School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

# Racing Game with an Artificial Intelligence Opponent

**Ben Hulse**

**Submitted in accordance with the requirements for the degree of Computer Science with High-Performance Graphics and Games Engineering**

2019/2020

**40 credits**

The candidate confirms that the following have been submitted.

| Items | Format | Recipient(s) and Date |
|---|---|---|
| Final Report | PDF File | Minerva (05/05/2020) |
| Source Code | Git Repository Access and URL via E-Mail | Supervisor, Assessor (05/05/2020) |
| Final Game Build | Windows 64 Bit Executable file as part of the GIT Repository | Supervisor, Assessor (05/05/2020) |
| Demonstration Video | Youtube video link vie E-Mail | Supervisor, Assessor (05/05/2020) |

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

[illegible]

(Signature of Student)

## Summary

The goal of this project is to create a prototype racing game in which the player can race against the clock, or against one or more Artificial Intelligence (AI) controlled opponents. The game will be built using the Unreal 4 Engine and its accompanying Blueprints visual scripting language. Originally the game was to be played in Virtual Reality (VR) using the HTC Vive, however due to the COVID-19 outbreak, access to the VR labs was closed off and the project had to be adjusted. One of the primary focuses in the project will be the implementation of the AI that the player competes against.

## Acknowledgements

I want to thank my supervisor, Dr. He Wang, who supplied great advice and guidance throughout the course of the project and helped me to achieve an end goal that I am extremely happy with.
I would also like to thank my parents for their constant love and support.

# Contents

# Chapter 1

# Project Description

## 1.1 General Overview

Video games are a great way to pass time, and can be enjoyed with friends or alone. In the case of racing games the most common way to enjoy the game alone is to race against one or more AI opponents. These opponents can range in difficulty and so long as they behave in a human way, and remain competitive with the player, they can provide hours of fun as adversaries.

Because of how long racing games have existed there is a huge number of methods that can be used to approach the designing of a racing AI. One of the earliest is to have the AI's vehicles simply follow a set path, unable to waver from that path or react to stimulus. A more modern approach is to use machine learning; the AI is comprised of a neural network with the appropriate input and outputs, then is "trained" before the game is released. I will talk more about existing technologies and how these affected my design decisions in chapter 3.

One problem in Computer Science that is very similar to that of controlling a racing AI, is the problem of path planning in robotics. Much like the AI problem, there are many different methods for path planning, one of which I have taken inspiration from in my AI design for this project: Potential Fields. Potential Fields is a method of path planning where the environment and its contents exert forces on the agent, in this case the robot. Where obstacles exert repulsive forces, and goals exert attractive. The agent then uses the resultant of these forces to dictate in which direction it should move.
The original idea for this project was to create a game playable in VR, however due to the CoV-19 outbreak restricting access to the VR labs I am unable to do formal testing and documentation of the VR capabilities and so it is no longer a focus of the project.

## 1.2 Aims and Objectives

The aim of this project is to create a piece of exploratory software that implements a basic racing game using a pre-built game engine. Being exploratory software, the aim is not to produce a finished project that could be shipped, but to create a prototype that demonstrates the primary features of the project.
The project objectives are split into four parts: The racing game, the race track, the AI implementation, and performance.

### 1.2.1 The Racing Game

The primary objectives of the racing game will be:

- The game should be created using a pre-built engine such as Unreal or Unity
- The game should be playable using a keyboard/mouse or joypad
- The game should feature driving mechanics that are realistic but also fun to drive
- The game should have two game modes: Time Trail and Head-to-Head
- The game should allow the player to compete against one or more AI in Head-to-Head
- The game should allow the player to spectate AI racers competing in both Time Trails and Head-to-Head races
- The game should keep track of the times of racers, displaying them when the race is finished

### 1.2.2 The Race Track

Defining objectives for a racing track is harder because a lot of it will be subjective.
Race track objectives:

- The race track should be built using the level editor of the engine chosen for the project
- The race track should be interesting enough that it is fun to drive on
- The race track should not be too complex that it is too difficult to drive
- The race track should be wide enough that multiple cars can be side by side on it
- The race track should be designed in a way that it is clear to the player where they should be going

### 1.2.3 The AI Implementation

My goal is to create an AI that races in a realistic looking way, can react to other racers on the track, and can perform at a range of difficulties.
Objectives for the AI racer:

- The AI should be able to control its vehicle
- The AI should be able to navigate the track
- The AI should be able to avoid collisions with other vehicles
- The AI should drive in a realistic looking way
- The AI should be able to operate at a range of skills

### 1.2.4 The Performance

With the original project being intended for use in VR there was a performance constraint of 90 frames per second (FPS). With the project's focus moving away from VR, this constraint is no longer necessary. However I feel that it is still a good target for the project to aim for as it would mean that if the project were to be used with VR in the future it is still capable. Therefore the objective for performance is:

- The game should run at a framerate that is consistently higher than 90fps at all times

### 1.2.5 Expandability

For smaller indie games, a great way to increase the playability and lifespan of the game is to make the game easy for a third party member to expand on. Whether this be in mod support or an open source model, having the game be expandable by the players will mean there is more content in the game available, without extra work for the developers. Hence I want to design my game in a way that will make it easier for other people to expand on the game and add new features.
Objectives for such a feature set:

- The level architecture needs to be simple enough that custom levels can be designed with relative ease
- The information given to the default AI should be a part of the level so that custom made levels can be designed to allow the AI to race on them
- The game should have the ability for a custom AI system to be usable within the game, that can still interface with the vehicles and compete in the races as intended

## 1.3 Deliverables

Because this project is exploratory software, the main deliverable will be the proof of concept game. This game will be in the form of a runnable exe for windows and will include the features outline in the previous section. However because it is only exploratory, there may be non-production quality assets used, and the game will likely not be polished to the standard expected from a final release of a game. The second deliverable will be the source code for the project, in the form of Unreal Project file, blueprint files, and the accompanying folder structure. This will be handed in as a link to the git repository.

# Chapter 2

# Planning and Methodology

## 2.1 Project Management

In order to plan this project out properly I had to chose a development approach that would be appropriate for the time frame and complexity of the project. Because this is an individual project I decided there was no need for more complex approaches such as Agile. Instead I thought it appropriate to go with a simpler iterative Waterfall approach; where the basic steps of waterfall are followed, but iteratively for different stages of development.

## 2.2 Initial Plan

Because of my choice of project management method, one of the first things to do when planning my project was to outline the different 'stages' of the project. The stages should only contain one large part of the project, but should be complicated enough that the project will not need to be comprised of too many iterations.
The 'stages' of the project I chose were as follows:

- Implement the track
- Implement the driving mechanism
- Implement the single player time trail
- Implement the single AI time trail
- Implement the player and AI head to head

By following these five stages I was able to conduct the appropriate research and design for each stage as I needed it, meaning there was no need to find all of the information that I would need for the project at the start.

## 2.3 Version Control

For the project version control I decided to use the Unreal Editor's built in version control that connects to a git repository that I had created on my Gitlab account. This allowed me to push changes and revert where needed.
As well as this I made zip archive packages at crucial points in the development as a second backups and so that different versions of the project could be compared side by side.

# Chapter 3

# Background Research

## 3.1 Engine Choice

The first thing to research when starting the project was the engine that would be used. Because of the time frame of the project, creating an engine from scratch to work with VR was completely infeasible, so from the start the decision was made to use a high level, pre-established 3D engine. Next was the decision for which engine to use; going into the project I already knew of two engines that would be perfect: Unreal Engine 4 (UE4) and Unity. So all I had to do is research the pros and cons of each and make a decision based on that. In the decision making process for these engines I refer frequently to the VR aspect of the game. Although this feature was abandoned, as mentioned, I have elected not to remove the references to it in the research summary as it contributed significantly to the decisions made, and is still relevant should the project be extended to include full VR support.

### 3.1.1 Unity

A blog post by Ilya Dudkin on skywell software[5], nicely outlines the features and summarises both engines for use in VR development. Unity seems to offer a set of tools that are more simplistic, making them ideal for a beginner to game development to learn. The unity engine also allows development in the C# language, which is in lots of ways easier to program in, however this comes at a performance cost. A lot of the advantages to using unity come when developing 2D or mobile games.

### 3.1.2 Unreal Engine4

The Unreal Engine 4 (UE4) seems to be the more advanced of the two, offering a more full suite of tools, but at the cost of being less beginner friendly and being harder to learn. The languages of choice for UE4 are: the built in blueprint scripting system, and C++. The latter of which is a less programmer friendly language than C#, but offers greater performance, which is important for a VR project as mentioned in the introduction.
Because of the greater performance and tool-set for developing 3D games I have chosen to go with the Unreal Engine 4 for my project. This will come at the cost of needing more time to learn the software, but I feel the time commitment will result in a better end product. In order to learn how to use the UE4 Editor I found a number of sources, two of which I decided to use: raywenderlich.com's 'Unreal Engine 4 Tutorial for Beginners'[9], and Game Dev Academy's youtube series by the same name[10]. The latter of the two was my primary source as the first dozen videos go in depth; describing the tools of the engine and level editor, which exactly what I needed to create the first part of my project. However I only used these to learn the basics of UE4, though the later parts of the tutorials do cover actual game development with the engine, I decided to use another, more racing game oriented, source to learn how to develop the game: Unreal Engine's 'Blueprint Time Attack Racer | v4.8 | Unreal Engine' series on

youtube [4]. This tutorial was particularly helpful as it not only showed all the tools I would need to implement my game, but it demonstrated the implementation of some basic methods of structuring a racing game; so it acted, not only as research into the engine, but research into developing a racing game as well.

## 3.2 The Race Track

After gaining the knowledge I would need to build the game, I moved on to researching the specifics of implementing a racing game and AI. To this end I found a section from the book 'Game AI Pro' by Steve Rabin [8]. This book contains a chapter: 'Representing and Driving a Race Track for AI Controlled Vehicles', which outlines a basic node oriented storage system for a race track. The chapter also outlines some of the maths for creating and using this storage system, as-well as other aspects of the system use by an AI.
The system outlined in this chapter is very simplistic, and contains only the bare minimum information needed for a very basic AI, which for the scope of this project is ideal. Therefore I will be basing my model on this one, with additions and changes where needed to implement the potential field like aspect of the AI.

## 3.3 Racing AI

### 3.3.1 Controlling the Car

One of the first things I will need to research for the AI is how it will interface with the vehicle it is controlling. Because I have decided to use UE4 to build this project, I am able to use a number of pre made assets from the engine that will help speed up development. One such asset is the 'Advanced Vehicle Blueprint' (AVB) which give access to a player controllable car. From the tutorials I mentioned earlier, I have found that such a blueprint will have an attached 'controller' that acts as a class through which either the player, or an AI, can control the car. I also inspected the 'Advanced Vehicle Blueprint' class to find that the user inputs affect the car through three functions: 'Set Throttle Input', 'Set Steering Input', and 'Set Handbrake Input'. When the AVB is not being controlled by a player controller, inputs are disabled, therefore I can simply have the AI controller control the vehicle by calling these three functions when needed.

