., Raspberry Pi, OpenCV OAK-1), stereo cameras (e.g., ZED, Intel Realsense), 2D LiDARs (e.g., Hokuyo models), IMUs, indoor GPS, and wheel speed sensors. They utilize embedded GPU systems like Nvidia Jetson (TX1, TX2, NX, AGX Xavier, Nano) for efficient deep neural network inference, facilitating advanced autonomous driving experiments.

#### F1Tenth

Being an affordable, open-source, and integrated autonomous vehicle test-bed the F1Tenth vehicle stands out as the one becoming more popular with already 20+ universities worldwide using it and with annual competitions that take place around the globe. Also, the F1tenth is employed for teaching practical courses on autonomous driving [24] .F1Tenth provides open-source documentation of both the Hardware and software setup that explains how to build the car step by step, and this guide is actively maintained by a community of users, besides different open source project can be found, as the [25] which provide a full stack autonomous racing software that user can directly install on the car. The F1Tenth also is fully standardized to make use of the ROS2 framework and its associated libraries.

The vehicle's chassis is based on a fast radio-controlled electric vehicle, the Traxxas 4x4, which can reach up to 100km/h and provides an Ackerman steering mechanism, for the computational side it mounts an NVIDIA Jetson TX2 GPU computer which allows for processing all the data coming from the sensor ob-board.

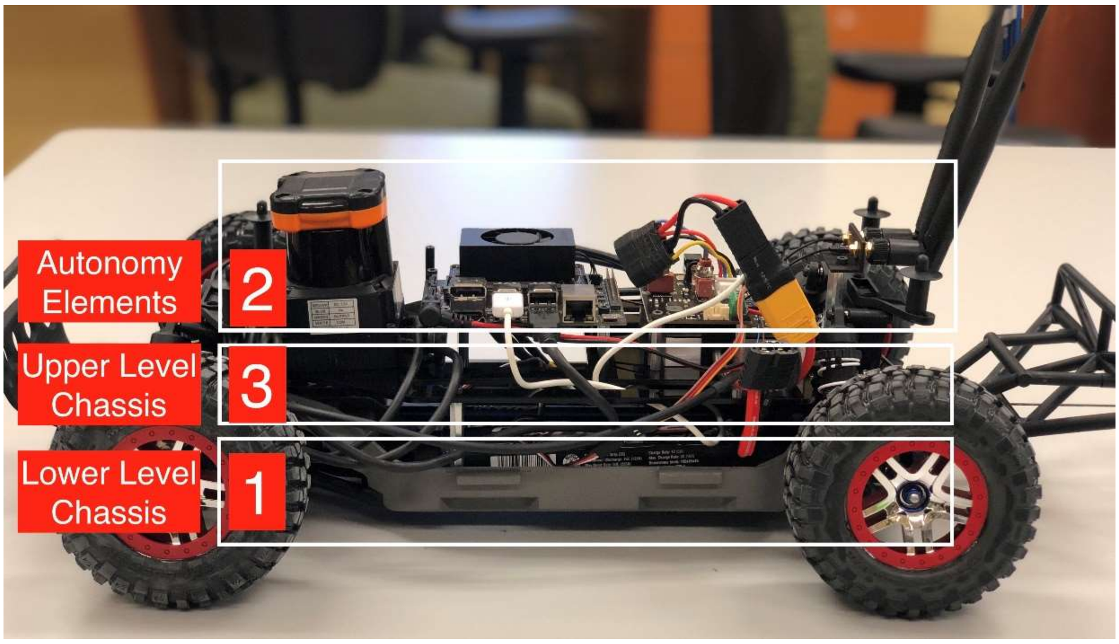


**Figure 1.2:** F1 Tenth car `https://f1tenth.org/build.html`

## 1.6 Autonomous racing software stack

The classical approach for autonomous driving that emerged is to divide the software into different modules, each targeting a different task, namely perception, planning, and control. More specifically:

- **Perception**: consists of using data coming from sensors like Cameras, LiDARS, and Radar to understand the environment around the car, such understanding aims to identify where there are obstacles and where is free space allowed to travel. Given the high dimensionality of the sensor measurements, in this step are found several AI ML techniques like the Yolo for analyzing camera images or the DBSCAN. The perception step can also include the localization and mapping which is usually performed with a SLAM (simultaneous localization and Mapping) algorithm.

- **Planning**: the goal of this module is to compute the path that the car should follow based on the understanding of the environment built on the previous step. In global planning, an optimal path is constructed based on the knowledge of the full racetrack. This method allows to build the optimal race line but it's only possible if a full map of the track is available, further, it doesn't consider other vehicles or dynamic obstacles that are present. Local planning compensates for this by providing a finite time horizon path to follow based on the current observation available. In combination with global planning, this allows for both an optimal and safe path to follow. Behavioral planners run on top of the previous two and deal with high-level decisions like if performing an overtake maneuver. The output of the planner is usually a path and a velocity profile to follow.

- **Control**: This last module is responsible for making the vehicle follow the desired trajectory by reducing the lateral and heading error with respect to the reference provided while at the same time keeping the velocity target. The control actions at a higher level are steering and throttle commands, which then are passed to lower-level controllers. A lot of different control methods can be used, from classic control theory to model predictive control and learning-based approaches.

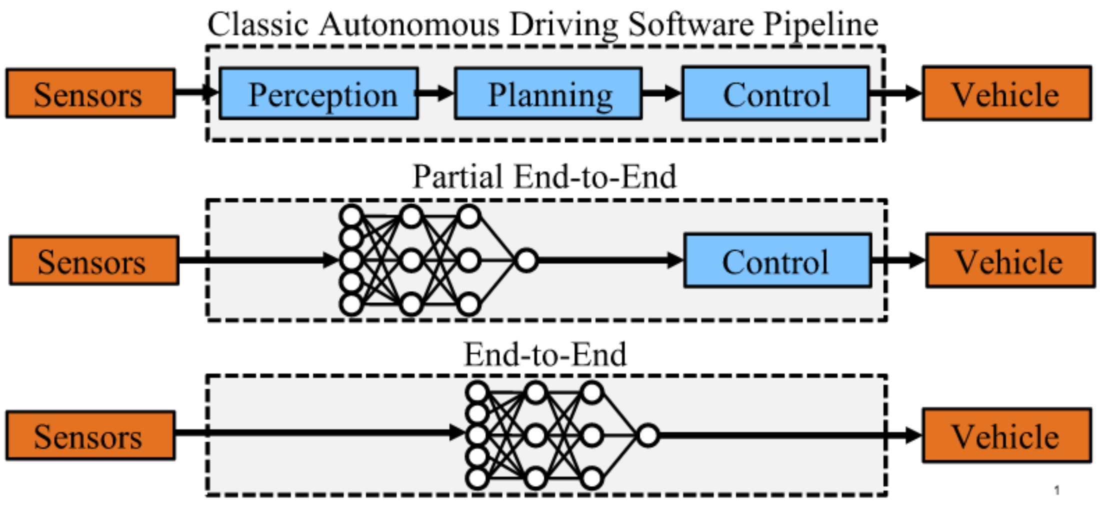


**Figure 1.3:** AV Software Pipeline [12]

## 1.6.1 End to End autonomous driving

The traditional modular pipeline is advantageous due to its interpretability and ease of debugging since each task is separated from the other. However, a negative effect of this is that the solution obtained can be sub-optimal since each software stack is optimized with respect to a different target and not for the overall unique target [26].

For this, an always increasing interest is a new approach where the above modular design is replaced, totally or partially, by a neural network in the so-called End-to-End approach or Partial End-to-End 1.3. This method processes directly the raw input data to produce motion plans or low-level control actions [27]. The End-to-End approach it's appealing because it can produce a more optimal solution and efficient use of the computing resources even though this comes with other downsides, which are less interpretability and the need for a lot of data to learn and which often require the setup of sophisticated simulation environments [28]. Training in simulation has the advantage of producing a large amount of data at a low cost and it

guarantees to test a lot of edge case scenarios, on the other side a problem when using synthetic data is that often the simulated environment lacks sufficient realism and the policy learned can't transfer to real-world [29]. This problem is known as Sim-to-Real (S2R) transferability [30], [15].
In the automotive field, this End-to-End approach is still in the early developing phase, but it is presumed that will play a crucial role in future technology [26], while E2E methods have shown promising results in other areas, such as quadcopter control, where AI-driven models have outperformed human pilots [31]. In addition to deep neural networks, reinforcement learning (RL) is another promising approach. In RL, the autonomous car acts as an agent that interacts with its environment, receiving observations and computing actions that maximize a reward signal. This method requires extensive trial and error, making it well-suited for simulation environments.

## 1.7 Thesis Outline

The thesis is organized as follows:

1. *Chapter 2*: This chapter introduces simulation environments for autonomous driving, categorizing them based on their features. It provides an overview of the submodules or building blocks of such software and concludes with a state-of-the-art review of the most relevant simulators.

2. *Chapter 3*: This chapter presents the AR-Sim simulator, detailing the motivations for its development and the target requirements. Each software component is then examined in detail.

3. *Chapter 4*: This chapter concludes with a discussion of the obtained results and presents suggestions for future improvements.

# Chapter 2

# Simulation Environments for Autonomous Vehicles

## 2.1 Introduction

Simulation environments are software aimed to replicate real-world systems for the purpose of analysis, testing, and development of engineering solutions. In the automotive industry, these tools are essential for developing advanced driver-assistance systems (ADAS), autonomous driving technologies, vehicle dynamics, and powertrain optimization, enabling engineers to test and refine solutions in a controlled, virtual environment before physical prototypes are created. They simulate various physical processes providing a controlled environment that allows to test hypotheses, validate designs, and optimize performance without the risk and costs associated with real-world testing. The adoptions of these tools additionally allow the comparison of performance metrics under a wide range of conditions including rare or dangerous conditions that would be impossible to reproduce in real-world tests. Simulation environments can also be used to create dataset used to train artificial intelligence AI and machine learning models, this application is very important due to the enormous quantity of data needed by those to improve. Is exactly this trend, of using data-driven algorithms, that has caused a great increase in the effort to develop high-fidelity simulation environments for robotics and autonomous driving applications [32]. This is because training an AI model in simulation is only useful if the synthetic data generated is a realistic reproduction of real-world data, otherwise, the model trained in simulation is only capable of working in the simulation and not in the real world. This problem in the literature is known as Sim-to-Real transfer [30].

An effective simulator must meet several key criteria to ensure comprehensive and accurate testing. According to [33], the ideal simulator should be:

1. fast: to get a large amount of data faster than real-time
2. physically-accurate: realistic proprioceptive measurement
3. photo-realistic: realistic exteroceptive measurements

These requirements are in contrast with each other, however, the emerging software and hardware allow to mitigate the trade-off.

### 2.1.1 Historical Context and Evolution

The first simulators for automotive began to appear in the 1990s with the main focus on traffic flow and vehicle dynamics. Examples of this are SUMO [34]for the former and CarSim [35] for the latter. Those kinds of simulators demanded relatively modest computational resources since rendering high quality images was of no need. Between 2000 to 2015 there was the "Dormient period", as named in [36], technology hardware reached a bottleneck resulting in a stagnation of new simulation tool development. During this time also there was not yet interest in learning-based algorithms. However, a notable new simulator released in this phase was VI-GRADE [37].

In the last decade, numerous new simulators have been developed. Unlike automotive simulators developed before, which were predominantly commercial,a growing ecosystem of open-source tools has emerged. These also cover new aspects of automotive technology and are designed to support the development and testing of autonomous driving technologies. Private manufacturers are also contributing to this open-source ecosystem with companies like Waymo providing for free their simulator and large dataset [38]. Additional other companies like Intel or the Toyota Institute are sponsoring the widely-used CARLA simulator [28]. Nevertheless there are also important commercial and closed-source simulators such the one from Ansys, VI-GRADE, Hexagon, and many others. These commercial simulators are primarily used in industrial applications, while open-source simulators are predominantly utilized in research. The adoption of open-source simulators facilitates resource sharing by defining standard data formats and allows researchers to avoid the time-consuming task of developing new simulation platforms [36].

The reasons for the increase in the number of simulators is twofold. First, the advent of autonomous driving has required the need for new requirements for simulation environments. Autonomous driving simulators must not only simulate vehicle dynamics but also accurately replicate the surrounding environment, including static obstacles or dynamic agents like other vehicles. This shift in requirements has led to the development of more sophisticated tools that integrate complex rendering engines and comprehensive sensor models such as cameras or LiDAR. The need for a more sophisticated tool can be noted in [39] where the GTA-V video game was used to train a convolutional neural network (CNN) to drive a car. This approach yielded good results in simulation, demonstrating both the utility and the necessity of more specialized tools.

Secondly, more resources have become available to ease the use and development of very realistic simulation environments. An always-increasing computing power is available on the market, with today's computers having multiple-core CPUs and next-generation high-performance GPUs. At the same time also the software technology available improved a lot, with very sophisticated rendering engines capable of representing advanced features such as material shaders, real-time reflections, and advanced illuminations pipelines. These solutions are now easily accessible and user-friendly, with game engines like Unreal Engine and Unity Engine providing all these features for free, along with high-level, well-documented APIs. Other software components that spread in popularity and performance have been the physics engines. Those software are now available for free in most cases and are already integrated or easy to integrate with a rendering engine. This combination is always available in game engines, which are software frameworks born for game development but are increasingly being used to enhance simulation software for robotics applications, as can be seen with [28] or [33]. Prior to this new trend way more limited capabilities were available to engineers or researchers with tools like Gazebo or Matlab offering very good physics engine and API to implement simulators but with the lack of realistic rendering functionalities.

## 2.2 Classification

Due to the very challenging technology related to autonomous driving the importance of simulators has been widely recognized, from 2022 to 2023 over 50 % of the methods published in this field were either trained or tested in simulation environments [36], for this a variety of simulation tools, specific for different use case are today available. Traditionally autonomous driving simulators were only using a simplified vehicle model since they are intended to operate far away from the vehicle handling limits, but now with the born of autonomous racing competition and data-driven controllers like RL also accurate physic simulations are needed. In this section, a brief categorization of these tools is described.

### 2.2.1 Vehicle Dynamics Simulators

Vehicle dynamics simulators have long been used in the automotive sector, due to the large research in this field very detailed vehicle models have been available for a long time but most reliable simulators are typically commercial and developed in collaboration with car manufacturers. They allow engineers to model and simulate vehicle behavior, including all aspects from power-train, aerodynamics, and suspensions under various conditions to optimize vehicle design. Examples of widely used simulators in this sector include:

- **ADAMS** which is a powerful multi-body dynamics simulation tool used for the design of complex mechanical systems, in particular for automotive are very well-known Adams/Car and Adams/Tire for the analysis and design of the suspension.

- **VI-GRADE**: A simulation platform that offers real-time vehicle dynamics simulation for automotive applications.

- **CARSim**: A software package for simulating the dynamic behavior of vehicles, widely used in the automotive industry for developing and testing vehicle systems.

- **Simulink/Matlab**: A versatile tool for modeling, simulating, and analyzing dynamic systems, including vehicle dynamics.

These traditional simulators provide robust tools for testing and optimizing vehicle performance. However, they primarily focus on the vehicle's dynamics and require less or no emphasis on the surrounding environment.

### 2.2.2 Driving Policy Simulator

Driving policy simulators are used to design and train algorithms concerning the driving policy of autonomous vehicles. This type of simulator, usually, uses a simplified representation of the environments and of the underlying vehicle dynamics, their goal is to ease the simulation complexity and only focus on what concerns the trajectory planning or behavioral decision. For road applications, the common functionalities consist of allowing to simulation of different traffic scenarios intersections, and road layouts. Examples of this kind of simulator are for example CommonRoad [40], Matlab, Waymax [38] and Nuplan [41]. This simulator can be used both in a classical pipeline approach or for E2E training.

### 2.2.3 Full Featured Simulators

These types of simulators allow us to gather realistic sensory data, simulate complex enough vehicle dynamics models and evaluate/train driving policy algorithms. This category of simulators allows to evaluate the full performance of the vehicle and software stack from perception, planning to control, at the same time they allow to experiment with E2E training. For road applications the most famous available today that are still maintained are CARLA [28] and AWSIM [42] which are open-source, while licensed simulator are, for example, the ones provided Ansys, VI-GRADE and Hexagon.

### 2.2.4 autonomous racing simulator

Autonomous racing simulators are different in respect to the above category because the environments are different, being limited to race tracks, and the physical accuracy of the dynamic model is more important since the car is required to drive at the handling limits. In this category, there are very few, if any, open-source solutions. Learn-to-Race [43] is the only simulator available that is realistic from both rendering and dynamics viewpoints. However, it has not gained much popularity in the research community, probably because there is a strict license agreement to get access to it since the core functionality is provided by the private company Arrival. Instead, a fully commercial solution is the Ansys simulator which was also used in the IAC, while today the new series of virtual racing for the IAC is going to use a new simulator, not yet available, which will be developed by the Autoware Foundation in collaboration with the start-up Autonoma. On the other hand are some simplified simulation environments for scaled autonomous vehicles but this lacks some of the key requirements needed today to advance research in this area. A more detailed comparison of all these simulators will be given later in a subsequent chapter. What emerges is the noticeable fact that currently, no open source simulator is present which is target for autonomous racing, this is the exact need from which this project has been carried out.

