

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Digital Twin Framework for Autonomous Racing Vehicles

Duarte Lima Silva



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Luís Tiago de Freixo Ramos Paiva

Co-Supervisor: Fernando Arménio da Costa Castro e Fontes

February, 2025

Resumo

Os carros autônomos estão-se a tornar cada vez mais populares, o que levou ao surgimento de um novo campo de pesquisa, as corridas autônomas. Com o rápido avanço da ciência e da tecnologia, tudo se tornou mais digital. Como resultado, é possível simular tanto o hardware quanto o software do carro autônomo através de um programa de computador, denominado *digital twin* (gêmeo digital), uma réplica digital de um objeto ou sistema físicos utilizado para monitorização, análise e otimização. O uso de *digital twins* traz muitas vantagens ao desenvolvedor, como poder identificar se o software apresenta bugs e testar os limites sem ter consequências na vida real. No entanto, para que o *digital twin* represente o sistema real de forma precisa, é necessário criar um ambiente preciso.

Esta dissertação tem como objetivo desenvolver um *digital twin* de um sistema composto por uma pista e carros de corrida numa escala de 1:43 utilizando o motor de jogo *Unity*. O objetivo é possibilitar a simulação de carros de corrida na pista, permitir a monitorização em tempo real dos carros de corrida com base nas suas coordenadas na pista real, e facilitar o desenvolvimento e treino de algoritmos de *machine learning*.

Primeiro, é realizada uma revisão completa da literatura de corridas autônomas e *digital twins*, nomeadamente projetos de *digital twins* relevantes. Posteriormente, o sistema da pista e do carro de corrida é analisado, juntamente com as equações utilizadas para modelar o carro de corrida. De seguida, o ambiente do *digital twin* é desenvolvido modelando os carros de corrida e a pista de corrida e, posteriormente, um algoritmo de monitorização e uma framework de *machine learning* são implementados.

O *digital twin* simula com sucesso o sistema pista e carro de corrida na escala de 1:43, demonstrando com precisão a funcionalidade do sistema. O sistema de monitorização em tempo real permite recolher as posições dos carros de corrida e funciona independentemente dos componentes da vida real, estando operacional devido à criação de um sistema de perceção virtual para compensar a sua ausência. Além disso, o *digital twin* desenvolvido neste trabalho oferece diversas funções, incluindo a capacidade de enviar comandos de movimento para carros da vida real e de obter estatísticas sobre o seu movimento, como velocidade máxima e número de voltas completadas. Ele também armazena informações importantes numa base de dados. Essas funções oferecem um recurso valioso para o desenvolvimento e teste de sistemas de controlo avançados. A framework de *machine learning*, implementada no *digital twin*, mostra resultados promissores na melhoria do desempenho do carro de corrida. Os algoritmos testados demonstram uma melhor tomada de decisão e adaptabilidade em ambientes de corrida dinâmicos, levando a estratégias de controlo mais eficientes e eficazes. O ambiente digital do *digital twin* prova ser uma plataforma robusta para testar e refinar esses algoritmos de *machine learning*, contribuindo para avanços na tecnologia de corrida autónoma.

Palavras-chave: corrida autónoma, digital twin, simulação, ambiente virtual, monitorização em tempo real, machine learning

Abstract

Self-driving cars are becoming more and more popular, which has led to the rise of a new research field: autonomous racing. With the rapid advancement of science and technology, everything has become more digital. As a result, both the hardware and software of the self-driving car can be simulated through a computer program, called a digital twin, a digital replica of a physical object or system used for monitoring, analysis and optimization. The use of digital twins gives a lot of advantages to the developer, such as being able to identify if the software has any bugs and testing the limits without having real-life consequences. Nevertheless, for the digital twin to accurately represent the real system, an accurate environment must be created.

This dissertation aims to develop a digital twin of a race track and race car system in the scale of 1:43 using the game engine *Unity*. Its purpose is to enable the simulation of the race cars on a race track, permit real-time monitoring of the race cars based on their coordinates on the real race track, and facilitate the development and training of machine learning algorithms.

First, a thorough literature review on autonomous racing and digital twins, including relevant digital twin projects, is conducted. Afterward, the race track and race car system is analyzed, along with the equations used to model the race car's movement. Following this, the digital twin environment is developed by creating the physical models of the race cars and race track, as well as algorithms that dictate their behavior. Then, a monitoring algorithm is developed, and finally, a machine learning framework is implemented.

The digital twin successfully simulates the 1:43 scaled race track and race car system, accurately demonstrating the functionality of the system. The real-time monitoring system enables precise tracking of the positions of the race cars and works regardless of the real-life components being operational due to the creation of a Virtual Perception System (VPS) to compensate for its absence. Additionally, the digital twin developed in this work provides several other functions, including the ability to send instructions to the real cars and to gather statistics on their movement, such as maximum speed and the number of laps completed. It also stores the position of the cars, direction, and velocity in a database. These features offer a valuable resource for the development and testing of advanced control systems.

Finally, the machine learning framework, implemented within the digital twin, not only shows promising results in enhancing the performance of the race car but also demonstrates improved decision-making and adaptability in dynamic racing environments, leading to more efficient and effective control strategies. The virtual environment of the digital twin proves to be a robust platform for testing and refining these machine-learning algorithms, contributing to advancements in autonomous racing technology.

Keywords: autonomous racing, digital twin, simulation, virtual environment, real-time monitoring, machine learning

United Nations Sustainable Development Goals (SDGs)



Figure 1: SDG 4, 9 and 11 [1]

The Sustainable Development Goals (SDGs), also referred to as the Global Goals, represent a universal call to action embraced by the United Nations to end poverty, protect the planet, and ensure peace and prosperity for all people by the year 2030. This set of 17 interlinked goals provides a shared blueprint for a sustainable future, not just for human societies but for the planet as a whole.

In the spirit of innovation and progress, the development of digital twins for autonomous racing is a notable example of how technology can directly contribute to the achievement of several of these Global Goals, particularly 4 (Quality Education), 9 (Industry, Innovation, and Infrastructure), and goal 11 (Sustainable Cities and Communities). This chapter highlights the significant role that autonomous racing and digital twin technologies play in advancing sustainable development and fostering a future where technology and sustainability go hand in hand, ultimately driving toward achieving the SDGs.

SDG 4 - Quality Education

- **Enhancing STEM Literacy (Target 4.1):** The project aligns with the goal of achieving quality education, as the advancement and integration of autonomous racing technologies, along with the development and use of digital twins, foster educational and skill enhancement opportunities within the realms of science, technology, engineering, and mathematics (STEM). This not only aids in the cultivation of a skilled workforce in these sectors but also contributes to the overarching aim of providing inclusive and equitable education.

SDG 9 - Industry, Innovation, and Infrastructure

- **Sustainable Infrastructure (Target 9.1):** The push for autonomous racing necessitates establishing cutting-edge infrastructure such as intelligent transportation systems, comprehensive communication networks, and energy-efficient facilities. Investments in such in-

Table 1: Contributions and Performance Indicators of the SDGs

SGD	Target	Contribution	Performance Indicator and Metrics
4	4.1	Enhancing STEM Literacy	Enrollment Increase
9	9.1	Sustainable Infrastructure	Infrastructure Advancement Energy Efficiency
	9.4	Clean Technologies Adoption	Resource Efficiency Operational Cost Reductions
	9.5	Scientific Research and Innovation	Technological Advancements
11	11.2	Improved Transport Efficiency	Congestion Reduction Emissions Reduction Accident Reduction Rate

frastructure not only bolster the automotive sector but also have the potential to positively impact the wider transportation network, promoting the creation of resilient, sustainable infrastructures.

- **Clean Technologies Adoption (Target 9.4):** The advancement of autonomous racing technologies signifies the creation of highly efficient algorithms that enhance the sustainability of vehicles, making them cleaner and more environmentally friendly. Conversely, digital twins stand out as a sustainable technology due to their inherent advantage of zero emissions and lower costs compared to their physical counterparts. Therefore, the integration of digital twins is instrumental in promoting the adoption of clean technologies, aligning with the objectives of SDG 9.4.
- **Scientific Research and Innovation (Target 9.5):** The advancement of digital twins and the integration of autonomous racing technologies represent pivotal contributions to the objective of enhancing scientific research and innovation, central to SDG 9.5. Digital twins offer a groundbreaking approach by facilitating the detailed simulation and analysis of infrastructures in virtual environments, thus accelerating the innovation cycle and promoting the application of optimized solutions in real-world settings. Similarly, autonomous racing technologies, through their development and application, push the boundaries of what's possible in robotics and artificial intelligence, offering valuable insights and fostering technological advancements that can be applied across various sectors.

SDG 11 - Sustainable Cities and Communities

- **Improved Transport Efficiency (Target 11.2):** The advancement of autonomous vehicle technology, complemented by digital twins and autonomous racing technology, significantly enhances sustainable city transitions by streamlining traffic and reducing congestion, emissions, and accidents. This integration not only optimizes transportation systems for efficiency but also leads to more eco-friendly and safer urban environments. Through smarter planning and cleaner vehicle innovations, these technologies support the creation of sustainable, resilient communities with improved urban mobility and quality of life, lining up with SDG 11.2.

Agradecimentos

Gostaria de agradecer ao meu orientador, professor Luís Tiago Paiva, não só pela oportunidade de trabalhar nesta dissertação, sendo um tema bastante interessante e que me permitiu crescer enquanto estudante e enquanto pessoa, mas também pelo seu apoio ao longo do desenvolvimento da dissertação.

Ao meu co-orientador, professor Fernando Fontes, pelo acompanhamento e ajuda dada ao longo desta longa etapa, tendo sido crucial o seu apoio.

Ao Conrado, ao Gabriel e ao Rui, que me auxiliaram durante fases críticas da tese.

Aos meus colegas do laboratório, que foram sempre prestáveis e estavam sempre dispostos a ajudar.

Aos meus amigos, pela ajuda e pela motivação dada durante os últimos anos.

Por fim, agradeço à minha família, pelo apoio dado não só nesta fase mas ao longo da minha vida, pois sem ela não seria possível estar aqui.

Pelos materiais fornecidos para o projeto, gostaria de agradecer:

Ao LSCOE (Laboratory of Systems, Control, Optimization, and Estimation) SYSTEC-ARISE, localizado no laboratório i202 DEEC-FEUP

E ao apoio da FCT/MCTES (PIDDAC) através dos projetos 2022.02320.PTDC, UIDB/00147/2020&UIDP/00147/2020 SYSTEC, e LA/P/0112/2020 ARISE.

Contents

1	Introduction	1
1.1	Context and Motivation	1
1.2	Goals	2
1.3	Contributions	3
1.4	Dissertation Structure	3
2	Digital Twin and Autonomous Racing: Literature Review	5
2.1	Autonomous Racing - State of the Art	5
2.1.1	Racing Competitions	6
2.2	Digital Twin - State of the Art	7
2.2.1	Evolution of Simulation	7
2.2.2	Origin of the Twin	8
2.2.3	Functionality	8
2.2.4	Similar Works	9
2.2.5	Digital Twin Evolution Stages	10
2.2.6	Implementation	14
3	Car and Track System	16
3.1	System Overview	16
3.1.1	Race Track	17
3.1.2	Race Car	17
3.1.3	Camera	19
3.2	Perception	20
3.2.1	Calibration and Distortion	20
3.2.2	Race Car Detection	21
3.2.3	Estimation of Speed, Position and Orientation	21
3.3	Control	22
3.3.1	Pygame Simulation Environment	22
3.3.2	Unity Simulation Environment	23
4	Digital Twin's Object Modeling	25
4.1	Physical Modeling - Race Track Models	25
4.1.1	Simplified Race Track Model	25
4.1.2	Final Race Track Model	28
4.2	Physical Modeling - Race Car Models	31
4.2.1	Simplified Race Car Model	31
4.2.2	Final Race Car Model	32
4.3	Modeling in Unity	34

4.3.1	Rigid Body and Colliders	34
4.3.2	Script Implementation - Initial Version	35
4.3.3	Script Implementation - Unity Refined Code	38
4.3.4	Virtual Cameras	40
5	Digital Twin Functionality and Monitoring	42
5.1	Digital Twin Functionality	42
5.2	Monitoring	45
5.2.1	Virtual Perception System	45
5.2.2	Virtual and Ghost Cars	46
5.2.3	Virtual Cars' Real-time Monitoring	48
5.3	Machine Learning Integration	56
5.3.1	ML-Agents Implementation	56
5.3.2	Race Car Scripts Modification	57
5.4	Results	57
6	Conclusion	60
6.1	Future Work	61
	References	63
A	Scripts	66
A.1	Refined Car Movement Script	66
A.2	Car Monitoring Script	71
A.3	Car Statistics Script	80

List of Figures

1	SDG 4, 9 and 11 [1]	vii
2.1	Waves of Simulation [2]	7
2.2	Simulators at Mission Control [3]	8
2.3	AutoDRIVE Simulation Used by [4]	9
2.4	Unity Simulation Used by [5]	10
2.5	Traditional Digital Twin Evolution Model [6]	11
2.6	5-layer Implementation Model for a Digital Twin [6]	11
2.7	Digital Virtualization Technologies [6]	12
2.8	Digital Twin Synchronization Technologies [6]	13
2.9	Modeling and Simulation Technologies [6]	13
2.10	MATLAB's 3D Interface for Vehicle Dynamics Blockset [7]	14
2.11	CARLA Simulator Interface [8]	15
2.12	Unity 3D Interface Example [9]	15
3.1	System Overview [10]	16
3.2	Race Track	17
3.3	The Three Race Cars	18
3.4	Car with PCB and ESP32	18
3.5	Overhead View [11]	19
3.6	Bicycle Model [11]	19
3.7	GoPro HERO10 Black [12]	20
3.8	ArUco Marker	21
3.9	Race Car Detection	21
3.10	Pygame's Car Sensors and Trajectory	22
3.11	Pygame's Car with Cross-Track error	23
3.12	Race Track's Checkpoints	23
3.13	Race Track Trajectory	23
4.1	The Seven Pieces Merged	26
4.2	Race Track's Surface (first step)	26
4.3	Race Track with Outer Walls (second step)	26
4.4	Race Track Outer Curves (third step)	27
4.5	Simplified Race Track (fourth step)	28
4.6	Inner Faces Created Inside an Outer Curve	28
4.7	Outer Curve With One Side Wall	29
4.8	Finished Outer Curve	29
4.9	Outer Curve Without Inner Faces	29
4.10	U-turn's Cut Cylinders	29

4.11 Race Track U-turn Curve (new fourth step)	29
4.12 Race Track with all the Walls and Curves Finished (fifth step)	30
4.13 Final Race Track Model	30
4.14 Smaller Race car Model's axle (first step)	32
4.15 Smaller Race car with Both Rear and Front Wheels (second step)	32
4.16 Smaller Race Car (third step)	32
4.17 Race Car with Both Rear and Front Wheels (first and second step)	33
4.18 Race Car with the Side Structure (third step)	33
4.19 Finished Physical Car Model (fourth step)	33
4.20 Car Wheels' Box Colliders	34
4.21 Car Body's Mesh Collider	34
4.22 Car with (Blue) and without (Purple) the Deceleration Force	37
4.23 Car With Default (Centered) Steering	38
4.24 Car Steering to the Left	38
4.25 ESP32 Pulse Function	39
4.26 Car Steering from Default (left) to Maximum (right)	40
4.27 Virtual Race Track Camera View	41
4.28 Real Race Track Camera View	41
4.29 Virtual Race Car Camera View	41
5.1 Cyber-Physical System Overview	43
5.2 Digital Twin Overview	44
5.3 Cyber-Physical System Without the Virtual Perception System	45
5.4 String sent from the Perception System	46
5.5 Virtual And Ghost Cars	46
5.6 Unity Collision Matrix	47
5.7 Ghost Car Losing its Horizontal Aligment	48
5.8 Virtual Car Faulty Positioning Resulting from a Collision	48
5.9 Communication Delay of 0s	49
5.10 Communication Delay of 0.5s	49
5.11 Virtual Car in its Last Recorded Safe Position	50
5.12 Virtual Car After Bypassing the Safe Position	51
5.13 Values Inserted in the Database (with Filter)	52
5.14 Finish Line Collision Box	54
5.15 Race Track Checkpoints	54
5.16 Individual Virtual Car's Statistics Display	55
5.17 Global Virtual Cars' Statistics Display	55
5.18 RollerBall Environment	56
5.19 Virtual Race Track Replicating the Real Race Track	58
5.20 Unity Controlling the Real and Virtual Car Simultaneously	58
5.21 Unity Communicating with the Three Cars	59
6.1 Car Trajectory Example	62

List of Tables

1	Contributions and Performance Indicators of the SDGs	viii
4.1	Smaller Car Specifications	31
4.2	1:43 Car Specifications	33
4.3	Real Race Car Circle Test Data	35
4.4	Car Commands	39

Abbreviations

GPS	Global Positioning System
LiDAR	Light Detection and Ranging
RADAR	Radio Detection and Ranging
ECU	Electronic Control Unit
NPC	Non-Player Character
FSM	Finite State Machine
API	Application Programming Interface
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
PCB	Printed Circuit Board
VC	Virtual Car
GC	Ghost Car
OM	Operations Module
RPS	Real Perception System
VPS	Virtual Perception System
FOV	Field of View
HSV	Hue, Saturation, Value
PID	Proportional-Integral-Derivative
PPO	Proximal Policy Optimization

Chapter 1

Introduction

This chapter is divided into four main sections. The section 1.1 provides an overview of the context and motivations behind this dissertation, emphasizing the current significance of autonomous racing, associated challenges, and the pivotal role of digital twins in addressing these issues. Section 1.2 outlines the objectives this dissertation seeks to fulfill. Section 1.3 details the advancements and contributions made to the scientific community through this work. Finally, section 1.4 delineates the structure of this dissertation.

1.1 Context and Motivation

Over the past decades, racing competitions like Formula 1, Indy Car, and the World Rally Championship have become increasingly popular. This rise in interest, coupled with the rapid advancement of autonomous driving technology, has given rise to the first autonomous racing competition: the DARPA Grand Challenge [13]. This event has inspired the creation of several other competitions, including Roborace, Indy Autonomous Challenge, and Formula Student Driverless [14]. These competitions have emphasized the need for the development of efficient software that enables racing cars to complete the races in the shortest time possible.

Testing software on a real car can be a costly affair due to the expenses associated with the initial cost and maintenance of the vehicle, such as the cost of fuel or even replacing the car's wheels, which can add up to a significant amount. Therefore, most car software testing is usually done towards the end of the development phase, when the software is nearly complete and the developers are ready to validate the software's functionality on a real car. Moreover, if the software contains bugs, this can lead to very expensive consequences, like the car crashing. To overcome this challenge, developers are turning to digital twins to simulate real-world environments on computers. The digital twin is a powerful tool that enables the simulation of a realistic environment the race car will encounter, all through a computer. Due to the simulation being digital, it is safe and allows the developer to test repeatedly, without risking any accidents or damages and without the maintenance cost of the vehicle while also enabling the developer to monitor and analyze the system in real-time. As autonomous racing progresses, the implementation of digital twins grows

progressively vital for the advancement of technology and competitive edge. This thesis delves into the utilization of digital twins to improve the development and testing of an autonomous racing system, composed of a race track and race cars on a scale of 1:43.

1.2 Goals

This work is part of a project consisting of three key works. Firstly, the development of a perception system that identifies the location of the race cars on the race track using a camera, and sends these coordinates to a central system, executed by [15]. Secondly, the creation of a control system for race cars using machine learning, specifically employing reinforcement learning techniques, implemented by [16]. Its aim is to enable the control of the car's movement and reduce the time it takes to complete each lap of the race track. Finally, the main focus of this dissertation is the development of a digital twin framework.

The digital twin must accomplish two main objectives: Firstly, it must enable real-time monitoring of the race cars directly on the race track. Since there is no perception system in the real world available, a Virtual Perception System needs to be implemented to gather positions of the digital cars on the race track. Moreover, it aims to accurately simulate race cars on the race track. This functionality is not only pivotal for visual representation but it is also essential to facilitate the creation and training of reinforcement learning algorithms within the control work.

To guarantee the successful achievement of these objectives, it is essential that both the race cars and the track are modeled with a certain accuracy. This accuracy is vital for ensuring that the simulated outcomes are not only closely aligned but also virtually indistinguishable from real-world scenarios. By doing so, the reliability and effectiveness of the digital twin are enhanced.

To achieve this, the dissertation focuses on achieving several core objectives:

- Research the literature regarding autonomous racing and digital twins;
- Develop reliable models of the race track and the race cars;
- Create a virtual environment where the algorithms and the models of the race cars and race track can be tested;
- Construct an algorithm that can communicate with the real cars and send them instructions;
- Design an algorithm that allows for the modeled race cars to be positioned in the virtual environment based on the coordinates provided by the perception work, allowing for real-time monitoring;
- In that same environment, implement a framework that allows the control work to develop and train its machine learning algorithms.

1.3 Contributions

The work presented in this dissertation makes several contributions to the field of digital twins in the context of autonomous racing.

- A thorough literature review regarding autonomous racing and digital twins;
- Development of the models for the race track and the race cars, for both their physical traits and the algorithms that dictate their behavior;
- Creation of a virtual environment where the cars and the race track can be tested;
- Design of an algorithm that not only allows for the cars to be monitored in real-time in the created environment but also ensures that the monitoring positions were not disturbed by the cars being either grabbed or leaving the race track;
- Implementation of a Virtual Perception System that takes the place of the real one, in case it is not operational, allowing the digital twin to work even if the real-life counterparts are not available;
- The virtual interface gathers and displays several statistics: average speed, maximum speed, distance traveled, current lap time, laps completed, and idle time. These values can be used to understand better how each of the cars behave on the race track;
- Important values are stored in a database, like cars' position, velocity, and direction, facilitating the training of machine learning algorithms;
- Implementation of a framework that allows the development and training of machine learning algorithms.

1.4 Dissertation Structure

This dissertation consists of six chapters. Following the introduction, it unfolds in the following manner:

- Chapter 2 offers a literature review covering autonomous racing and digital twins, including related competitions, its history, tools to implement it, and works similar to the one presented in this dissertation;
- Chapter 3 provides a description of the race track and the race car's system. It also includes the equations used to control the digital car's movement and concludes with a summary of the work developed by the control and perception works;
- Chapter 4 details the steps taken in creating the physical models and the scripts that dictate the behavior of the race cars and the race track for both the initial version and the final one, including the methods used to obtain the cars' parameters;

- Chapter 5 presents an overview of how the complete system functions, even when the real-life components or some of the virtual ones are not being used. It also provides details on the development of the monitoring phase, covering everything from the communication algorithm to the various values that were collected, calculated, stored, and displayed (in the case of the statistics), ensuring that the digital twin is utilized to its fullest potential. Additionally, it discusses the implementation of the machine learning framework for the digital twin and concludes with a comparison to existing works.
- Chapter 6 summarizes the conclusions of this dissertation, including the achieved goals, encountered problems, and future research recommendations.

Chapter 2

Digital Twin and Autonomous Racing: Literature Review

In this chapter, the current state of autonomous racing, along with its related competitions, is explored in section 2.1. The present state of the digital twin, its history and evolution over time, including a process for developing a digital twin and the introduction of various tools that facilitate its creation, are discussed in section 2.2.

2.1 Autonomous Racing - State of the Art

Autonomous racing has gained a lot of attention in recent years, especially in the field of artificial intelligence. The development of autonomous racing cars requires a lot of expertise in various fields, such as electronics, mechanics, and computer science. Engineers and programmers work together to design and build a car that can race autonomously and make real-time decisions based on the data collected by sensors.

The autonomous race cars must have 4 wheels, an electrical or combustion engine as the main power source, and can be either a real race car or, as is the case of this work, a smaller one [17]. They are typically equipped with several sensors like light detection and ranging (LiDAR), global positioning system (GPS), radio detection and ranging (RADAR), and cameras to provide them with a 360-degree view of their surroundings. These sensors help the car detect obstacles, other cars, and even the track layout. The microprocessor on the car then processes the data collected by the sensors and makes decisions on how to control the vehicle. The car must make real-time decisions with minimal reaction times based on the information available in order not to hit any other car and to improve its racing strategy by finding the best optimal route, all with the goal of finishing the race in the least time possible. An example of this is ETH Zurich's race car, which utilizes these techniques, showcasing its effectiveness and precision in the realm of high-performance autonomous racing [18].

The algorithms implemented must be robust enough to handle any situation that may arise during the race, such as sudden changes in track conditions, other cars blocking the way, or even

unpredictable weather conditions. The car must be able to adapt quickly to these changes and make the best decision possible to complete the race in the shortest amount of time.

2.1.1 Racing Competitions

There are many racing competitions that focus on autonomous vehicles, providing platforms for innovation and testing. Some of the most relevant competitions include:

DARPA Grand Challenge

The DARPA Grand Challenge was the first competition for autonomous vehicles [13]. The vehicles had to travel hundreds of miles across rough terrain without any human intervention in less than 10 hours, and the winner would receive a 1 million dollar prize [19][20]. In the 2004 competition, despite multiple participants, none of them was able to claim the prize since the longest distance one of the competitors covered was 7.3 miles. However, in the competition held a year later, the Stanford Racing Team won the challenge with the increased prize of 2 million dollars [21].

Roborace

Roborace is the first global championship for autonomous racing [22], created by the UK-based company of the same name. The competition features autonomous cars equipped with advanced sensors for fully autonomous racing. Teams compete by developing software to run on the standardized hardware platform. The competition format includes single and multi-vehicle races and introduces virtual obstacles on the track, which add a challenging dynamic for the teams, as hitting an obstacle results in a time penalty [17].