### 3.3.2 Implementation

One of the earliest examples of Racing Game AI is in the 1982 game Pole Position, in which the racing AI was very simple and would follow a pre-set racing line. This method was a product of the time, as computation power was very low meaning the AI had to be as simple as possible. On the other end of the spectrum is the most modern approach to racing AI: neural networks (NN). These are a way for the AI to find the best path around the track themselves, without much intervention from the developer needed. Another advantage of neural networks is that they offer a much easier approach to variable difficulties. To create a NN driven AI that is a lower difficulty, it just need to be trained less. The more a neural network is trained at a task the more effective it will become, therefore training less will mean the AI will perform at a

lower skill level without the need to be handicapped. Youtube user Samuel Arzt made a video demonstrating a neural network learning to navigate a race track in a very simple 2D game [3]. If this theory is applied to the more complex 3D environment of a full racing game, it could be used to control an AI vehicle.
However in order to stay within the scope of this project I decided to take a simpler approach that still allows me to create an AI that is realistic looking. The method I have decided to take is very similar to potential field path planning in Robotics. Michael A. Goodrich has written a small tutorial on potential fields[6], this pdf contains the basic theory, and some maths behind the generation of potential fields. A lot of the information given will be important to the project, as I will be exerting forces on the AI car to lead it towards its goal. However instead of using a pre-calculated potential field I have chosen to calculate the resultant force on the fly, as some of the objects, such as other vehicles, that will be exerting forces on the AI will be moving. There are some problems with potential fields that are raised in this documents: one of which is the possibility of an agent being trapped in a C-shaped object, however because of the nature of a race track this should not be an issue.

### 3.3.3 Difficulty

A large consideration in designing any AI that is made to compete against a human player is that of difficulty. If an AI is too good, or not good enough, it is no fun to compete against, because of this in most cases a variable difficulty system that can adjust to the player's ability is needed.

#### Rubber Banding

One standard way to balance such an opponent is to use 'rubber banding'[2] where when the AI is behind the player by some distance it is given some advantage, usually in the form of improved speed and/or handling, so that it can catch up. This can also be done in reverse, where if the AI is too far ahead of the player it is disadvantaged in a similar manner. This is all done in an attempt to make the game more interesting, and to not punish the player too much for mistakes. The degree of rubber banding can be adjusted to change the difficulty of the AI. However this method has many issues, the largest of which include: it is unfair on the player, and it makes a majority of the race pointless as only the final 25% or so of the race determine the winner.

#### A Better Approach

'The Pure Advantage: Advanced Racing Game AI'[7] is an article by Eduardo Jimenez that explores this floor in the "rubber band" design, and offers an alternative. The article outlines a method where each AI has a set of skills, that are not physical factors, but more performances in certain aspects of the game, which can be changed over time to change the difficulty of the AI. The article also talks about other additions to this that make the AI more fun to play against such has race scripts and the use of grouping the AIs. However I want to design my AI so that it does not require other AIs to be present to work. Because of this I will only be using the 'skills' portion to adjust the AI. Important considerations are the frequency and how heavily the skills are adjusted, what aspects of the racing the skills effect, and ensuring the skill values have enough discrete steps that they can represent a large range of abilities in the AI.

# Chapter 4

# The Race Track and Driving Mechanics

## 4.1 Designing and Creating the race track

Because I had no real knowledge of the UE4 Editor and how to use it properly going into the project, the first step I took into creating the track was to use tutorials to learn how to use the editor, in particular the level editor. As I mentioned in 3.1.2, there were two main sources that I used to learn the engine. After watching the appropriate parts of these tutorials I felt confident enough to be able to use the level editor tools for designing and building the track.
As mentioned in 1.2.2, the design of the track needed to be interesting to drive on, but simple enough that it is fun. As well as this there was a minimum width requirement of more than two cars, and the track needed to be visually clear to the user.
To satisfy the visual clarity objective I decided to use the assets supplied by UE4 in the 'Advanced Vehicle' example. These assets are a grey and black road, with a white and red checked border, which is very clear to the user what is the track and what is not. To contrast this I decided the rest of the world should be covered in a grass texture (also supplied by UE4, under 'Starter Content'). In order to make the track components large enough I had to scale them in the x and y directions by a factor of 2. I also scaled them by a factor of 0.25 in the z direction as the larger edges were causing unrealistic physics interactions when the drivers went over the edge of the track.
The objectives relating to the complexity of the track depended on its layout. I first designed the track on paper so that I could get a general shape in mind. While designing this track I went through a few iterations making changes to the amount and severity of turns. The main goal was to find a good balance between how long the drivers would be driving straight compared to how long they would be in turns. I tried to avoid chaining too many corners directly after one another as it could be disorienting and hard to follow, and would require the racers to navigate them slowly, which would make the race less fun. I also tried to add a few more unique turns, such as larger slow turns, and sharper turns that required braking long before the turn itself. These deign drawing can be seen in the Appendix, figures: A.1, A.2, and A.3.
Building the track itself was just a matter of placing down the different track components in a way that resembled my design as close as possible. The assets that I used from UE4 contained 7 pieces of track of different shapes: Straight Full, Straight Half, Straight Quarter, Turn 22 and Half, Turn 45, Turn 90, Turn 90 Large. I also created a duplicate asset of the Straight Quarter track piece, and changed the texture so that it was a black and white checked patter; This was then used as the start line. These pieces can be seen in appendix figure B.1.
In addition to the track, I used the 'Wall 400x200' asset that is also party of the UE4 'Starter Content' pack, to create a barrier around the track, this way if one of the drivers goes off the track they can't get too far before crashing into the walls. This also has the advantage that if the checkpoints cover the entire space between the barriers on either side of the track, a racer

won't be able to skip a checkpoint. The Final implementation of the track can be seen in appendix figure B.2.

### 4.1.1 The Driving

Going into the project I knew that if I were to create the vehicle and driving mechanics from scratch, in the time frame of this project, I wouldn't be left with a very impressive result. Instead it was clear that by using the pre-made vehicle asset and driving mechanics that the UE4 Engine supplies in its "Advanced Vehicle Blueprint" I would be able to instead focus my time on other aspects of the project which would give me a much more enjoyable end product. The AVB comes with fully implemented driving mechanics, which simulates a car with torque, proper gears and breaking mechanics. The model asset used by the blueprint is very basic, being only a chassis with wheels, but for the purposes of an Exploratory Software project such as this one, it is fine. The blueprint allows interaction through a set of functions in which the controller (be it AI or player) can change the throttle, set the steering wheel position, or set the hand-brake. There is also the ability to make the gear shifting manual, but I thought this kind of low level control into the AI would be out of the scope of this project.
However some aspects of the default AVB that are not perfect and need changing. In section 1.2.1 I mentioned that the driving should be realistic but fun. One thing I noticed about the UE4 implementation is that the driving was very "floaty", so to fix this I edited the AVB to increase the mass from 1500 units to 2000.
Another change I made to the AVB is to add the ability to reset the car if it gets flipped. To do this I added a variable to the AVB that stores a 'Reset Location', which represents the location that the vehicle is reset to at any given time. Then add a function that can be called by the AI or Player controller that moves the vehicle back to the reset location. The value stored as the reset location is updated when the vehicle hits a checkpoint which is explained in section 5.1.
I also added the ability to look behind the player's car when driving. To do this I added a key bind to the project called 'LookBehind'. Then had the vehicle listen out for this key-press, and rotate the arm attaching the camera to the car by 180 when the key is pressed, and rotate it back when it is released. This works because in UE4, if a blueprint is a player controllable pawn, it must have a camera for the player to see through as a component. In the AVB the camera is attached to the vehicle by a 'Spring Arm' to prevent jarring motion in the camera if the car goes over bumps. This worked to my advantage as it meant that I only needed to rotate the spring arm and in turn that would move the cameras position. The figure in appendix C: C.2, shows the rear camera (which is used in the third person cam) and the Spring Arm (the red line) connecting it to the body of the car.
Additionally I added a variable to the blueprint that stores an ID for the vehicle. This is initialized at runtime by the Tracker (Section 5.2) and acts as a unique ID for each instance of the blueprint, which in this case is each vehicle on the track.
Another function was added called 'Finished Race', the implementation of which is covered in chapter 7.

# Chapter 5

# The Single Player Time Trail

In section 3.1.2 I mentioned that for learning how to use the UE4 engine to develop a game I would be using a racing game focused tutorial on Youtube. This served two purposes, first it would allow me to learn the tools and skills I would need both for general use, and for the development specifically of a racing game. Secondly it meant that I could follow along and take the features I needed in order to build a very basic "first draft" version of the project. Although this version of the project would likely be completely different and need rebuilding for the final version, it would be valuable learning experience and therefore was worth the time commitment.
After building the "Draft" version of the project I found some fundamental problems with how the computation was divided between different blueprints, this meant that implementing a proper head-to-head system later would be very difficult to do and would likely result in a program flow that is very messy, and difficult to follow. So during the implementation of the head-to-head functionality I completely rebuilt the system, mostly from scratch to follow a more strict and coherent distribution of labour and to be much clearer, and easier to expand. This section will focus on the refactored version of the time-trail game mode and how it works.

## 5.1 Checkpoints

The most common approach to implementing a racing game is to use a checkpoint system, this is where there are a number of checkpoints spread across the track that must be hit by the racer in the correct order for them to complete a lap of the track. By having multiple checkpoints instead of just one at the start/finish line, it ensures that the racer traverses the whole lap and doesn't cut corners or large sections of the track all together. For this system there also needs to be some mechanism to keep track of all of the checkpoints and the racers, so that the game knows the progress each racer has made around the track, this is the job of the Tracker (See section 5.2).
For these checkpoints I created a new blueprint, which inherits the 'actor' type. In UE4 an 'actor' is any object that can have a physical location in the world. Each checkpoint contains an integer variable that stored the ID of that checkpoint, where the IDs correspond to the number of the checkpoint sequentially along the track. The blueprint contains a 'Box Trigger' that is the width of a track, this way it can listen for the 'On Component Begin Overlap' event of the trigger, which is triggered when another actor enter the box trigger area, in order to know when a vehicle reaches the checkpoint. The event graph, the portion of the blueprint code that listens for and reacts to events, only contains the one event listener. The blueprint also contains an event dispatcher, which in UE4 is used to send a global event signal that any other blueprint can listen for. When the 'OnComponentBeginOverlap' event is triggered, the Checkpoint will check that the overlapping actor is a vehicle, then dispatch the 'CheckpointActivated' event with the Checkpoint ID and the vehicle ID as parameters.

## 5.2 Tracker

The purpose of a tracker in my project is to store all of the information about the current race state and to do the main computation for running the race. When designing the tracker class I knew that it would be very complex and I had a lot of considerations to make in order to avoid the issues I had with the "draft" version of the project.
In the "draft" version, the tracker is responsible for storing all of the checkpoints and handling their dispatched events. The logic for the race is spread over the Tracker and the MyPlayerController class. However I felt that having all of the race logic be in the same class (The Tracker) would make the code easier to follow and much easier to expand upon.

