POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

TracksCAD - Computer Assisted Track Design Tool for Racing Games

Supervisor: Prof. Daniele Loiacono
Co-supervisor: Prof. Pier Luca Lanzi

M.Sc. Thesis by:
Davide Pons, 939643

Academic Year 2020-2021

Abstract

With the increasing costs of modern videogames production, developers are in continous search of new ways to reduce the effort taken to create them, without having to sacrify their overall quality. One of the aspects that has a major impact on such costs is the creation of game content which, in the case of racing games, largely consists in designing the greatest number of unique circuits on which to compete. The goal of this thesis is to develop a computer assisted design (CAD) tool that helps in the process of designing tracks for racing videogames, and that is capable of providing suggestions and feedback related to both the driving alone experience and the dynamics that show up when racing in group. The way the tool comes up with such suggestions is based on the analysis of a set of simulations that are performed whenever the designer needs them. Additionally, we discuss how Machine Learning techniques can be exploited to estimate some of the metrics that can usually be extracted only from simulations, using as inputs just the topological features of the track being created.

Sommario

Con il continuo aumento dei costi di produzione dei videogiochi moderni, gli sviluppatori sono alla costante ricerca di nuove metodologie per ridurre gli sforzi necessari al loro sviluppo, senza però essere costretti a sacrificarne la qualità complessiva. Uno degli aspetti che impatta maggiormente su questi costi è la creazione dei contenuti di gioco che, nel caso dei videogiochi di guida, consiste in gran parte nella progettazione di tracciati unici sui quali competere. L'obbiettivo di questa tesi è quello di sviluppare uno strumento di progettazione assistita al computer (CAD) che aiuti nel processo di creazione dei tracciati di videogiochi di guida, e che sia in grado di fornire suggerimenti e feedback riguardanti sia l'esperienza di guida in solitaria che di tutte quelle dinamiche che vengono a crearsi nelle competizioni di gruppo. Il metodo con cui lo strumento genera questi consigli è basandosi sull'analisi di simulazioni, che vengono eseguite ogniqualvolta il designer ne avesse bisogno. Inoltre, si discute di come tecniche di Machine Learning possono essere sfruttate per stimare alcune delle metriche normalmente estratte dalle simulazioni, usando come soli input le caratteristiche topologiche della pista che si sta creando.

Acknowledgements

I would like to thank all those who inspired and supported me during my academic path. My family, who have always been present in both difficult and joyful moments. All friends and colleagues who encouraged and accompanied me. A special mention goes also to my supervisor and co-supervisor, Daniele Loiacono and Pier Luca Lanzi, who guided me throughout this project with patience and professionalism, and gave me the chance to work on a thesis concerning a field I feel extremely passionate about.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Context: Track design in Racing Games

One of the major aspects that determines the quality and fun factor of racing games is the quality and quantity of their tracks. Since the release of the first videogames belonging to this genre, developers had to carefully think about the way obstacles were laid down and how turns and straights had to be placed to convey the maximum level of fun. With the introduction of 3D graphics and new, more realistic simulation techniques, the task of designing racetracks became more and more important, as it is now necessary to keep into consideration advanced gameplay mechanics introduced throughout the years. For instance, while in some older 2D games players had only to drive on a straight road avoiding other cars and obstacles, nowadays they have to frequently change direction, elevation, and sometimes they can even use power-ups and jumping platforms positioned on the surface of the track.

A second aspect to keep in mind while developing circuits, especially when talking about multiplayer games, concerns how different drivers will behave when competing against each other or versus the artificial intelligence. Here the layout of the track and the positioning of other functional elements (power-ups, escape routes, shortcuts etc.) have great importance, and doing a good job on this front means making the whole game more enjoyable, hence inducing more players to buy and play it.

The way designers build racetracks is often based on custom built tools that allow to visually place the asphalt strip and all decorations around it. Then, a great deal of effort is devoted to the testing phase, during which they have to actually perform a number of races to evaluate how well the topology of the circuit suits the expected dynamic behaviour of cars, and how unique it feels when driving on it. This building and testing process is

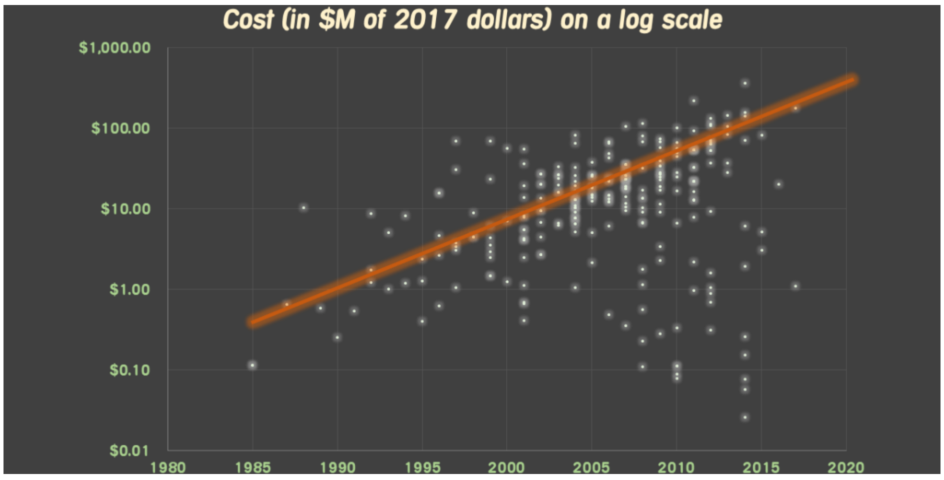


Figure 1.1: Average videogames production cost from 1980 to 2020 [12]

then reiterated multiple times, until it reaches a point where all the static requirements (those relative to the structure of the circuit) and dynamic ones (those regarding racing behaviour) are met.

1.2 Scenario and Problem Statement

Production costs in the videogame industry are increasing every year, reaching numbers that exceed the hundreds of millions of dollars for the largest productions (Figure 1.1). A big part of these costs resides in the creation of the content that establishes the longevity and variety of the game itself (e.g. levels, quests, items). The resulting question that developers regularly pose to themselves is: *how can we reduce costs when working at our games?*. Unfortunately there is no simple solution to this problem, and in reality it strictly depends on the genre of videogames we are working on.

For what concerns racing games, we already mentioned that creating fun and unique tracks is one of the core aspects for their success. However, it is also very expensive to produce each circuit, especially if designers have to take into consideration many racing dynamics arising from the presence of complex mechanics inside the game. Our focus for this thesis then turns to answering the question of *whether it is possible to exploit automatic racing simulations and machine learning techniques to make designing of racing game tracks more efficient.*

By automating the testing phase, we would indeed be able to reduce significantly the number of testing sessions that require real players, therefore sparing lots of resources. Finally, we would need less time to develop tracks if we were able to predict the behaviour of drivers in realistic racing scenarios, as we would need to run only few of those time consuming simulations.

1.3 Methodology

The work for this thesis has been divided into 3 phases: *Development of the simulation engine*, *Development of the track editor*, and *Model fitting of dynamic behaviours.*

The first phase consists on chosing an appropriate racing game that will be used as simulator for running races on circuits we create. For this purpose we decided to use Speed Dreams (SD), whose details will be better explained in Chapter 2, which needs to be modified for allowing the extraction of data while running the simulations.

During the second phase, we develop a track editor that allows to create and modify custom circuits, and that is capable of giving the designer feedback based on the simulations ran with the game that we chose during phase one. We use the game engine offered by Unity, which allows the addition of new functionalities through the use of the C# language, as the basis on which to develop the core architecture of this tool.

In the third phase we concentrate our efforts on the estimation of dynamic characteristics, those that are usually extracted from simulations, through the use of machine learning techniques. Once we have fitted the models used to estimate such features, we evaluate their performances and discuss them. The whole fitting and evaluation process is performed using the Python language, which offers many modules that already implement many of the most important machine learning algorithms.

1.4 Contributions

In this thesis we present as main contribution a set of tools that not only allows the creation and editing of race tracks for racing game, but also performs automatically a set of analyses that are used to give designers feedback and suggestions for the development of their circuits. This software is also built in such way to have the maximum flexibility with respect to the simulator in use, allowing to change it with any other game by applying only a minimum set of modifications.

We provide also objective evidences that, by analyzing the right set of topological features of a track, we are indeed able to predict in a quite precise way some of its main dynamic characteristics.

1.5 Structure of Thesis

The rest of this thesis is structured on five chapters:

- Chapter 2 introduces the state of the art, presenting briefly how track design in videogames has been carried out throughout the years, how testing and evaluation of tracks is performed, and we also present the main technologies used in the project of this thesis (Unity [1] and Speed Dreams [19]).
- Chapter 3 presents the changes we had to make to Speed Dreams as for being able to extract races evolution and other metrics that are needed in the next chapters. It is also shown how the game represents tracks, and the additional tools that we had to develop to allow the execution of simulations remotely.
- Chapter 4 shows the main architectural decisions made during the development of the editor built in Unity, and explains the set of analyses performed on data retrieved from simulations.
- Chapter 5 explains what is the procedure carried out to find predictors of dynamic features of tracks using machine learning techniques. We also give an interpretation of the set of evaluation metrics we computed for determining the performances of such models.
- Chapter 6 concludes the thesis. Here we provide a summary of the work we performed, and the conclusions we can draw from it. We also mention the main limitations of the software developed, and some possible additions that can be done in the future to expand it.

[1]https://unity.com/

Chapter 2

State of the Art

This chapter is dedicated to introduce the video game industry, particularly racing games, together with the main technologies used while developing this thesis.

Section 2.1 is a short overview of racing games, with a special focus on those that offered their players ways to expand their content with additional editors and tools. Given the fact that this is a very wide-ranging topic, this is not to be taken as a complete and exhaustive guide to the history of racing games, but more as a brief introduction aimed at better understanding the industry field of videogames production.

Section 2.2 presents the key elements to be considered when developing tracks in racing games and the main professional roles involved. It is also discussed what are the major issues that can arise while designing them, and how the industry tries to cope with such issues.

Section 2.3 explains what is the current standard workflow to test and evaluate tracks designed in racing games. It first shows what are the main reasons behind tesing and evaluation of tracks, how the current industry performs such operations, and some examples of emerging alternatives to the modern approach.

Section 2.4 is devoted to introducing TORCS and its direct evolution Speed Dreams, the video game used to develop this thesis. After a brief description about its main features and how it is used in the research community, it is explained how it is possible to access data concerning simulations.

The last section (Section 2.5) is focused on describing what is a game engine and, more specifically, what is the Unity game engine and what are the core features we will make use of in the following chapters.

2.1 Brief Introduction to Racing Games Editors

Racing games are one of the most common and most played genre of video games nowadays. Their origin can be traced back to 1974, with the release of Gran Trak 10[1] (Atari) and Speed Race[2] (Taito), two arcade video games. Both had a top-down view and the controls were based on the cabinet equipment, composed of a steering wheel, throttle, and brake pedals. With the evolution of technology and the increasing success of home entertainment solutions (Consoles and Personal Computers), the industry quickly reached high levels of visual quality, realism, and variety in the gameplay. Some games were developed pursuing realism as the main goal (e.g. Indianapolis 500: The Simulation[3]), while others tried to maximise the fun factor and gameplay uniqueness, even if this meant to sacrifice fidelty in the simulation.

In 1984 Excitebike[4] (Nintendo) was released. It was a motocross, side-scrolling video game developed for the Nintendo Entertainment System (*NES*). This was one of the first racing games which gave the players the ability to play on their own designed tracks [5]. Indeed, the game had a specific "*design*" mode with which the player could place obstacles, ramps, and platforms throughout the level.

Another example of racing game with an editor that allowed users to create custom tracks is Stunts[5] (Distinctive Software, 1990), where the player drives around a circuit with the aim of completing a number of laps as quickly as possible. Having relatively advanced physics simulation capabilities and 3D graphics, its editor allowed to also customize aspects like the surface of the track (dirt, asphalt, ice, grass), and to add special types of buildings like bridges and tunnels [3].

From the early 2000s to the most recent years we can name countless racing video games, both arcade and realistic ones, which implement some kind of track editing system. Gene Rally[6] (Curious Chicken Games) has a powerful track editor similar to that of Stunts (Figure 2.1), but with a much more modern and complete user interface. Analyzing Stunts and Gene Rally editors, we can distinguish two very distinct ways of designing such tools. In the former case, the player is provided with some predefined building blocks that can be placed in a grid-based terrain, whereas in the latter case the user can draw the road shape at hand, without limitations. A third

[1] https://en.wikipedia.org/wiki/Gran_Trak_10
[2] https://en.wikipedia.org/wiki/Speed_Race
[3] https://en.wikipedia.org/wiki/Indianapolis_500:_The_Simulation
[4] https://en.wikipedia.org/wiki/Excitebike
[5] https://en.wikipedia.org/wiki/Stunts_(video_game)
[6] https://gene-rally.com/

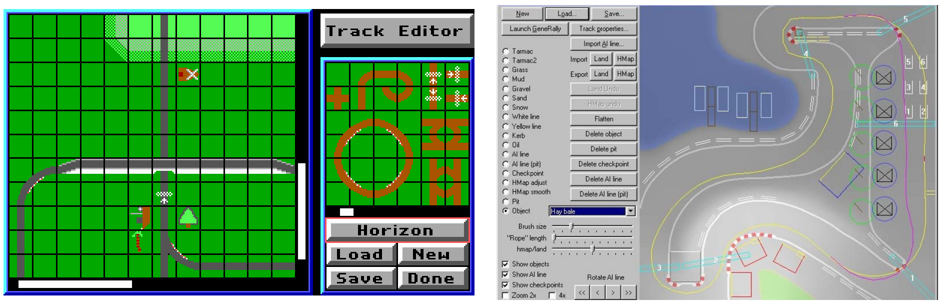


Figure 2.1: Screenshots of Stunts (left) and Gene Rally (right) track editor windows

way to let users build their circuits can be seen in the highly simulative Assetto Corsa[7] (Kunos Simulazioni, 2014). The developers, together with the release of the game, published an independent software which allows to import external 3D models that, once assembled, would make up the circuit itself. This approach, despite allowing a better customization of the visual appearance of the different track elements (e.g. trees and barriers), is less user-friendly, as it requires the player to pick up an external 3D modeling tool (such as Blender[8]), or to buy the necessary assets from external sources. For this reason, the majority of racing games provided with track editors (TrackMania[9] by Nadeo is one of the most famous examples) opt for the much more straightforward experience of giving players a large number of building blocks to assembly their circuits.

This approach works fine when the editor is developed to let the end users design their own tracks. However, developers usually tend to use much richer and more complex editors for developing the circuits that are released officially with the game. Such programs allow to highly customize the environment on and off the track, giving designers and artists a much higher level of freedom during the development of the game.

2.2 Developing Tracks in Racing Games

Two of the most important aspects that make a racing game enjoyable and fun to play are the number of tracks and their quality. Having a large number of circuits means more longevity to the game, together with a higher level of replayability. Still, developing tracks is no easy task. The main professional roles who deal with the designing of racing games tracks are designers and artists. Designers decide the overall layout of the road, the

[7]https://www.assettocorsa.it/en/

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

[9]https://en.wikipedia.org/wiki/TrackMania

racing line shape, where to put run-off areas and what kind of terrain is placed around the asphalt. They also choose the positioning of all those items that may influence the mechanics of the game such as power-ups, traps and checkpoints. During the pre-production phase, they have to decide whether to opt for a real-life circuit or to build one from scratch (if the former option is chosen, the license for that track has to be obtained), and whether to adopt a more realistic approach instead of a more creative one. Artists are in charge of developing the visual elements of the track. They have to create the 3D models, the textures, and the materials that will be assembled together to make the actual virtual circuit. They also have to keep in mind the impact of their work on performances, optimizing as much as possible the graphical elements they create by lowering the number of polygons and the overall complexity of the assets.

The actual production of the virtual circuit can be done in several ways, one consisting of directly scanning the real circuit with a laser scan technology, as done for the game iRacing[10]. This is a time consuming operation requiring expensive tools, so it is usually adopted in games that aim at the maximum available level of realism. Another option is to draw the track manually, or to procedurally generate it with a procedural content generation algorithm. An example of this latter option is Trackgen[11] [2, 14], a track generation system based on genetic algorithms developed at Politecnico di Milano. During the production phase the layout of the circuit is defined by deciding where to put straights and turns, and by setting some specific road parameters. These can vary between numerous different values, including road bank angle, slope and surface type. Eventually the track is enriched with run-off areas and barriers, together with aesthetic props like trees, buildings, and bleachers. This whole process is repeated after the track is tested and the critical aspects that need to be changed are identified. The costs of development raise accordingly every time a modification to the layout of the track is carried out.