Indy Autonomous Challenge

The Indy Autonomous Challenge represents an international contest that motivates college students to innovate in creating autonomous vehicle software for driverless racecars. Echoing Roborace, this competition is dedicated exclusively to software development since the hardware configuration remains the same across all participating teams [22]. The Indy Autonomous Challenge vehicle is equipped with a camera, LiDAR, and RADAR sensors for perception and contains one main electronic control unit (ECU) to run the autonomous software [17].

Formula Student Driverless

This contest encourages teams of college students to conceive, design, build, and compete with compact, formula-inspired autonomous racing vehicles [23].

The competition features cars in various single-car challenges, including acceleration (a 75m straight-line sprint from a standstill), skid pad (navigating two identical circles with an 18.25m diameter), autocross (competing on a closed-circuit track with an undisclosed layout), track drive

and efficiency (completing 10 laps on a circuit while being scored for energy efficiency). Beyond these driving events, the vehicles are also evaluated in a business plan competition, design, and cost [17].

F1TENTH

F1TENTH is a competition where participants, design and race 1/10th scale autonomous vehicles. Through this hands-on experience, the participants acquire valuable knowledge, culminating in a collaborative algorithm racing event [24]. Some universities also use F1TENTH in their courses.

One variation of this contest employs the AutoDRIVE Ecosystem to create and simulate a digital twin of an F1TENTH racecar on a virtual track. In this version, hardware is standardized, and each team has access to the digital twin simulation alongside the functional AutoDRIVE Devkit, enabling them to develop and test their algorithms effectively. Teams are tasked with creating algorithms for perception, planning, and control that interpret real-time sensor data streamed from the simulator and generate control commands to be relayed back to the simulated vehicle [25].

2.2 Digital Twin - State of the Art

2.2.1 Evolution of Simulation

Decades ago, the usage of simulation was limited to a small group of individuals (Fig. 2.1), primarily comprising computer experts and mathematicians, who employed it to compute particular physical phenomena for the purpose of addressing design challenges. With the increasing availability of personal computers and workstations, simulation tools for tasks like control unit design began to emerge. This led to a surge in the usage of simulations, which in turn boosted the growth of simulation tool providers. The expansion of the simulation industry has resulted in the development of more powerful computing capabilities for simulations. Nowadays, simulation plays a vital role in making design decisions, validating, and testing components and entire systems across various application fields. This trend is expected to continue advancing in the years to come [2].

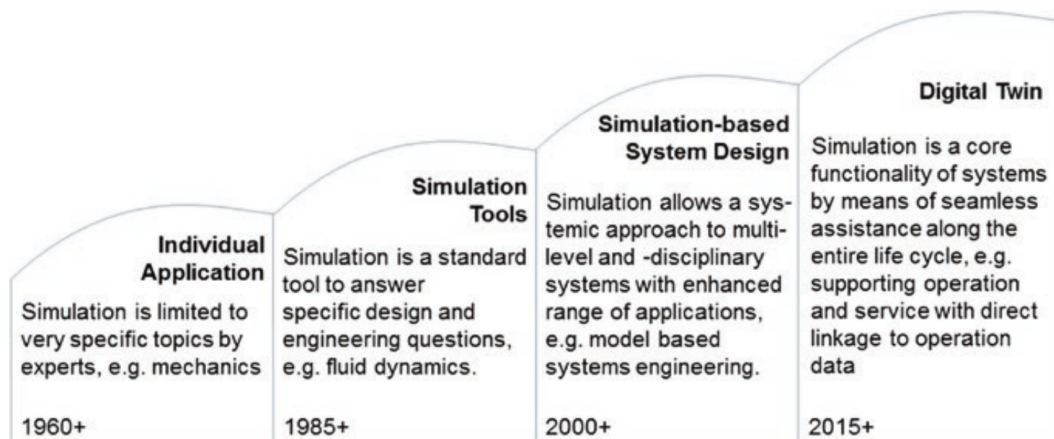


Figure 2.1: Waves of Simulation [2]

2.2.2 Origin of the Twin

The concept of a "twin" originated in NASA's Apollo project, where it served the purpose of replicating the conditions of the environment of the space vehicle during its mission. Two vehicles were built: one would go to space, and the other, the twin, would remain on Earth (Fig. 2.2). This allowed engineers to simulate alternatives on the twin during the mission and use the flight data gathered to help the astronauts in orbit during critical situations. Since then, any prototype used to mirror real operating conditions for the simulation of real-time behavior can be considered a twin.

Another example of a twin is the "Iron Bird", a tool used in the aircraft industry to integrate and validate aircraft systems. It allows engineers to test system components and discover any incompatibilities that may require modifications during the early development stages. Due to the growing strength of simulation technologies, virtual models are taking the place of physical components. The idea behind the Iron Bird is now utilized in earlier stages of development, resulting in the formation of complete digital models of physical systems, commonly referred to as digital twins [2].



Figure 2.2: Simulators at Mission Control [3]

2.2.3 Functionality

Being one of the main technologies of Industry 4.0 [26], a digital twin is a powerful tool that incorporates cutting-edge technologies such as the Internet of Things (IoT), smart algorithms, 5G, and cloud computing [27][28]. Using only a computer, it creates a virtual representation of a real-world system, such as a factory, aircraft, or autonomous racing car. It enables developers to optimize their system while in a remote environment although it can be directly connected to the physical system, allowing the digital twin to represent the physical system in real-time, making monitoring and analysis more efficient and effective [29]. As the real system evolves, so does the digital twin, which will assimilate any knowledge gathered from the real system. Additionally, it gives developers the ability to test their systems under different conditions. Digital twins offer numerous benefits, but they also come with some drawbacks, such as their reliance on highly

accurate models; otherwise, they may generate inaccurate predictions. Creating a complete model for each system from scratch requires a complex implementation that is challenging to achieve. To address this issue, co-simulation is commonly used, where multiple simulation tools or models from different domains work together in a synchronized manner to simulate a complex system [29]. Typically, each component or subsystem is simulated using a specialized simulation tool, and these tools collaborate to simulate the entire system.

2.2.4 Similar Works

A Scalable and Parallelizable Digital Twin Framework for Sustainable Sim2Real Transition of Multi-Agent Reinforcement Learning Systems

The study conducted by [4] reveals a scalable multi-agent reinforcement learning (MARL) framework for autonomous vehicles, utilizing a digital twin methodology for efficient policy transfer from simulations to physical vehicles. Initially, policies were trained and validated within a simulation before they were applied to a real-world vehicle incorporated in a digital twin framework, which employs LiDAR-based odometry for accurate state estimation (Fig. 2.3). This arrangement permits the real-world vehicle to engage with virtual counterparts. The digital twin dynamically updates the virtual environment with real-world data, thus enabling the real vehicle to implement actions devised in the virtual realm. Afterward, the vehicles were tested on a racetrack, where they were trained to lower lap times and avoid collisions through a hybrid approach combining imitation learning and reinforcement learning. The training process was significantly improved by incorporating a neural network along with the technique of parallelization. This allowed for running multiple simulation instances simultaneously. Parallelization ensured that each agent operated in its own virtual layer, interacting only with relevant counterparts. This strategy greatly increased the efficiency of the training by cutting down on computational constraints, thereby speeding up the rate of policy learning and adaptation.

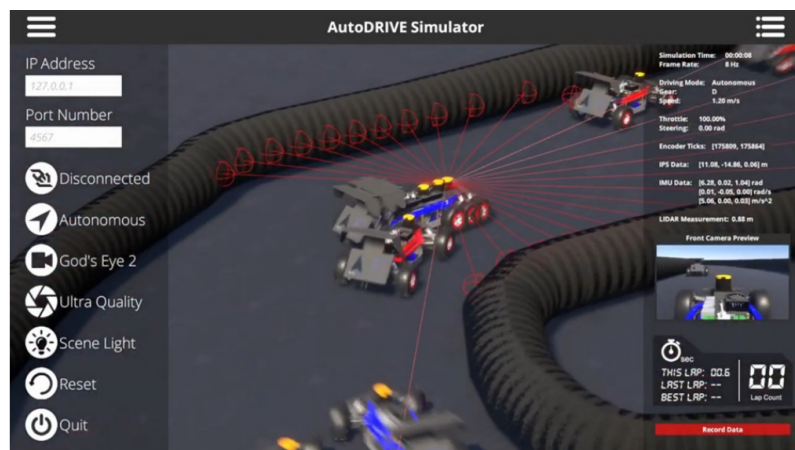


Figure 2.3: AutoDRIVE Simulation Used by [4]

Digital Twin Simulation of Connected and Automated Vehicles with the Unity Game Engine

In the study presented by [5], the Unity game engine was used to simulate four distinct types of vehicles: legacy vehicles (lacking connectivity and automation), connected vehicles (with no automation), automated vehicles (without connectivity), and connected and automated vehicles. In this study, a road with 3 lanes was created (Fig. 2.4), and several non-player characters (NPC) vehicles were generated on each lane. A driver would control a specific digital vehicle and follow the preceding vehicle. Data collected from this process was then used to train the algorithm for automated vehicle control.

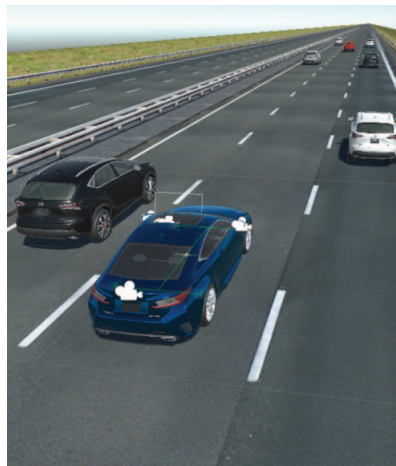


Figure 2.4: Unity Simulation Used by [5]

2.2.5 Digital Twin Evolution Stages

Creating a digital twin of a real-life system involves several stages due to its complexity. In Gartner's digital twin evolution model (Fig. 2.5), there are three stages that a digital twin goes through.

In the first stage of creating a digital twin, replicas of necessary objects are made in the digital environment. However, it is only crucial to replicate the required objects to avoid making this process complicated. For example, in the case of this work, the primary objects are the race cars and the race track. During the second stage, the digital twin monitors and controls the physical object based on its analysis. The third stage involves simulating the replicated objects to optimize the real world or solve problems. This results in improved and efficient operations, as well as the ability to test and simulate various scenarios before implementing changes in the real world [6].

As the implementation of digital twins can be complicated, [6] has introduced a 5-layer implementation model to simplify the process (Fig. 2.6). The study from [6] also provided technologies that would be required by each layer. Since the fourth and fifth layers are meant for systems needing multiple digital twins, which is not the case in this work, only the first three layers are covered.

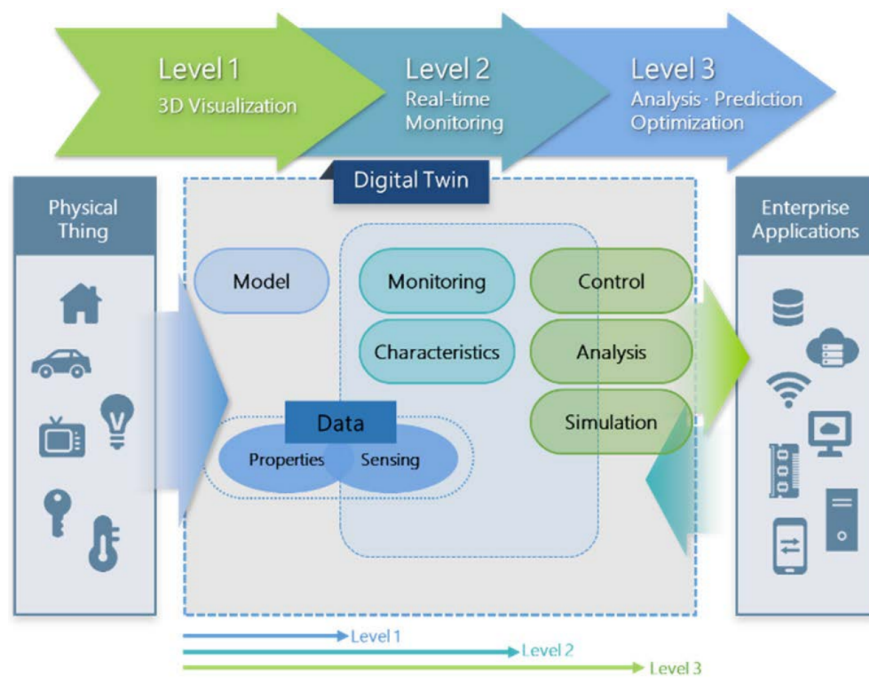


Figure 2.5: Traditional Digital Twin Evolution Model [6]

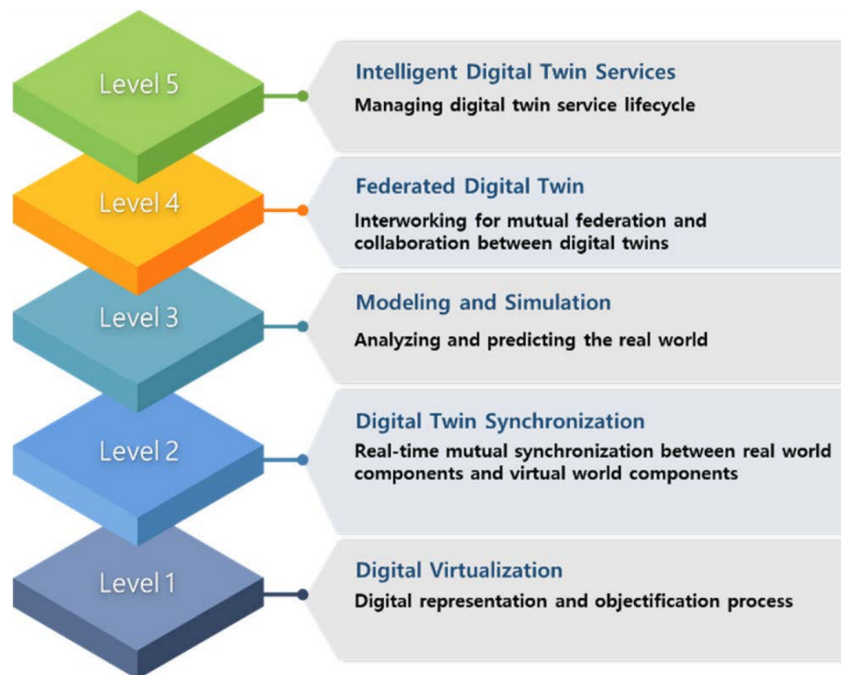


Figure 2.6: 5-layer Implementation Model for a Digital Twin [6]

2.2.5.1 Layer 1 - Digital Virtualization

The implementation of a digital twin starts from the digital virtualization layer. This layer involves collecting information from real-world objects and transferring it to a virtual world. The layer is composed of eight technological elements (Fig. 2.7): real-world data preprocessing, multi-

mensional data causal relation analysis and integration, digital object distributed storage, analysis framework technology, data collection and processing, multidimensional information and object visualization, object identification, sensor placement optimization and virtual sensor technology. The technologies of virtual sensors, sensor placement optimization, and object identification are used for sensing and detecting target objects, while multidimensional information and object visualization are used for visualizing virtual objects. These visualization techniques are able to transform physical objects into virtual objects. Finally, the rest of the technologies can be used for analyzing the corrected data.

Real world data preprocessing technology	Preprocessing technology based on data classification for integrating heterogeneous complex data of different characteristics collected in the real world
Multidimensional data causal relation analysis and integration technology	Database management and correlation analysis technology for integrating preprocessed heterogeneous complex data and transferring information between digital twin components
Digital object distributed storage, processing, and analysis framework technology	Intelligence framework technology for distributed storage, high-speed processing, and analysis efficiency of large data about digital twin objects
Data collection and processing technology	Technology that converts multi-format (e.g., text, table, photo, and video) data acquired in the real world into an unified form
Multidimensional information and object visualization	Technology that derives a relationship between various data and displays various information through multidimensional visualization/multidimensional data visualization/multi-scale transformation
Object identification technology	Technology to classify, measure, and track the objects constituting a physical space using RFID·USN·IoT technology
Sensor placement optimization technology	Technology to reduce redundant information, improve sensor utilization, and maintain stability of collected data
Virtual sensor technology	Technology to calculate physical areas that are difficult to measure with hardware using knowledge and algorithms (SW)

Figure 2.7: Digital Virtualization Technologies [6]

2.2.5.2 Layer 2 - Digital Twin Synchronization

The second layer connects real-world objects with their digital counterparts using seven technology elements (Fig. 2.8). The process involves object cleaning, network management, load reduction, and data verification while the synchronization between the real world and the virtual objects is done through space-time synchronization, information update and real-world actuation technologies.

Space-time synchronization technology	Synchronization technology between the real world and virtual world through the connection between real objects and virtual objects (Physical to Virtual)
Information update technology	Technology that minimizes the error between the real world and the virtual world by updating the changed information of the real world within a predetermined (or permitted) time through minimal resources in the already built virtual world
Sensor-based actuator technology	High-speed and high-precision control technology for an output device (actuator) that reflects optimization information inferred from the virtual world
Object cleaning technology	Technology that minimizes defective elements, such as redundancy, errors, and incomplete and delayed data
Data and information effectiveness verification technology	Technology that detects and corrects errors in information collected in the real world and delivers the event to the management system
High-speed/low-latency data transmission technology	Technology that delivers various heterogeneous complex data collected in the real world to the virtual world to minimize the temporal difference between the real and virtual world and synchronize them
Data transmission management and load reduction technology	Network resource management and load reduction technology to stably deliver information collected in the real world to the virtual world

Figure 2.8: Digital Twin Synchronization Technologies [6]

2.2.5.3 Layer 3 - Modeling and Simulation

The last layer described in this work involves the modeling of digital twin objects and performing simulations to solve real-world problems. This layer considers both tangible and intangible objects. The six technology elements involved (Fig. 2.9) are physics modeling, behavior modeling, system rule technology, automatic scenario generation and tailoring, digital twin simulation and modeling verification, and certification technologies.

Physics modeling technology	Multiple modeling and physics simulation technology including simulations of fluids, structures, air, acoustics, thermodynamics, etc.
Behavior modeling technology	Technology to model and simulate virtual objects according to actual physical phenomena, motions, and behaviors through images, graphs, networks, etc.
System rule technology	Technology that automatically models and simulates rules using large amounts of historical data and knowledge through big data and machine learning algorithms
Automatic scenario generation and tailoring technology	Scenario automatic generation for modeling and simulation and model tuning technology for simulation by mission based on basic model sharing
Digital twin simulation technology	Technology that reduces the probability of error occurrence, uncertainty, cost, etc. through simulation that reflects the real world such as multiphysics, multiscale, and probability
Modeling verification and certification technology	Modeling and simulation verification and certification technology to maintain precision, sensitivity, and robustness

Figure 2.9: Modeling and Simulation Technologies [6]

2.2.6 Implementation

The process of creating a digital twin can be complex, but there are tools available that can make it much easier. These tools have many of the technologies required for each layer of the digital twin implementation, discussed previously, in-built in them. In this report, three of these tools are highlighted: MATLAB, CARLA Simulator, and Unity. Each of these tools has its own unique advantages and disadvantages.

2.2.6.1 MATLAB

MATLAB is widely used by engineers and scientists in various industries including signal processing, control systems, and image processing. One of the key strengths of MATLAB is its extensive library of mathematical functions and toolboxes, which makes it easy to perform complex calculations and data analysis tasks. Additionally, it provides a user-friendly interface and supports multiple programming languages such as C, C++, and Java. Overall, MATLAB is a versatile platform that offers a range of capabilities for engineers and scientists to design and develop solutions that address complex problems. However, even though MATLAB has powerful simulation tools, it still has limitations in 3D visualization (Fig. 2.10), compared to Unity or CARLA simulator.



Figure 2.10: MATLAB's 3D Interface for Vehicle Dynamics Blockset [7]

2.2.6.2 CARLA Simulator

The CARLA simulator is a powerful tool for researchers and developers to test and refine autonomous driving algorithms in a realistic environment (Fig. 2.11). Its multi-architecture server allows for scalability, while its application programming interface (API) supports various in-built sensors in the digital environment, such as LIDARs, cameras, and depth sensors, enhancing autonomous driving research [30]. With comprehensive documentation, users can create custom simulation scenarios and evaluate their algorithms under different conditions. As an open-source platform, CARLA accelerates the development and testing of self-driving cars, improving safety and efficiency in autonomous driving technologies. However, it has some limitations, including a complex interface and a steep learning curve for new users, and it is primarily focused on autonomous driving research rather than other types of simulations.



Figure 2.11: CARLA Simulator Interface [8]

2.2.6.3 Unity

The last tool mentioned in this work is Unity, which is known for its versatility and ease of use, making it a popular choice for game developers of all levels of expertise. It allows for the creation of visually stunning games and simulations, thanks to its advanced 3D rendering capabilities (Fig. 2.12) and support for a variety of platforms, including mobile devices, consoles, and PCs. In addition to its use in the gaming industry, Unity is also utilized in other fields, such as architecture and automotive engineering, where it is used for creating interactive visualizations and simulations. The United States armed forces also use Unity for training purposes, as it provides a realistic and immersive environment for soldiers to practice various scenarios. Although Unity may not have the same level of control system design tools and mathematical 3D modeling capabilities as MATLAB, it is still a powerful tool for game development and simulation. Its intuitive interface and broad range of features make it a top choice for developers looking to create engaging and interactive experiences.

Due to all the reasons listed above, the tool chosen for this project was Unity.

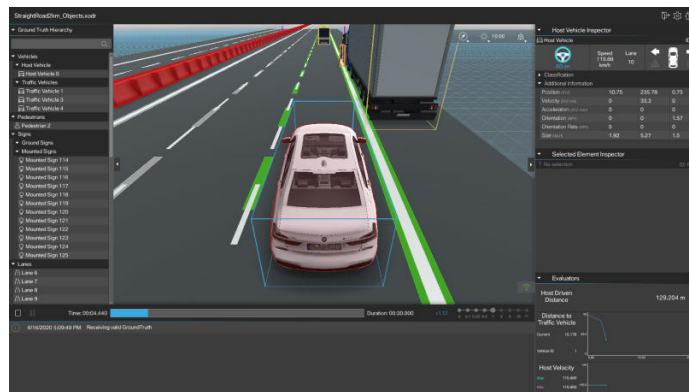


Figure 2.12: Unity 3D Interface Example [9]

Chapter 3

Car and Track System

To ensure a comprehensive understanding of the objective of this work, this chapter provides a description of each component and the car model, which dictates the car's dynamics, in section 3.1, as well as a summary of the contributions made by perception [15] and control [16] in sections 3.2 and 3.3, respectively.

3.1 System Overview

This work has several components, those being a camera, 3 race cars, a central system, and a race track on the scale of 1:43. Fig. 3.1 presents an overview of a similar system; however, it differs from the one used in this work. For example, the system used in this work does not utilize an Ethernet hub and communicates with the cars through WiFi instead.

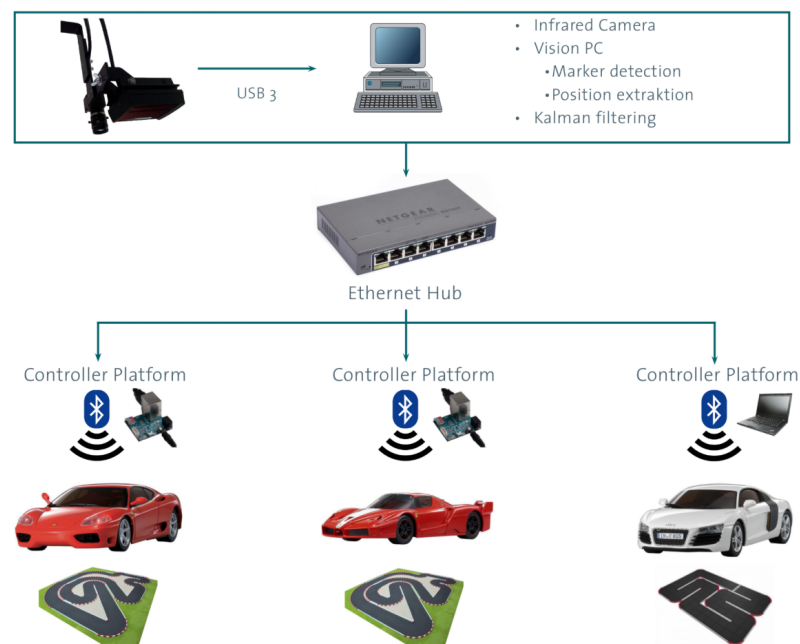


Figure 3.1: System Overview [10]

3.1.1 Race Track

3.1.1.1 Race Track Building

The race track is flat, without any ups or downs. However, to optimize the performance of the cars on the race track, it is crucial that the track is established on a solid and stable surface. Addressing this requirement, a robust surface construction was implemented using four tongue and groove plywood boards.

Four sturdy wooden battens were utilized to further enhance the stability and longevity of the setup. These battens were placed perpendicular to the direction of the plywood boards, acting as reinforcing agents that distribute the load and stresses evenly across the surface. Each batten was fixed into position using 12 screws, ensuring that the entire assembly remains tightly secured against any forces exerted.

After completing the base, the next step was setting up the race track's surface by placing many pieces together, which were designed to interlock with each other. These pieces are made of urethane in order to ensure the traction of the cars to the floor. The pieces were arranged in a 7x8 grid to create a large racing space. Afterward, outer walls around the edge of the track were added to guarantee that the race cars would not accidentally leave the track during a race.

Next, the inner boundaries of the race track were tackled, which outlined the actual path the cars would follow. Instead of building physical barriers, a different solution was opted- using tape. This allowed a challenging route with five sharp curves to be laid out. The idea was to make a track that was not just about speed but also required smart maneuvering and precision, putting the cars' machine-learning systems to the test. After this long process, the race track result is visible in Fig. 3.2.