### 5.2.1 Tracker Design and Variables

Firstly I created a class for the Tracker, again it inherits the 'actor' class, however this time it is done so that it can be instantised as a part of the level instead of at run time, as I want to ensure that there is only one instance of the class per level. I also gave the instance of the class the tag "Master", this way any object in the level that needs a reference to the tracker can simply use the 'Get All Actors Of Class With Tag' function with the tag parameter set to "Master".
There were some parts of the 'draft' tracker that I could simply copy over to the new version. The first of which was that the tracker needs to store an array of references, pointing to each of the checkpoints. This array can be filled manually by the level designer inside of the level editor, this way we can ensure the array references each of the checkpoints in the correct order. There must also be an integer variable in the class to store the amount of checkpoints.
The 'draft' version of the project was implemented as purely a time trail tracker, meaning changing aspects of it to allow for head to head play was difficult. Because of this I knew I must design the new tracker in a way that allows me to expand it to head-to-head mode easily. For this purpose I had it store, as arrays, references to: all of the vehicles in the race and a 'RacerData' structure instance (See section 5.6.1) for each of the vehicles. The contents of the vehicle array is set by the developer in the level editor, and a reference to every vehicle on the track must be added to this array for the game to work properly. The tracker also stores a 'Displayed Racer ID' which is configurable in the level editor and tells the tracker which Racer's stats (such as time and lap count) should be displayed on the heads up display (HUD). Additionally a reference to the HUD needs to be stored so that the tracker can initialize the HUD at runtime. The Tracker does this as there will only be one Tracker present, so there will only be one HUD present.
The Tracker needs to contain the configuration and state for the race itself which it does with four variables: CurrentRaceTime, MaxLaps, RaceState, TrackerMode. The first two are just float and integer respectively and are self explanatory. RaceState is an instance of a Race State enum (See 5.6.2). TrackerMode is an instance of the RaceMode enum (See 5.6.3). MaxLaps and TrackerMode are again configurable in the level editor. The ability to set the tracker to head-to-head mode is only implemented here to make it easier to add the head-to-head mode later, as whenever I add functionality that differs between modes I can add a simple branch statement (UE4 Equivalent of if statement) that checks the race mode. As well as this the class

contains a Timeline instance called RaceTimeline. In UE4, timelines can be used to keep track of time; In this case the timeline is used to count how long the race has been going for.
The Tracker stores four 'Display Text' variables which simply store the text that can change and needs to be written to the HUD during runtime. These are: CurrentRaceTimeText, DisplayedRacerLapText, CountdownText, and DisplayedFinalRaceTime. These variables are updated at the appropriate times by the tracker to display the relevant information to the player. The information that is displayed will be relevant to the racer with ID: 'DisplayedRacerID' which will always be the only car on track for a time trail. When the project is extended to Head-to-Head, the variable can be adjusted in the editor to follow a specific car on the track.

### 5.2.2 Tracker Functions and Events

There are 4 Functions defined for the Tracker Class: ActivateCheck, CheckRacerFinished, CheckRaceFinished, InitText.

#### Checkpoint activation logic

ActivateCheck - This function is responsible for checking that, when a vehicle activates a checkpoint, it is doing so in sequential order and advancing that racer's progress in the race. The function is called in response to a Checkpoint:CheckpointActivated event and takes the ID of the checkpoint and the racer as arguments. The function checks first that the checkpoint that has been activated is the correct checkpoint, then increments the 'Next Checkpoint' value of the racer. Next the function checks if the racer has finished a lap and increments the racer's 'CurrentLap' variable if so. Next the function updates the ResetLoc variable of the racer's vehicle. And finally updates the HUD text if the racer in question is the racer being displayed on the HUD, and returns True.
CheckRacerFinished - This function is called if ActivateCheck returns True, and takes the ID of the racer that was passed to the ActivateCheck function. The function checks if the racer in question has finished the race (If their current lap is greater than the max laps), returning false if not. If the racer is finished however it will set TotalTime in the racer's RacerData instance to the value of CurrentRaceTime, and their Finished value to true, then return True. If CheckRacerFinished returns True, the Tracker will call the 'Finished Race' function of the vehicle in question.
CheckRaceFinished - Called if CheckRacerFinished returns True, this function just checks if all racers in the race have Finished values of True; returning the result.

#### Other functions and the Event Graph

InitText - This function is called when the program starts, and simply sets all of the 'Display Text' variables default values.
The final part of the Tracker, and the most complex, is the Event Graph. The event graph contains 6 event handlers: BeginPlay, CountdownStart, RaceFinished, StartRaceTime, StopRaceTime, InputAction Accept.

BeginPlay - This event is triggered when the actor first loads, so in the case of the Tracker, when the level loads. The first thing the event handler does is initialise some variables. It fills the 'Racers' array with an instance of the RacerData struct for each racer in the level, with all values being default except for the 'Racer ID'. It then calls InitText and initialises the HUD reference by instantising the custom HUD class (See 5.4) and storing a reference to it. Finally it gets a reference to the player controller and sets the Tracker to being able to accept input from the player. This is necessary so it can handle the event for when the player presses the 'Accept' key to start the race. The handler also gives each checkpoint it's ID and binds the dispatched event (CheckpointActivated) to checkpoint activation logic functions mentioned above. Finally the handler initialises CheckCount and then displays the Start Screen to the user.
CountdownStart - This event is called when the race is ready to start and is responsible for the countdown at the beginning of the race, after which it sets the RaceState to 'Ongoing' and sets the HUD to 'Ingame' mode.
RaceFinished - This Event is called if the CheckRaceFinished function returns True and is responsible for post-race clean-up. It sets the RaceState to 'Post-Race', stops the timeline, and then updates the HUD appropriately.
StartRaceTime/StopRaceTime - These event handlers respectively start and stop the timeline from updating the CurrentRaceTime and CurrentRaceTimeText variables.
InputAction Accept - Is the handler for the user input that starts the race if the race is in the 'Pre-Race' state and resets the race if in the 'Post-Race' state. It starts the race by invoking the CountdownStart event, and it resets the race by reloading the level. After the MainMenu was added (See section 9.1.2) to restart the game, the Tracker instead loads the MainMenu level.
Finally the Tracker has two Event dispatchers: RaceStart and RaceEnd, which as the names suggest, are invoked at the start and end of the race. These are used by the PlayerController (See 5.3) and later the AIController (See 6.4).

## 5.3 The Player Controller

The player controller is the class that acts as an interface between the player and a pawn that they can control. In the case of the time trail mode there is a single AVB pawn that has a single player controller associated with it, the type of which is my custom 'RacingPlayerController'. The functionality of the 'RacingPlayerController' is very minimal as the accepting of inputs from the user is done in the AVB so the main function of the custom player controller is implemented in it's parent class: 'Player Controller'. This functionality is to allow the controlled pawn to accept player input events.
The 'RacingPlayerController' blueprint has a simple event graph; only handling the BeginPlay event, after which it: gets and stores a reference to it's controlled vehicle, gets and stores a reference to the level's Tracker, disables the inputs for the controlled vehicle (to stop the player from moving before the race starts), and binds two call-backs events to the Tracker's RaceStart and RaceEnd dispatch events (See 5.2). The call-back to the RaceStart event enables the controlled vehicle to accept player inputs. The RaceEnd call-back disables the inputs once again and sets the controlled vehicles throttle to 0 so that it does not keep accelerating after the race is over.

The final part of the player controller is a function called 'FinishedRace' that is called on the controller by the tracker when that individual controller's vehicle has finished the race, this function again disables inputs, and sets the throttle to 0.

### 5.3.1 Additional Changes to the AVB

For the implementation of the Time Trail game mode, another function needed to be added to the Vehicle Blueprint: Finished Race.
This function is called by the tracker when that vehicle finishes all of the laps of the race. It is responsible for telling the Controller of the vehicle that it has finished, and moving the vehicle off the track using a set of teleport functions.

## 5.4 HUD

An essential part of any racing game is the heads up display (HUD) that is used to display any information to the player. In the case of the Time Trail mode the information to be displayed is: the start screen, the countdown to race start, the race time, the current lap, and the end screen.
Because the HUD will need multiple screens, I added each required screen as separate 'Border' components to the HUD blueprint so that they can be hidden and shown when appropriate. 'Border' components are an element in a HUD that can contain other HUD elements, and can be shown/hidden during run time programmatically.
For the time trail implementation, the borders contained in the HUD are:

- Time - Displays the current race time
- Lap - Displays the current lap of the racer with ID = 'DisplayedRacerID'
- Countdown - Displays the countdown at the start of the race
- StartScreen - A simple splash screen welcoming the player and telling them to press Enter or Start to begin the race.
- TTEndScreen - A simple end screen displaying the racers time to complete the race

The HUD class also contains a set of ESlateVisibility variables, each variable is bound to one of the borders. This way the value of the variable will dictate whether or not the border is visible to the player.
Each element of the HUD that displays information is bound to the appropriate text variable in the Tracker blueprint (See 5.2). This way it will constantly display the information in that variable. With this approach, the contents of the text variable just need to be updated within the Tracker and the information on screen will be updated accordingly without the need for a call to the HUD blueprint.
As well as this the HUD blueprint contains 4 functions that display the appropriate borders for each screen of the HUD: ShowCountdown, ShowInGame, ShowStartScreen, and ShowTTEndScreen. This way, the tracker can use it's reference to the HUD to call any of these functions and display the appropriate screen.

## 5.5 Macros

In UE4, a macro is a simplified function, that can only make data calculations from a set of inputs and return them as outputs. For my project I needed two macros: TimeToText and NegMod. UE4 also contains MacroLibrary classes which is where Macros are defined, so for my project I created a macro library called 'MiscMacros'
TimeToText - Is a macro that takes a float variable that represents a time in milliseconds, and returns it as a readable text variable in the format MM:SS:mmm. This is used wherever time needs to be displayed on the HUD to make it more human readable.
NegMod - Is a simple macro that takes two integer arguments: Dividend and Divisor, and returns the mod of the two, where a negative value is wrapped around to be positive. This was needed as the UE4 implementation of modulo will return negative values if the dividend is negative. As an example in UE4 -16 % 10 will return -6 where NegMod will return 4.