Realistic racing games that simulate specific championships, like the F1[12] (Codemasters) and MotoGP[13] (Milestone) game series, are strictly bound with respect to the choice and fidelity level of tracks to the corresponding organization who grants the license. Therefore, in this case, the designers role is limited to the tweaking of the Artificial Intelligence (AI) and that of the vehicles setup for each race. On the contrary, designers play a core role

[10]https://www.iracing.com/track-technology/
[11]http://trackgen.pierlucalanzi.net/
[12]http://www.formula1-game.com/
[13]https://motogpvideogame.com/

in racing games like Mario Kart[14] (Nintendo) and TrackMania[15] (Nadeo). In this type of games tracks have to be carefully crafted in order to match the style of play of the average targeted user. A game like Mario Kart will need circuits that are large and quite simple to be appropriate for children, while games like TrackMania aim at faster and more technical ones.

As mentioned in Section 2.1 some software houses, with the aim of reducing tracks production costs, give their players the tools required to build their own levels. This approach often pays off very well, especially in creative video games (not only in the racing games industry, see Nintendo's Super Mario Maker[16], 2015), as designers can release less official tracks, relying on the players themselves to increase the contents of their video game. Indeed, even if the costs for developing a user-friendly editor may increase the overall game production budget, the resulting collection of tracks created by the community will greatly increase its value and longevity in the long run.

2.3 Testing and Evalutation of Racing Tracks

As discussed in the previous section, testing plays an important role for the good design of a racing game track. While, for real-life circuits, this process can be usually reduced to a minimum, it becomes of vital importance for newly created ones. But how can developers be sure that they are creating high quality tracks? What actually makes a track a *good track*?

The overall enjoyment of a circuit has correlations with a large number of parameters, mainly concerning dynamic aspects emerging when racing on it. Examples are the average speed throughout the racetrack, number of collisions, and overtaking probability. These may be linked to intrinsic topological characteristics of the track itself, like its length, width, and amount of turns [18]. Also, in the case of real circuits, the influence of real-life factors on their satisfaction must not be understimated. Most famous tracks will be perceived in-game similarly to how their real-life counterparts are perceived, even more so if they are frequently present in well-known championships [4].

Both for real-life and fictional circuits, it is extremely useful to have some kind of evaluation metrics that could steer the designers into the right direction during the development phases. Moreover, it can also help them to better understand the aspects of their game that differ from reality. Reason for this is that they can compare how well famous tracks behave with respect to real-life scenarios. As an example, we could take a very fast track like

[14] https://en.wikipedia.org/wiki/Mario_Kart)
[15] See note 9
[16] https://en.wikipedia.org/wiki/Super_Mario_Maker

Monza[17], and try it on our game. If, after having tested the track, the feelings we perceive are not those of a high speed circuit, we can tweak our simulation parameters to better emulate the expected behaviour.

Nowadays the most common way of performing testing on video games is by playing them. Often developers arrange special testing sessions that can be private, if the testers are chosen individually, or public, if the game to be tested is provided to the public without restrictions and before its actual release. These kinds of operations are quite expensive and time consuming, as they require getting in touch with many players and make them try different aspects of the game. In some cases, for private sessions, it may also prove dangerous as some testers could cause information to leak outside of the testing environment, causing potential harm to the company. An alternative to real-life testing sessions is based on Computational Intelligence (CI) [13], which represents a promising technology for greatly increasing the racing quality of the AI, making it behave much more realistically. A sufficiently sophisticated artificial intelligence would indeed allow to fully replace human players, allowing designers to rely exclusively on simulated races, therefore largely reducing the costs of expensive testing sessions. When using simulations ran with only *bots* (artificial intelligence agents driving around the circuit), the amount of data to gather from the races can become quite heavy to handle. *Bots* cannot communicate to designers what are the feelings and the possible improvements to perform on the track. It is now the job of the designers themselves to make such inferences by interpreting the analytic data taken from the progression of the races. It follows that choosing which metrics to record and which not is a quite delicate task. For this purpose, Machine Learning (ML) techniques can be exploited to find the relevant correlations between the dynamics of the race and the topological characteristics of the track being tested [18], hence limiting the data to be gathered during the simulations.

2.4 TORCS and Speed Dreams

2.4.1 Introduction to the simulator

The Open Racing Car Simulator (TORCS) [23] is a 3D car racing simulator developed in C++ as an open source project. As such, it is released with a General Public License (GNU), and it is available for a number of platforms. The video game can be classified as a simulative one, as it implements an accurate physics engine (*SimuV2*) and 3D graphics for rendering images.

[17]https://www.monzanet.it/

Apart from being open source, another of its main characteristics is the ease in extending its core functionalities, given by a modular system which allows the programmer to only recompile what has been actually modified or added, instead of doing so for the whole project every time. As an example, it is possible to add new racetracks and *bots* just by adding the required files and recompiling, respectively, the tracks and AI modules. These features have granted TORCS a large community of players, that modifies and extends its functionalities with new tracks and cars every year. The simulator is also widely used in the academic field, with students and researchers modyfing the simulator and using it as a platform on which to carry on their studies. In particular, there is a large interest regarding the fields of artificial intelligence [10], cars setup optimizations [11], and procedural content generation [20]. New, custom tools can be easily developed also thanks to well-defined data structures representing the behaviour of the artificial intelligence and the shape of tracks. Examples can be found concerning racetracks editors[18], tools to import vehicles from other games[19], and the previously mentioned procedural generation of tracks[20] [2, 14].

Speed Dreams (SD) is the racing car simulator used for the purpose of this thesis. It is also known as *TORCS-NG (TORCS Next Generation)*, as it is the direct evolution of TORCS. The two video games share the same underlying architecture, the same interfaces, and the same data structures for circuits and *bots*. As a consequence, backward compatibility is preserved for the majority of game contents. Nevertheless, Speed Dreams implements a new version of the physics engine (*SimuV4*), introduces custom weather conditions, and it comes with more advanced artificial intelligence robots. It also supports performing simulations without running any video output, allowing to perform races at increased speed by just calling a specific command on the Command Line Interface (CLI). This method requires to give all parameters about the simulation to perform, such as the circuit to load and each drivers' starting position, through an XML document.

2.4.2 Races in Speed Dreams

As mentioned in the previous section, Speed Dreams uses the same TORCS interfaces in most of its code. This also applies to those representing the race current situation and evolution, together with the ones from which it is possible to access data about the individual drivers. The interfaces provided

[18]`http://www.berniw.org/trb/index.php?page=downloads`
[19]`torcs.sourceforge.net/index.php?name=Sections&op=viewarticle&artid=31`
[20]`http://trackgen.pierlucalanzi.net/`

Algorithm 1: Structure representing current driver state

```
structure {
    InitCar Info;
    PubCar Public;
    CarRaceInfo Race;
    PrivCar Private;
    CarCtrl Control;
    CarSetup Setup;
    CarPitCmd PitCommand;
    RobotInterface Robot;
    DriverCurrentState* Next;
} DriverCurrentState;
```

to the programmer that are needed to access the current race state are in the form of data structures, instead of being implemented as methods to be called during the execution of the simulation. Algorithm 1 shows an example of such structure. DriverCurrentState represents, as the name says, the current state of each driver at a given instant in time.

InitCar contains information about the initialization of the car, like the name of the driver, its number, measures and other parameters that remain constant throughout the entire race.

PubCar and PrivCar are, respectively, data accessible by all drivers in the race and data accessible only by the considered driver. Examples of public data are the position of the car on the circuit and its current state (racing, at pits, DNF etc.), while in the private section we find fields like the damage taken by this car, the current gear and the level of fuel.

CarRaceInfo contains some of the most relevant information about the driver we are taking into consideration. It keeps track of the laps driven so far, the top speed reached, the current position in the leaderboard, the time difference between this driver and the leader of the race, the one behind the previous driver, and that before the next one.

CarCtrl, CarSetup and CarPitCmd represent, respectively, the current inputs given by the player (or the artificial intelligence) to the car, the current setup of the car, and the instructions to perform the next pit stop (whether to change tires and how much fuel to load).

RobotInterface exposes the common interface of functions implemented by every *bot* [22], while the field *Next* tells that this is one of many nodes that are part of an unidirectional linked list containing all drivers of the race.

2.5 Unity

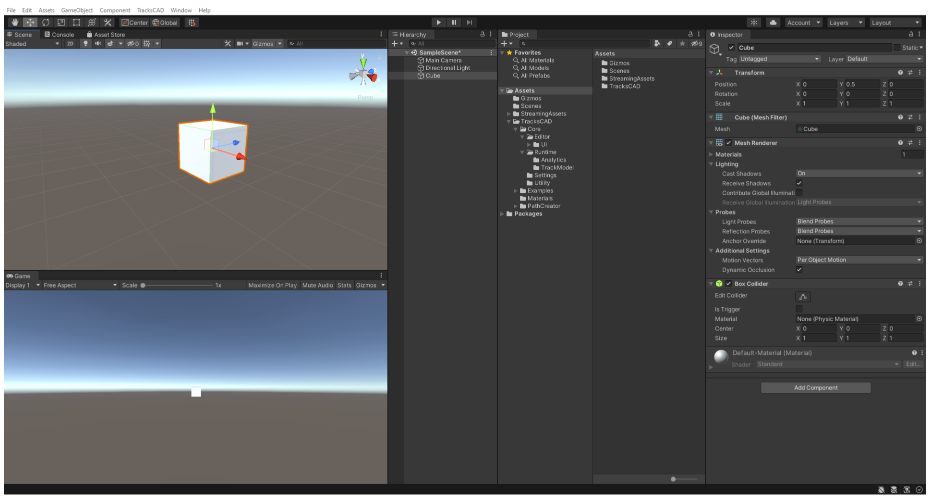
Unity[21] is a game engine developed by Unity Technologies, available for all major operating systems on the market. It is one of the most used engine nowadays, having around 1.5 million active creators each month [7].

A game engine is a piece of software which implements the basic framework for the development of video games. Generally, it includes useful libraries, and exposes an API (the interface required to use the framework) for helping developers in making their games. Also, one of the main features that make such programs very popular among a number of different industry fields, is the presence of a rendering engine for displaying 2D and 3D graphics. Other core functionalities provided may include physics simulation, support for sounds, animations, artificial intelligence, networking frameworks and many more. Often game engines offer generic components that can be reused and assembled by the user according to what the game needs. For instance, if we are developing a 2D game, we could attach collision boxes provided with the engine to our 2D images, with the intention of making objects in our levels react when they collide with each other. On the contrary, the contents of the game, models, textures, and the way objects interact with the world, are elements that need to be defined by developers, designers and artists [21].

Unity engine implements a graphical user interface (Figure 2.2), from which it is possible to see what items are currently in the scene, what it is currently shown in the game, and from which it is possible to select and modify the individual objects we have created. We can also test the game without having to build a separate executable every time. Moreover, we are able to perform debugging easily, by looking at what is happening in the scene while we are testing the game itself.

From a technical point of view, Unity offers a powerful but quite simple C# framework, based on a system of classes, libraries, and data structures. All objects placed in the scene have one or more components. Every custom component implements a functionality and/or keeps some data about the object it is part of. More precisely, every user-made component (*script*) is a C# class that has to inherit from the MonoBehaviour one. A custom *script* can modify and access all other components and all other objects in the scene, and it can even create new components and objects while the game is being played. All other types of components, those not inheriting from the same class as the *scripts* do, represent additional functionalities built into the editor that can be used to add elements to the game.

[21]https://unity.com/



Cube Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0.5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cube (Mesh Filter)
Mesh Cube

Mesh Renderer
Materials 1
Lighting
Cast Shadows On
Receive Shadows ✓
Contribute Global Illuminati
Receive Global Illumination Light Probes
Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Additional Settings
Motion Vectors Per Object Motion
Dynamic Occlusion ✓

Box Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

Default-Material (Material)
Shader Standard Edit...

Add Component

Figure 2.2: Screenshot of the Unity editor user interface

Another useful feature offered by Unity is the ability to easily extend the editor itself by adding new windows, buttons and even completely new functionalities to the engine. This allows users to create shortcuts for redundant operations, or to implement ways of better managing their resources (models, textures, audio files etc.). It is also frequently exploited to make custom interfaces for designers, with the aim of making their workflow more organized. This specific feature plays a central role in the development of this thesis, as the editor has been considerably extended for allowing the designing and testing of racetracks.

Chapter 3

Developed Tools for Tracks Analysis

In Section 3.1 the set of changes made to the Speed Dreams game is described. These consist in the addition of a new module that performs the data gathering process during races.

In Section 3.2 we describe the overall architecture of the framework that will expose the interface to perform simulations remotely, its components, and the way they work.

3.1 Changes made to Speed Dreams

For being able to run simulations on Speed Dreams and extract data about their progress, it is necessary to apply some changes to its code. This section explains how SD handles the information we need, how it is possible to extract it, and how to create and load new, custom tracks that satisfy its format. Eventually, the process of how the simulations starting grid are chosen is discussed, together with a generic overview of the performances of the chosen *bots*.

3.1.1 Track Representation in Speed Dreams

Every Speed Dreams track is loaded from an eXtensible Markup Language (XML) file, which follows a well defined structure [1]. This document, for being recognized by the video game as a valid track, must contain a number of mandatory specifications. In Table 3.1a it is possible to see what are such parameters. The meaning of fields Track name, Authors, Description, and Road width is quite straightforward. Value of Track type is needed

Track attributes
Track name
Authors
Description
Track type (`Circuit, Dirt, Speedway`)
Road width
Segment step length
Pit lane width
Pit lane side
List of pit lane segments
List of segments

(a) Generic track attributes

Segments attributes	
Name	
Road slope (`%`)	
Starting bank angle (`deg`)	
Ending bank angle (`deg`)	
Segment type (`Straight, Left turn, Right turn`)	
Straight	**Turn**
Length (`m`)	Arc (`deg`)
	Starting radius (`m`)
	Ending radius (`m`)

(b) Segment attributes

Table 3.1: Attributes required to define a Speed Dreams track and segments

to correctly classify the track in the game, helping it understand what types of cars are able to race in the considered circuit. For instance, in racetracks classified as *dirt*, the game will only allow rally cars to attend races located there. SEGMENT STEP LENGTH is used when building the track and its meaning will be better described later on, when discussing about what segments are, and how they are generated in practice. Another core part of a racetrack is its pit lane, which is described by its PIT LANE WIDTH, together with the side of the road where it resides (PIT LANE SIDE).

The actual structure of the track is defined as a double linked list of segments, which are its basic building blocks. Segments can be straights or turns and, depending on their type, they need specific parameters to be correctly recognized by the track building algorithm (Table 3.1b). In the case of straights, it is sufficient to define their LENGTH. When dealing with left or right turns, instead, three values are needed. The ARC is the angle, measured in degrees, between the tangents of the first and last points of

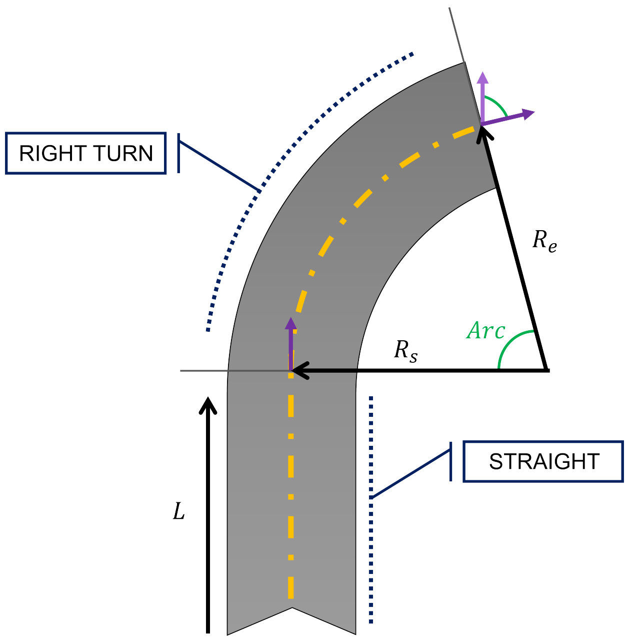


Figure 3.1: Geometric meaning of Straight and Turns mandatory parameters

the segment. To support variable radius curves, the STARTING RADIUS and ENDING RADIUS parameters can be defined. The latter value is optional, as it is set equal to the former by default. In Figure 3.1 it is shown the geometric interpretation of these parameters in a graphical way.