From this point forward, the term "simulation environment" will refer specifically to the autonomous racing simulators discussed in this category, not the general simulators mentioned previously.

## 2.3 Application and Use Cases

In developing autonomous driving software simulators are used in two ways: *Software in the loop* (SIL), conducted in the early stage of development and *Hardware in the Loop* (HIL), which is more complex and expensive and so done in later stages.

Software in the loop consists in testing the code making use of the simulator as a replacement of the real car. The consequences of this are numerous, first, it implies that the software doesn't need a real car to be tested so can be performed from a desktop computer making it easier and faster. Secondly, the simulation can be run faster than in real life, with this it's meant that 1 second of real-time can correspond to multiple seconds in simulation, provided that the simulation software is well designed and the computer running it is powerful enough. To speed the development also parallel programming can be exploited by running concurrently multiple instances of the simulation to test more cases. Hardware in the loop instead is the testing of the actual hardware by making use of a simulator to emulate the input received from sensors in order to understand if the hardware is faulty or capable to run real-time.

### 2.3.1 End-to-End and Machine Learning

Specifically the context of End-to-End, and more generally machine learning approach, simulation environments are fundamental because the car before being able to perform some meaningful action requires a lot of training which usually corresponds to a lot of crashes as well [44]. On the other side policy learned in simulation can be hard to transfer to real-life robotics systems because the simulated environments usually are only a limited representation of the environment. To this end of great use is domain randomization of observation and model used, as outlined in [15].

## 2.4 Building blocks

This section is devoted to identify which are the different building blocks or modules that constitute a simulator, the aim is to develop a general framework that will be useful to better compare different simulators and to locate the newly created AR-SIM. These modules are not to be confused with the functionalities of the simulators but they serve as a matter of logically subdividing the implementation of this complex software. The individual modules that can be identified are:

1. Physic Engine: all that concern the simulation of the physical system acting on the scene and the interaction of this with each other

2. Rendering Engine: rendering of the environment

3. Sensor Models: sensors simulation capability

4. Framework Compatibility and API: which framework or programming language is compatible with the simulator.

### 2.4.1 Physic Engine

Physics engines are crucial for simulating physical systems by modeling the laws of physics to provide realistic movement and interactions within virtual environments. Core components of a physics engine include rigid body dynamics, which manage the motion and interaction of solid objects; collision detection and response, which handle object collisions; and solvers, which compute the equations of motion and resolve constraints.

Advanced real-time physics engines like NVIDIA's PhysX, Bullet Physics, and MuJoCo offer additional features such as soft body dynamics for deformable objects, fluid dynamics for simulating liquids and gases, and particle systems for effects like smoke and fire. These engines are optimized for real-time performance and are widely used in interactive applications, from video games and virtual reality to robotics and scientific simulations. These physics engines typically construct ordinary differential equations (ODEs) internally, requiring users only to specify the properties of objects and their connections, with the engine handling the equation derivation.

In contrast, tailored physics engines used in engineering, particularly for vehicle dynamics simulation, focus specifically on solving the equations of motion and deriving an analytical model through some assumptions. These custom engines, often developed using tools like MATLAB, Python, or C++, involve explicitly coding the ODEs governing the vehicle's dynamics and employing specialized models for handling vehicle-specific phenomena such as tire-road interactions, suspension behavior, and aerodynamic effects.

Different levels of assumptions and simplifications can be applied in modeling. One important factor to consider is the capability to identify the parameters required by the model. It is counterproductive to use a highly complex dynamic model if the parameters cannot be accurately identified. More complex vehicle dynamic descriptions require additional parameters, which are often difficult to identify and may necessitate expensive setups. For instance, precise mathematical models of tire dynamics exist [45], but the tests needed to determine their parameters are highly complex. Additionally, more sophisticated models increase computational complexity, making real-time simulations challenging.

For these reasons, simplified vehicle descriptions are often used in academia as they offer a suitable trade-off between the richness of the dynamics described and the feasibility of parameter identification. These models provide effective simulations without overwhelming computational resources. A great open-source collection of commonly used models, along with implementations in Python and MATLAB, can be found on [40].

In the context of open-source autonomous driving simulators, the most prominent physics engine is NVIDIA PhysX. This is because many open-source simulators utilize either Unity or Unreal game engines, which come with NVIDIA PhysX pre-integrated due to their open-source nature and optimization for real-time performance. However, trade-offs in NVIDIA PhysX prioritize visualization over precise physics behavior. The new release of Unreal Engine is changing to a new physics engine called Chaos Physics so projects like CARLA are switching to it, nevertheless, NVIDIA is developing a new simulation tool called Omniverse [46] that is using Physx.

Both methodologies are good depending on the situation, to perform handling limit maneuvers it's needed to accurately describe the dynamic model or at least the user should be conscious of the underlying assumption made. This is easier to do if one is using an explicit analytical model over a physics Engine implementation. On the other side using a Physic engine allows the functionalities to detect and resolve collisions which are complex to implement.

The NVIDIA PhysX SDK implements a specialized vehicle dynamics instance, which is less commonly discussed in automotive literature and will be treated in the next sub-chapter. Conversely, the well-known vehicle dynamics models for analytical models will not be further detailed here, as they are extensively covered elsewhere like in [40].

### PhysiX Vehicle Model

The PhysX physics engine includes a dedicated vehicle dynamics Software Development Kit (SDK), known as the PhysX Vehicle SDK. This SDK extends the core PhysX SDK by providing specialized components for simulating wheels, tires, suspensions, and the chassis of vehicles. It computes the forces acting on the vehicle due to tire and suspension interactions, with the vehicle's rigid body dynamics being handled by the broader PhysX SDK. This includes considering forces from other actors in the scene to update the vehicle's acceleration.

In the PhysX Vehicle SDK, the vehicle is modeled as a collection of sprung masses, one for each wheel. To determine the forces acting on the vehicle, a ray cast is performed for each suspension to detect the road geometry beneath each wheel. This helps to determine the wheel's position and the suspension force generated. The tire forces are then calculated based on this vertical force, camber, and wheel slip. The aggregation of these forces is used to update the vehicle's position, factoring in external and inertial forces.

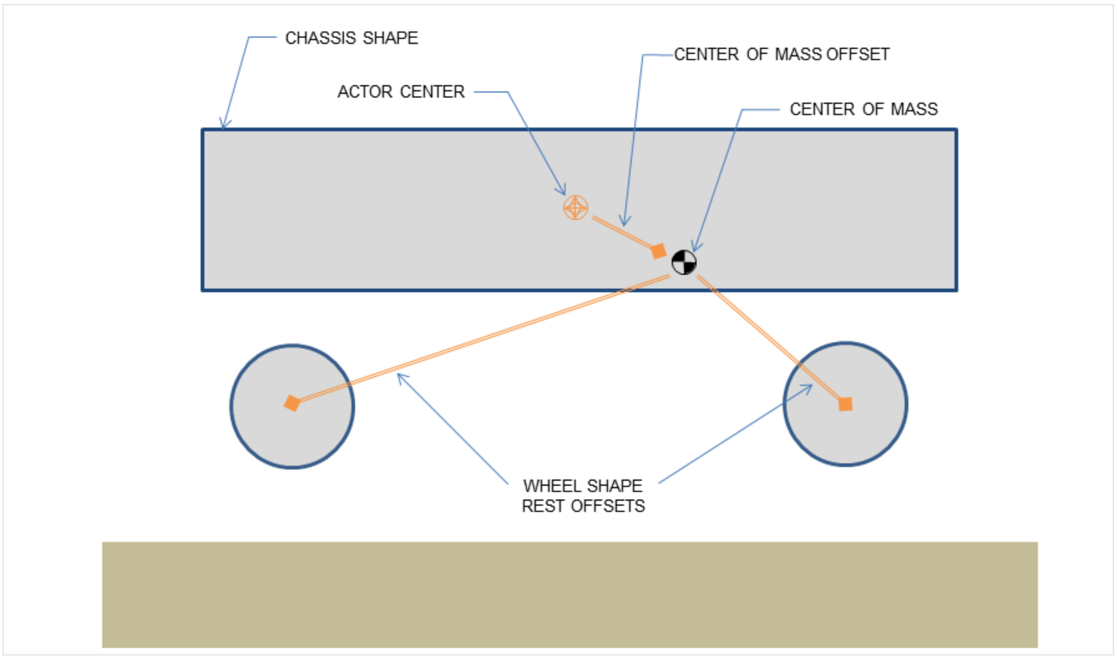


**Figure 2.1:** Caption

The PhysX Vehicle SDK also allows for the simulation of engine dynamics, clutch, gears, and differential mechanisms. However, these features are not implemented in Unity's PhysX API and are only available in older releases of Unreal Engine, which directly expose the PhysX API. With Unreal Engine transitioning away from PhysX, and AR-SIM being developed in Unity, the subsequent discussion will focus solely on Unity's implementation, which simplifies these functionalities.

### Unity Wheel Colliders

In Unity, wheel colliders are components that encapsulate the functionality of the PhysX Vehicle SDK. The wheel friction model in Unity is simplified, comprising two spline segments: one representing the friction curve from the origin to the peak value and another from the peak to the asymptote. The Unity documentation does not specify whether this friction value changes with vertical load or if it solely depends on slip.

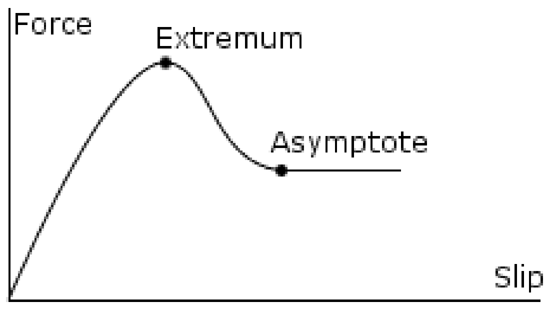


**Figure 2.2:** PhysX wheel friction curve

Suspensions in Unity can be tuned by defining the rest position as a normalized value between 0 and 1, corresponding to the wheel hub's position at maximum compression and maximum elongation. These two values can be set relative to the nominal position of the wheel center. Users can also adjust the suspension stiffness (in Newtons per meter) and damping (in Newtons per second). Additionally, it is possible to set the mass of the wheel collider and the vertical application point of the force relative to the wheelbase. The figure 2.3 represents the wheel collider of Unity where all the key elements can be visualized. The large 2D circle indicates the size of the physics wheel, which can be adjusted using the Wheel Collider's Radius property. The horizontal green line represents the halfway point of the Wheel Collider along the X-axis, with its angle indicating the rotation of the wheel. A small 3D sphere shows the point where the wheel forces are applied; this can be modified using the Wheel Collider's Force App Point Distance property. The vertical orange line marks the maximum distances the wheel can move up and down along the vertical Y-axis from its central point due to applied forces. This distance is adjustable via the Wheel Collider's Suspension Distance property. The intersection of the orange and green lines denotes the "resting" point of the wheel when no forces or equal forces are acting upon it, which can be altered using the Wheel Collider's Target Position property.

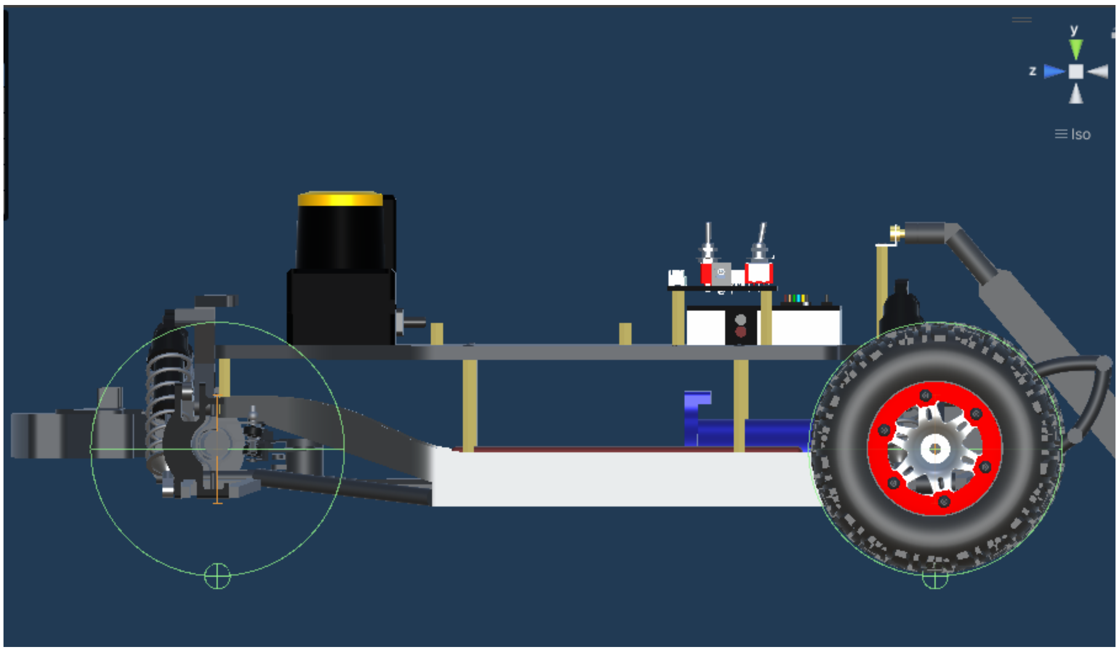


**Figure 2.3:** Unity wheel collider component

The PhysX physics engine includes a dedicated vehicle dynamics SDK, known as the PhysX Vehicle SDK. This SDK extends the core PhysX SDK by providing specialized components for simulating wheels, tires, suspensions, and the chassis of vehicles. It computes the forces acting on the vehicle due to tire and suspension interactions, with the vehicle's rigid body dynamics being handled by the broader PhysX SDK. This includes considering forces from other actors in the scene to update the vehicle's acceleration.

In the PhysX Vehicle SDK, the vehicle is modeled as a collection of sprung masses, one for each wheel. To determine the forces acting on the vehicle, a ray cast is performed for each suspension to detect the road geometry beneath each wheel. This helps to determine the wheel's position and the suspension force generated. The tire forces are then calculated based on this vertical force, camber, and wheel slip. The aggregation of these forces is used to update the vehicle's position, factoring in external and inertial forces.

The main criticism of Unity's implementation is its focus on visual behavior rather than physical accuracy. The available parameters are designed to allow users to tune how the vehicle responds to commands, but accurately reproducing a real car's behavior is challenging due to the difficulty in deriving these parameters from actual vehicle data. The underlying assumptions and simplifications are not exposed to the end user, making it difficult to close the simulation-to-reality (S2R) gap. However, this approach facilitates the simulation of various road profiles and their effects on the vehicle.

### 2.4.2 Rendering Engine

Rendering Engine are software tool that allows to reproduce complex 2D or 3D scenes by simulated light sources, texture and material properties. These software are vastly used in other domains like graphic design, architecture or video games. Especially for game development are available free software licenses for two very powerful game engines Unreal Engine and Unity Engine. Game engine are software that usually combines together a physic engine and a rendering engine in addition to a set of facilities to develop games. Both of the two aforementioned provide great rendering capabiliy and indeed are very popular used also for engineering simulation in various fields including autonomous driving. Since these two software are the most used a brief comparison of the two is reported in the following.

| **Property** | **Description** |
|---|---|
| radius | The radius of the wheel collider. |
| suspensionDistance | The maximum extension distance of the suspension, measured in meters. Orange segment of 2.3 |
| suspensionSpring | Defines the suspension's spring force and damping characteristics. |
| mass | The mass of the wheel. |
| wheelDampingRate | The damping rate of the wheel's rotation. |
| forwardFriction | The friction properties of the wheel in the direction it is moving. |
| sidewaysFriction | The friction properties of the wheel perpendicular to the direction it is moving. |
| center | The position of the wheel relative to the vehicle's transform. |
| forceAppPointDistance | The distance from the wheel to the point where the force is applied. |
| motorTorque | The torque applied to the wheel to make it rotate. |
| brakeTorque | The torque applied to slow down the wheel. |
| steerAngle | The angle at which the wheel steers. |

**Table 2.1:** Main properties that can be set with the Unity Wheel Collider

**Unreal Engine**

Unreal Engine, known for its high-fidelity graphics and powerful rendering capabilities, offers a rich simulation environment for autonomous driving research. Unreal Engine's Blueprint visual scripting system and its C++ API provide flexibility in developing complex simulations that can replicate real-world scenarios with high precision. Unreal Engine is considered to have a more steep learning curve with respect to Unity Engine, making it less appealing for nonexperts of the field.

#### Unity Engine

Unity Engine is a versatile and widely used platform for game development and interactive experiences. Known for its user-friendly interface and extensive documentation, Unity is accessible to beginners and small studios, as well as large development teams. It supports both 2D and 3D rendering, making it suitable for a variety of applications, from mobile games to virtual reality simulations. Unity's primary scripting language is C#, and it offers robust cross-platform capabilities, allowing developers to deploy their creations across multiple devices and operating systems. Unity is widely used in simulation applications due to the more simple API.