## 5.6 Structs and Enums

### 5.6.1 RacerData Struct

A simple structure that is used to store the data about a single racer in a race. The fields of this struct, their data types and their default values are:

- RacerID : Integer = 0

  - A unique ID number for the vehicle

- NextCheckpoint : Integer = 0

  - The ID of the checkpoint that the racer needs to hit next in order to do them in sequence

- CurrentLap : Integer = 1

  - The Lap number that the racer is currently on

- Finished : Boolean = False

  - Has the Racer finished the Race

- TotalTime : Float = 0.0

  - Set post race, how long the racer took to finish the race

- Position : Integer = 0

  - The position that the racer is in at the current time

- Distance : Float = 0.0

  - How far around the track the racer is at current time (See 7.7.1)

### 5.6.2 RaceState Enum

A Simple enumeration of the possible states that a race can be in:

- Pre-Race
- Countdown
- Ongoing
- Post-Race

### 5.6.3 RaceMode Enum

A Simple enumeration of the possible modes that the race can be:

- Time Trail
- Head-to-Head

# Chapter 6

# The AI Time Trail

## 6.1 The AI Theory

As mentioned the AI for this project will use a force oriented method of traversing the track, by having objectives exert either an attractive or repulsive force on the AI that it then uses to know which direction and at what speed the AI should be travelling.

## 6.2 AI Target Nodes

Because the system needs a set of objects that exert attractive forces on the AI, I needed a set of objects that will always be in front of the AI on the track. For this purpose I created the 'AITargetNode' class. This class inherits 'actor' but is invisible and has no collisions in the world during play time. The purpose of the class is to simply represent a point on the track that the AI should aim for, the nodes also store a 'Target Speed' variable that is set in the Level Editor and represents the speed that the AI should aim to be when it is at that node's position. This value is different for each instance of the node. NOTE: In the development build that is available on the git, the AITargetNodes are visible to show11 how they are distributed. I have laid these nodes at the centre of the track, all along the track separated by roughly 1200 units, this way the AI can look ahead to the next node(s) and have them exert forces on the AI until the AI gets close enough that the node is no longer needed. The amount of nodes that affect the AI at once is dependent on difficulty (See 7.3.2). However before the difficulty was implemented the value was set to 4 nodes.

## 6.3 AI Target Tracker

The 'AITargetTracker' is a simple class that just keeps an array of references to each of the AITargetNodes in order. Like the Tracker, it is an actor and will have a single instance in the world, and the array of nodes will be filled manually in the level editor. However unlike the tracker it does no actually computation and acts simply as an interface between the AI and the nodes.

## 6.4 AI Controller

The 'RacingAIController' is the class that is responsible for the computation of the AI, and for converting the AI decisions into inputs that can be accepted by the vehicle it is controlling.

### 6.4.1 Calculating Resultant Forces

In sections 6.2 and 6.3 I talked about the AI Target Node and Trackers that are used by the AI to traverse the track. The manner in which it does this is by having the next $n$ nodes on the

track attract the AI vehicle. The number of nodes that would act on the AI at one time will affect how the AI behaves by how effectively it cuts corners and how strictly it keeps to the centre of the track. The ID of the next node on the track from the AI will be stored as a variable in the AI controller, and is incremented when the AI gets within $x$ distance units of that node. The distance $x$ can also be adjusted to change the AIs behaviour, too small and it may drive past nodes and need to turn around, too large and it may discount a node too early and completely cut a corner, crashing into the barriers. The value I chose for this is 800 units, this way the AI cannot skip a node as the range covers the entire track, barrier to barrier. 800 Units is the smallest value I could use that would cover the entire track and still allow the AI to move onto the next node as late as possible.
If we consider:

- $A =$ The AI actor
- $N =$ The set of all AITargetNodes
- $j =$ the next node of the AI
- $n =$ the number of Nodes to look ahead
- $P(x) =$ The position of $x$ in the world
- $\Delta D(X, Y) =$ The normalised vector between $X$ and $Y$

Then the equation at each time-step to calculate the forces on the AI from the nodes is:

$$Force = \sum_{i=j}^{j+n} \frac{1}{2^{i-j}} \Delta D(P(A), P(N_i))$$

By having the AI controller calculate this force every 'Event Tick', which is every frame in UE4, then use the calculated force to control the car, the AI will be constantly updating it's controls as it moves around the track. This method of calculating the resultant forces at the AI vehicle's position in real time allows me to extend the implementation later to add other sources of forces, such as other vehicles in the head to head mode, to allow it to react to variable obstacles.
As a small addition to this system, the Nodes each have a variable that represents the speed that the AI should be trying to be at when passing the node, this speed is higher on straight sections of track and is slower on sharper corners. The AI will get this speed by simply looking ahead at the next node on the track and using it's target speed.
A small note on performance: Originally this functionality would be called in the event graph by the 'Event Tick' node, which is called every frame; however because the AI computation is relatively complex and does not need to be done as frequently as the frames are drawn, I moved the functionality to a separate event 'AITick'. This event is bound to a timer and is executed every 0.05s, meaning it is run 20 times a second. Because of the simplicity of the project, even with 9 AIs present at once, having them control their vehicle every 'Event Tick' did not affect the performance enough to make the framerate drop below the cap of 120 FPS, so the performance benefit of moving to the 'AITick' method cannot be seen, however in theory

it is still an improvement as each AI calls the functions to control the cars 1/6 as often with no noticeable change to the behaviour. The performance benefit would also be much more apparent in a more complex AI approach, such as the use of a neural network.

### 6.4.2 Controlling the vehicle

Once the AI has calculated the resultant force of the AI nodes acting upon it and the speed it should be currently aiming for, with the 'GetForces' function, it needs to translate those values into a set of inputs that the vehicle can accept. These inputs are a throttle value, a steering value and a handbrake value.

#### Throttle

For the throttle, the AI can simply compare its own speed with the target speed, setting the throttle >0 if it is going slower, and <0 if it needs to slow down. The throttle value can be a float between -1 and 1, so instead of having the values be either -1 or 1, which would cause very sharp accelerating and braking, the car should translate the desired speed into a value within that range. To do this I used the equation:

$$Throttle = clamp(-1, 1, 0.4 + \frac{T_v - A_v}{5})))$$

Where:

- $T_v$ = the target velocity of the AI
- $A_v$ = the current velocity of the AI
- $clamp(x, y, z)$ = returns x if $z < x$, y if $z > y$ and $z$ otherwise

The addition of 0.4 was done as this is roughly the throttle value that will maintain the vehicle speed while moving, so having the value centred around this meant that it is possible for the AI to simply throttle less if it only needs to slow down slightly, as oppose to always braking to slow down. The clamp function ensure that the value is between -1 and 1. This conversion to the range (-1,1) and passing to the vehicle by the function 'Set Throttle Input' is done in the AIRacerController function: 'Control Car'.

#### Steering

Converting the resultant force vector into a steering value between -1 and 1 required knowing the angle between the heading of the vehicle, and the heading of the resultant force vector. To calculated this I used a macro, described in Section 6.,5 in the function 'Set Dir'.
This angle difference, $\theta$, is then translated to the steering value by:

$$Steering = clamp(-1, 1, \frac{\theta}{25})))$$

By using this equation, any change in angle $<$ 25 degrees will be in the range -1 to 1, and any angle $>$ 25 degrees will require the vehicle steering wheel to be maxed out in one direction.
This conversion to the range (-1,1) and passing to the Vehicle function 'Set Steering Input' is done in the AIRacerController function: 'ControlCar'.

#### Resetting the Vehicle

The final consideration for controlling the vehicle is in the case of the vehicle getting stuck. In section 4.1.1 I mentioned the implementation of a reset car functionality to the vehicle blueprint that allowed the controller to reset the vehicle to the last checkpoint it passed. For the AI Time Trail, the likelihood for the AI to need to reset their vehicle is low because it is unlikely the AI will go off the track and get stuck, or get flipped over. However because I knew the project would eventually be extended to include head-to-head, where a collision could throw the AI vehicle off track or cause it to flip, I knew that the functionality would be needed eventually either way.
I had this portion of the control extend the 'AITick' functionality that the AI controller was using to send inputs to the vehicle. This way every time it sends an input it will check to see if it is stuck.
To check if the vehicle is stuck I defined a function, 'Check Stuck' that gets the up vector of the AI's vehicle, $V$ and compares the $z$ component of the vector with 0. If $V_z < 0$ then the vehicle is upside down. I also had the function check if the speed of the vehicle is $< 50$.
To determine if the AI Vehicle is stuck it calls 'Check Stuck' once, if true is returned it waits 1 second and calls 'Check Stuck' a second time, if the second call also returns true then the vehicle is considered stuck and the AI calls the 'Reset Vehicle' function of the vehicle it is controlling. By doing this the AI will only reset the vehicle if it is stuck upside down or stopped and is unable to recover for a full second.
To allow the AI to be reset properly I also had to add a variable to the AI controller: 'ResetNode'. This variable stores the ID of the AITargetNode that the AI should consider the next node on the track if it were to be reset. The value of 'ResetNode' is set by a function of the AIRacingController blueprint called 'UpdateResetNode' which is called by the Tracker when a vehicle controlled by an AI crosses a checkpoint. This function simply sets 'ResetNode's value to the value of 'CurrNode'.

### 6.4.3 Finishing the Race

When the AI Racer has finished the race, the Tracker will call the 'Finished Race' function of the Vehicle Blueprint, which in turn will call the 'Finished Race' function of the 'AIRacingController' if the Vehicle in question is controlled by an AI.
When this function is called the AI Controller will set the throttle and steering input to 0 and will stop the 'AITick' event from running, thereby disabling the AI.

## 6.5 Vector Macros

The 'VectorMacros' macro library contains a single macro: AngleBetween2DVectors. This macro simply calculates the positive angle between two 2D vectors using trigonometry.
The equation of this macro is:

$$A = Atan(\frac{V_1^y}{V_1^x}) - Atan(\frac{V_2^y}{V_2^x})$$

$$AngleBetween = ((A + 540)\%360) + 180$$

Where:

- $V_1$ = vector 1
- $V_2$= vector 2
- $A$ = a temporary value for storing angle

The purpose of the second equation is to ensure that if the value of $A$ is outside of the range $-180 \leq A \leq 180$ then it will be assigned to it's equivalent angle within the range. NegMod could not be used here as it is not in the scope of the VectorMacros library.

# Chapter 7

# The Head-to-Head

## 7.1 Changes to the Tracker

As mentioned in Section 5.2, I designed the Tracker with the knowledge that I would be extending it to include Head-to-Head mode. Because of this a lot of the changes needed to implement said head to head mode were simply to find where I had checked the 'TrackerMode' value and implement the case for if 'TrackerMode' = 'Head-to-Head'. Some examples of times when the Time Trail mode was implemented with Head-to-Head in mind are The Activate Check and Check Race Finish functions, both work as if there are multiple cars on the track, in the case of the latter it checks that all racers have finished the race, which in time trail is unneeded as there will only be one racer at a time.