It is also possible to specify values like the slope of the segment (the height difference every 100 meters of road), and the starting and ending angles at which the road is inclined around its longitudinal axis, with respect to the horizontal (bank angles). Every segment can be also enriched with information about the precence of curbs, the type of terrain around the asphalt, and the type of safety barriers (guard rails and tire walls) placed on the border of the track.

The first segment is always placed at the origin of the spacial reference system, unless the Z START parameter is added and it is different from zero. In such scenario, the whole track will be constructed beginning from that height value, which will represent its starting altitude.

For a track to be considered valid, it is also mandatory that the ending position of the last segment is equal (or very close) to the starting position of the first segment. Moreover, the last point tangent vector needs to be equal (or very close) to the starting tangent vector of the track. The game does not automatically close the circuits it loads, nor it fixes its tangents, therefore it is job of the designer to enforce the satisfaction of these constraints.

When building the track, Speed Dreams subdivides each segment into other, smaller pieces. These sub-segments, whose length is equal to SEGMENT STEP LENGTH, are necessary primarly to correctly approximate the shape of variable radius turns. A more in-depth view on how this type of segment is built can be found in Chapter 4 - Section 4.2.

3.1.2 Data Extraction of Simulations

As seen in Section 2.4, it is quite simple to keep track of the progress of races. It is sufficient to access the right data structures, and from there we can gather all the information we need. Furthermore, the modular structure inherited by TORCS allows to efficiently expand the simulator with new functionalities, without having to rebuild the whole project every time we need to modify our code. We can benefit from this feature by developing a new module responsible for keeping track of different metrics during the simulations. We will call this piece of code *rlog* (race log). Whenever a race is loaded, an instance of *rlog* is created, and its data structures are initialized. Whenever a relevant event takes place during the race, the corresponding information aimed at describing such event is collected and stored in these data structures. Once the race ends, the module creates a new file on which it writes the recorded data. A list of all metrics tracked and saved by *rlog*, together with their detailed description, is availabe in Table 3.2.

The format chosen to represent race reports is *json*. The main reason behind this choice is that, having a large number of entries to write (mainly given by the timeline), it is necessary to keep to a minimum the average overhead given by keywords of the chosen standard. *Json* only defines single character tokens, hence reducing the potential overhead and, consequently, the reports file size.

3.1.3 Choice of Races Start Configurations

Once the metrics to be tracked have been fixed, it is time to setup the actual files with all the instructions needed to perform the races. We define a total of four different competitions and, for each of them, we need to specify:

- **Name:** the name of the race. This is particularly useful as it uniquely identifies it with respect to other races.
- **Type:** structure of the competition. In SD it is possible to define both single event races, and more structured events, like championships with multiple tracks and qualifying sessions.

Metric	Description
Track name	
Laps	Total number of laps executed during the race
Track length	
Track width	
Grid start	Starting race grid. Each element is the name of the bot which started at the corresponding array position.
Grid end	Grid at the end of the race. Other than the names of each bot in finishing order, this array contains: • Top speed reached by each bot • Time gap from each bot to the preceding one • Total amount of damage taken by each bot
Overtakes	Each element represents an overtake that took place during the race, and it contains: • Name of segment in which the overtake happened • Timestamp during which the overtake took place • Lap during which the overtake took place
Collisions	Each element represents a collision between two drivers or between a driver and some part of the circuit, and it contains: • Type of collision (Driver-Driver, Driver-Track) • Damage of each of the cars involved • Name of segment the collision took place • Lap during which the collision happened
Time gaps	Information about the time between each bot and the one preceding it, computed once for each lap
Timeline	Complete timeline of the race. For each second, it holds the full race grid and, for each bot, it tracks its: • Name • Position (segment name and distance from its start) • Speed • Laps completed so far

*Table 3.2: Metrics extrapolated by the race logger (*rlog*) during each simulation, with their description*

- **Description:** a brief description of the race.
- **Track(s):** a list of one or more circuits for this race, depending on what the type of the competition is.
- **Starting grid:** a list of the race competitors, ordered with respect to their starting position. Each driver (or *bot*) is uniquely identified by its ⟨*module*, *index*⟩ tuple. *Module* is its AI name (in the case of real players, this is the *player* module), while the *index* is the specific instance of the chosen module. In Speed Dreams there can indeed be a maximum of 10 *bots* using the same AI module, each with an index, a name, and its custom car configuration.

The need of performing multiple races is given by the fact that, in this way, we can put the track to the test in different starting conditions, hence obtaining data that is more descriptive of the group racing dynamics we decide to extrapolate (e.g. collisions and overtakes). Ideally we would need to perform a number of races equal to the number of distinct permutations that we can build with the chosen set of *bots*. Despite allowing to consider every possible starting scenario, this would result in the need of an excessive amount of time to obtain simulation results. A potential solution to this problem would be to run a reduced number of races, each with a randomized starting order. For the work of this thesis, however, the results of the simulations need to be deterministic, as to ensure the evaluation of two structurally equivalent circuits to yield the same outcomes. Consequently, no starting grid randomization can be implemented and, on the contrary, a set of only four races is defined, each of them having a predefined starting grid order. We can therefore analyze the starting dynamics with multiple simulations, while still obtaining deterministic results with respect to the topology of the track we are designing.

First, it is necessary to choose the more suitable set of *bots*. Speed Dreams offers three quite advanced artificial intelligence modules, coming from improvements made to the previous generation ones present in TORCS: Simplix, Usr, and Dandroid. Simplix generates numerous stability issues to the game when starting races in particular tracks, mainly those with very large turns. Looking more in-depth into the source code, it seems that this is caused by its different approach to the generation of the racing line. The resulting list of *bots* that seems to be more suitable is composed of the modules Usr, Dandroid, Mouse and Shadow. Every *bot* is tested by first making it drive a lap around a circuit alone, and then by running races with random starting order, checking the behaviour of each driver when

Driver	**Best Lap**	**Avgerage Damage**	**Average Positions Gained**	***Bot* Type**
Dandroid	1:37:910	63.83	-1.6	default
Usr	1:39:291	379.5	3	default
Mouse	1:42:935	193.5	0.5	TORCS EWC
Shadow	1:45:538	134	-4.33	TORCS EWC

Table 3.3: Bots *performances across four races at Brondehach circuit. Those marked as* default *are the ones that come with the Speed Dreams game, while the* TORCS EWC *ones are those imported from the TORCS Endurance World Championship*

racing against others. The results of these tests can be seen in Table 3.3. As mentioned above, Usr and Dandroid come by default with the simulator, and demonstrated a pretty advanced level of skill when dealing with overtakes and solitary racing, scoring the two best laps. The other two modules need to be added from TORCS, in particular they have been developed for the TORCS Endurance World Championship (TORCS EWC). Mouse is the winner of the last iteration of the championship (2019), and it performed quite well both driving alone and with other competitors. Shadow has a more conservative racing technique, which grants it not to suffer too much damage when playing with others, although scoring slower lap times.

Eventually, the actual starting grids are chosen. Each of these is a permutation of the 8 drivers whose behaviour is based on the above four *bot* modules, each considered twice (two instances for each module).

3.2 Other Tools

In order to perform the simulations necessary to extrapolate the dynamic metrics from races, we develop a framework that allows to easily execute simulations remotely. The architecture of this framework, of which a high-level diagram is shown in Figure 3.2, must guarantee the maximum level of interchangeability of the simulator in use. In other words, the system must expose an interface that is independent of the video game to perform the simulations with. Therefore it is necessary to define and use a new format with which to represent tracks, that is also unconstrained by, in this specific case, Speed Dreams.

The resulting architecture is composed of three main components: the TracksCAD Simulation Server, the Track Converter, and the Simulator. The first two items are programs developed using the Python language,

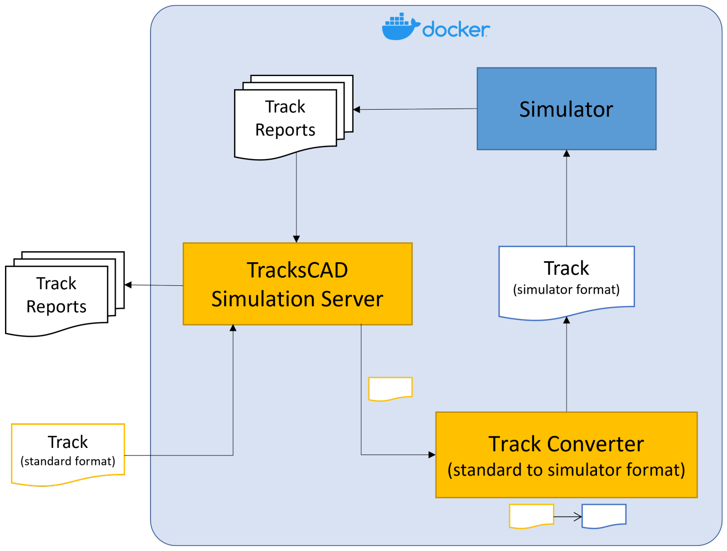


Figure 3.2: High level view of the remote simulation framework

and a more detailed description for each one of them is presented later in this section. Instead, as for the requirement seen before, the *Simulator* is represented as a generic component that can be selected based on the type of designing aids one wants to achieve, and on what kind of racing game we are working with (e.g. simulative or arcade).

To make everything as independent as possible of the operating system in use, an image was created for a Docker[1] container, containing all the modules mentioned above, combined with the dependencies required for them to work. Docker is a set of software products that uses OS-level virtualization to run individual, isolated packages of software called containers. Every change made when operating within a container are discarded once that container is deleted.

3.2.1 Track Format Converter

Still with the aim of ensuring maximum versatility in the choice of the simulator, the proposed approach is not to represent tracks in the Speed Dreams XML format, but rather to use a generic standard that includes only the essential information, and which can be easily converted to the format required by the simulator in use. When defining tracks, SD also requires a considerable amount of information useful exclusively for that specific game, introducing a significant overhead to the representation of

[1] https://www.docker.com/

tracks: another reason for finding a more compact way of describing them. For consistency, and for the same reasons described in the previous section concerning overhead avoidance, we choose *json* also as our new track format. An example of how a track is represented using the generic standard can be seen in Figure 3.3.

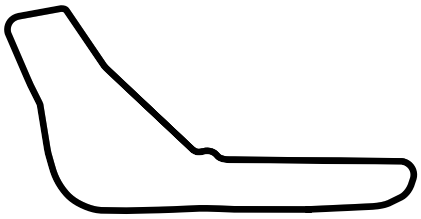
The job of the Track Converter module is thus to convert tracks from the generic representation to the simulator one. As a result, this tool can be changed whenever using a game that requires a different track specification standard. For instance, if our simulator requires binary files to represent its tracks, we would need to build a new converter from scratch which performs such conversion from *json* text to binary.

3.2.2 Remote Simulations Execution

This module starts a remote server that listens for TCP connections on a specific port. It exposes an interface that can be called up through plain text messages. Examples are:

- LOAD TRACK JSON: Start loading a new track in json format into the server. This will then be saved locally, converted using the Track Converter tool and, eventually, saved into the simulator folder.
- START BENCHMARK: Communicates to the server that the required track has been loaded, and that we are ready to receive the results of the simulations. The server will then perform the races, and send the respective logs one after the other.
- GET LOGS: Repeats the process of sending logs to the user, without the need of simulating again the races.

Figure 3.4 shows a UML Communication Diagram representing a typical interaction between a program that wants to run a simulation, the server, and all the components mentioned in this section. In the example, the client is the Unity tool developed in this thesis, and whose details will be explained more in depth in the next chapter.



(a) Top-down view of track Forza

```
{
  "name": "Forza",
  "category": "circuit",
  "version": 4,
  "author": "A. Sumner",
  "description": "A very fast
      and smooth circuit in
      Northern Italy",
  "width": 11,
  "profil_steps_length": 4,
  "pits": {
    "side": "right",
    "entry": "pit entry",
    "start": "pit start",
    "end": "pit end",
    "exit": "pit exit",
    "length": 13,
    "width": 5
  },
  "segments": [
```

```
    {
      "name": "main straight",
      "type": "str",
      "lg": 120,
      "grade": {
        "val": "0.0",
        "unit": "%"
      },
      ...
    },
    ...,
    {
      "name": "curva grande",
      "type": "rgt",
      "arc": 3.5,
      "radius": 376.5,
      "end_radius": {
        "val": "342.0",
        "unit": "m"
      },
      ...
    },
    ...,
  ]
}
```

(b) A piece of json representation of the Forza *track*

Figure 3.3: Example of json track encoding

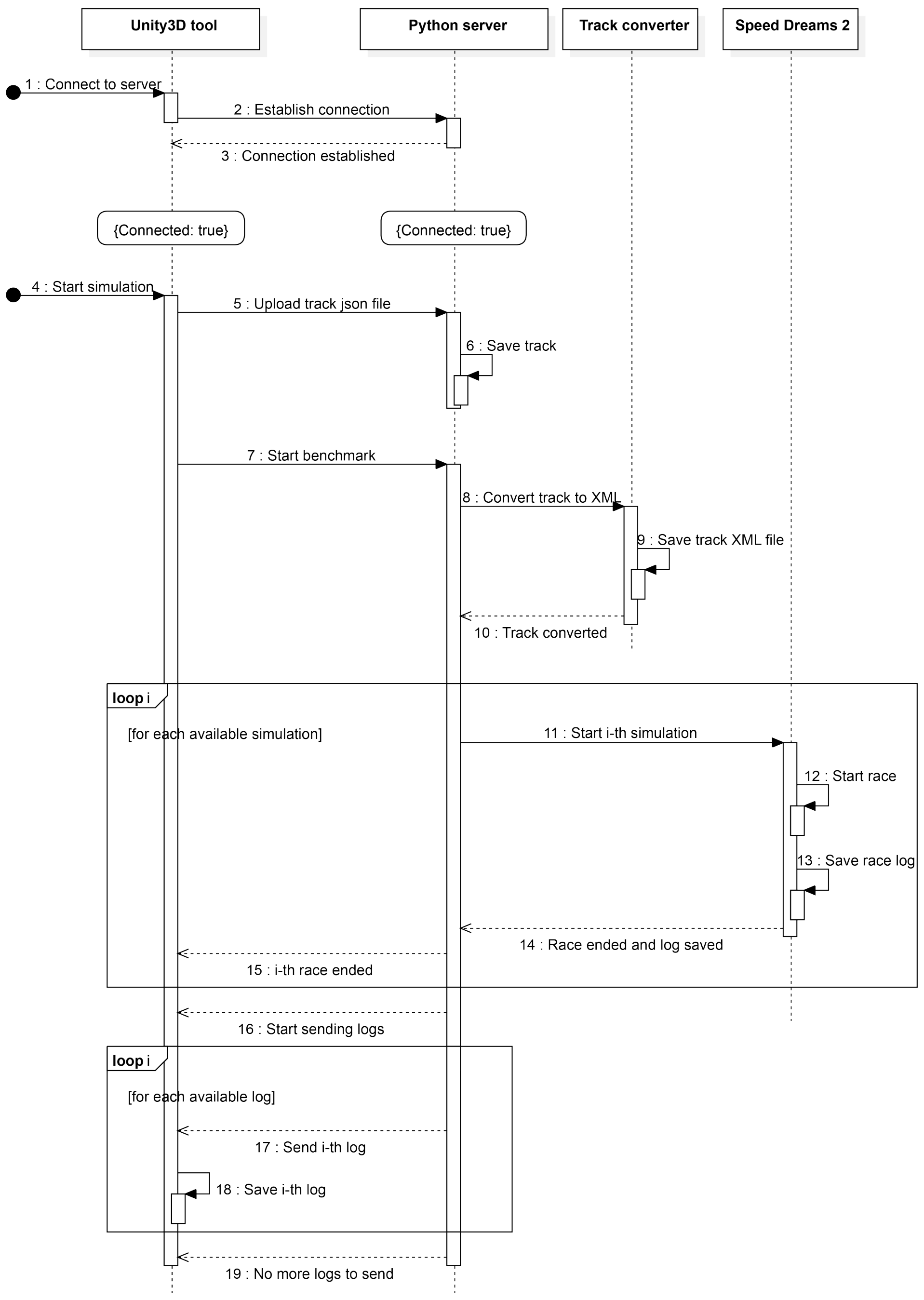


Figure 3.4: UML Communication Diagram

Chapter 4

Track Editor

In this chapter we discuss the main architectural decisions made during the development of the TracksCAD editor tool, and we take a brief look at some of the most relevant implementation details. No actual code, nor architectural schemes that are too specific (e.g. UML diagrams), have been placed in the sections of this chapter, as they would dive too deeply into the designing details of the tool, hence diverting from the description of how the editor itself can be used by designers to help developing new tracks.