Figure 3.2: Race Track

3.1.2 Race Car

There are three race cars, each at a 1:43 scale. The colors of these vehicles are distinct: one is yellow, another is purple, and the third is colored blue (Fig. 3.3). Each car is equipped with an ESP32-S2 Mini and a printed circuit board (PCB). The ESP32-S2 Mini is a microprocessor

designed for communication with the central system. It receives instructions and triggers the corresponding actions. These actions are indirectly linked to the PCB, which, in turn, connects to the car's motor and steering control for the wheels. These interconnected components enable the central system to remotely control the race cars. These boards are especially important for the control work, which needs to send to the ESP32-S2 instructions to steer, accelerate, decelerate, or stop the race car.



Figure 3.3: The Three Race Cars

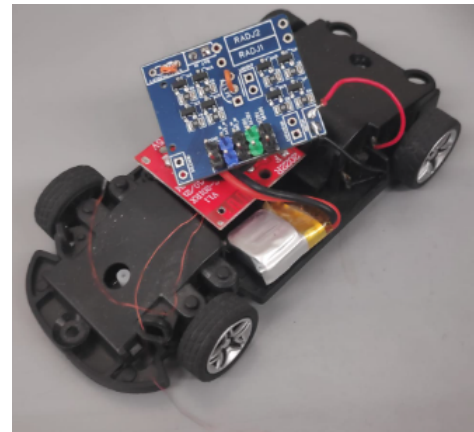


Figure 3.4: Car with PCB and ESP32

Leveraging the machine learning technique known as reinforcement learning, each car is designed to learn from its interactions and continually refine its course of action in real-time. By delving into the details of each car's performance every lap, the algorithm suggests enhancements to various aspects such as trajectory, speed, and braking points. This iterative learning process allows the car to efficiently navigate the track, steadily improving lap times.

3.1.2.1 Car Model

Having a clear understanding of the car model and the dynamics of vehicle steering was essential for this project. The car model was a crucial element as it provided the basis for the work that would be done. On the other hand, the steering dynamics played a vital role in ensuring that the car moved in the desired direction and responded correctly to the driver's input.

The digital twin, control, and perception, were influenced by the car model and steering dynamics. Any inaccuracies in the car model could lead to poor performance. As the digital twin is a virtual representation of the car, its accuracy relies on the accuracy of the car model. Therefore, it was essential to ensure that all project components use the same equations to maintain consistency and accuracy.

Fig. 3.5 represents the overhead view of a race car. The wheelbase is b , and the center of mass is slightly closer to the rear wheels than to the front wheels. By approximating the motion of the front and the rear wheels by a single front wheel and a single rear wheel, an abstraction denoted bicycle model can be obtained, shown in Fig. 3.6.

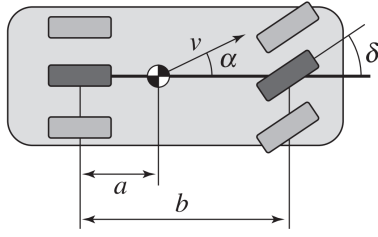


Figure 3.5: Overhead View [11]

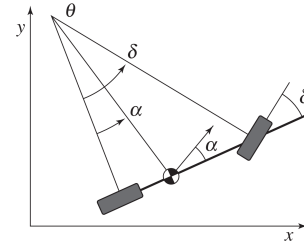


Figure 3.6: Bicycle Model [11]

In Fig. 3.6, it is possible to see that the steering angle is denoted by δ , and the velocity at the center of mass is shown by the angle α relative to the length axis of the vehicle. The vehicle orientation, also denoted heading, is given by θ and the position is defined by the coordinates (x, y) .

Considering the bicycle model, it is governed by the following equations:

$$\begin{cases} \dot{x} = v \cdot \cos(\alpha + \theta) \\ \dot{y} = v \cdot \sin(\alpha + \theta) \\ \dot{\theta} = \frac{v}{b} \cdot \tan \delta \end{cases} \quad (3.1)$$

where

$$\alpha = \tan^{-1} \frac{a \cdot \tan \delta}{b} \quad (3.2)$$

The position of the reference point of the vehicle in the plane is given by (x, y) , and the angle of the vehicle regarding the horizontal is θ . The vehicle inputs are given by (v, δ) , where v is the forward velocity of the vehicle, and δ is the angle of the steering wheel. The wheelbase b and the offset a , which measures the distance between the rear wheels and the reference point, are taken as parameters [11] [31].

- System state: x, y, θ
- System input: v, δ
- System output: x, y
- System parameters: wheelbase, offset.

3.1.3 Camera

The camera was positioned above the race track and was used to capture all the necessary information, including the coordinates and directions of the cars. This process utilized a Kalman filter to mitigate measurement noise or errors in the data, resulting in more accurate tracking [15]. The camera could also be connected to the central system, allowing for real-time updates on the race. Its main function was in perception work, identifying the position of the cars based on their colors and referencing the origin $(0, 0)$ using a white circle placed on one of the corners of the race track.

The information forwarded to the central system would be directly used to monitor the cars in the digital twin. Additionally, this information would also be used by the control work to understand the trajectory of the cars and send the appropriate commands to ensure efficient performance.

The choice of camera was crucial, and the GoPro HERO10 Black (Fig. 3.7) was the top pick. Its image quality and wide field of view capture the race track with clarity. Moreover, this camera supports a direct computer connection, which is important for fast data transfer, being a necessity for the control's and the digital twin's work.



Figure 3.7: GoPro HERO10 Black [12]

3.2 Perception

Perception was the work developed by [15]. It involved acquiring data through the camera regarding the car's position and which direction they were facing and sending that information to the central system.

3.2.1 Calibration and Distortion

The first part of the perception work was to calibrate the camera so that it could accurately capture the race cars and the track. This calibration could be achieved using either the pinhole model or the fisheye model. However, since the GoPro utilized a field of view (FOV) of less than 95° , the pinhole model was the best choice. The pinhole model allows the transformation of the 3D global coordinates into 3D local camera coordinates and finally into a 2D image, transforming world points into camera coordinates.

Subsequently, the distortion, which is due to imperfections in the camera lens, creates a discrepancy between the ideal image and what the camera ultimately captures. Distortion comes in two varieties: radial distortion, which makes straight lines seem curved; and tangential distortion, which causes some sections of the image to look nearer than they are, a consequence of the lens and the image sensor not aligning properly. By employing a matrix designed to retain all the essential elements of the original image and eliminating areas that fall outside the acceptable range, a distortion correction process on the image was implemented.

3.2.2 Race Car Detection

Initially, the race cars were going to be detected using ArUco markers (Fig. 3.8), which are synthetic square fiducial markers. These markers are composed of a black border and an inner binary matrix. They are used in computer vision applications for camera calibration, robot navigation, and augmented reality. However, due to the distance between the camera and the race track, it was not possible to detect the ArUco markers.

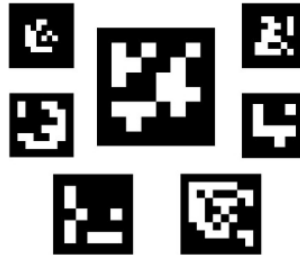


Figure 3.8: ArUco Marker

The next method for car detection involved using the different colors of the cars and applying color segmentation in the HSV color space (Hue, Saturation, Value). This approach enabled the identification and filtering of contours based on their area, superimposing the filtered contours on the original image and calculating the centroid of the largest contour, as is visible in Fig. 3.9.

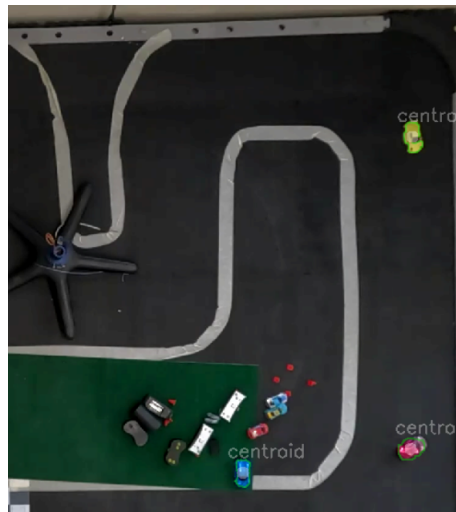


Figure 3.9: Race Car Detection

3.2.3 Estimation of Speed, Position and Orientation

The current state of the dynamic system was estimated using the Extended Kalman Filter. Rather than employing the standard Kalman Filter, the Extended Kalman Filter was favored due to the non-linear nature of the system. Implementing the Extended Kalman Filter involved two steps: the prediction step, which predicts the next state based on the previous state and control inputs,

and the update step, in which the state estimate is updated based on new measurements, and the error covariance matrices are adjusted to reflect uncertainties in the system.

The prediction step used the bicycle model equations to speculate the succeeding state based on the previous state, process noise, and the control inputs, specifically the steering angle and the speed of the car. Afterward, the update step calculated the error between the real measurement and the predicted measurement from the state estimate and modified the state estimate and covariance based on both this error and based on the Kalman gain.

With this filter, it was possible to estimate the car's position, orientation, and speed with great accuracy.

3.3 Control

The control work, implemented by [16], aimed to use the machine learning technique reinforcement learning on the race cars for them to improve their performance on the race track over time, finishing each lap in the least time possible. This work was divided into two parts: the first part involved testing in a Pygame environment, while the second part used a Unity simulation environment.

3.3.1 Pygame Simulation Environment

In the early stages of the development of the control work, a PID controller (Proportional-Integral-Derivative) was created to control the steering of a car, in an effort to keep it in the middle of the race track, which was created using the pygame's library. There were two strategies, one would be in an unknown environment, while the second would be in a known environment.

In the unknown environment, in order to estimate the vehicle's position, two lines were projected, one at -70° and the other at 70° regarding the car's forward position, to mimic sensors. This approach allowed for the trajectory to be drawn on the track and therefore made it known, as visible in Fig. 3.10.



Figure 3.10: Pygame's Car Sensors and Trajectory

In the second strategy, the environment was considered known. Cyclically, both the point that was in front of the car and the point behind the car would be selected, and then they would be united

by a line. With this line, it was possible to determine the middle of the car and then determine the cross-track error, which is the distance between the car and the trajectory (Fig. 3.11). This error would then be used to adjust the steering, controlling the car in driving into the trajectory.

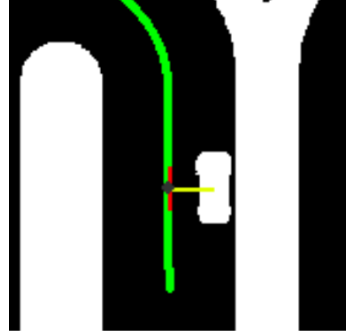


Figure 3.11: Pygame's Car with Cross-Track error

After the implementation of the PID controller, it was replaced using imitation learning. This involved a neural network, which consisted of an input layer of size 12 and included three hidden layers. Following its training, the neural network was found capable of effectively mimicking the functions of the PID controller, leading to the successful substitution of the PID controller. The decision to replace the PID controller with the neural network was motivated by its ability to learn from data, adapt to more complex scenarios, and improve performance over time.

3.3.2 Unity Simulation Environment

During this phase, [16] utilized the final models of the race track and the race cars, detailed in subsections 4.1.2 and 4.2.2, respectively, and the initial version of the movement script, presented in subsection 4.3.2, both of which were developed for the digital twin presented in this dissertation.

A PID controller was created to control the steering and the acceleration and deceleration of the race car. Since the environment is considered known, various checkpoints were placed across the race track (Fig. 3.12). These checkpoints permitted the race track to be divided into several segments, which allowed a trajectory to be defined (Fig. 3.13).

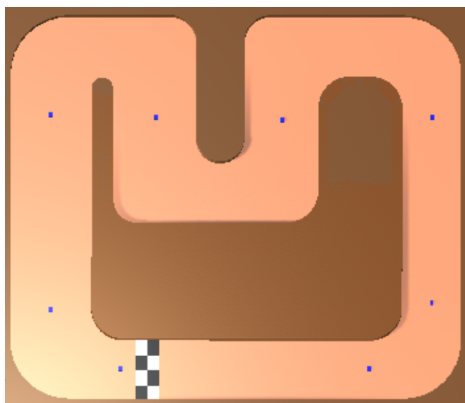


Figure 3.12: Race Track's Checkpoints

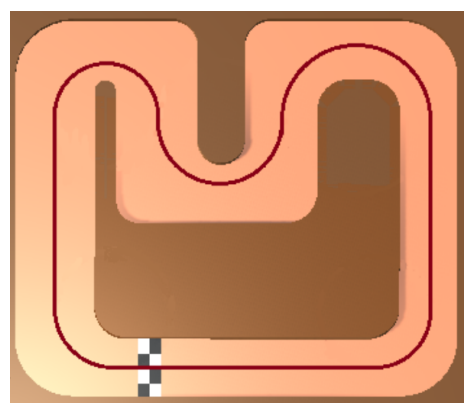


Figure 3.13: Race Track Trajectory

Given the trajectory, it was possible to calculate the cross-track error by calculating the difference between the car's position and the nearest point of the trajectory perpendicular to the car. The cross-track error was then used as an error to the PID controller as an input to control the steering angle of the car, to minimize the error by having the car drive towards the trajectory. After these implementations, the car was able to follow the trajectory.

The reinforcement learning implemented in Unity utilized the Proximal Policy Optimization (PPO) algorithm. To implement this algorithm, several parameters were used, like the progress made on the race track, the cross-track error, the speed in the x-axis and the y-axis, and the angular velocity, among others. These parameters would constitute the system's state, which determines the action the agent must take in order to maximize its reward.

In each training episode, the car spawns at a randomly selected position near a checkpoint, featuring a random orientation and velocity, all variables being within a predetermined range. Designed to enhance the learning speed, every episode extends across 360 ticks, compelling the car to navigate between one and two laps, thereby optimizing its adaptability to a variety of scenarios. If the episodes were longer, the car would lose some ability to comprehend and adapt to its operational dynamics.

The car would gain the most rewards if it could travel the largest distance without hitting the walls of the race track. To ensure the car behaved correctly, it would receive a penalty if it touched a wall or drove backward.

Reflecting on the final results of the training, several things became evident: The agent's cumulative rewards and the duration of each episode progressively increased; Initially, the penalties were substantial but swiftly diminished, despite occasional spikes from the agent experimenting with new strategies; Finally, early on, the agent's actions were somewhat random, yet it quickly adapted, and over time, its behavior started to stabilize, consistently following the same strategies which were the ones that would maximize the rewards.

Chapter 4

Digital Twin's Object Modeling

The first stage in the digital twin's development is modeling. Modeling serves as the basis for creating a digital twin since the accuracy of later steps relies on these models. While it's essential that these models reflect their real-world counterparts, modeling every detail is not always necessary. Careful selection of which components to include is vital to avoid unnecessary complexity while preserving the model's overall effectiveness.

This chapter aims to provide a detailed explanation of the methodology used to model the race car and track in the virtual environment, therefore being divided into three sections. The first two sections detail the development of the physical models for the race track and the race cars, including both their initial and final versions: section 4.1 focuses on modeling the race track, while section 4.2 outlines the race car modeling. Section 4.3 explores the methods used to derive the car's parameters and the development of C# algorithms to integrate these parameters into the digital cars in Unity. This includes adding physics components and scripting movement behavior.

4.1 Physical Modeling - Race Track Models

To create the physical models, the program *Blender* [32] was used. Blender is an open-source tool that serves a multitude of purposes, especially in rendering 3D objects. Beyond its primary function, Blender is extensively used across various fields, including animation, engineering, and architecture, showcasing its versatility and capability to manage a wide range of professional tasks.

4.1.1 Simplified Race Track Model

The simplified race track involved four steps in its creation, going from the surface plane to the outer walls, the round corners, and the inner walls. Several measurements were taken from the real race track using a measuring tape in order to ensure the correct modeling of the digital race track.

The first step was to build the race track's surface, which is composed of several square pieces that interlock each other, with each piece measuring 29.8cm on its side. This dimension was accurately replicated in Blender by crafting a plane with identical measurements. This plane was

successively duplicated and adjoined to the adjacent segment, continuing this process until a total of seven planes were arrayed side by side, as can be seen in Fig. 4.1. Subsequently, these seven pieces were amalgamated into a larger single piece using Blender's *Join* tool, which merges the selected objects into one. To accurately replicate the race track's 7x8 grid size, previously defined in the subsection 3.1.1.1, this newly formed larger piece was then duplicated seven times, ensuring a precise representation of the real race track surface. The result of the first step is visible in Fig. 4.2.

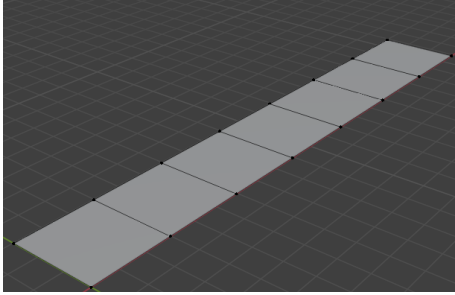


Figure 4.1: The Seven Pieces Merged

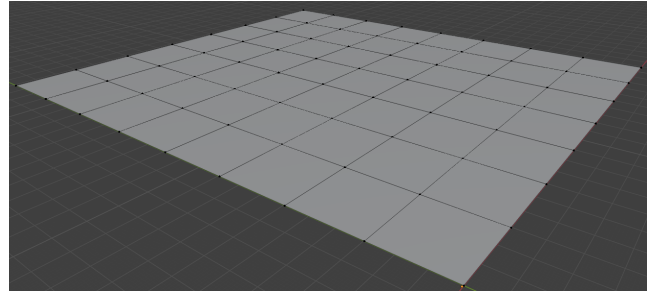


Figure 4.2: Race Track's Surface (first step)

With the surface in place, the second step was to form the outer walls. These walls are composed of cube segments, each with an edge length of 4.8 cm, mirroring the width of the walls found on the real race track. A cube was positioned at each corner of the race track's surface, followed by the utilization of one of Blender's features, the object modifier. This tool, among other capabilities, enables the setting of an array for the object in question. This array replicates the initial cube, placing it adjacent to its predecessor, thereby constructing a continuous wall along one side of the surface. This method was systematically repeated for the remaining three sides until the final configuration, wherein the walls were merged using the *Join* tool, was achieved, as shown in Fig. 4.3.

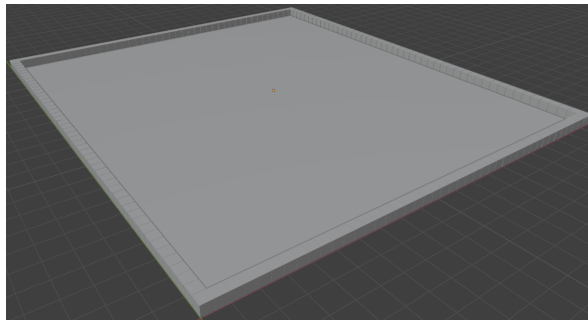


Figure 4.3: Race Track with Outer Walls (second step)

For the third step, outer turns were modeled by using a cylinder and a cube. The cube was placed on the edge of the track surface, while the cylinder was positioned slightly closer to the center of the track. The cylinder was then cut so that only a part of its surface remained, specifically the segment that would mimic the corner curves. Most of the cube's vertices were deleted, leaving only the two vertices that formed the vertical line which stood closer to the race track's edge. The cylinder and the two vertices were merged, using the *Join* tool. Then, using Blender's *Fill*

tool, which creates a surface by connecting selected vertices with faces, the remaining faces were constructed. This was achieved by selecting all the vertices of the cylinder's surface and the two remaining vertices of the cube, which resulted in the formation of a new object. This new object was then copied and pasted on the other three corners of the surface by rotating and translating the original object, visible in Fig. 4.4.

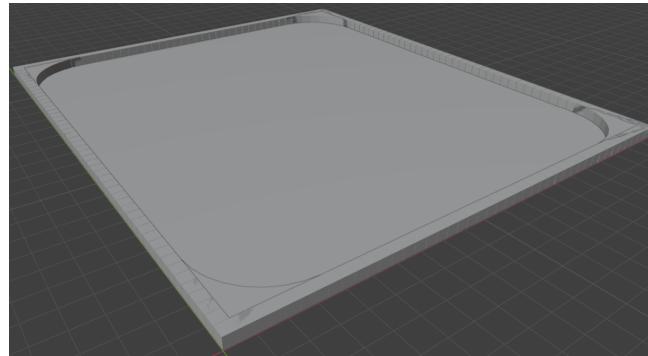


Figure 4.4: Race Track Outer Curves (third step)

At this stage of the project, there was no specific design for the interior of the race track. As a result, a straightforward central layout was created to enable the testing of the dynamics of the digital race cars.

For the fourth and final step of the simplified race track model, the inner curves were created using four semicircles and four walls. Four cylinders were placed at each of the four corners of the race track, at the same position and with the same diameter as the ones used on the outer curves. However, these cylinders were positioned 29.8cm, the size of a surface segment, closer to the middle of the race track's surface. Each cylinder was then sliced, and walls were overlapped onto the cylinders to form a continuous structure. These walls were constructed using the same method used for the race track's walls: an array of cubes with an edge length of 4.8cm, forming a total of four walls. The sliced cylinders and the walls were merged into a single object using the *Join* tool.

Once all the objects were in place, they were connected to each other. In Blender, this relationship is established through the concept of *Parenting*. By designating an object as the *parent*, any movements or transformations applied to it are also reflected in its *children*. This hierarchical relationship ensures a coordinated motion where the parent's adjustments inherently influence the children. However, it is important to note that this influence is unidirectional; the movements of the children have no impact on the parent. By assigning the race track surface as the parent of all the other objects, whenever it is required to move the whole race track from one position to another, the only object that needs to be moved is the surface. Without parenting, each object would need to be moved individually. The final result of the simplified race track is visible in Fig. 4.5.

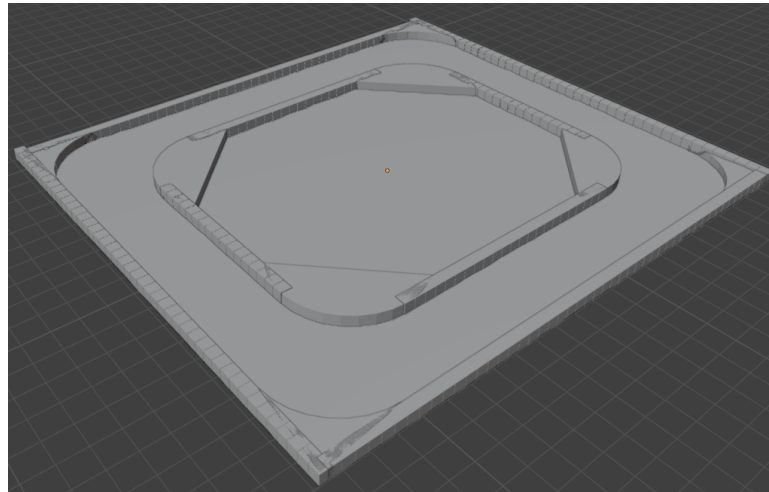


Figure 4.5: Simplified Race Track (fourth step)

4.1.2 Final Race Track Model

Once the inner boundaries of the real race track were defined, the final model of the race track could be designed. Several measurements were taken with a measuring tape, and two new objects were created: one to replicate the U-turn wall and the other for the inner curves present on the race track. Therefore, two new steps were introduced to the ones used in the simplified design. Since the inner curves of the older design were no longer relevant, they were removed, and thus, the fourth step was replaced by the U-turn wall.

Before modeling these objects, the race track was simplified to achieve a cleaner design. The *Fill* tool used earlier created faces that connected all the vertices, including those inside the objects. In Fig. 4.6, two of the several created faces are highlighted for easier identification.

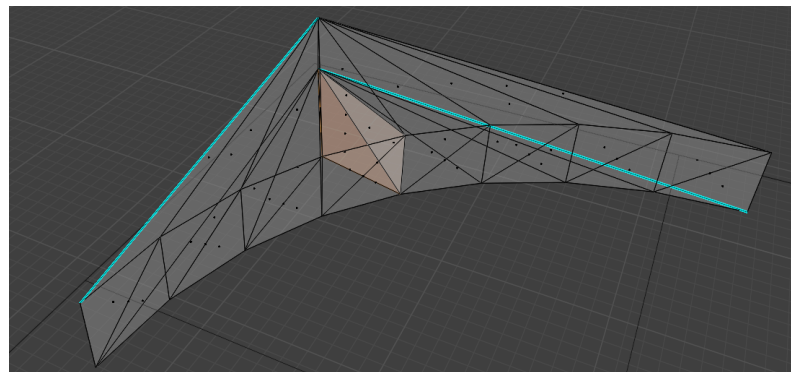


Figure 4.6: Inner Faces Created Inside an Outer Curve

This situation could be avoided by using a *simpler fill*, which creates a single face uniting the selected vertices instead of multiple faces. The outer curves of the racetrack were remade following the third step procedure described in subsection 4.1.1. However, instead of using the previous *Fill* tool, the faces of the outer curve object were created one at a time. By utilizing the *simpler fill*, two vertices from one edge of the remaining cylinder's surface were selected along

with the other vertices of the cube to create a face, as seen in Fig. 4.7. The face on the other side was created using the same method but by selecting the vertices on the opposite side of the cylinder's surface. For the top face of the object, all the top vertices were selected and merged using the *simpler fill* technique. Fig. 4.8 illustrates the final result of the outer curve, which was again replicated to the other four corners of the racetrack. Additionally, it is visible in Fig. 4.9 that there are no internal faces created within the object, unlike the initial design.