#### Gazebo

Gazebo is also a valid option since is very well integrated in ROS but it outperforms in terms of rendering capability by both unity and unreal. Gazebo though is way less computationally intensive to run and can be a great solution, as been until now, for all such cases where the rendering is not so important.

### 2.4.3 Sensor Model

All autonomous driving simulators need to provide both exteroceptive and proprioceptive sensor models. With exteroceptive sensors are meant sensor that measures information outside the ego vehicle and so allow to understand the environment and interact with it. Examples of used sensors in this category are cameras, LiDAR, radar and ultrasonic sensors. Proprioceptive sensors instead provide information about the internal state of the vehicle as position, velocity and acceleration. For this often are found on vehicles' wheel encoder to measure wheel velocity, IMUs for acceleration and in some advanced cases also ground speed sensors (GSS) sensors to directly measure the velocity of the vehicle.

The availability of a rendering engine highly influences which kind of sensor can be simulated, for instance without a rendering engine camera sensor is impossible to simulate. Instead, there are solutions to simulate LiDAR or ranging measurement sensors without a rendering engine but due to the complexity of it, this is only found in 2D environments like in the F1Tenth gym [47]. Usually, the use of a rendering engine allows to perform this task more easily and with more accuracy for 3D environment as well.

Proprioceptive sensor instead only requires a model of the dynamic equation and so are more easy to implement.

When dealing with sensor simulation, it is crucial to simulate the underlying noise because real-world data is always affected by noise. Additionally sensors operate at different acquisition frequencies between each other, also the software processing this data could run at another different frequency, meaning not all data is available at every processing interval. Moreover, non-solid-state LiDARs exhibit noise due to their rotational motion. These characteristics make the sensor simulation quite challenging and so more often only a simplified model is possible to implement.

### 2.4.4 Communication and API

End users of a simulation environment typically want to use the sensor data from the simulator in their own development framework which can be a specific programming language or operating system. In robotics applications, for example, ROS2 is commonly used as a base framework to manage different nodes and topics that send command and transport sensor data. However, this may not be the only case, for example in reinforcement learning the most used framework is Gymnasium, which is a specialized Python API. Therefore the simulator needs to be able to communicate with those environments efficiently.

One approach is to develop the simulator in the same framework as the end-user application. However this constrains the usability of the software to that framework or forces to introduce some overhead with bridges or wrappers to interface with other frameworks. Alternatively, a more versatile solution is to design a simulator that is independent of the end user framework used and then provide an efficient communication mechanism to enable the simulator's use in various frameworks. In addition by using a more general framework, the simulator does not restrict the end user to a specific environment but allows them to use the one they prefer the most.

To illustrate, a user developing autonomous driving software likely prefers to write the software in C++ and run it in ROS, rather than writing it in C# within Unity. Therefore, the simulator should interface seamlessly with ROS. Similarly, users working in reinforcement learning prefer to use Python and the Gymnasium API, so an interface with Gymnasium should be available.

## 2.5 State of the Art review

The following section provides a more detailed examination of well-known simulators for autonomous driving, Table 2.2 summarizes and compares the features of these simulators with those of AR-Sim. The chapter discusses and compares the different modules described in the section before.

### 2.5.1 F1TENTH Simulator

F1Tenth Simulator [47] is a 2D simulator that doesn't make use of a rendering engine, so no camera model can be simulated in this context. With 2D it's intended that the environment simulated is two dimensional and so also the LiDAR model is 2D. This is not a limitation since the default configuration of the F1Tenth is with a 2D LiDAR model, but it limits other further experimentation for users. The core library is developed on Python as a

Gym OpenAI interface, therefore reinforcement learning solutions can be easily developed. Besides this, a wrapper over the gym API is made to develop a ROS2 simulator that allows to perform some SIL. Visualization is performed using RVIZ and collision detection is implemented manually with a Gilbert–Johnson–Keerthi algorithm. A new map can be easily added using the one generated with the SLAM toolbox running on the real car. The simulator is designed to simulate up to 2 cars at the same time, The coordinate system used to place the car is Cartesian, which makes it not immediate to start the vehicle in a different position with respect to the origin. The main limitation of this simulator is that the vehicle can't be configured with different kinds of sensors, also is not possible to simulate camera sensors. In addition, the dynamic model implemented is only one and it's not present a way to allow users to choose different dynamic models. No, the sentence I provided could be more polished for an academic context. On the other hand, this simulator is resource-efficient, allowing it to run on a vast range of laptops.

Another known simulator for F1Tenth is [48]. This F1tenth simulator is a modular, ROS, and Gazebo-based autonomous racing simulator designed to mimic a physical F1/10 race car. It features highly adaptable architecture, used in virtual sensing and actuation components, a robust ROS Python API, and the use of XML macros for racecar descriptions. The simulator supports running several autonomous racecars, each controlled independently through ROS namespaces for communication. However, using Gazebo for rendering limits the realism of camera sensors, moreover, the simulator does not offer a built-in environment for reinforcement learning.

### 2.5.2 CARLA

The CARLA simulator [28] stands out in autonomous driving research due to its reach capabilities and API functionalities. It provides freely reusable

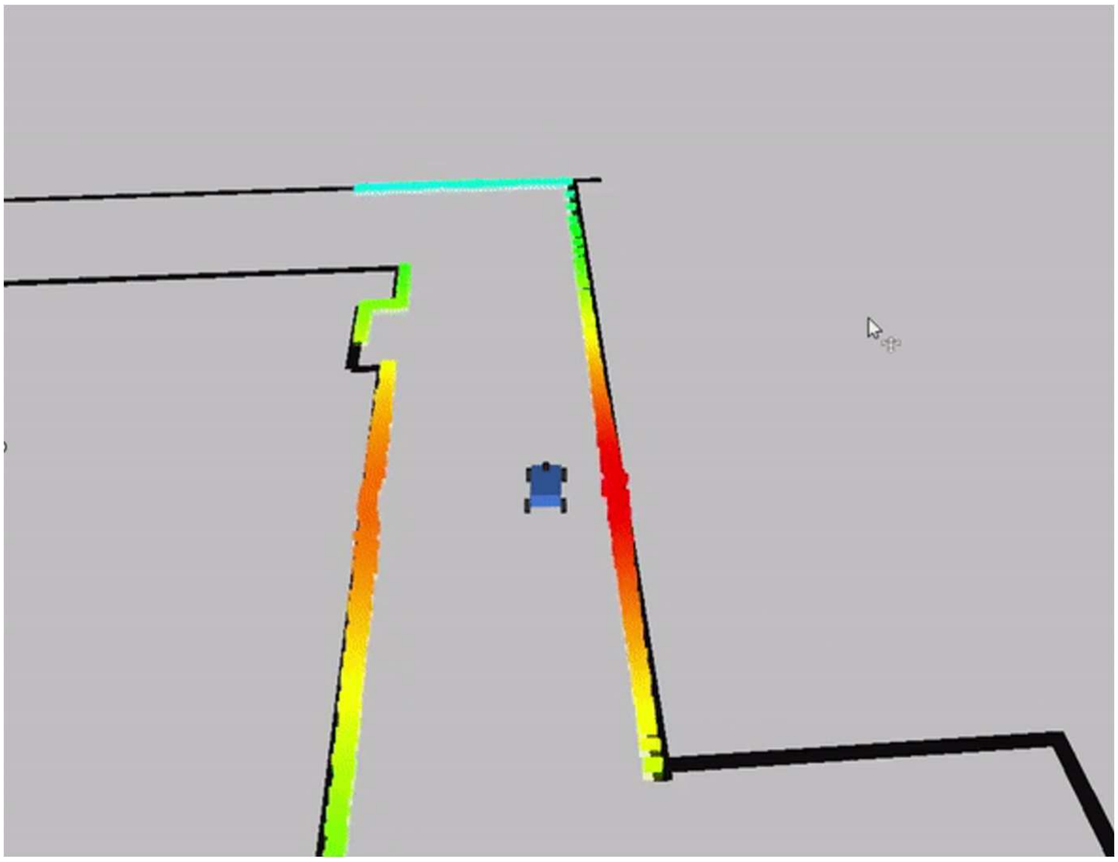


**Figure 2.4:** F1Tenth gym simulator

digital assets, including urban layouts, buildings, and vehicles. Built on Unreal Engine 4, it was the first large open-source simulator, released in 2016, to provide state-of-the-art rendering quality, ensuring high fidelity and realism of the environment. This was a big innovation in the field because for the first time allowed researchers to collaborate on the development of autonomous driving and provided a standardized data format and benchmark that wasn't present before. The simulator uses a client-server architecture, where the server runs and renders the simulation, and the client provides an interface for interaction. It can operate with ROS or directly with Python in different configurations allowing both the classic modular pipelines or to train deep learning models via imitation or reinforcement learning. CARLA supplies high-fidelity sensory data to support a majority of sensors, including RGB cameras, LiDARs, and event cameras. A variety of urban scenarios can be reproduced under various weather conditions, this controlled environment provided by CARLA allows large-scale, rapid sensor data synthesis, making it possible to reproduce extreme conditions and safely test algorithms.

Despite being a great open source software,CARLA is not optimized for racing, and most of the simulation effort and computational resources are for urban scenarios, which are unnecessary for racing applications. Furthermore, the user does not have full control over the customization of the vehicle dynamics model. In CARLA By default, only the PhysX vehicle dynamics model is available, and although there are extensions to link CARLA with CarSim, this is not an open-source solution. Therefore there is very little control that the user can have over the dynamic model. Besides, CARLA is a very high-volume software package, good for high-performance computers, but its function is not so smooth on an average laptop. Usability might therefore be limited for students.

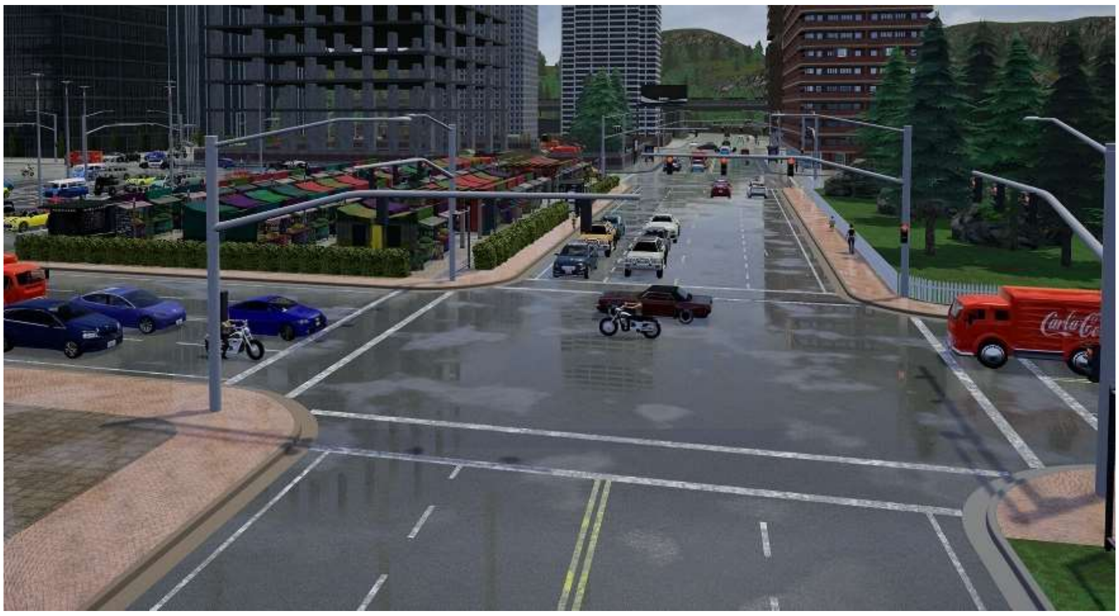


**Figure 2.5:** CARLA Simulator

### 2.5.3 AWSIM

AWSIM by Autoware [42] is another sophisticated open source simulation tool that can be considered as a competitor to CARLA. Developed in Unity Engine it delivers a high-fidelity simulation environment with detailed representations of road networks, traffic signals, pedestrians, and other vehicles. Fully integrated with ROS2, AWSIM enables seamless communication between the simulation and the vehicle's software stack, facilitating development and testing in a controlled and reproducible environment. The simulator supports various sensors, including LiDAR, radar, cameras, and GPS, generating realistic sensor data for testing sensor fusion algorithms and perception systems. Additionally, AWSIM allows for the creation and execution of complex driving scenarios, encompassing diverse weather conditions, lighting changes, and traffic variations, essential for evaluating the robustness of autonomous driving systems.

However, AWSIM has certain drawbacks when compared to CARLA. Being a relatively newer tool, AWSIM has a smaller user base and less community support, resulting in fewer resources, tutorials, and third-party extensions. While AWSIM allows for the creation of complex scenarios, the diversity and range of pre-built scenarios are more limited compared to CARLA, which offers a wide variety of urban and rural scenarios and traffic configurations. In terms of environmental detail and realism, AWSIM's simulation environment might not match the graphical fidelity of CARLA's urban environments built on the Unreal Engine. AWSIM moreover doesn't support an end-to-end or machine learning framework but it's limited to ROS2 usage.

As CARLA, AWSIM lacks control over vehicle dynamic models, a feature that can be critical for detailed vehicle dynamics simulation.

**Figure 2.6:** AWSIM

### 2.5.4 AutoDrive simulator

The AutoDRIVE Simulator [19] was primarily conceived for the AutoDRIVE platform, developed by the same team, to facilitate the testing and development of autonomous driving algorithms in scaled, city-like environments that include intersections and traffic signals. A subsequent release introduced an F1Tenth model, featuring a mesh and dynamic parameters compatible with NVIDIA's PhysX engine. This makes the simulator a viable alternative for F1Tenth, offering superior rendering capabilities compared to previously mentioned F1tenth simulators. AutoDRIVE simulator is developed within Unity. Multiple vehicle models are implemented, all of them using the NVIDIA PhysX engine and are equipped with sensors such as 2D LIDAR, RGB cameras wheel encoders, and IMU. All simulator functionalities are developed in C# and executed within the Unity environment, ensuring compatibility with Unity's ML agents for machine learning applications. Furthermore, the simulator supports integration with the Robot Operating System (ROS) and provides APIs for direct scripting in Python and C++, by using web sockets. However, the simulator does have some limitations. While reinforcement learning applications are available, they are designed to be used within Unity using the Unity ML asset, which is not commonly adopted by researchers who, instead, are more familiar with Python Gymnasium, since it integrates with all the standard libraries available for RL. Moreover, C# is not a commonly used programming language in the robotics community. Other critique can be made on the vehicle models, since the sensor configuration is fixed, with the LIDAR limited to a 2D type. Additionally, the ROS2 integration is unconventional and hard-coded for specific use cases, lacking a proper library developed in C++ or Python. Users must call WebSocket functions and parse JSON message data, which diminishes the simulator's user-friendliness and modularity. Additionally, the implementation of the ROS2 node does not follow standard procedures. One node runs continuously and writes messages to a configuration file about the car's steering and throttle values, while another waits for a reply from the Unity server. Upon receiving a reply, it reads the latest configuration file, parses the values into a JSON message string, and publishes the message to the corresponding topic. This approach is not designed to be modular concerning the number of vehicles in the scene. Furthermore, camera and LIDAR data are copied into JSON messages, a resource-inefficient process that involves converting and reconverting strings for each image and LIDAR dataset.