In Section 4.1 the overall high-level architecture is described, together with the interactions between the editor and the simulation server.

Section 4.2 shows the main features developed for the creation and editing of new custom tracks. Screenshots of the tool user interface are also present to better understand how users can interact with it.

In Section 4.3 the working principles of a system that automatically closes tracks are discussed.

Section 4.4 presents the set of analyses that are carried out after performing the simulations, and after retrieving the corresponding reports.

Eventually, Section 4.5 shows how, directly from the editor tool, it is possible to inspect the evolution of the performed races.

4.1 Overall Architecture

A major contribution of this thesis is the design of a tool that allows the creation, editing, and automatic evaluation of racing game tracks. We develop the mentioned piece of software in the form of an extension to the Unity editor. We can therefore take advantage of its already existing 3D renderer, as well as its APIs. Some of the particularly useful features implemented by Unity, or directly offered by the C# language, concern data structures (lists and dictionaries), vectors manipulation, and garbage collection. A big role in the choice of such underlying technology is played by the ease with which it is possible to customize it by adding functionalities and new custom windows. In regard of this aspect, Unity user interface can be conveniently extended by adding new classes which inherit from the built-in EditorWindow class. From there, it is possible to override a specific method (OnGUI()) whose job is to update, several times per second, the interface displayed to the user, whenever it is active on the screen. Part of Unity APIs are devoted to facilitate the writing of graphical interface elements, with functions that add buttons, text fields, dropdowns and other types of items. The programmer is therefore discharged from writing any HTML or CSS piece of code, as these are automatically generated by the mentioned methods.

From a designing point of view, the track editor tool needs to be able both to create a new track from scratch, and to load existing ones from external sources. Accordingly, another necessary functional requirement is the ability to save the current track on a file, using the same generic representation format discussed previously in this thesis (Chapter 3, Section 3.2.1).

Similarly to the way they are implemented in Speed Dreams, racetracks are represented as double linked lists of segments, and the designer can edit the circuit by adding, removing and modifying them through the use of an ad-hoc Graphical User Interface (GUI). Once the design phase is completed, it is possible to automatically generate a closing segment that precisely connects the last point of the track built so far with the start line.

The high-level architectural scheme in Figure 4.1 shows how the editor interacts with the tools discussed in Chapter 3. It employs the simulation server with the aim of obtaining data about the metrics collected while racing in custom designed tracks. It then uses this information to generate suggestions which can help the designer in fixing some of the critical aspects of the circuit. Also, by loading into the editor each log returned by the simulator, it is possible to view the progression of races by looking at the drivers' position and speed over time.

4.2 Track Design and Editing

4.2.1 Creating and Loading Tracks

In TracksCAD editor there are two ways of starting the designing of a track, one being to create a completely new one, and the other consisting of loading an existing file from our local machine. An exception to this is represented

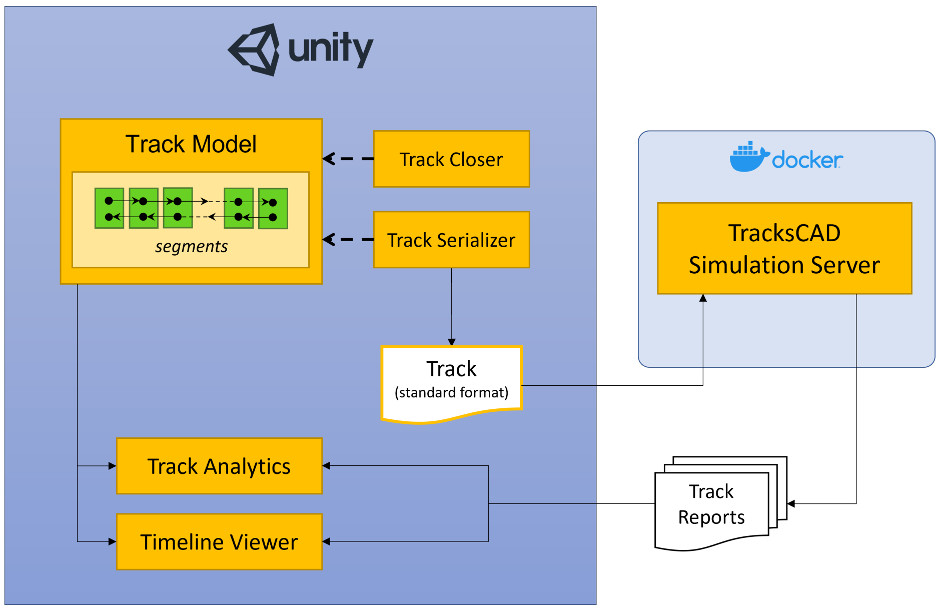
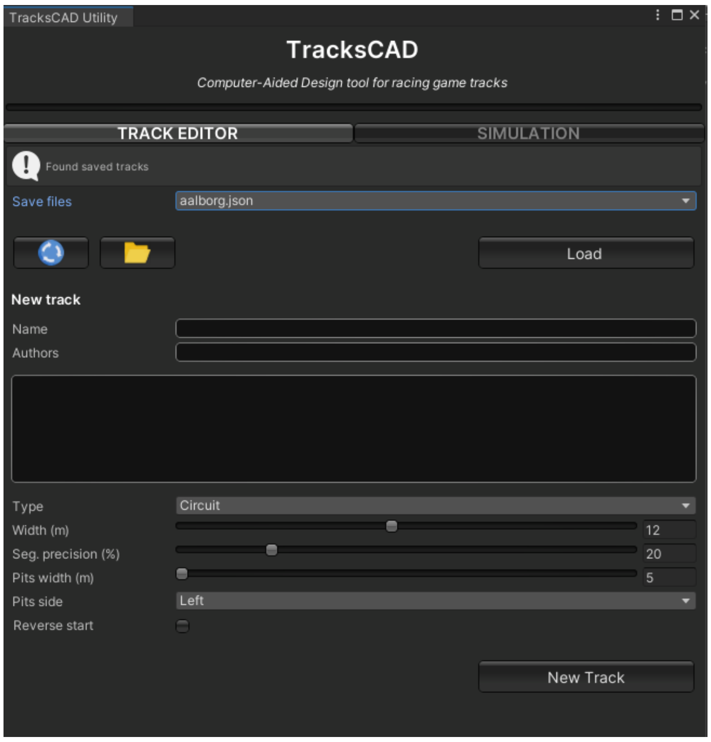


Figure 4.1: High level components scheme of the TracksCAD architecture. The arrows represent data flow, the yellow blocks are the main components of the system, while the dashed lines link components that extend other components.

by the scenario in which a track was still loaded the last time Unity was closed. In such case, the tool reloads the information of that track from its corresponding file, and uses the already existing segments in the scene for the 3D representation. When opening the TracksCAD Utility window (Figure 4.2), both options are displayed. In the upper part of the interface, the tool shows all track *json* files found in the `/StreamingAssets/Tracks` folder. In the case in which the designer wants to start the creation of a completely new circuit, it is possible to do so after filling all the mandatory parameters, and by using the *New Track* button. All required fields have been already discussed in the previous chapter except for the *Reverse start* flag. This toggle is used to tell the editor that the newly generated track will be traversed in the opposite direction with respect to the order in which the segments of the circuit will be laid down. In such case the starting grid will be directed towards the last piece of the track (the one that closes it) instead of the actual second segment built. This is particularly useful if, later on, the designer wants to test the circuit by racing in both clockwise and counterclockwise directions.

The tool performs different operations whether we are loading an existing track or creating a new one. In the former case, the relative *json* document is loaded, parsed, and the generic attributes are stored in a specific class


TracksCAD Utility

TracksCAD

Computer-Aided Design tool for racing game tracks

TRACK EDITOR | SIMULATION

Found saved tracks

Save files: aalborg.json

Load

New track

Name

Authors

Type: Circuit

Width (m): 12

Seg. precision (%): 20

Pits width (m): 5

Pits side: Left

Reverse start

New Track

Figure 4.2: TracksCAD Utility window - Loading of a new track and new track setup fields are displayed before beginning designing

Track. Afterwards, the list of segments is read and each of them is constructed following the specifications described in the file. If, instead, the goal is to generate a circuit from scratch, a new Track object is instantiated and populated with its name, type, authors, description, width, segment precision, and pit lane specifications. The segment precision attribute determines how precisely variable radius turns are approximated with respect to the corresponding geometric shape (Euler spiral). The lower the precision, the more turns are approximated to circular arcs having as radius the average between the start and end turn radiuses. Since a track without any segment would make no sense to exist, TracksCAD always adds one short straight to newly generated circuits, and sets it to be the start line. On the contrary, no pit lane entry or exit is set by default, as it would be too short to support the presence of all the teams' garages.

Once the racetrack is loaded the segment editing interface is displayed, and the designer can now start working on its layout. To view its current shape, the 3D object of each segment is generated automatically in the Unity scene (Figure 4.3), and the current racing direction is shown to the user through the use of purple arrows placed on its surface.

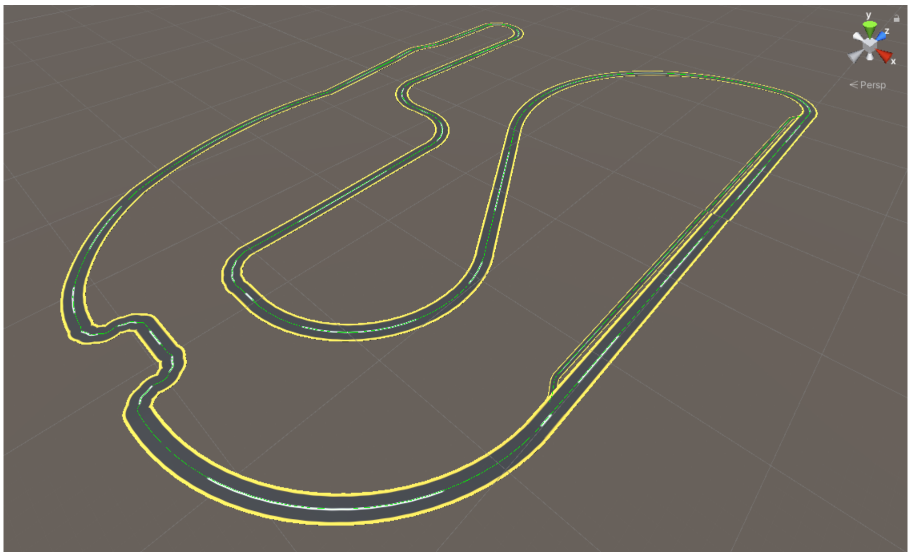
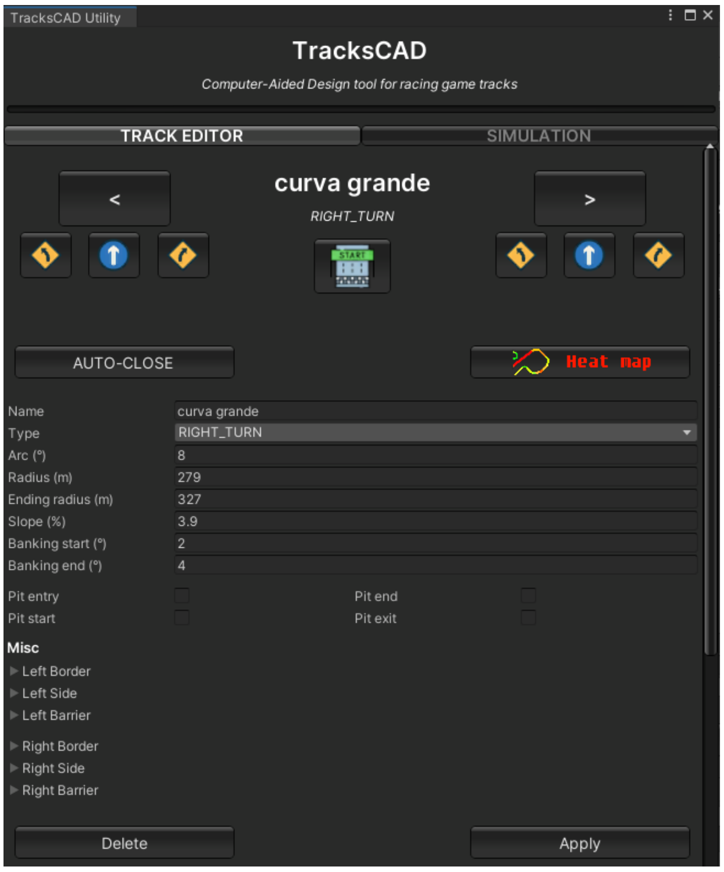


Figure 4.3: Unity scene when track E-Track 1 *is loaded*

4.2.2 Segment Editing

The main way of changing the layout of a circuit is to edit its segments. To do so, the TracksCAD Utility window offers all the necessary tools for adding, deleting, and setting segment parameters. Once the track is loaded and ready to be modified, the window changes its appearance, switching to the segment editor view shown in Figure 4.4.

Analyzing this piece of GUI from top to bottom, we can see the name of the currently selected segment and two arrowheads buttons for shifting the selection to the previous (left arrow) and next (right arrow) segment. Another, more intuitive way of switching between segments, is to directly pick up their 3D object from the Unity scene. The buttons below the navigation controls are meant to add segments to the track: left turns, straights, and right turns respectively. The three on the left part of the window will add a segment before the current selection, while those on the right will add it in front. At the center of the screen, just beneath the segment name, it is possible to set the current segment as the new start line of the track. This means that, once the circuit is passed to the simulator, the starting grid will be placed just behind the beginning of such segment. The bottom half of the window is dedicated to the editing of the selected segment, showing its specific parameters depending on whether it is a turn or a straight. The *Auto-Close* and *Heat map* buttons will be better discussed in Section 4.4. Whenever the *Apply* button is pressed, the program checks the validity of the inserted parameters, updates the track model and its 3D representation



Name	curva grande
Type	RIGHT_TURN
Arc (°)	8
Radius (m)	279
Ending radius (m)	327
Slope (%)	3.9
Banking start (°)	2
Banking end (°)	4
Pit entry	☐
Pit start	☐
Pit end	☐
Pit exit	☐

Figure 4.4: Segment editor screenshot

in the scene. This process causes a chain reaction that requires to recursively recompute the position and rotation of all segments that are placed after the modified one.

The practical construction of a segment depends on whether it is a straight or a turn. The former case is the simplest, as it only needs two points connected by a line. On the contrary, a turn needs an iterative procedure that approximates the points and tangents of an Euler spiral (Clothoid) based on its ARC, START RADIUS, and END RADIUS parameters. Algorithm 2 shows the main steps of such process. The general idea is to subdivide the turn into smaller, fixed length sections l_{step}, each being a circular arc whose radius is chosen as a linear interpolation between the starting and ending radiuses, and the angle being computed with the formula

$$\theta_i = \frac{l_{step}}{r_i} \tag{4.1}$$

Algorithm 2: Finding points and tangents of left/right turns

```
**Input:** $R_{start}$, $R_{end}$, $Arc^{deg}$, $Slope^{\%}$;
**Output:** $\langle \vec{p_0}, \vec{t_0} \rangle, \langle \vec{p_1}, \vec{t_1} \rangle, ..., \langle \vec{p_n}, \vec{t_n} \rangle$;
**begin**
1       $L \leftarrow Arc^{rad} \cdot \frac{(R_{start}+R_{end})}{2}$;
2       $steps \leftarrow \left\lceil \frac{L}{l_{step}} \right\rceil$;
3       $\Delta r \leftarrow (R_{end} - R_{start})/steps$;
4       $\Delta h \leftarrow l_{step} \times \frac{Slope}{100}$;
5       $r_0 \leftarrow R_{start}$;
6       $h_0 \leftarrow 0$;
7       $i \leftarrow 0$;
        **repeat**
8           Compute $\theta_i$ using (4.1);
9           $\langle \vec{p_i}, \vec{t_i} \rangle \leftarrow$ point and tangent of circular arc $A(r_i, \theta_i, h_i)$;
10          $r_{i+1} \leftarrow r_i + \Delta r$;
11          $h_{i+1} \leftarrow h_i + \Delta h$;
12          $i \leftarrow i + 1$;
        **until** $i >= steps$;
**end**
```

More precisely, l_{step} is computed by interpolating linearly the segment precision in such a way that a value of 0% results in having the maximum step size, while 100% corresponds to the minimum length. The number of sections to divide the turn into is computed approximating it as a circular arc, with radius equal to the average of its radiuses.