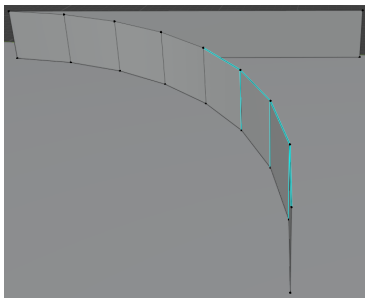


Figure 4.7: Outer Curve With One Side Wall

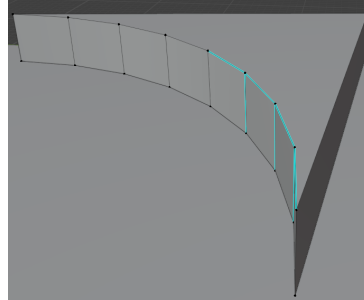


Figure 4.8: Finished Outer Curve

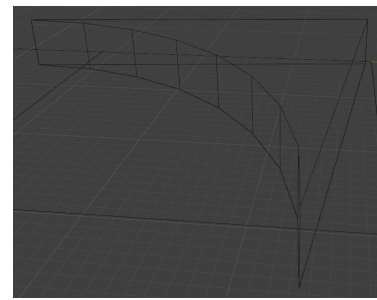


Figure 4.9: Outer Curve Without Inner Faces

Utilizing this same technique, the race track's walls were also modified. The cube's array was deleted, and instead, the walls were created directly, one face at a time. Even though these implementations did not have any effect on the functioning of the race track, it allowed for the overall design of it to be much cleaner. After the improvement on the design was finished, the U-turn wall and the new inner curves were modeled.

The new fourth step, the U-turn wall, was made using two cylinders for its first and final curves and a third one for the U-turn's edge. The first two cylinders were partially cut so that only a part of their surface remained, and the cylinder at the U-turn's edge was cut in half (Fig. 4.10). Faces were created between the U-turn cylinder and each of the two curves to form the walls. After merging all the objects using *Join*, the top face was created using the same method used on the outer curve. The result can be seen in Fig. 4.11.



Figure 4.10: U-turn's Cut Cylinders

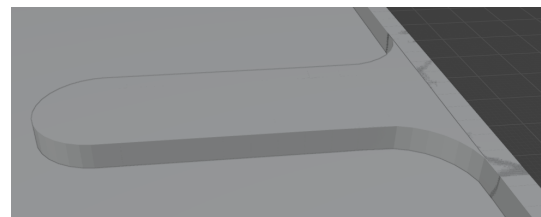


Figure 4.11: Race Track U-turn Curve (new fourth step)

Lastly, the fifth step was to model the most complex objects on the race track: the inner walls and curves, which were constructed using the methods previously applied. Specifically, seven cylinders were strategically placed on the race track, then cut, merged using the *Join* tool, and united with each other by using the *simpler fill*, which was also used to make the top face of the object. Fig. 4.12 represents the race track with the inner walls and curves finished.

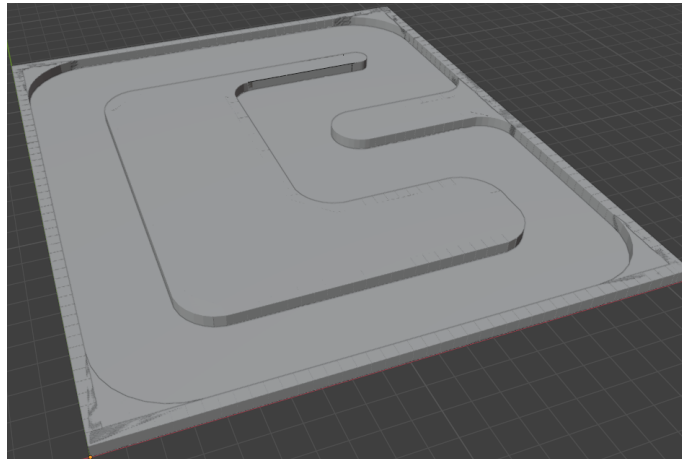


Figure 4.12: Race Track with all the Walls and Curves Finished (fifth step)

To make the digital environment clearer and distinguish the walls from the car, colors were added to the surfaces, walls, and curves. This was achieved by creating materials with specific colors in Blender and then assigning these materials to the objects. The surface was assigned a beige color, while the walls were assigned a brown color. The finish line was also added in the same position as the real-life counterpart by creating a plane, dividing it into several smaller parts using a modifier, and then assigning a black material to half of the parts and a white material to the other half, giving it a checked pattern. All the objects without parents were assigned the race track's surface as their parent.

The final version of the physical model of the race track is visible in Fig. 4.13.

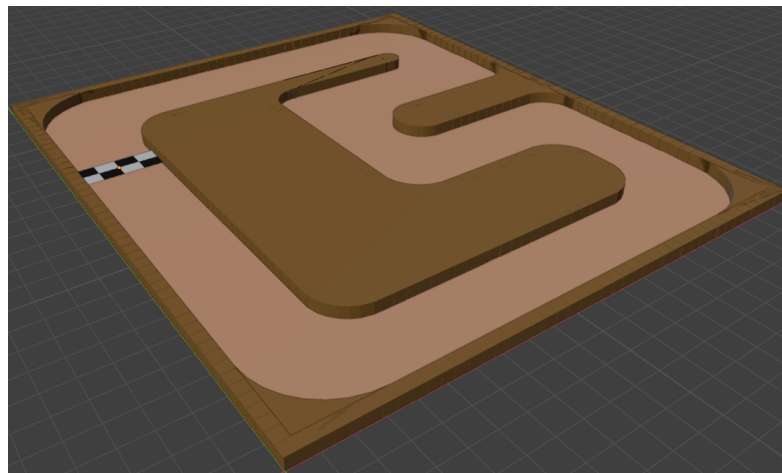


Figure 4.13: Final Race Track Model

4.2 Physical Modeling - Race Car Models

4.2.1 Simplified Race Car Model

The race cars were the most complex objects to replicate in this work. However, since the race track is flat, the car modeling was not focused on height but mostly on the possibility of colliding with the race track's walls.

At this stage, the 1:43 scale cars were not yet available, so no measurements could be obtained. To enable testing of movement algorithms, a smaller race car model was created based on an available design. The parameters of this model were taken to ensure that it would effectively support the development and testing of car movement and collision. The measurements encompass the diameter and width of the wheels, the distance between the front or rear wheels (referred to as the front and rear wheel track), the wheelbase, the distance from the front or rear wheels to the front or rear of the car (known as the front and rear overhang), and the location of the car's center of mass.

Table 4.1: Smaller Car Specifications

Parameter	Value
Diameter of the front wheels	1.2 cm
Diameter of the rear wheels	1.4 cm
Front wheels width	0.4 cm
Rear wheels width	0.4 cm
Front wheel track	2.7 cm
Rear wheel track	2.7 cm
Wheelbase	4.4 cm
Front overhang	2.6 cm
Rear overhang	2.3 cm
Car's center of mass	Center of the car's axle

Once all the necessary measurements were defined, based on the available design, the next step was to model the car in Blender. This modeling process was divided into three steps and involved using a series of cylinders of different sizes to accurately represent the wheels, the axle, and the car's body. For the first step, a cylinder was created in order to mimic the axle (Fig. 4.14). This axle served as the model's base, to which all the other parts would be linked. The car's center of mass is located exactly in the middle, just like the center of mass of the axle.

In the second step, the front wheels were modeled using three cylinders: one cylinder for each wheel and then a cylinder connecting the two of them. These three objects were merged using Blender's *Join* tool. Once the new object was created, it was placed at the front end of the axle cylinder. Then, the rear wheels were modeled using the same method, differing only on the size of the wheels, both in diameter and width, according to the measures taken previously (Fig. 4.15).

Regarding the third step, in order to test the collisions of the car with the walls of the race track, four cylinders were placed according to the front and rear overhang, forming a structure around the car, which was then merged using the *Join* tool. Since the race track's walls were high,

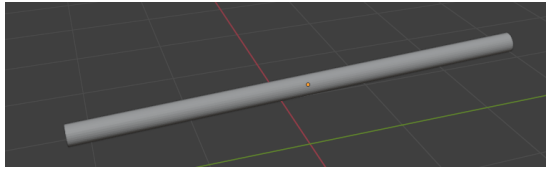


Figure 4.14: Smaller Race car Model's axle (first step)

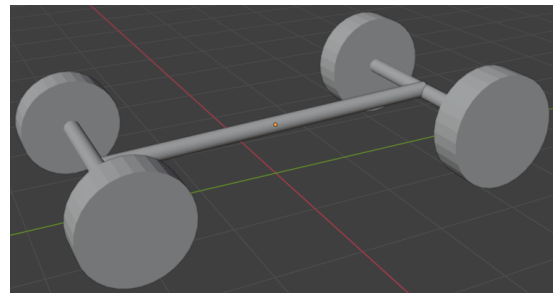


Figure 4.15: Smaller Race car with Both Rear and Front Wheels (second step)

this structure could be simplified by placing the four cylinders above the car. The axle object was also assigned as the parent of all the other objects.

While this design was primarily focused on aiding the development and testing of movement algorithms, it did not emphasize aesthetics since it was intended to be a temporary solution. This approach allowed for a thorough assessment of the race track and the race cars' overall functionality. The final design of the simplified race car can be seen in Fig. 4.16.

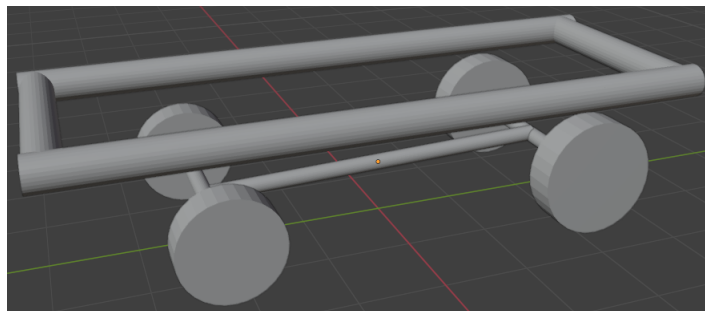


Figure 4.16: Smaller Race Car (third step)

4.2.2 Final Race Car Model

Once the 1:43 race cars became available, several measurements were taken, as listed in Table 4.2.

With the collected measurements, the modeling process started over but used the same methods applied in the simplified race car model: a sequence of cylinders of varying sizes was used to replicate the wheels and the side, front, and rear structure of the race car. Due to the increased complexity of the car's side, front, and rear structures, the entire modeling process was divided into four steps instead of three. The third step focuses on the side structure, while the fourth step addresses the front and rear structures. The first and second steps were recreated for the car's axle, and front and rear wheels, as can be seen in Fig. 4.17.

The final part of the race car physical modeling was adding the car's body which would collide with the walls. In the third step, two cylinders were incorporated—one on each side of the car. Each cylinder had a diameter of 0.6 cm, corresponding to the width of the rear wheels. These cylinders were carefully positioned so that they were close to the rear wheels but had some space

Table 4.2: 1:43 Car Specifications

Parameter	Value
Diameter of the front wheels	1.6 cm
Diameter of the rear wheels	1.8 cm
Front wheels width	0.55 cm
Rear wheels width	0.6 cm
Front wheel track	4.23 cm
Rear wheel track	4.28 cm
Wheelbase	6 cm
Front overhang	2.6 cm
Rear overhang	2.3 cm
Car weight	48 g
Car's center of mass	Center of the car's axle

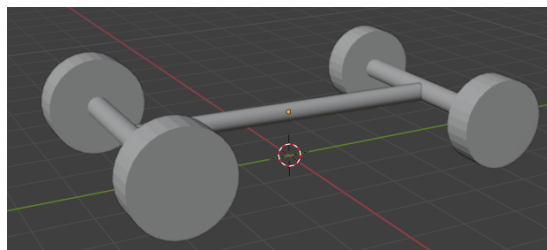


Figure 4.17: Race Car with Both Rear and Front Wheels (first and second step)

apart from the front wheels to allow their steering to be modeled later, preventing the collision of the wheels with the side structure (Fig. 4.18).

Subsequently, for the fourth step, the front and rear structures of the car's body were designed by using one cylinder for the front and one for the rear. The cylinders were resized to match the distance between the outer edges of the wheels, and their ends were positioned to match the front and rear wheel track accordingly. The car's side, front, and rear structures were merged using the *Join* tool, forming the car's body as a single object. The final result is shown in Fig. 4.19.

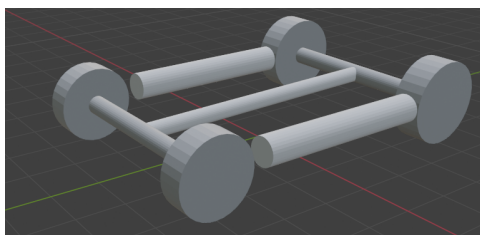


Figure 4.18: Race Car with the Side Structure (third step)

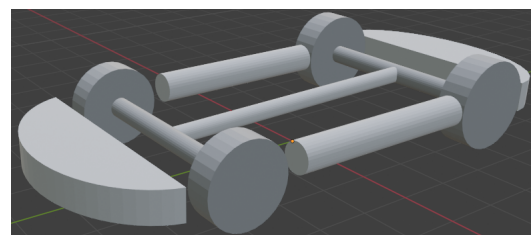


Figure 4.19: Finished Physical Car Model (fourth step)

Given that the components were assembled, their interconnections were crucial. To ensure a unified movement within the setup, all objects were parented to the central axle cylinder to which the movement algorithms would be assigned.

4.3 Modeling in Unity

With the completion of the physical models for both the race track and the race car, the following stage involved developing the algorithms and modifying the objects in Unity so that they behaved as intended. In Unity, the algorithms are written in scripts utilizing the programming language C#. This is a high-level and versatile language that offers strong performance, and a rich set of features, making it ideal for game development.

4.3.1 Rigid Body and Colliders

Before implementing the movement algorithms, it was essential to assign the race cars with a rigid body in Unity. A rigid body allows objects to adhere to physics principles, enabling them to receive forces and torque for realistic movement. Consequently, upon being assigned a rigid body, the race cars became subject to forces being applied to them, as forces applied through scripts. In Unity, after selecting the race car's axle, which served as the parent for the other car parts, the rigid body component could be added. By assigning the rigid body to the parent, the children behave as rigid bodies too.

On the other hand, it was crucial to assign a collider to the car's components. In Unity, a collider is a component that defines the shape of an object for physical interactions, enabling it to detect collisions and respond to them. Before the car and the race track were assigned colliders, they would not interact with each other, but after they were implemented, the race track functioned as an obstacle the race cars could not go through. Therefore, it was imperative to add a collider to ensure that the car could collide with the walls of the race track and other obstacles, preventing it from getting off of the race track's limits. Unity provides a variety of colliders and each part of the car needed to be assigned an appropriate collider. Box colliders were added to the front and rear wheels to match their box-like collision shapes. For the car's body, a mesh collider was used to accurately represent its complex shape with accuracy, which a box collider would not suffice. The box colliders of the front and rear wheels can be seen in Fig. 4.20 while the mesh collider of the car's body can be observed in Fig. 4.21, both represented by the light green lines.

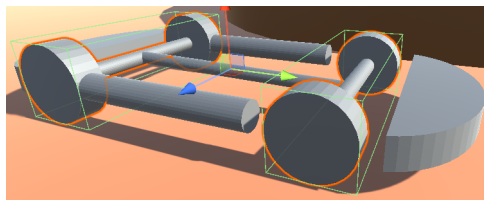


Figure 4.20: Car Wheels' Box Colliders

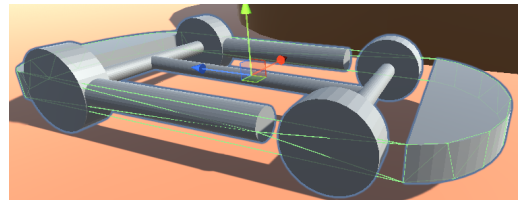


Figure 4.21: Car Body's Mesh Collider

The race track needed to be set up in a similar way but without including the rigid body component, as the race track is a stationary object. Adding a rigid body would subject it to physics interactions, such as collisions, which is not intended. A box collider was applied to the surface of the race track, while mesh colliders were used for the other race track's components due to the numerous curves.

4.3.2 Script Implementation - Initial Version

The final step in modeling the race cars involved creating the necessary movement algorithms in Unity and assigning them to the car objects through scripts. The initial version of the codes aimed to control the cars by sending commands using the arrow keys, controlling its movement and steering, continuously.

The primary algorithm governing the car's movement was based on the car model equations referenced in the subsection 3.1.2.1. Its purpose was to test the car's movement on the race track. However, certain parameters of the car, such as acceleration, maximum steering angle, and maximum speed, were unknown. To find the values of these parameters, several tests were conducted.

After repeatedly accelerating and stopping the car, it was observed that both acceleration and braking due to friction were almost immediate. To ascertain the maximum steering angle and maximum speed, two separate tests were conducted on one of the real cars. For both tests, the acceleration and braking times were not considered, given that they were almost immediate. In the first test, the car's steering was set to maximum, and it was accelerated through 5 full circles, with the time taken to complete those circles being measured. In the second test, the car, with its steering still at maximum, was accelerated until it reached the opposite side of the circle, and the distance across the circle's diameter was documented. Both tests were performed 5 times each to reduce errors, with the results displayed in Table 4.3.

Table 4.3: Real Race Car Circle Test Data

Iteration	1st	2nd	3rd	4th	5th	Average
Diameter d (cm)	49	51.5	50.6	49.5	50.55	50.23
Time per 5 circles (s)	8.68	8.48	8.74	8.66	8.76	8.664

With the average values of the diameter, d , and the speed, the maximum speed was calculated:

$$dist = 2\pi r = \pi d \quad (4.1)$$

$$v = \frac{5dist}{t} = \frac{5\pi d}{t} = \frac{5\pi \times 50.23}{8.664} = 91.068 \text{ cm s}^{-1} \quad (4.2)$$

Where $dist$ is the circumference length, d is the diameter, v is the maximum speed, 5 is the number of laps, and t is the average time taken to finish the laps.

Through the bicycle model, the radius of the circumference, and the wheelbase length b , it was also possible to calculate the maximum steering angle of the wheels using the following equation:

$$\tan(\delta_{\max}) = \frac{b}{r} \Leftrightarrow \delta_{\max} = \tan^{-1} \left(\frac{6}{50.23/2} \right) = 13.436^\circ \quad (4.3)$$

With $\delta_{\max}$ being the maximum steering, $wheelbase$ the one determined in Table 4.2, and r the radius of the circumference.

After calculating the previous values, the first C# algorithm was implemented in Unity. This algorithm was designed to simulate the movement of a car, focusing on physical realism and

responsiveness to user input. The Rigidbody component was utilized to manage physics-based motion. Key parameters, such as the distances between the car's axles, the maximum steering angle, acceleration, and friction coefficients, were defined to influence the car's behavior. In Unity's scripts, there are two main functions: *Start()* which is called once when the script is initialized and is typically used for setup tasks like initializing variables, setting object states, or preparing components. In the case of this algorithm, this function allowed to adjust the position of the car's center of gravity using the Rigidbody's *centerOfMass* property to enhance stability.

The other main function is *FixedUpdate()*, which runs a part of the script at fixed intervals, commonly used for physics-related updates, such as applying forces or handling Rigidbody interactions. Regarding this algorithm, it would receive user inputs from the Horizontal and Vertical axes (left, right, up, and down arrow keys) in order to control the steering and acceleration of the race car, respectively. The steering input was used to determine the angle of the front wheels (δ), which was clamped to the car's maximum steering, 13.436° . The car's forward velocity (v) was derived from the acceleration input. Using the steering angle, the angular displacement of the front wheels (α) was calculated, and the car's velocity in both the x and y directions ($\dot{x}$, $\dot{y}$) was updated, factoring in the car's orientation and steering.

Motion was applied using the function *AddForce* that applies an explicit force (which is a force applied on the rigid body through scripts) instead of direct position updates to ensure physics-based interactions. For example, using the direct position updates, the car would phase through the race track and leave it, which does not happen when utilizing *AddForce*, making this last one the best approach found. The forward movement of the cars was calculated by projecting the velocity onto the car's forward direction, and a proportional explicit force was applied via the Rigidbody.

When a car is in motion, but no explicit force is being applied to it, its speed gradually decreases due to implicit forces, like friction. Implicit forces differ from explicit forces in that they arise from Unity's physics system instead of scripts. Examples of implicit forces include gravity, collision impulses, and drag. While these implicit forces caused the car to decelerate, they did so at a slow pace. Two potential solutions were identified for this issue: adjusting the drag parameter of the rigid body, or simulating the drag through an algorithm. Since modifying the drag value on the rigid body would also impact the acceleration and maximum speed of the car, it was preferable to alter the car's script algorithm instead. This modification involved applying an explicit force, called deceleration force, in the opposite direction of the movement whenever there was no forward or backward input, using *AddForce*, to create a natural stopping behavior.

Without applying this deceleration force, in the scenario where the car crosses the finish line at maximum speed and no further explicit forces act on it afterward, it would continue to move forward. After two seconds, it would still have a significant amount of speed and would end up in the position shown as the purple car in Fig. 4.22. However, under the same scenario, if a deceleration force is applied, the car would slow down and stop shortly after, similar to what occurs to the real cars, represented by the blue car in the same figure.

The acceleration and deceleration values were found via tuning: the digital car and the real car

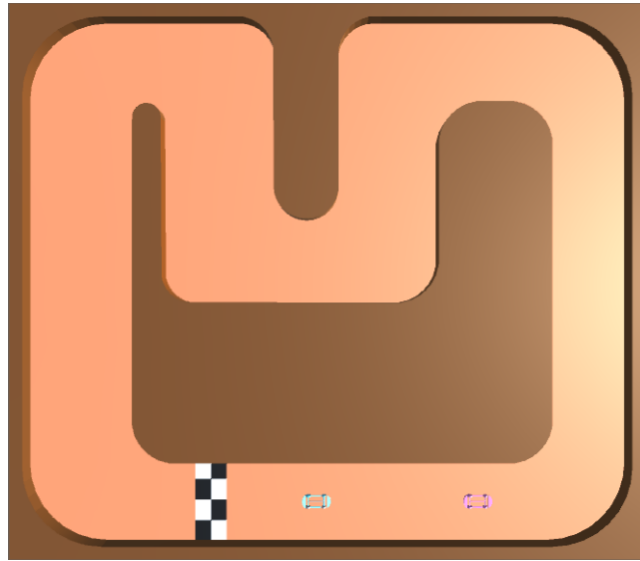


Figure 4.22: Car with (Blue) and without (Purple) the Deceleration Force

were tested simultaneously and different values were applied to the digital car's acceleration and deceleration forces until the values that depicted the most identical behavior were achieved.

Another modification was added to imitate the car sliding sideways. The digital cars would slide more than their real counterparts, thus a sideways explicit force was added to limit sideways drift, reducing the lateral velocity component relative to the car's orientation.

To enhance realism, the angular velocity ($\dot{\theta}$) was computed to adjust the car's rotation dynamically, mimicking how a car turned based on its speed and steering angle. The car's maximum speed was capped to prevent unrealistic motion, and its rotation was updated using Unity's Quaternion function. According to [33], a Quaternion is a mathematical concept used to represent 3D rotations. It avoids problems like gimbal lock, which can occur with other rotation methods, such as Euler angles. This allows for smoother transitions between different orientations, making it a preferred choice in 3D graphics and simulations. In this work, the Quaternion function was used to concatenate a series of rotations into a single representation. This combination of inputs, physics-based forces, and constraints provided a comprehensive simulation of car dynamics suitable for interactive environments or game development scenarios.

In order to better perceive the car's movement, another algorithm was implemented in Unity to rotate the front wheels based on the value of the steering. This procedure followed the same method as the car's rotation and limited the wheels' rotation based on the maximum steering angle, 13.436° . Fig. 4.23 shows the car with the default steering, and Fig. 4.24 shows the car steering to the left.



Figure 4.23: Car With Default (Centered) Steering

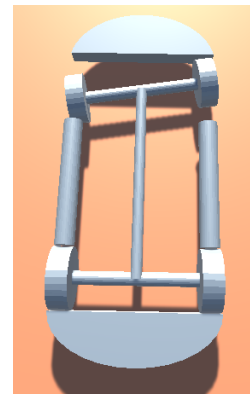


Figure 4.24: Car Steering to the Left

4.3.3 Script Implementation - Unity Refined Code

The real cars are equipped with the ESP32-S2 Mini, which runs code to receive car commands via UDP - User Datagram Protocol, a connectionless communication protocol. This property makes the protocol fast and efficient for transmitting the car commands, at the cost of not establishing a reliable connection [34].

To control the cars, an algorithm was developed in Unity to send commands via UDP. This algorithm assigned specific keys for sending these commands: the "left" and "right" arrow keys were used to send the command *ml* and *mr*, which change the steering of the car to the left, or to the right, respectively. On the other hand, the "up" and "down" arrow keys were used to send the movement commands *mf* and *mb*, to move the car forward or backward. The keys "+" and "-" were used to increase or decrease the duration of the movement commands *mf* and *mb* by 10ms by sending the *+d* or *-d* commands. However, the delay could not go under 10ms at any time.

The "F" key would send the *flash* command, which would behave differently depending if the target was the real car or the digital one. If the real car received the *flash* command, it would flash an LED on the ESP32, but if it was received by the digital car, it would send a message to the terminal. These behaviors were used to see if the cars were alive and responding. Lastly, the "R" key was used to execute the *rr* command, which resets the steering back to 0°.