### 2.5.5 Flightmare

Flightmare is a modular, flexible quadrotor simulator based on a general architecture, composed of a configurable rendering engine developed in Unity and a flexible, user-configurable physics engine for dynamics simulation. These components are decoupled and this allows for the high-speed regime

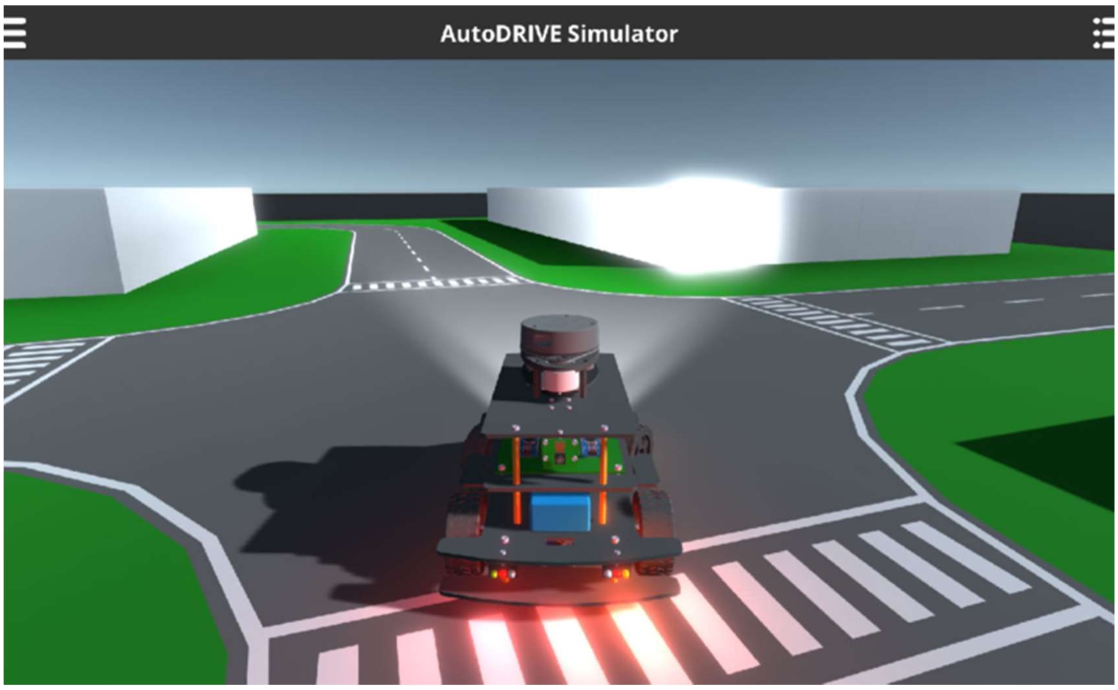


**Figure 2.7:** AutoDrive Simulator

of the simulator. The simulator supports advanced rigid body dynamics, including frictions and rotor drags, and yields high-fidelity graphical rendering with realistic lighting in 3D environments. The Flightmare simulator can simulate hundreds of quadrotors in parallel, which makes it excellent and rapid for data collection for large-scale reinforcement learning. It interfaces with important robotic simulators like Gazebo to enable human-robot interactions in realistic scenarios and safe pilot training through virtual reality headsets. In addition to this, Flightmare comes with a rich camera sensor suite that includes RGB cameras, depth cameras, semantic segmentation, and a functionality to extract 3D point cloud. The core library is developed in C++ but also a Python wrapper implementing an API for reinforcement learning is present. The C++ library can also be used to develop ROS2 simulation environments. This modularity is also made possible by the underlying communication mechanism implemented between the Unity server and the application. It's implemented on top of ZMQ, a powerful synchronous messaging library designed for distributed or concurrent applications. These features, along with the successful experimentation conducted with it [31], make this simulator a significant reference point, as no current autonomous vehicle simulator offers the capability to customize all these aspects of the simulation. Indeed, Flightmare has greatly inspired the development of the ARSim simulator.

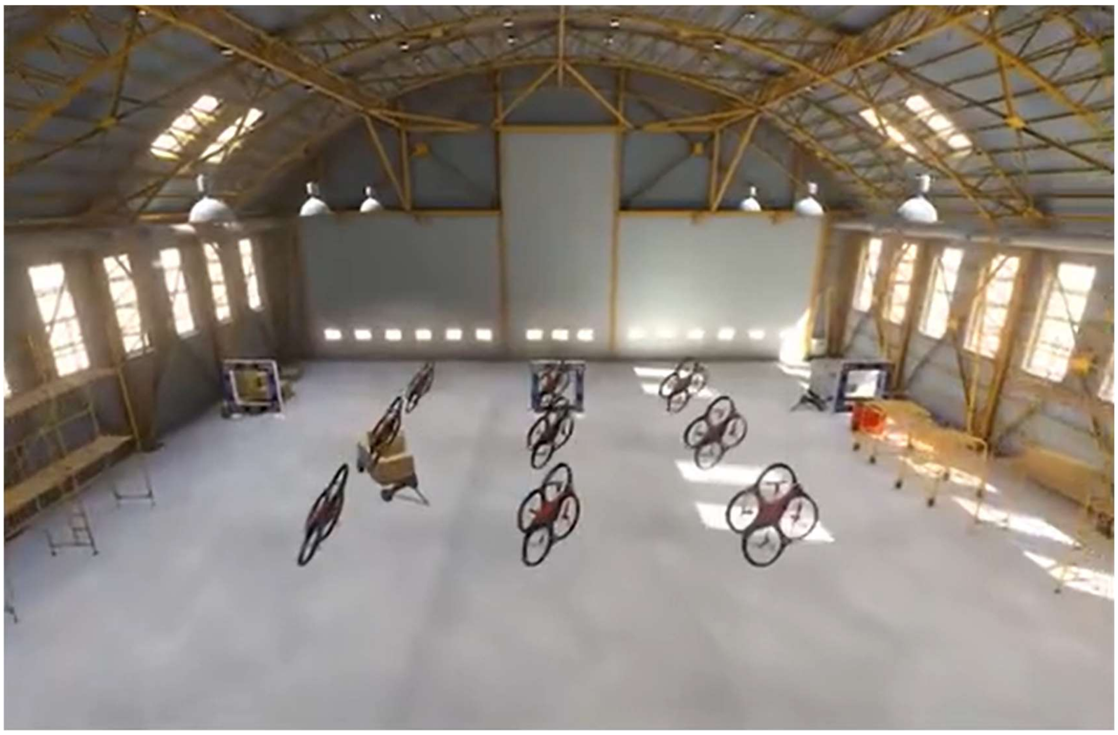


**Figure 2.8:** Flightmare simulator

### 2.5.6 Other Simulators

Several other simulation environments provide distinct features for autonomous vehicle research and development. The LGSVL Simulator is noteworthy for its robust capabilities, including high-fidelity sensor simulation and realistic urban environments; however, it is no longer actively maintained, limiting its long-term viability. Amazon's DeepRacer [16] and Donkey Gym [17] are entirely focused on an end-to-end learning approach using only camera sensors, which may restrict their applicability to more complex sensor fusion tasks. Learn-to-Race [49] was developed for Roborace competitions and offers sophisticated racing simulations, but it is not truly open-source as it requires users to sign a Software License Agreement, and it is likely no longer maintained following the cessation of Roborace. There are several reinforcement learning extensions available for TORCS (The Open Racing Car Simulator), and it's been used for research on end-to-end approach as in [50]. However, it is limited in terms of modern graphics and advanced sensor integration, making sim-to-real transfer challenging for cutting-edge research.

| | CARLA | F1Tenth Gym | AWSIM | AutoDrive | DonkeyCar Sim | Flightmare | AR-Sim |
|---|---|---|---|---|---|---|---|
| High Fidelity Rendering Capabilties | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | |
| Physic Engine Customization | Limited | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Physc Engine Compatibility | PhysX, CarSim | Single Track, Not Flexible API | PhysX | PhysX | PyhsX | C++ , External | Physx , C++ API, External |
| Configurable Sensor setup | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ |
| Collision Resolution | ✓ | Limited | ✓ | ✓ | ✓ | ✗ | Only for PhysX Engine |
| Framework & Programming Languages | ROS1/2, Python | ROS2, OpenAI Gym | ROS2 | Limited API for: C++,Python, ROS1/2. ML inside Unity | OpenAI Gym | OpenAI Gym,ROS, C++ API | Gymnasiuam, C++ API, ROS2 |
| Sensors Available | Complete | 2D LiDAR | Complete | IMU, 2D LiDAR, GNNS, Encoders, RGB Camera | Camera, IMU, GNSS | IMU, GNSS, RGB,Depth, and Segmentation Cameras | 2D/3D LiDAR, IMU, GNSS, RGB Camera |
| Race Track Modeling | ✗ | 2D image | ✗ | Prefab mesh | Built in Scenes | Run-time gate positioning | csv to 3D mesh , random generation, prefab mesh |
| Urban Scene Modeling | Hyper realistic | ✗ | Hyper realistic | Scaled urban environments | ✗ | ✗ | ✗ |

**Table 2.2:** Comparison of simulation environments

# Chapter 3

# Methodology

The Autonomous Racing Simulator AR-SIM implements a new kind of simulator in the field of autonomous driving, and it's a unique open-source simulator in the niche of autonomous racing simulators. It combines all the positive features of the existing simulators, providing a very flexible and modular framework that is possible to further extent by the user. The AR-Sim is composed of two main components: a C++ library, which implements all available API functionalities, and a Unity standalone application which implements photo-realistic environments used for high-fidelity sensor simulations. The unity executable can run either as an arcade mode, thanks to an interactive GUI, to allow fast experimentation of all the functionalities, or in conjunction with a custom C++ application written by the user. This is made possible by a communication library developed using ZeroMQ, a C library that was used to handle socket messages over IP/TCP protocol. In addition to these two modes the AR-SIM also make available also parametrized ROS2 simulator and a Python wrapper that allows the implementation of Gymnasium gym environments for E2E. The simulator core library is compatible only with Linux-based operating systems, whereas the Unity executable can also run on macOS and Windows. To extend

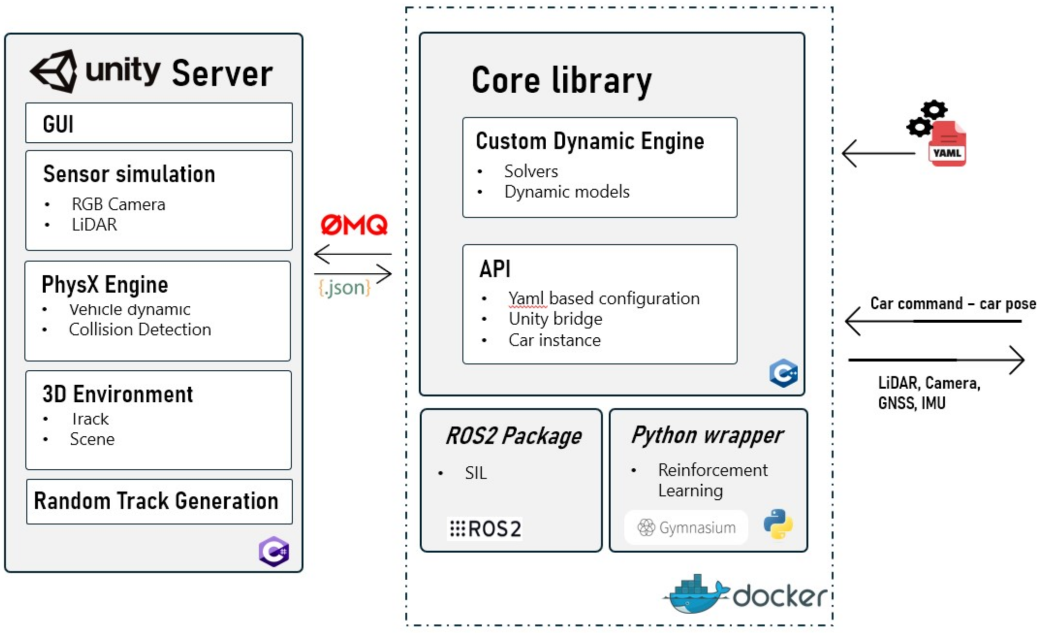


**Figure 3.1:** AR-Sim Software Architecture

the simulator's usability across different operating systems, a Docker file is provided. This allows the core library to run inside a container while the Unity executable runs directly on the host system, extending the support to all major operating systems.

The simulator presented in the state of the Art review AR-Sim do not offer an easily customizable framework, requiring users to adapt to the simulator's specifications. In contrast, this simulator allows users to tailor it to their needs, as in the flightmar simulator [33], but further extends the degree of customization with easy track creation, friendly Unity Editor and C++ Macros, and the possibility to use both PhysX and custom dynamic model. All existing simulators either provide only PhysX basic dynamical models, such as AutoDrive, DonkySim, or AWSIM, or they don't allow to use it at all such as in Flightmare. Some simulators like CARLA allow to use not only PhysX and provide some plugins for CARSim but it's not available to fully customize it with ease. The AR-Sim instead consents to use both the

PhysX model and the custom analytical equation, users can easily integrate the custom model they want by deriving a specified class from the base class available in the simulator API by leveraging C++ polymorphism. In the scene is possible to run instances of multiple vehicles each using a different dynamic model and physic engine, this is also a new feature. The flexible Unity editor functionalities implemented enable one to easily add custom scenes, vehicle models, and tracks without requiring much knowledge about the Unity editor, with just some drag and drop operation it's possible to make these new elements available in the simulator both in the GUI and from the C++ API by additionally writing just a single line of code calling a custom macro. Also for the track a novel feature is made available that allows the creation complex 3D mesh circuit layout just from a simple CSV file composed of x-y coordinates of the center-line and the width. This kind of file format is common in autonomous racing, a vast collection of race tracks saved in this format in [51] for full-scale race tracks and [52] for 1:10 race tracks, also a collection of algorithms is present to create optimal race trajectory starting from those file thanks to the work developed at TUM for autonomous racing motorsport [53]. Besides this is also available functionality to randomly create a 3D racetrack, features that isn't available in any other 3D simulator.

Even if supporting all these functionalities the simulator it's easy to use while still computationally efficient. This is made possible by providing a YAML-based configuration of the scene and agent present on it while providing an efficient C++ backend. Also, the target platform isn't only high-performing and costly computers but also, and mostly, the average laptop that any student or researcher can have at its disposal. This is achieved by optimizing the unity resource management of data and rendering quality, which can also be customized by the user at run time by choosing different quality levels.

A complete list of the requirements that drove the realization of this simulator is given in the following list:

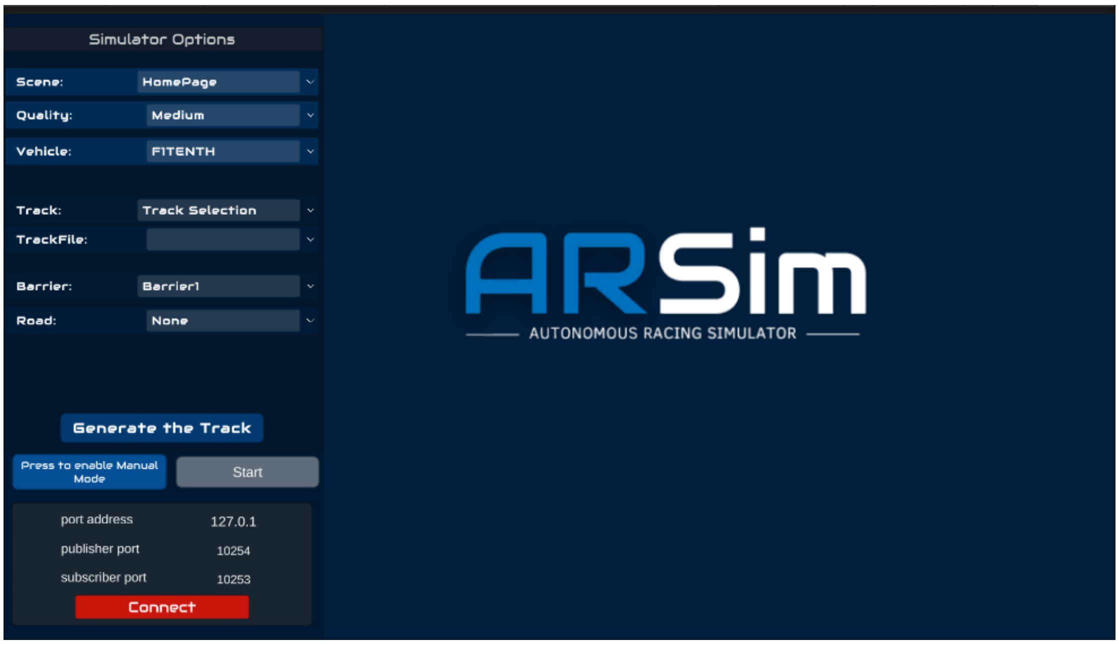
1. ease of use: quick learning curve to get to use it
2. ability to run on medium computing platform: large target end users
3. flexible physic engine: model complexity match system identification feasibility
4. high fidelity sensor simulation
5. Gymnasium interface
6. ready to use ROS2 simulator as F1Tenth gym
7. Reinforcement learning capability: domain randomization
8. ease to create custom racetracks and environments.

The rest of this chapter will give details about how each of these requirements has been reached through specific implementation choices by going into the details of the four modules.

## 3.1 Unity Server

The Unity executable is a binary file built with Unity, available for Linux, Windows, and MacOS. This binary is a standalone application that provides rendering functionalities, PhysX dynamics, and sensor simulation. Upon startup, the executable connects to a default IP address and socket port, listening for messages from the C++ library. However, this isn't the only way to run the executable. It also features a user-friendly GUI, allowing direct interaction with the simulator without the need for the external C++ API. The first mode is called in the GUI *Autonomous Mode*, while the alternative

*Manual Mode*. In both cases, the user can interact with the GUI to select environment simulation options. If then the mode is switched to Manual it's also possible to spawn the selected car model in the scene and drive it along the track. Additionally, the GUI allows users to select the port address for connections. All the functionalities provided by the GUI, plus some additional more, are accessible through C++ API and can be easily configured using a YAML configuration file.



Simulator Options

Scene: HomePage
Quality: Medium
Vehicle: F1TENTH
Track: Track Selection
TrackFile:
Barrier: Barrier1
Road: None

port address 127.0.1
publisher port 10254
subscriber port 10253

**Figure 3.2:** AR-Sim Unity Home page

### 3.1.1 Environment Scenes

The simulated environment consists of two components, the first is the actual environment, or scene, which represents a virtual reproduction of the real environment on which the track is positioned. The second component is the track itself. This modular configuration allows the creation of one single Unity scene that can be used to load a variety of race tracks for both scaled and full-scale vehicles, on the other hand, the same track can be loaded on different scenes.