Every type of segment can also be customized with respect to its bank angles and slope attributes. A banked road is defined as one in which vechicles incline while they drive along it. Usually we talk about banked turns, and the direction in which they are rotated is, most of the times, towards their inside edge. Nonetheless, the editor allows to have both banked turns and banked straights, and their angle can be directed in clockwise as well as counterclockwise direction. The slope of the road is the amount of height, in percentage, that it gains after one unit of horizontal length. When generating segments, the height of each point is computed using linear interpolation between 0 and the final height that must be reached.

At the end of the building procedure, segments objects are moved so that their starting position is in the same location than that of the final point of the previous segment, and they are also rotated so that their starting tangent is correctly directed as well. Eventually, the pit lane attributes are applied, and the corresponding 3D object is generated in the scene (Figure 4.5). The pit lane is considered incomplete if there exists one flag among those described below that is not checked on any segment.

- **Pit Entry:** piece of road that connects the racetrack with the start of the pit lane.
- **Pit Start:** point from which the pit lane speed limit starts to be valid.
- **Pit End:** point from which the speed is no more constrained by the pit lane limiter.
- **Pit Exit:** piece of road that connects the end of the pitlane with the racetrack.

Eventually, the designer can add or modify the borders, sides and barriers of the road for the selected segment. These consists, respectively, on the presence and physical characteristics of curbs, the type of terrain around the asphalt, and the type of safety barriers on the track (tirewalls or guard rails). Figure 4.6 better explains the meaning of each of the above mentioned elements, while Figure 4.7 shows an example of the entire TracksCAD GUI that is presented to the user while editing a segment.

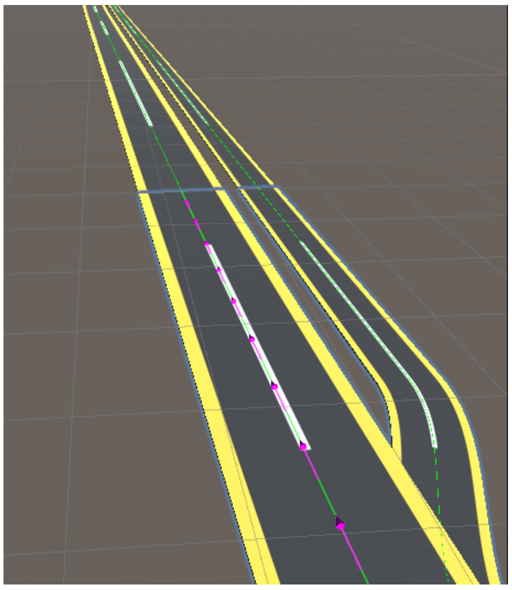


Figure 4.5: Example of pit lane 3D object

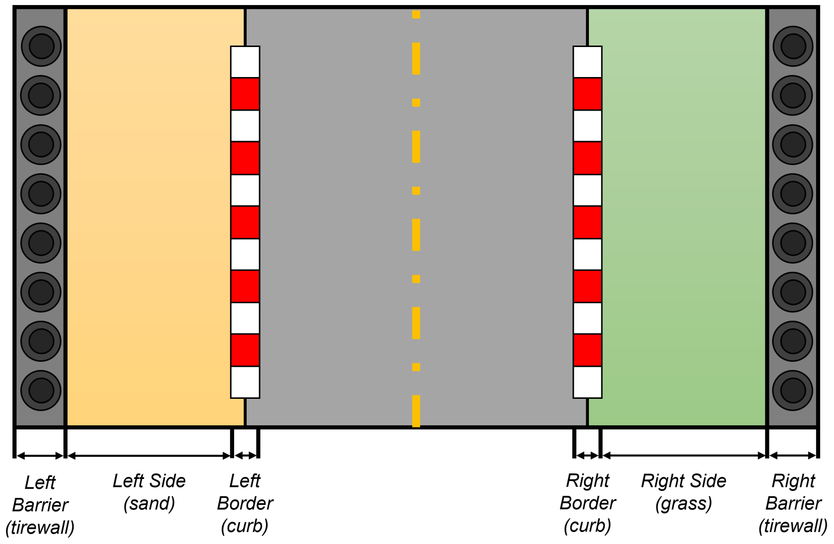
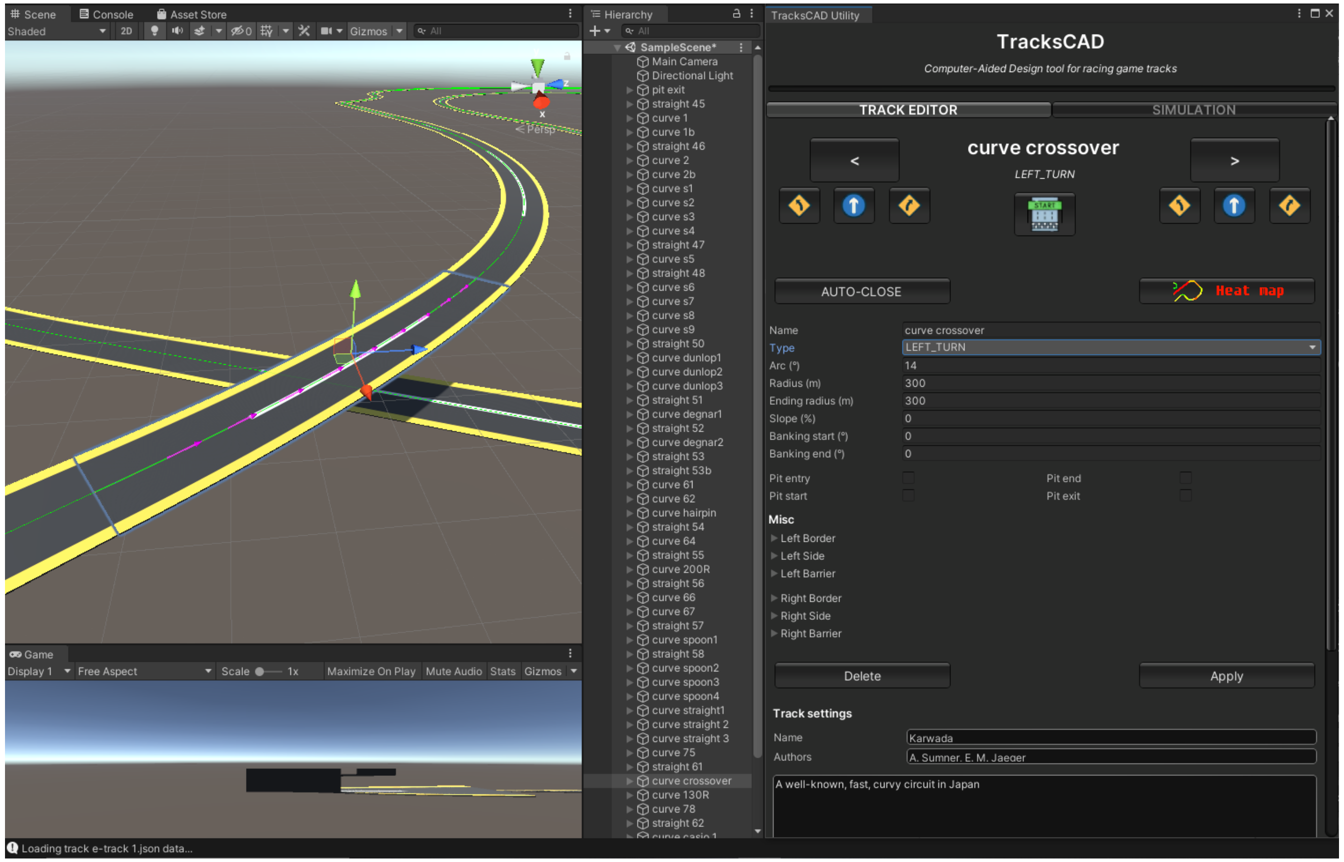


Figure 4.6: Segment borders, sides, and barriers



```
SampleScene*
  Main Camera
  Directional Light
  pit exit
  straight 45
  curve 1
  curve 1b
  straight 46
  curve 2
  curve 2b
  curve s1
  curve s2
  curve s3
  curve s4
  straight 47
  curve s5
  straight 48
  curve s6
  curve s7
  curve s8
  curve s9
  straight 50
  curve dunlop1
  curve dunlop2
  curve dunlop3
  straight 51
  curve degnar1
  straight 52
  curve degnar2
  straight 53
  straight 53b
  curve 61
  curve 62
  curve hairpin
  straight 54
  curve 64
  straight 55
  curve 200R
  straight 56
  curve 66
  curve 67
  straight 57
  curve spoon1
  straight 58
  curve spoon2
  curve spoon3
  curve spoon4
  curve straight1
  curve straight 2
  curve straight 3
  curve 75
  straight 61
  curve crossover
  curve 130R
  curve 78
  straight 62
  curve casio 1
```

TracksCAD

Computer-Aided Design tool for racing game tracks

curve crossover

LEFT_TURN

Name	curve crossover
Type	LEFT_TURN
Arc (°)	14
Radius (m)	300
Ending radius (m)	300
Slope (%)	0
Banking start (°)	0
Banking end (°)	0

Pit entry ☐ Pit end ☐
Pit start ☐ Pit exit ☐

Misc
- Left Border
- Left Side
- Left Barrier
- Right Border
- Right Side
- Right Barrier

Track settings

Name	Karwada
Authors	A. Sumner, E. M. Jaeger

A well-known, fast, curvy circuit in Japan

Loading track e-track 1.json data...

Figure 4.7: Screenshot of the tool while editing a segment

4.3 Automatic Track Closure System

When working on a new track, the designer will get to a point where only one segment is left to complete the job. The choice of the closing piece is decisive for the correctness of a circuit, as it must be perfectly aligned with the start line and, if not done properly, it may generate unexpected behaviour when racing. For instance, if the starting and ending points of a track are too far from each other, cars would fall under the road when encountering the discontinuity caused by this design mistake. Another example consists on placing the last point at a largely greater height with respect to the beginning of the track. In such case vehicles would perform a severe jump, falling on the road below, and potentially undergo a lot of unnecessary damage.

The solution proposed in TracksCAD is to offer the designer a button that, if pressed, finds and adds automatically the segment which closes the current track. To do so, the program exploits a useful property of segments: given two points $\langle \vec{p_s}, \vec{p_e} \rangle$ and two vectors $\langle \vec{t_s}, \vec{t_e} \rangle$, it exists at most one segment connecting the two points, and having as first and final tangents the two vectors, respectively. This tells us that not every set of points and tangents allows the creation of a segment but, if it does, then the segment is unique. Evaluating the existance algorithmically is not a trivial task, although it is possible to check it qualitatively by looking at the points and tangents from a graphical point of view. On the other hand, we will now see how to find the parameters that make the connecting segment. In our specific case we are talking about closing a track, therefore $\langle \vec{p_s}, \vec{p_e} \rangle$ is the tuple consisting, respectively, by the last and first points of the circuit, while elements of $\langle \vec{t_s}, \vec{t_e} \rangle$ represent the corresponding tangent vectors.

First of all, it is important to understand whether the closure can only be obtained using a turn, or if the goal can be achived also with a simple straight. To verify what kind of segment we need to build, we can check the alignment of $\vec{t_s}$ and $\vec{t_e}$. If the two vectors are aligned with the straight line connecting the two points, then a straight segment is sufficient to close the circuit, and its length is trivially computed. On the other hand, if that is not the case, we need to find the values of START RADIUS, END RADIUS, and ARC. This operation cannot be performed simply by solving an equation, because of the iterative procedure necessary for finding the points in variable radius turns (Algorithm 2). However, an efficient way of approximating these parameters can be found considering gradient method algorithms such as the Multivariate Newton's method [6] and Gradient Descent method [15].

Our problem can be reformulated as finding the set of input radiuses

$\vec{\boldsymbol{r}} = \langle r_s, r_e \rangle$ which minimizes the function:

$$d(\vec{\boldsymbol{r}}) = |\vec{p_e} - (f(\vec{\boldsymbol{r}}, \theta) + \vec{p_s})| \tag{4.2}$$

where $\vec{p_e}$ is the target endpoint of the turn (i.e. the start point of the track), $\vec{p_s}$ is its start point (i.e. the current endpoint of the track), $f(\vec{\boldsymbol{r}}, \theta)$ is the function that outputs the end point of the turn segment built with $\vec{\boldsymbol{r}}$, and θ is the angle between tangents $\vec{t_s}$ and $\vec{t_e}$. By minimizing function $d(\vec{\boldsymbol{r}})$, we can retrieve the set of radiuses that, together with parameter θ, make a turn segment which starts at $\vec{p_s}$, ends at $\vec{p_e}$, and has starting and ending tangents equal to $\vec{t_s}$ and $\vec{t_e}$, respectively.

The idea behind the mentioned gradient methods is to make, at each iteration, small steps in the direction of the local slope of the function, eventually finding its minimal point. Newton's method for multivariate functions would be the fastest solution, as it searches for the optimum in the direction of maximum slope [6], therefore its update rule is:

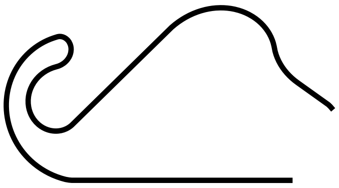
$$\vec{x}_{i+1} = \vec{x}_i - \alpha_i H(\vec{x}_i)^{-1} \nabla f(\vec{x}_i) \tag{4.3}$$

This additional information, however, requires the invertibility of the Hessian matrix $H(x_t)$, which in our case is a condition often not satisfied. As a result of that, we use a slightly slower algorithm like the Gradient method, which only uses the function gradient and does not require any invertibility condition [15]:

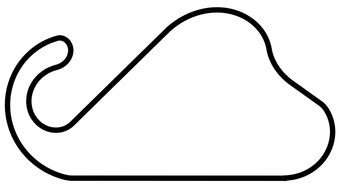
$$\vec{x}_{i+1} = \vec{x}_i - \alpha_i \nabla f(\vec{x}_i) \tag{4.4}$$

For the computation of the gradient, we use the *forward finite difference* formula [16] adapted for the multivariate case.

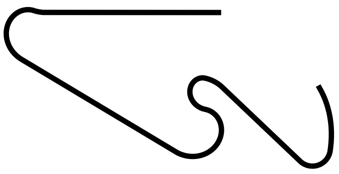
The resulting system is able to create precisely and with quite good performances a closing segment, whenever there exist one (Figure 4.8). If the existance is not satisfied, then the algorithm finds a tentative closure with the shortest distance between the endpoint and the target.



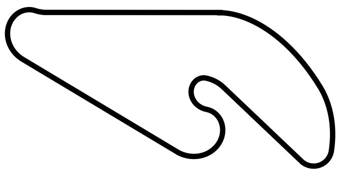
(a) Custom track #1 - open



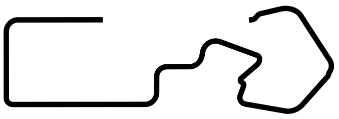
(b) Custom track #1 - closed



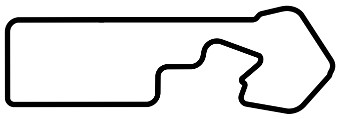
(c) Custom track #2 - open



(d) Custom track #2 - closed



(e) Custom track #2 - open



(f) Custom track #3 - closed

Figure 4.8: Examples of automatic track closures. All have been found in less than 300ms (5×10^3 iterations), and with a tolerance of less than 0.5m.