In the algorithm, to prevent the steering (*ml*, *mr*) or the movement time increase and decrease (*+d*, *-d*) commands from being sent multiple times by accident, the key presses were configured to register only on the rising edge. This means that when a key is pressed, the command is sent only once, regardless of how long the key is held down. However, the movement commands (*mf*, *mb*) function normally, allowing for the car to move smoothly. If the movement commands also functioned with rising edges, the car would be constantly between a breaking and an accelerating state, having an inconsistent motion.

The Table 4.4 presents the commands exchanged between Unity and the cars. The first column lists the commands, whose names were already defined in the code on the ESP32-S2, which was

available in the laboratory. The second column indicates the keys used to send each command in Unity, and the third column describes the function of each command.

Table 4.4: Car Commands

Command String	Keyboard Key	Function
mf	up arrow	moves the car forward
mb	down arrow	moves the car backward
+d	+	increases the delay
-d	-	decreases the delay
ml	left arrow	steers the car to the left
mr	right arrow	steers the car to the right
flash	F	checks if the car is still responding
rr	R	resets the steering to 0°

When a movement command is issued (either *mf* or *mb*), the car does not respond for a duration of d units of time. This delay happens because neither a finite state machine (FSM) nor threads were used; instead, a simpler method of listening, receiving and executing a command was implemented. This duration can be adjusted by sending the *+d* or *-d* commands. This feature helps the car move more steadily; with a shorter movement time, the car tends to accelerate and decelerate quickly. Conversely, a longer movement time allows for more stable motion. However, longer movement times require careful navigation to avoid obstacles, such as walls. In Fig. 4.25 an example is visible where a movement command was sent at time t_1 , in which *cmd* activated, which meant that ESP32-S2 would not receive more commands while it was active. At time t_2 , where $t_2 - t_1$ is the duration d , *cmd* returns to 0 as the movement command has expired, and it is ready to receive a new command.

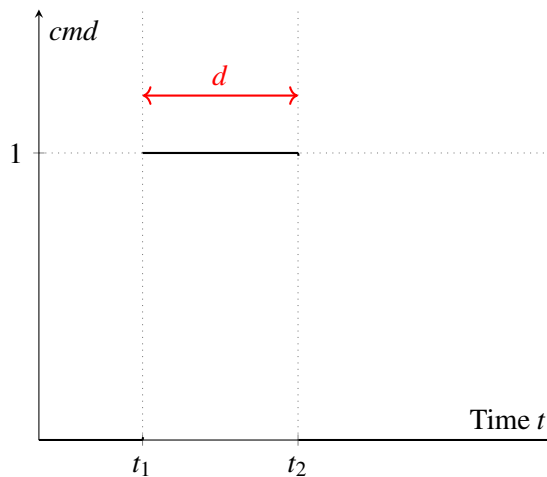


Figure 4.25: ESP32 Pulse Function

Since the code implemented on the initial script is incompatible with the ESP32-S2 code, several adjustments were necessary to ensure it operated correctly. Thus, a refined version of the script was created in Unity. This new algorithm followed the same logic stated in the initial script,

but one of the major changes in this algorithm is based on how the inputs are received, and how they are treated. In order to replicate the behavior of the ESP32-S2, a UDP listener would be initiated in the *Start* function, and it would receive the string with the car command listed in Table 4.4 instead of assigning these values directly.

In order to process the movement input and to replicate the time when the ESP32-S2 does not receive any new command due to having a movement command active, a function was implemented. This function received two inputs: the *speed* and the delay *d*. Whenever this function was called, it would set the value of the forward velocity (*v*) as the value of the *speed* parameter, which would be either 1 if the car is to move forward, or -1 if to move backward. The command delay would then activate for a *d* duration, which could be increased or decreased.

Regarding the steering of the virtual cars, seven possible values for the front wheel steering were defined: -13.4260° , -8.9573° , -4.4787° , 0.0000° , 4.4787° , 8.9573° , and 13.4360° . The values on the extremities were known from the calculations of the car parameters in the equation 4.3, and are the value the real car steers to whenever he receives *ml* or *mr*. The intermediate values were added to allow the car to have extra maneuverability. To arrive at the 4.4787° value, the extremity value was divided by three, as there are three steering positions on each side. 8.9573° was determined by multiplying 4.4787° by two. These angles were then mirrored to their opposite steering. These values were stored on an array that would have 0° as the default, and the other values could be navigated by receiving the commands *ml* and *mr* to steer to the left or the right, respectively. Fig. 4.26 shows the positions the wheels can now steer to, to the left.

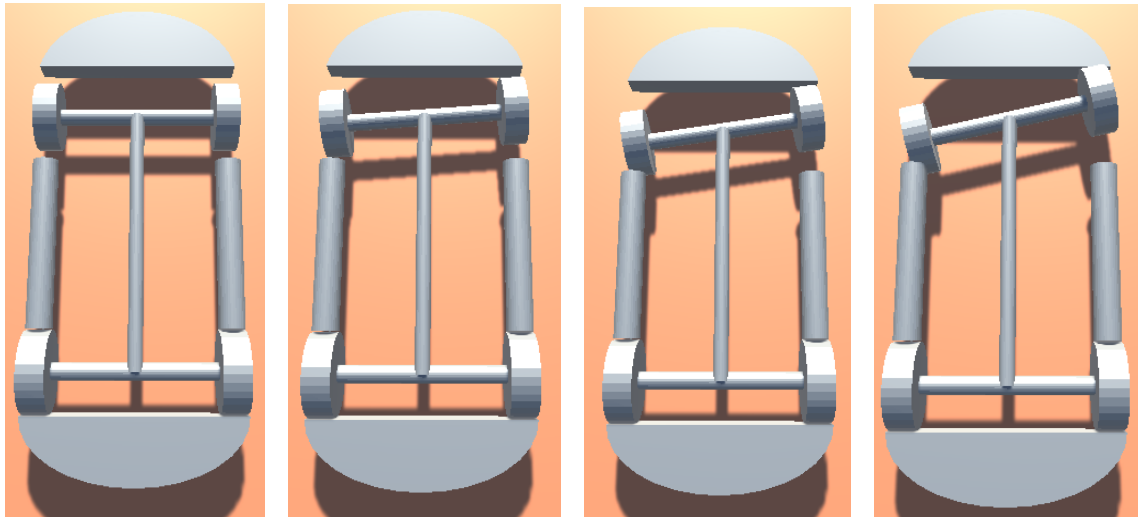


Figure 4.26: Car Steering from Default (left) to Maximum (right)

4.3.4 Virtual Cameras

In Unity, simulations are observed through virtual cameras, which are treated similarly to other objects. A virtual camera was positioned above the race track, replicating the position of its real-life counterpart. This setup allows for the entire race track to be visible at once, proving crucial

for the monitoring aspect of the digital twin. This viewpoint can be seen in Fig. 4.27, compared to the real race track in Fig. 4.28.



Figure 4.27: Virtual Race Track Camera View

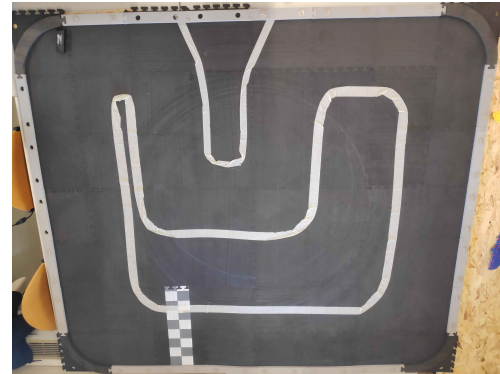


Figure 4.28: Real Race Track Camera View

Another virtual camera was added to the modeled car and linked to it as its *child*. This allowed the virtual camera to move in sync with the car, tracking it as it navigated the race track. The virtual camera was positioned behind the car, facing forward, with a wide enough field of view to capture the sides of the race track and the farthest wall, as shown in Fig. 4.29. This virtual camera did not exist in the real world; instead, it was created and used to provide a different insight into the digital twin, allowing the car to be followed directly during simulations or monitoring.

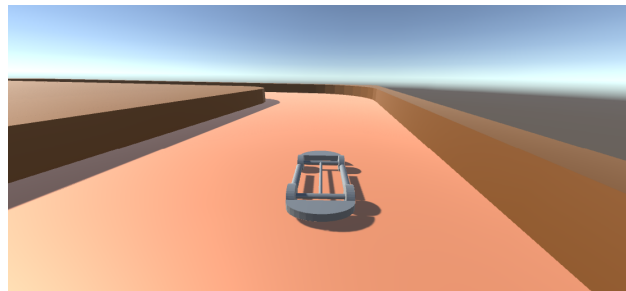


Figure 4.29: Virtual Race Car Camera View

Chapter 5

Digital Twin Functionality and Monitoring

In this chapter, an overview of the functioning of the cyber-physical system composed of the digital twin and the real-world components is presented in section 5.1. Section 5.2 outlines the second phase of the digital twin implementation, focusing on the development of the monitoring algorithm, including the communication protocol and all elements ensuring the digital twin's functionality, including statistics calculated, and the storage of important values. Finally, the third and last phase of the digital twin's development, the machine learning framework, is detailed in section 5.3. The results and a comparison between this work and two similar ones are presented in section 5.4.

5.1 Digital Twin Functionality

The cyber-physical system acts as a link between the physical and virtual environments. It merges physical and digital processes, allowing for real-time data collection, analysis, and interaction. On the other hand, the digital twin is designed not only to replicate the real world and enable real-time monitoring but also to conduct tests within a virtual environment and to send instructions, like movement or steering, to the real cars, enabling their control. If one part fails, the other part continues to function independently, regardless of whether the failure originates from the real world or the virtual environment. In Fig. 5.1, an overview of the cyber-physical system is presented.

The Real World consists of all tangible elements, such as the race track, the race cars, and obstacles. In contrast, the Digital Twin is represented by Unity, which includes everything implemented within it, such as the digital cars and the associated algorithms. In this context, the digital cars are divided into two types: the Virtual Car (VC) and the Ghost Car (GC). The VC is a digital representation of the real car, based on the positions provided by either the Real or the Virtual Perception Systems. It is used not only for real-time monitoring but also to gather data to be stored in a database. In contrast, the GC is more complex than the VC. This complexity arises

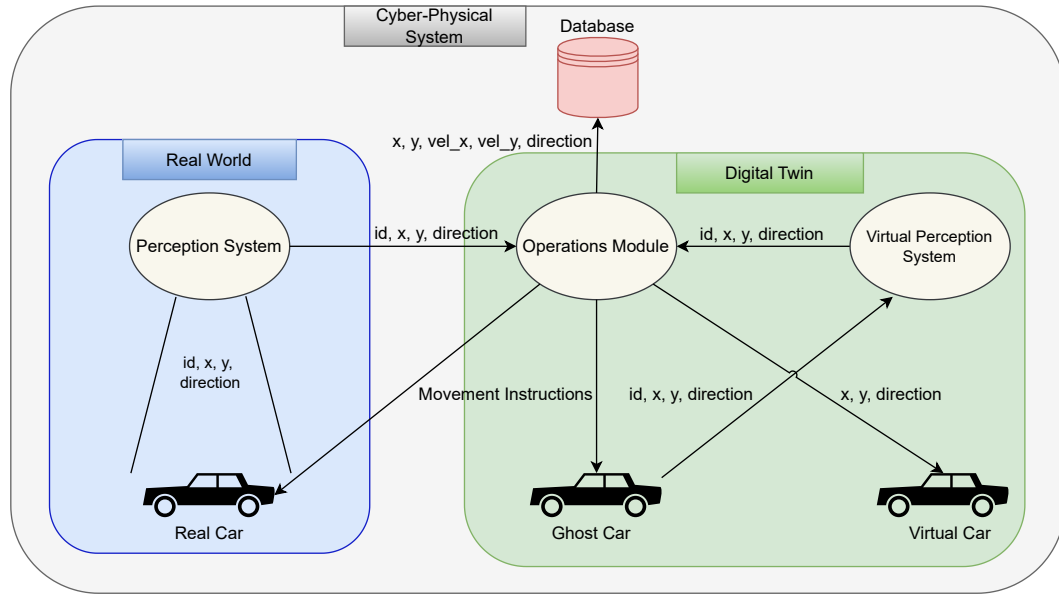


Figure 5.1: Cyber-Physical System Overview

from an algorithm that enables it to replicate the behavior of a real car. The GC receives instructions through UDP and moves accordingly, similar to its real-life counterpart. When a command is sent to the real car, it is also sent to the GC. Ideally, the VCs and GCs should always overlap, validating the accuracy of the simulation and ensuring that the digital system faithfully replicates the real car's behavior.

The Operations Module (OM) is used to drive the cars on the race track by sending instructions to both the Real Car and the GC simultaneously, ensuring that their movements are synchronized. A typical real-world perception system collects the Real Car's coordinates (x, y) , along with its direction and identification id (translated through the car's color), and sends this information back to the OM. The OM then positions the VC based on the values received. A similar process occurs with the VPS, which collects the GC's coordinates, direction, and id . The VPS was created to ensure that the virtual environment is still operational even if the Real Perception System (RPS) fails, or is not available. In this case, where the real-life system is not operational, the VC would receive the coordinates and direction of the GC, provided by the VPS, instead of the Real Car and it would be positioned accordingly based on the received data.

Additionally, The OM stores the coordinates (x, y) , the velocity along those axes, and the direction of the VC in a database. It also includes various functionalities, such as displaying statistics like current speed, idle time, and fastest lap time. Finally, it incorporates security measures to ensure that the values stored in the database are valid, preventing scenarios where the car could leave the track or if it was grabbed.

There may be situations where the RPS is not available or is not being used. This could occur, for example, if the physical camera fails or if more extreme scenarios are intended for testing. In such cases, as illustrated in Fig. 5.2, the OM sends the movement instructions solely to the GC,

and then the VPS determines its coordinates, direction, and id and forwards them back into the OM, which uses them to position the VC. In these instances, the RPS is not present, and only the digital twin is utilized. One disadvantage of this scenario is that it relies on the high accuracy of the car models. Otherwise the values sent to the OM by the VPS are different from the ones the RPS would send.

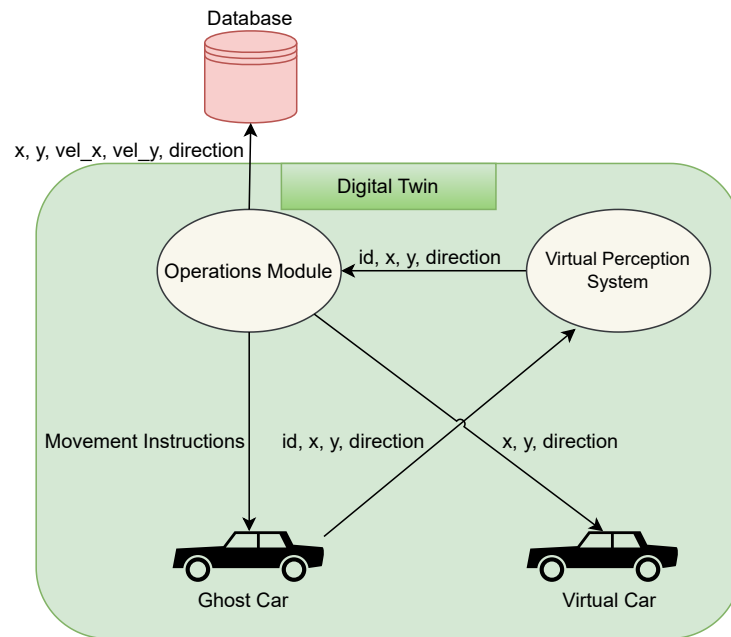


Figure 5.2: Digital Twin Overview

The final scenario describes a situation without a VPS, where autonomous vehicles navigate and respond to their environment using data provided by the RPS. This system allows the testing of the algorithms developed to control the cars to verify if they perform reliably in the real world. It provides accurate testing conditions, validates their performance, and helps identify any limitations that might not be visible in simulations. It also monitors real-world factors such as energy consumption, which cannot be simulated. This makes it essential for hybrid workflows that improve the accuracy of digital twins in predictive modeling and decision-making. For example, in contexts like Formula 1 races, these systems can account for elements like pit stops, where real-time data on tire wear, fuel levels, and track conditions—factors that are difficult to simulate—affect strategic decisions.

In this scenario, as depicted in Fig. 5.3, the OM sends movement instructions only to the Real Car. The RPS then determines the car's coordinates, direction, and id and forwards this information back to the OM. The OM subsequently positions the VC accordingly. This approach allows for a reduction in the digital twin, although it assumes that the RPS performs correctly, and with reasonable accuracy.

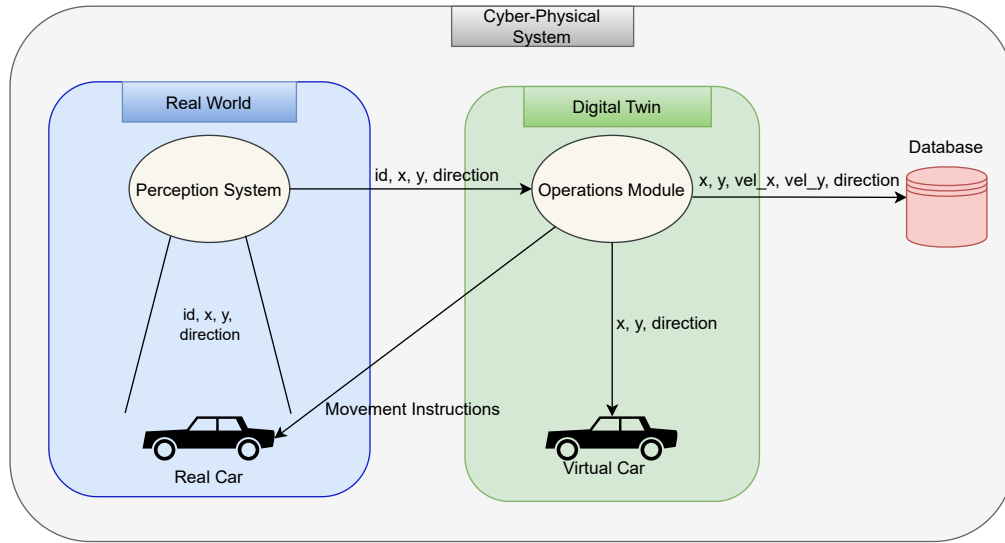


Figure 5.3: Cyber-Physical System Without the Virtual Perception System

5.2 Monitoring

The second phase of the implementation of the digital twin aims to monitor the race cars in real-time, which requires fast communication between Unity and the Perception Systems. This communication involves the reception of data regarding the coordinates and the direction in which each car is facing. Since the RPS was not available, a VPS was used in order to mimic the behavior of this entire system.

5.2.1 Virtual Perception System

The VPS is an algorithm designed to accurately identify the positions of race cars and relay that information to the OM. This process imitates the method that the RPS would use to transmit the information to Unity.

The protocol that establishes the connection between the VPS and the OM can be implemented using either UDP or TCP - Transmission Control Protocol. TCP is a connection-oriented protocol, meaning that it establishes a reliable connection between the sender and the receiver, ensuring that any lost information is retransmitted [34]. In contrast, UDP does not create a reliable connection, which means that some information may be lost. However, in the context of this project—autonomous vehicle racing—if a car's position is lost during communication, it is not critical for that information to be resent later. This is because the position would be outdated and no longer relevant. Therefore, UDP was chosen as the preferred protocol. In this scenario, the sender is either the Real or Virtual Perception System and the receiver is the OM, which forwards the information provided to the VC.

The information provided by the Perception System is sent in a string with four components divided by a comma in the following order: the car ID, given by the car's color; the x coordinate, the y coordinate; and finally, the direction, visible in Fig. 5.4.

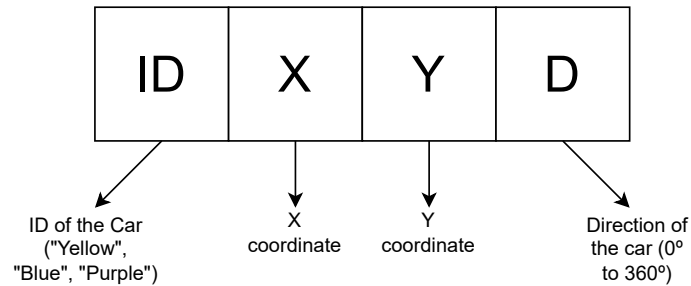


Figure 5.4: String sent from the Perception System

5.2.2 Virtual and Ghost Cars

To facilitate visual differentiation among the digital cars, the VCs were created by making three copies of the modeled car, each assigned a material to reflect the color of its real-life counterpart: one car is yellow, another is purple, and the last one is blue. Additionally, to enable effective monitoring, a copy of each car was made in a lighter color. These are the *Ghost Cars*. Fig. 5.5 shows the VCs closer to the camera and the GCs of the corresponding color behind them, which are not overlapping due to this image being solely for visual representation of the cars, thus the monitoring algorithm was not running.

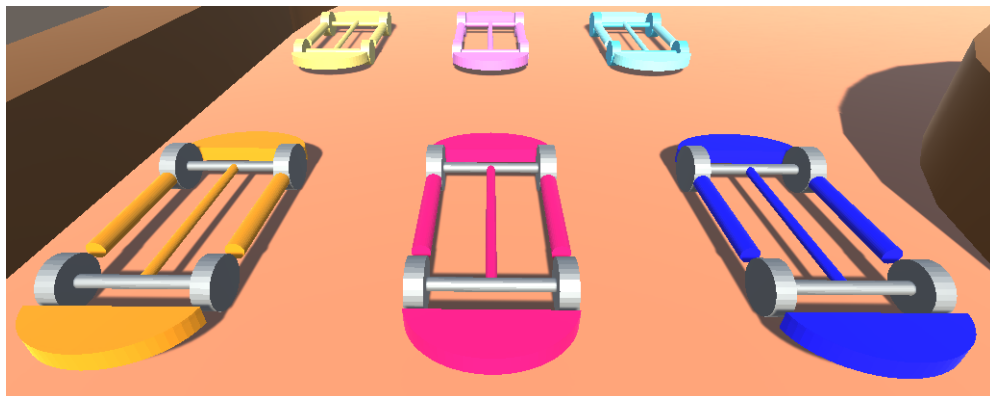


Figure 5.5: Virtual And Ghost Cars

The GC's objective is to replicate the real cars' behavior on the race track. Therefore they are assigned the refined movement script described in the subsection 4.3.3. On the other hand, the VCs act as digital representations of their real-life counterparts, accurately reflecting their position and direction. This information is derived from the RPS, which monitors the real cars' movements. However, in cases where the real camera is not functioning properly, the VCs rely on the data provided by the VPS, which sends the data regarding the position based on the GCs. Based on

the position and direction of the VCs, data is stored in a database, which can later be used to train models, and several statistics are gathered, like the number of laps by car.

The algorithm responsible for sending movement instructions has been enhanced to allow for individual control of each car. This improvement involved modifying the movement algorithms on each car and assigning a unique communication port in the protocol to each one. In this context, a "port" refers to a designated channel through which data is transmitted, allowing each car to receive specific instructions without interference from others. In practice, to select the car to which Unity would send instructions, keyboard buttons 1, 2, and 3 were used to select the port related to that car. For instance, pressing button 1 would select the port for the yellow car, button 2 for the purple car, and button 3 for the blue car.

5.2.2.1 Collisions

To prevent collisions between the VCs and GCs in Unity, the collision interactions between these two kinds of vehicles were disabled. This was achieved by creating two separate layers: one for the VCs and another for the GCs. After the appropriate layers were assigned to each car object, the project settings in Unity were accessed, and the Physics section was navigated to. The collision matrix was then modified by unchecking the box that represented the interaction between the layers of the VCs and the GCs, depicted in Fig. 5.6 as layers *NormalCars* and *ghostCar*, respectively. As a result, collisions between the VCs and GCs were eliminated, while they could still collide with other elements, such as the race track's walls.

	Default	TransparentFX	Ignore Raycast	Water	UI	ghostCar	NormalCars	Walls	TrackSurface	Checkpoints
Default	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ignore Raycast	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Water	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
UI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ghostCar	✓	✓	✓	✓	✓		✓	✓	✓	✓
NormalCars	✓	✓	✓	✓	✓	✓		✓	✓	✓
Walls	✓	✓	✓	✓	✓	✓	✓		✓	✓
TrackSurface	✓	✓	✓	✓	✓	✓	✓	✓		✓
Checkpoints	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Figure 5.6: Unity Collision Matrix

Although the GCs and VCs could no longer collide with each other, two main collision issues remained. The first issue involved collisions between GCs. In some instances, when a GC collided with another GC at a higher speed, it would begin to spin. This was caused by an angular velocity that was created from the collision, due to the lack of constraints on the rigid body of the car. This issue made the car lose its horizontal alignment, as shown in Fig. 5.7.

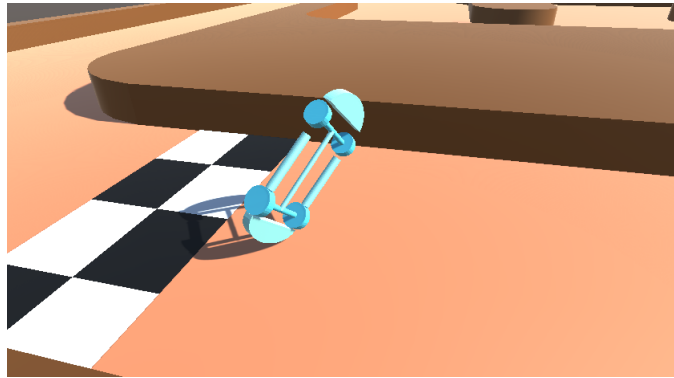


Figure 5.7: Ghost Car Losing its Horizontal Alignment

This could be solved by changing a constraint in the rigid body of the cars: enabling a constraint where the Z axis is frozen in place, which means that the cars would stay in the same layer of the race track all the time, which is valid for this work since the race track is a flat surface. This solution was applied to both the GCs and the VCs' rigid bodies.