### 7.1.1 Racer Positions

The changes that needed to be made to allow for a head-to-head mode include: Periodically calculate and assign race positions to the racers, and find the winner of the race once it is over. For this I implemented the function 'AssignPositions' to the Tracker to calculate the position of each racer in the race. It does so in one of two ways; when called the function is passed a boolean argument, 'Final', that represents whether the function should calculate the mid race positions or the final positions of the racers.
If 'Final' is True, then the function simply takes the value of 'Total Time' from each of the 'RacerData' instances, and assigns the positions in order of time, lowest to highest.
If 'Final' is passed as False, the function must calculate how far each car is around the track to know the order of the cars. To do this it calculates a 'Distance' value for each racer that represents the progress in the race each vehicle has made.
If for each racer, $r$, we consider:

- $l$ = the number of laps the racer has completed
- $c$ = the ID of the next checkpoint that the racer needs to activate
- $c'$ = the ID of the last checkpoint the racer activated
- $i$ = the RacerID for the racer
- $C$ = the set of all checkpoints on the track
- $V$ = the set of all vehicles on the track
- $Dist(X, Y)$ = The horizontal distance from X to Y

The formula for 'Distance' for that racer is as follows:

$$Distance = l|C| + c' + Min\left(\frac{Dist(C_{c'}, V_i)}{Dist(C_{c'}, C_c)}, 0.99\right)$$

In essence this equation calculates the value such that: The integer portion represents how many checkpoints the Racer has activated, and the decimal value represents the proportion of the distance that the Racer has covered between the last checkpoint and the next checkpoint it needs to activate. The use of $Min()$ is to ensure that the proportion of the distances can never be $\geq 1$, which would would happen in the case of a racer passing a checkpoint without activating it.
The function then uses these 'distance' values to assign a position to each of the racers, in order of highest to lowest.

## 7.2 HUD Additions

In order for the HUD to work properly in a Head-to-Head mode, an extra 'border' needs to be added that acts as a Head-to-head end screen to display the winner of the race.
The border 'H2HEndScreen' was added and the text that displays the winner is bound to the text variable 'H2HWinner' which is a part of the HUD class. The ESlateVisibility variable and the function to display the 'H2HEndScreen' border were also added. The function takes an integer parameter that represents the winner of the race and is used to fill the 'H2HWinner' variable before it is displayed. This way when the tracker calls the function to display the end screen it simply passes the ID of the winning racer to be displayed.

## 7.3 AI Additions

The Final change to the game to consider the Head-to-Head mode finished is to adjust the AI to work properly and be fun to race against.

### 7.3.1 Reacting to other Vehicles

The first change is to have the AI react to other vehicles on the track, meaning it drives in a more realistic manner. To do this I extended the 'Get Forces' function of the 'RacingAIController', by having it also consider every other vehicle on the track.
The AI will loop through each of the entries in the Tracker's vehicle array, and providing the entry is not the vehicle of the AI in question and the vehicle is within 200 units of the AI's vehicle, it will have the other vehicle exert a repulsive force on the AI, the magnitude of which follows the formula:

$$Magnitude = \frac{4000}{d^2}$$

Where $d$ is the distance between the AI's Vehicle and the other vehicle.
This way the closer a vehicle is, the more it will repel the AI. With these forces, in addition to the AITargetNode forces, the AI will attempt to drive away from another vehicle it is close too, but will still continue to drive along the track.

### 7.3.2 Difficulty

For the AI to be fun to race against for a range of skilled players, there must be some form of variable difficulty in place, that allows the AI to change how well it races. Instead of having a set difficulty be chosen before the race I decided to have the AI's difficulty change dynamically throughout the race, this way the race will be competitive no matter how good the player is. The difficulty of each instance of the RacingAIController is stored in the 'Difficulty' variable as a float between 0 and 3. With 0 being the lowest difficulty and 3 being the highest.

### Effect of Difficulty

A large part of my research was finding a method of difficulty that did not take away from the enjoyment of the game. One easy way to change the difficulty would be to give the racers who are behind an unfair advantage, such as better acceleration or handling. However, as discussed in Section 3.3.3, this method of rubber-banding can make the game frustrating to play, and can result in only the final 25% of the race actually dictating the outcome. Because of this I knew I wanted to have the difficulty of the AI affect something other than the physical properties of the car it is controlling.
With this in mind I decided that the difficulty of the AI should affect two aspects of the AIs functionality: how many nodes in front of the AI will exert a force on it, and how closely it will follow the target speed of the next node in the track.
To adjust the amount of nodes considered, I adjusted the for loop in the function 'GetForces' to instead truncate the difficulty value, and loop from 0 to that truncated value. This way if the AI difficulty is at 0, only one node is considered, and if the difficulty is at 3, the next 4 nodes are considered.
To implement the changing of target speed by difficulty I adjusted the 'Set Vel' function. This function originally was added as a placeholder and simply took in a velocity argument and returned it with no change. In the new version the return value follows the formula:

$$OutVel = 0.7 * (0.1 * d)$$

Where $d$ is the difficulty value.
This way the returned velocity will be the input velocity scaled from 0.7 at the lowest difficulty, to 1 at the highest difficulty.
With these changes in place, if the AI is racing at difficulty 3, then it will perform exactly how it did before the difficulty was introduced, however any value below that will both drive slower and will take corners less sharply.

### Changing the Difficulty

As mentioned I decided to have the AIs difficulty change over time depending on how well it is performing. This way the AIs difficulty will be suitable for a range of skill level players.
To have the difficulty be adjusted during the race I set up an event in the RacingAIController Event graph called 'UpdateDifficultyTimerCallback' which is bound to a timer and is executed every second. Having the AI update every second was frequent enough that it will react quickly but will not impact performance.
First the event adjusts difficulty differently for two scenarios: Against a player, against other AIs.
**Against a player**

If against a player the event will update the difficulty according to the distance between itself and the player. By distance here I am referring to the 'Distance' value that is stored as a part of the 'RacerData' struct. If $d$ is the distance from AI to Player vehicle, and $clamp()$ follows the definition given in Section 6.4.2, then the change in difficulty every second follows the formula:

$$\Delta Difficulty = clamp(-1.5, 1.5, \frac{d}{3})$$

This way the difficulty can change by a maximum of 1.5 per second and changes proportionally to how many checkpoints are between the AI and the player.
The difficulty itself is still clamped between 0 and 3.
**Against other AIs**
In a head-to-head race comprised of more than one AI and no players, the difficulty for each AI is still adjusted every second, however the method needs to change as there is no player vehicle for the AI to compare it's distance to.
Instead I have the AIs each compute the average distance value of every other AI vehicle on the track, then compare it's own distance to this average value. The adjustment to the difficulty remains the same but uses this new distance value. This way any AI that is below average will increase in difficulty and any AI that is near the front of the race will reduce in difficulty. This will make the race more fun to watch as if multiple cars crash, they will have a chance to catch up.

# Chapter 8

# Testing and Evaluation

## 8.1 Comparison with Objectives

When I defined the objectives in Section 1.2 I split the project into 4 discrete sections: The Racing Game, The Race Track, The AI Implementation, and performance.

### 8.1.1 The Racing Game

**The game should be created using a pre-built engine such as Unreal or Unity**

The Game was built entirely using the Unreal Engine 4.24; This objective was complete.

**The game should be playable using a keyboard/mouse or joypad**

The Final version of the game is playable with keyboard/mouse or joypad and features intuitive controls.
Keyboard/mouse Controls:

- W - Accelerate vehicle
- S - Brake/reverse vehicle
- A/D - Turn vehicle
- Space Bar - Handbrake
- Enter - Accept (In start/end screens)
- Backspace - Reset vehicle
- C - Reverse cam
- Tab - Toggle in car camera
- Left/Right - Skycam prev/next target
- Mouse movement - Look around (In-car view or skycam free-look mode)
- Mouse Scroll Wheel - Adjust skycam zoom

Joypad Controls:

- Right Trigger - Accelerate vehicle
- Left Trigger - Brake/reverse vehicle
- Left Analog - Turn vehicle
- A Button - Handbrake

- Start - Accept (in start/end screens)
- Y Button - Reset vehicle
- Left Analog In - Reverse cam
- Select - Toggle In car camera
- D-Pad Left/Right - Skycam prev/next target
- Right Analog - Look around (In-car view or skycam free-look mode)
- D-Pad Up/Down - Adjust skycam zoom

In order to test that this objective is completed properly I simply loaded up the game and tested that for both input methods, each of the controls are assigned to the correct buttons and function as expected. This testing was done throughout the development process, whenever a new key binding was set, it would be tested.

### The game should feature driving mechanics that are realistic but also fun to drive

The first portion of the objective is easy to evaluate, as I have used the basic driving mechanics supplied by Unreal, there is no need for testing. However the second half of the objective stipulating that the driving should be realistic but fun is entirely subjective. The vehicles behave like vehicles in real life, simulating friction, torque and gravity accurately, so it could be easily argued that they behave in a realistic manner. As for the matter of if the vehicles are fun to drive, I made design decisions, such as increasing the mass, in order to ensure that the vehicles are easy enough to control but still maintain the need for pre planning in braking before corners that adds an element of difficulty and fun to the experience. Therefore I would consider the driving mechanics to be fun and the objective to be complete.
As for testing, because of the subjective nature there is no real formal testing that can be done to validate the claim, instead the changes to the mechanics where made incrementally along with testing until I got to a point where the driving behaved in a manner that I was happy with.

### The game should have two game modes Time Trail and Head-to-Head

The Final game does support both time trail and head to head play. These game modes are split into the four different levels: 'TT_Track', 'AITT_Track', 'H2H_Track', and 'AIH2H_Track'. Where the levels with the prefix 'AI' has the player control the skycam while spectating an AI only race.
To be able to justify the the objective as complete, each of the two game modes should fill a check list of features. For the Time Trail mode I would consider the necessary features to be: A single car controlled by the player, clearly indicate when the trail is started, trail is timed correctly, the trail tracks laps done, the trail only counts a lap if the entire lap was complete, the trail finishes when the specified number of laps is complete, and the final time is displayed after the trail.
I ran the program and selected the single player time trail mode in order to test each of these features, ticking them off if they are done correctly. After my testing I found that all of these features were working correctly and so the time trail portion of the objective is complete.

The features I would deem necessary for the Head-to-Head mode are: Multiple racers on the track, clear indication of race start, collision physics between racers, race is times, race tracks laps of each racer, race finished once all racers have finished the set amount of laps, and the winner of the race is displayed after the race is finished.
Testing these features was again as simple as running the program and choosing the player head-to-head mode, then checking for each feature as they come up and ensure they work as intended. The game passed this test as well, hence I would consider this objective complete fully.

**The game should allow the player to compete against one or more AI in Head-to-Head**

This objective is very simple to test, I simply loaded the Player Head-to-Head mode and ensured that the other vehicles in the race are controlled by an AI that attempts to complete the race. The game passed this test and so the objective is complete.

**The game should allow the player to spectate AI Racers competing in both Time Trails and Head-to-Head races**

In order to have consider this objective complete I would expect the player to have the ability to view the race from a third person perspective with control over the camera.
In the game implementation when spectating a time trail or race the player controls a sky cam, that has two modes: auto-lock and free cam. In auto lock the camera will follow a specific vehicle; the vehicle being followed can be controlled by the player using the controls listed in 8.1.2. The free cam mode allows the user to move the camera with the mouse or joystick. In both modes the camera can be zoomed in and out by the player.
To test these features I simply loaded the two modes: AITT_Track and AIH2H_Track, and checked that each feature was working as intended. I made sure to check that the zooming of the camera worked properly in both free cam and auto-lock mode. Testing found that the features worked perfectly as intended and thus the objective is complete.