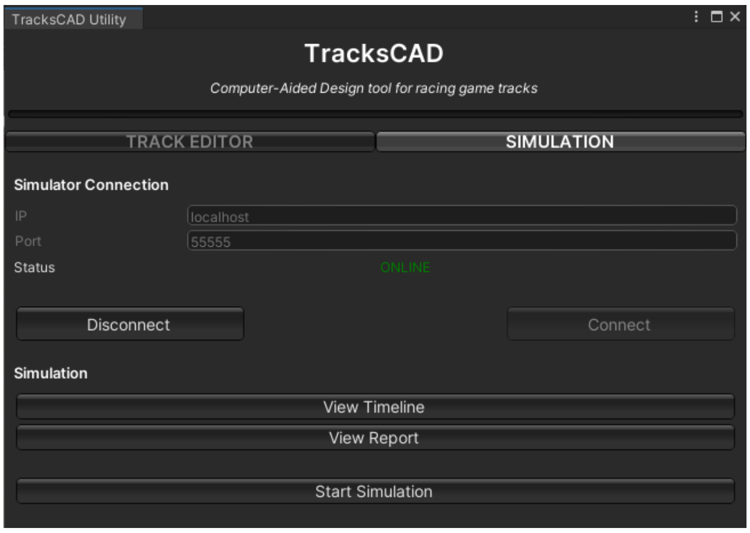


Figure 4.9: Simulation window

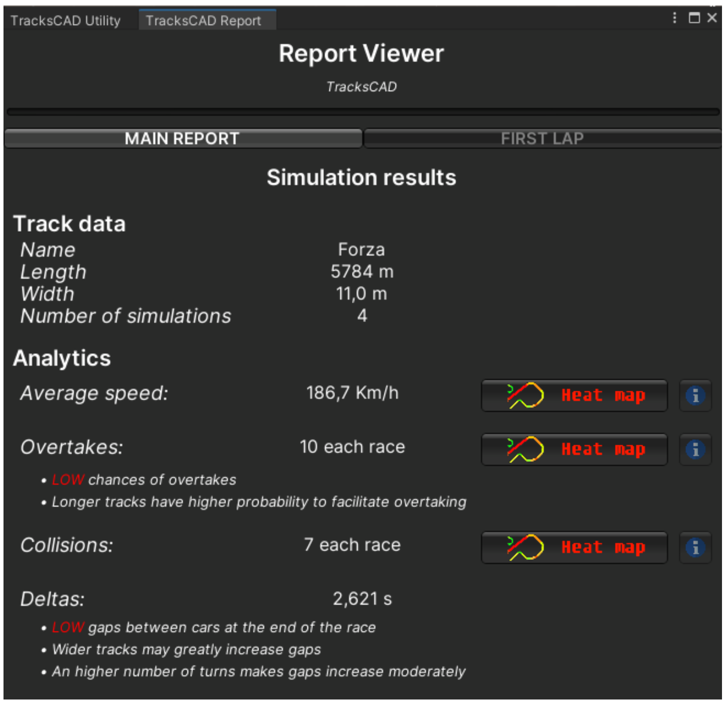
4.4 Track Evaluation

As discussed previously in this chapter, another important portion of the work for this thesis resides in the development of a system that interacts with the simulator and gives the designer useful information about the track dynamic characteristics. The primary requirements for being able to run simulations are:

- TracksCAD tool is connected to the TracksCAD Simulation Server.
- A track is loaded in the tool.
- All segments have unique names.
- The track has a complete pit lane (see 4.2.2).

Once the above specifications are satisfied, the designer can use the custom window shown in Figure 4.9 to start the execution of races and retrieving the corresponding reports. As soon as the simulations are ended and the logs are returned to the Unity tool, their data is aggregated.

Button *View Report* can then be pressed to open a new piece of interface which displays the results of the races (Figures 4.10 and 4.11), in conjunction with the outcomes of the analyses performed by the tool on such data. In the following subsections we describe how these aggregation and analysis procedures are performed for each of the metrics we considered in the race logs.



Report Viewer

TracksCAD

Simulation results

Track data

Name	Forza
Length	5784 m
Width	11,0 m
Number of simulations	4

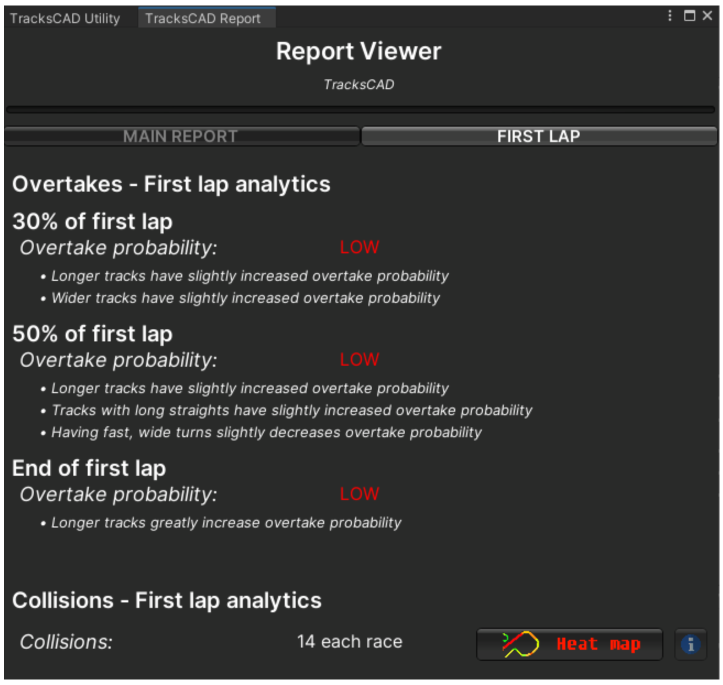
Analytics

Average speed:	186,7 Km/h
Overtakes:	10 each race

- LOW chances of overtakes
- Longer tracks have higher probability to facilitate overtaking

Collisions:	7 each race
Deltas:	2,621 s

- LOW gaps between cars at the end of the race
- Wider tracks may greatly increase gaps
- An higher number of turns makes gaps increase moderately



Report Viewer

TracksCAD

Overtakes - First lap analytics

30% of first lap

Overtake probability: LOW

- Longer tracks have slightly increased overtake probability
- Wider tracks have slightly increased overtake probability

50% of first lap

Overtake probability: LOW

- Longer tracks have slightly increased overtake probability
- Tracks with long straights have slightly increased overtake probability
- Having fast, wide turns slightly decreases overtake probability

End of first lap

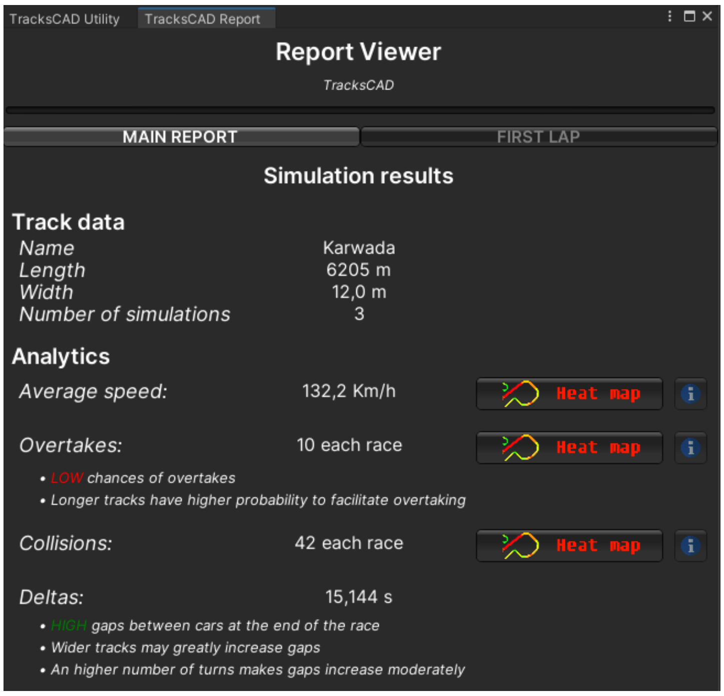
Overtake probability: LOW

- Longer tracks greatly increase overtake probability

Collisions - First lap analytics

Collisions: 14 each race

Figure 4.10: Simulation report window of track Forza



Report Viewer

TracksCAD

Simulation results

Track data

Name	Karwada
Length	6205 m
Width	12,0 m
Number of simulations	3

Analytics

Average speed: 132,2 Km/h

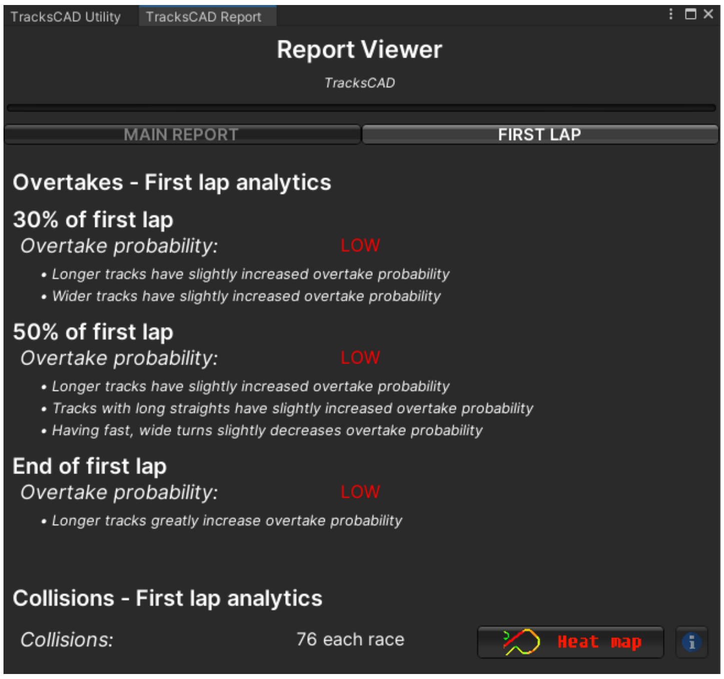
Overtakes: 10 each race

- *LOW chances of overtakes*
- *Longer tracks have higher probability to facilitate overtaking*

Collisions: 42 each race

Deltas: 15,144 s

- *HIGH gaps between cars at the end of the race*
- *Wider tracks may greatly increase gaps*
- *An higher number of turns makes gaps increase moderately*



Report Viewer

TracksCAD

Overtakes - First lap analytics

30% of first lap

Overtake probability: LOW

- *Longer tracks have slightly increased overtake probability*
- *Wider tracks have slightly increased overtake probability*

50% of first lap

Overtake probability: LOW

- *Longer tracks have slightly increased overtake probability*
- *Tracks with long straights have slightly increased overtake probability*
- *Having fast, wide turns slightly decreases overtake probability*

End of first lap

Overtake probability: LOW

- *Longer tracks greatly increase overtake probability*

Collisions - First lap analytics

Collisions: 76 each race

Figure 4.11: Simulation report window of track Karwada

4.4.1 Average Speed

The average speed of the track is computed by looking at the timeline of each race. For each time frame, the tool gets the sum of speed values of all drivers in the grid and adds it to the sum of speeds computed so far. Once all logs are read, this sum can be divided by the total number of samples considered, that is:

$$N_{samples} = \sum_{i=1}^{n_{logs}} T(i) \times N_{bots}(i) \tag{4.5}$$

where $T(i)$ is the number of elements in the i-th race timeline, and $N_{bots}(i)$ is the amount of *bots* that competed in the i-th race. The same operation is also performed so to have the average speed of each segment individually.

4.4.2 Collisions analysis

For what concerns collisions, the tool counts the number of incidents both throughout the entire racetrack, and for each segment, by way of the dedicated COLLISIONS section of the logs. The value is then divided by the number of simulations, returning the average amount of collisions during one race.

These operations are performed also considering data exclusively within the 30%, 50% and 100% of the first lap.

4.4.3 Overtaking dynamics analysis

This type of analysis consists in finding how many overtakes took place in total, and on each part of the track. The aggregation is performed summing up the number of overtakes and then dividing them by the number of races performed. Similarly to the aggregation of collisions data, the resulting value represents the average number of position swaps during one race.

Another operation concerning overtakes consists on the analysis of the number of positions gained by the drivers during a race. During this phase we compare the *bots* order in a specific time frame with respect to the starting grid order, and we count the number of positions gained or lost by each *bot*. For each number of positions, we then count the number of *bots* that gained or lost them, as shown in Figure 4.12. While computing the average of this metric gives us no useful information (it is always equal to zero, as for each driver gaining a position one loses it), the variance and skewness are used to classify the track as one in which there is either a

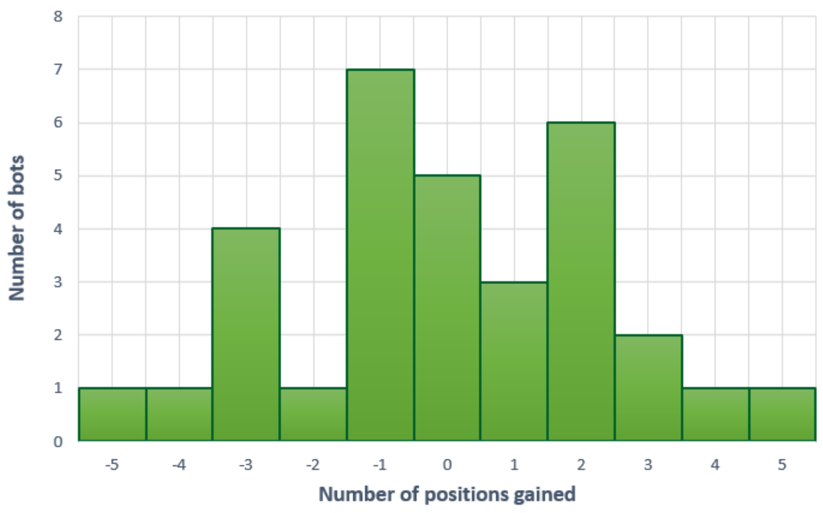


Figure 4.12: Position variations recorded across 4 races on Aalborg *track*

high or low probability of overtakes. This is done with the help of the work performed in [18], in which K-Means clustering algorithm [17] was used to identify the two mentioned classes on a set of TORCS circuits. We compare the euclidean distance of our set of $\langle var,\ skew\rangle$ with the centroid positions of the two clusters, associating the current track with the nearest one.

The same analysis is executed considering data exclusively within the 30%, 50% and 100% of the first lap.

4.4.4 Analysis of Time Gaps

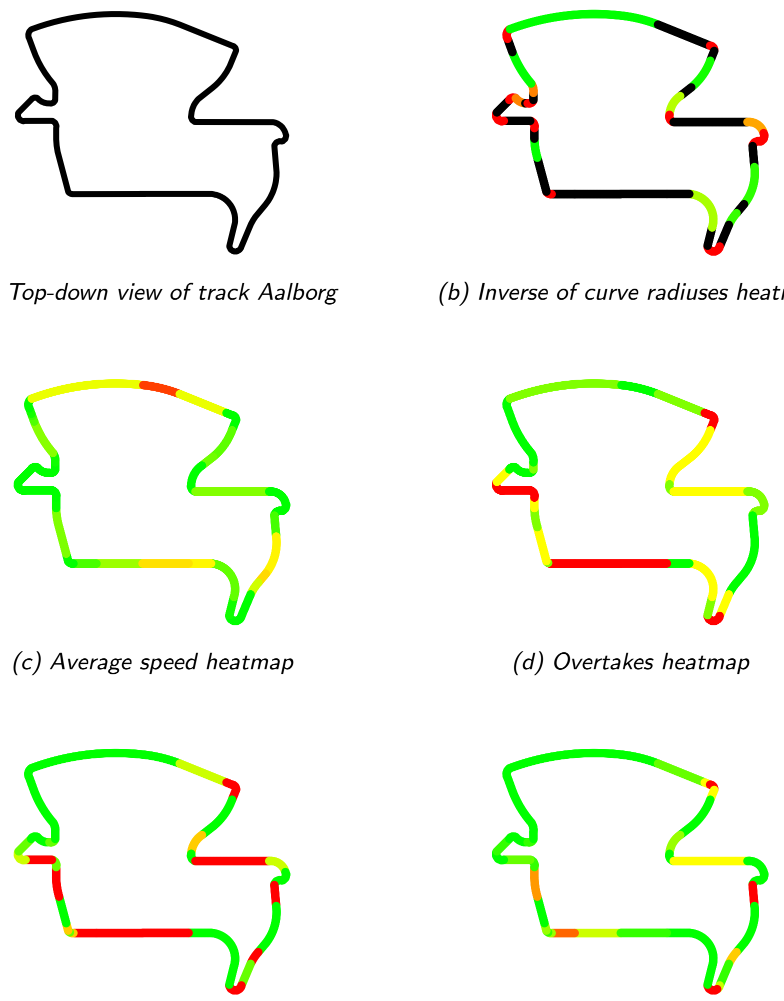
At the end of each lap the simulator records the time gaps between each *bot* and the one preceeding it. The analysis of such gaps consists on performing similar aggregations to those discussed for the gained positions. The main difference is that this time each bar of the graph represents the number of gaps found in the specified time interval.

Another computed metric is the total time gap between the first and last drivers, considered at the end of the first lap and at the end of the race.

Tracks are classified also with respect to the average, variance, and skewness of their time gaps. As a result, a circuit can be considered either with low or high time differences between players.