Another issue concerning collisions is that when a VC collides with another of the same type, it occasionally gains force as a result of that collision. This force should not be applied, as the aim of the VCs is solely to reach their assigned positions. Any forces resulting from potential collisions would already be considered for the GCs or real cars. If the VC gained a force from the collision, it would start to phase off the position provided by Perception Systems, resulting in the case of Fig. 5.8

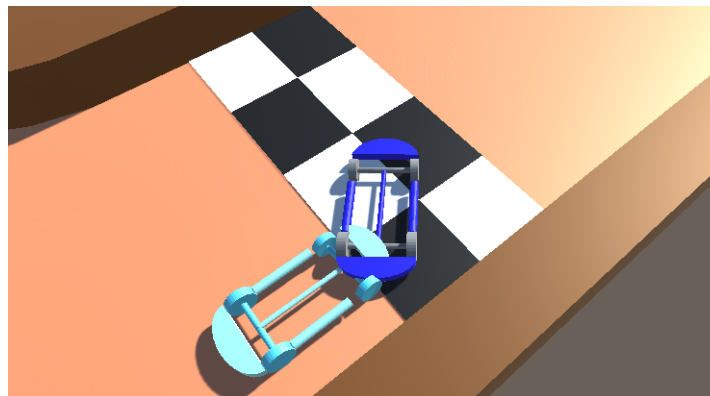


Figure 5.8: Virtual Car Faulty Positioning Resulting from a Collision

The issue was resolved by enabling the Kinematic option in the rigid body settings of the VCs. By activating this option, the VCs no longer experienced any physical forces, such as those resulting from collisions, which effectively solved the problem.

5.2.3 Virtual Cars' Real-time Monitoring

When the program begins running in Unity, the monitoring algorithm positions the GCs at specific locations defined within its code, typically right before the finish line, and the VCs are teleported

to the GC of the corresponding color. The initial position can be changed, and it is useful to select where exactly each car starts, allowing for customized starting options. The Perception System then begins collecting the position, direction, and the id of the cars, which is received by the monitoring algorithm. The algorithm then separates the values into four distinct strings and uses a function to convert the strings for the x and y coordinates, as well as the direction, into numerical values. Based on the id, the monitoring algorithm updates the position and direction of the corresponding car. To manage the reception of communication strings, for both the real (if it exists) and the VPS, a trigger was developed to enable and disable their reception. A customizable delay was also created in the VPS, which was meant to replicate real-life delays. Fig. 5.9 shows how the blue VC reacts to a communication delay of 0 seconds, while Fig. 5.10 contains a delay of 0.5 seconds.

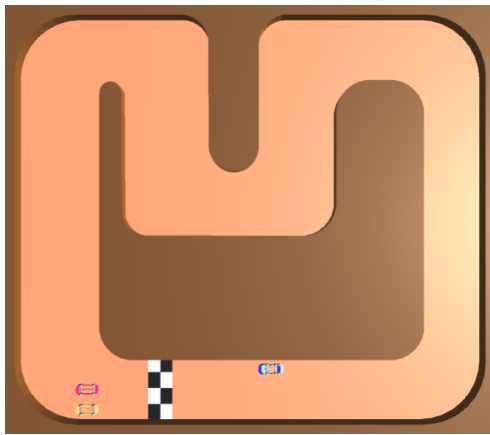


Figure 5.9: Communication Delay of 0s

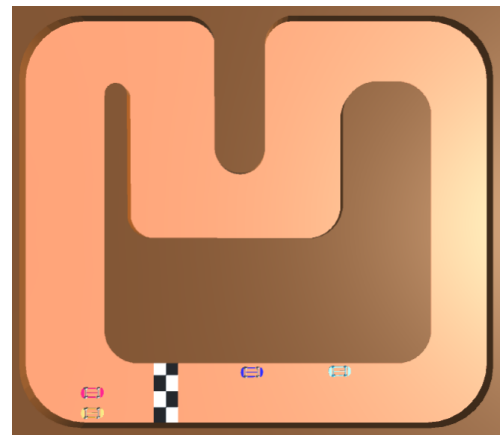


Figure 5.10: Communication Delay of 0.5s

5.2.3.1 Safe Position

Before updating the VCs' positions, it was essential to ensure that the new coordinates were considered "safe." A safe position means that the car's location is valid, without any external disturbances or malfunctions affecting its behavior. This work focused on two specific cases:

- The car was grabbed;
- The car left the race track.

To prevent both scenarios, the positions and directions of the cars were stored in a unique variable for each car, which contains their last valid position. Subsequently, a function was created to check if a car is in a safe position. This function addresses both cases: when a car is grabbed and when it leaves the race track.

In the first scenario, where a car is grabbed, the function calculates the distance between the car's current position and its last recorded position. If this distance exceeds a specified threshold,

the car is considered to be in an unsafe position. This threshold had to be carefully chosen to prevent false negatives, where the car was grabbed and placed in a nearby place but was not detected, and false positives, where it detected the car being grabbed when it was not, from happening.

In the second scenario, in order to detect when a car leaves the racetrack, a ray was added to the car. This ray is emitted downwards from the car, making the ray collide with the surface of the race track. If, at any moment, the ray stops detecting the surface—such as when the car leaves the racetrack—the unsafe position of the car is triggered.

Whenever one of the unsafe positions triggers, the car gets teleported back to the last safe position stored, and stays there until a new position considered safe is received, which requires the GC to be nearby the VC's last safe position. In Fig. 5.11, the GC left the race track, and then returned to it further ahead. The VC followed it but triggered the unsafe position when it left the race track, since the ray stopped detecting the race track's surface, and therefore it was teleported back to the last safe position that was stored, where it would stay until the GC gets close to it.

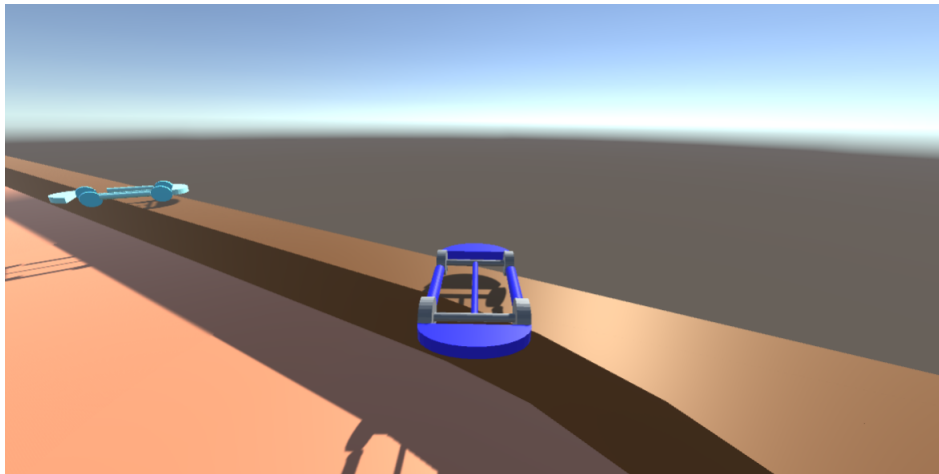


Figure 5.11: Virtual Car in its Last Recorded Safe Position

If the car does not trigger an unsafe condition, a function is used to update the car's position and move it to its new location. Before updating the position, this function verifies that the values provided by the Perception System are valid by checking whether the coordinates fall within the physical limits of the racetrack. This involves establishing minimum and maximum values for each coordinate on the surface, which adds an extra layer of security to the monitoring component of the digital twin.

If the coordinates are valid, the car's safe position variable is updated to reflect the values stored in an auxiliary variable. This auxiliary variable then records the current position and direction of the GC.

In a scenario where a car is deliberately lifted and relocated to a different spot on the race track, like when the race is about to start and the car is placed on the finish line, the unsafe position alert is triggered. However, this can be overridden by pressing a specific key, which allows the VC to bypass safety conditions and be teleported to the next position received from the Perception System, provided that this position is still within the limits of the race track. Fig. 5.12 reflects the

same scenario described in Fig. 5.11, but after pressing the bypass trigger, thus the VC overlapped with the GC's position.

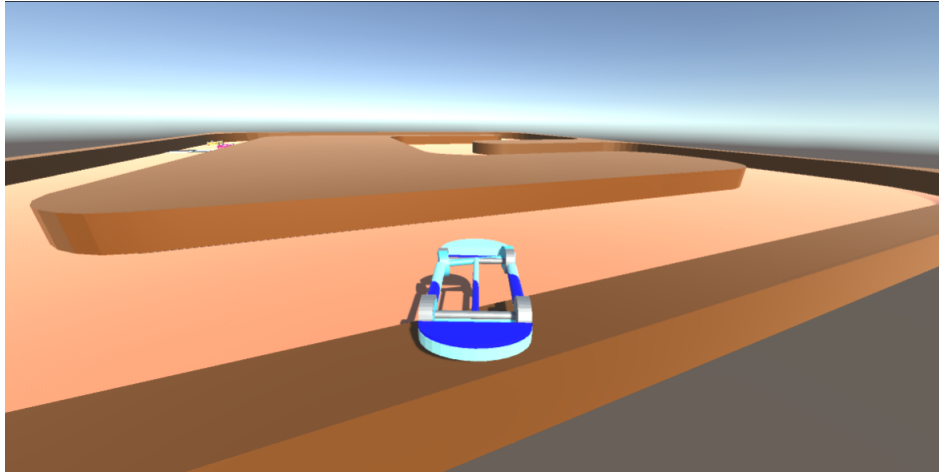


Figure 5.12: Virtual Car After Bypassing the Safe Position

5.2.3.2 Database

The database is considered to be crucial for this work, as the data stored within it can later be utilized for the development of machine learning algorithms.

For this project, the package SQLite4Unity3d [35] was chosen to create the database, since it was the most suitable option. SQLite4Unity3d is a library that enables SQLite database functionality to be integrated within Unity, allowing a local data storage solution to be created for Unity applications. SQLite, known as a lightweight, serverless relational database engine, is accessed and manipulated through a simple interface provided by SQLite4Unity3d directly within Unity.

This package was used to create a database with the following parameters: a timestamp with the current time and date, the car name (Blue, Yellow, or Purple), x coordinate, y coordinate, velocity in the x-axis, velocity in the y-axis, and the direction of the car. The x and y coordinates are determined by the global location on the race track, while the velocity is based on the individual axis of each car. The coordinates are transmitted based on the last safe location of each car. To calculate the velocity along the x and y-axis, the vector of the distance traveled is determined by subtracting the current position and the last safe position recorded, and then it is divided by the time taken. The resulting vector is separated into its two components, velocity of x and y, which are then inserted into the database. The direction of each car, going from 0 to 360°, is always based on the last reported value that is provided by the Perception Systems.

To visualize the values stored in the database, the application DB Browser for SQLite [36] was used. This application allows to filter results to see only the relevant results. For instance, if only the blue car was being tested, the application could be used to filter so that only the blue car's values are displayed. In Fig. 5.13, it is visible to see the values stored in the database in the scenario where the yellow and the purple cars were standing still, and the blue car was moving in a

straight line along the y-axis, but slightly turned to the side, which can be seen due to the constant direction value and the variation on the y-axis, while the x-axis had a small variation through the time.

Id	Timestamp	CarName	PositionX	PositionY	VelocityX	VelocityY	Direction
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
455	2025-01-30 14:40:31	Blue Car	27.9085006713867	126.283897399902	1.12796592712402	54.8572158813477	193.789398193359
456	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999801635742
457	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.0
458	2025-01-30 14:40:31	Blue Car	27.6289005279541	125.116798400879	1.19106769561768	59.9943542480469	194.60969543457
459	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999893188477
460	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.0
461	2025-01-30 14:40:31	Blue Car	27.3071994781494	123.854499816895	1.37429046630859	65.1178512573242	195.506698608398
462	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999893188477
463	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.000106811523
464	2025-01-30 14:40:31	Blue Car	26.9393997192383	122.49829864502	1.60250854492188	70.2412261962891	196.480499267578
465	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999893188477
466	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.0
467	2025-01-30 14:40:31	Blue Car	26.5209007263184	121.049896240234	1.86131286621094	75.3595657348633	197.530899047852
468	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999893188477
469	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.0
470	2025-01-30 14:40:31	Blue Car	26.137300491333	119.81600189209	1.56560897827148	64.58837890625	198.658294677734
471	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999801635742
472	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.000106811523
473	2025-01-30 14:40:31	Blue Car	25.7877998352051	118.765602111816	1.17900276184082	55.3383865356445	199.624404907227
474	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999801635742
475	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.0
476	2025-01-30 14:40:31	Blue Car	25.4731006622314	117.871696472168	0.87458324432373	47.3760833740234	200.452194213867
477	2025-01-30 14:40:31	Purple Car	20.0	200.0	0.0	0.0	179.999893188477
478	2025-01-30 14:40:31	Yellow Car	10.0	200.0	0.0	0.0	180.000106811523

Figure 5.13: Values Inserted in the Database (with Filter)

One issue encountered was that the velocity along the x-axis of the car objects, which corresponds to their lateral movement, exhibited significant values even when the car was moving in a straight line. Ideally, in this case, this value should be zero. To address this, a filter was implemented: if the variation in the velocity along the x-axis was minimal, it would not be considered in the calculations.

5.2.3.3 Statistics

The statistics were designed to offer valuable insights into key parameters of the race cars while on the track, highlighting potential optimization areas and revealing the vehicles' behavior. These parameters included:

- average speed (cm/s);
- maximum speed (cm/s);
- distance traveled (cm);
- current lap time (s);

- fastest lap time (s);
- laps completed;
- idle time (s).

The distance traveled was calculated by subtracting the current position and the last position and calculating its absolute value. This value would then be added to the existing distance traveled variable.

To calculate the average speed, the system updates the average when the vehicle's speed exceeds a small threshold, indicating movement. The updated average speed is determined using a running average formula, while a variable that counts how many times the average speed has been calculated is incremented. Whenever a new speed value is added, the current average speed is multiplied by the number of times it has been calculated. Then, the new speed is added to this total. Finally, the sum is divided by the total count of speed calculations plus one, resulting in the updated average speed. The formula can be seen below:

$$\text{Updated Average Speed} = \frac{\text{Current Average Speed} \times \text{SpeedCount} + \text{CurrentSpeed}}{\text{SpeedCount} + 1}$$

When the vehicle's speed drops below a certain threshold, it is classified as idle, meaning that the car is considered to not be in motion, and the idle time is increased while the idle state is active. This approach allows for ongoing monitoring of both movement and idle periods, offering a thorough understanding of the vehicle's operational behavior.

The maximum speed is calculated by checking the distance traveled between two successive *FixedUpdate* calls, dividing that distance by the time that passed between these calls, obtaining the car speed, and whenever that speed exceeds the current maximum speed, that value is updated.

To initiate the lap time counter, the cars needed to pass through the finish line. This action activated or restarted the lap time counter. A box collider was added to the finish line, with the "Is Trigger" option selected. This setting allowed the collider to respond to trigger events, such as a race car colliding with it. Additionally, tags were assigned to both the VCs and the finish line. In Unity, tags are identifiers that categorize objects for easier identification and management, which enable the implementation of logic for collision detection and object interactions.

Two tags were created, *Car* and *FinishLine*, assigned to the VCs and to the finish line, respectively. In the statistics algorithm, whenever an object with the Car tag collided with the object with the FinishLine tag, the lap time counter would start counting, or if it was already running, it would reset it. Collision triggers occurred whenever any part of the car's hitbox made contact with the finish line's hitbox, typically at the front of the race car and the beginning of the finish line. The collision box for the finish line was also given an increased size to ensure that the cars would not miss it, as seen in Fig. 5.14, delimited by the green lines.

The final parameter is the number of laps completed. To ensure accurate tracking, several checkpoints were placed along the race track. The purpose of these checkpoints was to prevent

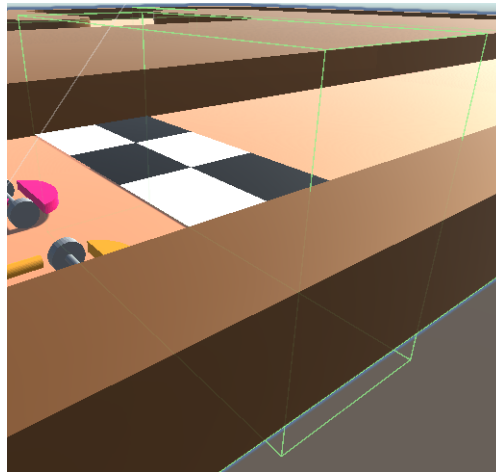


Figure 5.14: Finish Line Collision Box

any possibility of "cheating", which would allow a car to increase its lap count without completing the entire track. If a car did not pass through all the checkpoints before crossing the finish line, the lap count would not increase. This process functions like a state machine. These checkpoints were created by positioning small plane objects near the four corners of the race track. These planes were assigned an invisible material so they would not be visible. Similar to the finish line, each checkpoint was given a unique tag and equipped with a large box collider set to the "Is Trigger" option, as illustrated in Fig. 5.15. In order for the car to increase its lap count, it would need to pass through each checkpoint in the order they were meant to be crossed in the real race track. Even though four checkpoints were placed on the race track, more could be placed if necessary.

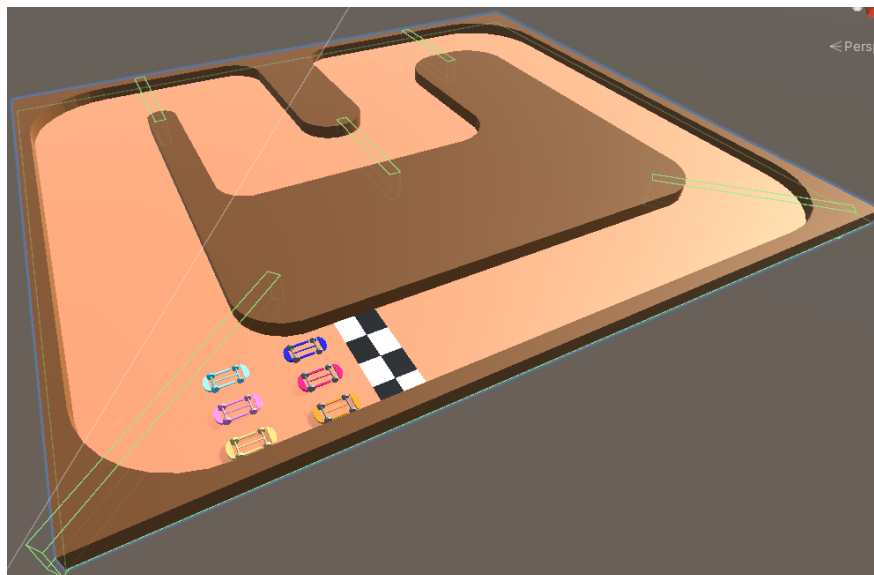


Figure 5.15: Race Track Checkpoints

Once all six parameters had been calculated, a method for visualizing them needed to be implemented. The Canvas function in Unity was used to achieve this. The Canvas in Unity serves

as the root for all UI elements, determining how they are rendered and displayed. For this project, several canvas objects were created, with each one designed to represent a specific parameter of a particular car. Specifically, three canvas objects were created, one for each car, and six child elements were added to each object, one for each parameter.

The child elements were positioned to ensure that the parameters could be visualized while the program was running, without obstructing important parts of the screen. The individual statistics for each car were displayed within their respective camera screens, as illustrated in Fig. 5.16.

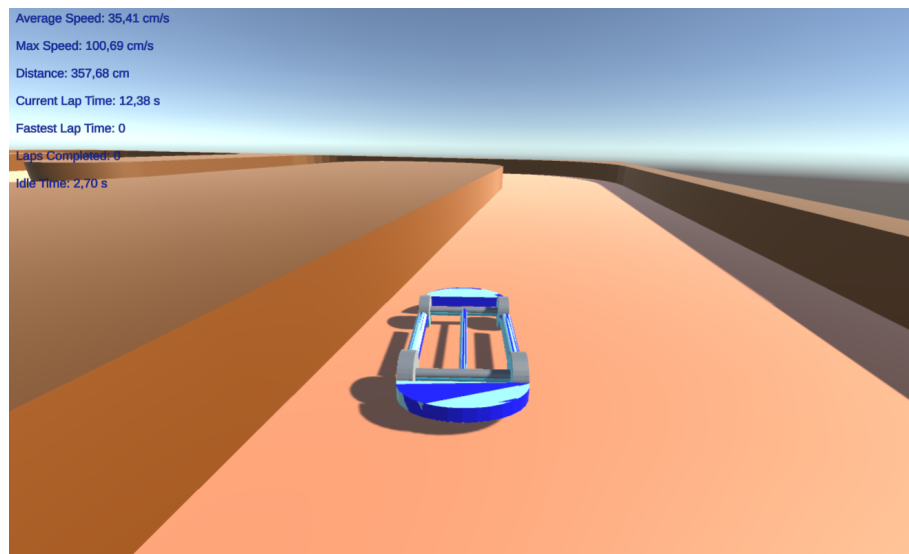


Figure 5.16: Individual Virtual Car's Statistics Display

It also was implemented a way to visualize all the statistics of the cars on the virtual camera screen placed above the race track, allowing for real-time comparison of these values instead of viewing them individually. This can be seen on Fig. 5.17.

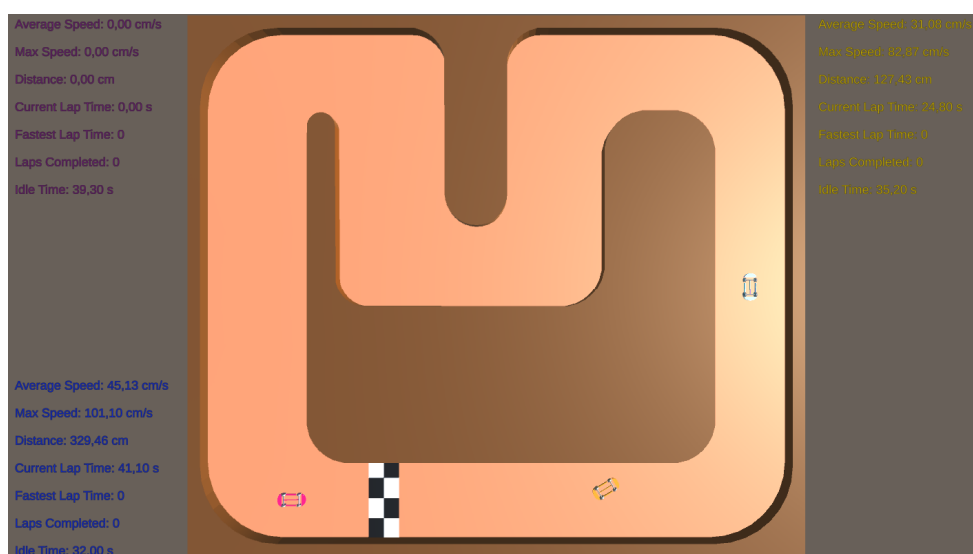


Figure 5.17: Global Virtual Cars' Statistics Display

5.3 Machine Learning Integration

To complete the implementation of the digital twin, the final step was to incorporate the machine learning framework. The most suitable framework for this purpose is *ML-Agents* ([37]), which is an open-source tool that enables developers to utilize reinforcement and imitation learning in Unity projects. In the case of this project, this framework was used by the control [16] to enhance the car's behavior over time, leading to a more efficient trajectory and, consequently, a reduction in lap time.

5.3.1 ML-Agents Implementation

The ML-Agents were implemented in Unity by following a guide provided by [38]. This involved downloading the ML-Agents package from a GitHub repository ([37]) and importing it into the Unity project. The package would contain multiple libraries used for implementing reinforcement and imitation learning, utilized by the control work. After importing the package, the ML-Agents were tested following another guide, [39].

This guide involved creating a Unity environment from scratch, where a plane served as the surface. A cube was added to be the target of a sphere, which symbolized the Agent (Fig. 5.18). An Agent, defined as a self-operating entity, perceives and engages with its surroundings. The plane and cube had a box collider, while the sphere had a mesh collider and a rigid body, allowing it to move. The purpose of the sphere was to collide with the cube without falling off the plane.

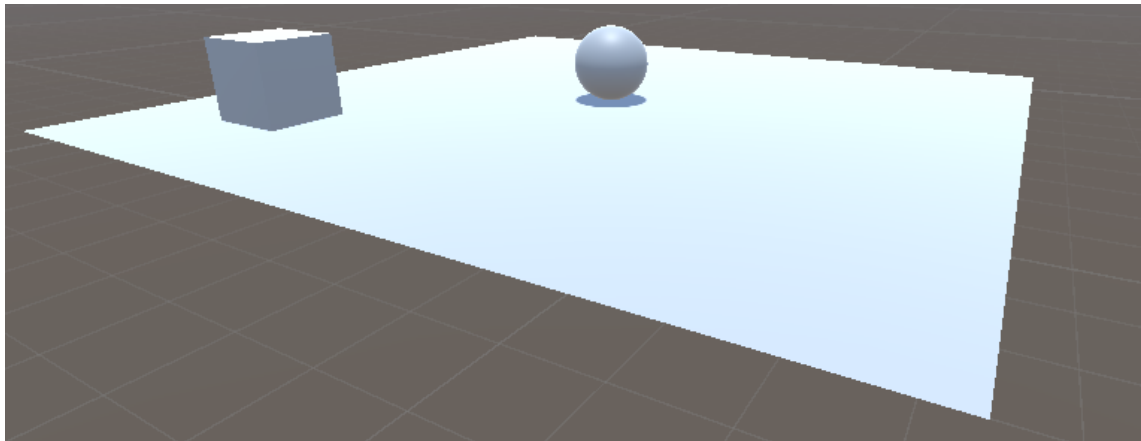


Figure 5.18: RollerBall Environment

A script named *RollerAgent* was created and assigned to the sphere. The script would depict the behavior of the Agent. If the sphere fell off the plane, it would be teleported back up to a random position on the plane. The Agent collects data and sends it to the Brain, which analyzes it using a neural network to make decisions. This process involved training the Agent with data that was converted into a feature vector. Therefore, in this script, the Agent would also receive observations referring to its position and velocity and the cube's position. In the final part of the script, two actions were added to the Agent: one would apply a force to the x-axis and another to

the y-axis, which allowed it to move freely in the plane. Additionally, a reward was received by the sphere whenever it collided with the cube, prompting the end of the episode. The episode was also concluded if the sphere fell off the plane.