**The game should keep track of the times of racers, displaying them when the race is finished**

During the race the time can be seen on the HUD by the player, and in both the time trail and head-to-head modes the final time is displayed after the race is finished (the final time of the winner in the case of head-to-head).

### 8.1.2 The Race Track

Some of the objectives in this section are very difficult to evaluate because of their subjective nature. However because of the expandability measures I have added into the game, which allows the development of further levels, I feel that designing the track to my personal tastes is good enough to consider the more subjective track objectives done. As well as this, with the project being exploratory software, the final game should only be a prototype; if it were that I was developing a full, release ready game, I would have to design many more tracks, and have them tested by other people to ensure that they are fun to play for a range of people's tastes.

#### The Race track should be built using the level editor of the engine chosen for the project

The race track was built entirely using the Unreal Engine 4 Level Editor and is made of assets supplied with the engine.

#### The Race track should be interesting enough that it is fun to drive on

When designing the track I made sure to follow a few criteria in order to make the track interesting to drive on. Some examples of features on the map that make it fun are: Having only 2 long stretches of straight mean that the majority of the lap is either: slowing down for a corner, turning a corner, or accelerating out of the corner. This means that the time that is spent just driving in a straight line, which is not much fun, is minimised. I also ensure that a majority of the corners are wide turns, so that they can still be traversed at relatively high speed. However some variation was added in the form of the final corner, being very sharp, this means that unlike the other corners the drivers need to slow down considerably to turn it properly. This final corner however is preceded and followed by long straight sections so that the drivers have time to prepare as if it was immediately after another set of corners it could prove frustrating.
This set of features to me makes the track interesting to drive and so I would consider the objective complete.

#### The Race track should not be too complex that it is too difficult to drive

As mentioned in the section above I designed the track in a way that a majority of the corners are wide turns, this way they can be done at speed, however this has the side affect that the track is not extremely difficult to race. That being said, the track cannot be raced while simply holding accelerate the entire time, so I would consider it hard enough to be interesting, just not too hard it is un-fun. The exception is the final corner, which as I mentioned above is preceded and followed by straights, because of this the corner's difficulty is reduced and so I would consider the track as a whole to be simple enough that the it is not too difficult for any skill level to drive: objective complete.

#### The Race track should be wide enough that multiple cars can be side by side on it

The track is made using the parts supplied in the Advanced Vehicle demo assets of the Unreal Engine. These parts are approximately 500 units wide, however I decided to scale them up by 2, making them 1000 units wide. As a result the track is 1000 units wide at all points.
The model used for the vehicles in the game are also part of the assets provided in the 'Advanced Vehicle' demo, this model is approximately 100 units wide. As a result the width of the track can fit up to 9 vehicles side by side and as a result the objectives is fulfilled.

#### The Race track should be designed in a way that it is clear to the player where they should be going

because of the use of the Unreal assets, the track surface is a grey colour and the edges of the track pieces are lines with a red and white striped border; As a result is it very clear where the track boundary is. The track also features no forks in the road. Because of these two features I believe it is very clear to the user exactly where they should be going to complete a lap and so the objective is complete.

### 8.1.3 The AI Implementation

#### The AI should be able to control its vehicle

As mentioned a few times the AVB, which is the class responsible for the player and AI vehicles in the race, is controlled by a set of functions that change the throttle, steering and handbrake of the vehicle. The AVB itself has is designed so that any controller class associated with it can call these functions and control the car fully. Because of this the RacingAIController, and in turn the AI itself, is able to control the vehicle it is responsible for fully, to the same degree that a player is. Hence the objective is complete.

#### The AI should be able to navigate the track

As was shown in the demo of the program, during testing the AI is able to consistently complete the track, regardless of the amount of vehicles on the track. Over all of the testing for the game that has been done, whether that be specifically testing the feature of completing the track, or testing other features, the AI has been able to complete the track as intended every time, regardless of other racers and such that are present. Because of this I feel that the objective is complete fully.

#### The AI should be able to avoid collisions with other vehicles

Finding a balance between the AI avoiding collisions, and the AI being bullied off the track by another vehicle is difficult and required a lot of fine tuning. With the system currently in place, the AI will not steer into another vehicle if they are in the path of the AI. However if the AI is following another vehicle at speed and the vehicle in front slows suddenly there will still likely be a collision, as well as this if another vehicle deliberately steers into the AI, it will attempt to steer away to a degree but collisions are still very possible.
This balance is intended to make the AI still behave realistically, as if a real person is controlling the vehicle, while still maintaining a good level of difficulty. Hence I feel that the objective has been completed fully.

#### The AI should drive in a realistic looking way

The underlying mechanism behind the AI was designed with this objective in mind, I wanted the AI to fundamentally act in a realistic looking way which is why I used the potential methodology. Because this method of computing a direction to go is constantly adjusting and can take into consideration variables in real time, it can react in a human like way to obstacles, where if the AI simply drove on a set path it would not do this. I feel that this adds a great deal of realism to the behaviour. In combination with the variable difficulty, as long as there is another vehicle on the track the AI will always take a slightly different path around the track as any small changes will then cause the entire rest of the lap to be different. The collision avoidance also adds to making the AI seem human-operated by reacting to close calls appropriately. With all of these measures in place I feel that the objective is complete sufficiently.

#### The AI should be able to operate at a range of skills

The final objective of the AI is to have it be able to race at a range of skill levels so that it can have a competitive race against a number of skill level players. This is done through the 'difficulty' rating of each instance of the AI, which affects how many 'AITargetNodes' ahead it looks and how closely it follows the 'TargetSpeed' of each node. In the game the 'difficulty' rating of the AI is variable throughout the runtime and changes depending on how well the AI is doing, this means that over multiple runs of the game, even against the same player, the difficulty will not be the same as there is variation in the difficulty. As an example if the player does slightly better at the start of the race, then the AI will be slightly harder as a result. This means that the AIs difficulty, and therefore behaviour is relatively unpredictable making it more fun to race against, regardless of player skill.
As well at this the difficulty value is not only a set of difficulty "levels", it is instead a range from 0-3, which means that in theory there are infinite difficulties so it can fit a greater range of player skills. At the low end of the scale the AI will drive at around 70% of the speed it should, and will follow the centre of the track. Where the most difficult of the AI will drive along an efficient racing line that hugs the close edges of corners, and will drive at an optimal speed for turning the corners as quickly as possible. Because of these ranges in difficult I feel that the objective is complete sufficiently as it allows a range of players to race.

### 8.1.4 The Performance

#### The game should run at a framerate that is consistently higher than 90fps at all times.

In order to test the performance of the game I ran the game while enabling the 'stat FPS' and 'stat UnitGraph' commands. These commands show the fps, and a time graph of the frame, game draw, and GPU times. By having these commands enabled while running the game I can see how long each frame will take to render and the framerate of the game.
Because I knew that the performance would be worst in the AI Head-to-Head race, I ran this first. My findings where that the FPS was consistently higher than 400 fps, with no noticeable drops below that thresholds. The Frame times where between 2-3 milliseconds which, in the worst case of 3 ms, is still more than 300 fps. These results show that the performance of the game is very acceptable and can easily beat 90 FPS. Therefore the objective is complete.

### 8.1.5 Expandability

This section of objectives focus on how easy developing custom content for the game is from the perspective of someone other than the developer.

#### The level architecture needs to be simple enough that custom levels can be designed with relative ease

When I was designing the architecture for my game I knew that I wanted to do it in a way that it would be able to run multiple levels, this meant that the level should not be responsible for any of the game features besides the race track instead.
To this end I decided the level should instead just feature the track itself, and then hold all of the components necessary for the race to work properly. In my game the 'Tracker' blueprint, with a set of 'Checkpoints', is responsible for organisation and control of the race itself, and the AVB with either a 'RacingPlayerController' or a 'RacingAIController' associated with it is responsible for all of the vehicles and the control of said vehicles. There is also the need for a set of 'AITargetNode's and an 'AITargetTracker' in order to have the AI controlled vehicles be able to traverse the track.
With this game architecture in order to create a new level that will work for the game, a developer simply needs to use the Unreal Engine level editor to create the race track by whatever means they choose, then add in a 'Tracker', configured properly through the editor available variables, including the tag 'master'. The developer will also need to add a set of checkpoints appropriately spaced around the track; after which the references to each of the checkpoints need to be added to the tracker in the correct order. The custom level should also include one or more AVBs that are configured for either Player or AI control. If AI control is present in one or more of the vehicles, the developer will also need to add a set of 'AITargetNode's that are appropriately spaced around the track, with the correct 'TargetSpeed' values set for each of them. As well as an 'AITargetTracker' that has a reference to all of them. The 'AITargetTracker' instance will also need the tag 'master'. If all of these steps are followed the game will run properly on a custom made track.
This list of requirements to create a custom track does not require any programming and is simple enough that it can be done by somebody with no underlying knowledge of how the game works. Therefore I would consider the objective complete.

#### The information given to the default AI should be a part of the level so that custom made levels can be designed to allow the AI to race on them

The information that the AI uses to traverse the track is the position and 'DesiredSpeed' values of each of the 'AITargetNode's on the track. So, provided a track has these nodes set up correctly, with an accompanying 'AITargetTracker', the AI will be able to traverse the track properly. Therefore the objective is complete.

**The game should have the ability for a custom AI system to be usable within the game, that can still interface with the vehicles and compete in the races as intended**

The implementation of a custom AI controller into the game is a little more complex than creating a custom level. This is because in order to implement a custom AI, the developer will need to create a custom 'AIController' blueprint, which does require programming. As well as this, unless the custom AI uses the 'AITargetNode's that the current AI system uses, there will need to be another system put in place to give the custom AI information about the track, this could be in the form of a custom blueprint that is placed in the level, or use of the image from the vehicle cameras and computer vision techniques.
The up side however is that when this custom AIController has been implemented, as long as it interfaces with the vehicle it controls using the same set of functions that the 'RacingAIController' does, it should work within the program properly. This is because the Tracker treats both player, and AI controlled vehicles the same. This abstraction allows any AVB to participate in the race full, regardless of how it is being controlled. Hence I would consider this objective done.

## 8.2 Evaluation of project management

Good project management is vital in order to ensure the project is complete to a high standard and in a timely manner. I elected to use an iterative waterfall approach in this project which would allow me to iteratively implement new features into the project without the need for months of analysis and design at once. This helped me get working prototypes ready much faster and meant that the progress of the project was much more clear.
In the intermediary report I created a Gantt chart, that allocated my time, by week, to different portions of this project. Though there were complications in timing, and one week had to be re-assigned to recreating the Time Trail mode as mentioned in section 5, overall I would say that I managed to keep to these timing effectively, and as a result I am very happy with the game that was created throughout.
My use of source control was lacking, as I had to completely recreate the project file in Unreal, which meant the original Git repository could not be used. This meant that there was a month long gap in my use of source control where instead I was using physical ZIP archive backups at major junctions. After this I created a second repository and continued to use source control properly however.