4.4.5 Heatmaps

TracksCAD tool takes advantage of the 3D representation of the track built in Unity to show designers some of the metrics computed previously in the



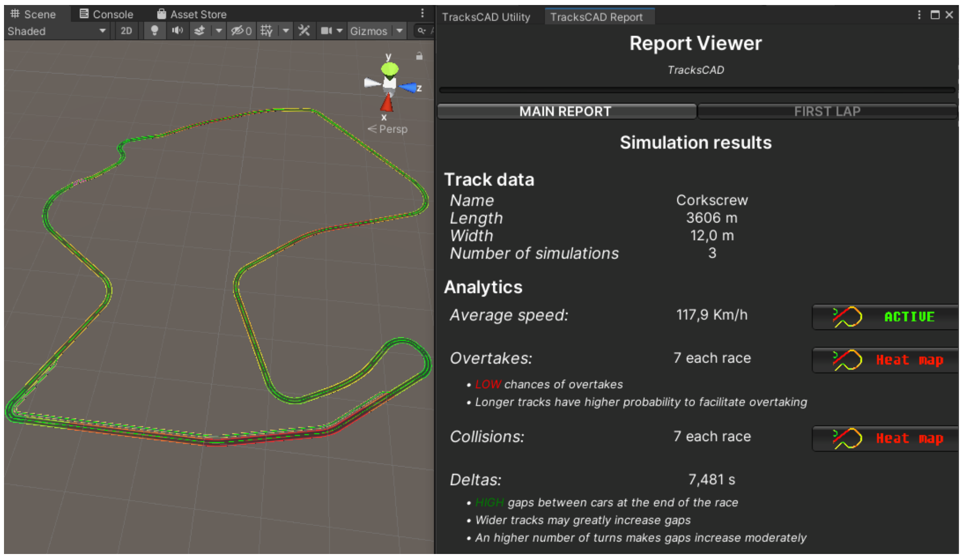
(a) Top-down view of track Aalborg *(b) Inverse of curve radiuses heatmap*

(c) Average speed heatmap *(d) Overtakes heatmap*

(e) Collisions heatmap *(f) Collisions during first lap heatmap*

Figure 4.13: Heatmaps computed from simulations on track Aalborg

form of heatmaps. This color-coded view can be activated by pressing the dedicated buttons in the GUI, and some examples are shown in Figure 4.13. The first heatmap (Figure 4.13b) shows, for each segment, the inverse of its curve radius. The reason behind using the inverse instead of the actual turn radius resides in the fact that in the tool straights have a default radius value of zero instead of infinity. Figure 4.14 shows how the same type of information is actually displayed to the user by coloring the 3D model of each segment coherently with the corresponding heatmap value. The other heatmaps in Figure 4.13 allow to graphically see the metrics extracted from the simulations after the aggregations performed using the previously discussed techniques.



Report Viewer

TracksCAD

Simulation results

Track data

Name	Corkscrew
Length	3606 m
Width	12,0 m
Number of simulations	3

Analytics

Average speed: 117,9 Km/h

Overtakes: 7 each race

- *LOW chances of overtakes*
- *Longer tracks have higher probability to facilitate overtaking*

Collisions: 7 each race

Deltas: 7,481 s

- *HIGH gaps between cars at the end of the race*
- *Wider tracks may greatly increase gaps*
- *An higher number of turns makes gaps increase moderately*

Figure 4.14: Screenshot of the tool while viewing heatmaps of track Corkscrew

4.5 Replaying Races

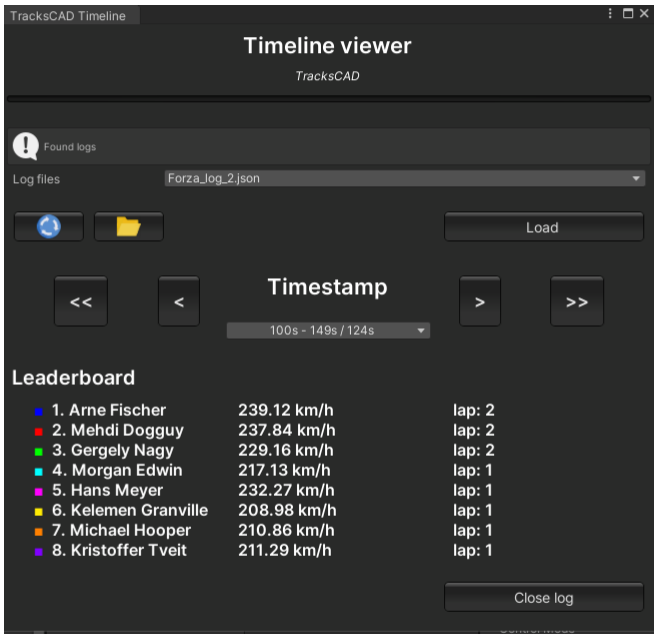
After simulating races on the tracks we designed, it may be convenient to actually see their evolution over time, and check how drivers behaved while competing against each other. Figure 4.15 shows how TracksCAD allows to review simulations, informing the user about each *bot* position in the leaderboard, its speed, and the lap it is currently performing. The Unity 3D scene is also populated with small coloured spheres, each one representing a driver that attended the chosen race. The window devoted to the navigation of the timeline is the timeline control panel, from which we are able to navigate back and forth time frames, analyzing snapshots of the race.

From a technical point of view, the implementation of this system is quite straightforward. The tool searches inside the `/StreamingAssets/Logs` folder for existing logs, and displays them to the user with a dropdown menu. The two requirements for opening a specific *log* are:

- A track is loaded in the tool.
- The current track has the same name as the one defined in the *log*.

The tool simply reads the content of the TIMELINE section inside the selected document, and finds almost all the necessary information already ready to be displayed to the user. The only computation it performs concerns finding the right 3D coordinates in which to place the drivers spheres. In order to do this, it gets from the current timestamp the segment name of each driver, and then it computes the coordinates using the value of the distance from the start of that segment.

Inspecting simulations is especially useful for exhibiting anomalies and atypical behaviours of the AI. It may also help understanding the reasons behind certain values of the extracted metrics. For instance, if we notice that during the simulations the game registered an exceptionally large number of collisions, we could inspect the timeline and discover that this irregularity came indeed from an incorrect closing of the track, which caused cars to crash (therefore stopping their movement) whenever crossing the start line.



Timeline viewer

TracksCAD

Log files: Forza_log_2.json

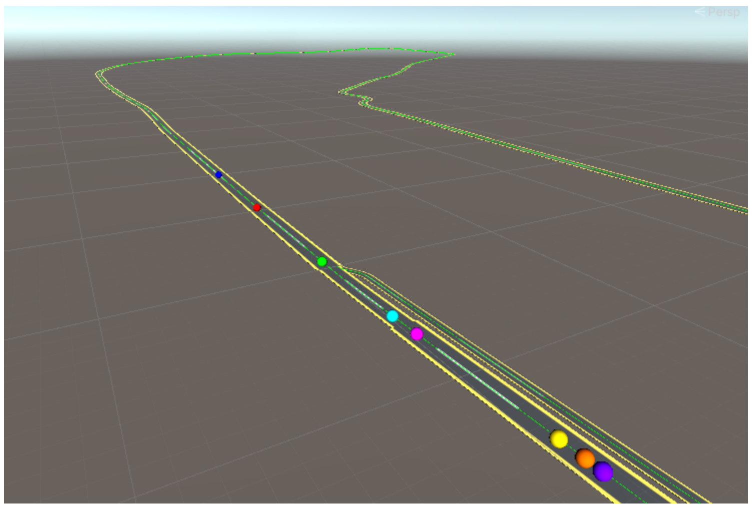
Timestamp

100s - 149s / 124s

Leaderboard

■ 1. Arne Fischer	239.12 km/h	lap: 2
■ 2. Mehdi Dogguy	237.84 km/h	lap: 2
■ 3. Gergely Nagy	229.16 km/h	lap: 2
■ 4. Morgan Edwin	217.13 km/h	lap: 1
■ 5. Hans Meyer	232.27 km/h	lap: 1
■ 6. Kelemen Granville	208.98 km/h	lap: 1
■ 7. Michael Hooper	210.86 km/h	lap: 1
■ 8. Kristoffer Tveit	211.29 km/h	lap: 1

(a) Timeline control panel



(b) View of the scene while replaying a race

Figure 4.15: Timeline Navigation System screenshots

Chapter 5

Estimation of Dynamic Track Characteristics from Topology Metrics

In this chapter we try to understand whether it is possible to use Machine Learning techniques to predict some of the dynamic metrics that we usually extract during simulations, by using a subset of the most relevant topological features of tracks (length, width, number of turns etc.).

In Section 5.1 we briefly discuss why it is useful to perform this type of study. After that, we present the procedure carried out to extract the values we used to build our dataset and train our models, and the list of metrics we picked to be predicted.

Section 5.2 shows how we performed the training of the models, their testing and, if necessay, how we chose their hyper-parameters. We do not dive into the details of how each ML technique works or how it is implemented, as this would diverge too much from the scope of this thesis.

In Section 5.3 the resulting performances of each trained model are discussed by presenting the values of the most relevant evaluation metrics.

5.1 Metrics and Data Extraction

Now that we have a working system that allows us to run simulations on tracks we create, we want to understand whether it is possible to find accurate models that are able to predict some of the dynamic characteristics that we usually obtain from races, starting from the topological features of the track in use. If this was possible it would allow designers to work on circuits faster, limiting the number of simulations to the sole purpose of confirming results coming from prediction models. Furthermore, if the performance of such models turns out to be sufficiently good, it would be possible to extract high-level knowledge about the relations between the topological features of the track and the racing dynamics emerging when competing. For instance, we could discover that a particular set of characteristics plays a significant role in the amount of collisions taking place just after the start of the race. With such knowledge we would be able to design circuits knowing what decisions to make as to decrease (or increase) the probability of accidents occurring during the first phase of races.

The dynamic metrics that we would like to be able to estimate are:

- **Average speed:** the average speed across all the bots throughout the entire simulation. As to avoid this value to be altered by first lap dynamics such as collisions, and to avoid the starting acceleration phase during which all cars have near-zero speed, we exclude from the computation of this value the entire first lap of each race.
- **Number of overtakes (end of race):** total number of overtakes which took place during the entire race.
- **Number of overtakes (first lap):** total number of overtakes, this time computed considering only three distinct phases of the first lap of the race: at the 30%, 50% and 100% of the first lap.
- **Position changes from starting grid (end of race):** sum of the absolute values of positions gained/lost by each driver at the end of the race, with respect to their position on the starting grid.
- **Position changes from starting grid (first lap):** sum of the absolute values of positions gained/lost by each driver at the end of the 30%, 50% and 100% of the first lap, with respect to their position on the starting grid.
- **Time gap from first to last driver (end of race):** total time difference (gap) from first to last driver at the end of the race.
- **Time gap from first to last driver (first lap):** total time difference from first to last driver at the end of the first lap.

All values concerning time gaps, overtakes, and collisions are normalized with respect to the length of the circuit, using the formula

$$\log \frac{1+x}{length} \tag{5.1}$$

where x is the target we are taking into consideration and for which we want to find an estimator.

Between all the available track topology features, we consider the followings as the most relevant for our purpose:

- **Track length**
- **Track width**
- **Number of straights**
- **Number of turns**
- **Average Track elevation:** average between all track segments of their mean elevation.
- **Inverse of turn radiuses (Average, Variance):** average and variance, between all track segments, of the inverse of their mean turn radius.

The fitting procedure of the regression models starts by picking a large enough number of existing circuits on which to perform the simulations, that will constitute our basic dataset. The resulting list of tracks (Table 5.1) has elements coming from different sources and, if necessary, they have been slightly adapted to fit the Speed Dreams representation requirements. Some of them are original SD racetracks, others come from the game TORCS, while four of them have been created by an automatic track generation tool developed by Politecnico di Milano [14, 2]. The actual dataset is then created by running four races on each of the selected tracks, and writing on a *json* file both inputs (the track topology features) and outputs (the dynamic metrics extracted from simulations).

Before we can start fitting the models, we need to prepare the data by performing standardization and normalization. The former procedure consists on rescaling the inputs in such way to making it have zero mean and unitary variance, while the latter consits on applying Formula 5.1 to the outputs. This last operation makes the targets independent from the length of the circuit and rescales them into a logarithmic scale, hence reducing the impact of outliers.

Track name	Source
Aalborg	Speed Dreams
Alpine-1	TORCS
Alpine-2	TORCS
Berhet-hill	Trackgen
Brondehach	Speed Dreams
Chemisay	Speed Dreams
Corkscrew	Speed Dreams
Espie	Speed Dreams
E-Track-1	TORCS
E-Track-2	TORCS
E-Track-3	TORCS
E-Track-4	TORCS
E-Track-6	TORCS
Forza	Speed Dreams
Hidden Valley	Speed Dreams
Karwada	Speed Dreams
Noye-hill	Trackgen
Ruudskogen	Speed Dreams
Spring	TORCS
Volcan-mountain	Trackgen
Watorowo-city	Trackgen

Table 5.1: List of tracks used to create the dataset

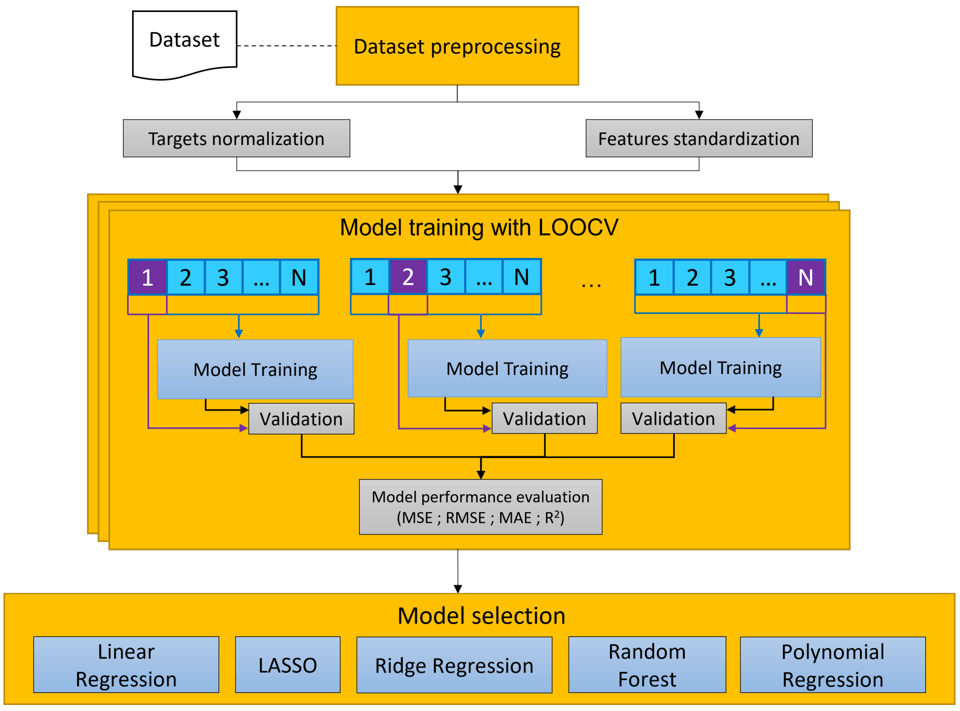


Figure 5.1: Model estimation procedure

5.2 Training the Models

After having prepared the data, we can start training the models that will predict our output targets. We first try to understand what are the most relevant topology features that can be exploited to predict each output. To do so, it is useful to visualize graphically the input-output space, and check whether there are significant linear or polynomial relationships. If we find some, we can start fitting models using as inputs the ones with the most promising relations. Otherwise, we include all inputs and then try feature selection techniques later on [9].

With the preselected topology characteristics, we fit linear, LASSO, Ridge, polynomial, and random forest regression models to our data [17]. The training procedure is schematically explained in the center block of Figure 5.1, and it is performed by running Leave-One-Out Cross Validation [17]. For each type of regression model, we subdivide the evaluation into N steps (N being the cardinality of the dataset). At each step, we train a model of the current type on the basis of $N-1$ data samples, while performing a prediction on the remaining one.

Eventually we can determine the overall performance of the type of regression model we have trained by computing the evaluation metrics on its predictions. Given the relatively small dataset, LOO Cross Validation technique allows us to find a more accurate and generalized measure of performance, mitigating the risk of overfitting the data.

In the case of LASSO and Ridge regression, the entire training procedure is repeated multiple times, and with different values of λ.

5.3 Results

The performance of each model is represented by the evaluation metrics we choose to compute on its predictions. The RMSE [8] and R-squared (R^2) [17] values are the operators we use for this study. The former is estimated as the average of the values scored at each iteration of the Leave-One-Out algorithm, while the latter is computed at the end of the cross validation procedure as it needs at least two predictions to be found. For Ridge and LASSO regression models, we also compute the optimal λ value between those tried during the training process.

Increasing the order of the polynomial used for regression from linear (n=1) to quadratic (n=2) did not introduce any significant improvement. On the contrary, it always resulted in worse predictions than the linear case, therefore such models have not been considered in the following analyses.