The computational complexity of the simulated environment largely depends on the scene. More realistic scenes with complex material shaders and multiple sources of light are more computationally demanding, at the same time they allow to close the gap between simulation and reality for the exteroceptive sensor viewpoint. Constructing realistic scenes that closely represent real-world ambient is quite complex and is outside the scope of this thesis work, which instead aims at giving the user a simulation environment with the possibility to add custom scenes. Nevertheless, some scenes are already present on the simulator. Some of them are very basic to save computer resources in the case where more complex features are not required, others show more advanced ones like the Industry one. Also, since this simulator aims to be used in the F1Tenth lectures held at TUM at the FTM(Institute of Automotive Technology) warehouse, a simple scene of it is present.

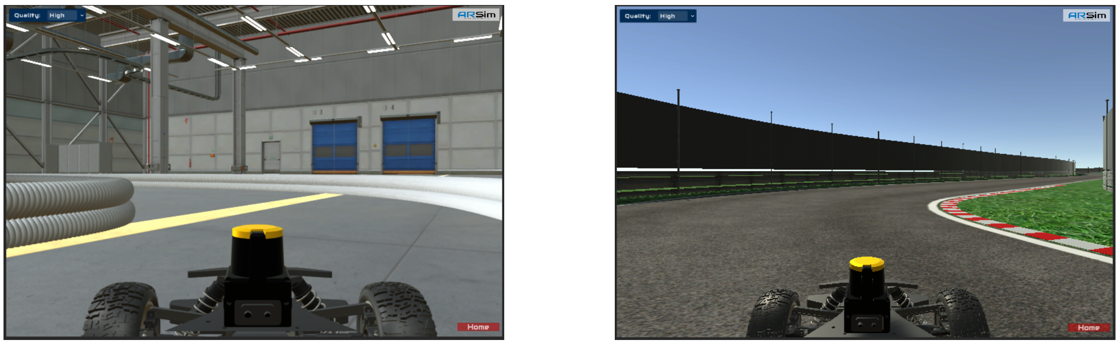


**Figure 3.3:** AR-Sim: F1Tenth race car PoV in 2 different scenes.

### 3.1.2 Race Tracks

Race tracks can be represented and stored in multiple file formats which allows for different levels of complexity and fidelity of the model created as well as the effort and time needed to build them. The first way is to create a

3D model of the track manually in 3D modeling software as Blender[1] and import the file as a *"fbx"* format inside the unity editor. This method has the advantage that users can create complex and realistic race tracks by accurately modeling race track boundaries and elements in the scene. The disadvantage of this approach is the double side of this flexibility which is that is time-consuming and can be a repetitive process in the case of simple track profiles such as in F1Tenth where they are made of simple pipe barriers. The second file format instead permits the rapid and easy creation of new tracks but with smaller customization capabilities. The file format used to store tracks in the second case is based on CSV files that store the center-line x-y position and width of the racetrack. This CSV file is converted at run time to a race track mesh using a unity package called Dreamteck Spline[2] that consents the extrusion of a mesh profile along a predefined path. Users in this way only need to store a CSV file, which in the case of F1Tenth can be easily created using the Hector slam and the python script developed by University of Waterloo[3]. Another source of multiple CSV files compatible with the one required by the track generator features can be found in [tum betz file] for both 1:10 and full-scale vehicles. In this second method, the user can choose, besides the CSV file, which kind of mesh to use for representing the track barrier and road path as well as which material to apply to it. This allows a great variety of track configurations with just a minimal effort. Users just need to add a mesh of the profile of the custom barrier and a texture to create a 3D model of the race track. This is a novel feature that is not present in any of the other simulators and it can improve the usability of this simulator in the research community, especially for working on F1Tenth.

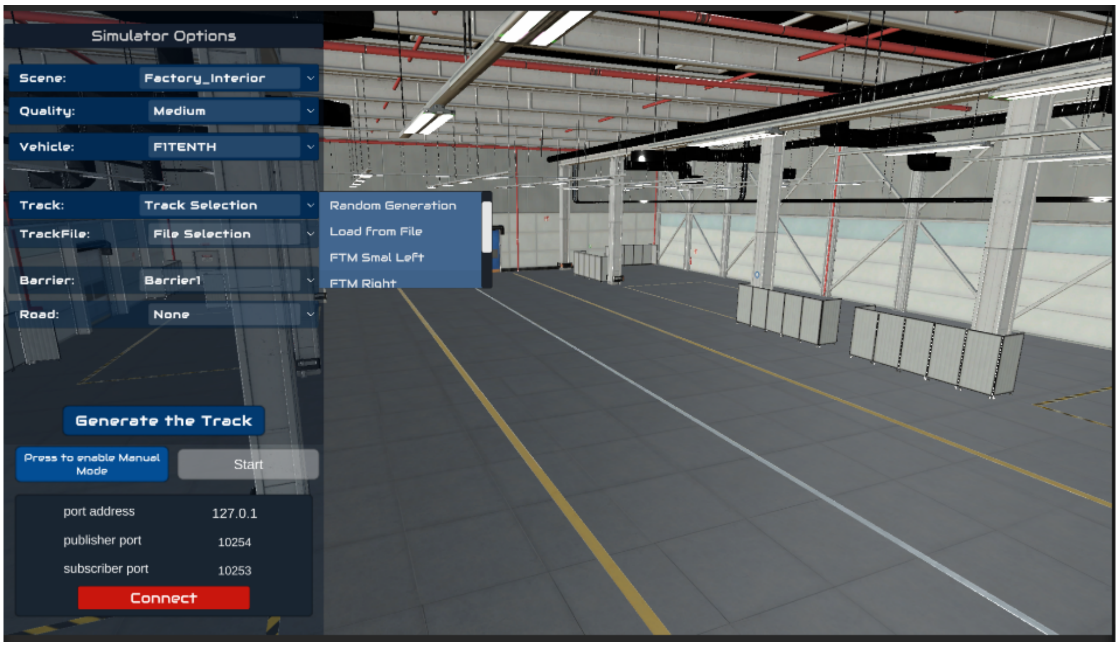
Figure 3.4 shows the Unity GUI of the AR-Sim simulator to create race

[1]https://www.blender.org/

[2]https://dreamteck.io/dreamteck-plugins/

[3]https://github.com/CL2-UWaterloo/Raceline-Optimization

tracks. After having selected a scene, is possible to choose how to generate the track, by clicking on the `Track` menu. The first two options,`Load from File` and `Random Generation`, if selected allow also to specify the pipe mesh and the road profile to use, an example of a different result obtained with the same CSV file input is given in figure 3.5. The other elements available in the drop-down view of the track menu instead are examples of track prefabs made before run-time and saved as Unity's prefabs.


Simulator Options

Scene: Factory_Interior

Quality: Medium

Vehicle: FITENTH

Track: Track Selection

TrackFile: File Selection

Barrier: Barrier1

Road: None

port address 127.0.1

publisher port 10254

subscriber port 10253

**Figure 3.4:** Track Generation from AR-Sim GUI

## 3.1.3 Random Track Generator

Another interesting and novel feature is the random track generator that enables the creation of random 3D track mesh at run time with some random track profile and material chosen between a set specified by the user. This feature is useful, especially in domain randomization during the training phase of machine learning algorithms. The algorithm to generate a random

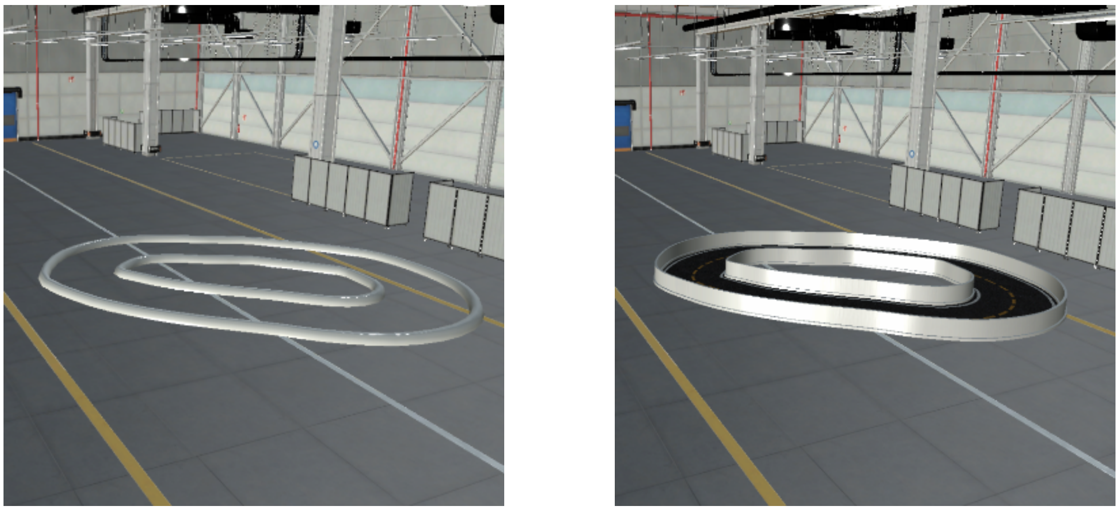


**Figure 3.5:** Track created with same csv file but different mesh options.

**Left**: track created extruding *pipe* profile. **Right**: track created extruding a *beam* profile and with a *road* mesh with a yellow center-line and dark asphalt material.

track is developed from scratch and requires as input just the number of waypoints, the two-dimensional space that the race track can occupy, the minimum turn radius, and the minimum track width. The generated track will then combine all these metadata attributes and create a 3D random track. It is also possible to implement a varying track width, useful as is the case of F1Tenth race tracks. The algorithm consists of three parts, first a set of random points is generated and sorted in a clockwise order. These points are filtered and adjusted to obtain a polygon that will give rise to a feasible spline when those vertexes are used as control points for a Catmull-Roll spline. Then such spline is created using the DreamTeck spline package and two additional splines are added on the right and on the left to create the barriers. The result of the creation of the center-line is depicted in Figure 3.6. The track can be saved as a CSV file so that it can be reused in the future. The random track generator can also be used as a way to manually create race tracks in the Unity editor. Indeed the user can run the algorithm and

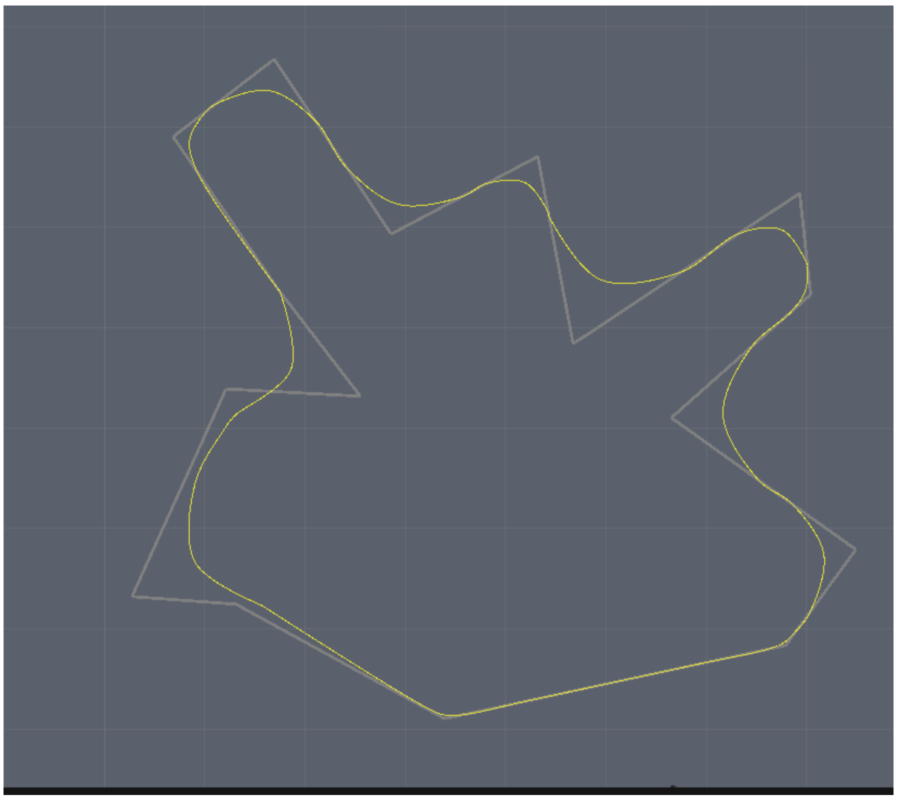


**Figure 3.6:** Random race track generation. *White*: original random polygon, *Yellow*: final center-line of the race track

then manually customize the obtained racetrack for more specific needs using the run time editor capabilities provided by the DreamTeck Spline. The generated track can then be saved as prefab and added to the list of available options. These functionalities are implemented in a C# script class named `TrackGenManager`. This script is attached, together with the track generation manager, to a game object present in the GUI scene. A custom editor panel has been developed to easily add new track barrier profiles and road materials.

### 3.1.4 Back end

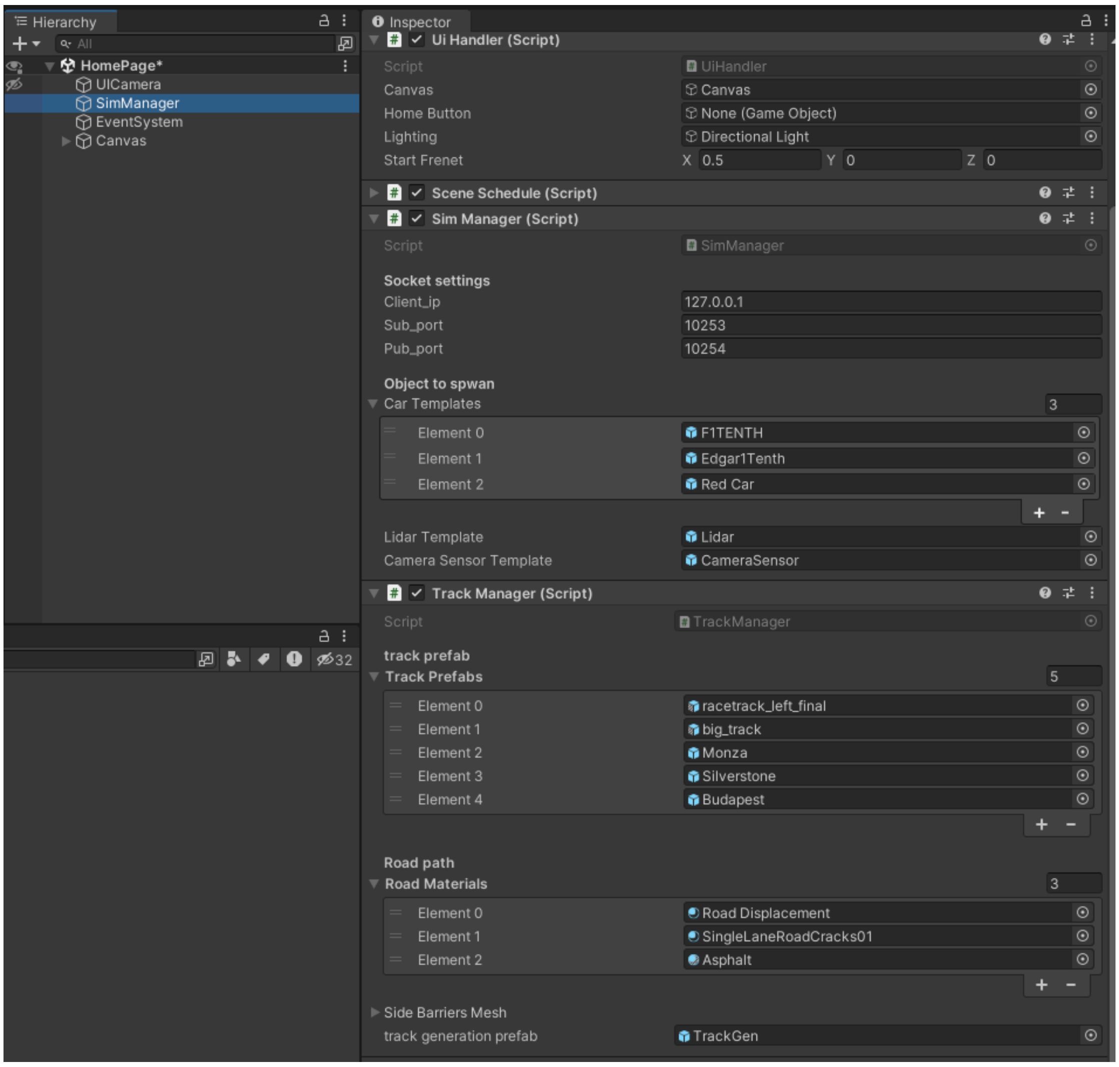
With the back end are intended all the scripts that bring together the functionalities of the simulators. From this category are excluded the sensor model and the vehicle controller which are instead treated in a separate subsequent section. Two main classes are used by the simulator, the first one is named `simManager` and is responsible for implementing all the simulation logic such as updating and simulating vehicle physics, spawning objects on the scene, and handling the message request replay pattern. The second main component is the `UIHandler` that handles all the callback of the GUI and on startup dynamically updates the GUI menus based on the option the user sets in the Unity Editor, Figure 3.7. In this way, custom options added by the user are automatically visible also in the GUI without any additional effort.