After training the environment, it was evident that initially the sphere was falling off frequently. However, after a few minutes, it became much more proficient at pursuing the cube without falling off for an extended period, meaning that the ML-Agents library was working the way it was expected.

5.3.2 Race Car Scripts Modification

Upon testing the ML-Agents framework, the subsequent step entailed modifying the race car's movement and wheel steering algorithms as detailed in subsection 4.3.2. The modification involved replacing the arrow key inputs (up, down, left, or right) with direct variable assignments through an algorithm devised by the control work. This update eliminated the use of the function that gathered the user arrow keys input for the forward velocity (v). Hence, the steering angle (δ) was no longer computed but directly assigned by the control work. The same logic was followed in the wheel steering algorithm, in which the rotation of the front wheels would be directly assigned in the control work instead of using the inputs from the left or right arrow keys.

Assigning the values of the variables directly through scripts was possible and done in the control script by invoking the classes defined in each script and subsequently assigning values to control steering and movement to their respective variables. The steering angle (δ) could be assigned any value, although it would be automatically limited by the code to be within the minimum and maximum value, allowing for precise control. Meanwhile, the forward movement (v) could be set to -1 for reverse, 0 to stop, and 1 to accelerate forward.

After all the modifications, the car could be controlled around the race track through an ML-Agents algorithm, without requiring any manual control of the car. This allowed the control work to develop and train its algorithms within the virtual environment.

5.4 Results

After the digital twin implementation was completed, Unity's virtual race track could accurately replicate the real track, as can be seen in Fig. 5.19.

Additionally, Unity could connect to the digital model and to the real car and give instructions to each simultaneously, both of which are highlighted in Fig. 5.20. This allowed them to be manually driven around the digital and real race tracks, respectively. This was crucial for testing the movement algorithms that had been implemented and for conducting imitation learning. However, due to a simplified cinematic model being used for the car, it does not take into consideration, for example, lateral velocity. Therefore it is not expected that the digital cars mimic the real system in its totality.

The real-time monitoring allowed the cars to be moved to their corresponding positions in the digital environment at the same pace as their real-life counterparts, thereby allowing the virtual



Figure 5.19: Virtual Race Track Replicating the Real Race Track

environment to mirror the real world. The monitoring ensured that the values provided by the VPS were valid, which meant the car was neither grabbed nor left the track. The statistics gathered, which could be visualized in real-time in Unity, allowed for the cars' behavior to be better understood, and several values regarding the cars' positions, velocity, and direction were stored in a database, which could be used by the machine learning algorithms later.

The machine learning framework that was implemented in this project made it possible to use imitation and reinforcement learning algorithms and to conduct parallel training. This significantly sped up the learning process by allowing separate instances to run at the same time.



Figure 5.20: Unity Controlling the Real and Virtual Car Simultaneously

Comparing with the works reviewed in the subsection 2.2.4, it is possible to highlight several distinguishing factors of this study that set it apart from the rest. One of the advantages this work had compared to the study by [5] lies in the capability gap observed in real-life system monitoring. The study by [5] was constrained to a digital environment and did not have the provision to monitor

systems in real life. This proved to be a significant limitation as the ability to observe and analyze real-world interactions provides deeper insights and validation for theoretical models.

This work also demonstrated numerous similarities with the studies described by [4], including that it allowed the usage of imitation and reinforcement learning, as well as the implementation of parallelization techniques, like parallel agents or environments. Nonetheless, a noteworthy distinction is that the research presented in this dissertation successfully enabled the monitoring of three vehicles, visible in Fig. 5.21, surpassing the capacity of the study by [4], which was limited to the monitoring of just a single real vehicle.

One aspect that distinguishes this work from the other two lies in the monitoring phase, which enabled the gathering of several statistics and the storage of various parameters in a database.

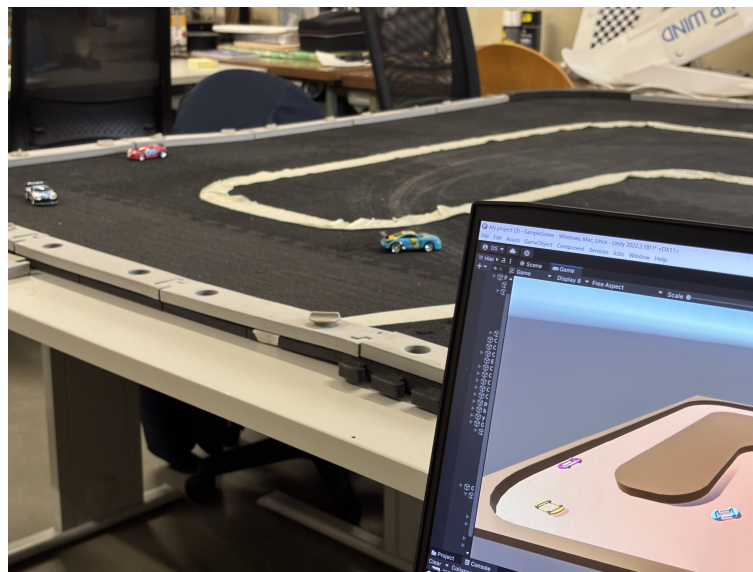


Figure 5.21: Unity Communicating with the Three Cars

Chapter 6

Conclusion

The goal of this dissertation was to develop a digital twin that could monitor and simulate autonomous race cars on a race track, on a scale of 1:43, in real-time. In order to achieve this, literature reviews were conducted on autonomous racing and digital twins. Based on the literature review, it was decided to use the game engine Unity to create the digital twin due to its user-friendly interface and extensive features. The development of the digital twin occurred in three phases.

In the first phase, measurements of the race cars and the racetrack were taken. This was followed by the initial creation of their 3D models, followed by their final version- a more accurate version- and the modeling was achieved in Blender. In Unity, a script containing the movement equations for the car model was assigned to the car object, controlling its movement based on parameters derived from the real race cars. This script was further modified to replicate the behavior of the ESP32-S2 Mini present in the real cars.

The second phase required the creation of an Operations Module, which had a communication protocol to receive real-time coordinates from the real race cars on the physical track and move their virtual counterparts in the digital twin accordingly. This protocol also worked if the real-world components were not present, as was the case in this work for the Real Perception System, for which the Virtual Perception System and the Ghost Cars were created, in order to compensate for its absence. This Operations Module allowed several parameters to be gathered as statistics and displayed on the interface, like the fastest lap time or the maximum speed, and several others. Multiple values like the cars' position, velocity, and direction, gathered from the Virtual Cars, could be stored in the database to be used by the machine learning training algorithms later. During this phase, an algorithm to communicate with the real cars and send them the car commands to control their movement was created.

The third phase involved implementing a framework that allowed for the development and training of machine learning algorithms, specifically imitation and reinforcement learning, within the digital twin.

The digital twin's implementation allowed the digital cars and the real ones to be manually driven, simultaneously, allowing the validation of the movement scripts and facilitating imitation

learning. The machine learning framework enabled imitation and reinforcement learning algorithms to run in parallel, significantly speeding up the learning process. This work surpassed previous investigations primarily due to constraints present in those studies. This project facilitated real-life monitoring unlike the work described in [5]. Additionally, it enabled the observation of multiple vehicles in real-world conditions, a capability that the study by [4] was limited to only one vehicle. Furthermore, unlike either of the previous two works, it gathered statistics and stored data in a database. The statistics and the machine learning framework are examples of how the digital twin can integrate digital services and technologies, allowing it to complement the real-world system.

Even though the digital twin encompasses multiple functions, it also possesses some limitations. The visual and mechanical design of the race car was specifically crafted to perform accurately on a flat surface. Additionally, the race track surface is consistently made from the same material, which results in the car's behavior becoming more monotonous.

In the context of autonomous racing, the development of a digital twin represents a significant advancement in the integration of simulation technologies within this field. By leveraging real-time monitoring and machine learning, this project deepens the understanding of vehicle dynamics and performance. The capacity to analyze multiple vehicles simultaneously generates valuable data that can provide improvements in autonomous driving strategies. Ultimately, this innovation not only enriches the racing experience but also sets the stage for future developments in autonomous technology.

6.1 Future Work

In terms of future work, there are many opportunities to address the limitations of the models developed in this work:

- Enhance the car movement equations to make them more robust and enable accurate performance in various environments beyond flat surfaces;
- Integrate advanced sensors and data collection methods in the race cars to obtain more comprehensive datasets;
- Create a more challenging mini race track with elevation changes, different surface types, and obstacles to simulate realistic environments, offering new challenges and learning opportunities for the cars and their controller algorithms.

However, the main focus for the next step is to adapt the system so that it receives a trajectory, and this trajectory is converted into the car commands that are sent to the car. Fig. 6.1 shows an example of the trajectory that the car would receive. Regarding the Autonomous Racing project, more research and development in this line should be done, such as the development of algorithms for obstacle avoidance and the implementation of vehicle-to-vehicle communication for collaborative racing.

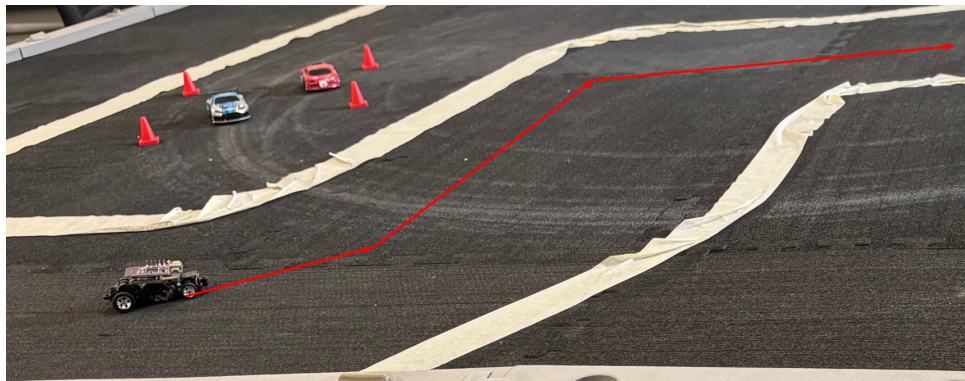


Figure 6.1: Car Trajectory Example

References

- [1] United Nations. The 17 goals. <https://sdgs.un.org/goals>, 2024. Accessed: June 18, 2024.
- [2] Stefan Boschert and Roland Rosen. Digital Twin—The Simulation Aspect. In Peter Hehenberger and David Bradley, editors, *Mechatronic Futures: Challenges and Solutions for Mechatronic Systems and their Designers*, pages 59–74. Springer International Publishing, Cham, 2016.
- [3] Luning Li, Sohaib Aslam, Andrew Wileman, and Suresh Perinpanayagam. Digital twin in aerospace industry: A gentle introduction. *IEEE Access*, 10:9543–9562, 2022.
- [4] Chinmay Vilas Samak, Tanmay Vilas Samak, and Venkat Krovi. A scalable and parallelizable digital twin framework for sustainable sim2real transition of multi-agent reinforcement learning systems, 2024.
- [5] Ziran Wang, Kyungtae Han, and Prashant Tiwari. Digital twin simulation of connected and automated vehicles with the unity game engine. In *2021 IEEE 1st International Conference on Digital Twins and Parallel Intelligence (DTPI)*, pages 1–4, 2021.
- [6] Deuk-Young Jeong, Myung-Sun Baek, Tae-Beom Lim, Yong-Woon Kim, Se-Han Kim, Yong-Tae Lee, Woo-Sug Jung, and In-Bok Lee. Digital twin: Technology evolution stages and implementation layers with technology elements. *IEEE Access*, 10:52609–52620, 2022.
- [7] MathWorks. 3d simulation for vehicle dynamics blockset. <https://www.mathworks.com/help/vdynblks/ug/3d-simulation-for-vehicle-dynamics-blockset.html>, 2025. Accessed: January 24, 2025.
- [8] Unreal Engine. Carla democratizes autonomous vehicle r&d with free open-source simulator. <https://www.unrealengine.com/fr/spotlights/carla-democratizes-autonomous-vehicle-r-d-with-free-open-source-simulator>, 2025. Accessed: January 24, 2025.
- [9] Unity. Simulated environments for autonomous vehicle training. <https://unity.com/how-to/simulated-environments-for-autonomous-vehicle-training>, 2025. Accessed: January 24, 2025.
- [10] ETH Zurich. Autonomous rc car racing. <https://control.ee.ethz.ch/research/team-projects/autonomous-rc-car-racing.html>, n.d. Accessed: January 29, 2025.
- [11] Karl J. Åström and Richard M. Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton University Press, Princeton, 2008. OCLC: ocn183179623.
- [12] GoPro. HERO10 Black, 2025. Accessed: January 30, 2025.

- [13] Leiv Andresen, Adrian Brandemuehl, Alex Honger, Benson Kuan, Niclas Vödisch, Hermann Blum, Victor Reijgwart, Lukas Bernreiter, Lukas Schaupp, Jen Jen Chung, Mathias Burki, Martin R. Oswald, Roland Siegwart, and Abel Gawel. Accurate mapping and planning for autonomous racing. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4743–4749, 2020.
- [14] Zhenghao Hu, Xiuxian Li, Min Meng, and Shiyu Zhao. A game theoretic method for two-team multi-player autonomous racing. *IEEE Robotics and Automation Letters*, 9(9):7581–7588, 2024.
- [15] Gonçalo Faria Ferreira Pimentel da Mota. Autonomous vehicle racing - perception. Master’s thesis, Universidade do Porto, October 2024.
- [16] João Paulo de Jesus Marques Sampaio. Autonomous vehicle racing - learning control. Master’s thesis, Universidade do Porto, October 2024.
- [17] 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. *IEEE Open Journal of Intelligent Transportation Systems*, 3:458–488, 2022.
- [18] ETH Zurich. Development of a Wireless Telemetry Solution for a Formula Student Race Car, 2013. Accessed: January 29, 2025.
- [19] A. Bacha, C. Reinholtz, A. Wicks, M. Fleming, A. Naik, M. Avitabile, and N. Elder. The darpa grand challenge: overview of the virginia tech vehicle and experience. In *Proceedings. The 7th International IEEE Conference on Intelligent Transportation Systems (IEEE Cat. No.04TH8749)*, pages 481–486, 2004.
- [20] R. Behringer, S. Sundareswaran, B. Gregory, R. Elsley, B. Addison, W. Guthmiller, R. Daily, and D. Bevely. The darpa grand challenge - development of an autonomous vehicle. In *IEEE Intelligent Vehicles Symposium, 2004*, pages 226–231, 2004.
- [21] Sebastian Thrun. Winning the darpa grand challenge: A robot race through the mojave desert. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, pages 11–11, 2006.
- [22] Bai Li, Ting Gao, Siji Ma, Youmin Zhang, Tankut Acarman, Kai Cao, Tian’ao Xu, Tantan Zhang, and Fei-Yue Wang. From formula one to autonomous one: History, achievements, and future perspectives. *IEEE Transactions on Intelligent Vehicles*, 8(5):3217–3223, 2023.
- [23] Bai Li, Yile Fang, Tian’ao Xu, Siji Ma, Haorui Wang, Yazhou Wang, Xinyuan Li, Tantan Zhang, Xuepeng Bian, and Fei-Yue Wang. Toward fair and thrilling autonomous racing: Governance rules and performance metrics for the autonomous one. *IEEE Transactions on Intelligent Vehicles*, 8(8):3974–3982, 2023.
- [24] F1tenth autonomous racing. <https://www.f1tenth.racing/>. Accessed: January 24, 2025.
- [25] Autodrive f1tenth sim racing @ cdc 2024. <https://autodrive-ecosystem.github.io/competitions/f1tenth-sim-racing-cdc-2024/#about>. Accessed: January 24, 2025.

- [26] Gaëlic Béchu, Antoine Beugnard, Caroline G. L. Cao, Quentin Perez, Christelle Urtado, and Sylvain Vauttier. A software engineering point of view on digital twin architecture. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2022.
- [27] Yunda Wang, Gang Zhang, Ruijun Chen, Zhigang Liu, and Ruichang Qiu. Analysis of digital twin application of urban rail power supply system for energy saving. In *2021 IEEE 1st International Conference on Digital Twins and Parallel Intelligence (DTPI)*, pages 29–32, 2021.
- [28] Yue Zhou, Pengfei Su, Jianzhong Wu, Wenqiang Sun, Xiandong Xu, and Muditha Abeysekera. Digital twins for flexibility service provision from industrial energy systems. In *2021 IEEE 1st International Conference on Digital Twins and Parallel Intelligence (DTPI)*, pages 274–277, 2021.
- [29] Lukas Esterle, Henrik Ejersbo, Mirgita Frasheri, Cláudio Gomes, Hugo Daniel Macedo, and Peter Gorm Larsen. Digital twins for autonomous intelligent systems: From development to deployment. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 53–54, 2022.
- [30] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.
- [31] Luís Tiago Paiva and Fernando A.C.C. Fontes. Time-mesh Refinement in Optimal Control Problems for Nonholonomic Vehicles. *Procedia Technology*, 17:178–185, 2014.
- [32] Blender Foundation. Blender, 2024.
- [33] Unity Technologies. Quaternion - unity script reference, 2024. Accessed: December 23, 2024.
- [34] Richard Zurawski, editor. *Industrial Communication Technology Handbook*. CRC Press, 2nd edition, 2015.
- [35] Inc. Krueger Systems. Sqlite4unity3d. GitHub repository, 2012. Available at: <https://github.com/roboryantron/SQLite4Unity3d>.
- [36] SQLite Database Browser. Db browser for sqlite, n.d. Accessed: October 20, 2024.
- [37] Unity Technologies. Unity ml-agents. <https://github.com/Unity-Technologies/ml-agents>, 2024. Accessed: March 28, 2024.
- [38] Unity Technologies. ML-agents: Installation. <https://unity-technologies.github.io/ml-agents/Installation/>, 2023. Accessed: March 28, 2024.
- [39] Unity Technologies. ML-agents: Making a new learning environment. <https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/>, 2023. Accessed: March 28, 2024.

Appendix A

Scripts

In this appendix, several scripts used in this work are displayed and explained in full. Due to them being used in Unity, all the scripts are made in C#.

A.1 Refined Car Movement Script

This script was used to test the car's movement, ensuring its model was working correctly. It received movement instructions through a UDP protocol that allowed it to move forward or backward, increase or decrease the duration of its movement, steer to the left or right, reset its steering, and send a message to the terminal to see if it was responding.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using System.Net;
4 using System.Net.Sockets;
5 using System.Text;
6 using UnityEngine;
7
8 public class UDPCarMovement : MonoBehaviour
9 {
10     public float v = 0f; // Forward velocity of the car
11     public float b = 6f; // Distance between front and rear axles
12     public float a = 3f; // Distance between front axle and car's center of mass
13     public float maxdegrees = 13.436f; // steering limit
14     public float decelerationRate = 7f; // strenght of car deceleration
15
16     private float delta; // Steering angle
17     private float steeringAngle = 0f; // Current steering angle
18     private readonly float[] steeringValues = { -13.436f, -8.9573f, -4.4787f, 0f,
19         4.4787f, 8.9573f, 13.436f }; // Combined steering angles in degrees
20     private int currentSteeringIndex = 3; // starts on default steering, 0
21     float maxSpeed = 91.068f;
22     public float centreOfGravityOffset = -0.7f;
```

```

22     public float acceleration = 273.20f;
23     private float direction = 1f;
24
25     private UdpClient udpClient;
26     private IPEndPoint remoteEndPoint;
27     private string receivedCommand;
28
29     private float commandDuration = 0f; // Duration to hold the command
30     private float commandDurationTimer = 0f; // Timer for the command duration
31     private float commandDelay = 0.1f;
32     private bool commandActive = false; // Indicates if a command is currently
        active
33
34     private Rigidbody rb; // Reference to the Rigidbody component
35     private GameObject frontaxis, frontleft, frontright;
36     private Quaternion initialFrontAxisRotation, initialFrontLeftWheelRotation,
        initialFrontRightWheelRotation;
37
38     public int port;
39     void Start()
40     {
41         // Get the reference to the Rigidbody component
42         rb = GetComponent<Rigidbody>();
43         rb.centerOfMass += Vector3.up * centreOfGravityOffset;
44
45         // Setup UDP listener
46         udpClient = new UdpClient(port); // Listen on port 12345
47         remoteEndPoint = new IPEndPoint(IPAddress.Any, 0);
48         Debug.Log("UDP Listener started on port " + port);
49
50         frontaxis = transform.Find("front axis").gameObject;
51         frontleft = transform.Find("front left").gameObject;
52         frontright = transform.Find("front right").gameObject;
53         if (frontaxis == null || frontleft == null || frontright == null)
54         {
55             Debug.LogError("Child object not found");
56         }
57         else
58         {
59             // Set the default rotation to 90 degrees to the left
60             initialFrontAxisRotation = frontaxis.transform.localRotation;
61             initialFrontLeftWheelRotation = frontleft.transform.localRotation;
62             initialFrontRightWheelRotation = frontright.transform.localRotation
                ;
63         }
64
65         // Start the UDP listening coroutine
66         StartCoroutine(ListenForCommands());
67     }

```



```

68
69 void FixedUpdate()
70 {
71     // Check if there is a movement command active where the car cannot receive
        new commands
72     if (commandActive)
73     {
74         commandDurationTimer -= Time.fixedDeltaTime;
75         if (commandDurationTimer <= 0f)
76         {
77             v = 0f; // Reset velocity after the command duration
78             commandActive = false; // Deactivate flag
79         }
80     }
81
82     // Calculate the steering angle based on the current steering index
83     delta = (steeringValues[currentSteeringIndex]) * Mathf.Deg2Rad;
84     delta = Mathf.Clamp(delta, -maxdegrees * Mathf.Deg2Rad, maxdegrees * Mathf.
        Deg2Rad);
85
86     // prevents the car from moving to the wrong side when going backward
87     if (v > 0)
88         direction = 1;
89     else if (v < 0)
90         direction = -1;
91
92     // Velocity calculations
93     float alfa = Mathf.Atan(a * Mathf.Tan(delta) / b); // Steering angle of the
        front wheels
94     float theta = transform.eulerAngles.y * Mathf.Deg2Rad; // Convert car's
        rotation to radians
95     float z_dot = v * Mathf.Cos(alfa + theta); // z velocity
96     float x_dot = v * Mathf.Sin(alfa + theta); // x velocity
97     float theta_dot = rb.velocity.magnitude * direction / b * Mathf.Tan(delta);
        // Angular velocity
98
99     // Calculate the movement direction based on the forward direction
100     Vector3 movementDirection = Vector3.ClampMagnitude(new Vector3(-x_dot, 0f,
        -z_dot), 1f);
101     movementDirection = Vector3.ProjectOnPlane(movementDirection, Vector3.up).
        normalized;
102
103     // Calculate the force to apply the movement
104     Vector3 force = Vector3.Dot(movementDirection, transform.forward) *
        transform.forward * acceleration * Time.deltaTime;
105
106     // Apply the force to the Rigidbody
107     rb.AddForce(force, ForceMode.VelocityChange);
108

```

```

109      // Friction when there is no movement command active
110      if (v == 0)
111      {
112          Vector3 velocity = rb.velocity;
113          Vector3 deceleration = -velocity * decelerationRate;
114          rb.AddForce(deceleration, ForceMode.Acceleration);
115      }
116
117      // Apply damping to reduce sideways drifting
118      float slippingFactor = 0.3f;
119      Vector3 relativeVelocity = transform.InverseTransformDirection(rb.velocity)
120      ;
121      relativeVelocity.x *= slippingFactor;
122      rb.velocity = transform.TransformDirection(relativeVelocity);
123
124      // Limit the speed
125      rb.velocity = Vector3.ClampMagnitude(rb.velocity, maxSpeed - 5);
126
127      // rotate the car
128      Quaternion deltaRotation = Quaternion.Euler(0f, theta_dot * Mathf.Rad2Deg *
129          Time.deltaTime, 0f);
130      rb.MoveRotation(rb.rotation * deltaRotation);
131
132      // Calculate the steering angle for the front wheels rotation
133      steeringAngle = steeringValues[currentSteeringIndex];
134      delta = (steeringAngle) * Mathf.Deg2Rad;
135      delta = Mathf.Clamp(delta, -maxdegrees * Mathf.Deg2Rad, maxdegrees * Mathf.
136          Deg2Rad);
137      float targetRotationAngle = Mathf.Clamp(steeringValues[currentSteeringIndex
138          ], -maxdegrees, maxdegrees);
139
140      // Apply the steering rotation to each front wheel and the front axis
141      if (frontleft != null && frontright != null && frontaxis != null)
142      {
143          frontaxis.transform.localRotation = initialFrontAxisRotation *
144          Quaternion.Euler(targetRotationAngle, 0f, 0f);
145          frontleft.transform.localRotation = initialFrontLeftWheelRotation *
146          Quaternion.Euler(targetRotationAngle, 0f, 0f);
147          frontright.transform.localRotation = initialFrontRightWheelRotation *
148          Quaternion.Euler(targetRotationAngle, 0f, 0f);
149      }
150
151      // set up UDP listener
152      private IEnumerator ListenForCommands()
153      {
154          while (true)
155          {
156              if (udpClient.Available > 0)

```

```
151         {
152             byte[] data = udpClient.Receive(ref remoteEndPoint);
153             receivedCommand = Encoding.ASCII.GetString(data);
154             ExecuteCommand(receivedCommand);
155         }
156         yield return null; // Wait for the next frame
157     }
158 }
159
160 // executes the instruction received
161 private void ExecuteCommand(string command)
162 {
163     if (!commandActive) {
164         // Execute the command and refresh it immediately
165         if (command == "mf")
166         {
167             SetVelocity(1f, commandDelay); // Move forward
168         }
169         else if (command == "mb")
170         {
171             SetVelocity(-1f, commandDelay); // Move backward
172         }
173         else if (command == "ml")
174         {
175             // Change steering to left
176             currentSteeringIndex = Math.Min(currentSteeringIndex, 3);
177             currentSteeringIndex = Mathf.Clamp(currentSteeringIndex - 1, 0,
178                 steeringValues.Length - 1);
179             steeringAngle = steeringValues[currentSteeringIndex]; // Update steering angle
180             SetVelocity(0f, 0);
181         }
182         else if (command == "mr")
183         {
184             // Change steering to right
185             currentSteeringIndex = Math.Max(currentSteeringIndex, 3);
186             currentSteeringIndex = Mathf.Clamp(currentSteeringIndex + 1, 0,
187                 steeringValues.Length - 1);
188             steeringAngle = steeringValues[currentSteeringIndex]; // Update steering angle
189             SetVelocity(0f, 0);
190         }
191         else if (command == "rr")
192         {
193             // Reset steering to default position
194             currentSteeringIndex = 3;
195             steeringAngle = 0f; // Reset steering angle to center
196             SetVelocity(0f, 0);
197         }
198     }
```

```

196         else if (command == "+d")
197         {
198             // Increase command delay
199             commandDelay += 0.1f;
200             Debug.Log($"Command delay increased to: {commandDelay} seconds");
201             SetVelocity(0f, 0f);
202         }
203         else if (command == "-d")
204         {
205             // Decrease command delay. It can't drop below 0 seconds
206             commandDelay = Mathf.Max(0.1f, commandDelay - 0.1f);
207             Debug.Log($"Command delay decreased to: {commandDelay} seconds");
208             SetVelocity(0f, 0f);
209         }
210     }
211 }
212
213 private void SetVelocity(float targetVelocity, float duration)
214 {
215     v = targetVelocity; // Set the desired velocity
216     commandDuration = duration; // Set the duration of the command
217     commandDurationTimer = duration; // Reset the command timer
218     commandActive = true; // Mark command as active
219 }
220
221 // close UDP listener
222 void OnApplicationQuit()
223 {
224     if (udpClient != null)
225     {
226         udpClient.Close();
227         Debug.Log("UDP Listener closed.");
228     }
229 }
230 }

```

Listing A.1: Refined Car Movement Script

A.2 Car Monitoring Script

The monitoring script aimed to receive the position and direction of either the Real Cars or the Ghost Cars and position the Virtual Cars in Unity in real time. It also verified if the cars were in a safe position, and restored them to the last safe position if needed. Finally, it inserted several values into the database.