Table 5.2 shows that there is a quite good relationship between the average speed assumed by the drivers throughout the races and the length of the track, its turns number and average radius. Ridge regression performs particularly well having the lowest RMSE and a R^2 over 0.6. Given that the mean average speed across all tracks is 130.66 *km/h*, this results in RMSE indicating an error of 8.42% with respect to that value. These results are coherent with the expectations as, intuitively, tracks with less changes in direction and wider turns are more likely to allow cars to reach higher speeds. In the same way, having a long circuit with a small number of turns means having longer straights, which is also a relevant factor that determines the average speed drivers may adopt when racing.

We analyze now the evaluation metrics for the prediction models concerning the total time difference from first to last driver, considering as inputs the track length, number of turns, average and variance of radiuses. The best fraction of variance expressed correctly by the models (R^2) is found using linear regression, both for the metric computed at the end of the race, and that computed at the end of the first lap. A behaviour that repeats itself in the majoriy of cases is that first lap dynamics seem easier to perdict, showing better coefficients of determination R^2 and lower average errors.

Model	**RMSE**	R^2	**Avg. Error**
Linear regression	13.2 ± 11.78	0.41	10.07%
LASSO regression ($\lambda = 1.459$)	11.31 ± 8.59	0.62	8.63%
Ridge regression ($\lambda = 3.99$)	11.04 ± 8.86	0.62	8.42%
Random Forest	11.38 ± 8.83	0.59	8.68%

Table 5.2: Performance metrics of prediction models for estimating average speed from length, number of turns, and average of inverse radius

In Table 5.4 we find the performances resulting after training the models with the goal of predicting the number of overtakes, both at the end of the race and in three phases of the first lap. In this case the selected features do not allow to estimate very well the target. Indeed, we can see that the R^2 value is negative in almost every case. As to better understand what this means, we can review the formula to compute this metric:

$$R^2 = 1 - \frac{RSS}{TSS} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{5.2}$$

where RSS (Residual Sum of Squares) represents how much the model deviates from the real values, and TSS (Total Sum of Squares) represents the sum of the quadratic errors between the real values and the constant model. Having a negative R^2 then means that our regression performs worse than by just using the average number of overtakes as prediction ($RSS > TSS$). These findings suggest that even for Random Forest models, which provide an average error of less than 10% of the mean value, the analysis of the R^2 value actually shows that the selected features are not predictive of the real racing behaviour, and therefore that the variation of the target is not well explained by the models we chose to fit.

The best performances can be found by estimating the number of absolute position changes using as inputs the track length, width, number of straight and turns, the average elevation and the radius variance. Table 5.5 presents such results, with R^2 values showing that a significant fraction of the target variance is expressed by the adopted models, particularly for first lap dynamics. Also, looking at the set of best models of the considered output metrics, the worst Root Mean Square Error consists in little more than 4% of the corresponding average value.

In Table 5.6 we report the p-values resulting from computing the F-

Model	RMSE	R^2	Avg. Error
Linear regression	0.18 ± 0.12	0.24	9.06%
LASSO regression ($\lambda = 0.004$)	0.19 ± 0.13	0.18	9.56%
Ridge regression ($\lambda = 0.217$)	0.19 ± 0.12	0.23	9.56%
Random Forest	0.21 ± 0.15	-0.36	10.57%

(a) First-to-last time gap - end of race

Model	RMSE	R^2	Avg. Error
Linear regression	0.13 ± 0.1	0.32	5.3%
LASSO regression ($\lambda = 0.003$)	0.14 ± 0.11	0.24	5.7%
Ridge regression ($\lambda = 1.428$)	0.14 ± 0.11	0.23	5.7%
Random Forest	0.15 ± 0.12	0.03	6.11%

(b) First-to-last time gap - end of first lap

Table 5.3: Performance metrics of prediction models for estimating first-to-last time gap from track length, number of turns, average and variance of radiuses

statistic for the evaluation of the overall significance of each model. Each column is a track topology feature, while rows are the target metrics to predict. The F-statistic tests whether the selected input feature is representative of the considered target, and obtaining a smaller p-value for such test means that the corresponding ⟨*input, output*⟩ metrics have a more significant relation with each other.

Looking at the overall results found in this chapter we can say that, indeed, it is be possible to predict some of the dynamic characteristics that are usually extracted from races using a subsets of track topology features. Such estimations can be considered as hints of what values the real metrics might assume during the actual simulations, allowing designers to use them to roughly estimate the quality and peculiarities of their tracks, and without the need of running numerous time consuming races in the simulator.

Model	**RMSE**	R^2	**Avg. Error**
Linear regression	0.22 ± 0.32	-0.64	8.72%
LASSO regression ($\lambda = 0.034$)	0.23 ± 0.32	-0.64	9.12%
Ridge regression ($\lambda = 6.229$)	0.2 ± 0.27	-0.22	7.93%
Random Forest	0.2 ± 0.26	-0.15	7.93%

(a) Number of overtakes - end of race

Model	**RMSE**	R^2	**Avg. Error**
Linear regression	0.21 ± 0.27	-0.17	8.52%
LASSO regression ($\lambda = 0.027$)	0.21 ± 0.28	-0.17	8.52%
Ridge regression ($\lambda = 2.961$)	0.19 ± 0.23	0.09	7.71%
Random Forest	0.19 ± 0.23	0.09	7.71%

(b) Number of overtakes - 30% of first lap

Model	**RMSE**	R^2	**Avg. Error**
Linear regression	0.2 ± 0.26	-0.25	8.36%
LASSO regression ($\lambda = 0.03$)	0.2 ± 0.25	-0.18	8.36%
Ridge regression ($\lambda = 3.229$)	0.19 ± 0.23	-0.03	7.95%
Random Forest	0.19 ± 0.22	0.1	7.95%

(c) Number of overtakes - 50% of first lap

Model	**RMSE**	R^2	**Avg. Error**
Linear regression	0.21 ± 0.3	-0.84	9.36%
LASSO regression ($\lambda = 0.036$)	0.2 ± 0.3	-0.84	8.92%
Ridge regression ($\lambda = 5.171$)	0.2 ± 0.27	-0.58	8.92%
Random Forest	0.18 ± 0.23	0.17	8.02%

(d) Number of overtakes - end of first lap

Table 5.4: Performance metrics of prediction models for estimating number of overtakes from length and width of track

Model	RMSE	R^2	Avg. Error
Linear regression	0.09 ± 0.08	0.61	3.67%
LASSO regression ($\lambda = 0.013$)	0.09 ± 0.09	0.55	3.67%
Ridge regression ($\lambda = 1.056$)	0.09 ± 0.09	0.62	3.67%
Random Forest	0.1 ± 0.1	0.36	4.08%

(a) Number of absolute position changes - end of race

Model	RMSE	R^2	Avg. Error
Linear regression	0.11 ± 0.09	0.53	4.29%
LASSO regression ($\lambda = 0.01$)	0.11 ± 0.08	0.62	4.29%
Ridge regression ($\lambda = 2.115$)	0.1 ± 0.07	0.69	3.9%
Random Forest	0.11 ± 0.09	0.38	4.29%

(b) Number of absolute position changes - 30% of first lap

Model	RMSE	R^2	Avg. Error
Linear regression	0.1 ± 0.07	0.69	3.93%
LASSO regression ($\lambda = 0.009$)	0.08 ± 0.06	0.81	3.14%
Ridge regression ($\lambda = 1.222$)	0.09 ± 0.06	0.78	3.54%
Random Forest	0.11 ± 0.11	0.31	4.32%

(c) Number of absolute position changes - 50% of first lap

Model	RMSE	R^2	Avg. Error
Linear regression	0.07 ± 0.06	0.82	2.81%
LASSO regression ($\lambda = 0.007$)	0.08 ± 0.07	0.8	3.21%
Ridge regression ($\lambda = 0.486$)	0.07 ± 0.06	0.82	2.81%
Random Forest	0.09 ± 0.11	0.39	3.61%

(d) Number of absolute position changes - end of first lap

Table 5.5: Performance metrics of prediction models for estimating number of absolute position changes from track length, width, number of straights and turns, average elevation, and radius variance

	Length	Width	N. Straight	N. Turns	Avg. Radius	Var. Radius	Avg. Elev.
Avg. Speed	0.28	0.24	0.33	0.69	1.20×10^{-6}	2.39×10^{-4}	0.86
Overtakes:							
• End of race	0.04	0.08	0.05	0.04	0.88	0.73	0.39
• First Lap - 30%	2.14×10^{-3}	0.17	5.43×10^{-3}	3.84×10^{-3}	0.71	0.62	0.23
• First Lap - 50%	1.97×10^{-3}	0.14	0.01	2.78×10^{-3}	0.71	0.54	0.23
• First Lap - 100%	4.74×10^{-3}	0.07	0.02	8.25×10^{-3}	0.67	0.48	0.24
Positions Gained:							
• End of race	4.48×10^{-7}	0.12	0.03	2.34×10^{-5}	0.37	0.14	0.18
• First Lap - 30%	2.95×10^{-6}	0.14	7.66×10^{-3}	1.10×10^{-5}	0.21	0.11	0.27
• First Lap - 50%	5.25×10^{-7}	0.17	0.02	4.75×10^{-6}	0.24	0.09	0.23
• First Lap - 100%	2.10×10^{-7}	0.16	0.02	6.65×10^{-6}	0.25	0.1	0.1
First-To-Last Delta:							
• End of race	0.23	0.34	0.66	0.69	0.07	0.02	0.18
• First Lap - 100%	0.02	0.2	0.98	0.13	0.05	0.01	0.37

Table 5.6: p-values of the F-Statistic for each ⟨feature, target⟩ tuple

Chapter 6

Conclusions and Future Work

In this thesis we presented the design and development of TracksCAD, a Computer-Aided Design (CAD) tool that gives racing game designers the ability to create, edit, and run simulations on their tracks. Moreover, we implemented a system that performs a set of different analyses on data extracted from those simulations, therefore helping the designer to better understand the dynamics that can arise while racing on the developed circuits. We discussed the necessary changes to retrieve that data from the game we chose to use (Speed Dreams), and the meaning of each considered metric. The tool, which was developed as a plugin to the well-known Unity editor software, allows also to review races performed into the simulator directly inside a three-dimensional scene, and it is able to perform automatically the closing procedure of tracks.

One of the requirements of the system was that its architecture had to be as independent as possible from the game used to run simulations. This was achived thanks to the decoupling of the functional block concerning the editing of the circuit from that designated to perform races, and thanks to the definition of a more generalized track representation which is still easily convertible to the one required by the simulator in use.

Eventually, we have shown that it is possible to exploit Machine Learning techniques to reduce the number of simulations to be performed by directly estimating the metrics generally extrapolated from races, using as inputs the topological features of the designed tracks. As we have discussed in the thesis, their dynamic characteristics can indeed be related to the set of structural choices a designer usually makes during development. We have seen that we can estimate quite accurately the average speed that drivers will adopt during the laps of a race by looking at the number and types of turns of the track. The error we make during such predictions is all in all acceptable, and it is suitable for giving us an idea about what characteristics make our circuit unique and whether we have area for improvements. Even more precisely we can compute a reasonable value for the number of position changes with respect to the starting grid at certain points during a race, using as inputs a number of topological parameters, and with a maximum error that resides around the 4% threshold.

During the work of this thesis, we had the opportunity to examine in depth the design process of racetracks and to try making it more efficient through the use of simulations and Machine Learning techniques, an approach that can also be extended outside the videogame industry. Analyzing the driving dynamics showing up during actual races is, in fact, at the basis of the creation procedure of every type of circuit. A track must first of all be fun and, as to ensure this condition, the designer must pay great attention in the definition of its layout, whether it belongs to the virtual world or to the real one. A similar approach might also be followed to facilitate the creation of levels in other types of racing games. As an example, in *kart racing games* (videogames with simplified driving mechanics, obstacles, and vehicular combat) an alike tool could be developed implementing metrics describing dynamics concerning, for example, fighting between players and power-ups usage.

To make the data extracted from simulations as reliable as possible, we would need to use more sophisticated *bots* as drivers, since those that come with Speed Dreams tend to not behave in a sufficiently realistic way when managing group racing dynamics (overtakes and collision avoidance especially). The need of an artificial intelligence that imitates human behaviour in a sufficiently precise way is certainly the biggest limitation of the approach followed in this thesis.

For what concerns possible future works, this thesis can be used as a starting point by racing game developers and designers to improve the workflow related to the creation of tracks for their games. As we have seen, the tool is completely detached from the simulator, therefore it is also possible to adopt a more realistic game, or even a completely different type.

If we would like to keep using Speed Dreams (SD), it would be useful to add races taking advantage of the weather engine built into the game. This would add variety in the simulations, and a whole new set of analyses could be performed concerning cars behaviour under adverse conditions (low visibility, low grip etc.).

Another possible addition would be to include a procedural content generation system that creates automatically some sections of the tracks, for example one that closes circuits with more than one segment, maybe following some kind of heuristic in the process.

As discussed in the thesis, the current classification that the tool performs on tracks is limited to determining whether it has a low or high probability of overtakes, and whether it has high or low average time gaps between players. The process of binding the circuit to one or the other can be made more precise, using a larger dataset on which to run the K-Means algorithm, or even trying new classification models. Additional analyses could be performed looking more at the group racing dynamics between *bots*, like those regarding the damage taken by their car, or estimating the probability with which each class of collision could happen.

We can also extend the work done with the prediction of dynamic metrics using Machine Learning techniques to other characteristics. One could also integrate such estimations directly into the editor tool, and try to fit new regression models that were not taken into consideration in this thesis.

Bibliography

[1] Speed Dreams — The greatest open source racing sim on Earth: my part in its success. https://commut3r.wordpress.com/, 2009.

[2] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 395–402, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Lance Carter. Lance carter's history of racing games - installment twelve. https://historyofracinggames.wordpress.com/12-2/.

[4] Riccardo Galdieri, Cristian Camardella, and Marcello Antonio Carrozzino. What makes a circuit likeable and how different input devices can influence the perception of tracks in racing games. *Computers in Human Behavior Reports*, 3:100072, 2021.

[5] Jeff Gerstmann. Classic nes series: Excitebike - review. https://web.archive.org/web/20040810000317/http://www.gamespot.com/gba/driving/famicominiexcitebike/review.html, 2004.

[6] Kris Hauser. Algorithms for Optimization and Learning CS B553 Multivariate Newtons Method and Quasi-Newton methods. Lecture. http://people.duke.edu/~kh269/teaching/b553/newtons_method.pdf, January 2012.

[7] Sean Hollister. Unity's IPO filing shows how big a threat it poses to Epic and the Unreal Engine. *The Verge*, 08 2020.

[8] Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679–688, 2006.

[9] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press, May 2014.

[10] Muhammet Köle, A. Etaner-Uyar, Berna Kiraz, and Ender Özcan. Heuristics for car setup optimisation in torcs. In *2012 12th UK Workshop on Computational Intelligence, UKCI 2012*, 09 2012.

[11] Muhammet Köle, A. Etaner-Uyar, Berna Kiraz, and Ender Özcan. Heuristics for car setup optimisation in torcs. *2012 12th UK Workshop on Computational Intelligence, UKCI 2012*, 09 2012.

[12] Raph Koster. The cost of games. https://venturebeat.com/2018/01/23/the-cost-of-games/, January 2018.

[13] Daniele Loiacono. Learning, evolution and adaptation in racing games. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, page 277–284, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Automatic track generation for high-end racing games using evolutionary computation. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3:245 – 259, 10 2011.

[15] Boris Polyak. *Introduction to Optimization*, chapter The gradient method: Heuristic Considerations, pages 20–21. Optimization Software, Inc., Publications Division, 2020.

[16] Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific computing with MATLAB and Octave*, chapter 4, pages 109–111. Springer-Verlag Berlin Heidelberg, 2010.

[17] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014.

[18] Jacopo Sirianni. Supporto alla progettazione e analisi di tracciati per un simulatore di guida. Master's thesis, Politecnico Di Milano, 2016.

[19] Joe Thompson, Simon Wood, and Xavier Bertaux. Speed Dreams - A free Open Motorsport Sim and Open Source Racing Game. http://www.speed-dreams.org/, 2016.

[20] Julian Togelius, Renzo De Nardi, and Simon Lucas. Towards automatic personalised content creation for racing games. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 252 – 259, 05 2007.

[21] Jeff Ward. What is a Game Engine? https://www.gamecareerguide.com/features/529/what_is_a_game_.php, 2008.

[22] Bernhard Wymann. TORCS Robot Tutorial. http://www.berniw.org/tutorials/robot/, 2013.

[23] Bernhard Wymann, Espié Eric, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.