There is also another important part of the back-end that implements the data structure for message serialization and de-serialization using C# structure and a third part JSON parser library developed for the DotNet framework.

#### Simulation Manger

The simulation manager is the core `Monobehaviour` class that runs at startup on the simulator and is responsible for establishing the connection on the socket and listening for incoming messages. The runtime of this object terminates if the user enables the `Manual mode` via the GUI and restarts when the `Autonomous mode` is re-enabled. In this way is possible to seamlessly switch between GUI GUI-based mode of the simulator or the script-based one. The messages, composed of a topic header and a payload, are implemented using a header-only binding of ZMQ called *zmqpp*[4]. The message's topics

[4]https://github.com/zeromq/zmqpp



Ui Handler (Script)

| | |
|---|---|
| Script | UiHandler |
| Canvas | Canvas |
| Home Button | None (Game Object) |
| Lighting | Directional Light |
| Start Frenet | X 0.5 Y 0 Z 0 |

Scene Schedule (Script)

Sim Manager (Script)

| | |
|---|---|
| Script | SimManager |
| **Socket settings** | |
| Client_ip | 127.0.0.1 |
| Sub_port | 10253 |
| Pub_port | 10254 |
| **Object to spwan** | |
| Car Templates | 3 |
| Element 0 | F1TENTH |
| Element 1 | Edgar1Tenth |
| Element 2 | Red Car |
| Lidar Template | Lidar |
| Camera Sensor Template | CameraSensor |

Track Manager (Script)

| | |
|---|---|
| Script | TrackManager |
| **track prefab** | |
| **Track Prefabs** | 5 |
| Element 0 | racetrack_left_final |
| Element 1 | big_track |
| Element 2 | Monza |
| Element 3 | Silverstone |
| Element 4 | Budapest |
| **Road path** | |
| **Road Materials** | 3 |
| Element 0 | Road Displacement |
| Element 1 | SingleLaneRoadCracks01 |
| Element 2 | Asphalt |
| Side Barriers Mesh | |
| track generation prefab | TrackGen |

**Figure 3.7:** Unity Editor of the project. The `SimManager` Game object is selected, and on the left its inspector is shown. From this menu is possible to add more options, such as car model, scene, tracks, and track profiles. Changes made here are automatically displayed on the GUI and callable from the C++ API.

are given in Table 3.1.

The initial setting topic message contains information for the scene, track, and vehicles configuration, an example of the initial setting message is given in the code snippet Code A.5.

| **Topic** | **Payload** |
|---|---|
| *Initial Settings* | marks the start of the simulation. Contains initial settings that include the settings for the simulated scene with track, scene, and vehicle configuration |
| *Update* | topic used to send and receive vehicle information updates about the position, collision, sensors data |
| *Ready* | Topic sent by Unity to the client application that marks that the server is ready to run. |

**Table 3.1:** messages topic and payload

Upon receiving this message the server starts to prepare the simulation environments by asynchronously loading the scene. Once the scene is loaded the track is placed in the specified position and the car(s) with their specific sensors are placed in the pose. Also, the message specifies if the car instance dynamic should be managed by Physx or if it will be handled by some external solver with respect to Unity, this can be shown in the Code A.5 in the keyword `is_kinematic`: if set to false it means that PhysX is used. At this point, the simulation is ready to start and the unity server sends a `Ready` message to the socket. After this initial phase, the messages are sent using the `Update` topic. The C++ API sends `Update`s topic containing different payloads for each vehicle based on the type of solver used. If the solver chosen for the vehicle is Unity then command action, composed of steer target and throttle percentage, are sent; otherwise if the solver is external pose updates are sent. On the Unity side, that information is parsed to the appropriate vehicle instance. After this, the physic solver is called and all the car poses are updated and packed into a new message. At this time also the LiDAR sensor is simulated for the specified delta time of the simulation step, which is defined at the beginning in the initial setting message. At the end of the frame then a synchronous GPU read is made to transfer camera data from the GPU to the CPU, this task is very computationally expensive since the CPU has to wait for the GPU command buffer to execute the request and transfer image data. This may take some milliseconds depending on the GPU workload which can't be controlled fully from Unity. Practically this means that the simulation speed depends heavily on the image quality and GPU workload caused by other processes running. On the other side, the reply message update that Unity sends to the C++ executable contains the vehicle's updated position, if simulated in Unity, vehicle collision information, and sensor information. The first two are sent using a JSON serialized message while the latter are sent using raw bite data to reduce runtime due to the message serialization.

### 3.1.5 Car model and physic engine

Car models are represented in unity as *prefab* composed of a mesh model, wheel colliders for physic simulation, and box colliders for collision detection. In addition, a set of reference frames can be specified for LiDARs and Cameras. These reference frames are used as relative reference frames for positioning sensors on the car. The dynamic of the car is simulated using the wheel colliders together with a C# script that implements some additional controller and that manages the input received. Users can further extend this script to implement custom logic for managing the wheel colliders. Figure Figure 3.8 depicts an example of car prefab, where a box collider has been used to reduce the runtime cost of computing collisions. Collisions detected by any of the colliders present on the car are detected and passed to the C++ client, in addition, if the model is using the PhysX engine those collisions are also resolved by PhysX.

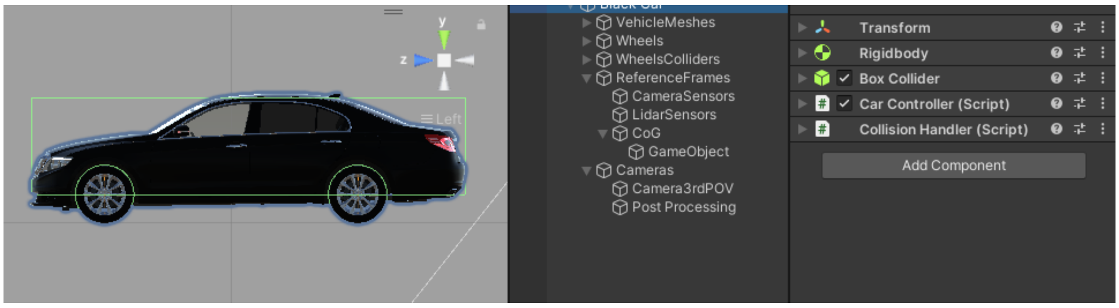


**Figure 3.8:** Car Prefab in Unity

**Green**: wheel colliders and box collider for collision detection

### 3.1.6 Sensors

Unity is used to simulate RGB cameras and LiDAR. RGB cameras are simulated using the functionalities already provided by Unity URP Camera and the cameras are rendered manually at the end of each frame. The option available allows one to set the quality of the camera by specifying the pixel size in the horizontal and vertical dimensions, the horizontal field of view, and the relative position with respect to the camera reference frame of the vehicle. The LiDAR sensor instead is custom and implements a parameterized model that can simulate both 2D or 3D LiDARs. This sensor is simulated using `Unity raycast`. Raycast permits casting a ray from a starting point up to a maximum distance and detecting the distance at which objects are intersected.

The parameter that can be set permits simulation of LiDAR with different numbers of vertical channel and horizontal channels, also both the horizontal and vertical field of view can be set together with the initial position of the field of view. The sensor model enables also the simulation of the effect of the revolution frequency by implementing the rotation behavior of the LiDAR: for each time interval of physic simulation, the LiDAR rotates by an angle that depends on the revolution frequency. The sensor data is stored in

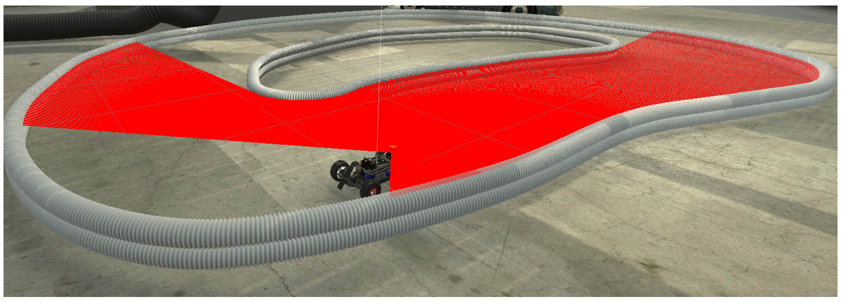


**Figure 3.9:** In *red* are displayed the rays cast by the LiDAR sensor present on the F1Tenth car. By specifying the direction of each beam is possible to simulate different LiDAR models.

a buffer which is updated and published on the "Update" topic only when a full revolution is made.

| **LiDAR Parameters** | |
|---|---|
| *relPosition* | relative position of the LiDAR sensor respect to the LiDAR reference frame of the vehicle prefab |
| *relRotation* | specify the relative rotation of the LiDAR sensor respect to the LiDAR reference frame of the vehicle prefab |
| *scanFreq* | rotation frequency of the LiDAR motor |
| *horizontalFOV* | horizontal FOV |
| *sartHorizontalFOV* | start horizontal angle at which is performed the first measurement, measured respect to the forward direction anti-clockwise. |
| *horizontalResolution* | number of channels in the horizontal direction |
| *verticalFOV* | field of view in the vertical direction |
| *startVerticalFOV* | start vertical angle at which is performed the first acquisition. Measured respect to horizontal plane. Negative value corresponding to the beam facing downward |
| *verticalResolution* | number of channels in the vertical direction |
| *maxLinearRange* | max distance at which an object can be detected |

**Table 3.2:** LiDAR sensor options

| **Camera parameters** | |
|---|---|
| *height* | number of vertical pixel |
| *width* | number of horizontal pixel |
| *fov* | horizontal FoV |
| *nearClipPlane* | nearest plane the camera render |
| *farClipPlane* | furhter plane the camera render |

**Table 3.3:** Camera parameters

### 3.1.7 Rendering

Several rendering engine options are available in Unity, including the Built-in Render Pipeline, the Universal Render Pipeline (URP), and the High Definition Render Pipeline (HDRP). Rendering pipelines in Unity are frameworks that manage how graphics are rendered on the screen, each tailored for different levels of visual quality and performance requirements. The Built-in Render Pipeline is the default option, providing basic rendering capabilities with minimal customization. In contrast, the URP is designed for optimized performance on a wide range of devices, offering a good balance between realism and computational efficiency. HDRP, on the other hand, targets high-end platforms, delivering advanced visual effects and high fidelity at the cost of increased computational demands.

For this project, the URP was chosen as it offers a good balance between realism and computational requirements, making it ideal for simulations that demand both performance and visual fidelity. Baked lighting, which pre-calculates the lighting information and applies it to static objects, is also used since it provides significant advantages in run-time performance without compromising on visual quality. Additionally, an option to specify the quality of the rendering was implemented, allowing users to adjust the visual settings according to their specific needs and the computational resources available. This flexibility ensures that the simulation can run efficiently on a variety of hardware configurations while maintaining an adequate level of realism.

## 3.2 Core C++ Library

AR-SIM implements its core library and API, which allows it to interact with the Unity engine, in C++. The choice of C++ goes against one of the requirements of an ease-to-use simulator since more beginner-friendly languages are available like Python, but more access to memory management and better performance motivate the choice of C++. To compensate for the less friendliness to not expert programmers, who may have little or no knowledge of C++, all the functionalities of the simulator are configurable by a YAML file **??** and **??**. In this way, a user doesn't need to write any line of code to use the simulator and can just use it out of the box with a simple configuration file that can be used to create ROS2 simulators or gym environments in Python. On the other side, for more experienced programmers, a rich and flexible C++ API is made available to further customize the simulation. The AR-SIM library is managed using CMake and can be installed as a shared or static library. All the dependencies are managed with CMake to provide an easy build, also a Docker file is available to build it. The library can only run on Linux-based OS since the communication library used to exchange data with Unity, ZeroMQ, it's only available for Linux. The availability of the Docker file though allows running the C++ application inside a container while the unity executable can run on the host machine. This library provides different classes and functionalities, in this paragraph and subsequent ones only the main ones are presented with some code example, full documentation of the implementation is available on the website of the simulator that is linked to the GitHub page.

| **AR-Sim main classes and functionalities** | |
|---|---|
| *Unity Bridge* | handle all objects in the scene and communicate with unity. |
| *Car* | main class that represents the vehicle. |
| *Sensor* | class for lidar and camera to handle unity data. |
| *Dynamics* | interface abstract class for dynamic and states models, solvers. |
| *Wrapper* | wrapper class that manages all the functionalities. |

**Table 3.4:** AR-Sim main classes and functionalities

### 3.2.1 Car and Dynamic

The `Car` class represents all car functionalities and allows for full customization of its instances, reflecting the properties and actions of a real car. Each car instance stores various properties such as sensors, dynamic models, collision status, and dynamic state. A key feature of the `Car` class is its decoupling from a specific dynamic model, it just provides methods to interact with the underlying dynamical model employed in the specific instance. Each dynamic model is derived from `ICarDynamics<Paramete,State>`, a templatized class based on the parameters and dynamic state used, which in turn derives from `IDynamics`. The reason is that the API aims to be flexible to any dynamic model, which is assumed to need a specific parameter structure and state space representation, as can be seen in [40]. The use of the `ICarDynamics<Parameter,State>` allows for compile time polymorphism on the parameter and state used within the car dynamic model. While the use of a non-templatize base class `IDynamics` consents to use run-time polymorphism. The dynamic model takes as input a generic vector $u$ of command values. The parameters are derived from `IParameters`, which provides a pure virtual method `load` that must be *overridden* by derived parameters classes. This method defines a common API to read a YAML file. The dynamic state of each dynamical model derives from a base class that provides methods to retrieve and set the state.

This structure facilitates the customization of the dynamic model used in simulations while still providing a common interface to make them work together. Consequently, each dynamic model can have a different parameter structure and appropriate state space representation.

The `Car` class stores a smart pointer of `IDynamics` coupled with the dynamic state achieving run-time polymorphism, **??**. A Collaboration diagram for dynamic models implementation, taken from the Doxygen[5] documentation of the library, is shown on the Figure 3.10, while Figure 3.11 pertains to the parameters.

An external solver can also be used to simulate the dynamic, in this setting the `Car` instance is used to update the underlying state values with the one computed from the external solver. The implemented physics solver, used for integrating the dynamic model implemented within the library, is a Runge-Kutta 4. To make it possible to use both C++ solver and PhysX a common interface is defined. The car accepts as input a command structure composed of throttle and steering values, which is then converted to the underlying vector input of the dynamic model. In this way, from the user's viewpoint is indistinguishable which of the two solvers is used. The input of the command then is simulated by the C++ library if the model is defined in C++, otherwise, it is sent to Unity as it is and then is stimulated in Unity. It is also possible to define a more complex Additionally, for interfacing with ROS2 Ackermann drive messages, a function can transform a velocity target into a duty cycle or throttle command. This common API is made to ease the use of both solvers at the same time but is not considered a limitation since for more tailored use it's possible to simulate the dynamic

[5]https://www.doxygen.nl/index.html

model externally from the car class using the input vector of the dynamic model. The car also stores an enumeration used to identify the mesh model used in unity. The parameters of the car dynamics are stored and loaded using YAML files and a custom parser can be defined to load a variety of car dynamic parameters. An example of this is Figure 3.11 where it's shown how the interface class `IParameters`, defined in the library, can be used to define a custom parameter structure for a specific dynamic model such as the single track.

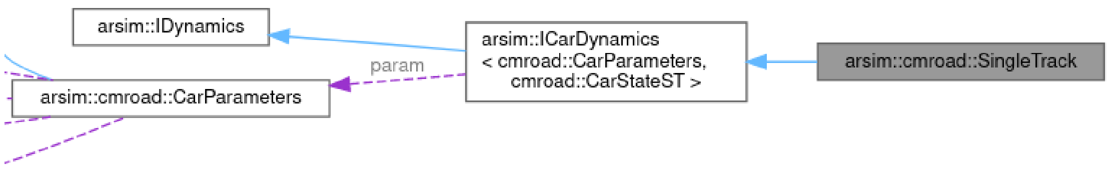


**Figure 3.10:** Single track dynamic model collaboration diagram

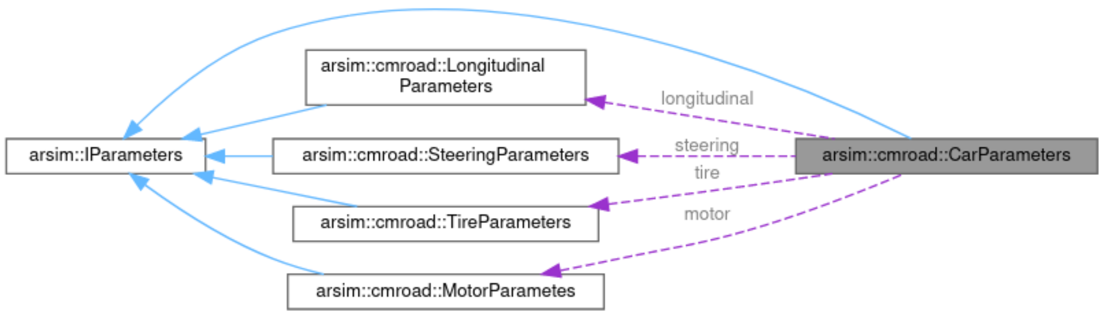

arsim::cmroad::Longitudinal
Parameters
arsim::IParameters
arsim::cmroad::SteeringParameters
arsim::cmroad::TireParameters
arsim::cmroad::MotorParametes
longitudinal
steering
tire
motor
arsim::cmroad::CarParameters

**Figure 3.11:** Parameter class collaboration diagram

Sensors can be easily added to the vehicle and are stored as shared pointers in dynamically allocated vectors. All the car instances can be configured using a single YAML file **??**.