# Chapter 9

# Self Appraisal

## 9.1 Challenges

Throughout the project there were a few major challenges that had to be overcome in order to produce a good final product in a timely manner. One of the most important processes in project management is the methodology used to overcome unexpected challenges and adapt the plan to best fit the change in circumstances.

### 9.1.1 The need for a refactor

As I have mentioned previously, early on in the development process I followed an Unreal Engine tutorial for building a Time Trail game using the AVB as it was an opportunity to learn how to develop with the engine and my initial thinking was that the end result would be expandable in order to make the rest of the game. However after I had finished following the tutorial I found that the program flow was designed in a way that extending the program would be very difficult, and would result in very messy source code. This was because both the 'Tracker' and the 'PlayerController' were equally responsible for the computation of the racer completing the track, which meant that having multiple vehicles would need to have multiple 'Trackers' and another object responsible for organizing those trackers. This method, which would have worked, would mean that the program flow was very spread across many objects and would be very hard to follow, making development of further features much harder.
The other solution to the problem would be to rebuild the game from the ground up, which is what I decided to do. I decided on this approach because I had already learned how to use the Engine now and I felt much more confident that I could design the new version in a way that would be much clearer and easier to extend in the future. Before beginning the implementation of the new version I decided to design the program flow on paper, still using the method of checkpoints and trackers, but doing it in a way that abstracted the control of the vehicles away from the tracker, and in turn the organization of the race away from the vehicle controllers. This new method of abstraction allowed me to treat all vehicles, player or AI controlled, the same which made the implementation of the AI down the road much simpler. In order to actually implement the new version of the game I did not have to create everything from scratch. Some parts of the old game would be usable as they are and were just migrated over to the new project such as: the track, the checkpoint blueprint, and the macros. As well as this parts of the 'tracker' class could be copied over as well. The ability to reuse lots of the old system's parts meant that the time to recreate the functionality of the first version with a better program structure only took around a week. To me this decision was most definitely the best as just extending the old system to add a second vehicle would have taken many weeks and would have been much more difficult.

### 9.1.2 The Availability of VR

The initial plan for the game was to be a VR enabled racing game, in which the player could play the game from an in car perspective using the HTC Vive VR headset. However with the outbreak of COVID-19 at the beginning of the year (2020), the University had to shut down fully, meaning that access to the campus, and in turn the VR labs, was cut off completely. Because I did not have access to a VR headset at home, and so had no way to test the VR functionality, I had to remove the VR features from the project plan.
Because the closing of the university was only announced mid way through March of 2020 the bulk of development for the game had been completed, with the VR functionality still in mind. The Unreal Engine supplies most of the VR capabilities as a part of the AVB by default, so in order to have the game be playable in VR I simply had to ensure that the changes I made to the AVB, and the features I implemented to the game did not affect the VR capabilities. Although I have no way to test this I am almost certain that during the development I did not make any changes that would break the VR capabilities. As a result I can confidently say that right now, the resulting game from my project is playable as a VR game, with the exception of the extra key bindings that I added needing buttons assigned on the Vive controllers, which could be added in a matter of seconds. However I have had to lessen the scope of the project to exclude this feature as I have no way of testing that it actually works in its current form.
A further feature was planned to add local multiplayer support, where one player would control a vehicle using the VR headset, and a second player would control a second vehicle using the mouse and keyboard on the PC. This idea also had to be scrapped, and adjusting it to be online multiplayer would be outside of the scope of the project.
To make up for the lack of VR in the final product I attempted to add some other quality of life features in the short time I had between finding out that I could no longer do VR, and having to finish the game. Such features include: a main menu system that allows the user to choose how many laps are to be complete and the game mode to be played, extra geometry to the track in the form of a small wall that is either side of the track to make the track more interesting to look at and to direct the vehicles better.

## 9.2 Future Considerations

It is important in any large scale projects, especially in game development, that the product is designed with expandability in mind. This way further extensions to the functionality are much easier to develop and as a result, extending the lifespan of a game based on demands from the player base can result in a much more profitable product in the long run. Business models offering optional DLCs (Downloadable Content) that the user can buy to add more features to the game are extremely common in modern gaming. Because of this it is important to consider what additional features could be added in the future during the development of a game.

### 9.2.1 Multiple Cars/Customisable Cars

One of the most obvious extensions to the game that could be made if the development were to continue after the completion of the dissertation is the addition of multiple and/or customisable vehicles that the Player or AIs can control. The implementation of such vehicles

would be extremely easy as a second copy of the AVB could be made, that has different values for the mass/torque and handling.
The different cars could also be customisable through a level system, where the player gets experience points for completing races and can use these to give their vehicle an upgrade in one of the aforementioned areas.
At this point it would also be a good idea to create a model for the body of the vehicles in the race. Right now the vehicles meshes being used are the bare minimum required to accurately simulate a car. The are comprised of a chassis with suspension and wheels, and a seat for the player. The addition of a body to this would make the game more visually pleasing, as well as making the different cars differentiable by sight.
By having multiple vehicles, with multiple models, it would be necessary to add an extra screen to the menu before the race level is loaded, that allows the player to choose the vehicle that each racer is controlling.

### 9.2.2 Multiplayer

Another feature that could be added if the game development time was longer is that of multiplayer. Initially there were plans for the game to be designed with local multiplayer functionality as this would not require any kind of server architecture. But another option is to add online multiplayer, which wold require a server, or one of the game instances to act like a server, and two clients so that multiple players, on separate machines could race against each other. The UE4 Editor does have capabilities to easily convert games into simple multiplayer games so I don't feel that this would be an extremely hard feature to add. Additionally the design of the game, having the bulk of the computation done by a single instance of an object, is well fitted to a client-server model as the server could contain the instances for the tracker and checkpoints, and simply distribute the relevant information to the clients.

## 9.3 Ethical Considerations

As with any projects there are a number of ethical issues that have to be considered when designing a game. The game I have designed, in its current form, handles no user-data and so there is no ethical concerns in this respect that need to be addressed. There is no images of people used and no sensitive information stored or displayed during the course of the game.
PEGI[1] is an organisation dedicated to assigning age ratings to games that are released to the public, in order to allow people to get a good idea of how suitable a game is for their children. As can be seen on the website the rating "PEGI 3" is most suitable for my game is it contains no sounds or pictures that could frighten children, no violence and no bad language.
Because of the relative simplicity of my game, and the fact that it is exploratory software, there was little need for user testing in the development process. Usually user testing would be used to test how usable the user interface is, for finding bugs within the game, and for getting feedback on how to adjust the game to make it more enjoyable. However the goal of this project was not to deliver a game that could be released to the public, instead it was to show off the technology, and as a result the usability, and enjoyability of the game were less important. Because of this I decided to do no outside user testing and instead elected to test

the game myself, as this would allow me to remove the bugs that were detrimental to the system. So as a result there were no ethical concerns around the topic of testing.

# References

[1] Pegi website.

[2] Dynamic game difficulty balancing, Mar 2020.

[3] S. Arzt. Deep learning cars, Oct 2016.

[4] W. Bunn. Blueprint time attack racer | v4.8 | unreal engine.

[5] I. Dudkin. Unreal vs unity for vr development, Jan 2019.

[6] M. A. Goodrich. Potential fields tutorial.

[7] E. Jimenez. The pure advantage: Advanced racing game ai, Feb 2009.

[8] S. Rabin. *Game AI pro: collected wisdom of game AI professionals.* CRC Press., 2014.

[9] T. Tran and T. Tran. Unreal engine 4 tutorial for beginners: Getting started.

[10] S. Whittington. Unreal engine 4 tutorial for beginners - season 1.