```

1 using UnityEngine;

```

```
2 using System.Net;
3 using System.Net.Sockets;
4 using System.Text;
5 using System.Threading;
6 using System.Globalization;
7
8 public class VehicleManager : MonoBehaviour
9 {
10     // References to the yellow, blue, and purple cars
11     public GameObject yellow_car;
12     public GameObject blue_car;
13     public GameObject purple_car;
14
15     public GameObject ghost_yellow_car;
16     public GameObject ghost_blue_car;
17     public GameObject ghost_purple_car;
18
19     // Variables to store the position and direction of each car
20     public Vector3 yellowCarPosition;
21     public Vector3 blueCarPosition;
22     public Vector3 purpleCarPosition;
23     public float yellowdirection;
24     public float bluedirection;
25     public float purpledirection;
26
27     // Variables to store the last safe positions and directions of the cars
28     private Vector3 yellowCarSafePosition;
29     private Vector3 blueCarSafePosition;
30     private Vector3 purpleCarSafePosition;
31     private Vector3 yellowlastsafeposition;
32     private Vector3 bluelastsafeposition;
33     private Vector3 purplelastsafeposition;
34     private float yellowCarSafeDirection;
35     private float blueCarSafeDirection;
36     private float purpleCarSafeDirection;
37
38     private UdpClient udpClient;
39     private Thread udpThread;
40     private bool isRunning = true;
41     public int port = 8085;
42
43     private int auxyellow, auxpurple, auxblue = 0;
44
45     // race track limits
46     private float minX = -200f;
47     private float maxX = 5f;
48     private float minZ = -235f;
49     private float maxZ = 5f;
50 }
```

```

51 // filter to eliminate noise on sideways car movement
52 public float xAxisMovementThreshold = 15f;
53
54 private bool reset_position_blue, reset_position_yellow, reset_position_purple;
55
56 // flags to indicate new data received
57 private volatile bool yellowDataReceived = false;
58 private volatile bool blueDataReceived = false;
59 private volatile bool purpleDataReceived = false;
60
61 public CarDataDatabase carDataDatabase;
62
63 void Start()
64 {
65     // start UDP listener
66     udpClient = new UdpClient(port);
67     udpThread = new Thread(new ThreadStart(ReceiveData));
68     udpThread.IsBackground = true;
69     udpThread.Start();
70
71     // Set initial positions
72     SetInitialPositions();
73
74     carDataDatabase = FindObjectOfType<CarDataDatabase>();
75 }
76
77 // close UDP listener
78 void OnDestroy()
79 {
80     isRunning = false;
81     udpClient.Close();
82     udpThread.Join();
83 }
84
85 void FixedUpdate()
86 {
87     // Check if new data is available and update car positions if the position
88     // is safe
89     UpdateCarPositions();
90
91     // Check if cars are in a safe position
92     CheckCarSafety();
93     SaveCarDataToDatabase();
94
95     // bypass the unsafe trigger and place the Virtual Cars on the Ghost Car's
96     // position if the new position is within the race track's limits
97     if (Input.GetKey(KeyCode.O))
98     {
99         reset_position_blue = true;

```

```
198         reset_position_yellow = true;
199         reset_position_purple = true;
200     }
201 }
202
203 private void SetInitialPositions()
204 {
205     yellowCarPosition = new Vector3(-30f, 0.1f, -180f);
206     blueCarPosition = new Vector3(-20f, 0.1f, -180f);
207     purpleCarPosition = new Vector3(-10f, 0.1f, -180f);
208
209     yellow_car.transform.position = yellowCarPosition;
210     blue_car.transform.position = blueCarPosition;
211     purple_car.transform.position = purpleCarPosition;
212
213     // Initialize safe positions and directions
214     yellowCarSafePosition = yellowCarPosition;
215     blueCarSafePosition = blueCarPosition;
216     purpleCarSafePosition = purpleCarPosition;
217     yellowCarSafeDirection = 0f;
218     blueCarSafeDirection = 0f;
219     purpleCarSafeDirection = 0f;
220
221     // force the Virtual Cars to overlap the Ghost Cars when the code is
222     // initiated
223     reset_position_blue = true;
224     reset_position_yellow = true;
225     reset_position_purple = true;
226 }
227
228 private void ReceiveData()
229 {
230     IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Any, 0);
231     while (isRunning)
232     {
233         try
234         {
235             byte[] data = udpClient.Receive(ref remoteEndPoint);
236             string message = Encoding.UTF8.GetString(data);
237             ParseMessage(message);
238         }
239         catch (System.Exception ex)
240         {
241             Debug.Log("Error receiving data: " + ex.Message);
242         }
243     }
244 }
```

```

145     // extracts data received from the string and updates the car position and
146     // direction's values
147     private void ParseMessage(string message)
148     {
149         string[] parts = message.Split(',');
150
151         // Validate the message format
152         if (parts.Length == 5)
153         {
154             string carId = parts[0];
155             if (float.TryParse(parts[1], NumberStyles.Float, CultureInfo.
156                 InvariantCulture, out float x) &&
157                 float.TryParse(parts[2], NumberStyles.Float, CultureInfo.
158                     InvariantCulture, out float y) &&
159                 float.TryParse(parts[3], NumberStyles.Float, CultureInfo.
160                     InvariantCulture, out float z) &&
161                 float.TryParse(parts[4], NumberStyles.Float, CultureInfo.
162                     InvariantCulture, out float direction))
163             {
164                 switch (carId)
165                 {
166                     case "yellow":
167                         yellowCarPosition = new Vector3(x, y, z);
168                         yellowdirection = direction;
169                         yellowDataReceived = true; // Set flag for yellow car data
170                         break;
171                     case "blue":
172                         blueCarPosition = new Vector3(x, y, z);
173                         bluedirection = direction;
174                         blueDataReceived = true; // Set flag for blue car data
175                         break;
176                     case "purple":
177                         purpleCarPosition = new Vector3(x, y, z);
178                         purpLEDirection = direction;
179                         purpleDataReceived = true; // Set flag for purple car data
180                         break;
181                     default:
182                         Debug.Log("Unknown car ID: " + carId);
183                         break;
184                 }
185             }
186             else
187             {
188                 Debug.LogError("Failed to parse position or direction as float.");
189             }
190         }
191         else
192         {
193             Debug.LogError("Invalid message format: " + message);
194         }
195     }
196 }

```



```

189     }
190 }
191
192 // updates the positions of the virtual cars
193 private void UpdateCarPositions()
194 {
195     if (yellow_car != null && yellowDataReceived && IsCarInSafePosition(
196         yellow_car, yellowCarPosition, yellowdirection, yellowCarSafePosition,
197         reset_position_yellow))
198     {
199         yellow_car.transform.position = yellowCarPosition;
200         yellow_car.transform.rotation = Quaternion.Euler(0f, yellowdirection, 0
201             f);
202
203         // verifies if the received values are within the race track's outer
204         limits
205         if((yellowCarPosition.x > minX) && (yellowCarPosition.x < maxX) && (
206             yellowCarPosition.z > minZ) && (yellowCarPosition.z < maxZ))
207         {
208             reset_position_yellow = false;
209             yellowlastsafeposition = yellowCarSafePosition;
210             yellowCarSafePosition = yellowCarPosition; // Update safe position
211                 if successful
212             yellowCarSafeDirection = yellowdirection; // Update safe direction
213                 if successful
214         }
215     }
216     else
217     {
218         yellowDataReceived = false; // Reset flag if position update was
219             unsuccessful
220     }
221
222     if (blue_car != null && blueDataReceived && IsCarInSafePosition(blue_car,
223         blueCarPosition, bluedirection, blueCarSafePosition,
224         reset_position_blue))
225     {
226         blue_car.transform.position = blueCarPosition;
227         blue_car.transform.rotation = Quaternion.Euler(0f, bluedirection, 0f);
228         if(blueCarPosition.x > minX && blueCarPosition.x < maxX &&
229             blueCarPosition.z > minZ && blueCarPosition.z < maxZ)
230         {
231             reset_position_blue = false;
232             bluelastsafeposition = blueCarSafePosition;
233             blueCarSafePosition = blueCarPosition;
234             blueCarSafeDirection = bluedirection;
235         }
236     }
237     else

```

```
227     {
228         blueDataReceived = false;
229     }
230
231     if (purple_car != null && purpleDataReceived && IsCarInSafePosition(
232         purple_car, purpleCarPosition, purpLEDirection, purpleCarSafePosition,
233         reset_position_purple))
234     {
235         purple_car.transform.position = purpleCarPosition;
236         purple_car.transform.rotation = Quaternion.Euler(0f, purpLEDirection, 0
237             f);
238         if (purpleCarPosition.x > minX && purpleCarPosition.x < maxX &&
239             purpleCarPosition.z > minZ && purpleCarPosition.z < maxZ)
240         {
241             reset_position_purple = false;
242             purpleLastSafePosition = purpleCarSafePosition;
243             purpleCarSafePosition = purpleCarPosition;
244             purpleCarSafeDirection = purpLEDirection;
245         }
246     }
247     else
248     {
249         purpleDataReceived = false;
250     }
251 }
252
253 private void CheckCarSafety()
254 {
255     if (!IsCarInSafePosition(yellow_car, yellowCarPosition, yellowDirection,
256         yellowCarSafePosition, false))
257     {
258         Debug.Log("Yellow car is not in a safe position. Reverting to last safe
259             position.");
260         RestoreCarToSafePosition(yellow_car, yellowCarSafePosition,
261             yellowCarSafeDirection);
262     }
263
264     if (!IsCarInSafePosition(blue_car, blueCarPosition, blueDirection,
265         blueCarSafePosition, false))
266     {
267         Debug.Log("Blue car is not in a safe position. Reverting to last safe
268             position.");
269         RestoreCarToSafePosition(blue_car, blueCarSafePosition,
270             blueCarSafeDirection);
271     }
272
273     if (!IsCarInSafePosition(purple_car, purpleCarPosition, purpLEDirection,
274         purpleCarSafePosition, false))
275     {
276         Debug.Log("Purple car is not in a safe position. Reverting to last safe
277             position.");
278         RestoreCarToSafePosition(purple_car, purpleCarSafePosition,
279             purpleCarSafeDirection);
280     }
281 }
```

```
265         Debug.Log("Purple car is not in a safe position. Reverting to last safe  
266             position.");  
267         RestoreCarToSafePosition(purple_car, purpleCarSafePosition,  
268             purpleCarSafeDirection);  
269     }  
270 }  
271  
272 private void RestoreCarToSafePosition(GameObject car, Vector3 safePosition,  
273     float safeDirection)  
274 {  
275     if (car != null)  
276     {  
277         car.transform.position = safePosition; // Restore position  
278         car.transform.rotation = Quaternion.Euler(0f, safeDirection, 0f); //  
279             Restore direction  
280     }  
281 }  
282  
283 private bool IsCarInSafePosition(GameObject car, Vector3 currentPosition, float  
284     currentDirection, Vector3 lastSafePosition, bool reset)  
285 {  
286     return (IsCarOnTrack(car) && (!HasCarBeenGrabbed(car, currentPosition,  
287         lastSafePosition) || reset));  
288 }  
289  
290 // Check if the car is on the track surface by emitting a ray downwards to  
291 detect the surface  
292 private bool IsCarOnTrack(GameObject car)  
293 {  
294     Vector3 rayOrigin = car.transform.position + Vector3.up * 0.5f; // 0.5  
295         units above the car  
296     Ray ray = new Ray(rayOrigin, Vector3.down);  
297     RaycastHit hit;  
298     int trackLayerMask = LayerMask.GetMask("TrackSurface");  
299  
300     if (Physics.Raycast(ray, out hit, Mathf.Infinity, trackLayerMask))  
301     {  
302         if (hit.collider.CompareTag("Track"))  
303         {  
304             return true;  
305         }  
306     }  
307     Debug.Log($"{car.name} is off the track.");  
308     return false;  
309 }  
310  
311 // verifies if the car had an abrupt sideways movement
```

```

305     private bool HasCarBeenGrabbed(GameObject car, Vector3 currentPosition, Vector3
        lastSafePosition)
306     {
307         // Convert the current and last safe positions to the car's local space
308         Vector3 localCurrentPosition = car.transform.InverseTransformPoint(
            currentPosition);
309         Vector3 localLastSafePosition = car.transform.InverseTransformPoint(
            lastSafePosition);
310
311         // Check movement along the local X-axis
312         float xAxisMovement = Mathf.Abs(localCurrentPosition.x -
            localLastSafePosition.x);
313         return xAxisMovement > xAxisMovementThreshold;
314     }
315
316     // saves the timestamp, coordinates, velocity, and direction of the cars to the
        database
317     private void SaveCarDataToDatabase()
318     {
319         float deltaTime = Time.fixedDeltaTime; // Time between FixedUpdate calls
320         float fluctuationThreshold = 0.05f; // Threshold to filter fluctuations on
            the values
321
322         // Yellow Car
323         Vector3 yellowPositionDelta = yellowCarSafePosition -
            yellowlastsafeposition;
324         Vector3 yellowLocalvelocity = yellow_car.transform.
            InverseTransformDirection(yellowPositionDelta / deltaTime);
325
326         // apply fluctuation threshold and max velocity limit
327         yellowLocalvelocity.x = Mathf.Abs(yellowLocalvelocity.x) <
            fluctuationThreshold ? 0 : yellowLocalvelocity.x;
328         yellowLocalvelocity.z = Mathf.Abs(yellowLocalvelocity.z) <
            fluctuationThreshold ? 0 : yellowLocalvelocity.z;
329         if (Mathf.Abs(yellowLocalvelocity.x) > 200 || Mathf.Abs(yellowLocalvelocity
            .z) > 200)
330         {
331             // Set velocity to (0, 0, 0) if the velocity value was highly above the
                limit to filter errors
332             yellowLocalvelocity = Vector3.zero;
333         }
334         Vector3 yellowCarvelocity = new Vector3(yellowLocalvelocity.x, 0, -
            yellowLocalvelocity.z);
335         carDataDatabase.SaveCarData("Yellow Car", -yellowCarSafePosition,
            yellowCarvelocity, yellowdirection);
336
337         Vector3 bluePositionDelta = blueCarSafePosition - bluelastsafeposition;
338         Vector3 blueLocalvelocity = blue_car.transform.InverseTransformDirection(
            bluePositionDelta / deltaTime);

```

```

339
340     // Blue Car
341     blueLocalvelocity.x = Mathf.Abs(blueLocalvelocity.x) < fluctuationThreshold
        ? 0 : blueLocalvelocity.x;
342     blueLocalvelocity.z = Mathf.Abs(blueLocalvelocity.z) < fluctuationThreshold
        ? 0 : blueLocalvelocity.z;
343     if (Mathf.Abs(blueLocalvelocity.x) > 200 || Mathf.Abs(blueLocalvelocity.z)
        > 200)
344     {
345         blueLocalvelocity = Vector3.zero;
346     }
347     Vector3 blueCarvelocity = new Vector3(blueLocalvelocity.x, 0, -
        blueLocalvelocity.z);
348     carDataDatabase.SaveCarData("Blue Car", -blueCarSafePosition,
        blueCarvelocity, bluedirection);
349
350     // Purple Car
351     Vector3 purplePositionDelta = purpleCarSafePosition -
        purplelastsafeposition;
352     Vector3 purpleLocalvelocity = purple_car.transform.
        InverseTransformDirection(purplePositionDelta / deltaTime);
353
354     purpleLocalvelocity.x = Mathf.Abs(purpleLocalvelocity.x) <
        fluctuationThreshold ? 0 : purpleLocalvelocity.x;
355     purpleLocalvelocity.z = Mathf.Abs(purpleLocalvelocity.z) <
        fluctuationThreshold ? 0 : purpleLocalvelocity.z;
356     if (Mathf.Abs(purpleLocalvelocity.x) > 200 || Mathf.Abs(purpleLocalvelocity
        .z) > 200)
357     {
358         purpleLocalvelocity = Vector3.zero; // Set velocity to (0, 0, 0)
359     }
360     Vector3 purpleCarvelocity = new Vector3(purpleLocalvelocity.x, 0, -
        purpleLocalvelocity.z);
361     carDataDatabase.SaveCarData("Purple Car", -purpleCarSafePosition,
        purpleCarvelocity, purpLEDirection);
362 }
363
364 }

```

Listing A.2: Car Monitoring Script

A.3 Car Statistics Script

The statistics script calculated the average speed, maximum speed, distance traveled, current lap time, fastest lap time, laps completed, and the idle time of each car.

```

1 using UnityEngine;

```

```

2  using System.Collections;
3
4  public class CarStatistics : MonoBehaviour
5  {
6      // Variables for tracking the car's statistics
7      private Vector3 lastPosition;
8      private float distanceTraveled = 0f;
9      private float speed = 0f;
10     private float maxSpeed = 0f;
11     private float idleTime = 0f; // time spent not moving
12     private float averageSpeed = 0f;
13     private int speedCount = 0; // Number of speed calculations
14     private float updateInterval = 0.1f; // Update stats every 0.1 seconds
15     private float timeSinceLastUpdate = 0f;
16
17     // Lap tracking variables
18     private bool isOnFinishLine = false;
19     private bool isLapActive = false;
20     private float lapTime = 0f;
21     private int lapsCompleted = 0;
22     private bool hasExitedFinishLine = false;
23     private float currentTime = 0f;
24
25     private bool aux = false;
26     private Transform carTransform;
27     private int checkpointindex = 0;
28     private bool IsPositionFirst = false;
29
30     private bool canUpdate = false; // Flag to start Update logic
31     private float delay = 3f; // Delay in seconds (gives time for the cars' initial
        // positions to be set)
32
33     void Start()
34     {
35         carTransform = this.transform;
36         // Initialize the car's last position to the current position at start
37         lastPosition = transform.position;
38         StartCoroutine(StartUpdateAfterDelay());
39     }
40
41     void Update()
42     {
43         // Track time between updates
44         timeSinceLastUpdate += Time.deltaTime;
45
46         // Only update every 'updateInterval' seconds
47         if (timeSinceLastUpdate >= updateInterval)
48         {
49             CalculateStatistics();

```

```
50         timeSinceLastUpdate = 0f; // Reset the timer
51     }
52     // If the lap is active, increment the lap time
53     if (isLapActive)
54     {
55         currentTime += Time.deltaTime;
56     }
57 }
58
59 void CalculateStatistics()
60 {
61     // Calculate the distance traveled since the last update
62     Vector3 currentPosition = transform.position;
63     if (IsPositionFirst && canUpdate) {
64         float distanceThisFrame = Vector3.Distance(lastPosition, currentPosition);
65
66         // Accumulate the total distance traveled
67         distanceTraveled += distanceThisFrame;
68
69         // Calculate current speed (distance/time), in meters per second
70         speed = distanceThisFrame / updateInterval;
71
72         // Update the max speed
73         if (speed > maxSpeed)
74         {
75             maxSpeed = speed;
76         }
77
78         // Calculate the average speed using a running average formula
79         if (speed > 0.1f) // Only include non-idle speeds
80         {
81             // Running average formula
82             averageSpeed = ((averageSpeed * speedCount) + speed) / (speedCount + 1)
83             ;
84             speedCount++; // Increment the count of speed updates
85         }
86
87         // Check if the car is idle (speed is near 0)
88         if (speed < 0.1f)
89         {
90             idleTime += updateInterval;
91         }
92
93         // Store the current position for the next calculation
94     }
95     lastPosition = currentPosition;
96     IsPositionFirst = true;
97 }
```

```

98      // used for when the car collides with the hitbox of the checkpoints or the
99      // finish line
100  private void OnTriggerEnter(Collider other)
101  {
102      if (other.CompareTag("FinishLine") && (!isLapActive || checkpointindex==5))
103      {
104          isOnFinishLine = true; // Update the status
105
106          if(!aux)
107              StartLap();
108
109          if (other.CompareTag("FinishLine"))
110          {
111              // Car has exited the finish line
112              isOnFinishLine = false; // Update the status
113              hasExitedFinishLine = true; // Mark that the car has exited the
114              // finish line
115
116              if (isLapActive)
117              {
118                  CompleteLap(); // Complete the lap when exiting the finish line
119                  checkpointindex = 0;
120              }
121          }
122
123          // updates whenever the car passes through the next checkpoint
124          else if(other.CompareTag("Checkpoint") && checkpointindex==0)
125              checkpointindex = 1;
126
127          else if(other.CompareTag("Checkpoint2") && checkpointindex==1)
128              checkpointindex = 2;
129
130          else if(other.CompareTag("Checkpoint3") && checkpointindex==2)
131              checkpointindex = 3;
132
133          else if(other.CompareTag("Checkpoint4") && checkpointindex==3)
134              checkpointindex = 4;
135
136          else if(other.CompareTag("Checkpoint5") && checkpointindex==4)
137              checkpointindex = 5;
138      }
139
140  private void StartLap()
141  {
142      isLapActive = true; // Set lap to active
143      aux = true;
144      if(checkpointindex==5){

```



```
145         lapsCompleted++; // Increment laps completed
146     }
147 }
148
149 public void CompleteLap()
150 {
151     if (aux) {
152         if (lapsCompleted > 0) {
153
154             if ((lapTime == 0 || (currentTime < lapTime)) && checkpointindex
155                 == 5)
156                 lapTime = currentTime;
157         }
158         Debug.Log($"Lap {lapsCompleted} completed in {lapTime:F2} seconds");
159     }
160     currentTime = 0;
161     aux = false;
162 }
163 // Public methods to retrieve statistics
164 public float GetDistanceTraveled()
165 {
166     return distanceTraveled;
167 }
168
169 public float GetSpeed()
170 {
171     return speed;
172 }
173
174 public float GetMaxSpeed()
175 {
176     return maxSpeed;
177 }
178
179 public float GetIdleTime()
180 {
181     return idleTime;
182 }
183
184 public float GetAverageSpeed()
185 {
186     return averageSpeed;
187 }
188
189 public int GetLapsCompleted()
190 {
191     return lapsCompleted; // Return the number of completed laps
192 }
```

```
193
194     public float GetCurrentLapTime()
195     {
196         return currentTime; // Return the current lap time
197     }
198
199     public float GetMinLapTime()
200     {
201         return lapTime;
202     }
203
204     private IEnumerator StartUpdateAfterDelay()
205     {
206         yield return new WaitForSeconds(delay);
207         canUpdate = true; // Enable Update logic after the delay
208     }
209
210 }
```

Listing A.3: Car Statistics Script