### 3.2.2 Sensors

Exteroceptive sensors are simulated in Unity and then the data is sent to the C++ runtime application. The C++ library has API facilities and classes to store and retrieve sensor data as well as to set the corresponding settings in the code that are then sent to Unity.

#### RGB Camera

The camera data is stored using a deque, which is a double-ended queue allowing efficient insertion and deletion from both ends, containing structures with two `cv::Mat` elements. To avoid race conditions, a mutex, which is a synchronization mechanism, is used to ensure safe access to the data structure. Camera settings can be configured either through the API or via a YAML file.

#### LiDAR

The LiDAR class, like the Camera, enables to handle the LiDAR data and to set the LiDAR configuration options. Data is stored in pre-allocated vectors.

### 3.2.3 Settings

The settings structure is designed to store and configure the various parameters required for simulation, which are subsequently published to Unity to set the simulation. This structure includes options such as the simulation time step and the substep size, both of which are fundamental for defining the temporal resolution and accuracy of the simulation. Additionally, the settings allow for the selection of different tracks and scenes, enabling the customization of the simulation environment. These options provide a high degree of flexibility and control over the simulation setup. Users can configure these settings through both an API and YAML files, ensuring that the configuration process can be integrated into automated workflows and scripts, as well as manually adjusted through user-friendly interfaces. The tracks and scenes stored in Unity are defined as enumeration, with the values corresponding to the order of those items in the Unity Editor. Adding a new track or scene is made easy with a custom *macro*, `REGISTER_UNITY(Enum,<Scene name>)` that registers the track to a static factory method pattern.

### 3.2.4 Bridge

The bridge is the class enabling the connection between the C++ executable and unity. It is the C++ counterpart of the `simManager` class in Unity. The bridge instance stores a vector of shared pointers to all car instances in the simulation environment and a publisher/subscriber ZeroMQ sockets pairs. The user needs first to add all the instances of the car to the bridge, using the apposite API, once this is done the bridge has a method called `connectToUnity` that sends to unity messages on the topic `InitialSettings` that are used to set the simulation. The bridge blocks the process until it receives a `ready` message. At this point, two methods of the Bridge are used to communicate with Unity. One is `SendToUnity` which updates the car pose or sends the car command to unity and is intended to be called after having updated the command to all vehicles, the other is called `ReceiveFromUnity` and is responsible for parsing the information of sensors and car pose sent by Unity. The bridge is also the component that allows the synchronization of the two process simulated times, this is achieved by the receive `ReceiveFromUnity` method that blocks the thread waiting for the update message from unity. In this way, the simulated time in C++ and Unity are updated with the same speed.

### 3.2.5 Wrapper

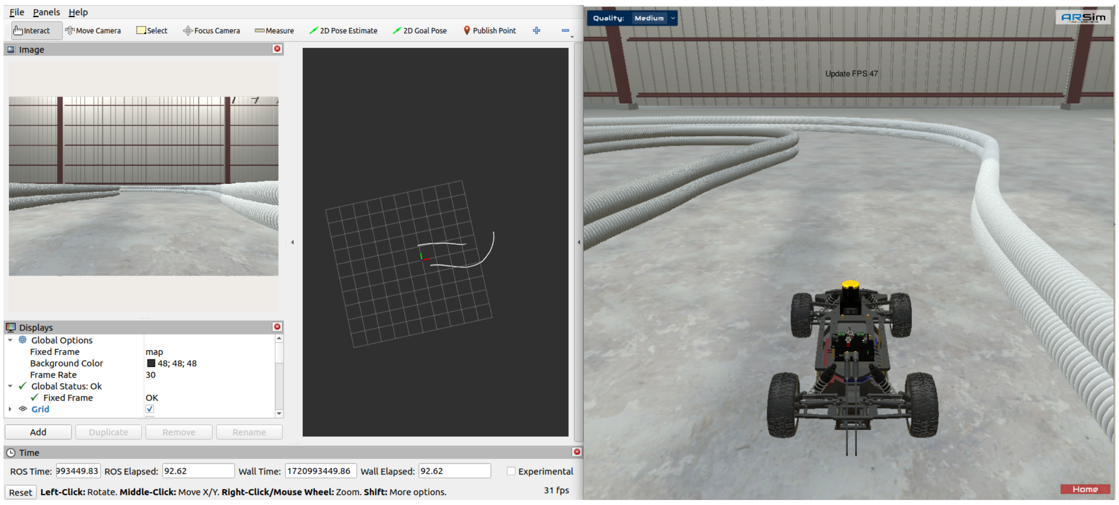
The wrapper class is called `Asim` and allows wrapping all the above functionalities in one class instance that simplifies the definition of the simulation environment with one single YAML file and class instance. This class stores a `Bridge` instance and a `Settings` instance and provides a method to retrieve them directly with the API. Also, it provides a step command that advances the simulation time step by the delta time specified in the simulation settings. An example of the use of this class is given in Code A.1.

## 3.3 Ros2 Bridge Package

The simulator library can be used, as any other library, inside Ros/Ros2 since it is sufficient to include the CMake target defined in the library `CMakeList.txt` file, so it's possible to build custom simulators and applications within ROS frameworks. Nevertheless, the idea of creating a custom ROS simulator can be time-consuming and could discourage beginners from using the simulator, so, for this reason, a fully YAML file parameterized ROS2 simulator has been developed as an external package. This simulator can also be used as a reference for more custom user-defined ones since provides a great example of how to use the AR-Sim library.

With *YAML file parameterized* it meant that all publisher and subscribers nodes are created at run time based on the YAML file passed to the launch file of the package. This is achieved using 2 YAML files, one is the same used in the wrapper class and the second one instead is specific to ROS2. This second YAML contains information about the topic name for the different information and the namespace to use for the ego car, all the opponent cars instead have as a prefix *car<id>* where "id" specifies the car number based on the order in which are placed inside the setting YAML file. Then each sensor type in the car is created a topic, with the right namespace. In this way is possible to easily customize topic names in order to have them match the ones used in the real car. This simulator is mainly thought to be used with the F1tenth since it reproduces the same topics available in the already existing F1Tenth simulators with additional topics for 3D point-cloud and images, anyway its flexibility makes it easy to adapt to other platforms.

Each vehicle subscribes to two topics: the `drive` topic and the command topic. The *drive topic* is used to send Ackermann drive messages, which are the ones used in the F1Tenth vehicles, the callback of this function calls the car dynamic method `Ackermann2Command` that converts the drive message to a command structure passed to the vehicle dynamic. This method needs to be overridden in a derived class of `IDynamics` class only if used. The other subscribed topic *command drive* is used to exchange custom messages defined in the ROS2 package of the ROS2 simulator which contain throttle and steer angle targets.



Global Options
Fixed Frame map
Background Color ■ 48; 48; 48
Frame Rate 30
✓ Global Status: Ok
✓ Fixed Frame OK
Grid ✓

ROS Time: 993449.83 ROS Elapsed: 92.62 Wall Time: 1720993449.86 Wall Elapsed: 92.62 Experimental

**Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:** Zoom. **Shift:** More options. 31 fps

**Figure 3.12:** Rviz2 visualization of the ROS2 AR-Sim package. On the right is visible the Unity executable, while on the left the Rviz visualization showing the camera image and the 2D LiDAR data.

## 3.4 Python wrapper

The development of a Python wrapper aims at providing high-performance computational capabilities of C++ with the user-friendly and flexible environment of Gymansium[6], a widely-used toolkit for reinforcement learning. Gymnasium provides a standardized API for RL application, making it easier to develop and benchmark RL algorithms. This integration has been created using pybind11[7], a lightweight header-only library that exposes C++ types in Python and vice versa. One of the critical aspects of this integration is managing memory and data ownership between C++ and Python. For this purpose it's been developed some custom capsules to transfer data ownership to the Python garbage collector, ensuring that resources are correctly managed without the user having to care about them and keeping a Pythonic style. Capsules in Python are objects that can store pointers to C++ data and define a destructor to manage the object's lifetime.

These functionalities have been used to create a gym environment in Python to train a single car to be driven, users can then use this environment as a reference to define more custom-specific ones.

[6]https://gymnasium.farama.org/

[7]https://github.com/pybind/pybind11

# Chapter 4

# Conclusions

This thesis has detailed the design and implementation of AR-Sim, a comprehensive simulator for autonomous racing vehicles, developed to address the limitations of existing open-source tools, and enhance the development of autonomous driving technologies, especially in the context of F1Tenth racing vehicles.

AR-Sim integrates advanced components including a C++ library, Unity for visual simulations and sensor models, a ROS2 package, and a Python wrapper implementing a gymnasium environment for reinforcement learning. Besides the ROS2 package, the C++ API can be used to create custom simulation setups both within the ROS2 framework and without, providing a very versatile customization. AR-sim can also generate random 3D race tracks for domain randomization and can simulate IMU, GNS, LiDAR, and fully customizable RGB cameras.

## 4.1 Results

AR-Sim demonstrated appealing performances in terms of simulation speed. Several tests have been made on a system with an Intel i7 processor, 16GB of RAM, and an NVIDIA GeForce GTX 1050 GPU. All measurements have been carried out using the Wharehouse scene and with a graphic quality level set to medium. The measured metric is the simulation rate, which corresponds to the time it takes to simulate the car dynamics and receive back the sensor readings from the Unity executable. Figure Figure 4.1 illustrates the simulator performance concerning the number and resolution of camera sensors. It can be seen that the resolution of the camera highly influences the simulation speed, the reason is due to the cost of transferring the data from the GPU to the CPU. Table 4.2 reports the simulation rate obtained for different LiDAR models, what influences the performance is only the number of channels and not the other LiDAR options.

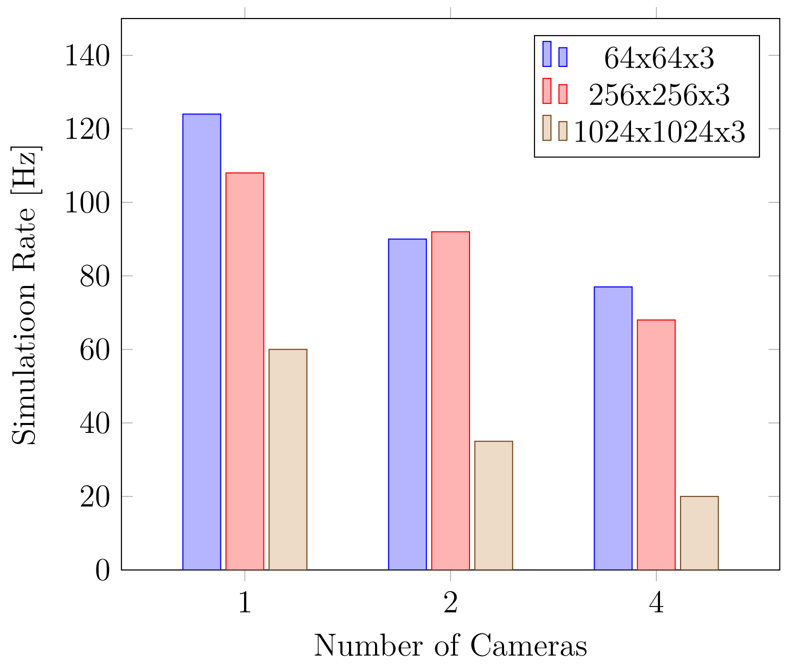


**Figure 4.1:** Simulation rate with different cameras configurations

Table4.4 compares the simulator performances for combined sensor setups, indicating the simulator's efficiency even on commonly available, mid-range hardware with a simulation. In particular, the first two rows indicate that it

| LIDAR Sensor | | |
|---|---|---|
| **Model** | **Resolution: (horizontalxvertical)** | **Simulation Rate [Hz]** |
| Hokuyo | 1080x1 | 140 |
| Ouster OS1 | 512x32 | 93 |
| Ouster OS1 | 1024x32 | 58 |
| Ouster OS1 | 1024x64 | 33 |

**Table 4.1**

**Table 4.2:** Simulation rate for different LiDARS

satisfies the requirement of enabling camera-based algorithms simulation for the F1Tenth vehicles.

| Combined car configuration | | |
|---|---|---|
| **Cameras** | **Lidar** | **Simulation Rate [Hz]** |
| 2 cameras: 256x512x3 | Hokuyo | 84 |
| 1 camera: 1024x760x3 | Hokuyo | 63 |
| 1 camera: 1024x760x3 | Ouster OS1 1024x32 | 28 |
| 1 camera: 256x512x3 | Ouster OS1 91024x32 | 33 |

**Table 4.3**

**Table 4.4:** Performance with combined sensor setup

The simulator can effectively handle multi-agent scenarios with different physic solvers running at the same time, such as the PhysX model, Gazebo, or any analytical equation model, e.g. the single track model. It also supports both autonomous operation and user input via ROS2 topics. Further, it integrates seamlessly with existing F1Tenth algorithms and can be used with packages like Hector SLAM. From Hector SLAM, it is possible to generate a CSV file of the map, using other open source software, which can then be used to create a 3D race track model inside the simulator with custom barrier meshes and materials.

Despite not yet being tested for end-to-end learning and Sim2Real transferability, the foundational aspects of AR-sim have been successfully established. The Python environment functions correctly, and future benchmarks will focus on these areas.

## 4.2 Future work

AR-sim is an evolving open-source software, continuously updated to include new features and improvements. An upcoming paper will be published, and it will provide comprehensive benchmark results, alongside its publication, the software will be released with a set of algorithms and detailed API documentation on the TUM-AVS Lab GitHub [1].

Future work will concern the development of more realistic Unity environments to close the simulation-to-reality gap by exploiting more advanced methods as photogrammetry to reproduce the TUM-AVS lab inside the virtual environments. Also, future work will include additional dynamic models and more camera simulation options such as depth images. The creation of random tracks will be enhanced adding the possibility to not only create 3D tracks by extruding a mesh profile but also by placing objects alongside the track, such as cones, to delimit the race track. For the F1Tenth use case, an additional useful feature would be the capability to place obstacles along the tracks as this is commonly used to test reactive autonomous algorithms.

Another aspect that hasn't been addressed is the system identification of the PhysX model of the F1Tenth race cars. This procedure requires external measurement equipment, such as the ViCon[2] motion capture system to obtain ground truth data, which was not available at the the time in the lab.

[1]https://github.com/TUM-AVS

[2]https://www.vicon.com/

Additionally, further work is needed to implement a parallelized simulation environment to speed up data gathering for ML application. At the moment, it is possible to simulate multiple vehicles simultaneously but they interact with each other. Instead would be beneficial to have them operate as ghosts for other viewpoints, allowing data collection for multiple vehicle instances at the same time within a single simulation epoch.

The work presented in this thesis marks a significant step forward in autonomous racing vehicle simulation, offering a robust, flexible, and user-friendly platform that enhances accessibility for researchers and students to experiment and test autonomous driving algorithms.