# Appendices

# Appendix A

## Track Design Drawings

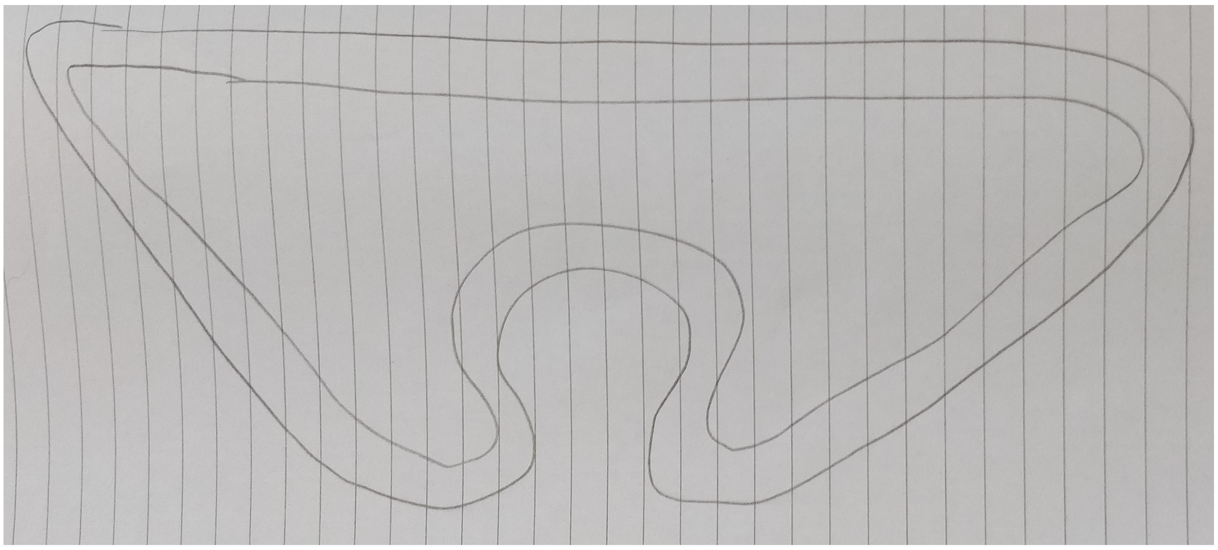
Figure A.1: Design Drawing 1

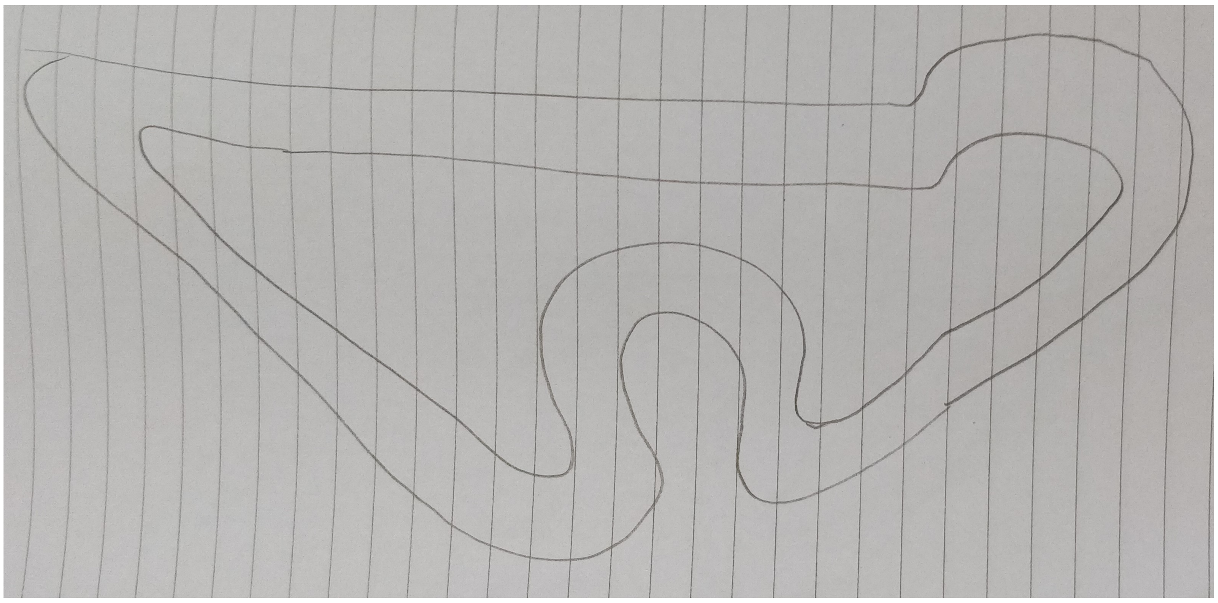
Figure A.2: Design Drawing 2

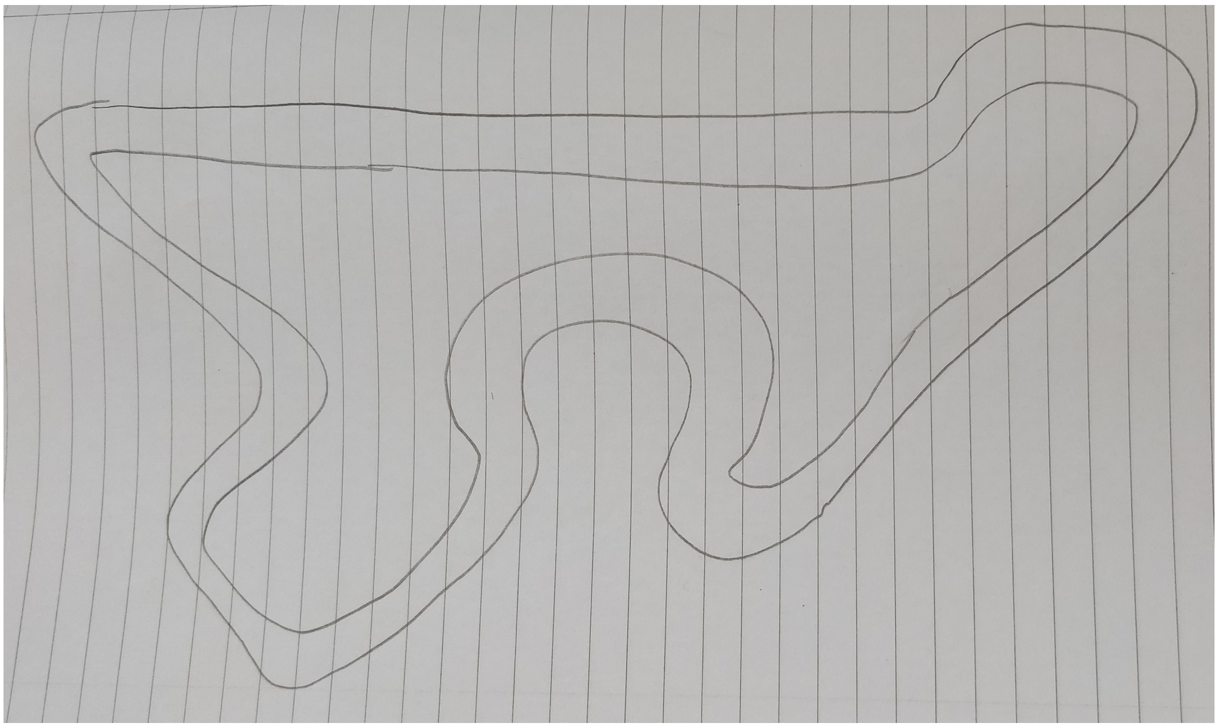

Figure A.3: Design Drawing 3

# Appendix B

## Track Implementation Images

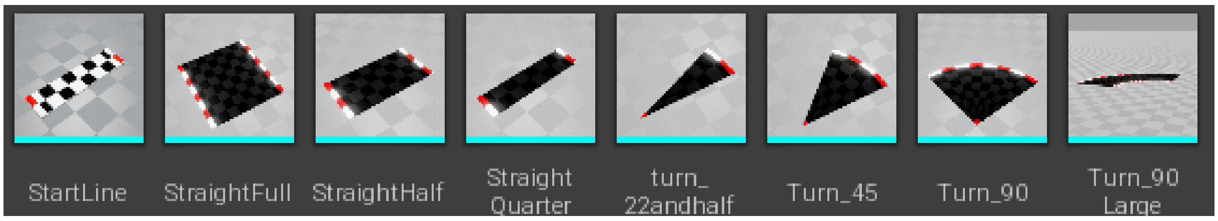

Figure B.1: Track Pieces

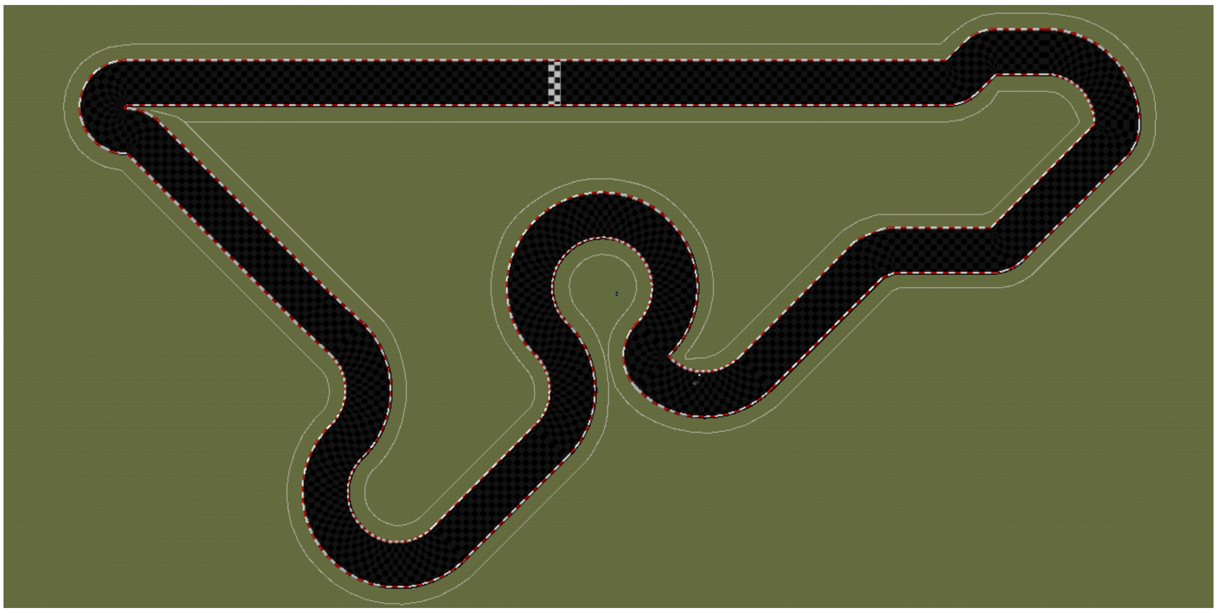

Figure B.2: Final Track Implementation

# Appendix C

## The Vehicle

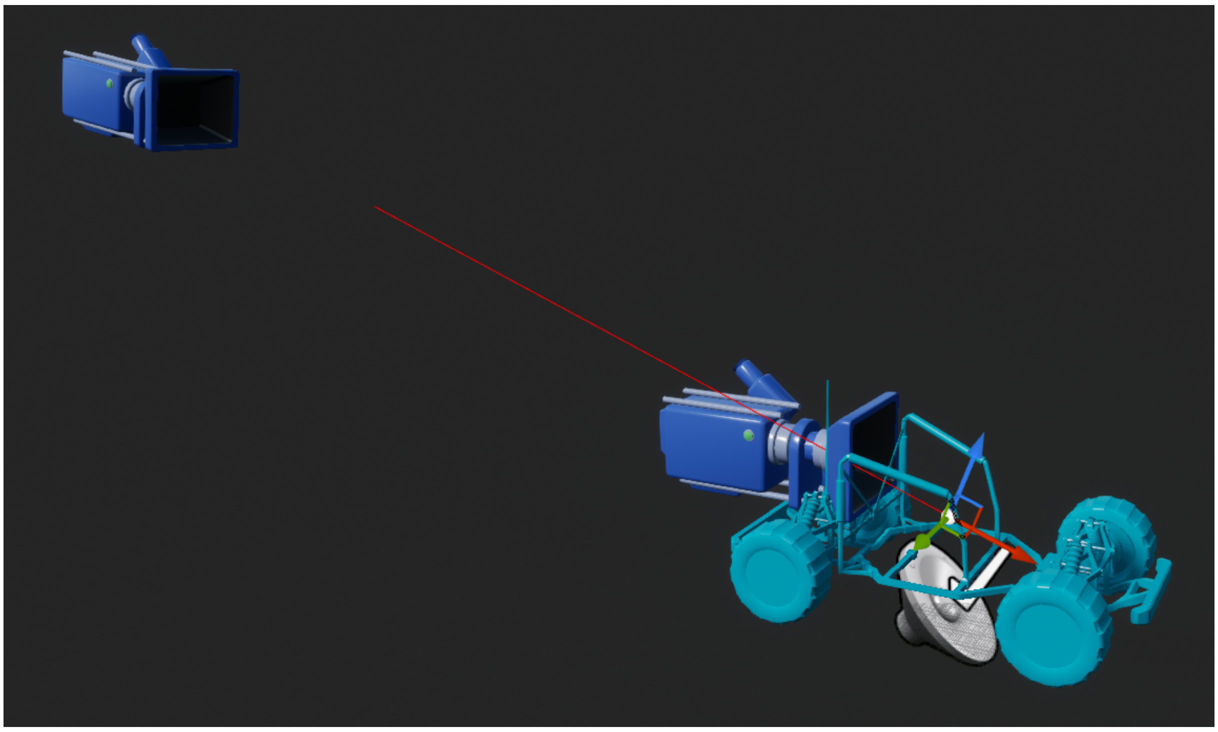

Figure C.1: The vehicle mesh