Appendix A

# AR-Sim Code Examples

**Code A.1:** Arim C++ code example

```cpp
int main()
{
  Arsim sim(ARSIM_LOAD_YAML("parameters/setting.yaml"));
  bool unity_ready = sim.connect2Unity();

  // simulation commands
  Commands_t c;

  //* Simulation loop example:
  while (unity_ready) {

    sim.step();
    sim.bridge_ptr->receiveFromUnity();

    // Computing command using the data, just example
    c = ComputeCommand( sim.cars()[0])

    // apply the command to the car
    sim.cars()[0]->setCommand(c);

    // send the command to unity
    sim.bridge_ptr->sendToUnity();
  }
  return 0;
}
```

**Code A.2:** Simulation yaml file settings

```yaml
SimulationSettings:
  asyncMode: false
  track: Track02
  scene: WareHouse
  simulationInterval: 0.02 # seconds
  stepSize: 0.01
  optionPhysxSolver: step

CarList:
  car0:
    profile: redcar1tenth_parameters.yaml # relative path from this file to the car yaml file
    position: [1,0,1]                # optional: set position of the car, if not present use the position inside the profile
    rotation: [0,0,10]               # optional: set the rotation of the car, if not present use the rotation inside the profile
```

**Code A.3:** Car yaml file configuration

```
initialPosition: [0,0,0]
initialOrientation: [0,0,0] # RPY in degrees

model: SingleTrack_cmroad # dynamic model
prefab: F1Tenth # prefab used in unity

CarParameters:
  lf: 0.15875
  lr: 0.17145
  h: 0.074
  m: 3.74
  Iz: 0.04712
  mu: 1.0489
  TireParameters:
    C_sf: 4.718
    C_sr: 5.456
  MotorParameters:
    C1: 21.85
    C2: 3.17
  LongitudinalParameters:
    Cr: 3.24
    Cd: 0.05
    v_switch: 0.5 # change between kinematic and dynamic
    v_max: 20
    v_min: -5
    a_max: 9.51
  SteeringParameters:
    max_angle: 0.418
    v_max: 3.2

Lidars:
  Lidar:
    # horizontal
    horizontalFov: 270        # [degree]
    startHorizontalFov: -135  # [degree]
    horizontalResolution: 1080
    # vertical
    verticalFov: 0
    startVerticalFov: 0
    verticalResolution: 1
    #
    maxLinearRange: 10 # [m]
    minLinearRange: 0 # [m]
    relPosition: [1,0,0]
    relRotation: [0,0,0] # degrees
    scanFreq: 10

    #Add here more Lidars
Cameras:
  camera:
    height: 760
    width: 1024
    fov: 60
    nearClipPlane: 0.1 # nearest plane the camera render
    farClipPlane: 1000 # further plane the camera render
```

**Code A.4:** dynamic model API

```cpp
int main()
{
  std::unique_ptr<cmroad::SingleTrack> single_track;
  single_track->loadParameters(ARSIM_LOAD_YAML("parameters/
   f1tenth_parameters.yaml"));

  CarPtr f1tenth = std::make_shared<Car>(single_track);
  IDynamics* dynamic_model = f1tenth->getDynamicModel();
}
```

**Code A.5:** JSON setting message

```json
{
  "settings": {
    "asyncMode": true,
    "sceneID": 2,
    "simInterval": 0.001,
    "solver": 1,
    "stepSize": 0.002,
    "trackID": 0
  },
  "vehicles": [
    {
      "ID": "car0",
      "cameras": [
        {
          "ID": "camera_0",
          "farClipPlane": 1000.0,
          "fov": 60.0,
          "height": 760,
          "nearClipPlane": 0.100000001490116,
          "positionRel": [-1.0, 1.0, 1.0],
          "rotationRel": [0.0, -0.0, -0.0, 1.0],
          "width": 1024
        }
      ],
      "commands": {
        "steering": 0.0,
        "throttle": 0.0
      },
      "is_kinematic": true,
      "lidars": [],
      "position": [-3.0, 1.0, 5.0],
      "prefabID": 0,
      "rotation": [0.0, -1.0, -0.0, 0]
    }
  ]
}
```

# Bibliography

[1] Arshardh Ifthikar and Saman Hettiarachchi. «Analysis of Historical Accident Data to Determine Accident Prone Locations and Cause of Accidents». en. In: *2018 8th International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*. Kuala Lumpur, Malaysia: IEEE, May 2018, pp. 11–15. ISBN: 978-1-5386-6539-8. DOI: 10.1109/ISMS.2018.00012. URL: https://ieeexplore.ieee.org/document/8699325/.

[2] Moneim Massar, Imran Reza, Syed Masiur Rahman, Sheikh Muhammad Habib Abdullah, Arshad Jamal, and Fahad Saleh Al-Ismail. «Impacts of Autonomous Vehicles on Greenhouse Gas Emissions—Positive or Negative?» en. In: *International Journal of Environmental Research and Public Health* 18.11 (May 2021), p. 5567. ISSN: 1660-4601. DOI: 10.3390/ijerph18115567. URL: https://www.mdpi.com/1660-4601/18/11/5567.

[3] Alan Ohnsman. *A Robotaxi Business Is A Dream For Elon Musk–But Already A Reality For Waymo*. en. URL: https://www.forbes.com/sites/alanohnsman/2024/06/17/a-robotaxi-business-is-a-dream-for-elon-muskbut-already-a-reality-for-waymo/.

[4] B. Padmaja, Ch. V. K. N. S. N. Moorthy, N. Venkateswarulu, and Myneni Madhu Bala. «Exploration of issues, challenges and latest developments in autonomous cars». en. In: *Journal of Big Data* 10.1

(May 2023), p. 61. ISSN: 2196-1115. DOI: 10.1186/s40537-023-00701-y. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00701-y.

[5] L. Sieber, C. Ruch, S. Hörl, K. W. Axhausen, and E. Frazzoli. «Improved public transportation in rural areas with self-driving cars: A study on the operation of Swiss train lines». In: *Transportation Research Part A: Policy and Practice* 134 (Apr. 2020), pp. 35–51. ISSN: 0965-8564. DOI: 10.1016/j.tra.2020.01.020. URL: https://www.sciencedirect.com/science/article/pii/S0965856418314083.

[6] Alexandros Nikitas, Alexandra-Elena Vitel, and Corneliu Cotet. «Autonomous vehicles and employment: An urban futures revolution or catastrophe?» In: *Cities* 114 (July 2021), p. 103203. ISSN: 0264-2751. DOI: 10.1016/j.cities.2021.103203. URL: https://www.sciencedirect.com/science/article/pii/S0264275121001013.

[7] TechCamp POLIMI. *Autonomous mobility: present and future, starting from the Indy Autonomous Challenge experience.* 2023. URL: https://www.youtube.com/watch?v=oleuy8JXPH4.

[8] Harprinderjot Singh, Mohammadreza Kavianipour, Mehrnaz Ghamami, and Ali Zockaie. «Adoption of autonomous and electric vehicles in private and shared mobility systems». In: *Transportation Research Part D: Transport and Environment* 115 (Feb. 2023), p. 103561. ISSN: 1361-9209. DOI: 10.1016/j.trd.2022.103561. URL: https://www.sciencedirect.com/science/article/pii/S136192092200387X.

[9] *Abu Dhabi Autonomous Racing League in UAE | A2RL.* en. URL: https://a2rl.io.

[10] Sebastian Thrun et al. «Stanley: The robot that won the DARPA Grand Challenge». en. In: *Journal of Field Robotics* 23.9 (Sept. 2006),

pp. 661–692. ISSN: 1556-4959, 1556-4967. DOI: `10.1002/rob.20147`. URL: `https://onlinelibrary.wiley.com/doi/10.1002/rob.20147`.

[11] Juraj Kabzan et al. *AMZ Driverless: The Full Autonomous Racing System*. May 2019. DOI: `10.48550/arXiv.1905.05150`. URL: `http://arxiv.org/abs/1905.05150`.

[12] Johannes Betz, Hongrui Zheng, Alexander Liniger, Ugo Rosolia, Phillip Karle, Madhur Behl, Venkat Krovi, and Rahul Mangharam. «Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing». en. In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 458–488. ISSN: 2687-7813. DOI: `10.1109/OJITS.2022.3181510`. URL: `https://ieeexplore.ieee.org/document/9790832/`.

[13] *Autonomous Challenge @ CES Rules*. en-US. URL: `https://www.indyautonomouschallenge.com/autonomous-challenge-ces-rules`.

[14] Alexander Liniger. «Path Planning and Control for Autonomous Racing». en. PhD thesis. ETH Zurich, 2018. DOI: `10.3929/ETHZ-B-000302942`. URL: `http://hdl.handle.net/20.500.11850/302942`.

[15] Eugenio Chisari, Alexander Liniger, Alisa Rupenyan, Luc Van Gool, and John Lygeros. *Learning from Simulation, Racing in Reality*. en. May 2021. URL: `http://arxiv.org/abs/2011.13332`.

[16] *AWS DeepRacer: il modo più rapido per partire con il machine learning*. it-IT. URL: `https://aws.amazon.com/it/deepracer/`.

[17] *Donkey® Car - Home*. URL: `https://www.donkeycar.com/`.

[18] Liam Paull et al. «Duckietown: An open, inexpensive and flexible platform for autonomy education and research». In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. May 2017, pp. 1497–1504. DOI: `10.1109/ICRA.2017.7989179`. URL: `https://ieeexplore.ieee.org/abstract/document/7989179`.

[19] Tanmay Vilas Samak, Chinmay Vilas Samak, and Ming Xie. «Auto-DRIVE Simulator: A Simulator for Scaled Autonomous Vehicle Research and Education». en. In: *2021 2nd International Conference on Control, Robotics and Intelligent System.* Aug. 2021, pp. 1–5. DOI: 10.1145/3483845.3483846. URL: http://arxiv.org/abs/2103.10030.

[20] *Goldeneye.* URL: https://goldeneye.studentorg.berkeley.edu/barc.html.

[21] Süleyman Eken, Muhammed Şara, Yusuf Satılmış, Münir Karslı, Muhammet Furkan Tufan, Houssem Menhour, and Ahmet Sayar. «A reproducible educational plan to teach mini autonomous race car programming». en. In: *International Journal of Electrical Engineering & Education* 57.4 (Oct. 2020), pp. 340–360. ISSN: 0020-7209. DOI: 10.1177/0020720920907879. URL: https://doi.org/10.1177/0020720920907879.

[22] Kyle Hart, Corey Montella, Georges Petitpas, Dylan Schweisinger, Armon Shariati, Ben Sourbeer, Tyler Trephan, and John Spletzer. «RoSCAR: robot stock car autonomous racing». In: *Proceedings of the 2014 workshop on Mobile augmented reality and robotic technology-based systems.* MARS '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 3–8. ISBN: 978-1-4503-2823-4. DOI: 10.1145/2609829.2609837. URL: https://doi.org/10.1145/2609829.2609837.

[23] Matthew O'Kelly et al. *F1/10: An Open-Source Autonomous Cyber-Physical Platform.* en. Jan. 2019. URL: http://arxiv.org/abs/1901.08567.

[24] Johannes Betz et al. «Teaching Autonomous Systems Hands-On: Leveraging Modular Small-Scale Hardware in the Robotics Classroom». In:

(2022). DOI: 10.48550/ARXIV.2209.11181. URL: https://arxiv.org/abs/2209.11181.

[25] Nicolas Baumann et al. *ForzaETH Race Stack – Scaled Autonomous Head-to-Head Racing on Fully Commercial off-the-Shelf Hardware.* en. Mar. 2024. URL: http://arxiv.org/abs/2403.11784.

[26] Li Chen, Penghao Wu, Kashyap Chitta, Bernhard Jaeger, Andreas Geiger, and Hongyang Li. *End-to-end Autonomous Driving: Challenges and Frontiers.* en. Apr. 2024. URL: http://arxiv.org/abs/2306.16927.

[27] Mariusz Bojarski et al. *End to End Learning for Self-Driving Cars.* en. Apr. 2016. URL: http://arxiv.org/abs/1604.07316.

[28] Alexey Dosovitskiy. «CARLA: An Open Urban Driving Simulator». en. In: ().

[29] Alexander Amini, Igor Gilitschenski, Jacob Phillips, Julia Moseyko, Rohan Banerjee, Sertac Karaman, and Daniela Rus. «Learning Robust Control Policies for End-to-End Autonomous Driving From Data-Driven Simulation». en. In: *IEEE Robotics and Automation Letters* 5.2 (Apr. 2020), pp. 1143–1150. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2020.2966414. URL: https://ieeexplore.ieee.org/document/8957584/.

[30] Sebastian Höfer et al. *Perspectives on Sim2Real Transfer for Robotics: A Summary of the R:SS 2020 Workshop.* en. Dec. 2020. URL: http://arxiv.org/abs/2012.03806.

[31] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. «Champion-level drone racing using deep reinforcement learning». en. In: *Nature* 620.7976 (Aug. 2023), pp. 982–987. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/

s41586-023-06419-4. URL: https://www.nature.com/articles/s41586-023-06419-4.

[32] Yueyuan Li, Wei Yuan, Songan Zhang, Weihao Yan, Qiyuan Shen, Chunxiang Wang, and Ming Yang. «Choose Your Simulator Wisely: A Review on Open-source Simulators for Autonomous Driving». en. In: *IEEE Transactions on Intelligent Vehicles* (2024), pp. 1–19. ISSN: 2379-8904, 2379-8858. DOI: 10.1109/TIV.2024.3374044. URL: http://arxiv.org/abs/2311.11056.

[33] Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, and Davide Scaramuzza. *Flightmare: A Flexible Quadrotor Simulator*. en. May 2021. URL: http://arxiv.org/abs/2009.00563.

[34] *Eclipse SUMO - Simulation of Urban MObility*. en. URL: https://www.eclipse.dev/sumo/.

[35] *CarSim | Speedgoat*. URL: https://www.carsim.com/.

[36] Yueyuan Li, Wei Yuan, Songan Zhang, Weihao Yan, Qiyuan Shen, Chunxiang Wang, and Ming Yang. «Choose Your Simulator Wisely: A Review on Open-source Simulators for Autonomous Driving». en. In: *IEEE Transactions on Intelligent Vehicles* (2024), pp. 1–19. ISSN: 2379-8904, 2379-8858. DOI: 10.1109/TIV.2024.3374044. URL: http://arxiv.org/abs/2311.11056.

[37] *Driving Simulator | VI-grade*. en. URL: https://www.vi-grade.com/en//.

[38] Cole Gulino et al. «Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research». In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. 2023.

[39] Mark Martinez, Chawin Sitawarin, Kevin Finch, Lennart Meincke, Alex Yablonski, and Alain Kornhauser. *Beyond Grand Theft Auto V for Training, Testing and Enhancing Deep Learning in Self Driving Cars*. en. Dec. 2017. URL: `http://arxiv.org/abs/1712.01397`.

[40] Matthias Althoff, Markus Koschi, and Stefanie Manzinger. «CommonRoad: Composable benchmarks for motion planning on roads». en. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. Los Angeles, CA, USA: IEEE, June 2017, pp. 719–726. ISBN: 978-1-5090-4804-5. DOI: `10.1109/IVS.2017.7995802`. URL: `http://ieeexplore.ieee.org/document/7995802/`.

[41] *nuPlan*. en. URL: `https://www.nuplan.org/nuplan`.

[42] *AWSIM document*. URL: `https://tier4.github.io/AWSIM/`.

[43] James Herman et al. *Learn-to-Race: A Multimodal Control Environment for Autonomous Racing*. en. Aug. 2021. URL: `http://arxiv.org/abs/2103.11575`.

[44] Benjamin D. Evans, Hendrik W. Jordaan, and Herman A. Engelbrecht. «Safe reinforcement learning for high-speed autonomous racing». en. In: *Cognitive Robotics* 3 (2023), pp. 107–126. ISSN: 26672413. DOI: `10.1016/j.cogr.2023.04.002`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S2667241323000125`.

[45] Hans B. Pacejka and Egbert Bakker. «The Magic Formula Tyre Model». In: *Vehicle System Dynamics* 21.sup001 (Jan. 1992), pp. 1–18. ISSN: 0042-3114. DOI: `10.1080/00423119208969994`. URL: `https://doi.org/10.1080/00423119208969994`.

[46] *NVIDIA Omniverse*. it-it. URL: `https://www.nvidia.com/it-it/omniverse/`.

[47] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. «F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning». In: *NeurIPS 2019 Competition and Demonstration Track*. PMLR. 2020, pp. 77–89.

[48] Varundev Suresh Babu and Madhur Behl. «f1tenth.dev - An Open-source ROS based F1/10 Autonomous Racing Simulator». en. In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. Hong Kong, Hong Kong: IEEE, Aug. 2020, pp. 1614–1620. ISBN: 978-1-72816-904-0. DOI: `10.1109/CASE48305.2020.9216949`. URL: `https://ieeexplore.ieee.org/document/9216949/`.

[49] Jonathan Francis et al. *Learn-to-Race Challenge 2022: Benchmarking Safe Learning and Cross-domain Generalisation in Autonomous Racing*. en. May 2022. URL: `http://arxiv.org/abs/2205.02953`.

[50] Kıvanç Güçkıran and Bülent Bolat. «Autonomous Car Racing in Simulation Environment Using Deep Reinforcement Learning». In: *2019 Innovations in Intelligent Systems and Applications Conference (ASYU)*. 2019, pp. 1–6. DOI: `10.1109/ASYU48272.2019.8946332`.

[51] TUM-Institute of Automotive Technology. *global race trajectory optimization*. URL: `https://github.com/TUMFTM/global_racetrajectory_optimization`.

[52] *f1tenth/f1tenth_racetracks*. May 2024. URL: `https://github.com/f1tenth/f1tenth_racetracks`.

[53] Alexander Heilmeier, Alexander Wischnewski, Leonhard Hermansdorfer, Johannes Betz, Markus Lienkamp, and Boris Lohmann. «Minimum curvature trajectory planning and control for an autonomous race car». en. In: *Vehicle System Dynamics* 58.10 (Oct. 2020), pp. 1497–1527. ISSN: 0042-3114, 1744-5159. DOI: `10.1080/00423114.2019.1631455`.

URL: `https://www.tandfonline.com/doi/full/10.1080/00423114.2019.1631455`.